A Shading Reuse Method for Efficient Micropolygon Ray Tracing

Qiming Hou Kun Zhou
State Key Lab of CAD&CG, Zhejiang University[*]

Abstract

We present a shading reuse method for micropolygon ray tracing. Unlike previous shading reuse methods that require an explicit object-to-image space mapping for shading density estimation or shading accuracy, our method performs shading density control and actual shading reuse in different spaces with uncorrelated criterions. Specifically, we generate the shading points by shooting a user-controlled number of shading rays from the image space, while the evaluated shading values are assigned to antialiasing samples through object-space nearest neighbor searches. Shading samples are generated in separate layers for first bounce ray paths to reduce spurious reuse from very different ray paths. This method eliminates the necessity of an explicit object-to-image space mapping, enabling the elegant handling of ray tracing effects such as reflection and refraction. The overhead of our shading reuse operations is minimized by a highly parallel implementation on the GPU. Compared to the state-of-the-art micropolygon ray tracing algorithm, our method is able to reduce the required shading evaluations by an order of magnitude and achieve significant performance gains.

Keywords: micropolygon, GPU, Reyes, ray tracing

Links: DL PDF

1 Introduction

Shading is typically the performance bottleneck in cinematic-quality rendering, which is often based on the Reyes architecture and uses micropolygons to represent high order surfaces or highly detailed objects [Cook et al. 1987]. In order to reduce shading costs, state-of-art micropolygon renderers (e.g., Pixar's RenderMan) perform shading computation on micropolygon vertices, and reuse the shading values to evaluate the color of each visibility sample (or antialiasing sample) and composite the final image. Such a shading reuse strategy enables a shading rate significantly lower than the visibility sampling rate, which is vital for efficient high-quality rendering where extremely high supersampling of visibility is necessary, especially when rendering defocus and motion blur.

Existing shading reuse methods for micropolygon rendering are primarily designed for rasterization based pipelines. Ray tracing effects such as reflection and refraction are typically considered as a part of shading in such methods. Consequently, all reflected/refracted samples have to be shaded, incurring significant overhead. As ray tracing achieves more significance in modern high-quality rendering [Parker et al. 2010], this may become a major obstacle in future applications.

[*]Email: hqm03ster@gmail.com, kunzhou@acm.org

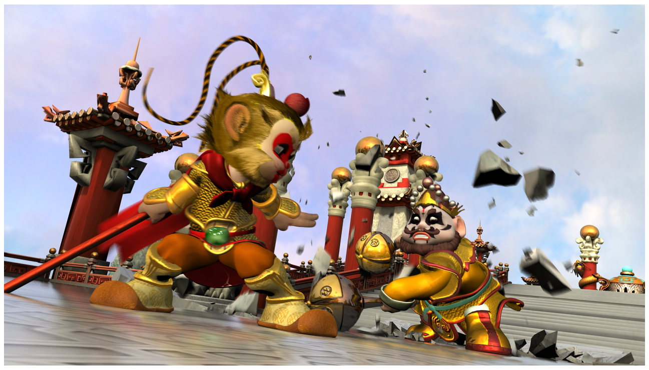


Figure 1: *An animated scene rendered with motion blur and reflection. This scene contains 1.56M primitives and is rendered with 2 reflective bounces at* 1920×1080 *resolution and* 11×11 *supersampling. The total render time is 289 seconds. On average only 3.48 shading evaluations are performed for each pixel and are reused for other samples.*

In this paper, we introduce a simple but effective method to reuse shading evaluations for efficient micropolygon ray tracing. Compared to the state-of-the-art micropolygon ray tracing algorithm, our method is able to reduce the required shading evaluations by an order of magnitude and achieve significant performance gains.

1.1 Related Work

Extensive research has been done on micropolygon rendering and ray tracing.

Researchers have explored efficient parallel implementations of micropolygon rendering on GPUs [Wexler et al. 2005; Patney and Owens 2008; Zhou et al. 2009; Hou et al. 2010]. In particular, Hou et al. [2010] introduced a GPU-based micropolygon ray tracing algorithm. They demonstrated that for high-quality defocus and motion blur ray tracing can greatly outperform rasterization methods. In their method, ray tracing is only used for visibility sampling and shading is still performed on micropolygon vertices. Another branch of research also seeks to accelerate micropolygon rendering using GPUs [Fisher et al. 2009; Fatahalian et al. 2009; Fatahalian et al. 2010; Ragan-Kelley et al. 2011; Burns et al. 2010]. The key difference between our work and theirs is that while they propose new GPU architecture designs that support real-time micropolygon rasterization, we aim to accelerate high-quality, off-line ray tracing using software approaches on current GPU hardware.

