Flexible Ray Traversal with an Extended Programming Model

Won-Jong Lee
Intel Corporation

Gabor Liktor
Intel Corporation

Karthik Vaidyanathan
Intel Corporation

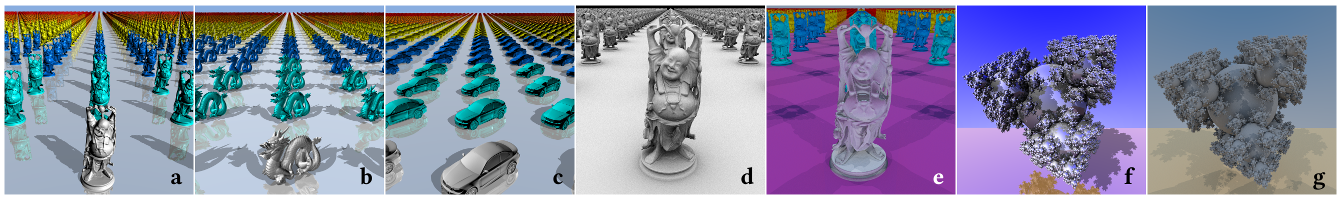

Figure 1: Our programming model enables flexible control of scene hierarchies during traversal. Some example use cases are: distance based LOD with reflections and shadows (a–c), ray-type based LOD with ambient occlusion and diffuse rays (d–e), and multi-level instancing (f–g).

ABSTRACT

The availability of hardware-accelerated ray tracing in GPUs and standardized APIs has led to a rapid adoption of ray tracing in games. While these APIs allow programmable surface shading and intersections, most of the ray traversal is assumed to be fixed-function. As a result, the implementation of per-instance Level-of-Detail (LOD) techniques is very limited. In this paper, we propose an extended programming model for ray tracing which includes an additional programmable stage called the traversal shader that enables procedural selection of acceleration structures for instances. Using this programming model, we demonstrate multiple applications such as procedural multi-level instancing and stochastic LOD selection that can significantly reduce the bandwidth and memory footprint of ray tracing with no perceptible loss in image quality.

CCS CONCEPTS

• **Computing methodologies → Ray tracing**.

KEYWORDS

ray tracing, programming model, level-of-detail, instancing

ACM Reference Format:
Won-Jong Lee, Gabor Liktor, and Karthik Vaidyanathan. 2019. Flexible Ray Traversal with an Extended Programming Model. In *SIGGRAPH Asia 2019 Technical Briefs (SA '19 Technical Briefs), November 17–20, 2019, Brisbane, QLD, Australia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3355088.3365149

1 INTRODUCTION

Recent programming models for real-time ray tracing such as the Microsoft DirectX Ray Tracing (DXR) API [Microsoft 2018] have enabled developers to easily incorporate ray tracing in their game engines by building upon existing APIs like Direct3D 12. DXR models the scene representation using a two-level hierarchy that comprises a single top-level acceleration structure (TLAS) that is build over instances of bottom level acceleration structures (BLAS). This abstraction gives GPU vendors the flexibility to choose any implementation of acceleration structures and deploy hardware acceleration for ray-scene intersections [NVIDIA 2018]. While DXR provides some programmability during traversal such as intersection shaders for custom shapes and any-hit shaders for sampling alpha textures, it does not allow procedural selection and transformation of instances during traversal.

This programming model trades flexibility for performance, but in complex rendering scenarios, programmers may need to dynamically modify ray traversal behavior on a higher (e.g. per-object) level that cannot easily be baked into fixed-function hardware. For example, multi-level instancing, which is widely used in production rendering and dynamic LOD techniques, could be quite useful in ray-traced game engines [Tatarchuck 2019]. By adaptively selecting objects with different LODs and constructing hierarchical instances during traversal, the number of accesses to nodes and primitives can be reduced, improving memory bandwidth and cache efficiency.

We propose to address this limitation by introducing a new shader stage called the *traversal shader*, which can be invoked at higher-levels of the AS traversal upon intersecting a programmable node. By redirecting traversal to a different AS, and potentially transforming the ray, augmented by user data, it provides a new methodology to support a wide range of dynamic, per-ray-instance traversal algorithms. We demonstrate a variety of LOD selection scenarios and mitigate artifacts caused by self-occlusion or abrupt LOD transitions. We also illustrate procedural multi-level instancing through fractal-rendering. Based on experimental data, we argue that for complex scenes this level of programmability does not necessary penalize performance by achieving a significant reduction in bandwidth (43% to 71%) as well as the memory footprint.

