Pacific Graphics 2008
T. Igarashi, N. Max, and F. Sillion
(Guest Editors)

Volume 27 (2008), Number 7

Interactive Glossy Reflections using GPU-based Ray Tracing with Adaptive LOD

Xuan Yu[1] and Rui Wang[2] and Jingyi Yu[1]

[1]University of Delaware
[2]University of Massachusetts Amherst

Abstract

We present an interactive GPU-based algorithm for accurately rendering high-quality, dynamic glossy reflection effects from both HDR environment maps and local scene objects. Our method uses hardware rasterization to produce primary pixels, and GPU-based BRDF importance sampling [CK07] to quickly generate reflected rays. We utilize a fast GPU ray tracer proposed by Carr et al. [CHCH06] to compute reflection hits. Our main contribution is an adaptive level-of-detail (LOD) control algorithm that greatly improves ray tracing performance during reflection shading. Specifically, we use the solid angle represented by each reflected ray to adaptively pick the level of termination in the BVH traversal step during ray tracing. This leads to $2 \sim 3x$ *speedup over an unmodified implementation of [CHCH06]. Based on the same solid angle measure, we derive a texture filtering formula to reduce reflection aliasing artifacts, taking advantage of hardware MIP mapping. This extends the filtering algorithm presented in [CK07] from environment mapping to local scene reflection. Using our algorithm, we demonstrate interactive rendering rates for several scenes featuring dynamic lighting and material changes, spatially varying BRDF parameters, and rigid-body object movement.*

1. Introduction

Accurate simulation of complex surface reflections plays a central role in creating photorealistic images. In computer graphics, reflection of lights from surfaces is typically modeled by the Bidirectional Reflectance Distribution Function (BRDF) – a 4D function that uniquely captures the appearance properties of each different material. Traditionally in real-time global illumination systems, diffuse or ideal specular (mirror) materials are assumed because these are two special cases of the BRDF that simplify the computation of reflections. General glossy BRDFs are much more expensive to simulate, as they require integrating many reflected rays over the hemisphere of incoming light. The importance of each reflected ray is determined by the BRDF, and is further dependent on the viewing angle, making it very difficult to cache and reuse previously computed samples.

Assuming distant lighting environment, researchers have studied environment mapping techniques that render accurate glossy reflections in real-time. These include prefiltered environment maps [CON99, HS99, KVHS00], spherical harmonics or wavelet basis projection [RH02, WNLH06], and GPU-based BRDF importance sampling [CK07]. The fundamental limitation of environment mapping is that global illumination effects such as cast-shadows and indirect lighting are ignored. Recently, Precomputed Radiance Transfer (PRT) [SKS02, IDYN07, SZC*07, AUW07] has been shown to enable real-time lighting effects that include local scene reflection, global and self-shadowing, and dynamic BRDFs. These methods typically require fixing scene models so that they can make use of precomputed data to amortize shading costs on the fly.

Our goal in this paper is to design a fast GPU-based algorithm for simulating dynamic glossy reflection effects in real-time. To allow for general BRDFs and dynamic scenes, we make use of a GPU-based ray tracer to shade reflection rays directly on the fly, without requiring hefty precomputed data. Recent progress in GPU-based ray tracing has made it possible to trace tens of millions of rays in one second [PBMH02, HSHH07], and there has been an increasing interest in utilizing such algorithms for computing distribution ray tracing effects such as glossy reflections. The major

challenge with current methods is that they rely on coherent bundled rays to exploit the GPU's massive parallel computation power; however, the initial coherence is likely to be lost upon glossy reflection, where the rays diverge from each other as they travel away from the origin.

As our main contributions, we present an algorithm that combines adaptive LOD control with hardware texture filtering to avoid shooting excessive rays, reducing the overall computation cost. We first use hardware rasterization to produce primary pixels, and GPU-based BRDF importance sampling [CK07] to generate reflected rays. We then utilize a fast GPU ray tracer presented by Carr et al. [CHCH06] to compute reflection hits. During ray tracing, we make use of the solid angle represented by each reflected ray to adaptively pick the level of termination in the BVH traversal hierarchy. This approach in general leads to $2 \sim 3$x speedup over an unmodified implementation of [CHCH06]. Based on the same solid angle measure, we also derive a texture filtering formula to efficiently reduce reflection aliasing artifacts, taking advantage of standard hardware MIP mapping. This method can be seen as an extension of [CK07] from environment mapping to local scene reflections. Using our algorithm, we demonstrate several scene models featuring dynamic lighting and material changes, spatially varying BRDF parameters, and rigid-body object movement.

Currently, our particular ray tracing implementation limits us to models represented by geometry images. It is possible, however, to extend the work to more general representations, which requires providing a reasonable estimate of the solid angle and an approximate intersection geometry at each node in the ray tracing hierarchy (acceleration structure). We would like to explore these options in future work.

2. Related Work

Real-time Glossy Reflections In recent years, image-based lighting, which represents natural illumination by distant HDR environment maps [DM97], has been widely used for generating convincing shading effects. Under the assumption of distant lighting, the expensive cost of computing glossy reflections can be amortized by prefiltering environment maps [CON99, HS99, KVHS00], or using projection basis such as spherical harmonics or wavelets [RH02, WNLH06]. These methods require preprocessing of BRDFs and thus are not suitable for arbitrary BRDFs dynamically applied in real-time. In addition, they ignore global illumination effects such as cast-shadows and interreflections.