A majority of micropolygon rendering algorithms are designed to reuse the expensive shading computations across multiple visibility samples, assuming that shading is continuous and does not vary significantly across adjacent visibility samples. Existing shading reuse methods can be categorized into object-space methods [Cook et al. 1987; Burns et al. 2010] and image-space methods [Ragan-Kelley et al. 2011]. In [Cook et al. 1987], shading is performed on micropolygon vertices and reused within the same micropolygon. Burns et al. [2010] define a shading grid based on a priori shading density estimation, while shading values are evaluated lazily after visibility is resolved and reused within the same shading grid cell. Decoupled sampling [Ragan-Kelley et al. 2011] implicitly partitions antialiasing samples into equivalent classes using shading density controlling hash functions. Shading is reused within the same equivalent class.

Stoll et al. [2006] introduced object-space shading reuse into a ray tracing pipeline. They use ray derivatives [Igehy 1999] to control shading computation rates. Specifically, they conservatively discretize the minimum width of the derivative beam cross section into a few predefined object space tessellation grids. Shading is then computed for all tessellation grids hit by at least one ray. As noted in their paper, such an approach may result in considerable over-shading when there are highly anisotropic derivatives or significant over-tessellation. To avoid such reliance on ray derivative behaviors, our method controls shading reuse using nearest neighbor searches, which are independent of tessellation and automatically adapt to anisotropy.

1.2 Challenge and Contribution

The main challenge in generalizing existing shading reuse methods to ray tracing is that ray tracing complicates the object-to-image space mapping. Such a mapping is inherently required for shading reuse as the ideal shading evaluation density is defined in the image space while the shading continuity assumption is only valid in the object space. Object-space methods such as [Burns et al. 2010] reuse shading values based on object-space proximities and rely on image-space polygon size to control the shading evaluation density. In a rasterization-based pipeline, the size can be directly computed by simply projecting polygons to the image space. However, ray tracing may introduce arbitrary distortion and the image-space polygon size is no longer practical to compute. Image-space methods (e.g., [Ragan-Kelley et al. 2011]) reuse shading values based on image-space proximities and rely on a continuous object-image space mapping for shading accuracy. This assumption is trivially valid for direct rasterization. Defocus and motion blur effects can be handled by using the non-blurred image space for the shading reuse purpose. However, the same cannot be done for ray tracing effects like glossy reflection, which introduce a discontinuous object-image space mapping that cannot be worked around.

We propose to perform shading density control and actual shading reuse in different spaces with uncorrelated criterions. Specifically, we generate the shading points by shooting a user-controlled number of shading rays from the image space, while the evaluated shading values are assigned to antialiasing samples through object-space nearest neighbor searches. This method eliminates the necessity of an explicit object-to-image space mapping, enabling the elegant handling of ray tracing effects.

The nearest neighbor search employed in our method may cause significant performance overhead, which should be minimized to maintain the high efficiency of shading reuse. Observing the fact that micropolygons are often generated by dicing high-level parametric primitives, we design an algorithm to convert the 3D object-space nearest neighbor search to a 2D parametric-space search, and facilitate an accurate and efficient nearest neighbor search in shading reuse.

We further developed a highly parallel implementation of the shading reuse method on the GPU. We show that by replacing vertex shading with our method in a GPU-based micropolygon ray tracer, shading costs can be greatly reduced by $17\times$ and end-to-end render time can be improved by $3.5\times$. The quality and performance of our method was evaluated on a variety of complex scenes.

Listing 1 Pseudocode of our shading reuse method

```
 1 //Generate and shade seeding rays
 2 shading_rays = generateShadingRays(n*shading_rate)
 3 shading_samples = pathTraceAll(shading_rays)
 4 shade(shading_samples)
 5 //Trace antialiasing rays and reuse shading
 6 aa_rays = generateAntialiasRays(n*m)
 7 aa_samples = pathTraceAll(aa_rays)
 8 failed = new List
 9 for each s in aa_samples
10     s.shading = shading_samples.findNearest(s)
11     if s.shading == NOT_FOUND:
12         failed.add(s)
13 //Shade failure samples with image space reuse
14 cache = new Hash
15 for each s in failed
16     if cache.canReuseShadingAt(s):
17         s.shading = cache.fetch(s)
18     else
19         shade(s)
20         cache.add(s)
21 //Filter all samples to produce the final image
22 final_image = downSampleAndFilter(aa_samples)
```

2 Shading Reuse Method

Our shading reuse method works as the top level rendering loop of a ray tracer. For an image rendered at the resolution of n pixels with $m\times$ supersampling, Listing 1 shows the pseudocode of our method.

The method mainly consists of four steps. First, a certain number of shading rays are generated uniformly in the image space and traced to get the shading samples, which are then shaded. Second, antialiasing rays are generated and traced to obtain the antialiasing samples. For each antialiasing sample, the nearest reusable shading sample is located and its shading value is assigned to the antialiasing sample. Third, for those antialiasing samples that cannot find a reusable shading sample in the last step, a fall-back image-space shading reuse method is used to reshade these samples. This process is repeated for several layers of samples to separate potentially different shading ray paths. Finally, all antialiasing samples are filtered to produce the final image.

