Ray Cones for Texture Filtering

David Cline, Nvidia, May 2023

This note describes how to use ray cones to drive texture filtering in a ray tracer or path tracer. The OTK implementation is based on the work of Akenine-Möller et al. [1, 2, 3], Boksansky et al. [4], and Qin et al. [5]. It can be found in ray_cone.h, and is used by the OTK RayCones sample.

Definition

A *ray cone* consists of two numbers (angle α, and width w) that track the area that a ray is trying to sample as it propagates. A positive angle means the cone is diverging, and a negative angle means it is converging.

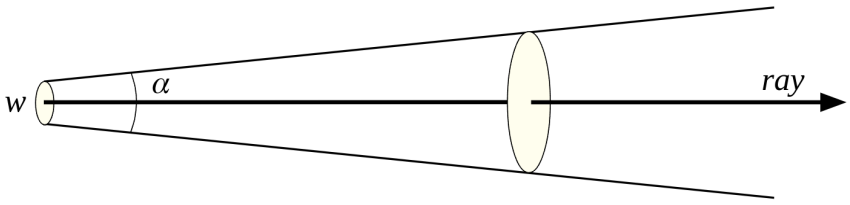


Figure 1: Ray Cone.

The advantages of ray cones over ray differentials are lower memory cost and simplified calculations. The main drawback is that the cones are always circular, so anisotropy is not modeled during propagation. However, anisotropy is handled at surfaces by projecting the cone onto an elliptical footprint. Thus, first hit texture lookups will be very similar to those derived using ray differentials.

Typically, the angle and width of a ray cone are stored as floats, but Akenine-Möller et al. [1] point out, and we have verified, that bfloat16 is sufficient. RayCone.h provides methods to pack and unpack a ray cone to bfloat16 format.

Ray Cone Propagation

Ray cones use the small angle approximation followed by paraxial ray theory, which makes the simplifications that for any angle *a:*

$$sin(a) = tan(a) = a, \;\; cos(a) = 1$$

Based on the small angle approximation, a ray cone propagates through some distance d by changing its width as follows:

$$w' = w + \alpha d$$

In the OTK implementation, if the width and angle of a cone are both negative, the signs of both are reversed, since the cone is actually diverging.

Initializing a Ray Cone

To begin ray or path tracing, a ray cone must be initialized based on the camera model.
Let (U, V, W) be three vectors describing the camera's reference frame, where W is the vector from the eye point to the center of the view rectangle, and $||U||$ and $||V||$ are half the width and height of the view rectangle:

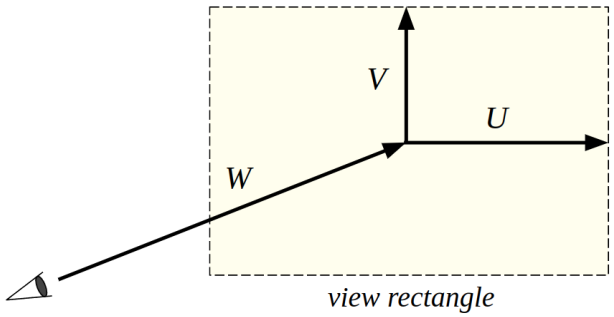


Figure 2: Camera frame used for ray cone initialization.

Pinhole Camera

For a pinhole camera, the goal is to initialize the ray cone so that its width is 0 at the eye point and it expands to the width of a pixel at the image plane. Let (w_{img}, h_{img}) be the image resolution in pixels, and D be the direction of a camera ray. The ray cone for a pinhole camera is then:

$$\alpha = 2 \min(||U||\, D \bullet W / (w_{img}\, W \bullet W),\ ||V||\, D \bullet W / (h_{img}\, W \bullet W))$$
$$w = 0$$

Akenine-Möller et al. [3] use the pixel diagonal instead of width, which reduces aliasing, but introduces some slight blurring. We note here that our implementation is tuned for final frame rendering, so we favor some aliasing over blurring.