Recently Colbert et al. [CK07] presented GPU-based BRDF importance sampling to compute real-time, dynamic glossy reflections with minimal requirement imposed on the BRDF. They make use of mipmap filtering to reduce aliasing artifacts caused by low sample count. Similar to environment mapping, this method only works for local illumination effects, ignoring shadows and interreflections. Our approach

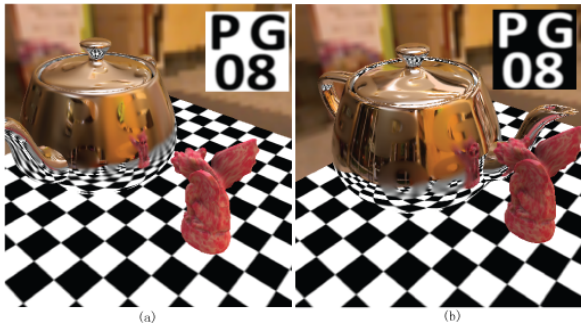

Figure 1: *A glossy teapot rendered with spatially varying BRDFs: the intensity of the "PG 08" texture is used to modulate the specularity of a Phong BRDF. This example runs at 5 fps with 512×512 resolution on NVidia 8800 GTX.*

extends their work to global effects by utilizing a fast GPU ray tracer. We improve the ray tracing performance for reflected rays by using adaptive LOD control; in addition, we extend their mipmap filtering algorithm from distant environment maps to textured, local scene objects.

Precomputed Radiance Transfer (PRT) [SKS02, IDYN07, SZC*07, AUW07] has been shown to enable real-time realistic lighting effects that include local scene reflections, global and self-shadowing, and dynamic BRDFs. These methods rely on a large amount of precomputed data sampled from fixed scene models, disabling dynamic object movement or deformation. In general, he precomputation requirement makes them inflexible at handling arbitrary and per-pixel shading effects, such as BRDFs with broad frequency scales, bump mapped normals, and spatially varying BRDF parameters. By using real-time ray tracing, our approach permits great flexibility in handling arbitrary reflections. Furthermore, it eliminates the need for heavy precomputation: we require only a small amount of precomputed data (i.e. the BVH structure for ray tracing), which can be quickly updated on the fly to allow for deformable scenes [CHCH06].

Real-time Ray Tracing By carefully exploiting the coherence of rays, real-time ray tracing has been shown possible on commodity CPUs [WBWS01, RSH05, DHW*]. Recently, ray tracing on programmable GPUs has also received significant attention. Purcell et al. [PBMH02] designed a GPU ray tracer using a uniform grid acceleration structure. Foley et al. [FS05] proposed an improved algorithm using kd-tree, which is later extended by Horn et al. [HSHH07] with several major enhancements. Carr et al. [CHCH06] use a bounding volume hierarchy (BVH) and a fixed-order traversal algorithm to create a ray tracer suitable for geometry image models [GGH02]. This method allows the GPU to efficiently stream through the BVH nodes without maintaining a stack, a major challenge in GPU ray tracing. In addition, the structure of the BVH is simple enough to be updated online, permitting dynamic deformable objects.

It is possible nowadays to trace tens of millions of rays per

second for primary and shadow rays, thanks to the excellent coherence of these rays, which can be successfully exploited by the massive parallel computation power of modern GPUs. This kind of speed is sufficient for interactive applications that require only OpenGL style shading. For distribution ray tracing, however, the coherence of reflected rays is usually quite poor, as they diverge strongly from each other upon reflection. In addition, multiple rays must be evaluated to robustly estimate the integral of lighting with BRDFs. As a result, the ray tracing performance slows down significantly when glossy interreflections are enabled.

LODs for Ray Tracing Level-of-details (LODs) has been widely used for accelerating rendering with large polygon sets [LWC*02], especially in rasterization-based graphics. Its use in ray tracing, on the other hand, is relatively new. Christensen et al. [CLF*03] introduced an LOD approach that combines *ray differentials* [Ige99] with multiresolution caching for offline rendering. Their method supports 3 discrete resolution levels. Djeu et al. [DHW*] presented *Razor* – a ray tracing architecture that supports watertight multiresolution geometry using a continuous interpolation scheme and a dynamic kd-tree built on-demand. Due to their complexity, these methods are only suitable for CPU implementation. Yoon et al. [YLM06] introduced *R-LOD* – a simple and fast LOD representation that is designed to dramatically simplify massive models and improve ray tracing speed. Our approach is fundamentally similar to theirs. The difference is that we make use of the existing hierarchy of geometry image models [GGH02, SWG*03] to simplify the representation of LOD, making it suitable for GPU computing. This idea was previously explore by [HR06] and [JJ05] in rasterization based rendering. In addition, we focus on glossy reflections, and combine LOD with fast, hardware texture filtering to further reduce computation.

3. Algorithms and Implementation

3.1. Overview and Assumptions

According to the rendering equation [Kaj86], the reflected radiance from viewing direction ω_o at a surface point x due to incident lighting L is computed by:

$$B(x,\omega_o) = \int_\Omega L(x,\omega)\, f_r(\omega_i,\omega_o)\cos\theta\, d\omega_i \qquad (1)$$

where ω_i is the incident direction, θ is the incident angle, and f_r is the BRDF. The incident lighting includes illumination from both distant and near-field sources in the scene. For near-field sources, we can rewrite the equation by integrating over the surface area of unoccluded source patches:

$$B(x,\omega_o) = \int_A L(x_i)\, f_r(x_i \to x,\omega_o)\frac{\cos\theta_i\cos\theta_o}{|x_i-x|^2}\, dA(x_i) \qquad (2)$$

where x_i is a point on the patch being reflected (i.e. an indirect light source). When the patch is far away or small enough, the BRDF and geometry factors can be treated as

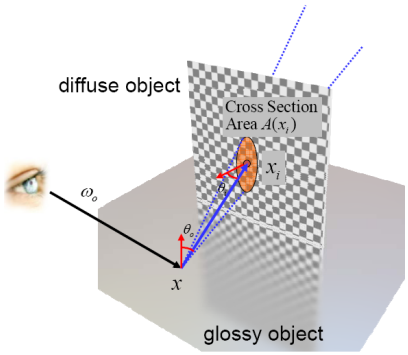

Figure 2: *An intersection point and the ray's cross section.*

constant within the patch. In that case, the equation can be approximated by (see Figure 2):

$$B(x,\omega_o) \approx \bar{L}(x_i)\, f_r(x_i \to x,\omega_o)\frac{\cos\theta_i\cos\theta_o}{|x_i-x|^2}A(x_i) \qquad (3)$$

where x_i is a single sample point (in our case, the ray tracing intersection point), $\bar{L}$ is the average (blurred) radiance across the patch, and A is the total area of the patch (in our case, the cross section area at the point of intersection).

