Chapter 20 in Ray Tracing Gems, edited by Eric Haines and Tomas Akenine-Möller, Apress, 2019.

Texture Level of Detail Strategies for Real-Time Ray Tracing

Tomas Akenine-Möller,[1] Jim Nilsson,[1] Magnus Andersson,[1] Colin Barré-Brisebois,[2] Robert Toth[1], and Tero Karras[1]
[1]NVIDIA, [2]SEED / Electronic Arts

Abstract

Unlike rasterization, where one can rely on pixel quad partial derivatives, an alternative approach must be taken for filtered texturing during ray tracing. We describe two methods for computing texture level of detail for ray tracing. The first approach uses ray differentials, which is a general solution that gives high-quality results. It is rather expensive in terms of computations and ray storage, however. The second method builds on ray cone tracing and uses a single trilinear lookup, a small amount of ray storage, and fewer computations than ray differentials. We explain how ray differentials can be implemented within DirectX Raytracing (DXR) and how to combine them with a G-buffer pass for primary visibility. We present a new method to compute barycentric differentials. In addition, we give previously unpublished details about ray cones and provide a thorough comparison with bilinearly filtered mip level 0, which we consider as a base method.

1 Introduction

Mipmapping [17] is the standard method to avoid texture aliasing, and all GPUs support this technique for rasterization. OpenGL [7, 15], for example, specifies the level of detail (LOD) parameter λ as

$$\lambda(x, y) = \log_2 \lceil \rho(x, y) \rceil, \tag{1}$$

where (x, y) are pixel coordinates and the function ρ may be computed as

$$\rho(x, y) = \max\left\{ \sqrt{\left(\frac{\partial s}{\partial x}\right)^2 + \left(\frac{\partial t}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial s}{\partial y}\right)^2 + \left(\frac{\partial t}{\partial y}\right)^2} \right\}, \tag{2}$$

for two-dimensional texture lookups, where (s, t) are texel coordinates, i.e., texture coordinates ($\in [0, 1]^2$) multiplied by texture resolution. See Figure 1. These functions help ensure that sampling the mipmap hierarchy occurs such that a screen-space pixel maps to approximately one texel. In general, GPU hardware computes these differentials by always evaluating pixel shaders over 2×2 pixel quads and

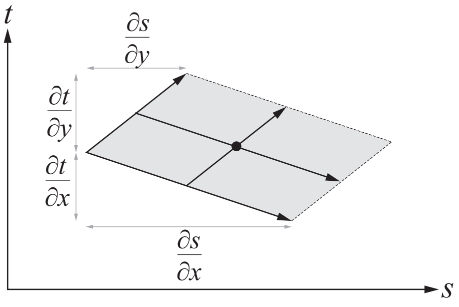

t
$\frac{\partial s}{\partial y}$
$\frac{\partial t}{\partial y}$
$\frac{\partial t}{\partial x}$
$\frac{\partial s}{\partial x}$
s

Figure 1: The footprint of a pixel approximated as a parallelogram in texture space. This notation is used in Equation 2.

by using per-pixel differences. Note, however, that Equation 2 is not conservative for a single trilinear lookup, as it does not compute a minimum box around the footprint. The maximum side of such a conservative box can be computed as $\rho(x, y) = \max(|\partial s/\partial x| + |\partial s/\partial y|, |\partial t/\partial x| + |\partial t/\partial y|)$. OpenGL allows use of more conservative estimates than Equation 2, but we are unaware of any such approach or implementation. As a consequence, it is easily shown via GPU texturing that most methods can produce both overblur and aliasing.

For ray tracing, a method to compute texture LOD is desired and it should be capable of handling recursive ray paths as well. Since pixel quads are not generally available for ray tracing (except possibly for eye rays), other approaches are needed. This chapter describes two texturing methods for real-time ray tracing. The first, *ray differentials* [9], uses the chain rule to derive expressions that can accurately compute texture footprints even for specular reflections and refractions. Ray differentials are computationally expensive and use a substantial amount of per-ray data, but provide high-quality texture filtering. The second, called *ray cones*, is less expensive and uses a cone to represent ray footprints as they grow or shrink depending on distance and surface interactions. We describe implementations of these two methods in DXR. See also Chapter **??** for information about how to filter environment map lookups in a ray tracing engine.

2 Background

For filtered texture mapping, it is common to use a hierarchical image pyramid, called a mipmap, for acceleration [17]. Each pixel footprint gets mapped to texture space and a λ-value is computed. This λ, together with the current fragment's texture coordinates, is used to gather and trilinearly filter eight samples from the mipmap. Heckbert [7, 8] surveyed various texture filtering techniques, and McCormack et al. [10] presented a method for anisotropic sampling along with a survey of previous methods. Greene and Heckbert [6] presented the elliptical weighted average (EWA) filter, often considered the method with best quality and reasonable performance. EWA computes an elliptical footprint in texture space and samples the mipmap using several lookups with Gaussian weights. EWA can be used both for rasterization and for ray tracing.

Ewins et al. [5] presented various approximations for texture LOD, and we refer readers to their survey of current methods. For example, they describe a crude approximation using a single LOD for an entire triangle. This is computed as

$$\Delta = \log_2\left(\sqrt{\frac{t_a}{p_a}}\right) = 0.5\log_2\left(\frac{t_a}{p_a}\right), \tag{3}$$

where the variables t_a and p_a are twice the texel-space area and twice the triangle area in screen space, respectively. These are computed as

$$\begin{aligned} t_a &= wh\left|(t_{1x}-t_{0x})(t_{2y}-t_{0y})-(t_{2x}-t_{0x})(t_{1y}-t_{0y})\right|, \\ p_a &= \left|(p_{1x}-p_{0x})(p_{2y}-p_{0y})-(p_{2x}-p_{0x})(p_{1y}-p_{0y})\right|, \end{aligned} \tag{4}$$

where $w \times h$ is the texture resolution, $T_i = (t_{ix}, t_{iy})$ are two-dimensional texture coordinates for each vertex, and $P_i = (p_{ix}, p_{iy})$, $i \in \{0, 1, 2\}$, are the three screen-space triangle vertices. Twice the triangle area can also be computed in world space as

$$p_a = ||(P_1 - P_0) \times (P_2 - P_0)||, \tag{5}$$

where P_i now are in world space. We exploit that setup as part of our solution for ray cones filtering, since Equation 3 gives a one-to-one mapping between pixels and texels if the triangle lies on the $z = 1$ plane. In this case, Δ can be considered as a base texture level of detail of the triangle.

Igehy [9] presented the first method to filter textures for ray tracing. He used ray differentials, tracked these through the scene, and applied the chain rule to model reflections and refractions. The computed LOD works with either regular mipmapping or anisotropically sampled mipmapping. Another texturing method for ray tracing is based on using cones [1]. Recently, Christensen et al. [3] revealed that they also use a ray cone representation for filtering textures in movie rendering, i.e., similar to what is presented in Section 3.4.

3 Texture Level of Detail Algorithms

This section describes the texture LOD algorithms that we consider for real-time ray tracing. We improve the ray cones method (Section 3.4) so that it handles curvature at the first hit, which improves quality substantially. We also extend ray differentials for use with a G-buffer, which improves performance. In addition, we present a new method for how to compute barycentric differentials.

3.1 Mip Level 0 with Bilinear Filtering

One easy way to access textures is to sample mip level 0. This generates great images using many rays per pixel, but performance can suffer since repeated mip level 0 accesses often lead to poor texture caching. When tracing with only a few rays per pixel, quality will suffer, especially when minification occurs. Enabling bilinear filtering provides a small improvement. However, with a competent denoiser as a post-process, bilinear filtering may suffice, as the denoised result is blurred.