Thin Lens Camera

The ray cone for a thin lens camera starts with the lens width w_{lens}, and converges to zero at the image plane:

$$\alpha = -w_{lens}\ \mathrm{D} \bullet \mathrm{W} \ /\ W \bullet W$$
$$w = w_{lens}$$

Orthographic Camera

In an orthographic camera, the initial angle is zero since eye rays do not converge or diverge, and the cone width is the pixel width on the image plane:

$$\alpha = 0$$
$$w = 2\ \min(\ ||U|| / w_{img},\ ||V|| / h_{img}\ )$$

Tracking Two Cones

A problem that occurs with ray cones is that when they converge they produce regions where the cone width goes to zero. These "infinite detail" contours may not be a big issue for fully loaded textures, since they just direct the renderer to use mip level zero. However, they can be a real nuisance if used to direct texture filtering.

Boksansky et al. [4] and Qin et al. [5] use the envelope of two cones (track two cones and keep the larger width) to deal with the infinite detail problem for thin lens cameras. This is an effective solution both for primary and for secondary hits, and we use it for all three camera setups we support, not just thin lens. They ray cones for the different setups are as follows:

Camera Setup	**First Cone**	**Second Cone**
Thin Lens	Thin Lens	Pinhole
Pinhole	Pinhole	Thin lens with w_{lens} = 1 pixel width on image plane
Orthographic	Orthographic	Thin lens with w_{lens} = 1 pixel width on image plane

Using two cones eliminates most of the infinite contours and works well in most cases. However, the extra ray cone for a pinhole camera can cause some texture blurring near the camera, and the extra cone for an orthographic camera leads to blurring for points far beyond the image plane. In the OTK RayCones sample this is fixed by adding special cases to ignore the extra ray cone on a first hit for these projections.

Computing Curvature

When a ray cone reflects off a surface or refracts through it, the surface curvature affects the resulting angle. Curvature can be thought of as the deflection of the normal per unit distance traveled on the surface. On an polygon edge with vertices A and B, and normals N_a and N_b the curvature is estimated by calculating this normal deflection:

$$k_{ab} = (N_b\text{-}N_a)\bullet(B\text{-}A) \;/\; (B\text{-}A)\bullet(B\text{-}A)$$

To compute curvature on a triangle, we use the minimum magnitude edge curvature (or zero if edge curvatures have mixed signs). Other work favors using edge curvature, but this can lead to some blurring. Calculating curvature per triangle leads to discontinuities, but we have not noticed artifacts from this. Akenine-Möller et al. [3] use a more sophisticated estimate for first hits.

Reflecting a Ray Cone

To reflect a cone from a surface with curvature k, add double the curvature angle ($k|w|$):

$$\alpha' = \alpha + 2k|w|$$

We also tried expanding the curvature angle based on the incident angle of the ray, but this led to visible blurring, so it was not kept in the final implementation.

Refracting a Ray Cone

When a ray passes through a transparent surface, it bends according to Snell's law. Simplifying Snell's law by the small angle approximation gives the expression $\theta_{in} = (n_o/n_i)\,\theta_{out}$, where n_o and n_i are the refractive indices outside and inside the surface. Referencing figure 3, if a ray cone refracts through a surface, the sum of the current cone angle and curvature angle get compressed or expanded by (n_o/n_i):

$$\alpha' + k\,|w| = (n_o/n_i)\,(\alpha + k\,|w|)$$

Solving for α' gives

$$\alpha' = (n_o/n_i)\,(\alpha + k|w|) - k|w|$$

Boksansky et al. [4] present more accurate formulas for ray cone refraction that require computing refracted rays for the cone edges. Figure 3 shows the geometric setup for ray cone reflection and refraction.

