Improved Shader and Texture Level of Detail Using Ray Cones

Tomas Akenine-Möller
NVIDIA

Cyril Crassin
NVIDIA

Jakub Boksansky
NVIDIA

Laurent Belcour
Unity Technologies

Alexey Panteleev
NVIDIA

Oli Wright
NVIDIA

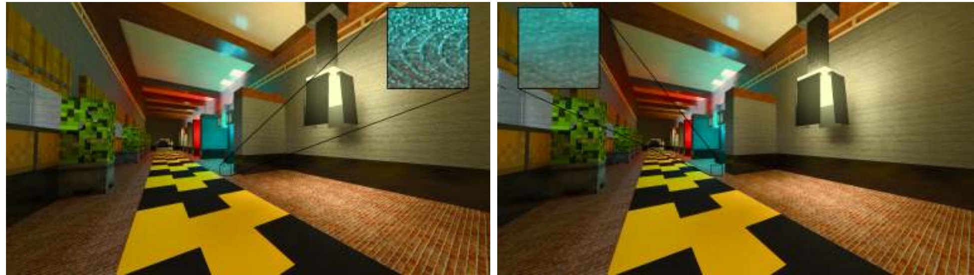


Figure 1. Naïve texture filtering using mipmap level 0 (left) versus our new anisotropic method (right) implemented in *Minecraft with RTX* on Windows 10. The insets show an $8\times$ enlargement. During animation, the left version aliases severely, while the anisotropic version is visually alias-free. All results in this paper were generated using an NVIDIA 2080 Ti, unless otherwise mentioned.

Abstract

In real-time ray tracing, texture filtering is an important technique to increase image quality. Current games, such as *Minecraft with RTX* on Windows 10, use ray cones to determine texture-filtering footprints. In this paper, we present several improvements to the ray-cones algorithm that improve image quality and performance and make it easier to adopt in game engines. We show that the total time per frame can decrease by around 10% in a GPU-based path tracer, and we provide a public-domain implementation.

1. Introduction

Texture filtering, most commonly using mipmaps [Williams 1983], is a key component in rasterization-based architectures. Key reasons for using mipmapping are to reduce texture aliasing and to increase coherence in memory accesses, making

 ISSN 2331-7418

rendering faster. Ray tracing similarly benefits from texture filtering. For example, Pharr [2018] has shown that texture filtering can give a performance boost for offline ray tracing. In ray-based renderers, texture filtering requires one to track the footprint of a pixel (or light) along with the ray. Common approaches are ray differentials [Igehy 1999] or covariance tracing [Belcour et al. 2017], both of which track an anisotropic spatio-angular footprint, but one may also use the ray-cones method [Amanatides 1984; Akenine-Möller et al. 2019], which is more compatible with real-time constraints.

The ray-cones method [Akenine-Möller et al. 2019] targets deferred rendering, i.e., G-buffer–based renderers. This requires one to store an additional float per pixel in the G-buffer, which is an inexpensive and approximate representation of curvature at the first hit point. The computation uses the per-pixel differences for positions and normals (`ddx()` and `ddy()`). This is a serious limitation, since rendering engines might not perform a G-buffer pass or have enough space left in the G-buffer to store this value; it also excludes the ability to do ray tracing for primary visibility. In addition, ray cones only evaluate an isotropic (circular) texture footprint, which may generate some overblurring, and these previous ray-cone methods could not exploit hardware-accelerated anisotropic filtering. Furthermore, the ray-cones method only handled curvature at the first hit (due to being G-buffer–based), while at secondary hits, the surface was always assumed to be planar with constant normal.

In this paper, we present a handful of improvements to the ray-cones method. Specifically, we:

- introduce more stable methods to compute approximations to surface curvature needed by the ray-cones method, which at the same time overcome the G-buffer limitation. Furthermore, these curvature approximations can also be computed after the first hit, which improves image quality in recursive reflections (Section 3);
- derive a method to change the ray-cone spread based on BRDF roughness (Section 4);
- introduce a novel method for anisotropic filtering using ray cones (Section 5);
- present a simple trick for how several textures of different sizes can be applied to the same triangle, with minimal overhead (Section 6);
- show how ray cones can be used to select the level of detail of shading for indirect rays (Section 7).

Some of these techniques have been used in the ray-traced version of *Minecraft with RTX* on Windows 10 [Boksansky and Wright 2020]. See Figures 1, 15, and 17 for some example renderings. Before introducing our contributions, we review some necessary background on ray cones.

2. Level of Detail Selection

In this section, we briefly review the selection of texture level of detail (LOD) with ray cones, where isotropic texture lookups are used. We refer the reader to the original work for details and exposition on the ray cone itself [Akenine-Möller et al. 2019]. For isotropic texture LOD, ray cones allow us to compute a value for λ that can be directly fed to `SampleLevel()` in order to sample the appropriate mipmap hierarchy level. Akenine-Möller et al. [2019] computed λ as

$$\lambda = \underbrace{\Delta}_{\text{Eqn. (2)}} + \underbrace{\log_2 |W|}_{\text{distance}} - \underbrace{\log_2 |\hat{n} \cdot \hat{d}|}_{\text{normal}}, \tag{1}$$

where $\cdot$ is the dot product, and vectors with a hat on top are normalized. The first term, Δ, computes the mipmap level needed for proper sampling when the triangle is located at $z = 1$ with its normal aligned to the view direction. The second term adjusts λ as the triangle is moved away, and the third term adjusts λ when the orientation of the triangle changes. Here, Δ is defined as

$$\Delta = \frac{1}{2} \log_2 \left(\frac{t_a}{p_a} \right), \tag{2}$$

where t_a is twice the texture area and p_a is twice the triangle area:

$$\begin{aligned} t_a &= wh \, |(t_{1x} - t_{0x})(t_{2y} - t_{0y}) - (t_{2x} - t_{0x})(t_{1y} - t_{0y})| = wht_t, \\ p_a &= \|(P_1 - P_0) \times (P_2 - P_0)\|, \end{aligned}$$

and $w \times h$ is the texture resolution, $t_{ix|y}$ are the triangle vertex i's texture coordinates, and P_i the world-space positions of the triangle's vertices.

3. Curvature Approximations

In this section, we present two approaches for computing approximations for the spread change due to curvature for primary and secondary hit points. In Section 3.1, we first show how to overcome the dependency on rasterized differentials for primary hits using a subset of the ray differential technique [Igehy 1999]. Section 3.2 introduces a new way of computing the curvature spread that is slightly less accurate, but is not limited to primary hits and can be used for any hit points. It is based on a local approximation of the mean geometric curvature of the mesh described in Section 3.3.

This results in a first method, called *ray-cones combo*, which uses the technique described in Section 3.1 for primary hit points and that of Section 3.2 with Equation (7) for secondary hit points. The second method, called *ray-cones unified*, uses the procedure of Section 3.2 for all hit points, including primary hits. It uses Equation (3.3) for primary hits, while secondary hit points require less precise curvature estimation and can be optimized using Equation (7).

Generally, the choice of whether to rely on a curvature-estimation method that will either slightly underestimate or overestimate the spread of the cone will depend on the target application. When ray cones are used inside a Whitted ray tracer, for instance, one would probably want to favor slight overestimation of the spread, in order to always avoid aliasing. On the other hand, when ray cones are used inside a Monte Carlo path tracer, one would prefer slightly underestimating the spread angle, since antialiasing will be handled by stochastic supersampling anyway, and the main objective would be to avoid introducing overblur in the results.

3.1. Primary Hit Spread Change Using Differential Normals

In this subsection, we describe how the surface spread angle, β_c, which is due to curvature, is computed at primary hit points. In previous work [Akenine-Möller et al. 2019], β_c is computed as (where we used the default values $k_1 = 1$ and $k_2 = 0$)

$$\beta_c = 2 \text{ sign}\left(\frac{\partial P}{\partial x} \cdot \frac{\partial \hat{n}}{\partial x} + \frac{\partial P}{\partial y} \cdot \frac{\partial \hat{n}}{\partial y}\right) \sqrt{\frac{\partial \hat{n}}{\partial x} \cdot \frac{\partial \hat{n}}{\partial x} + \frac{\partial \hat{n}}{\partial y} \cdot \frac{\partial \hat{n}}{\partial y}}, \tag{3}$$

where $\hat{n}$ is the world-space normal and P is the world-space position at the hit point, and the differentials were found in the G-buffer pass using `ddx()` and `ddy()`. Recall that it is the β_c-value that is stored in the G-buffer by the ray-cones method.