3.2 Ray Differentials

Assume that a ray is represented (see Chapter **??**) as

$$R(t) = O + t\widehat{\mathbf{d}}, \tag{6}$$

where O is the ray origin and $\widehat{\mathbf{d}}$ is the normalized ray direction, i.e., $\widehat{\mathbf{d}} = \mathbf{d}/\|\mathbf{d}\|$. The corresponding ray differential [9] consists of four vectors:

$$\left\{ \frac{\partial O}{\partial x}, \frac{\partial O}{\partial y}, \frac{\partial \widehat{\mathbf{d}}}{\partial x}, \frac{\partial \widehat{\mathbf{d}}}{\partial y} \right\}, \tag{7}$$

where (x, y) are the screen coordinates, with one unit between adjacent pixels. The core idea is to track a ray differential along each path as it bounces around in the scene. No matter the media that the rays traverse, all interactions along the path are differentiated and applied to the incoming ray differential, producing an outgoing ray differential. When indexing into a texture, the current ray differential determines the texture footprint. Most equations from the ray differential paper [9] can be used as presented, but the differential for the eye ray direction needs modification. We also optimize the differential barycentric coordinate computation.

3.2.1 Eye Ray Setup

The non-normalized eye ray direction $\mathbf{d}$ for a pixel at coordinate (x, y) for a $w \times h$ screen resolution is usually generated in DXR as

$$\begin{gathered} \mathbf{p} = \left(\frac{x + 0.5}{w}, \frac{y + 0.5}{h} \right), \quad \mathbf{c} = (2p_x - 1, 2p_y - 1), \quad \text{and} \\ \mathbf{d}(x, y) = c_x \mathbf{r} + c_y \mathbf{u} + \mathbf{v} = \left(\frac{2x + 1}{w} - 1 \right) \mathbf{r} + \left(\frac{2y + 1}{h} - 1 \right) \mathbf{u} + \mathbf{v}, \end{gathered} \tag{8}$$

or using some minor modification of this setup. Here, $\mathbf{p} \in [0, 1]^2$, where the 0.5 values are added to get to the center of each pixel, i.e., the same as in DirectX and OpenGL, and thus $\mathbf{c} \in [-1, 1]$. The right-hand, orthonormal camera basis is $\{\mathbf{r}', \mathbf{u}', \mathbf{v}'\}$, i.e., $\mathbf{r}'$ is the right vector, $\mathbf{u}'$ is the up vector, and $\mathbf{v}'$ is the view vector pointing toward the camera position. Note that we use $\{\mathbf{r}, \mathbf{u}, \mathbf{v}\}$ in Equation 8, and these are just scaled versions of the camera basis, i.e.,

$$\{\mathbf{r}, \mathbf{u}, \mathbf{v}\} = \{af\mathbf{r}', -f\mathbf{u}', -\mathbf{v}'\}, \tag{9}$$

where a is the aspect ratio and $f = \tan(\omega/2)$, where ω is the vertical field of view.

For eye rays, Igehy [9] computes the ray differentials for the direction as

$$\frac{\partial \mathbf{d}}{\partial x} = \frac{(\mathbf{d} \cdot \mathbf{d})\bar{\mathbf{r}} - (\mathbf{d} \cdot \bar{\mathbf{r}})\mathbf{d}}{(\mathbf{d} \cdot \mathbf{d})^{\frac{3}{2}}} \quad \text{and} \quad \frac{\partial \mathbf{d}}{\partial y} = \frac{(\mathbf{d} \cdot \mathbf{d})\bar{\mathbf{u}} - (\mathbf{d} \cdot \bar{\mathbf{u}})\mathbf{d}}{(\mathbf{d} \cdot \mathbf{d})^{\frac{3}{2}}}, \tag{10}$$

where $\bar{\mathbf{r}}$ is the right vector from one pixel to the next and $\bar{\mathbf{u}}$ is the up vector, which in our case are

$$\bar{\mathbf{r}} = \mathbf{d}(x + 1, y) - \mathbf{d}(x, y) = \frac{2af}{w}\mathbf{r}' \quad \text{and} \quad \bar{\mathbf{u}} = \mathbf{d}(x, y + 1) - \mathbf{d}(x, y) = -\frac{2f}{h}\mathbf{u}', \tag{11}$$

derived using Equation 8. This is all that is needed to set up the ray differential for eye rays.

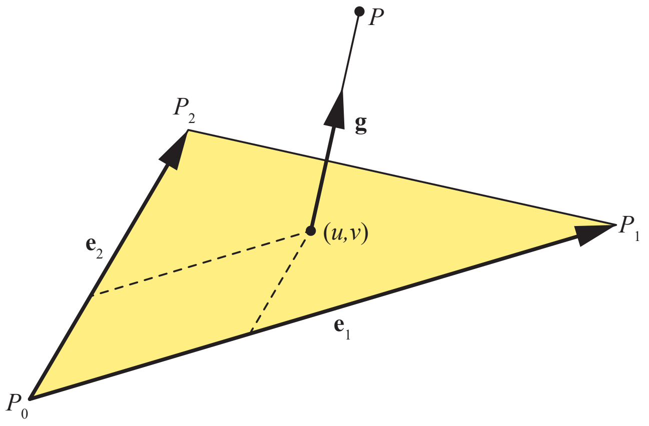


Figure 2: The setup for the derivation of differential barycentric coordinate computation.

3.2.2 Optimized Differential Barycentric Coordinate Computation

Any point on the triangle can be described using barycentric coordinates (u, v) as $P_0 + u\mathbf{e}_1 + v\mathbf{e}_2$, where $\mathbf{e}_1 = P_1 - P_0$ and $\mathbf{e}_2 = P_2 - P_0$. Once we have found an intersection, we need to compute the differentials of these, i.e., $\partial u/\partial x$, $\partial u/\partial y$, $\partial v/\partial x$, and $\partial v/\partial y$. Now let P be an arbitrary point in space and let $\mathbf{g}$ be a projection vector, which is not parallel to the triangle plane. The point $P = (p_x, p_y, p_z)$ can be described as

$$P = \underbrace{P_0 + u\mathbf{e}_1 + v\mathbf{e}_2}_{\text{point on triangle plane}} + s\mathbf{g}, \tag{12}$$

where s is the projection distance. This is illustrated in Figure 2.

This setup is similar to the one used in some ray/triangle intersection tests [11] and can be expressed as a system of linear equations and hence solved using Cramer's rule, which results in

$$\begin{aligned} u &= \frac{1}{k}(\mathbf{e}_2 \times \mathbf{g}) \cdot (P - P_0) = \frac{1}{k}\big((\mathbf{e}_2 \times \mathbf{g}) \cdot P - (\mathbf{e}_2 \times \mathbf{g}) \cdot P_0\big), \\ v &= \frac{1}{k}(\mathbf{g} \times \mathbf{e}_1) \cdot (P - P_0) = \frac{1}{k}\big((\mathbf{g} \times \mathbf{e}_1) \cdot P - (\mathbf{g} \times \mathbf{e}_1) \cdot P_0\big), \end{aligned} \tag{13}$$

where $k = (\mathbf{e}_1 \times \mathbf{e}_2) \cdot \mathbf{g}$. From these expressions, we can see that

$$\frac{\partial u}{\partial P} = \frac{1}{k}(\mathbf{e}_2 \times \mathbf{g}) \quad \text{and} \quad \frac{\partial v}{\partial P} = \frac{1}{k}(\mathbf{g} \times \mathbf{e}_1). \tag{14}$$

These expressions will be useful later on in the derivation. Next, assume that a point of intersection is computed as $P = O + t\mathbf{d}$ (note that the ray direction vector $\mathbf{d}$ needs not be normalized), which means that we can express $\partial P/\partial x$ as

$$\frac{\partial P}{\partial x} = \frac{\partial(O + t\mathbf{d})}{\partial x} = \frac{\partial O}{\partial x} + t\frac{\partial \mathbf{d}}{\partial x} + \frac{\partial t}{\partial x}\mathbf{d} = \mathbf{q} + \frac{\partial t}{\partial x}\mathbf{d}, \tag{15}$$

where $\mathbf{q} = \partial O/\partial x + t(\partial \mathbf{d}/\partial x)$. The same is done for $\partial P/\partial y$, where we instead use $\mathbf{r} = \partial O/\partial y + t(\partial \mathbf{d}/\partial y)$. We use the results from Equations 14 and 15 together with the chain rule to obtain