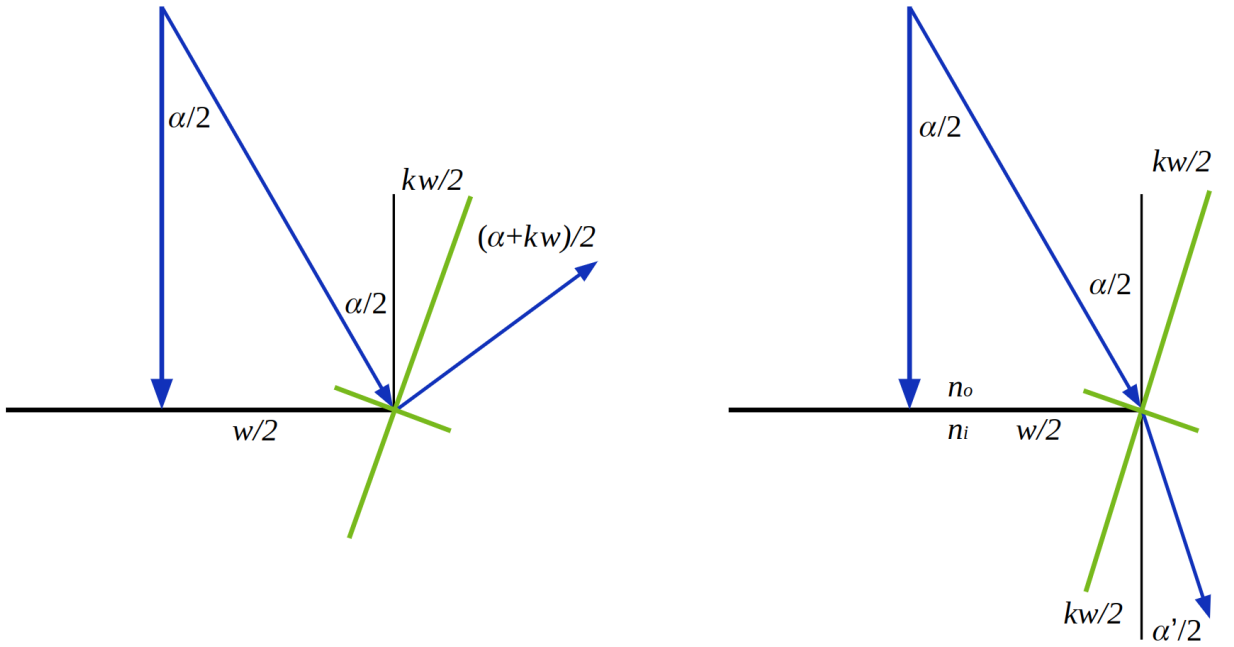


Figure 3: Geometric setup for reflecting and refracting ray cones. The diagrams show half a ray cone (blue), so the final angle is twice what is in the picture. In a reflection (left figure), the normal at the ray cone edge is rotated by angle $k|w|/2$ (shown in green), where k is the surface curvature and w is the cone width. The reflected angle is $\alpha' = 2(k|w|/2+(\alpha+k|w|)/2) = \alpha+2k|w|$. In a refraction (right figure), the normal at the cone edge is again rotated by angle $k|w|/2$. By the small angle approximation of Snell's law $k|w|/2+\alpha'/2 = (n_o/n_i)(\alpha/2+k|w|/2)$. Solving for α' yields $(n_o/n_i)\,(\alpha + k|w|) - k|w|$.

BSDF scattering

When a ray scatters off a surface, the ray cone should expand based on the surface properties. Akenine-Möller et al. [3] use the roughness of the BSDF to derive an angular spread, but many BSDF representations do not have an explicit roughness parameter, or they mix diffuse and glossy terms. To support all surface types, we opt for a simple heuristic that only requires evaluating the BSDF: first, set the angular spread for diffuse surfaces to be a constant, $s_{max} = 0.25$