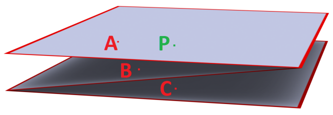
2.1 Sample Generation and Shading Formulation

Our method controls the shading density through shading ray generation. The user specifies the number of required shading evaluations for each pixel, i.e., `shading_rate`. As illustrated in Listing 1, lines 2-4, our method uses conventional jittered grid sampling [Cook et al. 1984] to generate `n*shading_rate` shading rays as uniformly as possible in the image space and shades the hit points of these rays. Antialiasing rays (`n*m`) are generated in the same way, as shown in line 6.

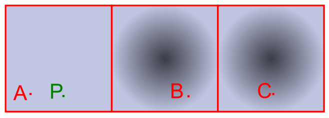
Lines 3 and 7 in Listing 1 use path tracing to trace the generated rays. A key difference between the path tracing here and traditional path tracing is that we limit our path tracer to Whitted-style [Whitted 1980] and highly glossy effects by pruning the ray path. Listing 2 is the pseudocode of our path tracer. Note that it returns an attenuation color and a hit point for each ray path, instead of just a color value as in traditional path tracing. A ray path's final color is obtained by executing a programmable shader on the hit point and multiplying the shading result with the attenuation color. Shading reuse is only applied to the shading results, and attenuation colors are always evaluated at supersampled resolution.

Listing 2 Pseudocode of our path tracing routine

```
function pathTrace(color, ray)
    hitpoint = rayTrace(ray)
    effect = randomEffect(hitpoint.shader)
    if effect.isWhittedOrGlossy():
        //Only Whitted/glossy bounces are traced
        ray_out = randomRay(hitpoint, effect)
        color *= brdf(hitpoint, ray, ray_out)
        return pathTrace(color, ray_out)
    else
        //In addition to an attenuation color, a
        //hitpoint is also returned for shading.
        return (color, hitpoint)
```



(a) 3D space



(b) Parametric space

Figure 2: *A Z-shaped primitive with 3 micropolygons. Points A, B, C have reusable shading values. We need to find the correct shading point (namely A) for point P and reuse A's shading value.*

2.2 Nearest Neighbor Based Shading Reuse

Our method makes use of an object-space nearest neighbor search algorithm to find the most accurate reusable shading value for every antialiasing sample. The algorithm is carefully designed to exploit micropolygon properties to facilitate an accurate and efficient nearest neighbor search.

Note that micropolygons are typically generated by dicing high-level parametric primitives like a subdivision surface. In a micropolygon based ray tracer, any ray hit point can be uniquely represented by a tuple (i, u, v), where i is an integer primitive ID and (u, v) is the hit point's parametric-space coordinate within the primitive. This allows us to perform the nearest neighbor search within each individual primitive's parametric space. Our algorithm thus constructs a 2D kd-tree for the shading samples in each primitive as the search acceleration structure.

Our parametric-space searching algorithm has several advantages. First, by limiting shading reuse candidates to the same primitive of the antialiasing sample, the risk of reusing shading from a different object or a different shader is trivially eliminated. Second, confining the nearest neighbor search within the same primitive can reduce the number of search candidates by orders of magnitude, resulting in a considerable performance gain. Finally, our parametric-space search algorithm remains robust in handling displacement mapped or highly curved primitives. For example, the scene shown in Fig. 5 contains a displacement mapped object.

Fig. 2 illustrates the robustness of our parametric-space search. Consider an extreme example of a Z-shaped primitive with three micropolygons (Fig. 2(a)). Only three shading rays hit the primitive at points A, B, C correspondingly. We need to find the nearest shading point for point P. Due to ambient occlusion, the shading values at B and C are significantly darker than A and should not be reused for P. However, spatially both B and C are closer to P than A. While B can be excluded by considering normal directions, C and A have an identical normal. It is thus difficult, if not impossible, to exclude C using only geometric information. On the other hand, in the parametric space the nearest neighbor search is able to find A for P correctly, as illustrated in Fig. 2(b).

Our nearest neighbor search ignores view direction. In our pipeline, highly view dependent effects like specular highlights are typically computed during the path tracing process described in Section 2.1, which is not a part of shading reuse. In such case, shading reuse takes effect by reusing the diffuse component of surfaces *reflected* by the highlights, instead of the highlights themselves.

2.3 Handling Nearest Neighbor Failures

Note that it is possible for the shading rays to entirely miss some small or thin primitives in the scene. For antialiasing samples on such primitives, the subsequent nearest neighbor search would fail to find any reusable shading value. As shown in Listing 1, lines 11-20, we collect such samples into a dedicated list and reshade them in a later pass. The hash-based technique in decoupled sampling [Ragan-Kelley et al. 2011] is employed to reuse shading within the reshaded samples. Here shading is reused and only reused among samples that are from the same primitive and ultimately contribute to the same pixel.

A direct consequence is that our approach falls back to decoupled shading for subpixel primitives such as furs and particles. While this fall back results in over-shading, it effectively reduces artifacts. We plan to investigate dedicated algorithms for rendering such primitives and combine the results into either shading samples or visibility samples.