Our new approach is summarized as

$$\begin{aligned} \beta_x &= \arctan\left(\left\|\frac{\partial \hat{n}}{\partial x}\right\|\right), \\ \beta_y &= \arctan\left(\left\|\frac{\partial \hat{n}}{\partial y}\right\|\right), \\ \beta_c &= 2s\sqrt{\beta_x^2 + \beta_y^2}, \end{aligned}$$

where the sign s is computed as

$$s = \begin{cases} \text{sign}\left(\vec{r} \cdot \frac{\partial \hat{n}}{\partial x}\right) & \text{if } \beta_x \geq \beta_y, \\ \text{sign}\left(\vec{u} \cdot \frac{\partial \hat{n}}{\partial y}\right), & \text{else,} \end{cases}$$

and $\vec{r} = \hat{d}(x+1, y) - \hat{d}(x, y)$, a local right vector. The function $\hat{d}(x, y)$ returns the normalized camera ray direction at pixel (x, y). Similarly, we have $\vec{u} = \hat{d}(x, y+1) - \hat{d}(x, y)$, which is a local up vector. The rationale here is to compute one angle per x and y, compute the combined angle, and let the curvature sign of the largest magnitude determine the sign of β_c. To overcome the dependency of rasterized differentials, we compute $\frac{\partial \hat{n}}{\partial x}$ and $\frac{\partial \hat{n}}{\partial y}$ using a subset of the ray-differential technique [Igehy 1999] in the ray-generation step. In principle, we could have computed the positional differentials using ray differentials as well and fed that to Equation (3). However, as we will see in Figure 5, our new method generates a more visually pleasing result than the previous

method. Contrary to the ray cone itself, the estimation of the curvature is not subject to the small angle approximation (curvature can be arbitrary low or high) and thus the arctan formula should be preferred. We only show the equations for the differential in x, since both differentials are computed similarly. The normalized differential of the normal is computed as [Igehy 1999]

$$\frac{\partial \hat{n}}{\partial x} = \frac{(\vec{n} \cdot \vec{n})\frac{\partial \vec{n}}{\partial x} - (\vec{n} \cdot \frac{\partial \vec{n}}{\partial x})\vec{n}}{(\vec{n} \cdot \vec{n})^{3/2}},$$

where $\vec{n}$ is the non-normalized interpolated triangle normal , $\frac{\partial \vec{n}}{\partial x}$ is

$$\frac{\partial \vec{n}}{\partial x} = \frac{\partial u}{\partial x}(\vec{n}_1 - \vec{n}_0) + \frac{\partial v}{\partial x}(\vec{n}_2 - \vec{n}_0),$$

and $\left(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}\right)$ are differential barycentric coordinates in x [Akenine-Möller et al. 2019]. Our source code shows the details of this procedure.

3.2. Spread Change from Curvature

At each bounce, the ray-cone's spread is modified by the local curvature, β_c (Equation 32 [Akenine-Möller et al. 2019]). While curvature at primary hit points can be evaluated using screen-space differentials in the G-buffer or by using the method in Section 3.1, further bounces require a different technique. Akenine-Möller et al. used zero curvature for indirect bounces, which may lead to an incorrect estimation of the ray-cone's footprint. Our solution relies on a triangle-local approximation of mean curvature, which can be computed on the fly during ray-triangle intersection. Since our method is not dependent on any screen-space differentials, the curvature estimation is much more stable with respect to camera position. In addition, nothing needs to be stored in the G-buffer.

We start from the curvature of an edge $k = \Delta\varphi/\Delta s$, that is, the ratio of the rotation angle of the tangent $\Delta\varphi$ to the traversed arc length Δs [Woodward and Bolton 2018]. From that definition and knowing that an angular change in the normal results in twice as large a change in the reflected vector, we define $\beta_c = 2\Delta\varphi = 2k\Delta s$ as the change in cone-spread angle (or the angular change in the reflected vector) due to an angular change in the normal $\Delta\varphi$.

As illustrated in Figure 2, the intersection of a cone with a curved surface can be locally approximated as the intersection of a cone with a sphere of radius $1/k$. Within such a configuration, the arc length Δs can be computed exactly as

$$\Delta s = 2r \arcsin\left(\frac{|w'|}{2r}\right), \tag{4}$$

where $r = 1/k$ is the radius of the sphere (radius of curvature) and $|w'| = |w|/(-\hat{n} \cdot \hat{d})$ is the chord length that is the maximum length of the intersection of the cone with the

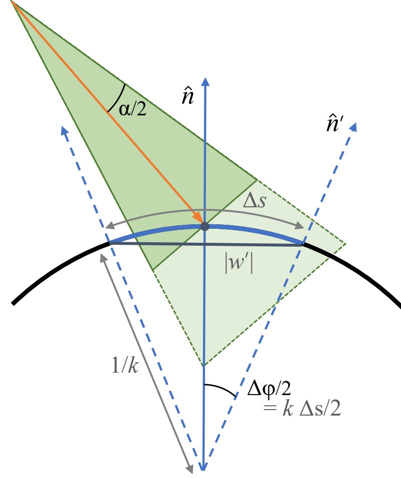


Figure 2. Geometry involved in the calculation of the angular spread $\Delta\varphi$ from surface curvature k and the arc length Δs under the cone's intersection with the surface.

tangent plane of the surface. The $|w|$ is the ray-cone width and $-\hat{n} \cdot \hat{d}$ accounts for the foreshortening effect, which makes the length of the intersection larger at grazing angles [Belcour 2012]. For relatively small cone-spread angles (resulting in small chord lengths $|w'|$), Δs can be approximated as

$$\Delta s = 2r \arcsin\left(\frac{|w'|}{2r}\right) \approx 2r \left(\frac{|w'|}{2r}\right) = |w'|,$$

where we have used the small angle approximation $\sin x \approx x$. In practice, we observe that this approximation gives good quality results. Using this approximation, the extra spread induced by curvature can be expressed as

$$\beta_c = -2k \frac{|w|}{\hat{n} \cdot \hat{d}}. \tag{5}$$

3.3. Curvature Approximation

The technique in Section 3.2 requires the curvature, k, but evaluating the exact curvature k at any point on a triangle is not easily done. Instead, we rely on a per-edge approximation. First, we note that a curvature approximation, k_{01}, for the edge from vertex 0 to vertex 1 can be computed as [Reed 2014]

$$k_{01} = \frac{1}{r} = \frac{(\hat{n}_1 - \hat{n}_0) \cdot (P_1 - P_0)}{(P_1 - P_0) \cdot (P_1 - P_0)}, \tag{6}$$

where P_i are the positions and $\hat{n}_i$ are the normals as shown in Figure 3. We refer to Reed's article for the derivation of the expression in Equation (6). The curvature approximations for the other edges of a triangle, k_{12} and k_{20}, can be computed similarly.

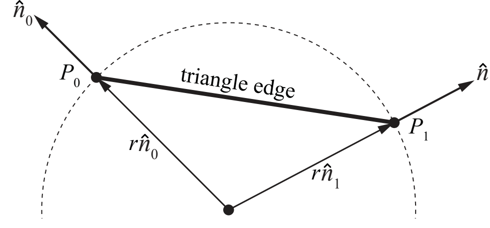


Figure 3. The geometry involved in computing the edge curvature approximation. The curvature approximation is $k_{01} = \frac{1}{r}$, where r is the (signed) radius of the circle.

For secondary hits, we have found that

$$k = \frac{k_{01} + k_{12} + k_{20}}{3}, \tag{7}$$

works well, i.e., using the average of the per-edge curvatures as the curvature approximation for the entire triangle. However, this may lead to aliasing on surfaces for which the curvature is not isotropic, especially if used for primary hits.

On such surfaces, the anisotropic curvature should generate an anisotropically shaped cone of reflected directions. Since our cone representation is only isotropic, we want to ensure aliasing is avoided by making the isotropic cone shape enclose the anisotropic-shaped cone that should ideally be tracked, and bound it as closely as possible. In order to do so, we observe that our triangle-based curvature calculation provides us with three directional curvatures, one for each edge of the triangle. Instead of averaging those directional curvatures, we approximate the visibility of each of them by clipping the edges with the elliptic footprint of the intersection of the cone with the surface (cf. Figure 10). This edge-clipping operation is illustrated in Figure 4.

