CHAPTER 26

RAY TRACED LEVEL OF DETAIL CROSS-FADES MADE EASY

Holger Gruen
Intel Corporation

ABSTRACT

Ray tracing techniques in today's game engines need to coexist with rasterization techniques in hybrid rendering pipelines. In the same throw, level of detail (LOD) techniques that are used to bring down the cost of rasterization need to be matched by ray traced effects to prevent rendering artifacts. This chapter revisits solutions to some problems that can arise from combining ray tracing and rasterization but also touches on methods for more generalized ray traced LOD transitions.

26.1 INTRODUCTION

The DirectX Raytracing (DXR) API [7] has incited a new wave of high-quality effects that replace rasterization-based effects. As of the writing of this text, fully ray traced AAA games are the exception. Even high-end GPUs struggle with ray tracing the massive amounts of dynamic geometry that some AAA games require. As a result, hybrid rendering pipelines that mix and match ray tracing and rasterization are commonplace.

Games engines employ a variety of techniques (see [6]) to reduce the geometric complexity of scene elements to bring down rendering cost. In order to avoid the complexity and limitations of continuous geometry decimation through vertex animation and the resulting edge collapse (also known as geomorphing, see [3]), many game engines instead cross-fade or cross-dither between two discrete geometry LODs (see, e.g., [2]), e.g., between LOD0 and LOD1. Here, LOD0 denotes a geometry that is comprised of fewer vertices and triangles than LOD1.

Typically, a transition factor f in the interval [0.0, 1.0] is used to control cross-faded LOD transitions (see, e.g., the ray traced transition example in Figure 26-1). During rasterized transitions, game engines need to draw both LOD0 and LOD1. When drawing LOD0, the pixel shader then uses a uniform

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_26

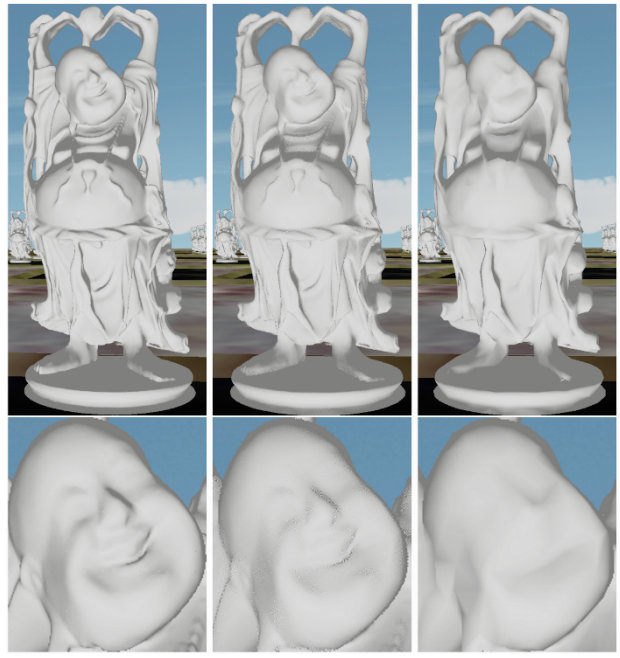

Figure 26-1. *Three stages of a ray traced LOD cross-fade. The highest LOD is shown on the left, a half-way transition is shown in the middle, and the lowest LOD is shown on the right.*

pseudo-random number r (from the interval [0.0, 1.0]) and discards the current pixel if $r < f$. The pixel shader that draws LOD1 uses the same pseudo-random number r and discards the current pixel if $r \geq f$.

This implies either that r is computed using the 2D position of a pixel as a seed or that a screen-space aligned texture containing random numbers is used. Low-discrepancy pseudo-random number sequences (see, e.g., [1]) or precomputed textures that store such sequences help to make the transition less noticeable.

Cross-faded LOD transitions are also a good choice for ray tracing applications as they prevent bounding volume hierarchy (BVH) refitting or rebuild operations for geomorphing geometries. Instead, bottom-level acceleration structures (BLASs) for discrete LODs can be instanced in the top-level acceleration (TLAS) structure as needed. So, similar to the fact that rasterization needs to render two LODs during transitions, it is necessary to put the geometry of two or more LODs in the BVH in order to enable ray traced transitions (see also [5]).

At this point in time, DXR doesn't provide obvious direct API support for letting potential traversal hardware handle high-quality LOD transitions, though.

The DirectX specification (see [7]) mentions *traversal* shaders (see [4]) as a potential future feature, but it is unclear if and when they will become a reality. In a hybrid ray traced technique, where the ray origin is usually derived from the world-space position of a given pixel, a traversal shader would allow LOD-based cross-fading using the same logic that the pixel shaders described previously use. Instead of discarding pixels, the ray would just be forwarded into the BLAS of the selected LOD according to the per pixel uniform random number r.