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial p_x}\frac{\partial p_x}{\partial x} + \frac{\partial u}{\partial p_y}\frac{\partial p_y}{\partial x} + \frac{\partial u}{\partial p_z}\frac{\partial p_z}{\partial x} = \underbrace{\frac{\partial u}{\partial P} \cdot \frac{\partial P}{\partial x}}_{\text{dot product}} = \frac{1}{k}(\mathbf{e}_2 \times \mathbf{g}) \cdot \left(\mathbf{q} + \frac{\partial t}{\partial x}\mathbf{d}\right). \quad (16)$$

Next, we choose $\mathbf{g} = \mathbf{d}$ and simplify the previous expression to

$$\frac{\partial u}{\partial x} = \frac{1}{k}(\mathbf{e}_2 \times \mathbf{d}) \cdot \left(\mathbf{q} + \frac{\partial t}{\partial x}\mathbf{d}\right) = \frac{1}{k}(\mathbf{e}_2 \times \mathbf{d}) \cdot \mathbf{q}, \quad (17)$$

since $(\mathbf{e}_2 \times \mathbf{d}) \cdot \mathbf{d} = 0$. Now, the expressions for which we sought can be summarized as

$$\begin{aligned} \frac{\partial u}{\partial x} &= \frac{1}{k}\mathbf{c}_u \cdot \mathbf{q} \quad \text{and} \quad \frac{\partial u}{\partial y} = \frac{1}{k}\mathbf{c}_u \cdot \mathbf{r}, \\ \frac{\partial v}{\partial x} &= \frac{1}{k}\mathbf{c}_v \cdot \mathbf{q} \quad \text{and} \quad \frac{\partial v}{\partial y} = \frac{1}{k}\mathbf{c}_v \cdot \mathbf{r}, \end{aligned} \quad (18)$$

where

$$\mathbf{c}_u = \mathbf{e}_2 \times \mathbf{d}, \;\; \mathbf{c}_v = \mathbf{d} \times \mathbf{e}_1, \;\; \mathbf{q} = \frac{\partial O}{\partial x} + t\frac{\partial \mathbf{d}}{\partial x}, \;\; \mathbf{r} = \frac{\partial O}{\partial y} + t\frac{\partial \mathbf{d}}{\partial y}, \;\; \text{and} \;\; k = (\mathbf{e}_1 \times \mathbf{e}_2) \cdot \mathbf{d}. \quad (19)$$

Note that $\mathbf{q}$ and $\mathbf{r}$ are evaluated using the ray differential representation in Equation 7 along with t, which is the distance to the intersection point. In addition, since $w = 1 - u - v$, we have

$$\frac{\partial w}{\partial x} = -\frac{\partial u}{\partial x} - \frac{\partial v}{\partial x}, \quad (20)$$

and similarly for $\partial w / \partial y$.

Once the differentials of (u, v) have been computed, they can be used to compute the corresponding texture-space differentials, which can be used in Equation 2, as

$$\begin{aligned} \frac{\partial s}{\partial x} &= w\left(\frac{\partial u}{\partial x}g_{1x} + \frac{\partial v}{\partial x}g_{2x}\right), \quad \frac{\partial t}{\partial x} = h\left(\frac{\partial u}{\partial x}g_{1y} + \frac{\partial v}{\partial x}g_{2y}\right), \\ \frac{\partial s}{\partial y} &= w\left(\frac{\partial u}{\partial y}g_{1x} + \frac{\partial v}{\partial y}g_{2x}\right), \quad \frac{\partial t}{\partial y} = h\left(\frac{\partial u}{\partial y}g_{1y} + \frac{\partial v}{\partial y}g_{2y}\right), \end{aligned} \quad (21)$$

where $w \times h$ is the texture resolution and $\mathbf{g}_1 = (g_{1x}, g_{1y}) = T_1 - T_0$ and $\mathbf{g}_2 = (g_{2x}, g_{2y}) = T_2 - T_0$ are the differences of texture coordinates between neighboring vertices. Similarly, differentials for the ray origin O' of a subsequent reflection/refraction ray can be computed as

$$\frac{\partial O'}{\partial(x,y)} = \frac{\partial u}{\partial(x,y)}\mathbf{e}_1 + \frac{\partial v}{\partial(x,y)}\mathbf{e}_2. \quad (22)$$

We have seen slightly better performance using this method compared to the traditional implementation following Igehy's work [9].

3.3 Ray Differentials with the G-Buffer

For real-time ray tracing, it is not uncommon to render the eye rays using rasterization into a G-buffer. When combining ray differentials [9] with a G-buffer, the ray differential for the eye rays can be created as usual, but the interaction at the first hit must use the content from the G-buffer, since the original geometry is not available at that time. Here, we present one method using the G-buffer, which we assume has been created with normals $\widehat{\mathbf{n}}$ and with distances t from the camera to the first hit point (or alternatively the world-space position). We describe how the ray differential is set up when shooting the first reflection ray from the position in the G-buffer.

The idea of this method is simply to access the G-buffer to the right and above the current pixel and create a ray differential from these values. The normals and the distances t, for the current pixel (x, y) and for the neighbors $(x+1, y)$ and $(x, y+1)$, are read out from the G-buffer. Let us denote these by $\widehat{\mathbf{n}}_{0:0}$ for the current pixel, $\widehat{\mathbf{n}}_{+1:0}$ for the pixel to the right, and $\widehat{\mathbf{n}}_{0:+1}$ for the pixel above, and similarly for other variables. The eye ray directions $\widehat{\mathbf{e}}$ for these neighbors are computed next. At this point, we can compute the ray differential of the ray origin at the first hit as

$$\frac{\partial O}{\partial x} = t_{+1:0}\widehat{\mathbf{e}}_{+1:0} - t_{0:0}\widehat{\mathbf{e}}_{0:0}, \tag{23}$$

and similarly for $\partial O/\partial y$. The ray differential direction is computed as

$$\frac{\partial \widehat{\mathbf{d}}}{\partial x} = \mathbf{r}(\widehat{\mathbf{e}}_{+1:0}, \widehat{\mathbf{n}}_{+1:0}) - \mathbf{r}(\widehat{\mathbf{e}}_{0:0}, \widehat{\mathbf{n}}_{0:0}), \tag{24}$$

where $\mathbf{r}$ is the shader function `reflect()`. Similar computations are done for $\partial\widehat{\mathbf{d}}/\partial y$. We now have all components of the ray differential, $\{\partial O/\partial x, \partial O/\partial y, \partial\widehat{\mathbf{d}}/\partial x, \partial\widehat{\mathbf{d}}/\partial y\}$, which means that ray tracing with ray differentials can commence from the first hit.

The method above is fast, but sometimes you hit different surfaces when comparing to the pixel to the right and above. A simple improvement is to test if $|t_{+1:0} - t_{0:0}| > \epsilon$, where ϵ is a small number, and, if so, access the G-buffer at $-1{:}0$ instead and use the one with the smallest difference in t. The same approach is used for the y-direction. This method is a bit slower but gives substantially better results along depth discontinuities.

3.4 Ray Cones

One method for computing texture level of detail is based on tracing cones. This is quite similar to the method proposed by Amanatides [1], except that we use the method only for texture LOD and we derive the details on how to implement this, which are absent in previous work. The core idea is illustrated in Figure 3. When the texture LOD λ has been computed for a pixel, the texture sampler in the GPU is used to perform trilinear mipmapping [17].

In this section, we derive our approximation for texture LOD for ray tracing using ray cones. We start by deriving an approximation to screen-space mipmapping using cones and then extend that to handle recursive ray tracing with reflections. Ideally, we would like to handle all sorts of surface interactions, but we will concentrate on the cases shown in Figure 4. This excludes saddle points, which exist in hyperbolic paraboloids, for example.

