Micropolygon Ray Tracing With Defocus and Motion Blur

Qiming Hou* Hao Qin† Wenyao Li† Baining Guo*‡ Kun Zhou†

*Tsinghua University †State Key Lab of CAD&CG, Zhejiang University ‡Microsoft Research Asia

(a) Perfect focus (b) Motion blur + defocus

Figure 1: *A car rendered with defocus, motion blur, mirror reflection and ambient occlusion at* 1280×720 *resolution with* 23×23 *supersampling. The scene is tessellated into 48.9M micropolygons (i.e., 53.1 micropolygons per pixel). The blurred image is rendered in 4 minutes on an NVIDIA GTX 285 GPU. The image rendered in perfect focus takes 2 minutes and is provided to help the reader to assess the defocus and motion blur effects.*

Abstract

We present a micropolygon ray tracing algorithm that is capable of efficiently rendering high quality defocus and motion blur effects. A key component of our algorithm is a BVH (bounding volume hierarchy) based on 4D hyper-trapezoids that project into 3D OBBs (oriented bounding boxes) in spatial dimensions. This acceleration structure is able to provide tight bounding volumes for scene geometries, and is thus efficient in pruning intersection tests during ray traversal. More importantly, it can exploit the natural coherence on the time dimension in motion blurred scenes. The structure can be quickly constructed by utilizing the micropolygon grids generated during micropolygon tessellation. Ray tracing of defocused and motion blurred scenes is efficiently performed by traversing the structure. Both the BVH construction and ray traversal are easily implemented on GPUs and integrated into a GPU-based micropolygon renderer. In our experiments, our ray tracer performs up to an order of magnitude faster than the state-of-art rasterizers while consistently delivering an image quality equivalent to a maximum-quality rasterizer. We also demonstrate that the ray tracing algorithm can be extended to handle a variety of effects, such as secondary ray effects and transparency.

Keywords: GPUs, Reyes, rasterization, depth-of-field, motion blur, bounding volume hierarchy, hyper-trapezoid

ACM Reference Format
Hou, Q., Qin, H., Li, W., Guo, B., Zhou, K. 2010. Micropolygon Ray Tracing With Defocus and Motion Blur. *ACM Trans. Graph. 29,* 4, Article 64 (July 2010), 10 pages. DOI = 10.1145/1778765.1778801 http://doi.acm.org/10.1145/1778765.1778801.

1 Introduction

Current cinematic-quality rendering systems are based on the Reyes architecture [Cook et al. 1987], which uses *micropolygons* to represent high order surfaces and highly detailed objects. In a Reyes pipeline, all input geometric primitives are first tessellated into micropolygons (*i.e.*, quads less than one pixel in size). Shading is then calculated on micropolygon vertices. In the subsequent sampling stage, micropolygons are rasterized into fragments, which are filtered to produce the final image.

Rasterization has long been regarded as the best way to sample micropolygons and widely used in existing micropolygon renderers. When sampling stationary geometry (*i.e.,* without defocus and motion blur), rasterization is simple to implement and performs efficiently because tight bounding boxes can be easily computed for micropolygons. However, when rendering defocus and motion blur, tight bounding boxes can no longer be computed and brute-force rasterization becomes very inefficient. Non-intuitive parameters for quality and performance tradeoffs have to be exposed to end-users to achieve a satisfactory balance between artifacts and render time, as noted in Fatahalian et al. [2009].

In this paper, we tackle the problem of sampling defocused and motion blurred micropolygons using ray tracing. We propose to replace the rasterizer in the sampling stage of micropolygon renderers with a ray tracer. Ray tracing does not need to compute tight bounding boxes of micropolygons and is known to work efficiently for irregular visibility sampling. By designing an effective acceleration structure, we demonstrate that for high quality defocus and motion blur, the advantage of ray tracing in irregular sampling outweighs its inherent algorithmic overhead. Experiments show that our ray tracer is up to an order of magnitude faster than the state-of-the-art rasterizers. Moreover, with our algorithm, users do not need to make non-intuitive tradeoffs between the rendering performance and image quality.

Designing an efficient ray tracing algorithm for micropolygons with defocus and motion blur is far from straightforward. The key challenge is the acceleration structure. First, the structure has to be

flexible to support the 4D space-time extension to handle motion blur. It also has to be robust such that the traversal complexity will not degenerate for challenging geometries such as hair and furs represented by spatially-clustered long curves. Furthermore, the memory consumption of the structure has to be minimized to process millions of micropolygons in-core for parallelism utilization. Secondly, the structure has to be constructed very efficiently without significantly compromising traversal performance. Since the ray tracer only processes the sampling stage in the Reyes pipeline, only a fixed amount of primary rays generated by defocus and motion blur supersampling are traced. Therefore, the structure construction cost cannot be amortized over a large amount of rays and may constitute a significant portion of the rendering time. Finally, rays have to be kept coherent during traversal. While primary rays are inherently coherent, high quality defocus and motion blur require rays to be generated using stochastic sampling. Directly tracing rays in their generation order may result in very poor coherence. As a consequence, the rays have to be reorganized to exploit the inherent coherence.

The basis of our ray tracing algorithm is a novel acceleration structure – a BVH based on 4D hyper-trapezoids that project into 3D OBBs in spatial dimensions. This structure has several features making it very suitable for tracing defocused and motion blurred micropolygons. First, the hyper-trapezoids are able to exploit the natural coherence on the time dimension in motion blurred scenes. Second, by using OBBs in spatial dimensions, degeneration is avoided for challenging geometries. Third, as a BVH, the memory consumption of the structure is naturally bounded. And finally, the structure can be efficiently constructed by utilizing the *micropolygon grid*, a natural structure generated during micropolygon tessellation.

Based on the OBB hyper-trapezoid BVH, we develop a micropolygon ray tracing algorithm to render high quality defocus and motion blur effects. During ray traversal, the algorithm sorts rays using a coherence heuristic based on the per-ray motion time and ray directions to improve traversal coherence. The ray tracer, including both the BVH construction and ray traversal, can be efficiently implemented on the GPU, and integrated into existing GPU-based micropolygon renderers [Zhou et al. 2009]. The algorithm can be also extended to deal with a variety of effects, such as occlusion culling, secondary ray effects including shadow, reflection and refraction, and memory-bounded (and layer-limited) transparency.

In the rest of the paper, we first briefly review related work. In Section 3, we detail the acceleration structure and the micropolygon ray tracing algorithm, followed by the description of several extensions of the algorithm in Section 4. Section 5 evaluates the rendering quality and performance of our algorithm using several examples, and Section 6 concludes the paper.

2 Related Work

Micropolygon Rendering Micropolygons are the basic geometric element of the Reyes rendering architecture [Cook et al. 1987]. Several Reyes implementations, including Pixar's PRMan, have been widely used in film production. Recent research efforts focus on developing data-parallel algorithms for micropolygon rendering. Patney and Owens [2008] mapped the tessellation algorithms for micropolygon generation to the GPU. Fatahalian et al. [2009] investigated data-parallel implementations of a variety of micropolygon rasterization algorithms. RenderAnts [Zhou et al. 2009] implements a complete micropolygon-based renderer on the GPU and demonstrates significant speedups over CPU-based renderers. Our work provides micropolygon ray tracing as an alternative algorithm for the sampling stage of micropolygon renderers.