NVIDIA's developer blog (see [5]) describes a technique that uses the per-instance mask of a BLAS instance along with an appropriate ray mask to implement LOD cross-dithering that is accelerated by the traversal hardware. In this technique, the transition factor f is mapped to two different instance masks, e.g., *mask*0 for LOD0 and *mask*1 for LOD1. If, e.g., $f = 1.0$, all bits in the instance mask for LOD1 are set to 1 and no bits are set in the instance mask for LOD0. A transition factor of $f = 0.6$ means that *mask*1 has the first $uint(0.6 * 8)$ bits of the instance mask for LOD1 set to 1. The instance *mask*0 for LOD0 is then set to $\sim$ *mask*1&0*xFF*. This approach ensures that a per-pixel ray mask can be computed by randomly setting only one of its eight bits.

As Lloyd et al. [5] describe, their use of the 8-bit masks limits the number of levels for stochastic LOD transitions. Also, employing a transition technique that utilizes instance masks blocks these instance masks from being used for other purposes. If the limited number of LOD transitions turns out to be a quality issue for your application or if the mask bits are needed otherwise, it is possible to move the transition logic to the *any-hit* shader stage. The any-hit shader can evaluate stochastic transition without the limits described in [5]. It can store r in the ray payload. The any-hit shader can then ignore hits if $r < f$ and a triangle in the BLAS for LOD0 is hit. In the same manner, it can ignore hits for the BLAS for LOD1 if $r \geq f$.

However, using the any-hit shader stage for LOD cross-fades has a much higher performance impact than using the technique from [5] as it delays the decision to ignore a hit to a programmable shader stage that needs to run on a per-hit/triangle basis. Shader calls interrupt hardware traversal and can have a significant performance impact. Please note that a potential traversal shader implementation would also need to interrupt hardware traversal,

though not at the frequency of triangle hits but at the much lower frequency of hitting traversal bounding boxes.

In general, in situations where more than two LODs of an object are part of the BVH and the programmer wants to pick and chose between a number of LODs (e.g., purely based on the distance from the ray origin), any-hit or traversal shaders are the only tools of choice, as instance masks don't allow for this degree of flexibility.

26.2 PROBLEM STATEMENT

As outlined in the prevoius section and adding more details to the description in [5], in hybrid rendering pipelines ray traced techniques need to be able to manage the fact that rasterized G-buffers may represent different LODs of the same cross-faded object in neighboring pixels.

Adding, e.g., hard ray traced shadows to an engine that uses rasterization to lay down a G-buffer can be done like this:

1. Reconstruct the world-space position *WSPos* for the current pixel from the value in the depth buffer.
2. Compute the starting point of the shadow ray by offsetting *WSPos* along the unit-length per-pixel normal **N** that has been scaled by a factor *s*: $ray.Origin = WSPos + s * N$.
3. Trace the ray from the origin toward the light source.

The normal scaling factor *s* is chosen in a way that prevents self-shadowing (Lloyd et al. [5] call these self-shadowing artifacts *spurious shadows* if they result from mismatching LODs) but also prevents the localized loss of shadows from small local details. Assuming that the current G-buffer pixels contain some LOD cross-faded object, as depicted in Figure 26-2, then neighboring pixels may well belong to different LODs.

It is possible, depending on how the simplified LODs are built, that the geometry of LOD1 locally is farther out when compared to the geometry of LOD0 or vice versa, as shown in Figure 26-3.

As game engines currently place either the geometry of LOD0 or the geometry of LOD1 in the BVH, this can create problems. Assume that LOD1 is outside of the geometry of LOD0 and that only the geometry of LOD1 has been placed in the BVH. Under this scenario, a shadow ray that emanates from the

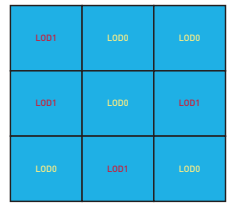

Figure 26-2. *A 3 × 3 portion of a G-buffer that contains pixels that have been rasterized from geometry of LOD0 and LOD1.*

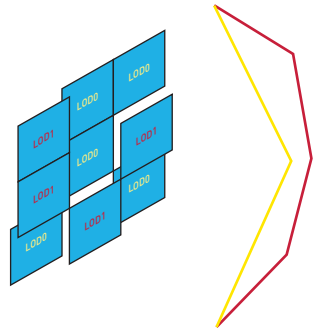

Figure 26-3. *A 3 × 3 portion of a G-buffer with pixels of LOD0 and LOD1 and a depiction of the underlying geometry of LOD0 in yellow and LOD1 in red.*

central G-buffer pixel (see Figure 26-2) and that belongs to LOD0 may well hit the geometry LOD1. The result is that this pixel is falsely assumed to be in shadow, as shown in Figure 26-4. Please note that which LOD is outside may well change across a model, so it isn't possible to pick the most suitable LOD from an engine point of view to prevent this.