Assumptions We make two assumptions to approximate the rendering equation and reduce the required computation. First, we classify a scene model as either a diffuse or a glossy object, and we compute only one bounce of reflection from the diffuse objects (plus environment) to the glossy objects. Transfer paths that start from glossy objects are ignored due to the difficulty in storing a large amount of view-dependent information. Second, we assume that the direct lighting on diffuse objects can be computed quickly at run-time, by using a shadow mapper or prefiltered environment maps. This assumption allows us to quickly obtain the diffuse radiance of the reflected objects without having to trace rays further.

Our rendering algorithm consists of three key components: BRDF importance sampling, LOD-based ray tracing, and image space texture filtering. We have implemented the entire pipeline on the GPU. In the following we describe each component in detail.

3.2. BRDF Importance Sampling

To simulate Eq. 2, we use BRDF importance sampling, where samples are drawn from a normalized distribution function that is closely correlated to the BRDF itself. Most commonly used BRDFs have efficient importance sampling functions; in particular, diffuse and Phong BRDFs have analytic integrals and therefore can be perfectly sampled.

For simplicity, we use the Phong model throughout the paper. In this case, the importance sampling function p is simply the normalized Phong BRDF:

$$p(\theta,\phi) = \frac{(n+1)}{2\cdot\pi}\cdot(\cos\theta)^n \qquad (4)$$

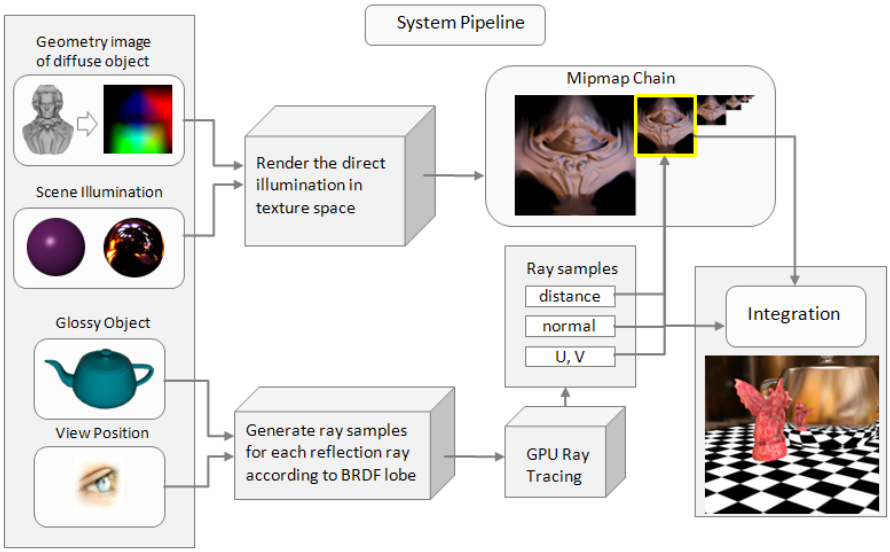

Figure 3: *A diagram showing the pipeline of our rendering algorithm.*

Here n is the Phong exponent parameter, θ and ϕ are the two spherical angles used to define a direction. This direction is described in the coordinate frame centered around the principle reflection direction $\vec{R} = \text{reflect}(\omega_i, \vec{N})$. The constant $\frac{(n+1)}{2\cdot\pi}$ is a normalization factor. Once a sample direction is chosen, a ray can then be constructed that originates from x and points toward the sampled direction (θ, ϕ). Although we choose the Phong model in our implementation, more complicated BRDFs can be incorporated as long as their importance sampling functions are known.

Solid Angle Similar to [CK07], we assign each sampled ray a solid angle representing its angular width:

$$\Omega_s = \frac{1}{N \cdot p(\theta, \phi)} \tag{5}$$

where N is the total number of samples. According to this definition, the solid angle of a ray is inversely proportional to the sampling density (importance). This is intuitively correct, as directions that are sampled more frequently should represent smaller solid angles. The expected value of Ω_s should be $\frac{4\pi}{N}$, which can be easily verified.

3.3. Adaptive LOD for Ray Tracing

We use a GPU ray tracer described in [CHCH06] to trace sampled reflection rays into the scene. This implementation requires the intersecting geometry to be represented by geometry images [GGH02]. The structure of a geometry image makes it very easy to build a Bounding Volume Hierarchy (BVH), which is used as an acceleration structure for ray tracing. Specifically, each node of the BVH stores an axis-aligned bounding box of the triangles belonging to that node, and the entire BVH can be easily built using recursive 4-to-1 reduction, similar to the construction of a Mipmap. A fixed-order traversal of the BVH is built ahead of time, permitting the GPU ray tracer to quickly stream through the structure online without using a stack.

To improve the ray tracing performance, we use a dynamic LOD control algorithm. Assume that each reflected ray has a 'cone' shape that starts at the ray origin (see Figure 2): whenever an intersection point is found, the radiance returned by the ray should be averaged (blurred) across a large area in object space. As the ray travels farther away, the cross section of the cone becomes bigger. In this case, precise intersection tests are no longer necessary, since the perceptual error caused by inaccurate intersection will be low-pass filtered and become less noticeable. Therefore, we can use progressively coarser geometry LODs to compute the intersection. In addition, the average radiance value around the intersection point can be efficiently estimated by using hardware texture filtering.