We first transform the edge vectors to the tangent frame defined by the ellipse vectors, $\vec{a}_1$ and $\vec{a}_2$ (cf. Equations (8) and (9)). This is done as $\vec{e}_{ij} = (e_{ij,x}, e_{ij,y}, 0) =$

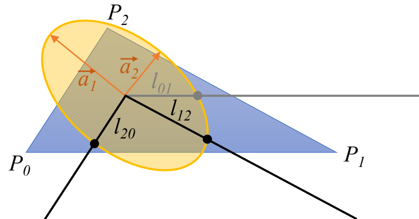


Figure 4. Approximation of the visible directional curvature: the three edges of the triangle are clipped by the elliptic footprint of the intersection of the cone with the surface in order to evaluate the contributions of the three associated curvatures.

$\mathbf{M}(P_j - P_i)$, where the matrix $\mathbf{M}$ is

$$\mathbf{M} = \begin{pmatrix} \frac{1}{\|\vec{a}_1\|}\vec{a}_1^\top \\ \frac{1}{\|\vec{a}_2\|}\vec{a}_2^\top \\ \hat{f}^\top, \end{pmatrix}$$

with the first two rows normalized ellipse axes and the last row the normalized geometrical normal, $\hat{f}$. Note that all vectors in the construction of the matrix are transposed, since we assume that vectors are column vectors, and that the third components of $\vec{e}_{ij}$ are zero, which is so since all $P_j - P_i$ lie in the plane defined by the normal $\hat{f}$.

The length of the intersection of the ellipse and a line, defined by $(e_{ij,x}, e_{ij,y})$, passing through the origin of the ellipse is given by

$$l_{ij} = \frac{\|\vec{a}_1\|\|\vec{a}_2\|}{\sqrt{\|\vec{a}_1\|^2 e_{ij,y}^2 + \|\vec{a}_2\|^2 e_{ij,x}^2}},$$

where $ij \in \{01, 12, 20\}$ are the three edges of the triangle, and $\|\vec{a}_1\|$ and $\|\vec{a}_2\|$ are the lengths of the two semiaxes of the ellipse of equation $\frac{x^2}{\|\vec{a}_1\|^2} + \frac{y^2}{\|\vec{a}_2\|^2} = 1$. This allows us to compute the relative curvature contribution of the edges by scaling their directional curvatures, k_{ij}, with the ratio of their clipped lengths and the maximum length $l_{\max} = \max(l_{01}, l_{12}, l_{20})$, that is

$$k'_{ij} = k_{ij}\frac{l_{ij}}{l_{\max}}.$$

We elect to use the k'_{ij}, which together with the ray-cone's spread angle α, gives the largest absolute value of the reflected spread angle $|\alpha + \beta_c|$ according to Equation (5). In practice, we observe that this calculation can be optimized by only accounting for the two edges that support the maximum and minimum curvature values, instead of all three edges:

$$k = \begin{cases} \min\{k_{ij}\} & \text{if } |\alpha + \beta_c(\min\{k_{ij}\})| \geq |\alpha + \beta_c(\max\{k_{ij}\})|, \\ \max\{k_{ij}\}, & \text{else.} \end{cases}$$

3.4. Curvature Results

Some results are shown in Figure 5, where the two top rows (hemisphere) show that the ray-cones method from *Ray Tracing Gems* (RC RTG) [Akenine-Möller et al. 2019] aliases, RC combo does not, but it instead slightly overblurs. RC unified generates a result that is close to that of isotropic ray differentials (RD isotropic). We note that RC combo and RC unified produce substantially less aliasing than RC RTG in recursive reflections, which is to be expected since RC RTG uses $\beta_c = 0$ for all hits beyond the first. The differences between RC combo and RC unified are best seen

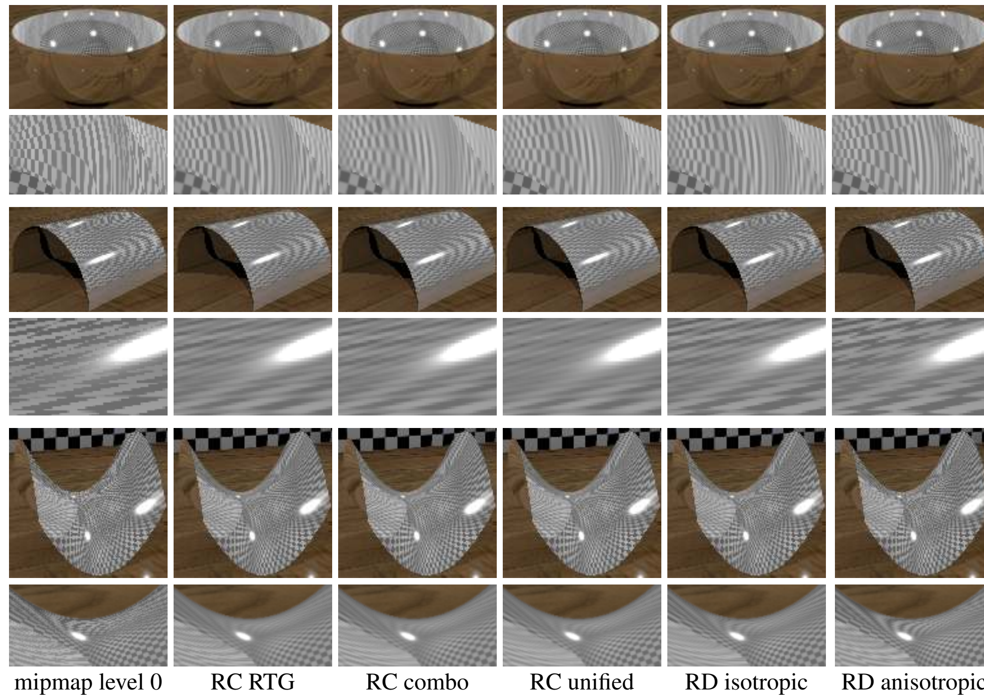


Figure 5. Top rows: a reflective hemisphere with recursive reflections; Middle rows: a reflective hemicylinder; Bottom rows: a hyperbolic paraboloid with one recursive reflection. The second column (RC RTG) shows the results from the method in *Ray Tracing Gems* based on ray cones (RC), and our new methods are shown in the third and fourth columns. For reference, results from both isotropic and anisotropic ray differentials (RD) are shown in the two rightmost columns.

during camera animation, where RC combo tends to alias less in some situations, with a slight overblur as a result, and RC unified tends to have sharper reflected textures but may alias a little more in certain situations. Note that the hemicylinder (zero curvature in one direction and positive circular curvature in the other) and the hyperbolic paraboloid (both negative and positive curvature in all points on the surface) are difficult surfaces for any texture-filtering method whose curvature representation is isotropic, which is the case for all ray-cone methods. This is the reason ray differentials, which have a much richer curvature representation, handle these cases better. Note that the ray-differential method requires 12 extra floats for its representation though, compared to two floats for ray cones. We also note that for the cylinder, RC unified is blurrier than RC combo. Whether to use RC combo or RC unified depends on the actual ray-tracing algorithm used, as well as the type of geometry, textures, and camera positions, and the appropriate method is best determined by testing both.

Performance depends on scene, camera position, texture properties (size, compression, etc), register pressure of the algorithm, graphics card, and more. In our

experience, though, all three ray-cones (RC)–based methods are equivalent in performance using a simple Whitted ray tracer. Recall, though that RC combo and RC unified produce better quality as shown in Figure 5 and can handle recursive reflections better. The ray-differential methods are usually about 5–7% slower for a range of test scenes. Note that the more relevant path-tracing results are reported in the next section.

While we have not shown any examples with normal mapping, it should be pointed out that the previous ray-cones method [Akenine-Möller et al. 2019] took normal mapping into account by using screen-space differentials of normals, though with limited success. Our new methods do not take perturbed normals into account, but it could be possible to add some kind of normal map filtering [Toksvig 2004; Olano and Baker 2010; Dupuy et al. 2013]. This would serve two purposes, namely, (1) allowing us to extract a spread-angle change from the filtered normal distribution similar to the roughness integration (Section 4), and (2) properly filter nonlinear shading parameters in order to provide correct calculation of shading or scattered energy when using mipmapped texture lookups. This is left for future work.