It is, of course, possible to work around this by increasing the normal scaling factor *s* to a point that prevents this problem for all scene elements. But, as described previously, this may well lead to the loss of local details, in ambient occlusions or shadows. Also, increasing *s* raises the probability of moving the ray origin into some close-by geometry. Similar problems exist with almost all other ray traced techniques that need to work from a LOD-dithered G-buffer.

26.3 SOLUTION

The following steps help to work around the problems described above by making sure that rasterization and ray tracing use the same random number to implement LOD cross-fading based on an individual per-object transition factor *f*:

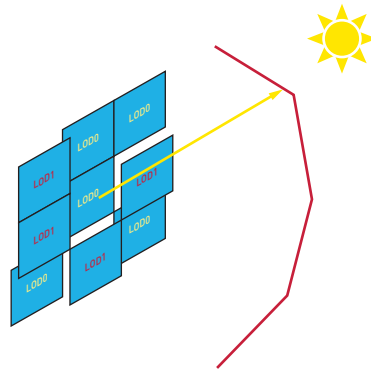

Figure 26-4. *Choosing the wrong LOD (e.g., LOD1) can lead to self-shadowing artifacts.*

1. During rasterization, pick a uniform pseudo-random number r from the interval [0.0, 1.0] that purely depends on the 2D position of the current pixel, or that depends on a combination of the 2D position and the instance ID or object ID of the current object.

   > If you intend to use any-hit or traversal shader–based ray traced transitions, then do the following:

   - When drawing LOD0, the pixel shader discards the current pixel if $r < f$.
   - The pixel shader that draws LOD1 discards the current pixel if $r \geq f$.

   > If you intend to use instance mask–based transitions (see [5]), then do the following:

   - Compute a binary $mask1$ that has the first $uint(f * 8)$ bits set to one to be used by the pixel shader for LOD1: $mask1 = (1 << uint((8 + 1) * f)) - 1$.
   - Compute $mask0$ to be used by the pixel shader for LOD0: $mask0 = (\sim mask1)\&0xFF$.
   - Multiply r by 7 to arrive at the bit index to compute $rmask$ to be used by the pixel shader for LOD1: $rmask = 1 << uint(r * 7)$.
   - When drawing LOD0, the pixel shader discards the current pixel if $(rmask\&mask1) == 0$.
   - The pixel shader that draws LOD1 discards the current pixel if $(rmask\&mask0) == 0$.

   Please note that an instance mask–based transition may limit the quality of a rasterized cross-fade in the same way as it may limit ray

traced LOD transitions. If quality is a concern for you, you may need to resort to any-hit shader–based transitions as outlined in Section 26.1.

2. Put BLAS instances for LOD0 and LOD1 of all objects into the TLAS that are currently undergoing a LOD transition.

3. Use the same uniform random number r that was used during rasterized cross-fading in your ray traced cross-fading setup for the current pixel.

   - For an any-hit (or a future traversal) shader–based transition, put r into the ray payload and ignore hits if $r < f$ and a triangle in the BLAS for LOD0 is hit. In the same vein, ignore hits for the BLAS for LOD1 if $r \geq f$. Local root signature constants in the shader binding table for the hit entries for LOD0 and LOD1 can be used to detect if a triangle from LOD0 or LOD1 has been hit.
   - For the case of instance mask–based transitions, as described in [5], do the following:
     - Compute a binary $mask1$ that has the first $uint(f * 8)$ bits set to one to be used by the pixel shader for LOD1: $mask1 = (1 << uint((8 + 1) * f)) - 1$.
     - Compute $mask0$ to be used as the instance for LOD0: $mask0 = (\sim mask1)\&0xFF$.
     - When tracing a ray, use $uint(f * 7)$ to set one random bit in the ray mask $rmask$: $rmask = 1 << uint(r * 7)$.

   Using the approach outlined here, the rays cast from the reconstructed world-space positions of a pixel always only return hits with the same LOD that was used when rendering the pixel.

26.4 FUTURE WORK

Animating a LOD cross-fade shows that there is a clear difference in the fluidity of the transition between a mask–based fade and a comparison value–based fade using an any-hit shader. It would be beneficial to extend future ray traversal hardware to include a per-BLAS instance comparison value and a flag indicating either a less than or a greater than comparison value, which then gets stored in the BVH. The `TraceRay` intrinsic could be extended to add an additional 8-bit comparison value parameter as well.