Our LOD-based ray tracing is implemented by using a simple algorithm to determine the level of termination when traversing rays through the BVH. As explained before, each node of the BVH stores a bounding box; as we traverse a ray through the BVH, we chec the solid angle subtended by the bounding box of the current node, and compare it with the solid angle represented by the ray. The solid angle of a bounding box can be approximated by:

$$\Omega_{BBox} = \pi \frac{|p_{max} - p_{min}|^2}{4\,|(p_{max} + p_{min})/2 - x|^2} \tag{6}$$

where p_{max} and p_{min} are the two corner points of the bounding box. We compare this value with Ω_s – the solid angle represented by the current ray. When $\Omega_{BBox} > \Omega_s$, the intersection test has to be more precise, therefore we keep

traversing the BVH without any change. On the other hand, if $\Omega_{BBox} \leq \Omega_s$, it indicates that the geometry to be intersected with is already smaller than the angular width of the ray. Since the radiance values within the solid angle have to be averaged, this suggests that a more precise intersection is not necessary. In this case, instead of further tracing the children nodes, we can stop at the current node, treating it as a leaf node, and using its approximate geometry directly to compute intersection. If an intersection is found, the average radiance of the ray will then be retrieved using texture filtering explained in the next step.

One drawback of this approach is that when the number of samples is insufficient, the solid angle represented by each ray will be quite large. In this case, many of the rays will be intersecting at very coarse levels of the LOD, resulting in noticeable aliasing artifacts. Overall this problem should be addressed by increasing the number of samples. However, we do note that due to importance sampling, rays with higher importance are associated with smaller solid angles, forcing them to intersect with finer LOD levels. Therefore this property helps to solve the problem partly by providing higher accuracy for important rays.

We found through experiments that Eq 5 often underestimates the solid angle of important rays, resulting in very small values, especially when the BRDF is very sharp. Therefore, instead of directly comparing Ω_{BBox} with Ω_s, we add a scaling factor α^2 to Ω_s, which provides a more flexible control on how the adaptive LOD criteria is applied. Hence in practice, we use the following comparison:

$$\Omega_{BBox} \leq \alpha^2 \cdot \Omega_s \tag{7}$$

Experiments suggest that an α value between $[1,10]$ produces good results. Adjusting this parameter provides a tradeoff between performance and intersection accuracy. Note that a very small α value is equivalent to disabling the adaptive LOD control, forcing the ray tracer to perform full intersection tests. When $\alpha = 0$, the algorithm falls back to the original implementation by [CHCH06].

3.4. Texture Filtering

When an intersection is found, we compute its cross section in object space, and use this information to guide texture filtering in order to estimate the average diffuse radiance within the cross section. In general the intersection cross section may have an arbitrary shape, making it very difficult to perform an accurate estimation. In practice, however, we can assume the cross section is locally flat, thus can be estimated from the solid angle of the ray:

$$A(x_i) \approx \frac{|x_i - x|^2 \cdot \Omega_s}{\cos\theta_i} \tag{8}$$

This is approximately equal to the cross section resulting from intersecting a ray cone with the tangent plane at the point x_i. See Figure 2 for an illustration.

With the estimated intersection area, we can then evaluate the appropriate Mipmap level for computing the average radiance. The general idea is that the area $A(x_i)$ covers a certain number of pixels in the texture space, therefore we would like to pick a Mipmap level that roughly corresponds to that number of pixels. To do so, we assume that the texture coordinates in the local neighborhood of x_i is uniform. This assumption is reasonable since the geometry image construction algorithm guarantees uniformity up to a maximum distortion factor. Next, we estimate the Mipmap level as:

$$l = \frac{1}{2}\log_2\left(\frac{A(x_i)}{A_{pixel}}\right) = \frac{1}{2}\left(\log_2 A(x_i) - \log_2 A_{pixel}\right) \tag{9}$$

where A_{pixel} is a measure called *area per pixel* – the object space area covered by one pixel. This is used to convert area from object space to texel space. It can be easily computed from the geometry image by estimating the area covered the two triangles spanning one pixel size in geometry image space. These values are precomputed and stored together with the geometry image.

Note that this LOD formula implicitly assumes that the texture scale is roughly equal in the u and v directions; in other words, the texture scale is isotropic. If this is not the case (i.e., the texture is stretched differently along u and v), we can separate the formula for each direction, and use anisotropic filtering to achieve more accurate results.

3.5. Implementation Details

Scene Models As mentioned before, we require that diffuse objects in our scenes be represented as geometry images. Glossy objects do not participate in intersection tests, therefore do not need to have geometry image representations. Note that self-reflections are not supported on glossy objects, although it would be easy to modify the algorithm so that a glossy object can reflect the diffuse portion of itself.

We generate the BVH and fixed traversal order offline for each geometry image model, by following the algorithm presented in [CHCH06]. In addition, we precompute A_{pixel} – the area per pixel measure for each texel, and store it in the alpha channel of the geometry image textures. Figure 3 shows a diagram of our rendering algorithm.

BVH for Ray Tracing The construction of the BVH for geometry images is very similar to constructing a Mipmap, which uses a simple 4-to-1 reduction algorithm in bottom-up fashion. At each node, the axis-aligned bounding box is aggregated from its four children node; and the two corners p_{min} and p_{max} of the bounding box are stored in a texture.

Following this step, we build a fixed-order traversal link as in [CHCH06]. Specifically, a hit map and miss map are constructed which indicate where the ray should go when it hits or misses the current node. When traversing a ray through the BVH, it is tested against the bounding box at each node.

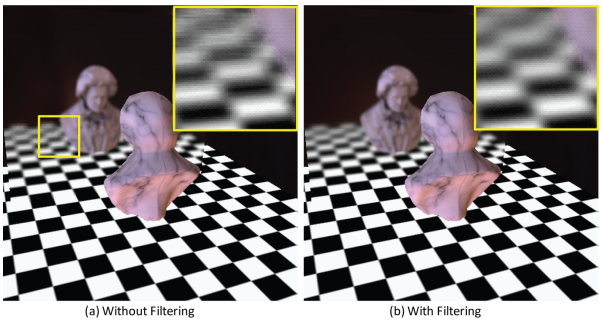

Figure 4: *Comparing the rendering results with and without texture filtering. Note the differences in aliasing artifacts.*

If the ray hits the volume, it will follow the traversal order to the next node in the sequence, until it reaches a leaf node, where the ray will be tested against the actual triangles. If, on the other hand, the ray misses the current node, then the next node to visit is indicated by the miss map.