2.4 Scene Layer Separation

Prior to entering the rendering loop in Listing 1, the scene is separated into several layers based on the first ray bounce types. Specifically, a layer is rendered for each type of ray bounce, including but not limited to direct absorption, reflection and refraction. For example, Fig. 3(a-d) illustrates a scene rendered in three layers. Our entire rendering loop is executed once for each layer, confining shading reuse within each individual bounce type.

Scene layer separation based on bounce types serves two purposes. First, it reduces path tracing noise. During the first bounce, the separation eliminates the stochastic decision on which ray type to follow. Therefore, all types of first bounce ray paths are combined using precisely evaluated weights instead of weights implicitly approximated through Monte Carlo integration. This eliminates the noisy artifacts related to ray type combination. Second, it alleviates the risk of reusing the shading value from a shading sample with a different texture filter size. Note that the same primitive may be visible in multiple layers. Each layer may have a different degree of magnification and distortion, resulting in different texture filter sizes. This is also true if there are multiple ray paths to the same point within the same layer, but we find it usually reduces the problem in practice. In some cases, more layers may need to be added to further reduce spurious shading reuse. Without the layer separation, the parametric-space nearest neighbor of an antialiasing sample may come from an entirely different layer with a different texture filter size, resulting in blurry or aliasing artifacts. Fig. 3(e) and (f) compare the rendering results with and without layer separation.

When a primitive is visible to two or more second bounce ray paths, blurry or aliasing artifacts may still occur in our pipeline. Fig. 4 illustrates such a situation. Our method produces Fig. 4(b), which contains a few black dots produced by reusing shading values evaluated from excessively large texture filters in another ray path.

Our current layer separation mechanism aims to alleviate the problem by partially separating ray paths prior to shading reuse. Separating first bounce effects eliminates erroneous shading reuses between directly visible objects and their immediate reflections/refractions, which is the most commonly occurring case in practice. Straightforward generalization to multiple ray bounces

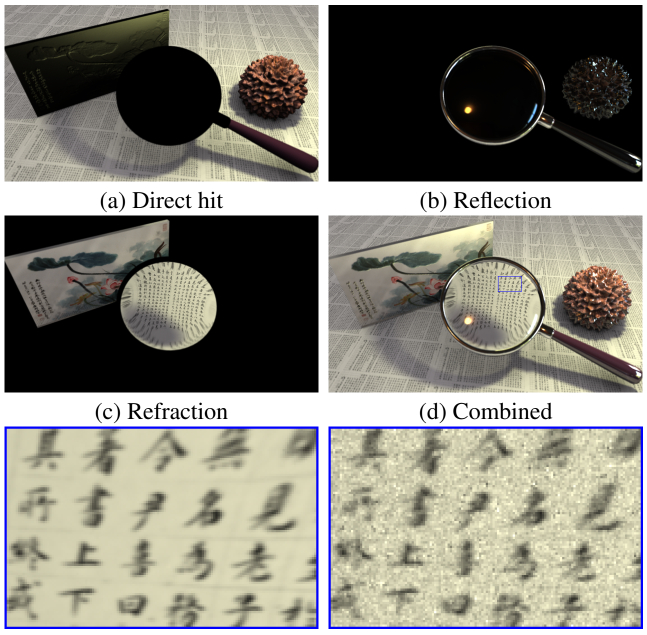

(a) Direct hit (b) Reflection (c) Refraction (d) Combined (e) With separation (f) Without separation

Figure 3: *Scene layer separation based on first ray bounce types. The scene in (d) is rendered as the three layers shown in (a), (b), (c) respectively. (d) is generated by adding up (a), (b), and (c). (e) and (f) are zoom-ins of the marked region in (d).*

suffers from combinatorial explosion, but the ray paths may be pruned using attenuation colors to yield a reasonable bound. This is a potential direction for future investigation.

With extensive experiments, we found that the problem can be fixed to produce Fig. 4(c) in a more robust manner by using in nearest neighbor searches a non-Euclidean distance metric $\sqrt{|\mathbf{p}_a - \mathbf{p}_b|^2 + c|\log_2(r_a/r_b)|^2\,(r_a^2 + r_b^2)}$. Here a and b are two sample points, $\mathbf{p}_a$ is the parametric space coordinate vector of a, r_a is the average length of parametric coordinate derivatives at a, and c is a constant that requires manual tweaking. On the other hand, the non-linear distance metric increased the nearest neighbor query cost significantly, causing a 3.2$\times$ slowdown of end-to-end render time. Therefore, considerable effort is still required to fit this metric for practical use.

3 GPU Implementation

We developed a highly parallel implementation of our shading reuse method on current GPUs. In the following, we discuss several non-trivial GPU implementation details that are not directly related to our core pipeline design. Listing 3 illustrates the pseudocode of the implementation.