Defocus and Motion Blur Defocus and motion blur are important and common effects in real-world images. For cinematic rendering, these effects are typically computed using supersampling. Stochastic sampling is immune to aliasing artifacts [Cook et al. 1984; Cook 1986] while challenging for rasterizers to compute tight bounding boxes. Deterministic sampling like the accumulation buffer [Haeberli and Akeley 1990] provides tighter bounding boxes for rasterizers while being more prone to aliasing. State-of-the-art rasterizers make tradeoffs between sampling randomness and bounding box tightness to balance the rendering performance and image quality [Cook et al. 1993; Fatahalian et al. 2009].

A few recent publications seek to accelerate the rendering of defocus and motion blur effects using adaptive sampling and frequency analysis [Hachisuka et al. 2008; Egan et al. 2009; Soler et al. 2009]. The key difference between our work and theirs is that we focus on reducing the cost of sampling itself rather than the number of samples computed. Any sampling patterns proposed in their work can be used in our algorithm. In this paper we assume a uniform sampling strategy is used.

There are a few methods for approximating defocus and motion blur effects [Sung et al. 2002; Demers 2004; Kass et al. 2006; Lee et al. 2009]. These methods are designed to achieve high performance and may produce artifacts in some image regions. On the contrary, our method is designed to provide guaranteed quality while running as fast as possible. Our method guarantees an image quality no worse than a maximum-quality brute force rasterizer.

Ray Tracing and Acceleration Structures Distributed ray tracing [Cook et al. 1984] formulates defocus and motion blur rendering as a ray tracing problem while leaving out the acceleration of such formulation. Space-time ray tracing [Glassner 1988] accelerates ray tracing of motion blurred scenes using BVH based on 4D polyhedrons with up to twelve fixed-orientation faces. The algorithm simply treats the time dimension as an additional spatial dimension and ignores the unique geometry coherence property along the time dimension. Our work follows the formulation of distributed ray tracing and uses OBB hyper-trapezoids to exploit the inherent temporal coherence in motion blurred scenes to construct a space-time 4D hierarchy with significantly tighter bounding volumes.

Micropolygon ray tracing has been proved to be very useful in film production [Christensen et al. 2003; Christensen et al. 2006]. The basic idea of our hyper-trapezoid BVH resembles the moving Kay-Kajiya tree in [Christensen et al. 2006]. However, these previous works focus on tracing secondary rays. Primary-ray effects like defocus and motion blur are still rendered via rasterization. In contrast, our ray tracing algorithm is designed and optimized for tracing defocus and motion blur rays. In addition, our algorithm, including the BVH construction and ray traversal, is highly parallel and efficiently implemented on GPUs, which has never been achieved in previous micropolygon ray tracing.

Several parallel algorithms for constructing ray tracing acceleration hierarchies have been developed recently [Shevtsov et al. 2007; Wald 2007]. Parallel hierarchy construction has also been mapped to GPUs [Zhou et al. 2008; Lauterbach et al. 2009]. All these techniques are designed for axis-aligned hierarchies, which may degenerate for spatially-clustered non-axis-aligned geometries like hair and furs. Oriented hierarchies are less prone to non-axis-aligned geometries and more efficient in pruning intersection tests during ray traversal [Ize et al. 2008]. The high construction cost, however, remains an obstacle to the on-the-fly construction of oriented hierarchies. We present a novel oriented hierarchy design that allows fast GPU construction by utilizing the micropolygon grid, a natural structure generated during micropolygon tessellation.

Hyper-trapezoids have been first proposed for collision detection [Hubbard 1995]. This work simply uses axis-aligned hyper-trapezoids as approximation geometry for the moving object intersection test and does not construct any hierarchy. We extend their hyper-trapezoid definition with orientation in spatial dimensions and develop hyper-trapezoid hierarchy construction and traversal algorithms for ray tracing.

3 Micropolygon Ray Tracing

In this section, we first briefly revisit the ray tracing formulation of defocus and motion blur in Section 3.1, followed by the description of OBB hyper-trapezoids in Section 3.2. Section 3.3 details our hyper-trapezoid BVH construction algorithm. In Section 3.4, we discuss ray generation and coherence reorganization, and present the BVH traversal algorithm in Section 3.5. Finally, in Section 3.6, we describe how to integrate our ray tracer into a micropolygon rendering pipeline.

3.1 Ray Tracing Formulation

Both defocus and motion blur rendering can be formulated as ray tracing problems as in distributed ray tracing [Cook et al. 1984]. These formulations can be adapted to micropolygons in a straightforward manner.

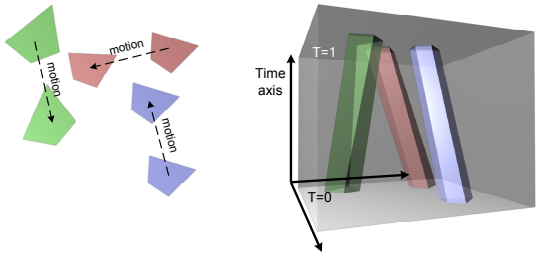
Our algorithm takes a set of shaded micropolygons and the camera configuration as input. A number of subpixel samples with assigned lens points and time stamps are generated for each pixel to produce defocus and motion blur effects. One primary ray is then generated and traced for each sample. For defocus, the ray starts from the assigned lens point of the corresponding subpixel and goes through the point on the focal plane corresponding to the subpixel. For motion blur, the ray is extended to 4D space-time with the assigned time stamp as the fourth dimension coordinate.

3.2 OBB Hyper-trapezoids

The key difference between traditional BVHs and our BVH is the shape of the bounding volume. The bounding shapes we use are 4D *OBB hyper-trapezoids* that project to 3D OBBs in the spatial dimensions. For the sake of simplicity, in the following we describe our hyper-trapezoids in the 3D space-time of motion-blurred 2D scenes. We also assume all motion to be linear, *i.e.*, the geometry at time $T = T_i$ is a linear interpolation of the geometry at time $T = 0$ and $T = 1$. Non-linear motions can be easily handled by piecewise-linear approximation [Christensen et al. 2006].

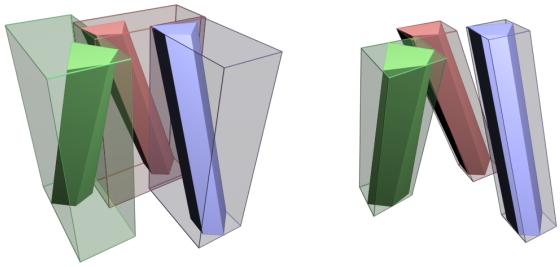
Consider a 2D dynamic scene shown in Fig. 2(a). By adding a third time axis, we can get a 2D space translated continuously in time, *i.e.*, the 3D space-time shown in Fig. 2(b). Motion rays with time stamps in the 2D space are mapped to ordinary rays perpendicular to the time axis in the 3D space-time. Motion blurred rendering of the original scene may thus be performed by tracing rays in the space-time using a 3D ray tracer.