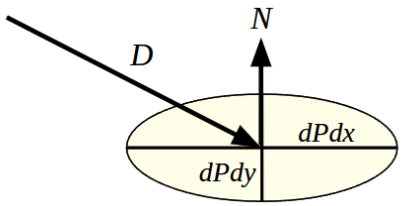
radians in our case. Then, scale the angular spread for smoother surfaces by the ratio of the BSDF value F_s to $1/\pi$ (the BSDF value for an ideal diffuse surface), giving

$$\alpha' = \alpha + s_{max} \min(1, 1/(\pi F_s))$$

This allows the angle to spread out more for directions away from the specular peak on glossy surfaces. The same heuristic works for reflection, transmission, and scattering in a participating medium, substituting the phase function value for the BSDF.

Projecting a Ray Cone to a Surface

To sample a texture from a ray cone, a renderer must extract texture gradients, *ddx* and *ddy*, to plug into a texturing function such as tex2DGrad. The first step in this process is to project the ray cone's circle to an elliptical footprint on the surface in world space. The axes of this ellipse, *dPdx* and *dPdy*, are found as follows

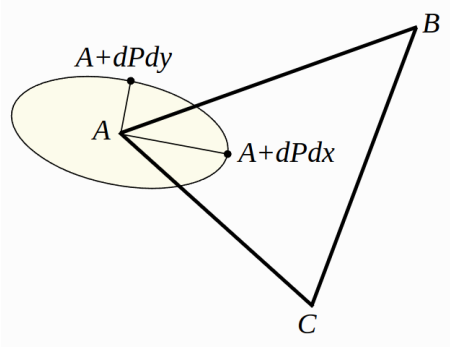


$$dPdx = \text{normalize}(D - (D \bullet N)\, N)\ (\ |w| \,/\, \max(\ |D \bullet N|,\ 1/\text{MAX\_ANISOTROPY}\ )\ )$$
$$\text{dPdy} = \text{normalize}(D \times N)\ |w|$$

where *D* is the ray direction, *N* is the surface normal, and MAX_ANISOTROPY is the maximum anisotropy supported by the texturing system (typically 8 or 16).

Texture Derivatives on a Triangle

Finding texture gradients *ddx* and *ddy* for a triangle with vertices (*A, B, C*), and texture coordinates (T_a, T_b, T_c) involves projecting *dPdx* and *dPdy* onto the triangle to get their barycentric offsets, which in turn are used to compute texture derivatives. The barycentric coordinates of vertex *A* are (1, 0, 0). Let (a_x, b_x, c_x) and (a_y, b_y, c_y) be the barycentric coordinates of point *A*+*dPdx* and *A*+*dPdy*. Then



$$ddx = (a_x - 1)\, T_a + b_x\, T_b + c_x\, T_c$$
$$ddy = (a_y - 1)\, T_a + b_y\, T_b + c_y\, T_c$$

Texture Derivatives on a General Surface

On non-triangular surfaces, *ddx* and *ddy* can be found by projecting *dPdx* and *dPdy* onto the world space texture derivatives of the surface, *dPds* and *dPdt*:

$$ddx = \{ (dPdx \bullet dPds) / (dPds \bullet dPds), (dPdx \bullet dPdt) / (dPdt \bullet dPdt) \}$$
$$ddy = \{ (dPdy \bullet dPds) / (dPds \bullet dPds), (dPdy \bullet dPdt) / (dPdt \bullet dPdt) \}$$

Intuitively, the x component of *ddx* increases when *dPdx* and *dPds* are aligned. It also increases if *dPdx* is longer, but decreases if *dPds* is longer.

Texture Derivatives on an Environment Map

An environment map is considered infinitely far away, so only the angle portion of the ray cone matters. When sampling environment maps, we mostly follow the derivations in Akenine-Möller et al. [2], and assume that direction-independent samples are sufficient, In a cube map all texels cover roughly the same solid angle, and in a latitude-longitude map, the texture should be pre-filtered in the x direction to account for compression at the poles.