Geometry Image LODs Geometry images impose natural LOD structures. At each node in the BVH, we simply take the reduced pixels to form an approximated geometry at that level. For example, if the original geometry image is at 512×512 resolution, then the finest level LOD has $512 \times 512 \times 2$ triangles, and the next level has $256 \times 256 \times 2$ triangles, by skipping every other pixel in the rows and columns of the geometry image. The level following that has $128 \times 128 \times 2$ triangles, and so on. Each level reduces the triangle count by 4. The approximate geometry is used for ray tracing whenever the LOD criteria is met.

Rendering Diffuse Objects The first step in our rendering algorithm is to shade diffuse objects. This can be done by shadow mapping for small light sources. As we focus more on environment lighting, it is actually non-trivial to compute accurate, shadowed direct illumination in real-time. To produce convincing shading effects, we make use of unshadowed, prefiltered environment maps such as [RH01], and combine them with ambient occlusion maps to produce global shadowing effects. Ideally we could also use our ray tracer to keep tracing rays upon secondary reflections, but this would significantly increase the computation costs. At run-time, we compute and store the radiance of diffuse objects together with their geometry image. This allows us to quickly obtain the direct lighting value from diffuse objects without the need for further ray tracing. Note that this is analogous to use caching schemes, such as irradiance cache, to reduce the computation cost of global illumination.

To compute the diffuse radiance, we take each geometry image with its ambient occlusion map as input to a fragment shader. We use the surface normal of each geometry image pixel to index into a prefiltered irradiance environment map and obtain an irradiance value. This value is then multiplied with the diffuse reflectance and ambient occlusion color, and the result is output to a target texture. The target texture will be bound to the GPU shader in the ray tracing pass for computing glossy reflections. We enable hardware built Mipmaps for this texture to achieve texturing filtering described in Section 3.4.

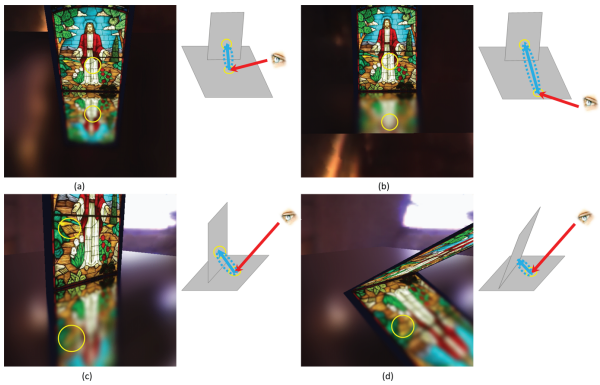

Figure 5: *A cathedral facade on top of a glossy mirror rendered at 25 fps with 160 reflection ray samples. (a) and (b) show how glossy reflections change with respect to the viewing angle. (c) and (d) show how how glossy reflections change as we rotate the facade towards the mirror.*

Ray Tracing We use hardware rasterization to generate primary pixels for all glossy objects, and bind a fragment shader to compute the final results. The first step in the shader is to generate sampled reflection directions. Similar to [CK07], we use a 2D Hamersley sequence, which has excellent low discrepancy properties. We then sample reflected directions according to Eq 4. Due to importance sampling, many directions will be distributed around the principle reflection direction $\vec{R}$.

With our adaptive LOD control enabled, the ray tracing step is implemented as follows: when a ray successfully intersects a bounding box, before moving on, we check the solid angle subtended by the bounding box against the ray's solid angle, using Eq 7. If the check fails, we keep traversing the BVH without any change. If the check succeeds, we will treat the current node as a leaf node and stop the traversal further down the node. Instead, we use the approximate geometry represented by the current node to perform an intersection test. The next node that the ray should traverse to will follow the miss link, independent of whether the ray intersects the approximate geometry or not. When the traversal is completed, the intersection point with the closest distance along the ray will be returned.

Mipmap Texture Filtering If the ray fails to intersect any scene geometry, we use the ray's direction to index into the environment map and return the radiance contribution. If, on the other hand, an intersection point is found, we use the Mipmap level selected by Eq 9 to index into the radiance texture of the diffuse objects and obtain an average radiance value. In Figure 4 we compare the rendering results with and without Mipmap filtering. As can be seen, enabling Mipmap

Figure 6: *A glossy teapot reflecting several diffuse objects rendered at 3.45 fps. Note the change in glossy reflections as we move the teapot around the scene.*

			Adjust Number of Samples			Adjust LOD Control (α)		
Scene	G.I. Size	Triangles	20	40	60	10	3	1
Facade	N/A	N/A	185.11 fps	92.55 fps	45.58 fps	N/A	N/A	N/A
Gargoyle + teapot	64x64	8K	11.29 fps	5.91 fps	3.76 fps	7.89 fps	6.4 fps	5.91 fps
Gargoyle + plane	256x256	128K	6.45 fps	2.91 fps	1.87 fps	6.44 fps	4.23 fps	2.91 fps
Complex	32x32x4(objects)	8K	7.11 fps	3.45 fps	2.29 fps	3.74 fps	3.61 fps	3.45 fps

Figure 7: *Performance profiles of our algorithm. From left to right, the columns present the geometry image size, triangle count, rendering frame rates by adjusting the number of samples, and adjusting the LOD control parameter* α.

filtering effectively reduces aliasing artifacts and produces smooth rendering results.

4. Results and Discussion

Our algorithm is implemented using DirectX 9.0c with Shader Model 3.0. All experiments are recorded on a PC with 2.13 Ghz Intel Core2 Duo CPU, 2GB memory, and an NVidia 8800 GTX graphics card. Rendering frame rates are reported as the total frame rates at 512×512 image resolution for the entire pipeline. Since the step to shade diffuse objects is very fast, the rendering cost is dominated by the glossy reflection shading.