4. Integrating BRDF Roughness

Beyond the integration of the geometric curvature of the surfaces that a cone will bounce on, we need to account for the spread modification due to the material's BRDF. This spread depends directly on the roughness of the material and can only increase it. It follows that indirect bounces will likely access mipmap levels higher up (i.e., with lower resolution) in the hierarchy.

As the ray cone is an angular Heaviside (step) function (the representation is valid inside the cone and invalid outside), it cannot accurately represent continuous distributions, such as rough BRDFs. Instead, we are forced to rely on approximations of the increase in spread. We opted for an analytical form instead of tabulating a fitted angular spread. We further reduced the complexity of the spread by making it invariant to the incident direction. While this leads to an approximate form, it is cheap to evaluate while still giving visually acceptable results.

We derived an approximation of the variance in the tangent plane of the reflected direction for a GGX microfacet model (Appendix A). Using that, we approximate the GGX lobe with a Gaussian distribution in this space with zero mean and standard deviation σ (Figure 6). We follow the classical paraxial approximation (see Section II.1 of Gerrard and Burch's book [1994]), i.e., $\sigma = \tan(\beta_r/2) \approx \beta_r/2$ to approximate the bounding angle as $\beta_r = 2\sigma$ for the cone. This can be interpreted as a bounding angle using the area within $\pm\sigma$ of the Gaussian, covering approximately 68% of the energy of the Gaussian corresponding to the specular lobe.

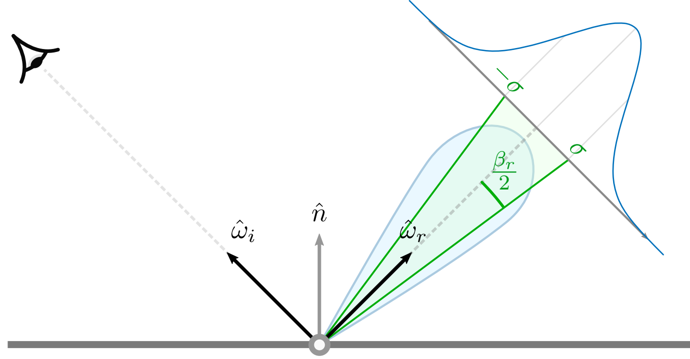


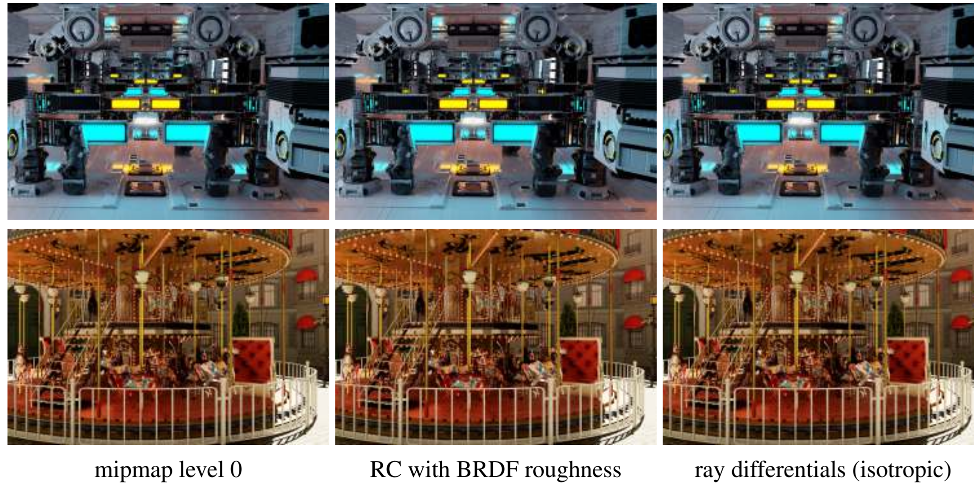
Figure 6. The increase in spread angle, β_r, due to the specular lobe of a surface's BRDF can be bound by σ^2. We use a Gaussian approximation (in blue) of the BRDF in the tangent plane of the reflected direction $\hat{\omega}_r$ to specify the BRDF angle. We use one standard deviation, σ, from the mean to obtain the BRDF angle and capture most of the lobe's energy ($\approx 68\%$).

Given a surface with roughness α, the variance in the tangent frame can be approximated as $\sigma^2 = \frac{1}{2}\left(\frac{\alpha^2}{1-\alpha^2}\right)$. The full details of the derivation can be found in Appendix A. At each hit point along a path, the spread change from curvature β_c and the roughness-induced spread change β_r combine into a global spread change $\beta = \beta_r + \beta_c$. When this prefiltering is used inside a Monte Carlo path tracer, which stochastically samples the reflected directions, the actual spread needs to be reduced below the bounding angle of the specular lobe in order to avoid overblurring. In practice, we found that using $\beta_r = \frac{1}{4}\sigma$ for primary hits and $\beta_r = \sigma$ for subsequent hits produces visually acceptable quality in most cases.

We also account for the spread-angle change with diffuse BRDFs by forcing roughness to 1 for such cases. This allows us to greatly reduce used texture resolution in the presence of Lambertian materials. However, we suffer from the same limitation as all previous approaches regarding handling caustics, which appear on paths composed of a diffuse primary hit followed by a specular hit connected to a light. We refer to the work by Belcour et al. [2017] for a discussion of that issue, which is partially solved using bidirectional path tracing. Handling bidirectional path tracing using ray cones is left for future work.

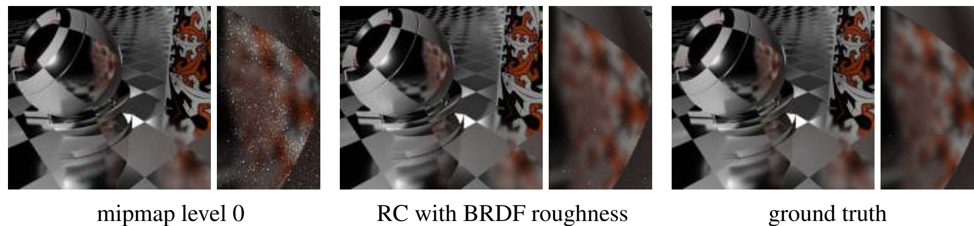
It should also be noted that with roughness integration, we reach the limit of the ray-cone representation, which approximates the 4D spatial-angular light field of scattered rays using a compact 2D representation that merges spatial and angular spreads. With such a representation, it is not possible to precisely model the effects of rough reflections on concave surfaces for instance.

Some performance results are shown in Figure 7 and convergence results are shown in Figure 8. It should be noted that, ray differentials [Igehy 1999] cannot support this type of prefiltering, but path differentials could be used instead [Suykens



mipmap level 0 RC with BRDF roughness ray differentials (isotropic)

Figure 7. All images were rendered with path tracing, different types of texture filtering, and with non-compressed textures. Due to the random nature of path tracing, the images on each row are similar. However, ray cones and ray differentials are better at exploiting texture locality since they access mipmap levels higher up in the mipmap hierarchy. This gives a performance advantage. Top row: the Zero Day scene rendered 13% faster with ray cones and 11% faster with ray differentials, compared to accessing mipmap level 0. Bottom row: the Bistro Carousel rendered 10% faster with ray cones and 8% faster with ray differentials, compared to accessing mipmap level 0. Zero Day ©beeple, Carousel ©carousel_world, Bistro ©Amazon Lumberyard.



mipmap level 0 RC with BRDF roughness ground truth

Figure 8. The left and middle images were path traced with 500 samples per pixel (SPP), while the ground truth image (right) used $54,000$ SPP. As can be seen, ray cones converge substantially quicker in the reflections with textures.

and Willems 2001]. However, this later technique is even more costly to evaluate than ray differentials. In contrast, non-differential techniques, such as ray cones and covariance tracing (not shown) [Belcour et al. 2017], handle roughness by design.