For a cube map, each face covers $\pi/2$ radians in x and y, so the sample width should be $2\alpha/\pi$ radians, and the corresponding texture gradients are: $ddx = \{2\alpha/\pi, 0\}$, $ddy = \{0, 2\alpha/\pi\}$.

For a latitude-longitude map, the map spans 2π radians in the x direction but only π radians in y, so we set the texture gradients to: $ddx = \{\alpha/(2\pi), 0\}$, $ddy = \{0, \alpha/\pi\}$.

Examples

Figure 4 compares rendering with ray cones vs. a simple distance metric. The texture uses different colors on different mip levels to highlight which level is chosen. Ray cones choose proper mip levels on the reflected spheres, whereas the distance metric overestimates the mip levels needed for the reflections. The lower left corners of the images show the texture tiles that were loaded by the renderer. The ray cones rendering used 425 texture tiles, vs. 1564 for the distance metric.

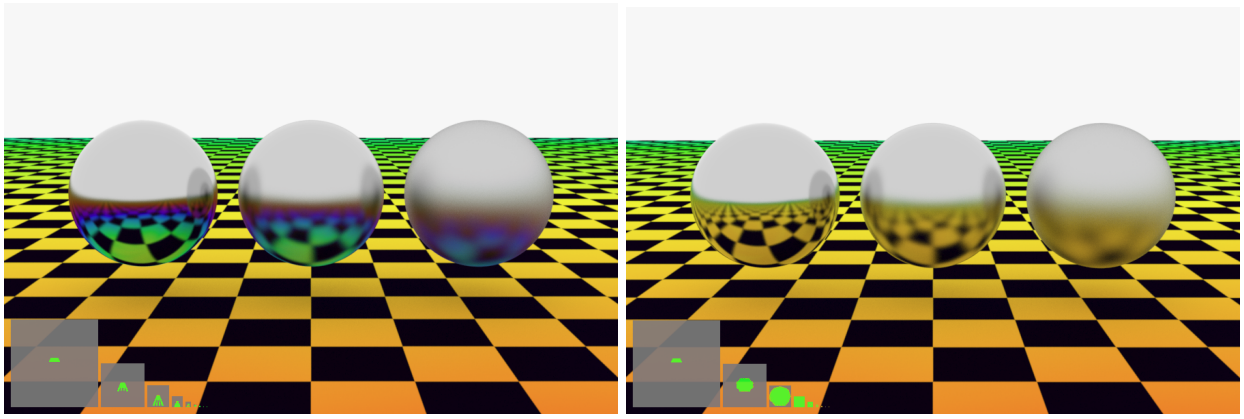


Figure 4: Convex reflectors. Ray cones (left) vs. distance metric (right).

Figure 5 shows a reflection from a concave mirror. The ray cones track the ray footprint with enough accuracy to resolve the magnified texture, while the distance metric underestimates the needed resolution, leading to texture blurring.