3.1 Memory Management and Scalability

One difference from past shading reuse techniques is that our method requires storing all shading samples. To achieve scalability with respect to image size, a top level rendering loop is wrapped around Listing 3 to first splits the image into buckets such that the shading reuse data structure approximately consumes half of the available GPU memory. Listing 3 is then executed on each bucket sequentially, which partitions antialiasing samples in each bucket into fixed-size batches and processes the batches one by one. Both the bucket size and batch size are computed by dividing one half of the available GPU memory size by the corresponding maximal

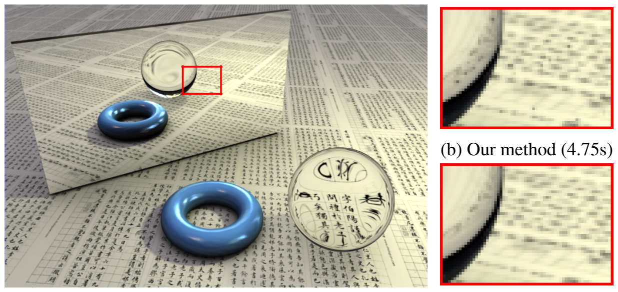

(a) Complete scene (new metric) (b) Our method (4.75s) (c) New metric (15.3s)

Figure 4: *A scene illustrating shading reuse errors. The mirror reflects both the ground and the glass ball. Shading may be erroneously reused between directly reflected hit points and points reached through the glass ball, which results in a few black spots in (b). The image is rendered at 720×480 with 13×13 antialiasing. The torus on the ground has a rough reflection component, which is handled correctly in our approach, as expected.*

per-sample memory consumption.

As the antialiasing samples in each batch are discarded once the batch finishes processing, it is no longer possible to simply get the final image by filtering all samples. Therefore, we implement filtering using forward splatting. Specifically, for each antialiasing sample we splat a filter-weighted disk with the sample's color onto the final image using additive blending. This allows samples to be generated in arbitrary order, which is also useful for failure sample handling (see Section 3.3).

3.2 En-masse KD-Tree Construction

The nearest neighbor search described in Section 2.2 requires constructing a 2D kd-tree for each individual primitive. The size of each kd-tree varies from one node to millions of nodes, and hundreds to millions of kd-trees have to be constructed simultaneously. To utilize the GPU efficiently, the construction algorithm has to be parallelized both within and across kd-trees. Fortunately, state-of-art GPU kd-tree construction algorithms (e.g., [Zhou et al. 2008]) only rely on nodes and points as parallelization units. We make two extensions to Zhou et al.'s method to allow multiple kd-trees to be built simultaneously in parallel. First, we replace their initial node with a list of root nodes for all our kd-trees. The root node list is generated on the GPU from the list of primitives that are hit by at least one shading ray. Second, we replace all their scan, reduce and sort parallel primitives with the corresponding segmented version to isolate operations in individual trees. Specifically, we replace reductions with segmented reductions, scans with segmented scans, and for sorts we modify the comparison functions to compare the tree ID before comparing the original sort key.

3.3 Failure Sample Handling

For efficient GPU utilization, failure samples are collected across antialiasing batches and processed in separate batches. All samples in a failure sample batch are discarded once the patch finishes processing.

The failure sample handling algorithm in Section 2.3 is parallelized within each individual failure sample batch. Specifically, instead of sequentially enumerating the samples while maintaining a hash table, we first sort all samples by their respective hash values. Then each sample's hash is compared with its previous sample. If the hashes are different, the sample is shaded. Otherwise, the previous sample's shading value is reused in a subsequent parallel loop. This is illustrated in Listing 3, lines 23-37.

3.4 Ray Tracing Implementation

Our shading reuse method has been integrated into an out-of-core GPU ray tracing system. The reason for using an out-of-core ray tracer is to bound the peak memory consumption during ray tracing and simplify the bucket/batch size computation in Section 3.1. Fig. 1 and Fig. 7 are two example scenes that require significant out-of-core swapping to handle.

The view-dependent dicing scheme in Reyes does not work well with the extensive ray tracing in our system. Our micropolygon dicing rate is calculated based on the world-space edge length rather than the projected edge length as in Reyes. The user specifies a target edge length for each object and the dicer generates micropolygons approximately of the target world-space edge length.

We trace ray differentials to provide basic derivatives in the form of d/dx, d/dy, where x and y are the image space coordinates. Tangent directions are interpolated from precomputed values on polygon/patch vertices. Currently we do not support systematic computation of arbitrary derivatives. A possible solution is to make all shader functions compute the derivatives of their output, and optimize out unnecessary ones in the shader compiler.

4 Experimental Results

We have implemented the proposed method in a micropolygon ray tracing system and evaluated its quality and performance on a variety of scenes. All performance-wise significant steps in our method run on the GPU and all data are measured on an Intel Core i5 2.67GHz CPU (4GB) with an NVIDIA GeForce 470 GPU (1280MB). All scenes are rendered with ambient occlusion and soft shadows, which are processed as a part of shading and reused. Our shading density is set to 2 shading evaluations per pixel per layer. The actual shading rate is smaller than this number because pixels that do not have a certain type of first bounce effect are not shaded in the corresponding layer.

Table 1 summarizes the performance statistics of our method on all test scenes. As illustrated, our method stays at a controlled and small shading evaluation density per pixel. In all scenes except for Fig. 7, only a small fraction of samples fail the nearest neighbor search and are reshaded. Our shading reuse may consume a considerable portion of the render time, especially when significant reflection/refraction effects are rendered at a high antialiasing rate. However, such cost is still small compared to the saved shading cost, as demonstrated in the car scene comparison.