5. Anisotropic Lookups

One of the drawbacks of trilinear filtering with mipmapping is that even though it selects the best-matching texture resolution level for sampling, it does not consider

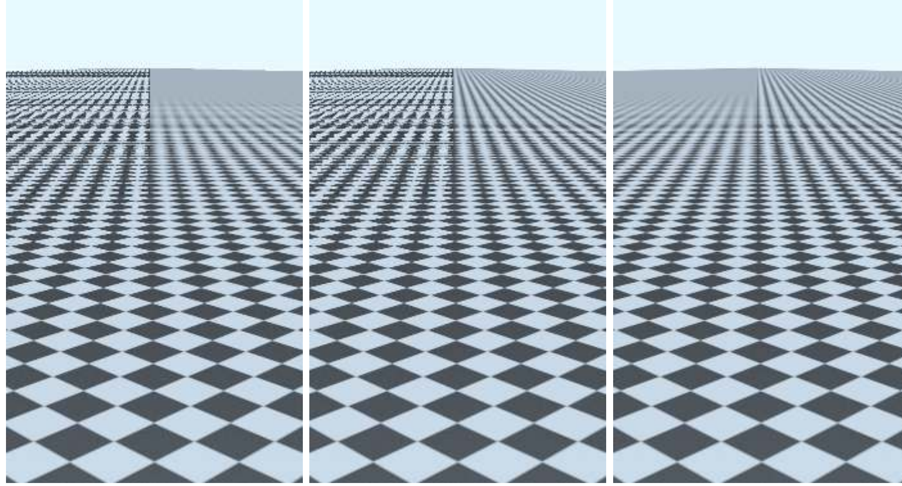


Figure 9. Comparison of texture filtering using mipmap level 0 versus our isotropic method using ray cones (left), mipmap level 0 versus our anisotropic method (center), and our isotropic versus anisotropic methods (right).

cases when the intersection of the ray cone and the textured surface covers more texels along one of the axes than the other. The resulting artifacts exhibit themselves as overblur, which is particularly noticeable on surfaces viewed at grazing angles (Figure 9). Reducing this type of artifact is important when using ray tracing for primary visibility, but also for secondary rays following a specular chain.

One possible solution is to find the texture footprint that matches the intersection of the ray cone and the triangle plane and apply a suitable filtering kernel over all texels within its elliptical bounds. When using rasterization, this can be achieved by leveraging hardware-supported anisotropic filtering, which returns a weighted average of multiple mipmap samples taken along the main axis of the pixel footprint [Schilling et al. 1996].

In this section, we present our solution for anisotropic filtering using ray cones. The texture footprint is determined by the intersection of a cone and a plane, which is an ellipse. We seek the major and minor ellipse axes, $\vec{a}_1$ and $\vec{a}_2$ that will be used to compute texture-coordinate gradients that will be fed to the hardware-accelerated anisotropic texture-lookup unit of the GPU. The geometry involved is shown in Figure 10. To reduce the computations, the ray cone is approximated as a cylinder oriented along the cone direction, $\hat{d}$, and with the width equal to the ray-cone width at the intersection point, P, on the triangle with normal $\hat{f}$. For primary hit points, this width is calculated as $w = 2r = 2t \tan\alpha$, where t is the distance to the intersection point and α is the angle of the cone. This means that the radius is $r = t \tan\alpha$. To find the direction, $\vec{h}_1$, of the first axis of the ellipse, we project the ray-cone direction

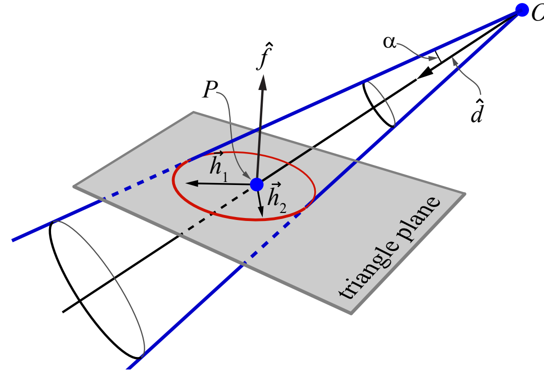


Figure 10. The direction, $\hat{d}$, of the ray cone is projected onto plane defined by the normal, $\hat{f}$, which gives us one of the ellipse axes, $\vec{h}_1$. The other ellipse axis is $\vec{h}_2 = \hat{f} \times \vec{h}_1$. These axes are rescaled later to fit the size of the ellipse.

$\hat{d}$ onto the triangle plane defined by its normal $\hat{f}$. This is done as

$$\vec{h}_1 = \hat{d} - (\hat{f} \cdot \hat{d})\hat{f}.$$

To find the length of this axis, $\vec{a}_1$, so that it spans the major direction of the ellipse, we use similar triangles as shown in Figure 11, i.e.,

$$\frac{\|\vec{a}_1\|}{\|\vec{h}_1\|} = \frac{r}{p},$$

where $\vec{a}_1$ is parallel to $\vec{h}_1$ but has the correct length and p is the length of the projection of $\vec{h}_1$ onto the plane whose normal is $\hat{d}$, i.e., $p = \|\vec{h}_1 - (\hat{d} \cdot \vec{h}_1)\hat{d}\|$. This, in turn, means that the scaled ellipse axis is

$$\vec{a}_1 = \frac{r}{p}\vec{h}_1 = \frac{r}{\|\vec{h}_1 - (\hat{d} \cdot \vec{h}_1)\hat{d}\|}\vec{h}_1. \tag{8}$$

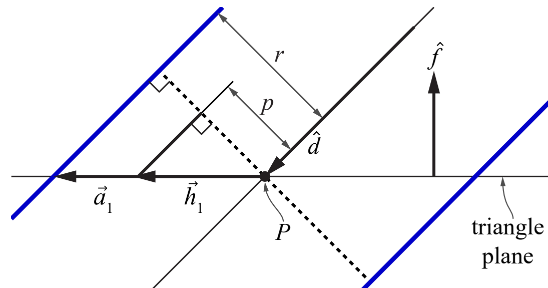


Figure 11. A side view of the blue cylinder approximation of the cone, whose direction is $\hat{d}$, at the intersection point, P, with normal $\hat{f}$. The radius of the cylinder is r and previous calculations have given us the axis $\vec{h}_1$. At this point, we need to find $\vec{a}_1$, which should extend all the way out to the blue cylinder surface. This is achieved using similar triangles.

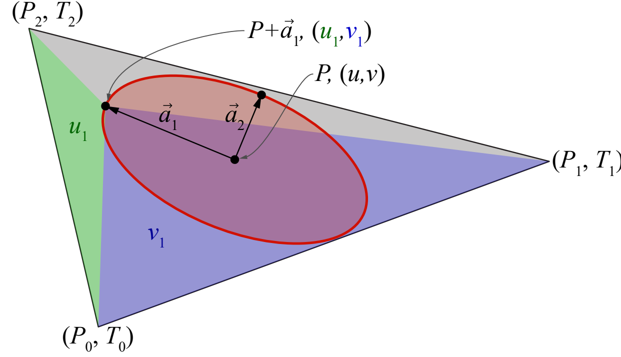


Figure 12. A triangle with positions P_i and texture coordinates $T_i = (t_{ix}, t_{iy})$. An elliptic footprint is shown with center point P and barycentric coordinates (u, v), which are different from texture coordinates. A point along the major axis of the ellipse is found as $P + \vec{a}_1$. Its barycentric coordinates are denoted (u_1, v_1). The u_1-coordinate is found by computing the area of the green triangle divided by the entire triangle area, while v_1 uses the blue triangle area divided by the entire triangle area.

The second axis, $\vec{h}_2$, is determined using a cross product between the first axis and the normal, and then it is rescaled using the same technique as shown above, i.e.,

$$\vec{h}_2 = \hat{f} \times \vec{h}_1, \qquad \vec{a}_2 = \frac{r}{\|\vec{h}_2 - (\hat{d} \cdot \vec{h}_2)\hat{d}\|}\vec{h}_2. \tag{9}$$

Note that when the ray-cone direction $\hat{d}$ is parallel to the plane normal $\hat{f}$, the scale of the axes becomes zero. To prevent division by zero in our code, we clamp the calculated axes lengths to a small constant. This also handles the case when ray direction is perpendicular to the plane normal and makes the calculation more robust.