The traversal hardware could then, on top of the binary `AND` operation on the mask, perform the comparison operator on the values of the instance and the

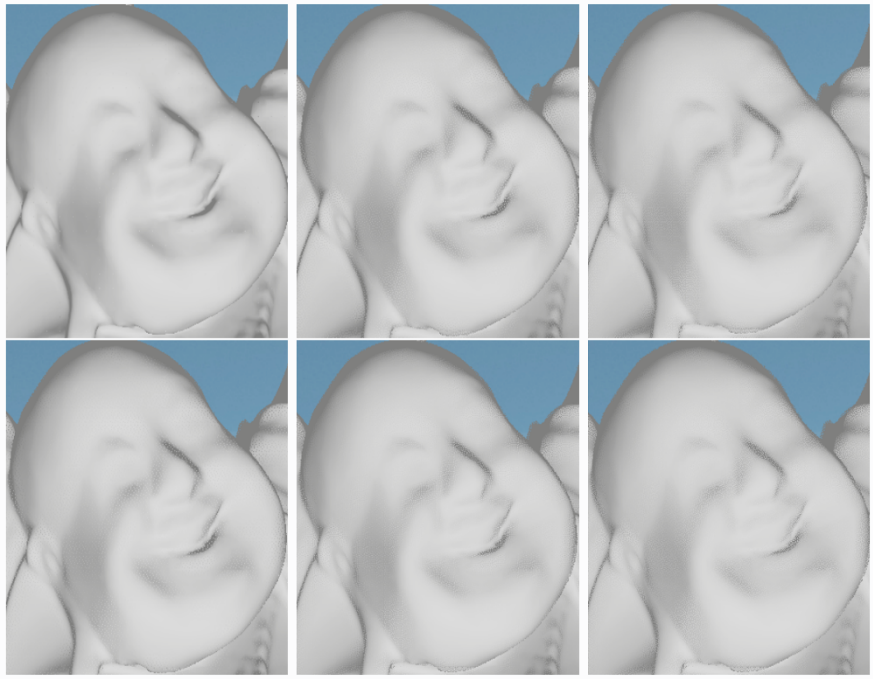

Figure 26-5. *Three stages of a ray traced LOD cross-fade at f = 0.904 (left), f = 075 (middle), and f = 0.647 (right). Top: images produced using a mask-based transition. Bottom: Using an emulated comparison value–based transition.*

value of the ray. Ray traversal into an instance could only continue if the comparison operations returns `true`.

This new functionality would enable hardware-assisted LOD cross-fades that are comparable in quality to what any-hit shader–based cross-fades produce today.

Figure 26-5 shows three images from a LOD transition using a mask-based and an emulation of a comparison value–based transition. While the full difference in quality and fluidity can only be shown in a video, please note that the comparison-based method produces transitions with many more intermediate steps than the mask-based transition.

26.5 CONCLUSION

This chapter expands on existing solutions that make sure that there are no mismatches between rasterized LOD and ray traced LOD. The method

described can make use of fully accelerated hardware traversal, if the quality of instance mask–based transitions is good enough to meet your quality requirements. If this is not the case, you will need to pick the slower path that involves calls to any-hit shaders for objects that are currently undergoing a LOD transition. Future traversal hardware that features comparison value–based conditional traversal can deliver high-quality and fluid LOD transitions at full speed without the use of any-hit shaders.

REFERENCES

[1] Ahmed, A. G. M., Perrier, H., Coeurjolly, D., Ostromoukhov, V., Guo, J., Yan, D.-M., Huang, H., and Deussen, O. Low-discrepancy blue noise sampling. *ACM Trans. Graph.*, 35(6), Nov. 2016. ISSN: 0730-0301. DOI: 10.1145/2980179.2980218. https://doi.org/10.1145/2980179.2980218.

[2] Arlebrink, L. and Linde, F. A study on discrete level of detail in the Unity game engine. https://draketuroth.files.wordpress.com/2018/06/a-study-on-discrete-level-of-detail-in-the-unity-game-engine.pdf, 2018.

[3] Hoppe, H. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 99–108, 1996. DOI: 10.1145/237170.237216.

[4] Lee, W.-J., Liktor, G., and Vaidyanathan, K. Flexible ray traversal with an extended programming model. In *SIGGRAPH Asia 2019 Technical Briefs*, pages 17–20, 2019. DOI: 10.1145/3355088.3365149.

[5] Lloyd, B., Klehm, O., and Stich, M. Implementing stochastic levels of detail with Microsoft DirectX Raytracing. https://developer.nvidia.com/blog/implementing-stochastic-lod-with-microsoft-dxr, June 15, 2020.

[6] Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., and Huebner, R. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 1st edition, 2012.

[7] Microsoft. DirectX Raytracing (DXR) functional spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html, 2021.