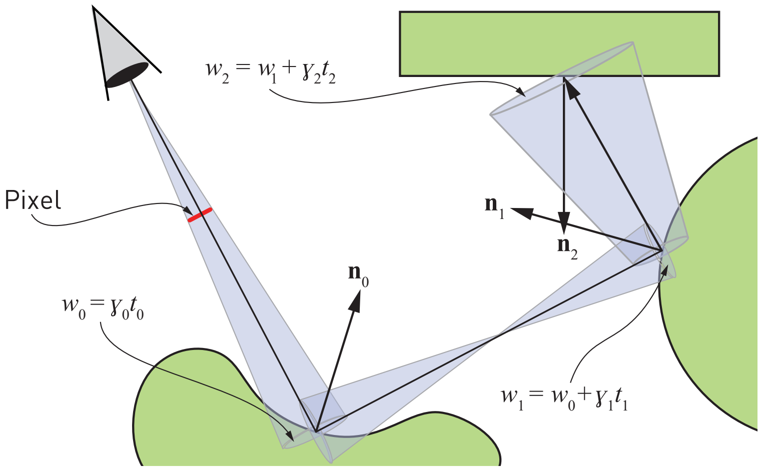


Figure 3: Illustration of how a cone is created through a pixel and how it is transported through the scene, growing and shrinking. Assuming that the rectangle is textured and the other objects are perfectly reflective, we will perform a texture lookup at the hit point on the rectangle using the width of the cone and the normal there, and a textured reflection would appear in the leftmost object. Computation of the cone widths w_i is explained in the text.

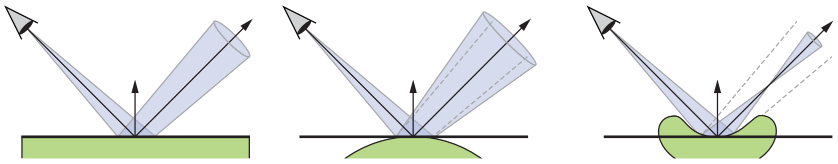


Figure 4: Illustrations of cones reflected in a planar (left), a convex (center), and a concave (right) surface. Note how the convex surface increases the angle of the cone, while the concave surface reduces it, until it becomes zero and starts growing again.

3.4.1 Screen Space

The geometrical setup for a cone through a pixel is shown in Figure 5. The footprint angle, also called *spread angle*, of a pixel is called α, $\mathbf{d}_0$ is the vector from the camera to the hit point, and $\mathbf{n}_0$ is the normal at the hitpoint. This cone is tracked through a pixel and the cone parameters are updated at each surface the center ray hits.

The footprint width will grow with distance. At the first hit point, the cone width will be $w_0 = 2||\mathbf{d}_0|| \tan(\alpha/2) \approx \alpha||\mathbf{d}_0||$, where the index 0 will be used to indicate the first hit. This index will be used extensively in the next subsection. We have used the small angle approximation, i.e., $\tan \alpha \approx \alpha$, in this expression. The footprint projected onto the plane at the hit point will also change in size due to the angle between $-\mathbf{d}_0$ and $\mathbf{n}_0$, denoted $[-\mathbf{d}_0, \mathbf{n}_0]$. Intuitively, the larger the angle, the more the ray can "see" of the triangle surface, and consequently, the LOD should increase, i.e., texel access should be done higher in the mipmap pyramid. Together

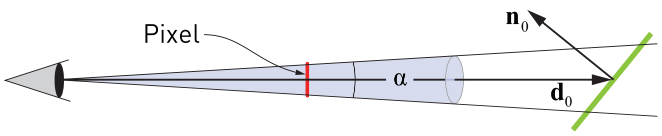


Figure 5: The geometrical setup of a cone through a pixel.

these factors form the approximated projected footprint as

$$\alpha \|\mathbf{d}_0\| \frac{1}{|\widehat{\mathbf{n}}_0 \cdot \widehat{\mathbf{d}}_0|}, \tag{25}$$

where $|\widehat{\mathbf{n}}_0 \cdot \widehat{\mathbf{d}}_0|$ models the square root of the projected area. The absolute value is there to handle frontfacing and backfacing triangles in the same way. When $[-\mathbf{d}_0, \mathbf{n}_0] = 0$, we have only the distance dependency, and as $[-\mathbf{d}_0, \mathbf{n}_0]$ grows, the projected footprint will get larger and larger toward infinity, when $[-\mathbf{d}_0, \mathbf{n}_0] \to \pi/2$.

If the value of the expression in Equation 25 doubles/halves, then we need to access one level higher/lower in the mipmap pyramid. Therefore, we use $\log_2$ on this term. Hence, a heuristic for texture LOD for the first hit, i.e., similar to what screen-space mipmapping produced by the GPU would yield, is

$$\lambda = \Delta_0 + \log_2 \left(\underbrace{\alpha \|\mathbf{d}_0\|}_{w_0} \frac{1}{|\widehat{\mathbf{n}}_0 \cdot \widehat{\mathbf{d}}_0|} \right), \tag{26}$$

where Δ_0 is described by Equations 3 and 5, i.e., using world-space vertices. Here, Δ_0 is the base texture LOD at the triangle seen through a pixel, i.e., without any reflections at this point. This term needs to be added to provide a reasonable base LOD when the triangle is located at $z = 1$. This term takes changes in triangle vertices and texture coordinates into account. For example, if a triangle becomes twice as large, then the base LOD will decrease by one. The other factors in Equation 26 are there to push the LOD up in the mipmap pyramid, if the distance or the incident angle increases.

3.4.2 Reflection

Our next step is to generalize the method in Section 3.4.1 to also handle reflections. The setup that we use for our derivation is shown in Figure 6, where we want to compute the width, w_1, of the footprint at the reflected hit point. Note that the angle β is a curvature measure (further described in Section 3.4.4) at the surface hit point, and it will influence how much the spread angle will grow or shrink due to different surface interactions. See Figure 4. We first note that

$$\tan\left(\frac{\alpha}{2} + \frac{\beta}{2}\right) = \frac{\frac{w_0}{2}}{t'} \quad \Longleftrightarrow \quad t' = \frac{w_0}{2\tan\left(\frac{\alpha}{2} + \frac{\beta}{2}\right)} \tag{27}$$

and

$$\tan\left(\frac{\alpha}{2} + \frac{\beta}{2}\right) = \frac{\frac{w_1}{2}}{t' + t_1} \quad \Longleftrightarrow \quad w_1 = 2(t' + t_1)\tan\left(\frac{\alpha}{2} + \frac{\beta}{2}\right). \tag{28}$$

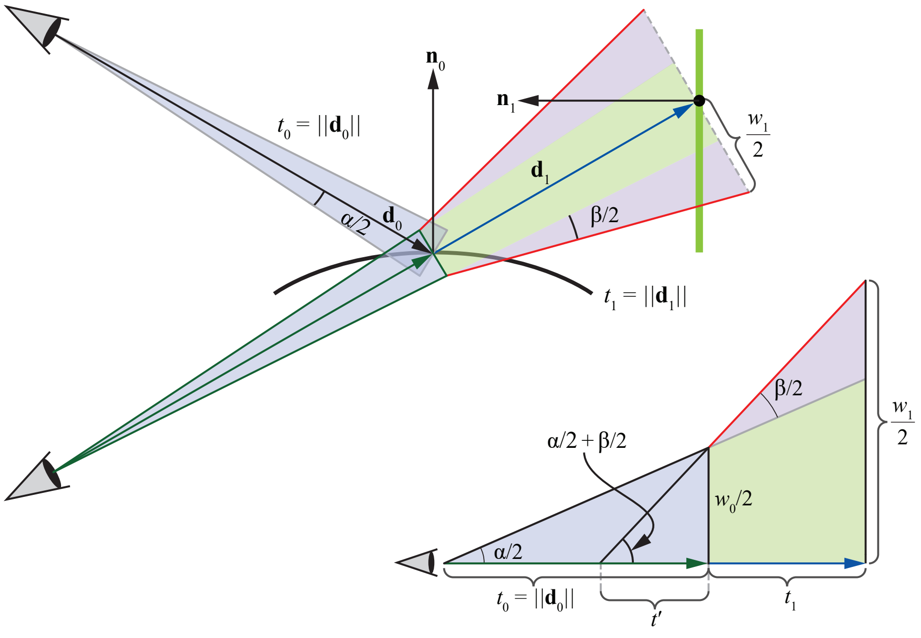


Figure 6: Top left: the geometrical setup for our computations for texture LOD for reflections, where the camera has been reflected in the plane of the first hit, which makes the green and blue rays collinear. The reflected hit point is the black circle on the green line. Bottom right: exaggerated view along the green and blue rays. We want to compute the footprint width w_1. Note that the surface spread angle β models how the cone footprint grows/shrinks due to the curvature of the surface, which in this case is convex and so grows the footprint ($\beta > 0$).