Our goal is to find gradients of texture coordinates along the ellipse axes in texture space, in order to take an advantage of hardware-supported anisotropic filtering, which is accessible via, e.g., the `SampleGrad` HLSL intrinsic. See Figure 12 for the explanation of how the gradient $\vec{g}_1$, corresponding to the $\vec{a}_1$ axis, is found. The other gradient $\vec{g}_2$ is found analogously using $\vec{a}_2$. The first step is to compute the barycentric coordinates, (u_1, v_1), at the point $P + \vec{a}_1$. The u_1-coordinate is computed as the area of the green triangle divided by the entire triangle area, that is,

$$u_1 = \frac{\hat{f} \cdot (\vec{e}_P \times \vec{e}_2)}{\hat{f} \cdot (\vec{e}_1 \times \vec{e}_2)},$$

where $\vec{e}_1 = P_1 - P_0$, $\vec{e}_2 = P_2 - P_0$, $\vec{e}_P = P + \vec{a}_1 - P_0$, and $\hat{f}$ is the normalized triangle normal. The numerator is $\hat{f} \cdot (\vec{e}_P \times \vec{e}_2) = \|\vec{e}_P \times \vec{e}_2\|$. This is twice the area of the green triangle and this holds only if $\hat{f}$ is normalized and $\hat{f}$ is perpendicular to $\vec{e}_P$ and $\vec{e}_2$ [Ström et al. 2020], which is the case for us. This also implies that

the denominator is the area of the triangle spanned by P_0, P_1, and P_2. The other barycentric coordinate, v_1, is computed as

$$v_1 = \frac{\hat{f} \cdot (\vec{e}_1 \times \vec{e}_P)}{\hat{f} \cdot (\vec{e}_1 \times \vec{e}_2)}.$$

At this point, we know the barycentric coordinates, (u_1, v_1), at the point $P + \vec{a}_1$ on the ellipse. The last step is to compute the texture coordinate gradient based on (u_1, v_1) and (u, v). First, we note that texture coordinates can be interpolated as $T(u, v) = (1 - u - v)T_0 + uT_1 + vT_2$. This means that the gradients are

$$\vec{g}_1 = T(u_1, v_1) - T(u, v), \quad \vec{g}_2 = T(u_2, v_2) - T(u, v)$$

where (u_2, v_2) are computed analogously as (u_1, v_1), but based on $\vec{a}_2$. An anisotropic lookup is then performed as SampleGrad$(\vec{g}_1, \vec{g}_2)$. Pseudocode for our approach is shown in Listing 1.

```
// P is the intersection point
// f is the triangle normal
// d is the ray cone direction
void computeAnisotropicEllipseAxes(in float3 P, in float3 f,
 in float3 d, in float rayConeRadiusAtIntersection,
 in float3 positions[3], in float2 txcoords[3],
 in float2 interpolatedTexCoordsAtIntersection,
 out float2 texGradient1, out float2 texGradient2)
{
    // Compute ellipse axes.
    float3 a1 = d - dot(f, d) * f;
    float3 p1 = a1 - dot(d, a1) * d;
    a1 *= rayConeRadiusAtIntersection / max(0.0001, length(p1));

    float3 a2 = cross(f, a1);
    float3 p2 = a2 - dot(d, a2) * d;
    a2 *= rayConeRadiusAtIntersection / max(0.0001, length(p2));

    // Compute texture coordinate gradients.
    float3 eP, delta = P - positions[0];
    float3 e1 = positions[1] - positions[0];
    float3 e2 = positions[2] - positions[0];
    float oneOverAreaTriangle = 1.0 / dot(f, cross(e1, e2));
    eP = delta + a1;
    float u1 = dot(f, cross(eP, e2)) * oneOverAreaTriangle;
    float v1 = dot(f, cross(e1, eP)) * oneOverAreaTriangle;
    texGradient1 = (1.0-u1-v1) * txcoords[0] + u1 * txcoords[1] +
        v1 * txcoords[2] - interpolatedTexCoordsAtIntersection;
    eP = delta + a2;
    float u2 = dot(f, cross(eP, e2)) * oneOverAreaTriangle;
    float v2 = dot(f, cross(e1, eP)) * oneOverAreaTriangle;
    texGradient2 = (1.0-u2-v2) * txcoords[0] + u2 * txcoords[1] +
        v2 * txcoords[2] - interpolatedTexCoordsAtIntersection;
}
```

Listing 1. Gradient computations for anisotropic filtering.

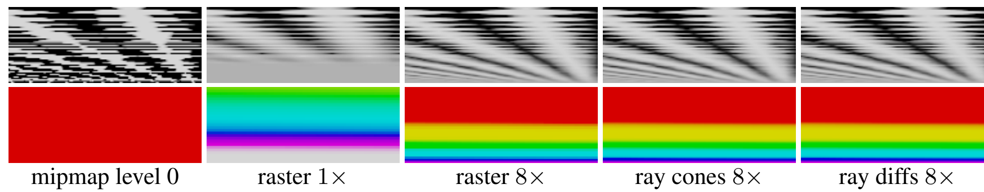


Figure 13. Comparison of different types of texture filtering at the first hit. The top row shows a checkerboard on a plane at a grazing angle. All methods with 8× next to them use 8× anisotropic texture lookups. The bottom row illustrates the mip levels that have been accessed. To generate those images, a rainbow texture was used, where all pixels in level 0 are red, all in level 1 are yellow, then green, cyan, blue, purple, and white.

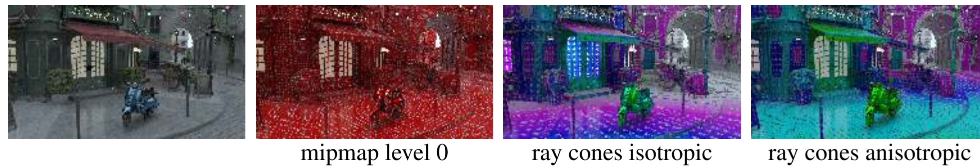


Figure 14. Rendering of the Bistro scene showing a visualization of accessed mipmap levels using naïve filtering, using mipmap level 0 (mid-left), our isotropic method (mid-right), and our anisotropic method (right). Average rendering times per frame were 98 ms (mipmap level 0), 90 ms (isotropic ray cones), and 96 ms (anisotropic ray cones). These results were generated using an NVIDIA 2080 graphics card.

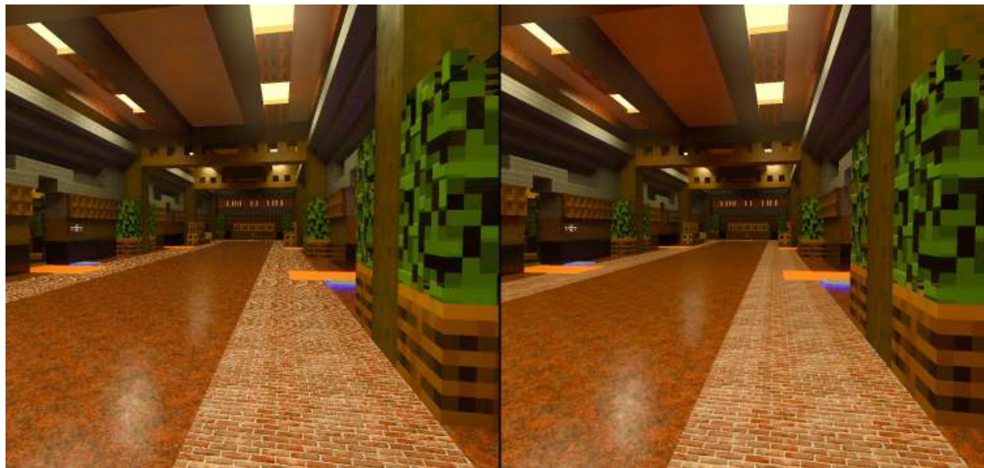


Figure 15. Comparison of naïve approach of using mipmap level 0 (left) and our anisotropic filtering using ray cones (right) in *Minecraft with RTX* on Windows 10. Note the Moiré pattern on the brick texture with naïve approach.

See Figure 13 for a comparison between different methods and Figure 14 for a visualization of mipmap levels and performance numbers. As can be seen, ray differentials, ray cones, and rasterization using $8\times$ anisotropic lookups all look similar. Note also that, despite the extra computations needed by ray cones (both isotropic and anisotropic), using mipmap level 0 is always slower. This is due to the reduction in memory bandwidth usage of ray cones, due to better texture cache locality, which turns into a performance benefit. See also Figures 1 and 15 for comparisons with and without anisotropic filtering.

6. Handling Many Textures

In this section, we present a simple trick for efficiently handling several textures (e.g., a base color and a specular texture) with different sizes per triangle.