A side benefit of our method is that micropolygons do not have to be diced as densely as in Reyes because the micropolygon size is no longer correlated with the shading rate. Specifically, our renderer does not dice non-displaced polygons at all. In addition, instanced primitives only have to be diced once and the micropolygons can be reused for all instances. For example, in Fig. 7, significantly less micropolygons were diced than the total number of primitives.

Fig. 5 is a test scene illustrating a variety of effects that our shading reuse method can handle. The main effects in this scene include defocus, a magnifying glass, a rough glass panel, a displacement mapped object and a high frequency text background. In the following we will discuss how our method compares to existing shading reuse methods in handling these effects.

Fig. 5(b-e) compares the reference solution, our method, and the decoupled sampling approach [Ragan-Kelley et al. 2011]. Our result (c) is visually identical to the reference result (b), which is produced by shading all anti-aliasing samples, i.e., without any shading reuse. Fig. 5(d) is rendered with a straightforward extension of Ragan-Kelley et al.'s method for ray tracing. Shading is reused across samples which belong to the same primitive and have the same hash value under their standard decoupling mapping, which computes the integer hash value of a sample by projecting it to the image plane while disregarding defocus. Since this mapping disregards the magnification and distortion caused by ray tracing, it resulted in blocky artifacts when rendering magnified text. A straightforward workaround to this problem is to modify the hash function to map a sample to the image pixel it ultimately contributes to instead of using a simple projection. This takes ray tracing into consideration and solves the problem as illustrated in the first row of Fig. 5(e). However, this approach relies on a continuous object-to-image space mapping and fails to handle stochastic ray paths that span multiple pixels like the rough glass shown in the second row of Fig. 5(e).

Note that the artifacts in Fig. 5(d,e) only manifest when decoupled shading is used on a multiple-pixel primitive. In contrast, our failure sample handling in Section 2.3 only falls back to decoupled shading for subpixel primitives or subpixel portions of larger primitives. Therefore, our approach is not affected by such artifacts even when there are a significant amount of failure samples like in Fig. 7.

Listing 3 Pseudocode of our GPU implementation. The keyword `forall` indicates a parallel loop.

```
final_image = new Image
for each layer in first_bounce_layers
    //Generate and shade seeding rays
    shading_rays = generateShadingRays(n*shading_rate)
    shading_samples = pathTraceAll(shading_rays, layer)
    shade(shading_samples)
    shading_samples.buildKdTrees()
    //Trace antialiasing rays and reuse shading
    failed = new List
    for each batch (p,p+batch_size) in (0,n*m-1)
        aa_rays = generateAntialiasRays(p,p+batch_size)
        aa_samples = pathTraceAll(aa_rays)
        forall s in aa_samples
            s.shading = shading_samples.findNearest(s)
            if s.shading == NOT_FOUND:
                //".add" is implemented using compaction
                failed.add(s)
                s.shading = 0
        //Shade failure samples in independent batches
        if failed.size()>GPU_UTILIZATION_THRESHOLD:
            //Parallelized failure sample handling
            //Sort failed samples by hash
            failed.sortByHash()
            //Shade the "head" samples
            forall i in (0, failed.size()-1)
                s = failed[i]
                if canReuseBetween(s, failed[i-1]):
                    pass
                else
                    shade(s)
            //Reuse shading at other samples
            forall i in (0, failed.size()-1)
                s = failed[i]
                if canReuseBetween(s, failed[i-1]):
                    s.shading = failed[i-1].shading
            final_image.splatSamples(failed)
            failed.clear()
        //Splat shaded samples to the final image
        final_image.splatSamples(aa_samples)
        delete aa_samples
        delete aa_rays
```

Scene	Resolution	Sampling	#prims	#mpoly	#shade	#reshade	$\frac{\#shade}{\#pixel}$	T_{reuse}	$T_{shading}$	T_{total}
Fig. 1	1920×1080	11×11	1.56M	3.90M	6.89M	963K	3.48	4.79s	125s	289s
Fig. 5	1920×1080	11×11	4.4K	543K	6.44M	16.6K	3.26	6.17s	20.8s	57.9s
Fig. 6	1280×720	23×23	207K	208K	2.76M	75.25K	3.14	6.52s	8.85s	52.6s
Fig. 7	1920×720	8×8	296M	3.25M	9.78M	6.28M	7.41	1.67s	337s	409s

Table 1: *Test data statistics. #prims is the total number of geometric primitives. #mpoly is the total number of micropolygons generated during rendering. #shade is the total number of shading evaluations performed. #reshade is the total number of reshading evaluations for samples that failed the nearest neighbor search. Note that #reshade is included in #shade.* T_{reuse} *is the total time cost of shading reuse operations, which primarily consists of kd-tree construction, nearest neighbor search and reshading sample collection. The cost of tracing shading rays is not included in* T_{reuse}*, since the rays are a subset of antialiasing rays and have to be traced regardless of shading reuse.* $T_{shading}$ *is the total shading time.* T_{total} *is the total rendering time.*