Next, we use the expression from Equation 27 for t', substitute it into Equation 28, and arrive at

$$\begin{aligned} w_1 &= 2\left(\frac{w_0}{2\tan\left(\frac{\alpha}{2}+\frac{\beta}{2}\right)}+t_1\right)\tan\left(\frac{\alpha}{2}+\frac{\beta}{2}\right) \\ &= w_0 + 2t_1\tan\left(\frac{\alpha}{2}+\frac{\beta}{2}\right) \approx w_0 + (\alpha+\beta)t_1, \end{aligned} \tag{29}$$

where we have used the small angle approximation $\tan\alpha \approx \alpha$ in the last step. Intuitively, this expression makes sense because $w_0 \approx \alpha||\mathbf{d}_0||$ makes the footprint grow with the distance from the eye to the first hit times the size α of a pixel, and the second term models the growth from the first hit to the second hit, which depends on the distance t_1 (from first to second hit) and the angle $\alpha + \beta$.

3.4.3 Pixel Spread Angle

In this subsection, we present a simple method to compute the spread angle α of a pixel, i.e., for primary rays. The angle from the camera to a pixel varies over the screen, but we have chosen to use a single value as an approximation for all pixels,

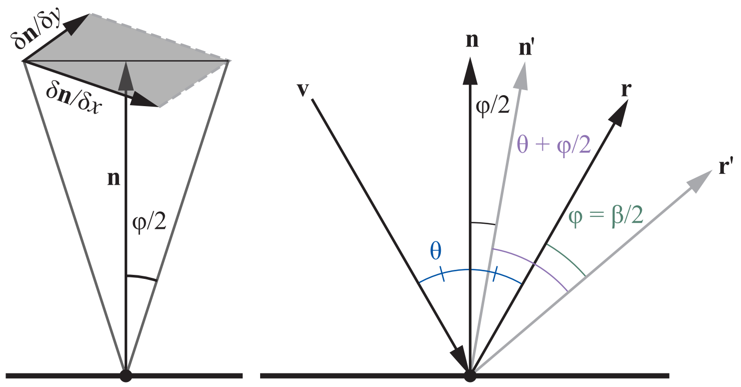


Figure 7: Left: the geometry involved in computing ϕ. Right: the view vector $\mathbf{v}$ is reflected around the normal $\mathbf{n}$, which generates $\mathbf{r}$. If $\mathbf{n}$ is perturbed by $\phi/2$ into $\mathbf{n}'$, we get another reflection vector $\mathbf{r}'$. Since $[-\mathbf{v}, \mathbf{n}'] = \theta + \phi/2$, we have $[\mathbf{r}', \mathbf{n}'] = \theta + \phi/2$, which means that the angle $[\mathbf{r}, \mathbf{r}'] = \phi$, i.e., is twice as large as $[\mathbf{n}, \mathbf{n}'] = \phi/2$.

i.e., we trade a bit of accuracy for faster computation. This angle α is computed as

$$\alpha = \arctan\left(\frac{2\tan\left(\frac{\psi}{2}\right)}{H}\right), \tag{30}$$

where ψ is the vertical field of view and H is the height of the image in pixels. Note that α is the angle to the center pixel.

While there are more accurate ways to compute the pixel spread angle, we use the technique above because it generates good results and we have not seen any discrepancies in the periphery. In extreme situations, e.g., for virtual reality, one may want to use a more complex approach, and for foveated renderers with eye tracking [12], one may wish to use a larger α in the periphery.

3.4.4 Surface Spread Angle for Reflections

Figure 4 illustrates reflection interactions at different types of geometry: planar, convex, and concave. In addition, Figure 6 illustrates the surface spread angle β, which will be zero for planar reflections, greater than zero for convex reflections, and less than zero for concave reflections. Intuitively, β models the extra spread induced by the curvature at the hit point. In general, the two principal curvatures [4] at the hit point or the radius of the mean curvature normal would be better to model this spread. Instead, we have opted for a simpler and faster method, one that uses only a single number β to indicate curvature.

If primary visibility is rasterized, then the G-buffer can be used to compute the surface spread angle. This is the approach that we take here, though there are likely other methods that could work. The normal $\mathbf{n}$ and the position P of the fragment are both stored in world space, and we use `ddx` and `ddy` (in HLSL syntax) to obtain their differentials. A differential of P in x is denoted $\partial P/\partial x$.

The left part of Figure 7 shows the geometry involved in the first computations for β. From the figure we can see that

$$\phi = 2\arctan\left(\frac{1}{2}\left\|\frac{\partial \mathbf{n}}{\partial x} + \frac{\partial \mathbf{n}}{\partial y}\right\|\right) \approx \left\|\frac{\partial \mathbf{n}}{\partial x} + \frac{\partial \mathbf{n}}{\partial y}\right\|. \tag{31}$$

An angular change in the normal, in our case $\phi/2$, results in change in the reflected vector, which is twice as large [16]; this is illustrated to the right in Figure 7. This means that $\beta = 2\phi$. We also add two additional user constants k_1 and k_2 for β and a sign factor s (all of which will be described below), resulting in $\beta = 2k_1 s\phi + k_2$, with default values $k_1 = 1$ and $k_2 = 0$. In summary, we have

$$\beta = 2k_1 s\phi + k_2 \approx 2k_1 s\sqrt{\frac{\partial \mathbf{n}}{\partial x}\cdot\frac{\partial \mathbf{n}}{\partial x} + \frac{\partial \mathbf{n}}{\partial y}\cdot\frac{\partial \mathbf{n}}{\partial y}} + k_2. \tag{32}$$

A positive β indicates a convex surface, while a negative value would indicate a concave surface region. Note that ϕ is always positive. So, depending on the type of surface, the s factor can switch the sign of β. We compute s as

$$s = \texttt{sign}\left(\frac{\partial P}{\partial x}\cdot\frac{\partial \mathbf{n}}{\partial x} + \frac{\partial P}{\partial y}\cdot\frac{\partial \mathbf{n}}{\partial y}\right), \tag{33}$$

where `sign` returns 1 if the argument is greater than zero and -1 otherwise. The rationale behind this operation is that $\partial P/\partial x$ and $\partial \mathbf{n}/\partial x$ (and similarly for y) will have approximately the same direction when the local geometry is convex (positive dot product) and approximately opposite directions when it is concave (negative dot product). Note that some surfaces, such as a hyperbolic paraboloid, are both concave and convex in all points on the surface. In these cases, we have found that it is better to just use $s = 1$. If a glossy appearance is desired, the values of k_1 and k_2 can be increased. For planar surfaces, ϕ will be 0, which means that k_1 does not have any effect. Instead, the term k_2 can be used.