Note that it is inefficient to use general-purpose 3D acceleration hierarchies in the space-time ray tracing. All shapes in the space-time are prismatoids that span across the entire time dimension. Axis-aligned hierarchies would be unable to utilize the time dimension at all and produce highly inefficient bounding volumes as shown in Fig. 3(a). General oriented hierarchies can produce tight bounding volumes, but would be forced to introduce rotation in the time dimension which destroys time-perpendicular properties and complicates intersection tests. Without special handling of the time dimension, direct 3D space-time ray tracing risks performance degradation due to such a special geometry distribution.



(a) A 2D scene with motion (b) Corresponding 3D space-time

Figure 2: *A 2D scene with motion and its corresponding 3D space-time. Each slice perpendicular to the time axis in the space-time corresponds to the 2D scene at a particular instant of time.*



(a) Axis-aligned bound. volumes (b) Hyper-trapezoid bound. volumes

Figure 3: *Axis-aligned bounding volumes and OBB hyper-trapezoid bounding volumes in a 3D space-time.*

We solve this problem by using OBB hyper-trapezoids as bounding volumes. We define an OBB hyper-trapezoid to be the linear combination of two oriented boxes with the same orientation lying on plane $T = 0$ and $T = 1$ respectively. Fig. 3(b) illustrates an example of OBB hyper-trapezoids. Intuitively, an OBB hyper-trapezoid in the 3D space-time corresponds to a moving oriented box in the 2D space whose start and end poses correspond to the top and bottom bases of the 3D hyper-trapezoid. Our hyper-trapezoids are able to adapt to motion and bound moving objects more tightly than axis-aligned volumes in space-time. The effectiveness of OBB hyper-trapezoids is proved in our experiments as described in Section 5.2. In addition, OBB hyper-trapezoids in space-time have the following property:

Property 1 *An OBB hyper-trapezoid bounds a moving object in the space-time if and only if the hyper-trapezoid bounds the object's start and end poses.*

This property allows some key hyper-trapezoid operations to be performed conveniently on the start and end poses in the original space. We utilize the property to design efficient BVH construction and traversal algorithms. Proof of Property 1 is straightforward: OBB hyper-trapezoids are convex shapes. If a hyper-trapezoid bounds an object's start and end poses, all points formed by the linear combination of the start and end poses are enclosed in the corresponding hyper-trapezoid, *i.e.*, the hyper-trapezoid bounds the moving object.

The hyper-trapezoid described above can be extended to the 3D space (*i.e.*, 4D space-time) in a straightforward manner. A 4D OBB hyper-trapezoid is defined to be the linear combination of two 3D OBBs with the same orientation at the start and end of a motion.

3.3 BVH Topology and Construction

Our BVH construction follows the general process of a top-down SAH-based BVH constructor [Wald et al. 2007]. We begin with one root node containing all input micropolygons and recursively perform node partitioning until a termination criterion is met. When a node is partitioned, all micropolygons inside it are partitioned into its two child nodes. This top-down recursive partitioning process generates a BVH topology, *i.e.*, a binary tree where each leaf node stores a list of micropolygons. After the topology is constructed, a bounding shape is computed for each node in a bottom-up manner to get the traversal-ready BVH. As aforementioned, the bounding shapes are 4D hyper-trapezoids with 3D OBBs as bases.

While the OBB hyper-trapezoid allows compact bounds of arbitrarily moving objects, it also significantly increases the number of possible node-splitting candidates [Ize et al. 2008], leading to high construction overheads. To address this issue, we exploit the temporal coherence in motion and the micropolygon grid structure used in typical micropolygon tessellation algorithms to constrain the split candidates within a small subset. First, we only consider the scene pose at $T = 0.5$ during BVH topology construction. Due to temporal coherence, the topology of this pose is expected to work reasonably well for temporally-close scene poses [Wald et al. 2007]. Second, we assume that the input data structure is an unorganized soup of micropolygon grids where each grid is a collection of micropolygons with a 2D parametric space. The majority of current micropolygon generation algorithms [Patney and Owens 2008; Fisher et al. 2009] consist of a recursive splitting stage and a regular or semi-regular tessellation stage. Such algorithms naturally generate the micropolygon grid structure we need as illustrated in Fig. 4.

Our BVH topology construction consists of two stages to match the grid data structure. We first build a top-level SAH BVH of the grids using the centroid-based SAH partitioning algorithm [Wald et al. 2007]. Grids in each node are recursively partitioned into child nodes with respect to their centroids using axis-aligned planes. During SAH computation, the traversal cost of each grid is computed as the number of micropolygons it contains. We continue splitting even if no split candidate can reduce the SAH cost. This guarantees that each leaf node in this top-level BVH contains at most one grid. Once the top-level BVH is constructed, each of its leaf nodes is replaced with an in-grid BVH constructed using approximated SAH splits. Within each grid, we approximate the optimal SAH split by simply splitting at the midpoint of the longer axis in the parametric space of the grid. The midpoint split stops once a node has less than eight micropolygons. Note that for uniform grids, the optimal SAH split coincides with the midpoint split. As micropolygon grids map to uniform grids in the parametric space, intuitively the midpoint split in the parametric space is an accurate approximation of the optimal SAH split in the world space when the mapping between the two spaces is sufficiently uniform. This intuition is verified in our experiments. In 94% of the cases we experimented with, the final SAH cost of the output subtrees produced by our parametric space algorithm is within 120% of the SAH cost produced by greedy SAH optimization. For details about the SAH experiments, please refer to Section 5.2.

The top-level BVH topology construction described above only uses axis-aligned partition planes, and is thus prone to degenerate geometries like dense non-axis-aligned curves in hair and fur scenes. We address this issue by switching to an orientation-aware split strategy when no split candidate can reduce the SAH cost or when a predefined depth limit is reached. The orientation-aware strategy constructs the BVH topology using 1D sorting. All grids in the node to be processed are sorted with respect to a 5D Morton code [Morton 1966] consisting of the grid orientation and centroid position. The orientation of a grid is computed as the average micropolygon normal for surfaces and the average segment direction for curves. The orientations are converted to 2D using a low distortion sphere-disk map [Shirley and Chiu 1997] and packed in the more significant position during Morton code construction. This Morton code sort groups grids with coherent orientation together to attempt to make the oriented bounding boxes to be computed more compact. A subtree topology is constructed from the sorted list by recursively median partitioning the sorted list.

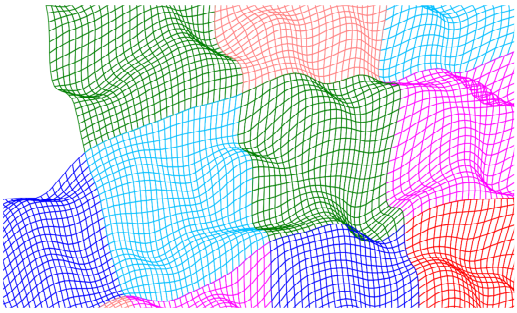


Figure 4: *Example micropolygon grids generated by a micropolygon renderer.*

After the BVH topology is constructed, we perform a bottom-up bounding and merging pass to compute the bounding hyper-trapezoids. A bounding hyper-trapezoid is first computed for each leaf node. Then for each inner node in the bottom-up order, the bounding hyper-trapezoids of its child nodes are merged to generate the bounding hyper-trapezoid of this inner node. Although the hyper-trapezoids are 4D, these two hyper-trapezoid operations can be implemented with only 3D operations:

- To compute the bounding hyper-trapezoid of a set of micropolygons, we compute two 3D OBBs with the same orientation for the start and end poses of the moving micropolygons as the respective hyper-trapezoid bases.