Equation (2) bakes the texture resolution, $w \times h$, into Δ, which is not ideal when you have more than one texture (with different dimensions) applied to a triangle, since a separate texture level-of-detail value, λ, needs to be used per texture. To that end, we rewrite Equation (2) as

$$\Delta = \frac{1}{2}\log_2\left(\frac{t_a}{p_a}\right) = \frac{1}{2}\log\left(\frac{wht_t}{p_a}\right) = \underbrace{\frac{1}{2}\log_2(wh)}_{\text{texture dep.}} + \underbrace{\frac{1}{2}\log_2\left(\frac{t_t}{p_a}\right)}_{\text{texture indep.}}.$$

As can be seen, there is now one term that depends on texture resolution and one that is independent (still dependent on texture coordinates, though).

So, instead of computing λ using Equation (1), we first compute λ_t, which is independent of texture resolution:

$$\lambda_t = \frac{1}{2}\log_2\left(\frac{t_t}{p_a}\right) + \log_2|W| - \log_2|\vec{n}\cdot\vec{d}|,$$

The advantage of this is that λ_t can be shared among all textures for a triangle, and then, just before texture i is sampled, we compute the final λ_i for that texture as

$$\lambda_i = \lambda_t + \frac{1}{2}\log_2(w_i h_i),$$

where $w_i \times h_i$ is the resolution of texture i. The code snippet in Listing 2 shows how this can be done.

```
float4 sample(Texture2D t, SamplerState s, float2 uv, float lambda_t)
{
    uint w, h;
    t.GetDimensions(w, h);
    float lambda = 0.5 * log2(w * h) + lambda_t;
    return t.SampleLevel(s, uv, lambda);
}
```

Listing 2. Texture sampling code.

In one of our implementations, made in Falcor [Benty et al. 2017], we use the `interface` feature of Slang [He et al. 2018], which hides this from the user.

7. Shading Level of Detail

In addition to selecting mipmap level, ray cones can also be useful for selecting level of detail (LOD) in shading. Selecting shading LOD in rasterization-based engines has been successful [He et al. 2015], but the general approach also applies to ray tracing. Burley et al. [2018] have experimented with shading LOD in their Hyperion renderer. However, they disabled the effect in the end, because their simple shading model was evaluated per vertex, which became excessive for subpixel-sized geometry.

When shading secondary specular rays, we use the cone radius at the shading point to determine the level of detail for shading. This is illustrated in Figure 16. In addition, the roughness can be used to widen the ray cones (Section 4).

This simple technique has been used successfully in *Minecraft with RTX* on Windows 10 to select between two shading LODs. The technique was also used when shading diffuse secondary rays, but then using a fixed cone angle in that case. When a fixed cone angle is used to select between two levels of detail, the selection reduces to a simple distance threshold for the ray. Figure 17 shows an example scene from *Minecraft with RTX* on Windows 10 that demonstrates how shading LODs can be used efficiently. The final image is not obviously different to one produced with a single, high level-of-detail shading model. In this example, the lower-detail shading LOD is a prefiltered value per quadrilateral with no texture mapping, while the higher-detail shading LOD performs texture mapping, local shading computations, and shadow rays. For the simple scene in Figure 17, the indirect lighting computations were reduced from 2.0 ms to 1.75 ms, while the low-detail shading took 1.5 ms.

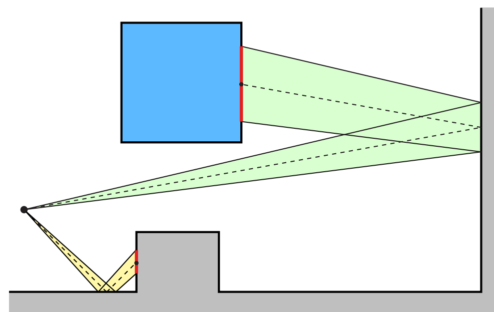


Figure 16. The yellow ray cone has a small footprint (red) at the second hit and therefore needs accurate shading evaluation there. The green ray cone has a much larger footprint at the second hit, which indicates that a lower-accuracy method for shading can be used.

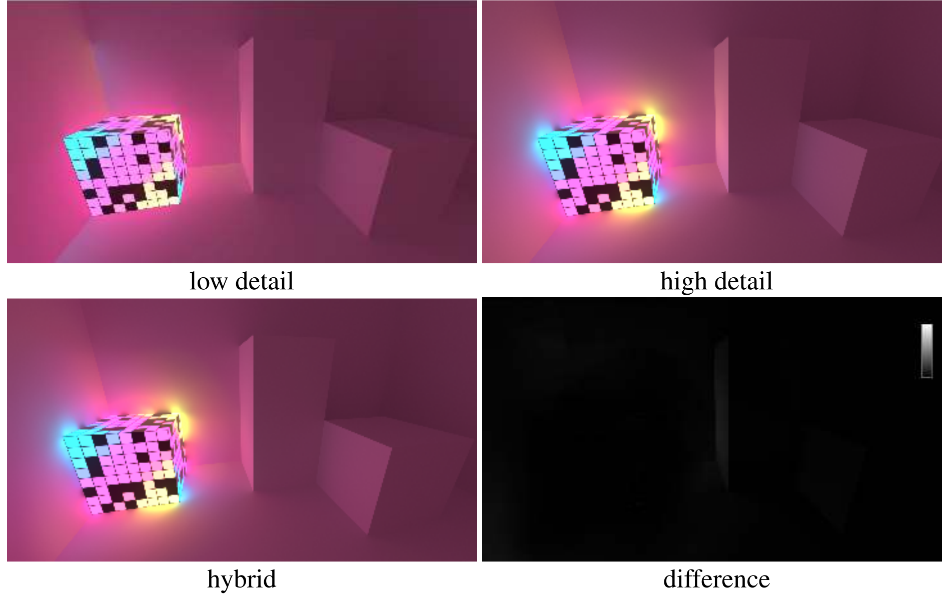


Figure 17. Examples using shading LOD for indirect lighting. The hybrid method is a combination of the low-detail image and the high-detail image, with the mix being determined by the width of the ray cones for the secondary rays. Note that the low-detail shading sees the world as flat shaded polygons, so the per-pixel emissive effect from the Disco Cube is missing. The image at the bottom right is a difference between the high detail and the hybrid images.

Acknowledgements

Thanks to Aaron Lefohn for supporting this work. We are grateful to the people sharing their test scenes, e.g., Zeroday [Winkelmann 2019], Bistro [Lumberyard 2017], and Carousel from carousel_world.

A. BRDF Roughness Derivations

Given a surface with roughness α, we convert this roughness to an approximate Phong exponent m using [Walter et al. 2007]

$$\alpha = \sqrt{\frac{2}{m+2}} \iff m = 2\left(\frac{1-\alpha^2}{\alpha^2}\right). \tag{10}$$

This allows us to obtain a BRDF lobe that is independent of the incident angle. We convert the Phong exponent to variance in angular frequency (Appendix B.1 [Belcour 2012]) as

$$\bar{\sigma}^2 = \frac{m}{4\pi^2}.$$

After replacing m with Equation (10) in the expression above, we obtain

$$\bar{\sigma}^2 = 2\left(\frac{1-\alpha^2}{\alpha^2}\right)\frac{1}{4\pi^2} = \frac{1-\alpha^2}{2\pi^2\alpha^2}.$$

Note that $\bar{\sigma}^2$ represent the variance of the Fourier transform of the reflected lobe in the tangent frame. To obtain the variance in the primal domain, we need to apply the inverse Fourier transform on it. Approximating the lobe by a Gaussian with equivalent variance and zero mean, the Fourier transform of the BRDF lobe is

$$\bar{\rho}(f) = A \exp\left(-\frac{f^2}{2\bar{\sigma}^2}\right) = A \exp\left(-f^2\left(\frac{\pi^2\alpha^2}{1-\alpha^2}\right)\right),$$

where f is the frequency and A is the amplitude of the Gaussian. Using the formula for the inverse Fourier transform of a Gaussian, i.e.,

$$\mathcal{F}^{-1}\left[\sqrt{\frac{\pi}{a}} \exp\left(-\frac{\pi^2 f^2}{a}\right)\right](x) = \exp\left(-ax^2\right),$$

with $a = \frac{1-\alpha^2}{\alpha^2}$, we obtain

$$\rho(x) = \sqrt{\frac{a}{\pi}} A \exp\left(-x^2\left(\frac{1-\alpha^2}{\alpha^2}\right)\right)$$

Hence, the variance in the tangent frame is approximately $\sigma^2 = \frac{1}{2}\left(\frac{\alpha^2}{1-\alpha^2}\right)$.