3.4.5 Generalization

Let i denote the enumerated hit point along a ray path, starting at 0. That is, the first hit is enumerated by 0, the second by 1, and so on. All our terms for texture LOD for the ith hit point are then put together as

$$\lambda_i = \Delta_i + \log_2\left(|w_i| \cdot \left|\frac{1}{\widehat{\mathbf{n}}_i \cdot \widehat{\mathbf{d}}_i}\right|\right) = \underbrace{\Delta_i}_{\text{Eqn. 3}} + \underbrace{\log_2 |w_i|}_{\text{distance}} - \underbrace{\log_2 \left|\widehat{\mathbf{n}}_i \cdot \widehat{\mathbf{d}}_i\right|}_{\text{normal}}, \tag{34}$$

and as can be seen, this is similar to Equation 26, with both a distance and a normal dependency. Refer to Figure 6 for the variables and recall that $\mathbf{n}_i$ is the normal at the surface at the ith hit point and $\mathbf{d}_i$ is the vector to the ith hit point from the previous hit point. The base triangle LOD, Δ_i, now has a subscript i to indicate that it is the base LOD of the triangle at the ith hit point that should be used. Similar to before, $\widehat{\mathbf{d}}_i$ means a normalized direction of $\mathbf{d}_i$. Note that we have added two absolute value functions in Equation 34. The absolute value for the distance term is there because β can be negative, e.g., for concave surface points (see the right part of Figure 4). The absolute value for the normal term is there to handle backfacing triangles in a consistent manner.

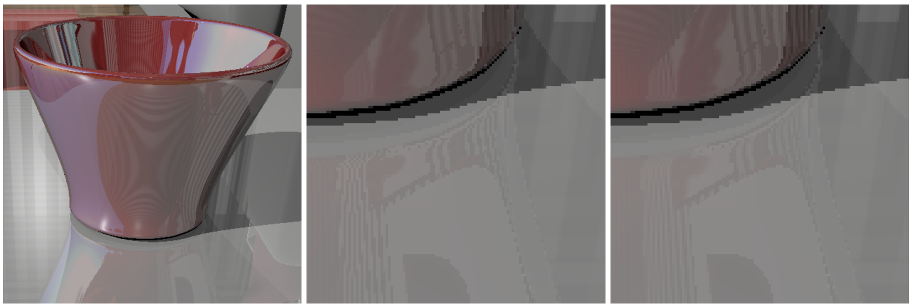


Figure 8: Zooming in on the base of a vase reflected in a table top shows that the ray cones method is weaker than the method based on ray differentials in areas of recursive reflections. In the lower part of the ray cones image, there is a substantial amount of aliasing, which is caused by the fact that, in our implementation, the method assumes that all surfaces beyond the first hit are planar.

Note that $w_0 = \alpha t_0 = \gamma_0 t_0$ and $w_1 = \alpha t_0 + (\alpha + \beta_0) t_1 = w_0 + \gamma_1 t_1$, where we have introduced $\gamma_0 = \alpha$ and $\gamma_1 = \alpha + \beta_0$, and β_0 is the surface spread angle at the first hit point. Hence, Equation 34 handles recursion, which we describe with pseudocode in Section 6, and in general it holds that

$$w_i = w_{i-1} + \gamma_i t_i, \tag{35}$$

where $\gamma_i = \gamma_{i-1} + \beta_{i-1}$. This is illustrated in Figure 3.

4 Implementation

We have implemented the ray cones and ray differentials techniques on top of Falcor [2] with DirectX 12 and DXR. For the texture lookups in ray cones, we compute λ_i according to Equation 34 and 35 and feed it into the `SampleLevel()` function of the texture.

Since rasterization is highly optimized for rendering primary visibility, where all rays share a single origin, we always use a G-buffer pass for ray cones and for the ray differentials method in Section 3.3. When a G-buffer is used, ray tracing commences from the first hit described by the G-buffer. As a consequence, texturing is done using the GPU's texturing units for the first hits and so, using the methods in this chapter, λ is computed only after that. For ray cones, β_i is computed using the G-buffer differentials from rasterization, which implies that there is a curvature estimate β_0 at only the first hit point. In our current implementation, we use $\beta_i = 0$ for $i > 0$. This means that beyond the first hit point, all interactions are assumed to be planar. This is not correct but gives reasonable results, and the first hit is likely the most important. However, when recursive textured reflections are apparent, this approximation can generate errors, as shown in Figure 8.

Next, we discuss the precision of the ray cones method. The data that needs to be sent with each ray is one float for w_i and one for γ_i. We have experimented with both `fp32` and `fp16` precision for β (in the G-buffer), w_i, and γ_i, and we conclude that 16-bit precision gives good quality in our use cases. In a hyperbolic paraboloid scene, we could not visually detect any differences, and the maximum error was a pixel component difference of five (out of 255). Depending on the application, textures, and scene geometry, it could be worthwhile to use `fp16`, especially when G-buffer storage and ray payload need to be reduced. Similarly, errors induced by using the small angle approximation ($\tan(\alpha) \approx \alpha$) for β resulted in nothing that was detectable by visual inspection. With per-pixel image differences, we could see another set of errors sparsely spread over the surface, with a maximum pixel component difference of five. This is another trade-off to be made.

The per-triangle Δ (Equation 3) can be computed in advance for static models and stored in a buffer that is accessed in the shader. However, we found that it equally fast to recompute Δ each time a closest hit on a triangle is found. Hence, the ray cones method handles animated models and there are no major extra costs for handling several texture coordinate layers per triangle. Note that $|\widehat{\mathbf{n}}_i \cdot \widehat{\mathbf{d}}_i|$ in Equation 34 will approach $+0.0$ when the angle between these vectors approaches $\pi/2$ radians. This does not turn out to be a problem, as using IEEE standard 754 for floating-point mathematics, we have $\log_2(+0.0) = $ `-inf`, which makes $\lambda =$ `inf`. This in turn will force the trilinear lookup to access the top level of the mipmap hierarchy, which is expected when the angle is $\pi/2$ radians.

Our ray differentials implementation follows the description of Igehy [9] fairly well. However, we use the λ computation in Equations 1 and 2, unless otherwise mentioned, and the methods in Sections 3.2.1 and 3.2.2. For ray differentials, each ray needs 12 floats of storage, which is rather substantial.

5 Comparison and Results

The methods that we use in this section are:

- GROUNDTRUTH: a ground-truth rendering (ray traced with 1,024 samples per pixel).
- MIP0: bilinearly filtered mip level 0.
- RAYCONES: ray cones method (Section 3.4).
- RAYDIFFS GB: ray differentials with the G-buffer (Section 3.3).
- RAYDIFFS RT: our implementation of ray differentials with ray tracing [9].
- RAYDIFFS PBRT: ray differentials implementation in the *pbrt* renderer [14].

Note that MIP0, RAYCONES, and RAYDIFFS GB always use a G-buffer for primary visibility, while RAYDIFFS RT and RAYDIFFS PBRT use ray tracing. For all our performance results, an NVIDIA RTX 2080 Ti (Turing) was used with driver 416.16.

To verify that our implementation of ray differentials [9] is correct, we compared it to the implementation in the *pbrt* renderer [14]. To visualize the resulting mip level of a filtered textured lookup, we use a specialized *rainbow* texture, shown in Figure 9. Each mip level is set to a single color. We rendered a reflective hyperbolic paraboloid in a diffuse room in Figure 10. This means that the room only shows the mip level as seen from the eye, while the hyperbolic paraboloid shows the mip level of the reflection, which has some consequences discussed in the caption of Figure 10. It

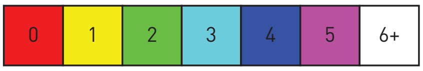
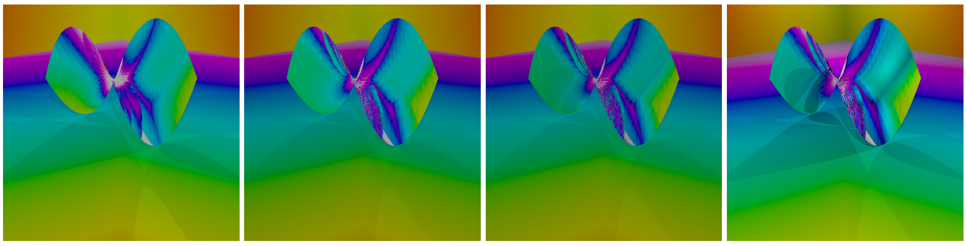


Figure 9: The mip level colors in the rainbow texture are selected according to this image, i.e., the bottom mip level (level 0) is red, level 1 is yellow, and so on. Mip levels 6 and above are white.