- To merge two hyper-trapezoids into their bounding hyper-trapezoid, the two bases of the two input hyper-trapezoids are separately joined into two 3D OBBs with the same orientation that are used as the bases of the final hyper-trapezoid.

Both operations require the orientation of the resulting hyper-trapezoid as input. Therefore, an orientation needs to be computed for each node before its bounding hyper-trapezoid is computed. Again, we utilize the micropolygon grid structure to efficiently compute orientations that allow compact bounding boxes. Specifically, we first compute the orientation for each micropolygon grid. For each grid, a main direction for each of its two tessellation directions is computed as the average of all respective micropolygon edges. The two main directions are then orthogonalized and combined with the normal vector of the plane they form to yield the per-grid orientation. We intentionally avoid PCA (Principal Component Analysis) during the main direction computation to reduce computational cost and improve numerical stability. After each per-grid orientation is computed, it is assigned to all nodes in the sub-tree the grid corresponds to. The orientations are then propagated bottom-up to top-level nodes. For each top-level node, two bounding hyper-trapezoids are computed according to the orientations of its two child nodes respectively. The orientation that leads to the bounding hyper-trapezoid with smaller surface area is chosen as the final orientation.

3.4 Ray Generation and Organization

We generate subpixel, defocus and motion blur sampling rays using jittered stratified sampling [Cook 1986]. To generate $N \times N$ rays for a given pixel, the three sampling domains, *i.e.*, subpixel, lens and time, are each divided into $N \times N$ regions of equal area. One sample is taken from each region in each sampling domain. All samples are randomly jittered within the corresponding regions. Finally, one lens sample and one time sample are assigned to each subpixel sample using two independent permutations for lens and time. Fig. 5 illustrates the sampling domains and permutations. Note that we use a per-pixel time permutation to eliminate the correlation between time and lens. Such per-pixel permutation is very challenging for rasterizers as the permutation conflicts with assumptions required for computing tight rasterization bounding boxes. The only rasterization method capable of handling our sampling pattern is to test all samples covered by the expanded bounding boxes of all micropolygons in a brute-force manner.

We generate all the random numbers required in the ray generation in parallel by repeatedly hashing the ray index using the final avalanching step of the SuperFastHash [Hsieh 2004]. The lens permutation we use is a magic square [Weisstein] and the time permutation is a shuffled version of the defocus magic square shifted by the hash of the pixel ID. This shift is omitted if defocus is disabled. For lens samples, currently we support disk and regular polygon-shaped lenses to enable a variety of bokeh effects. We first generate lens samples on a square and then map them to a disk using a low distortion disk-square map [Shirley and Chiu 1997]. Polygon-shaped lenses are handled by modifying the radius in the disk-square mapping.

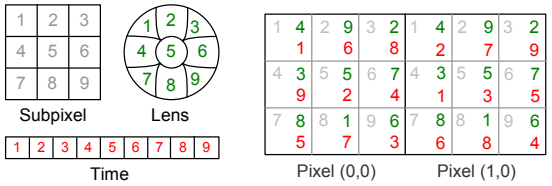
As pointed out by Aila and Laine [2009], SIMD efficiency is critical for GPU ray traversal performance. Therefore, we group together rays that are likely to yield similar branch decisions during traversal to improve the SIMD efficiency. First, we generate rays in 16×16 tiles in the image space. Within each tile, the 256 rays are organized in a Z-order curve [Morton 1966]. If motion blur is enabled, the rays are sorted by the 4-bit quantized motion time using a stable-sort. If defocus is enabled, the rays are sorted by the x and y components (each quantized to 4 bits) of ray direction vectors using a stable-sort. If both motion blur and defocus are enabled, the defocus sort is performed after the motion blur sort as maintaining coherence in defocus rays is more significant performance-wise.

3.5 BVH Traversal

Our BVH ray traversal algorithm follows the per-ray persistent while-while traversal algorithm [Aila and Laine 2009]. The main challenge in designing the algorithm is the intersection tests. Namely, we need to intersect motion rays with the 4D space-time hyper-trapezoids and moving micropolygons. Note that a space-time hyper-trapezoid may be interpreted as a moving OBB. Intersecting a motion ray with a hyper-trapezoid is thus equivalent to intersecting the ray with the corresponding OBB at the ray's assigned time. The same scheme can be applied to moving micropolygons. The problems are thus converted to intersecting rays with OBBs and micropolygons in the 3D spatial dimensions.

We convert oriented box intersection tests to axis-aligned box intersection tests by transforming rays into the per-box local frame. An advantage of our constructed BVH is that the box orientations do not change frequently during top-down hierarchy traversal. Therefore, ray transformation is performed less frequently than box intersections and does not introduce significant overhead compared to pure axis-aligned BVH traversal.

The micropolygon intersection test is more complicated. From a mathematical point of view, the 3D geometry of micropolygons is inherently ill-defined. The four vertices of a micropolygon are typically not coplanar. The edges may be concave or degenerate. It is difficult, if not impossible, to define an intersection-friendly 3D geometry for an arbitrary micropolygon. Therefore, we use a rasterization-like method to compute pseudo-intersections for micropolygons. The four vertices of the micropolygon to be tested and the ray termination point are first perspectively projected to 2D space along the z-axis as in a rasterizer. A 2D quad is constructed by connecting pairs of projected vertices that correspond to the original micropolygon edges. An even-odd rule 2D point-in-polygon test between the projected ray termination point and the 2D quad is then performed. If the point-in-polygon test returns positive, a potential intersection point is computed by bilinearly interpolating the z component of the four micropolygon vertices and inverse-projecting the projected ray termination point with the computed z. The selection of interpolation weights is insignificant as long as all weights are valid, *i.e.*, the sum of the four weights is 1 and all weights are non-negative. If the minimum and maximum z of micropolygon vertices and the minimum and maximum z of the ray overlap, the intersection test returns positive and the potential intersection point is returned. Otherwise the intersection test returns negative.