Table 7 summarizes the performance data. Our algorithm performs computation in image-space, therefore the rendering cost scales linearly with the total number of pixels covered by glossy objects. In addition, adjusting the LOD control parameter α changes the frame rates: lowering α reduces the effect of LOD control, thus reducing the frame rates as well. When α is zero, the LOD control is turned off, and the algorithm falls back to [CHCH06].

Dynamic Glossy Reflection Effects In Figure 5, we show a cathedral facade rendered on top of a glossy mirror that has a Phong BRDF with $n = 1000$. We sample 160 rays per pixel, and the scene is rendered at 25 fps. Note that the blurriness of the reflections changes in accordance with the relative distance between the two objects. It also changes as the viewing direction moves towards the grazing angle, as shown in Figure 5 (a) and (b). As the object is moved closer to the mirror, the reflections become sharper as show in (c) and (d).

We allow the users to dynamically change the BRDF parameters at run-time. In Figure 8 (a)-(c), we render the reflections of a gargoyle model on a glossy teapot with changing Phong exponent parameter n. The gargoyle model is a 64×64 geometry. We sample 40 rays per pixel and use $\alpha = 1$ for the LOD control. This scene is rendered at 5.91 fps at 512×512 resolution. In Figure 8 (d)-(f), we gradually move the gargoyle model away from the teapot, and observe the change in reflection blurriness. In Figure 6, we show a more complex scene with multiple diffuse objects, each represented by a 32×32 geometry image. This scene is rendered at 3.45 fps.

We can also render various per-pixel shading effects such as spatially varying BRDF parameters modulated by a texture map on the fly. In Figure 1, the intensity value of a "PG 08" texture map is used to modulate the Phong exponent parameter n. Note the per-pixel reflection effects. The textures applied in the two images are inverted, therefore in (a) the area covered by the sign is more specular while in (b) it is more blurry. Since we compute the BRDF sampling on the fly, we can manipulate the spatially varying BRDF parameters in real-time at no additional cost. Complex BRDFs can also be incorporated as long as their importance sampling functions are known.

Aliasing Artifacts When the number of sample rays is insufficient, the glossy reflections are subject to aliasing arti-

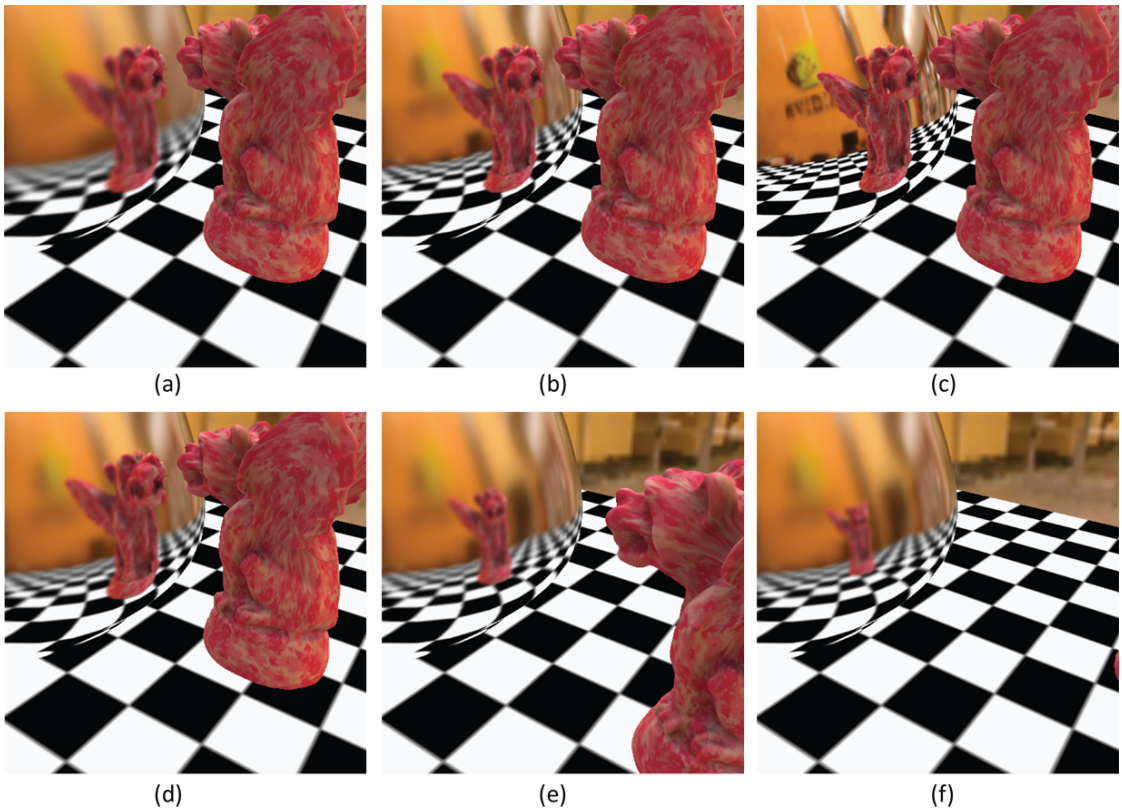

Figure 8: *This gargoyle model reflected on a glossy teapot is rendered at 5.97 fps. In (a)-(c), we dynamically change the sharpness of the BRDF applied on the teapot; in (d)-(f), we move the gargoyle model farther away from the teapot. Note the change of blurriness in the reflected images.*

Res.	LOD off	$\alpha = 1$	$\alpha = 3$	$\alpha = 10$	Speedup
1024	2.45	3.04	3.98	5.37	2.2
512	2.86	3.5	4.81	6.55	2.3
256	3.61	4.04	5.92	8.6	2.4
128	4.33	4.55	6.03	8.63	2.0
64	6.16	5.92	6.6	8.72	-
32	8.55	7.99	8.21	9.26	-

Figure 9: *A comparison of rendering fps (gargoyle + plane with 30 samples) by varying the geometry image size as well as the LOD control parameter α. Our algorithm achieves 2∼3 times speedup over [CHCH06]. By applying adaptive LOD control, the decrease in rendering performance is less sensitive to the increase in geometry image size.*

facts. In this case, our Mipmap based texture filtering (Section 3.4) is effective at reducing the aliasing artifacts. Figure 4 provides a comparison by turning on and off texture filtering. Observe the differences in the rendered results.