RayCones RayDiffs GB RayDiffs RT RayDiffs PBRT

Figure 10: Visualization of mipmap level, where red is level 0, yellow is level 1, and so on, as defined in Figure 9. Both RayCones and RayDiffs GB use a G-buffer pass for the eye rays, and so we used the texture derivatives generated in that pass to compute the mipmap level using the same formula as used by *pbrt* in order to get a reasonable match on the floor. Since the hyperbolic paraboloid is both concave and convex in all points, we used $s = 1$ in Equation 32. Note that the shading overlayed on top of the "rainbow" colors does not match perfectly, but the focus should be on the actual colors. The three images to the right match quite well, while RayCones is a bit different, in particular in the recursive reflections. This difference is to be expected, since reflections are assumed to be planar after the first bounce for this method.

is noteworthy that one can see the triangular structure of the hyperbolic paraboloid surface in these images. The reason for this is that the differentials of barycentric coordinates are not continuous across shared triangle edges. This is also true for rasterization, which shows similar structures. As a consequence, this discontinuity generates noise in the recursive reflections, but it does not show up visually in the rendered images in our video.

Some further results are shown in Figure 11. We have chosen the hyperbolic paraboloid (top) and the bilinear patch (bottom) because they are saddle surfaces and are difficult for RayCones, since it is based on cones that handle only isotropic footprints. The semicylinders were also chosen because they are hard for RayCones to handle as the curvature is zero along the length of the cylinder and bends like a circle in the other direction. As a consequence, RayCones sometimes generates more blur compared to ray differentials. It is also clear that the GroundTruth images are substantially more sharp than the other methods, so there is much to improve on for filtered texturing. A consequence of this overblurring is that both the peak signal-to-noise ratio (PSNR) and structural similarity index (SSIM) values are relatively poor. For the hyperbolic paraboloid, i.e., the top row in Figure 11, the PSNR against GroundTruth is 25.0, 26.7, and 31.0 dB for Mip0, RayCones, and RayDiffs RT, respectively. PSNR for Mip0 is lower as expected, but the numbers are low even for the other methods. This is because they produce more

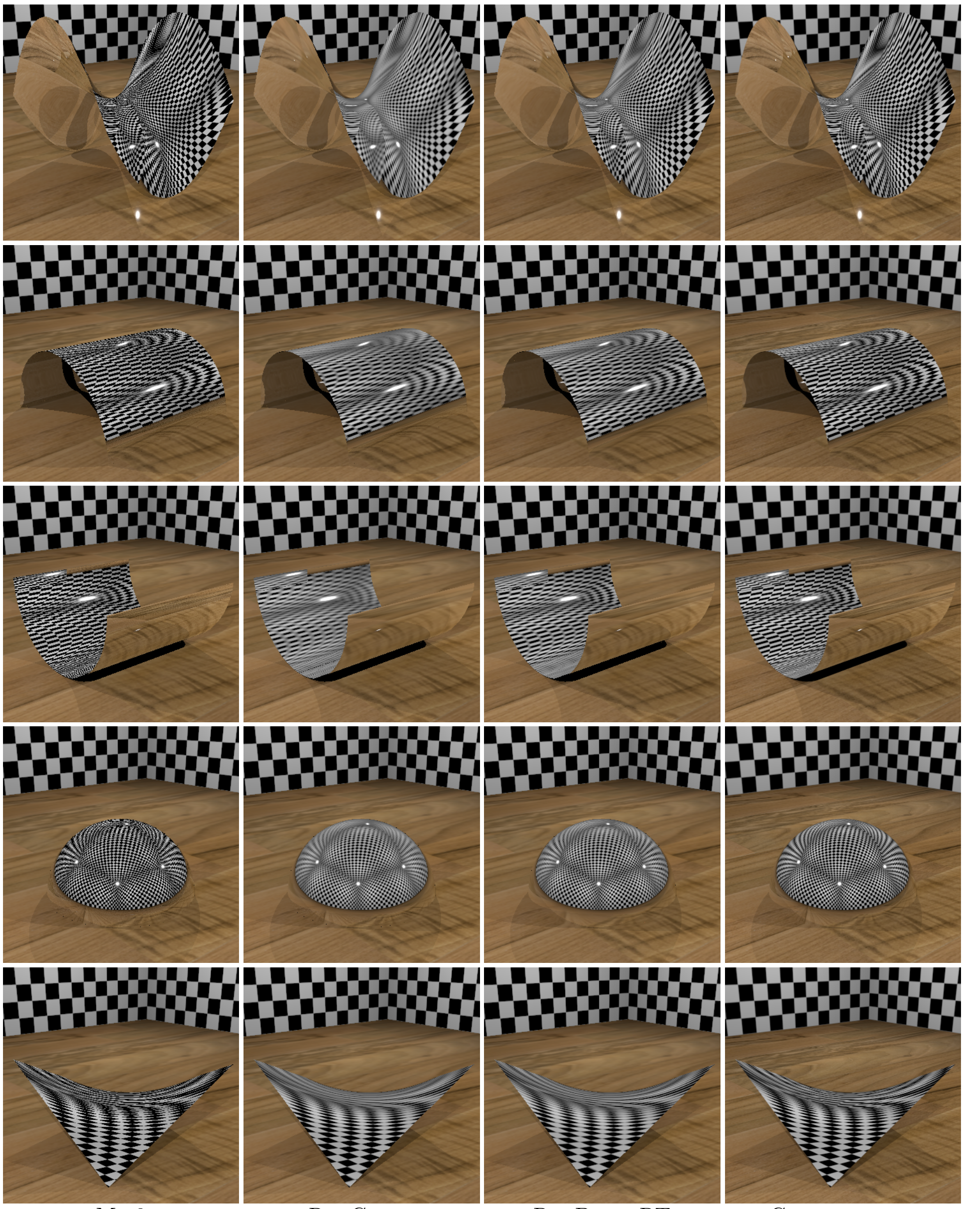


Figure 11: Comparison of textured reflections for different types of surfaces using different techniques. The GroundTruth images were rendered using 1,024 samples per pixel and by accessing mipmap level 0 for all texturing. For RayCones, we used the sign function in Equation 33.

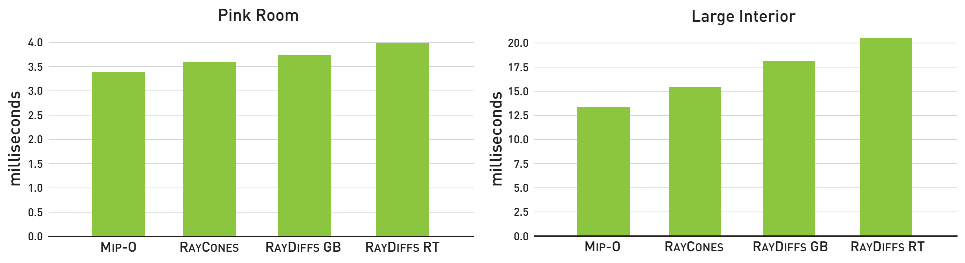


Figure 12: Performance impact of texture level of detail selection methods: Pink Room (left) and Large Interior (right). The smaller scene (Pink Room) is less susceptible to an extra filtering cost, the larger scene (Large Interior) more so. For both scenes, however, the performance impact of RayDiffs GB is about 2×, and RayDiffs RT about 3×, the impact of RayCones.

blur compared to Groundtruth. On the other hand, they alias substantially less than Mip0. The corresponding SSIM numbers are 0.95, 0.95, and 0.97, which convey a similar story.

While still images can reveal the amount of overblurring in an image fairly well, it is substantially harder to truthfully show still images that expose the amount of aliasing they contain. As a consequence, most of our results are shown in our accompanying video (available on this book's website), and we refer to this video in the following paragraph. We will write *mm:ss* to reference a particular time in the video, where *mm* is minutes and *ss* is seconds.

At *00:05* and *00:15* in our video, it is clear that RayCones produces images with substantially less aliasing, as expected, compared to Mip0, since the reflective object always uses mip level 0 for Mip0. At some distance, there is also a slight amount of temporal aliasing for RayCones, but even GPU rasterization can alias with mipmapping. The comparison between RayCones and Mip0 continues with a crop from a larger scene at *00:25*, where the striped wallpaper of the room generates a fair amount of aliasing for Mip0, while RayCones and RayDiffs RT fare much better.