(a) Sampling domains and jittering regions (b) Permutations

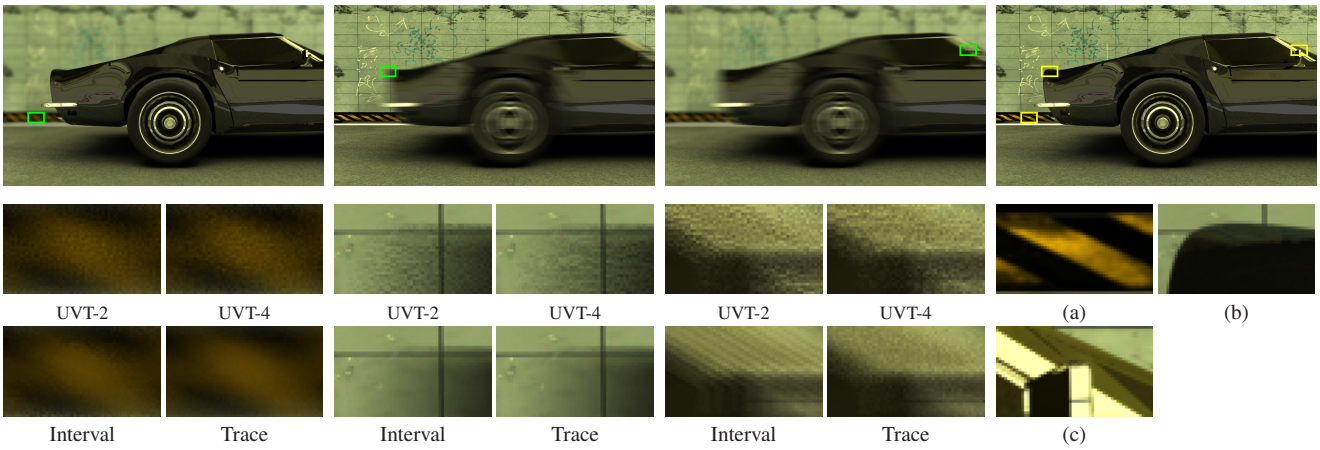
Figure 5: *Example ray generation for* 3×3 *supersampling.*

The use of the projection-based pseudo-intersection test guarantees two important properties of our algorithm. First, our micropolygon intersection test is equivalent to a coverage test in a rasterizer. Therefore, our algorithm produces no visual artifacts other than those produced by rasterization. Second, despite that our intersection test does not correspond to any well-defined 3D geometry, adding a bounding volume intersection test before the micropolygon intersection test does not result in cracks. The reason is as follows. All bounding volumes in our BVH are convex and bound all vertices of the contained micropolygons. Pseudo-intersection points inside the convex hull of the micropolygon vertices are never pruned by bounding volumes, and the potential pseudo-intersection points still form a water-tight shape.

3.6 Reyes Pipeline Integration

Our micropolygon ray tracer can be used in the sampling and composition stages in a Reyes pipeline. The ray tracer takes the shaded micropolygon grids from the preceding shading stage, constructs a BVH for the micropolygons, traces rays and returns a set of subpixel sample colors to the subsequent filtering stage.

We have implemented the described BVH construction and traversal algorithms on GPUs using the BSGP language [Hou et al. 2008]. Since most operations in our algorithms (*e.g.*, SAH-based splitting and ray traversal) have available GPU implementations in previous publications [Zhou et al. 2008; Lauterbach et al. 2009], implementing our ray tracer on the GPU is straightforward. We also integrated the ray tracer into RenderAnts [Zhou et al. 2009], an open source GPU-based Reyes renderer. We removed their original accumulation buffer based defocus and motion blur implementation and



(a) Defocus only (b) Motion blur only (c) Defocus + motion blur (d) Perfect focus

Figure 6: *Render quality comparison. The small images in the bottom rows are close-up views for the boxes in the top. All large images shown here are generated by our ray tracer. Images are rendered at* 1280×720 *with* 13×13 *supersampling.*

modified their pipeline to call our ray tracer.

4 Extensions

In this section, we describe several extensions of our micropolygon ray tracing algorithm.

Secondary Rays An obvious extension is to use our ray tracer to compute secondary ray effects like hard shadow, reflection and refraction. However, directly tracing secondary rays on Reyes-generated micropolygons would yield incorrect results as Reyes micropolygons are generated using view-dependent tessellation. Therefore, we first generate a secondary ray friendly micropolygon representation of the scene using non-culled view-independent tessellation. Ray tracing interfaces on this representation are then provided to shaders. We currently provide two ray tracing interfaces, one only tests whether there is an intersection and one computes shading at the intersection point. Using these interfaces, we implemented ray-traced hard shadow, reflection and refraction. These effects are rendered in our fur ball and car scenes shown in Fig. 1 and Fig. 7(a).

The geometric precision for secondary intersections with parametric surfaces are limited by the available memory because we currently do not support out-of-core BVH traversal. For strongly enlarging reflection/refraction effects, parametric surface silhouettes may not be very smooth. On the other hand, shading quality for reflection/refraction is less affected by this limitation as we recompute surface parameters for parametric surfaces at secondary ray hit points during shading. In practice, we find that our secondary effect quality is acceptable for near-flat reflective/refractive surfaces and for hard shadows at non-extreme angles.

Ray-micropolygon intersection for secondary rays has a minor difference with primary ray intersection. Instead of projecting micropolygons along the z axis, we project micropolygons along the axis on which the ray direction has the maximum absolute value.

Transparency Memory-bounded transparency and layer-limited transparency can be conveniently implemented using ray tracing. One important property of ray tracing is that it can start at any point, and can return the nearest m intersections instead of just one. This allows us to process transparency in multiple passes in the front-to-back order and only compute a limited number of layers in one pass. For memory-bounded transparency, we restrict the total number of transparent layers processed in one pass according to the available memory. After each pass, all generated layers are accumulated to the final image and all memory used during the pass is freed. This property guarantees that the algorithm never fails due to out-of-memory errors, even in extreme cases where the transparent layers on a single subpixel sample cannot fit in memory. For layer-limited transparency, we simply ignore all transparent samples beyond the nearest m layers, where m is a user-specified value. With a proper m, rendering of hair/fur scenes may be significantly accelerated without introducing visible artifacts. Our fur ball scene shown in Fig. 7(a) requires memory-bounded transparency to correctly render in-core, and its number of transparent layers is limited to five to accelerate rendering.

Culling Ray tracing can be also used as a visibility testing tool for pre-shading culling. Prior to shading, we detect visible micropolygon grids by tracing a few primary rays on random points of each grid and only shade the grids that are hit by at least one ray. Other grids are only shaded when they are intersected during the subsequent sampling stage. In our experiments, the culling may reduce shading workload by approximately 40%. However, the parallelism during the shading stage becomes fragmented and the GPU becomes under-utilized. Although we do not observe any acceleration in our test scenes, we expect that such culling would be more effective for film production scenes with more complicated shaders.

5 Experimental Results

In this section, we evaluate the described ray tracing algorithm using several scenes on a machine with one NVIDIA GTX 285 GPU. Our evaluation consists of two parts. Rendering performance and quality comparisons are reported in Section 5.1, and statistical data obtained through controlled study are analyzed in Section 5.2.