2 RELATED WORK

A flexible programming model is an important component of any ray tracing system. Frameworks such as OptiX [Parker et al. 2010] or Embree [Wald et al. 2014] are used extensively in production rendering. OptiX 6 provides limited control of ray traversal through *Visit* programs, allowing the user to select a child node to be traversed when a ray visits a *Selector* node. However, this selection was limited to one of the predefined children. Embree has been

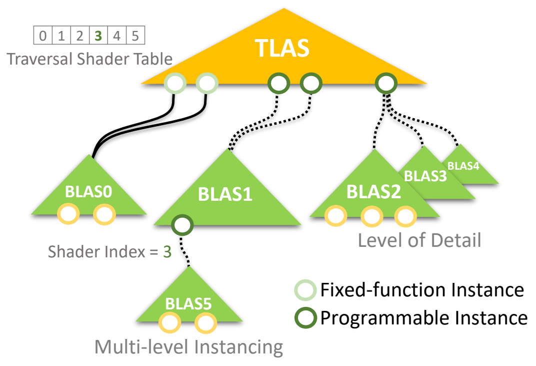

Figure 2: Flexible scene hierarchy with multi-level instancing and LOD selection using Traversal Shaders.

supported multi-level instancing by doing user-defined objects and just recently added a direct support from 3.6. In contrast, our traversal shader allows arbitrary AS selection, naturally scaling to multiple levels. Current real-time APIs such as DXR are limited to single-level instancing without programmable traversal controls.

LOD and multi-level instancing are widely-used rendering optimizations to handle complex scenes. Christensen et al. [2003] adaptively utilized geometry caches based on ray differentials. Some production renderers have adopted LOD to incoherent rays [Pantaleoni et al. 2010; Tabellion 2010], while others support multi-level instancing to reduce memory footprint and tree build times [Dietrich et al. 2006; Georgiev et al. 2018]. In real-time rendering, game engines have primarily relied on camera distance to switch between pre-defined details of geometry, using methods like sub-pixel screen-door transparency to smoothly blend between discrete LOD levels [Mittring 2007], or dynamically tessellate surfaces for continuous LOD transition [Nießner et al. 2012]. Rendering performance can be further increased by extending the concept beyond geometry to shader simplification [He et al. 2016].

Ray tracing introduces additional constraints for these applications since they also need to consider the distribution of indirect rays. The main focus of this paper is to introduce a novel programming model for future real-time ray tracing APIs. This can support various geometry and shader LOD techniques for real-time ray tracing with fully programmable control of ray traversal.

3 PROGRAMMING MODEL

Due to its increasing adoption in the industry, we base our discussion on the terminology of the Microsoft DXR API. As mentioned earlier, the DXR scene hierarchy includes a top-level acceleration structure (TLAS), referencing instances, and bottom-level acceleration structures (BLAS), referencing triangles and custom primitives.

We extend this model with a new leaf node called a *programmable instance* (*PI*), which can be referenced by both the TLAS and BLAS. A PI can be associated with a *traversal shader* that can transform the ray and select a BLAS for continuing traversal, as shown in Figure 2. In order to specify a traversal shader for execution, we rely on DXR shader tables and introduce a new type called *traversal shader table*. For each PI node, the user provides a shader index to select an entry from this table for execution, when a ray intersects it. Listing 1 shows an example traversal shader that transforms the ray and selects an instance LOD. The traversal shader can access the ray payload if needed and can also access a payload from its parent instance. Every execution of the traversal shader creates a new entry at the top of the *instance stack* and the *SetInstance()* intrinsic updates this entry. For each instance that is traversed, the user can specify an offset in the hit shader table. This can be used to select the shader LOD by specifying a different offset for each LOD instance. The user can also pass a payload value to the next child which can be useful for procedurally generated instance transformations.