We have measured the performance of the methods for two scenes: Pink Room and Large Interior. All renderings are done at a resolution of 3840 × 2160 pixels. To disregard warmup effects and other variability, we rendered the scenes through a camera path of 1,000 frames once, without measuring frame duration, and then through the same camera path again, while measuring. We repeated this procedure 100 times for each scene and collected the frame durations. For Mip0, the average frame time was 3.4 ms for Pink Room and 13.4 ms for Large Interior. In Figure 12, the average total frame times are shown for the two scenes, for Mip0, RayCones, RayDiffs GB, and RayDiffs RT. Pink Room is a fairly small scene, where the added complexity of texture level of detail computation shows up as a minor part of the total frame time, while for Large Interior—a significantly larger scene—this effect is more pronounced. For both scenes, however, the trend is quite clear: RayDiffs GB adds about 2× the cost and RayDiffs RT adds about 3× the cost of texture level of detail calculations compared to RayCones.

Our goal in this chapter is to provide some help in selecting a suitable texture filtering method for your real-time application by implementing and evaluating different methods and to adapt them to using a G-buffer, since that usually improves performance. When a sophisticated reflection filter is used to blur out the results or when many frames or samples are accumulated, the recommendation is to use the MIP0 method because it is faster and may give sufficient quality for that purpose. When nicely filtered reflections are required and ray storage and instruction count need to be minimized, we recommend RAYCONES. However, curvature is not taken into account after the first hit, which might result in aliasing in deeper reflections. In these cases, we recommend one of the RAYDIFFS methods. For larger scenes, any type of texture filtering will likely help performance due to better texture cache hit ratios, as pointed out by Pharr [13]. When using ray tracing for eye rays, we have seen slight performance improvements when using texture filtering instead of accessing mip level 0. Experimenting further with this for larger scenes will be a productive avenue for future work.

6 Code

In this section, we show pseudocode that closely follows our current implementation of RAYCONES. First, we need a couple of structures:

```
struct RayCone
{
    float width;              // Called $w_i$ in the text
    float spreadAngle;        // Called $\gamma_i$ in the text
};

struct Ray
{
    float3 origin;
    float3 direction;
};

struct SurfaceHit
{
    float3 position;
    float3 normal;
    float  surfaceSpreadAngle; // Initialized according to Eq. 32
    float  distance;           // Distance to first hit
};
```

In the next pseudocode, we follow the general flow of DXR programs for ray tracing. We present a ray generation program and a closest hit program, but omit several other programs that do not add useful information in this context. The `TraceRay` function traverses a spatial data structure and finds the closest hit. The pseudocode handles recursive reflections.

```
void rayGenerationShader(SurfaceHit gbuffer)
{
    RayCone firstCone = computeRayConeFromGBuffer(gbuffer);
    Ray viewRay = getViewRay(pixel);
    Ray reflectedRay = computeReflectedRay(viewRay, gbuffer);
    TraceRay(closestHitProgram, reflectedRay, firstCone);
}

RayCone propagate(RayCone cone, float surfaceSpreadAngle, float hitT)
{
    RayCone newCone;
    newCone.width = cone.spreadAngle * hitT + cone.width;
    newCone.spreadAngle = cone.spreadAngle + surfaceSpreadAngle;
    return newCone;
}

RayCone computeRayConeFromGBuffer(SurfaceHit gbuffer)
{
    RayCone rc;
    rc.width = 0;                           // No width when ray cone starts
    rc.spreadAngle = pixelSpreadAngle(pixel); // Eq. 30
    // gbuffer.surfaceSpreadAngle holds a value generated by Eq. 32
    return propagate(rc, gbuffer.surfaceSpreadAngle, gbuffer.distance);
}

void closestHitShader(Ray ray, SurfaceHit surf, RayCone cone)
{
   // Propagate cone to second hit
   cone = cone.propagate(0, hitT); // Using 0 since no curvature
                                   // measure at second hit
    float lambda = computeTextureLOD(ray, surf, cone);
    float3 filteredColor = textureLookup(lambda);
    // use filteredColor for shading here
    if (isReflective)
    {
        Ray reflectedRay = computeReflectedRay(ray, surf);
        TraceRay(closestHitProgram, reflectedRay, cone);  // Recursion
    }
}

float computeTextureLOD(Ray ray, SurfaceHit surf, RayCone cone)
{
    // Eq. 34
    float lambda = getTriangleLODConstant();
    lambda += log2(abs(cone.width));
    lambda += 0.5 * log2(texture.width * texture.height);
    lambda -= log2(abs(dot(ray.direction, surf.normal)));
    return lambda;
}

float getTriangleLODConstant()
{
    float P_a = computeTriangleArea();                    // Eq. 5
    float T_a = computeTextureCoordsArea();               // Eq. 4
    return 0.5 * log2(T_a/P_a);                           // Eq. 3
}
```

Acknowledgments

Thanks to Jacob Munkberg and Jon Hasselgren for brainstorming help and comments.

References

[1] Amanatides, J. Ray Tracing with Cones. *Computer Graphics (SIGGRAPH) 18*, 3 (1984), 129–135.

[2] Benty, N., Yao, K.-H., Foley, T., Kaplanyan, A. S., Lavelle, C., Wyman, C., and Vijay, A. The Falcor Rendering Framework. https://github.com/NVIDIAGameWorks/Falcor, July 2017.

[3] Christensen, P., Fong, J., Shade, J., Wooten, W., Schubert, B., Kensler, A., Friedman, S., Kilpatrick, C., Ramshaw, C., Bannister, M., Rayner, B., Brouillat, J., and Liani, M. RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. *ACM Transactions on Graphics 37*, 3 (2018), 30:1–30:21.

[4] do Carmo, M. P. *Differential Geometry of Curves and Surfaces*. Prentice Hall Inc., 1976.

[5] Ewins, J. P., Waller, M. D., White, M., and Lister, P. F. MIP-Map Level Selection for Texture Mapping. *IEEE Transactions on Visualization and Computer Graphics 4*, 4 (1998), 317–329.

[6] Green, N., and Heckbert, P. S. Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter. *IEEE Computer Graphics and Applications 6*, 6 (1986), 21–27.

[7] Heckbert, P. S. Survey of Texture Mapping. *IEEE Computer Graphics and Applications 6*, 11 (1986), 56–67.

[8] Heckbert, P. S. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California, Berkeley, 1989.

[9] Igehy, H. Tracing Ray Differentials. In *Proceedings of SIGGRAPH* (1999), pp. 179–186.

[10] McCormack, J., Perry, R., Farkas, K. I., and Jouppi, N. P. Feline: Fast Elliptical Lines for Anisotropic Texture Mapping. In *Proceedings of SIGGRAPH* (1999), pp. 243–250.

[11] Möller, T., and Trumbore, B. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools 2*, 1 (1997), 21–28.

[12] Patney, A., Salvi, M., Kim, J., Kaplanyan, A., Wyman, C., Benty, N., Luebke, D., and Lefohn, A. Towards Foveated Rendering for Gaze-Tracked Virtual Reality. *ACM Transactions on Graphics 35*, 6 (2016), 179:1–179:12.

[13] Pharr, M. Swallowing the Elephant (Part 5). Matt Pharr's blog, https://pharr.org/matt/blog/2018/07/16/moana-island-pbrt-5.html, July 16 2018.

[14] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*, third ed. Morgan Kaufmann, 2016.

[15] Segal, M., and Akeley, K. The OpenGL Graphics System: A Specification (Version 4.5). Khronos Group documentation, 2016.

[16] Voorhies, D., and Foran, J. Reflection Vector Shading Hardware. In *Proceedings of SIGGRAPH* (1994), pp. 163–166.

[17] Williams, L. Pyramidal Parametrics. *Computer Graphics (SIGGRAPH) 17*, 3 (1983), 1–11.