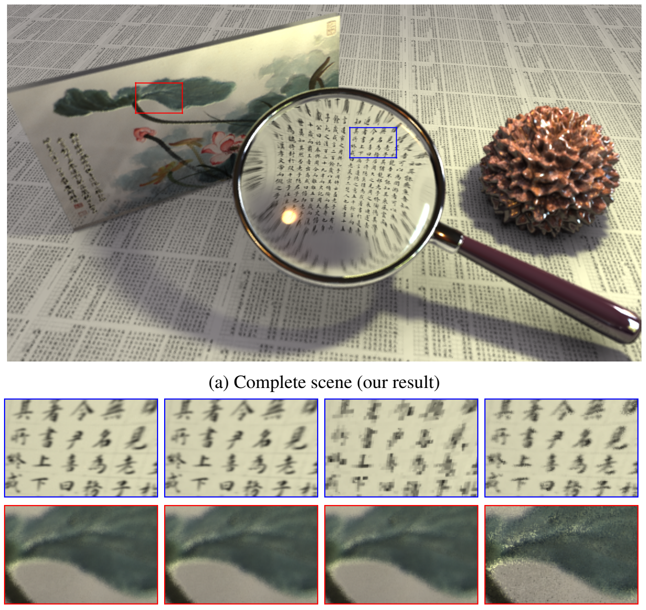


Figure 5: *A test scene rendered with our method (a), and zoom-ins of the marked regions for the reference result (b), our result (c), and two results generated by decoupled sampling based methods (d,e). DS stands for decoupled sampling. The image is rendered at* 1920×1080*,* 11×11 *supersampling with defocus effect and 2 bounces of reflection/refraction. The spiky orange object is modeled using a displacement mapped sphere.*

Compared to traditional Reyes shading which considers reflection/refraction effects as a part of shading, our main advantage is that we are able to handle the combined effects of refraction and defocus correctly. In Fig. 5(a), the camera is focused on the image refracted by the magnifying glass. Therefore, the refracted image appears clear even though the magnifying glass itself is blurred due to defocus. This effect is impossible to produce using micropolygon vertex shading as the refraction shader's output would always get blurred when resolving visibility.

Compared to the lazy shading framework proposed by Burns et al. [2010], our method is able to achieve shading reuse without relying on explicit shading density estimation or relying on the Reyes-style irregular splitting. For ray tracing effects such as the magnifying glass in Fig. 5(a), it is difficult, if not impossible, to estimate the required shading density on geometric primitives. If their shading density estimation were used here as is, it would result in blocky artifacts similar to Fig. 5(d). Finally, even if an accurate density estimation is provided, their method still requires excessively deep and irregular Reyes splits to convert the density estimation into grids with uniform shading rates.

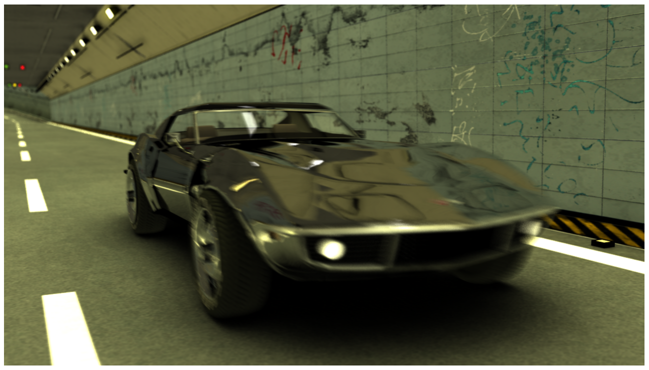


Figure 6: *The car scene from Hou et al. [2010] rendered by our method. The image is rendered at* 1280×720 *resolution with* 23×23 *supersampling, the same as in the original paper.*

We also compare our performance data with the GPU-based micropolygon ray tracer [Hou et al. 2010], which uses a shading reuse strategy similar to Christensen et al. [2006]. Fig. 6 shows our rendering of a car scene provided by Hou et al. [2010]. Their algorithm took 185 seconds to finish rendering on our hardware. The reflection effect on the car has to be handled as a part of shading. To get antialiased reflection, the car has to be shaded at a shading rate of 0.1 and about 49M micropolygons were shaded in total. Our method is able to produce the same image quality in 53 seconds. Compared to their algorithm, our method handles ray tracing elegantly and does not require their high shading rate for reflection antialiasing. Only 2.76M hit points were shaded in this scene, resulting in over $17\times$ less shading evaluations and $3.5\times$ higher overall rendering performance.

Fig. 7 is an extreme example with a tremendous number of primitives. The large amount of small leaves result in a high failure rate of our nearest neighbor search, where small primitives missed by shading rays are seen by visibility rays, which leads to a considerable amount of reshading. The fall-back image-space shading reuse ensures our method does not degenerate to per-sample shading. The total number of shading evaluations is still less than 12% of antialiasing samples.