Listing 1: Example Traversal Shader.

```
[shader("traversal")]
void myLodShader(in RayPayload rp, in InstancePayload ip) {
  // Transform Ray
  float3 objRayOrigin = worldToObj * WorldRayOrigin()
  float3 objRayDirection = worldToObj * WorldRayDirection()
  RayDesc transformedRay = { objRayOrigin, RayTMin(),
                             objRayDirection, RayTMax()};
  // Compute LOD based on Instance distance
  RaytracingAccelerationStructure myAccStructure;
  myAccStructure = FetchLOD(RayTInstance())
  // Set the next level instance and hit shader table offset
  SetInstance(myAccStructure, transformedRay,
              myHitShaderOffset, myInstPayload);
}
```

4 APPLICATIONS

4.1 Programmable LOD Selection

Distance and Object Size: A popular way to select a LOD level is based on the pixel size of an object projected to screen space, widely used in many game engines [Akenine-Möller et al. 2018]. This combines the distance to the object and the length of the radius of its bounding sphere. We use the following equations:

$$r_{\text{pixel}} = \frac{r_{\text{world}}}{tan(\frac{FOV_y}{2})D}\frac{H}{2} \quad (1) \qquad \text{LOD} = \left\lfloor \log_2(2^{N-1}\frac{r_{\text{pixel}}}{r_{\text{max}}}) \right\rfloor \quad (2)$$

where r_{pixel} and r_{world} are the length of the radius in screen and world space, FOV_y is the vertical field of view, D is the distance between the ray origin and the object, H is the screen height in pixels, N is the number of the LOD levels used, and r_{max} is a predefined length corresponding to the projection of the most detailed LOD.

Figure 1(a–c) shows an example of LOD selection implemented through equations (1-2). For easy identification, we assigned different colors to the instances on different levels. The different BLASes are adaptively selected according to the distance.

Ray Type: Production renderers often apply coarser LOD to divergent indirect rays since these rays affect the quality to a lesser degree. That is, the primary ray traces the full-resolution geometry and the secondary rays trace the lower LOD, which could reduce the cost of indirect GI computations [Pantaleoni et al. 2010; Tabellion 2010]. This technique can be naturally implemented using traversal shaders, by selecting a different BLAS based on the ray type. Figure 1(d–e) is an example of applying this technique with ambient occlusion and diffuse indirect rays. The user can achieve further performance improvements by combining this technique with distance-based LOD, which is not possible in the standard DXR model.

4.2 Stochastic LOD Transitions

The traversal shader can also be used to reduce the visual artifacts that often occur in LOD applications. A typical example is the "popping" caused by abruptly transitioning between two LOD ranges.

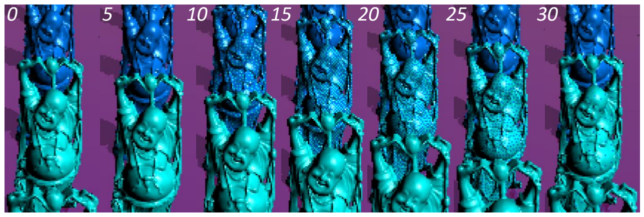

Figure 3: Frame sequence from a camera animation using stochastic LOD transitions. The frame numbers are shown in the top-left corners.

This is often resolved by a stochastic approach in game engines, for example by applying a smooth screen-door dithering mask to each of the two LODs spanning the transition [Mittring 2007].

In Figure 3 we demonstrate a stochastic LOD transition using our programming model. The traversal shader compares the fractional part of the LOD value at each pixel with a sample value in a blue noise dither mask [Ulichney 1993], then returns one of the two integer LOD values according to the binary result. Unlike the stencil buffer method, which renders twice, the traversal shader can handle this in a single pass without any overdraws, because it assigns only one LOD value to each ray.

We can also handle another artifact called *tunneling* caused by self-occlusion [Djeu et al. 2011]. There are many cases where a primary ray intersects an instance with a certain LOD geometry and the secondary ray from this hit may intersect the same instance using a different LOD. We can use traversal shaders to ensure the usage of the identical LOD level of the previous surface hit. By storing the instance ID and the selected LOD level of the previous hit point in the ray payload, the traversal shader can detect if the same instance is to be traversed by the secondary ray and automatically select the stored LOD, overriding the current selection scheme. This way we can apply coarse LOD for certain indirect illumination rays more aggressively without being concerned with false self-occlusions (Figure 4).

Finally, the ray payload can also be used to accumulate the path length of multiple ray segments, potentially also considering cone angles. In Figure 5 we show this algorithm applied to reflection rays, where the closest hit shader passes the total path length to the traversal shader for LOD computation. Note that for this simple demonstration we consider only the ray length, but considering the surface curvature and BRDF roughness would allow a more sophisticated scheme using very coarse LODs for rough materials.