The aliasing artifacts are more severe near the silhouettes on the reflected images, where the ray cone partially intersects with the foreground and partially with the background environment. In this case, the texture filtering alone does not help much, and we need to increase the ray samples to improve the accuracy. Figure 10 (a)-(c) provide a comparison by changing the number of sampled rays. Note as we use more samples, the rendered image gradually converges to the reference image shown on the left of the figure; at the same time, the performance decreases linearly to the number of samples, which is expected.

Adaptive LOD control By applying our adaptive LOD control, we can achieve $2 \sim 3$x speedup in rendering performance while maintaining high rendering quality. The LOD control allows early termination of rays to reduce ray tracing cost. Figure 10 (d)-(f) provides a set of experiments. Using a larger α value, we can avoid tracing excessive rays and improve the frame rates. While this makes the intersection tests less accurate, the rendering quality is still quite high compared to the reference. Our parameters are typically set as follows: when the Phong BRDF is very sharp ($n > 3000$), we use 20 rays per pixel and set $\alpha = 3$; otherwise we use 40 rays and set $\alpha = 10$.

In [CHCH06], the geometry image resolution is a key factor affecting the ray tracing performance. A small geometry image contains less triangles, thus the ray tracing speed is

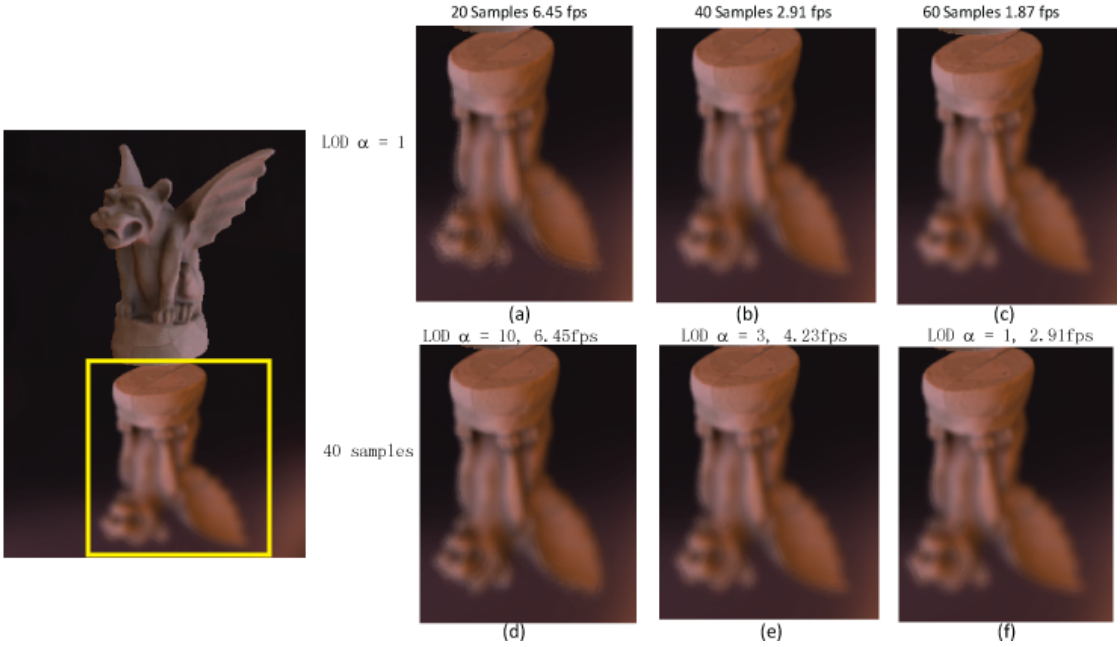

Figure 10: *Close-up view of the gargoyle scene. The left image shows a reference image computed with sufficient samples until the image converges. In (a)-(c), we fix the LOD parameter* α *to 1 and increase the number of samples per pixel; in (d)-(f) we fix the number of rays to 40 while decreasing* α*. Compare the rendering quality and note the change in performance.*

high but the reflections are at a very course resolution. A large geometry image contains more triangles, thus suffers from greatly reduced ray tracing speed. By using adaptive LOD control, our algorithm is less sensitive to the geometry image resolution. Here our solid angle criteria is used to control the maximum depth of traversal as a ray is traced through the BVH. Even when the geometry image is at a very high resolution, our algorithm can quickly eliminates the traversal of rays at unnecessarily deep levels, saving computation while maintaining accuracy. Although the ray tracing cost still increases as the size of the geometry image increases, the growth is strongly sublinear. As shown in Table 9, when LOD control is turned off (equivalent to [CHCH06]), the performance using a 32×32 geometry image is about 3.5 times that of a 1024×1024 one; when LOD is turned on with $\alpha = 10$, the ratio decreases to 1.7.

We note that when the geometry image is small, such as 32×32, turning on LOD control with a small α value (e.g. 1) actually produces slightly worse performance than turning it off. This is due to the overhead by applying the LOD control, which will be amortized as the model becomes bigger or the α value increases.

5. Limitations and Future Work

To summarize, we have presented a GPU-based algorithm for simulating dynamic glossy reflections in real-time. Our solution combines a fast GPU ray tracer with adaptive LOD control and hardware texture filtering, achieving 2∼3x speedup over the base ray tracer [CHCH06] that does not apply LOD control.