We compare our algorithm with the INTERVAL and INTERLEAVEUVT rasterizers described in Fatahalian et al. [2009]. Code-optimized GPU implementations of the rasterizers are used in the comparisons. The rasterizers do not perform any occlusion culling, as GPU occlusion culling in the presence of defocus and motion blur is a challenging problem in itself. Note that both rasterizers make performance/quality tradeoffs. The INTERVAL rasterizer splits the sampling domain to S intervals and uses jittered stratified

(a) Furball Family, defocus
17.6M mpolys, 1280 × 960 (13 × 13)
14.3 mpolys per pixel

(b) Ladybug, defocus
24.1M mpolys, 1280 × 960 (23 × 23)
19.6 mpolys per pixel

(c) Fairy, defocus + motion blur
5.96M mpolys, 1280 × 720 (13 × 13)
6.47 mpolys per pixel

Figure 7: *Images and statistics of several test scenes. "mpolys" is the total number of micropolygons in the scene.*

	DOF (a)	MB (b)	DOF + MB (c)
Our RAYTRACE	10.5	13.2	17.2
INTERVAL	42.9	18.6	91.9
INTERLEAVEUVT, $k = 2$	77.1	68.7	80.7
INTERLEAVEUVT, $k = 4$	261.3	227.8	266.8

Table 1: *Render time (in seconds) for all method/effect combinations shown in Fig. 6.*

	Furball	Ladybug	Fairy	Car
Our RAYTRACE	52.7	44.4	10.9	28.2
INTERVAL	119.3	163.4	32.3	178.0
INTERLEAVEUVT, $k = 2$	57.5	148.1	20.2	214.4
INTERLEAVEUVT, $k = 4$	155.9	484.4	64.2	710.6

Table 2: *Sampling time (in seconds) for all test scenes. For animations, only the statistics of the frame shown in Fig. 7 or Fig. 1 is reported.*

	Furball	Ladybug	Fairy	Car
Tessellation	1.4	2.6	0.1	1.1
Shading	8.1	81.0	7.7	213.2
Sampling	52.7	44.4	10.9	28.2
Composition	158.8	5.5	44.6	15.9
Total	221.0	133.5	63.3	258.4

Table 3: *Total render time and per-stage time of our rendering system for all test scenes. All timings are in seconds.*

sampling to generate a subpixel sample in each interval. We set S to be the same as our super-sampling rate. Jittering allows alias-free rendering of individual effects. However, INTERVAL always pairs the same time region with the same lens region. This pairing creates correlation between defocus and motion blur and may result in aliasing if both effects are combined. The INTERLEAVEUVT rasterizer disables jittering and uses a fixed number of subpixel-lens-motion tuples in the entire screen while allowing each pixel to contain a different set of tuples. The lack of jittering produces aliasing and the algorithm hides aliasing by permutating tuple-pixel assignments. The amount of subpixel-lens-motion tuples is controlled by a parameter called *tile size*. In the following we use the letter k to represent tile sizes. Increasing k improves quality while also increasing render time.

Four test scenes are used in our evaluation:

- **Furball Family** shown in Fig. 7(a). A static scene of two fur balls rendered with only defocus. The scene contains 495K transparent hairs. The marble floor is reflective.

- **Ladybug** shown in Fig. 7(b). A flower field with a ladybug and a few dewdrops. The scene has multiple layers of grass and flowers. Out-of-focus dewdrop highlights produce hexagonal bokeh effects. Some mild HDR glow effects are added as a postprocess.

- **Fairy** shown in Fig. 7(c). We exported 189 frames of the Fairy animation in the Utah Animation Repository. The scene contains a dancing fairy, a dragonfly flying in and out of focus and a moving camera.

- **Car** shown in Fig. 1. A car driving in a tunnel. The defocus blur of distant and bright traffic lights requires very high supersampling to render smoothly. The body of the car is reflective. The scene contains ambient occlusion rendered using a GPU implementation of the point-based approximate color bleeding algorithm [Christensen 2008]. HDR glow effects are added as a postprocess.

5.1 Quality and Performance

Fig. 6 illustrates the rendering quality of our method and the rasterization-based approaches under different effect combinations in the Car scene. Render time for the images in Fig. 6 is provided in Table 1. Our method consistently achieves higher performance than the rasterization-based approaches. Compared with INTERVAL, our method does not produce aliasing artifacts for combined defocus and motion blur effects and achieves higher performance. For relatively mild, non-combined effects, our method and INTERVAL are comparable. Compared with INTERLEAVEUVT, our method produces less noise and achieves significantly higher performance than high-quality settings of INTERLEAVEUVT.

Table 2 shows the sampling time for the test scenes. All scenes are rendered within a few minutes at supersampling rates above 11×11, PRMan's "production" preset. As illustrated, our method is up to an order of magnitude faster than INTERLEAVEUVT with $k = 4$ and several times faster than INTERVAL. The total render time and per-stage time of our rendering system are also reported in Table 3.

We evaluate the efficiency of our OBB hyper-trapezoid BVH by comparing with alternative acceleration structures, including a simple 4D AABB BVH, an AABB-based hyper-trapezoid BVH and a greedy SAH BVH. The topology of all BVHs are the same. Table 4 shows the comparison results. As illustrated, the hyper-trapezoid is orders of magnitude more efficient than simple 4D AABBs. Up to 200× traversal speedups have been observed for high-motion

		Furball	Ladybug	Fairy	Car
Ours	Build	5.0	6.7	0.8	4.6
	Trace	47.6	38.2	10.0	23.5
AABB Hyper.	Build	-	-	0.7	3.6
	Trace	-	-	13.5	60.0
3D/4D AABB	Build	4.4	5.6	0.7	3.7
	Trace	62.3	79.2	16.9	5077.4
No parametric -space split	Build	51.5	67.8	8.2	104.3
	Trace	60.9	84.0	10.0	53.0
No orientation -aware split	Build	5.1	6.8	0.8	4.7
	Trace	47.6	42.9	10.0	23.5
No orientation propagation	Build	5.1	6.6	0.7	4.3
	Trace	43.6	57.2	10.3	26.7
No ray sort	Trace	51.2	38.1	10.2	23.3

Table 4: *Build and trace time comparison for our OBB hyper-trapezoid BVH, AABB hyper-trapezoid BVH, 3D/4D AABB BVH and several other alternative build/trace schemes. All timings are in seconds. For defocus-only scenes, our approach falls back to OBBs and we only compare with 3D AABB BVH. For motion blur scenes, we compare with 4D AABB BVH.*

scenes like Car. OBB hyper-trapezoids also achieve significant traversal speedups compared with AABB hyper-trapezoids. The construction time for all other BVHs only takes a small fraction of the total rendering time and the difference in construction time is negligible with respect to the choice of bounding shapes.

We also evaluate the performance impact of several algorithmic optimizations by comparing build/trace time with individual optimizations disabled. As illustrated in Table 4, the parametric-space split is the most critical part of our algorithm. Without it, the BVH construction time is significantly increased and ray tracing time is also increased. The orientation propagation helps traversal performance considerably for most scenes. The orientation-aware split is mainly a fallback for special cases and it only helped the Ladybug scene. The performance impact of ray pre-sorting depends on the percentage of blurry pixels and blur radius. In some cases (*e.g.*, Ladybug and Car) the cost of sorting rays may cancel out the benefit of coherence improvement.

The peak memory consumption of our algorithm is about 10 bytes per micropolygon during BVH construction and 650 bytes per ray during ray traversal with the input data allocated by the Reyes pipeline excluded. Assuming 100MB GPU memory is available in the sampling stage, we are able to process millions of micropolygons and hundreds of thousands of rays in parallel, which is sufficient for full utilization of modern GPUs.

Note that all algorithms implemented in this paper consume all available GPU memory to process as many micropolygons and rays as possible in parallel to maximize the rendering performance. We utilize the dynamic scheduling system described in RenderAnts [Zhou et al. 2009] to recursively subdivide the rendering region until the memory required by an algorithm is less than the available GPU memory. As a result, the memory consumption differences between the algorithms are converted to execution time differences.