5 Conclusion and Future Work

We have presented a novel shading reuse method for efficient micropolygon ray tracing. By handling ray tracing effects such as reflection and refraction with antialiasing, we are able to greatly reduce shading evaluations and achieve significant performance gains over the state-of-the-art micropolygon ray tracing algorithm. We control the shading evaluation density through image space sampling and use nearest neighbor search to maintain the accuracy of shading reuse. Fine features missed by our initial shading are handled using an image-space fall-back to ensure visual correctness.

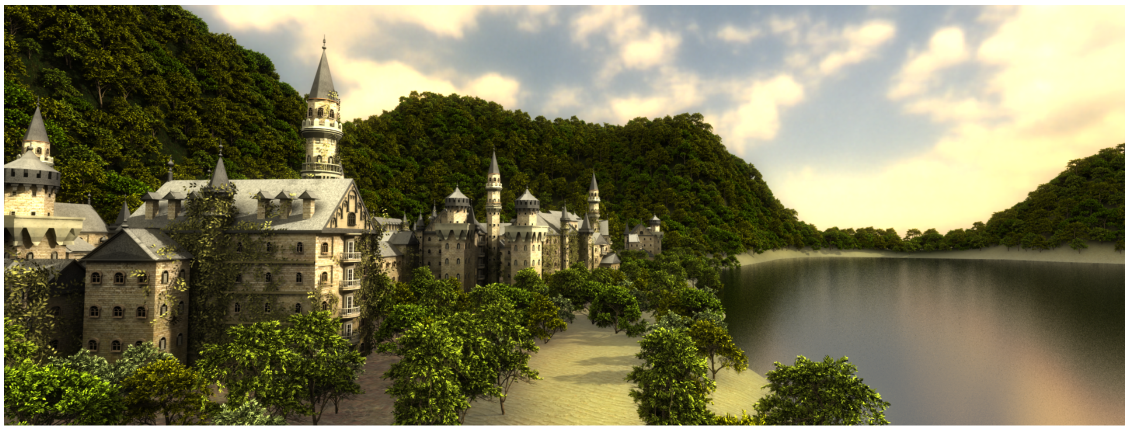


Figure 7: *A large procedurally generated scene. It consists of 296M primitives instanced from 888K base primitives. The scene is rendered at* 1920×720 *with* 8×8 *supersampling and one bounce of glossy reflection. Some mild HDR glow is applied for visual effect.*

Our shading reuse method still has two limitations. First, currently we do not support per object shading rates. The reason is that we generate shading rays in the image space to obtain hit points for shading evaluation. All objects thus have the same shading rate. A potential workaround is to trace additional shading rays from pixels that overlap with objects with higher shading rates. Second, during the nearest neighbor search we do not take into account motion blur time, view ray direction, and texture filter size. Our preliminary experiments have shown that simply incorporating them as extra dimensions degrades the correlation between the distance metric and shading continuity, which results in suboptimal shading reuse. It would be an interesting future work direction to design a high dimensional distance metric suitable for shading reuse.

Although our shading reuse method is designed for micropolygon ray tracing, the only dependency on a micropolygon pipeline is the primitive parametric space the pipeline provides. It is possible to generalize our method to general ray tracing pipelines if an alternative distance metric with similar properties can be designed.

Acknowledgements

We would like to thank BlueArc Animation Studios for modeling the battle scene (Fig. 1) and Steve Lin for proofreading the paper. We are grateful to the anonymous reviewers for their valuable comments. The project is partially supported by the NSF of China (No. 60825201).

References

BURNS, C. A., FATAHALIAN, K., AND MARK, W. R. 2010. A lazy object-space shading architecture with decoupled sampling. In *Proceedings of HPG 2010*, 19–28.

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

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

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

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 2009*, 59–68.

FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing shading on GPUs using quad-fragment merging. *ACM Trans. Graph. 29*, 4 (July), 67:1–67:8.

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 (December), 150:1–150:10.

HOU, Q., QIN, H., LI, W., GUO, B., AND ZHOU, K. 2010. Micropolygon ray tracing with defocus and motion blur. *ACM Trans. Graph. 29*, 4 (July), 64:1–64:10.

IGEHY, H. 1999. Tracing ray differentials. In *Proceedings of ACM SIGGRAPH ’99*, 179–186.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: a general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July), 66:1–66:12.

PATNEY, A., AND OWENS, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Trans. Graph. 27*, 5 (December), 143:1–143:8.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled sampling for graphics pipelines. *ACM Trans. Graph. 30*, 3 (May), 17:1–17:17.

STOLL, G., MARK, W. R., DJEU, P., WANG, R., AND ELHASSAN, I. 2006. Razor: An architecture for dynamic multiresolution ray tracing. Tech. rep., The University of Texas at Austin.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. GPU-accelerated high-quality hidden surface removal. In *Proceedings of Graphics Hardware 2005*, 7–14.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6 (June), 343–349.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5 (December), 126:1–126:11.

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. RenderAnts: interactive Reyes rendering on GPUs. *ACM Trans. Graph. 28*, 5 (December), 155:1–155:11.