Our current algorithm is limited in several ways. First, our particular ray tracing implementation limits us to geometry image models and the BVH acceleration structure. It is possible to extend the work to more general representations and acceleration structures such as kd-tree. This would require providing a reasonable estimate of the solid angle and an approximate intersecting geometry at each node in the ray tracing hierarchy. We believe this is possible and would like to explore various options in future work.

Second, our algorithm currently requires a scene model to be classified as either a diffuse or glossy object, and we compute only one bounce of reflections from the diffuse objects to glossy objects. This disables self-reflections or transfer paths that start from glossy objects. This restriction can be alleviated by using spherical harmonics or other basis functions to store low-frequency view-dependent information. However, high-frequency glossy to glossy reflections will still be challenging to simulate.

As the major bottleneck of our algorithm is the computation cost of ray tracing, we would like to incorporate several recently published techniques into our implementation, such as the GPU-based kd-tree construction and ray tracing by [ZH*08].

Finally, we would like to investigate hybrid techniques

such as combining ray tracing with image-space filtering to accelerate the computation of glossy reflections further.

Acknowledgements We would like to thank the PG reviewers for their generous comments and suggestions. This work is supported in part by the UMass Faculty Research Grant P1-FRG-0029 and the National Science Foundation under grant NSF-MSPA-MCS-0625931.

References

[AUW07] AKERLUND O., UNGER M., WANG R.: Precomputed visibility cuts for interactive relighting with dynamic brdfs. In *Proceedings of Pacific Graphics* (2007), pp. 161–170.

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast gpu ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006* (2006), pp. 203–209.

[CK07] COLBERT M., KŘIVÁNEK J.: *GPU Gems 3*. Addison Wesley, 2007, ch. 20. GPU-based Importance Sampling, pp. 459–479.

[CLF*03] CHRISTENSEN P., LAUR D., FONG J., WOOTEN W., BATALI D.: Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum 22*, 3 (2003), 543–552.

[CON99] CABRAL B., OLANO M., NEMEC P.: Reflection space image based rendering. In *Proceedings of SIGGRAPH '99* (1999), pp. 165–170.

[DHW*] DJEU P., HUNT W., WANG R., ELHASSAN I., STOLL G., MARK W. R.: Razor: An architecture for dynamic multiresolution ray tracing. *conditionally accepted to ACM Trans. Graph.*

[DM97] DEBEVEC P. E., MALIK J.: Recovering high dynamic range radiance maps from photographs. In *Proceedings of SIGGRAPH '97* (1997), pp. 369–378.

[FS05] FOLEY T., SUGERMAN J.: Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of Graphics Hardware '05* (2005), pp. 15–22.

[GGH02] GU X., GORTLER S. J., HOPPE H.: Geometry images. *ACM Trans. Graph. 21*, 3 (2002), 355–361.

[HR06] HERNÁNDEZ B., RUDOMIN I.: Simple dynamic lod for geometry images. In *Proceedings of GRAPHITE '06* (2006), pp. 157–163.

[HS99] HEIDRICH W., SEIDEL H.-P.: Realistic, hardware-accelerated shading and lighting. In *Proceedings of SIGGRAPH '99* (1999), pp. 171–178.

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree gpu raytracing. In *Proceedings of SI3D 2007* (2007), pp. 167–174.

[IDYN07] IWASAKI K., DOBASHI Y., YOSHIMOTO F., NISHITA T.: Precomputed radiance transfer for dynamic scenes taking into account light interreflection. In *Proceedings of the 18th Eurographics Symposium on Rendering* (2007), pp. 35–44.

[Ige99] IGEHY H.: Tracing ray differentials. In *Proceedings of SIGGRAPH '99* (1999), pp. 179–186.

[JJ05] JUNFENG JI ENHUA WU S. L. X. L.: Dynamic lod on gpu. In *Proceedings of CGI 2005* (2005), pp. 108–114.

[Kaj86] KAJIYA J. T.: The rendering equation. In *Proceedings of SIGGRAPH '86* (1986), pp. 143–150.

[KVHS00] KAUTZ J., VÁZQUEZ P.-P., HEIDRICH W., SEIDEL H.-P.: Unified approach to prefiltered environment maps. In *Proc. of the 11th Eurographics Rendering Workshop* (2000), pp. 185–196.

[LWC*02] LUEBKE D., WATSON B., COHEN J. D., REDDY M., VARSHNEY A.: *Level of Detail for 3D Graphics*. Elsevier Science Inc., 2002.

[PBMH02] PURCELL T., BUCK I., MARK W., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Trans. Graph. 21*, 3 (2002), 703–712.

[RH01] RAMAMOORTHI R., HANRAHAN P.: An efficient representation for irradiance environment maps. In *Proceedings of SIGGRAPH '01* (2001), pp. 497–500.

[RH02] RAMAMOORTHI R., HANRAHAN P.: Frequency space environment map rendering. *ACM Trans. Graph. 21*, 3 (2002), 517–526.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3 (2005), 1176–1185.

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph. 21*, 3 (2002), 527–536.

[SWG*03] SANDER P. V., WOOD Z. J., GORTLER S. J., SNYDER J., HOPPE H.: Multi-chart geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (2003), pp. 146–155.

[SZC*07] SUN X., ZHOU K., CHEN Y., LIN S., SHI J., GUO B.: Interactive relighting with dynamic brdfs. *ACM Trans. Graph. 26*, 3 (2007), 27.

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164.

[WNLH06] WANG R., NG R., LUEBKE D., HUMPHREYS G.: Efficient wavelet rotation for environment map rendering. In *Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 173–182.

[YLM06] YOON S.-E., LAUTERBACH C., MANOCHA D.: R-lods: fast lod-based ray tracing of massive models. *Vis. Comput. 22*, 9 (2006), 772–784.

[ZH*08] ZHOU K., HOU Q., , WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *Microsoft Technical Report, MSR-TR-2008-52* (2008).