Our algorithm is designed for off-line software rendering and it does not necessarily perform well under real-time level parameter settings. One important reason is that the BVH construction becomes a major bottleneck at lower sampling rates. Table 5 shows the ray-tracing/rasterization performance comparison for several scenes at various polygon sizes (controlled via shading rates) and sampling rates. As illustrated, INTERLEAVEUVT becomes faster than ray tracing at lower sampling rates. However, our ray tracing

Furball	1×1	4×4	8×8	Ref	SR×4	SR×16
RAYTRACE	5.2	10.2	24.9	52.7	64.4	153.4
INTERVAL	18.7	34.6	65.2	119.3	74.0	71.6
UVT-2	0.6	5.6	21.9	57.5	33.3	31.6
UVT-4	1.2	15.0	59.1	155.9	70.2	52.2
Ladybug	1×1	4×4	8×8	Ref	SR×4	SR×16
RAYTRACE	3.0	4.2	7.9	44.4	40.7	38.1
INTERVAL	7.2	15.8	32.9	163.4	137.3	52.3
UVT-2	0.5	4.6	17.9	148.1	124.4	35.0
UVT-4	1.1	15.0	59.4	484.4	406.9	74.6
Fairy	1×1	4×4	8×8	Ref	SR×4	SR×16
RAYTRACE	1.0	1.9	4.5	10.9	8.4	7.7
INTERVAL	3.6	7.9	16.5	32.3	15.6	11.3
UVT-2	0.2	1.9	7.7	20.2	9.3	5.7
UVT-4	0.4	6.1	24.3	64.2	23.9	11.7
Car	1×1	4×4	8×8	Ref	SR×4	SR×16
RAYTRACE	4.4	5.3	7.5	28.2	24.3	24.7
INTERVAL	13.5	9.3	24.4	178.0	57.0	27.8
UVT-2	0.8	6.8	26.0	214.4	53.2	28.3
UVT-4	1.7	21.6	85.2	710.6	177.7	72.5

Table 5: *Performance comparison at different sampling rates and shading rates (i.e., polygon sizes). The 1×1, 4×4, 8×8 numbers are sampling rates. SR×4 means 4× polygon size, i.e., 4× coarser shading rate. SR×16 means 16× polygon size. The "Ref" column contains the reference performance data where sampling/shading rate settings are the same as the measurements in Table 2. The detailed shading rate assignment is: Furball (0.8), Ladybug (1 for grass and sky, 0.1 for ladybug, flowers and dewdrops), Fairy (0.3), Car (0.2).*

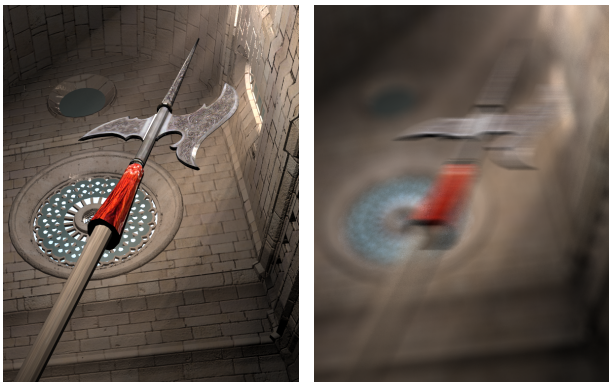
remains faster than INTERVAL as low sampling rates lead to fewer intervals and more bloated bounding boxes.

Polygon sizes have different impacts on ray tracing and rasterizers. For ray tracing, polygon sizes take effect indirectly by affecting the memory access coherence during BVH traversal. Larger polygons make ray-micropolygon intersection tests more coherent. However, they are more likely to overlap and harder to separate during BVH construction, decreasing the BVH quality. Furball is an extreme example of BVH quality degradation for larger polygons. Rasterizers are significantly more efficient for larger and coarser polygons. It should be noted that the shading rates in our scenes are manually tweaked by our artist. Specifically, as Reyes only computes shading at micropolygon vertices, the artist purposefully over-tessellated reflective/bumpy objects to improve shading anti-aliasing. At the coarser shading rates where rasterizers become efficient, bumpy faces look flat and the reflections/refractions contain severe aliasing artifacts. Therefore, such settings are not suitable for off-line rendering.

5.2 Controlled Study

Our controlled studies are performed in a stand-alone program using a simple test scene shown in Fig. 8 in order to minimize the influence of external factors. The simple scene does not require the various scheduling operations in a Reyes pipeline to render and its motion and defocus scales may be easily controlled.

To evaluate the SAH approximation accuracy of the parametric space split algorithm in our bottom-level BVH construction, we tested the algorithm on the micropolygon grids in the halberd scene. For each grid, we compute the relative difference in SAH cost between our approximation and the greedy SAH optimization in Wald



(a) Perfect focus (b) Defocus + Motion blur

Figure 8: *The simple scene used for controlled study.*

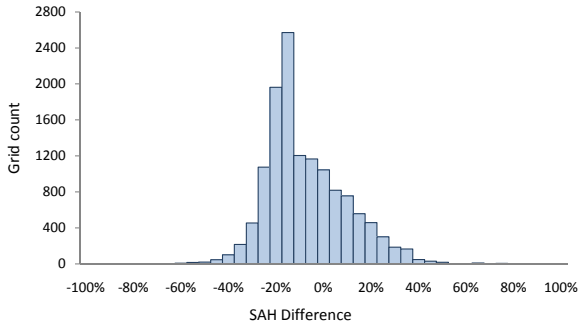


Figure 9: *Relative SAH difference distribution in SAH approximation evaluation. A negative value indicates our method has smaller SAH costs.*

et al. [2007]. Since the original greedy optimization only considers axis-aligned splits, we modify it to consider splits aligned with each grid's final bounding box orientation to make the comparison fair. For simplicity, the comparison is performed in 3D without regarding motion blur. Fig. 9 illustrates the distribution of the relative differences. Negative values indicate that our method produces smaller SAH costs. The average difference is -7.4%. We outperform the greedy SAH optimization in 75% of the cases, and perform no worse than 20% in 94% of the cases. The minimum and maximum differences are -76% and $+101\%$ respectively. This comparison demonstrates that our parametric space split performs competitively to the greedy SAH optimization, and may achieve better results in certain cases.

Theoretically, the time complexity of our ray tracing algorithm stays constant with respect to motion and defocus scales. In practice, however, our GPU implementation suffers from decreased control flow and memory coherence as the motion and defocus scale increases. To evaluate such behaviors, we plot the render time of INTERVAL, INTERLEAVEUVT ($k = 4$) and our algorithm at different motion and defocus scales. The render times are for the halberd scene at motions of $0 - 190$ pixels and $0 - 9.5\times$ defocus scales. Larger defocus scales correspond to larger circles of confusion. As illustrated in Fig. 10, compared with INTERVAL, the render time of our method is significantly less sensitive to the defocus scale and we are able to achieve higher speedups for large defocus blurs. Compared with INTERLEAVEUVT ($k = 4$), our method is more sensitive to motion and defocus scales. Nevertheless, our render time does not degrade in an unbounded manner like INTERVAL and our method remains faster than INTERLEAVEUVT even at extreme motion and defocus scales.