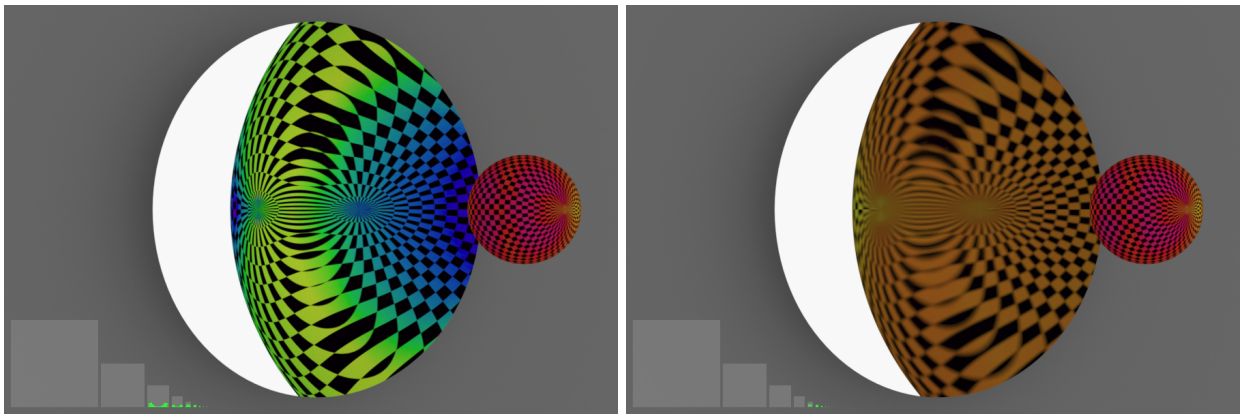


Figure 5: Concave reflectors. Ray cones (left) vs. distance metric (right).

Figure 6 shows the need to track two cones instead of one. In the single cone rendering (right), infinite precision contours show up, seen as a red stripe in the image. Beyond aliasing implications, detailed mip tiles are loaded needlessly in primary as well as secondary hits, taxing the texturing system. Tracking two cones (left) eliminates the infinite detail contours, and reduces texture usage more than 7 times (841 vs. 6180 tiles).

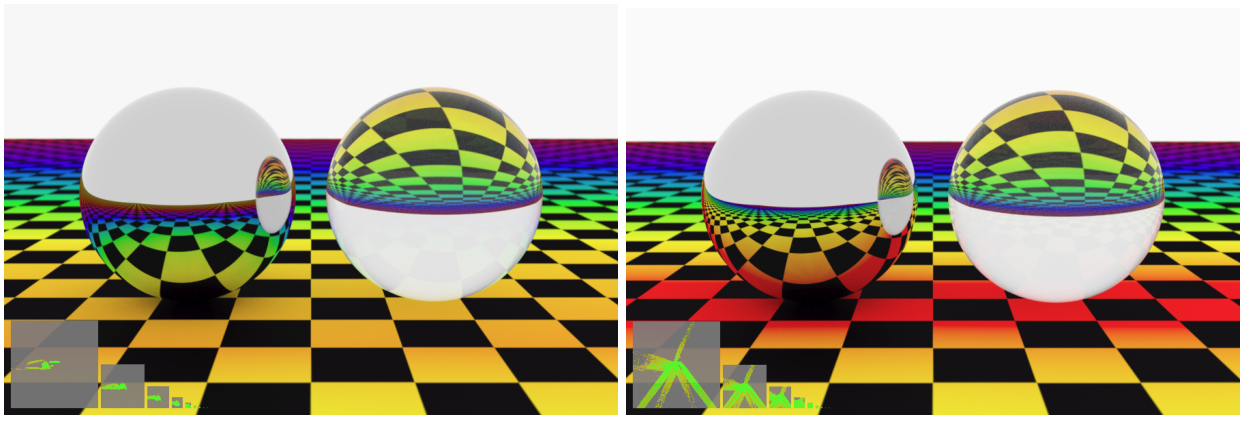


Figure 6: Tracking two cones (left), and a single cone (right).

The scene in figure 7 exhibits “negative curvature traps”. These occur when rays wander into a bounded region with negative curvature, such as inside a ball or vase. As a ray bounces around, the ray cones can converge, leading to detailed mip level requests that are usually unnecessary. The right image removes the negative curvature traps by lining the vases with a diffuse material, reducing texture tile loads from 1249 to 704.