References

AKENINE-MÖLLER, T., NILSSON, J., ANDERSSON, M., BARRÉ-BRISEBOIS, C., TOTH, R., AND KARRAS, T. 2019. Texture Level-of-Detail Strategies for Real-Time Ray Tracing. In *Ray Tracing Gems*, E. Haines and T. Akenine-Möller, Eds. Apress, Berkeley, CA, USA, ch. 20, 321–345. URL: https://doi.org/10.1007/978-1-4842-4427-2_20. 2, 3, 4, 5, 8, 10

AMANATIDES, J. 1984. Ray Tracing with Cones. *Computer Graphics (SIGGRAPH) 18*, 3, 129–135. URL: https://doi.org/10.1145/964965.808589. 2

BELCOUR, L., YAN, L.-Q., RAMAMOORTHI, R., AND NOWROUZEZAHRAI, D. 2017. Antialiasing Complex Global Illumination Effects in Path-Space. *ACM Transactions on Graphics 36*, 1. URL: https://doi.org/10.1145/2990495. 2, 11, 12

BELCOUR, L. 2012. *A Frequency Analysis of Light Transport from Theory to Implementation*. PhD thesis, Université de Grenoble. URL: https://tel.archives-ouvertes.fr/tel-00766866v1/file/thesis.pdf. 6, 20

BENTY, N., YAO, K.-H., FOLEY, T., KAPLANYAN, A. S., LAVELLE, C., WYMAN, C., AND VIJAY, A., 2017. The Falcor Rendering Framework. https://github.com/NVIDIAGameWorks/Falcor, July. 19

BOKSANSKY, J., AND WRIGHT, O. 2020. Minecraft with RTX: Crafting a Real-Time Path Tracer for Gaming. In *GDC Digital talk*. URL: https://gdcvault.com/play/1026716/Minecraft-with-RTX-Crafting-a. 2

BURLEY, B., ADLER, D., CHIANG, M. J.-Y., DRISKILL, H., HABEL, R., KELLY, P., KUTZ, P., LI, Y. K., AND TEECE, D. 2018. The Design and Evolution of Disneys Hyperion Renderer. *ACM Transactions on Graphics 37*, 3. URL: https://doi.org/10.1145/3182159. 19

DUPUY, J., HEITZ, E., IEHL, J.-C., POULIN, P., NEYRET, F., AND OSTROMOUKHOV, V. 2013. Linear Efficient Antialiased Displacement and Reflectance Mapping. *ACM Transactions on Graphics 32*, 6. URL: https://doi.org/10.1145/2508363.2508422. 10

GERRARD, A., AND BURCH, J. M. 1994. *Introduction to Matrix Methods in Optics*. Dover Publications, New York, NY, USA. 10

HE, Y., FOLEY, T., TATARCHUK, N., AND FATAHALIAN, K. 2015. A System for Rapid, Automatic Shader Level-of-detail. *ACM Transactions on Graphics 34*, 6, 187:1–187:12. URL: https://doi.org/10.1145/2816795.2818104. 19

HE, Y., FATAHALIAN, K., AND FOLEY, T. 2018. Slang: Language Mechanisms for Extensible Real-Time Shading Systems. *ACM Transactions on Graphics 37*, 4. URL: https://doi.org/10.1145/3197517.3201380. 19

IGEHY, H. 1999. Tracing Ray Differentials. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '99, 179–186. URL: https://doi.org/10.1145/311535.311555. 2, 3, 4, 5, 11

LUMBERYARD, A., 2017. Amazon lumberyard bistro, open research content archive (orca), July. URL: http://developer.nvidia.com/orca/amazon-lumberyard-bistro. 20

OLANO, M., AND BAKER, D. 2010. LEAN Mapping. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Association for Computing Machinery, New York, NY, USA, 181–188. URL: https://doi.org/10.1145/1730804.1730834. 10

PHARR, M. 2018. Swallowing the Elephant (Part 5). In *Matt Pharr's blog*. URL: https://pharr.org/matt/blog/2018/07/16/moana-island-pbrt-5.html. 2

REED, N., 2014. What is the Simplest Way to Compute Principal Curvature for a Mesh Triangle? https://computergraphics.stackexchange.com/questions/1718/what-is-the-simplest-way-to-compute-principal-curvature-for-a-mesh-triangle, November. 6

SCHILLING, A., KNITTEL, G., AND STRASSER, W. 1996. Texram: A Smart Memory for Texturing. *IEEE Computer Graphics and Applications 16*, 3, 32–41. URL: https://doi.org/10.1109/38.491183. 13

STRÖM, J., ÅSTRÖM, K., AND AKENINE-MÖLLER, T. 2020. *Immersive Linear Algebra*, 1.1 ed. URL: http://www.immersivemath.com. 15

SUYKENS, F., AND WILLEMS, Y. D. 2001. Path Differentials and Applications. In *Eurographics Workshop on Rendering*, The Eurographics Association, Aire-la-Ville, Switzerland, 257–268. URL: http://diglib.eg.org/handle/10.2312/EGWR.EGWR01.257-268. 12

TOKSVIG, M., 2004. Mipmapping Normal Maps. NVIDIA Technical Brief, https://developer.download.nvidia.com/whitepapers/2006/Mipmapping_Normal_Maps.pdf, April. 10

WALTER, B., MARSCHNER, S. R., LI, H., AND TORRANCE, K. E. 2007. Microfacet Models for Refraction Through Rough Surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, Eurographics Association, Aire-la-Ville, Switzerland, 195–206. URL: `http://dx.doi.org/10.2312/EGWR/EGSR07/195-206`. 20

WILLIAMS, L. 1983. Pyramidal Parametrics. *Computer Graphics (SIGGRAPH) 17*, 3, 1–11. URL: `https://doi.org/10.1145/964967.801126`. 1

WINKELMANN, M., 2019. Zero-day, open research content archive (orca), November. URL: `https://developer.nvidia.com/orca/beeple-zero-day`. 20

WOODWARD, L. M., AND BOLTON, J. 2018. *A First Course in Differential Geometry*. Cambridge University Press, Cambridge, UK. 5

Index of Supplemental Materials

Our code is available in Falcor.

Author Contact Information

Tomas Akenine-Möller
NVIDIA
Ideon Science Park
Scheelevägen 28
223 70 Lund
Sweden
takenine@nvidia.com

Cyril Crassin
NVIDIA Ltd.
10 Avenue de l'Arche
92400 Courbevoie
France
ccrassin@nvidia.com

Laurent Beclour
Unity Grenoble
51 avenue Jean Kuntzmann
38330 Montbonnot
France
laurent.beclour@gmail.com

Jakub Boksansky
NVIDIA
Einsteinstraße 172/5th Floor
81677 München
Germany
jboksansky@nvidia.com

Alexey Panteleev
NVIDIA
2788 San Tomas Expressway
Santa Clara, CA 95051
USA
alpanteleev@nvidia.com

Oli Wright
NVIDIA Ltd.
100 Brook Drive
Reading RG2 6UJ
United Kingdom
owright@nvidia.com

T. Akenine-Möller, C. Crassin, J. Boksansky, L. Belcour, A. Panteleev, and O. Wright, Improved Shader and Texture Level of Detail Using Ray Cones, *Journal of Computer Graphics Techniques (JCGT)*, vol. 10, no. 1, 1–24, 2021
http://jcgt.org/published/0010/01/01/

Received:	2020-06-10		
Recommended:	2020-11-24	Corresponding Editor:	Wenzel Jakob
Published:	2021-01-25	Editor-in-Chief:	Marc Olano

© 2021 T. Akenine-Möller, C. Crassin, J. Boksansky, L. Belcour, A. Panteleev, and O. Wright (the Authors).
The Authors provide this document (the Work) under the Creative Commons CC BY-ND 3.0 license available online at http://creativecommons.org/licenses/by-nd/3.0/. The Authors further grant permission for reuse of images and text from the first page of the Work, provided that the reuse is for the purpose of promoting and/or summarizing the Work in scholarly venues and that any reuse is accompanied by a scientific citation to the Work.