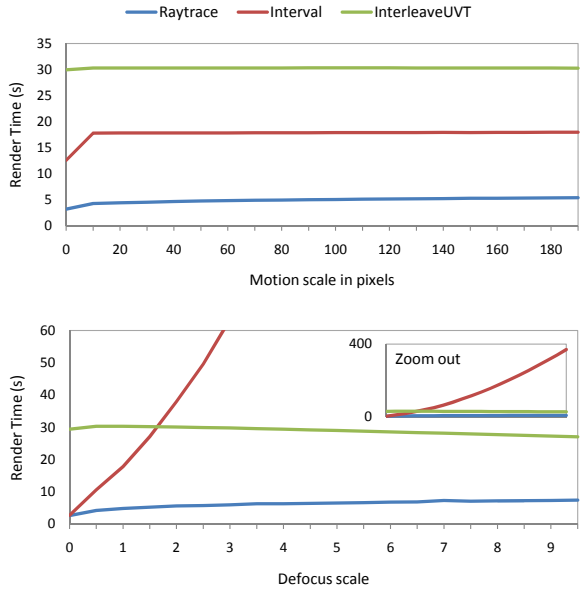


Figure 10: *Render time at different motion and defocus scales. Top: fixed defocus scale and varied motion scale. Bottom: fixed motion scale and varied defocus scale.*

6 Conclusion and Future Work

We have presented a micropolygon ray tracing algorithm that is more efficient than existing rasterization-based methods for rendering high quality defocus and motion blur effects. To the best of our knowledge, this is the first ray tracing algorithm with on-the-fly acceleration structure construction that outperforms the state-of-the-art rasterization methods under certain circumstances (*i.e.*, high quality defocus and motion blur). We are able to eliminate the quality-performance tradeoff in defocus and motion blur rendering and provide maximum quality with superior performance.

Although our tracing algorithm greatly accelerates the sampling stage, the speedup in overall rendering time may not be significant if rendering cost is dominated by very complicated shaders, as is typically the case in off-line movie rendering systems. But for simpler shaders, the speed-ups can be significant. In both cases, our ray tracing algorithm serves as a competitive alternative to rasterization and can fundamentally dictate the design of the remainder of these systems.

One limitation of our algorithm is the inefficiency of transparency handling. While our algorithm is able to process large sets of transparent samples in-core, our cost of processing each transparent sample is higher than rasterizers. The BVH becomes considerably less efficient in pruning intersection tests when tracing rays inside objects. Improving ray tracing performance for transparent scenes is an interesting direction for future work.

There are a number of future work directions in addition to improving transparency efficiency. First, we are interested in combining recent adaptive sampling techniques [Hachisuka et al. 2008; Overbeck et al. 2009] with our ray tracing pipeline. Such combination would combine the strengths of adaptive sampling and shading-sampling decoupling in micropolygon pipelines to further accelerate the rendering. Second, faster micropolygon BVH construction suitable for realtime ray tracing is also an interesting research direction.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments, Shuitian Yan for artistry support, and Steve Lin for proofreading this paper. Kun Zhou was partially funded by the NSF of China (No. 60825201).

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of HPG '09*, ACM, New York, NY, USA, 145–149.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Proceedings of Eurographics 2003*, Blackwell Publishers, 543–552.

CHRISTENSEN, P., FONG, J., LAUR, D., AND BATALI, D. 2006. Ray tracing for the movie 'cars'. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 1–6.

CHRISTENSEN, P. H. 2008. Point-based approximate color bleeding. Tech. rep., Pixar Technical Memo #08-01.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3, 137–145.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21*, 4, 95–102.

COOK, R. L., PORTER, T. K., AND CARPENTER, L. C., 1993. Pseudo-random point sampling techniques in computer graphics, US Patent 5239624.

COOK, R. L. 1986. Stochastic sampling in computer graphics. *ACM Trans. Gr. 5*, 1, 51–72.

DEMERS, J. 2004. Depth of field: A survey of techniques. *GPU Gems*, 375–390.

EGAN, K., TSENG, Y.-T., HOLZSCHUCH, N., DURAND, F., AND RAMAMOORTHI, R. 2009. Frequency analysis and sheared reconstruction for rendering motion blur. *ACM Trans. Gr. 28*, 3, 1–13.

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of HPG '09*, 59–68.

FISHER, M., FATAHALIAN, K., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Diagsplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Trans. Graph. 28*, 5, 1–10.

GLASSNER, A. S. 1988. Spacetime ray tracing for animation. *IEEE Comput. Graph. Appl. 8*, 2, 60–70.

HACHISUKA, T., JAROSZ, W., WEISTROFFER, R. P., DALE, K., HUMPHREYS, G., ZWICKER, M., AND JENSEN, H. W. 2008. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Trans. Gr. 27*, 3, 1–10.

HAEBERLI, P., AND AKELEY, K. 1990. The accumulation buffer: hardware support for high-quality rendering. In *Proceedings of ACM SIGGRAPH '90*, 309–318.

HOU, Q., ZHOU, K., AND GUO, B. 2008. BSGP: Bulk-synchronous GPU programming. *ACM Trans. Gr. 27*, 3, 9.

HSIEH, P., 2004. Hash functions. http://www.azillionmonkeys.com/qed/hash.html.

HUBBARD, P. M. 1995. Collision detection for interactive graphics applications. *IEEE TVCG 1*, 3, 218–230.

IZE, T., WALD, I., AND PARKER, S. G. 2008. Ray tracing with the BSP tree. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 159–166.

KASS, M., LEFOHN, A., AND OWENS, J. 2006. Interactive depth of field. Tech. rep., Pixar Technical Memo #06-01.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2, 375–384.

LEE, S., EISEMANN, E., AND SEIDEL, H.-P. 2009. Depth-of-field rendering with multiview synthesis. *ACM Trans. Gr. 28*, 5, 1–6.

MORTON. 1966. A computer oriented geodetic data base and a new technique in file sequencing. Tech. Rep. Ottawa, Ontario, Canada.

OVERBECK, R. S., DONNER, C., AND RAMAMOORTHI, R. 2009. Adaptive wavelet rendering. *ACM Trans. Graph. 28*, 5, 1–12.

PATNEY, A., AND OWENS, J. D. 2008. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Gr. 27*, 5, 1–8.

SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. 2007. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Proceedings of Eurographics '07*, 395–404.

SHIRLEY, P., AND CHIU, K. 1997. A low distortion map between disk and square. *Journal of Graphics Tools 2*, 3, 45–52.

SOLER, C., SUBR, K., DURAND, F., HOLZSCHUCH, N., AND SILLION, F. 2009. Fourier depth of field. *ACM Trans. Gr. 28*, 2, 1–12.

SUNG, K., PEARCE, A., AND WANG, C. 2002. Spatial-temporal antialiasing. *IEEE TVCG 8*, 2, 144–153.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Gr. 26*, 1, 6.

WALD, I. 2007. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 33–40.

WEISSTEIN, E. W. Magic square. http://mathworld.wolfram.com/MagicSquare.html.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Trans. Gr. 27*, 5, 126.

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. RenderAnts: interactive Reyes rendering on GPUs. *ACM Trans. Gr. 28*, 5, 1–11.