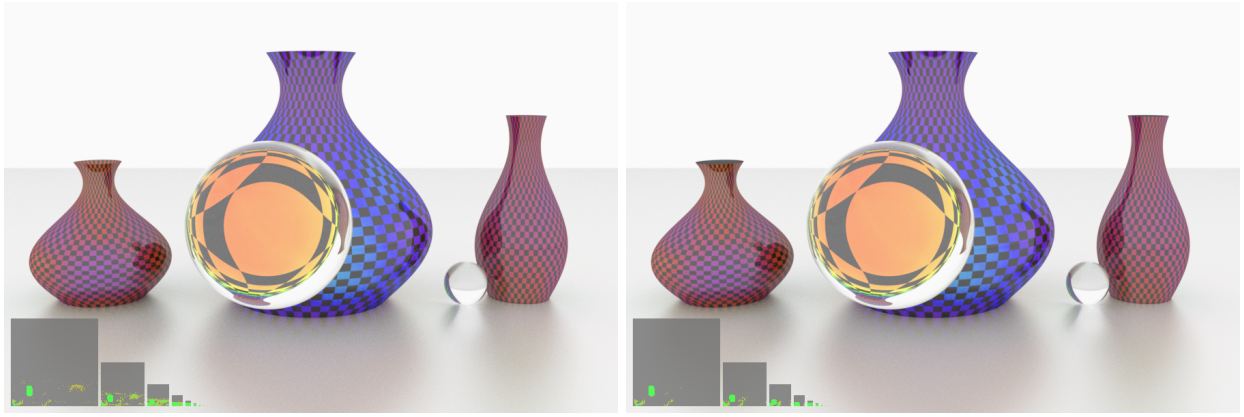


Figure 7: Negative curvature traps inside vases (left). The traps are removed by lining the vases with a diffuse material (right).

Limitations

We have been pleased with the simplicity and effectiveness of using ray cones to drive the OTK demand texturing system. Still, there are a few issues that are not resolved in our implementation.

One problem is that ray cones attempt to combine effects of reflection and refraction with BSDF effects from rough surfaces. These can fight or cancel out in unexpected ways, such as when a ray reflects from a rough concave mirror. This might be addressed by separating geometry and roughness effects, but would entail tracking yet more ray cones, something we would like to avoid.

"Negative curvature traps" can occur when rays bounce around in an enclosed region with negative curvature, such as the inside of a ball or a vase. In these cases, the ray cones converge and detailed mip levels get requested, often without contributing visually to the image. A general solution to this problem would make ray cones a more robust method for driving texture filtering.

The randomized nature of a path tracer causes the demand texture system to continue to request new texture tiles for many OptiX launches. The general strategy of invalidating an entire launch when a tile is requested may therefore be suboptimal. It may be useful to use coarser mip levels when samples are missing, or invalidate individual texture samples rather than entire launches.

Finally, our current implementation does not handle bidirectional methods such as photon mapping, virtual point lights, or bidirectional path tracing. For these methods, we need a way to initialize ray cones at light sources that minimizes detailed texture requests while still rendering with high accuracy.

References

[1] Akenine-Möller, T., Nilsson, J., Andersson, M., Barré-Brisebois, C., Toth, R., Karras, T. (2019). Texture Level of Detail Strategies for Real-Time Ray Tracing. In: Haines, E., Akenine-Möller, T. (eds) Ray Tracing Gems. Apress, Berkeley, CA.

[2] Tomas Akenine-Möller and Jim Nilsson (2019). Simple Environment Map Filtering Using Ray Cones and Ray Differentials. In: Haines, E., Akenine-Möller, T. (eds) Ray Tracing Gems. Apress, Berkeley, CA.

[3] Tomas Akenine-Möller, Cyril Crassin, Jakub Boksansky, Laurent Belcour, Alexey Panteleev, and Oli Wright. Improved Shader and Texture Level of Detail Using Ray Cones. Journal of Computer Graphics Techniques (JCGT), vol. 10, no. 1, 1-24, 2021

[4] Boksansky, J., Crassin, C., Akenine-Möller, T. (2021). Refraction Ray Cones for Texture Level of Detail. In: Marrs, A., Shirley, P., Wald, I. (eds) Ray Tracing Gems II. Apress, Berkeley, CA.

[5] Hao Qin, Menglei Chai, Qiming Hou, Zhong Ren, & Kun Zhou. (2014). Cone Tracing for Furry Object Rendering. IEEE Transactions on Visualization and Computer Graphics, 20(8), 1178–1188.