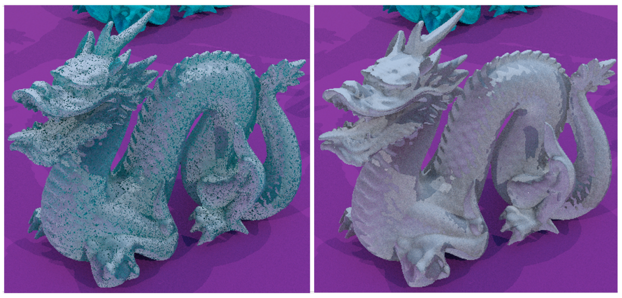

Figure 4: Artifacts caused by self-occlusions of different LODs (left) and our result after consistent selection by tracking the previous hit instance using the Traversal Shader (right).

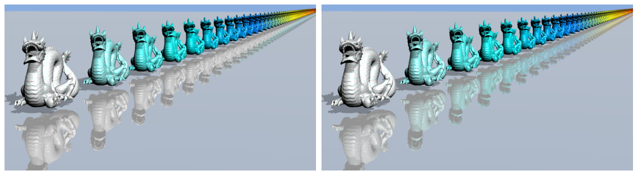

Figure 5: Incorrectly selected LOD levels by using a simple hit distance (left), corrected by using accumulated distances computed in the Closest-Hit and Traversal Shaders (right).

4.3 Multi-level Instancing

Instancing has been used extensively in production rendering to reduce scene complexity and tree construction times. By storing multiple references to the same geometry, several copies of the same object can be rendered with different transformations and shading parameters. While DXR has been specifically designed to have this feature, it only supports one level of instances within a two-level AS. If the scene graph contains more levels of hierarchy, this restriction forces the renderer to flatten the bottom levels into a single BLAS, therefore expanding the number of primitives.

Multi-level instancing can benefit from traversal shaders in many scenarios. As mentioned in Sec. 3, each execution of a traversal shader pushes a new entry to the *instance stack*, which may include user-defined data besides standard ray transformation information. Up to the maximum depth of this stack (pre-allocated upon context initialization), the user is free to select a new level of instancing using a BLAS reference and a ray transformation. It is also the responsibility of the user to provide a valid bounding box in the PI node that the actual transformed instance would fit into.

Figure 1(f–g) is an extreme example of multi-level instancing naturally supported by our model. The *sphereflake* fractal recursively copies nine child spheres from a parent, which is a benchmark of *Standard Procedural Databases* [Haines 2019]. When implementing through traversal shaders, the number of primitives can be drastically reduced compared to single-level instancing (820× with a recursion depth of 4). Recursive instancing is only possible using dynamic traversal, and the maximum recursion depth is only limited by the instance stack preallocation. Akin to ray depth in recursive ray tracing, the traversal shader may abort ray traversal at a threshold instance depth. While we illustrate the concept on this simple fractal, we believe this feature holds an interesting potential for complex procedural worlds in future games.

5 EVALUATION

In order to evaluate different LOD schemes, we use a user space implementation of our programming model and a software ray tracing runtime. We implement ray tracing shaders as C++ functions, with an emulation of intrinsics and data types available in HLSL. To compute ray-scene intersections, we use the traversal algorithm of Vaidyanathan et al. [2019] with a 64B quantized BVH6 layout and 64B triangle-pair (quad) primitives at its leaves. In the runtime, we model a 6MB cache with a 64B line size to reflect the size of the last level cache on modern GPUs. We use three models, Buddha, Dragon, and BMW for our analysis with 3,864 uniformly distributed instances in the scene as shown in Figure 1. For each model we use six LOD levels, where the number of primitives decreases 4× with each increase in LOD.

Table 1 summarizes the cache miss rates (%) and the average bandwidths (Bytes/ray) with full geometric detail (no LOD), discrete LOD selection (LOD-D) and stochastic LOD selection (LOD-S). Using coarser LODs for distant camera ray hits results in fewer cache misses and a 58% reduction in bandwidth with discrete LODs. With stochastic LODs, we see a slightly smaller reduction in bandwidth of up to 43% but with image quality that is comparable to no LODs. With low-complexity meshes like the BMW model, we see a smaller reduction in the bandwidth (2.5%) with stochastic LOD selection for camera rays. In this case, the additional cost of sampling two LODs close to the camera may be significant compared to the benefits of reduced geometric complexity at a distance. We also evaluate scenes with ambient occlusion (AO) and two-bounce diffuse indirect lighting. Using a coarser LOD (level 4) for indirect and AO rays results in a significant bandwidth reduction across all scenes (60% to 89%).

Table 1: Comparison of the cache miss rates and average bandwidth for different ray types with discrete and stochastic LOD selection.

		Miss rate (%)	**BW (Bytes/ray)**
	Buddha (1M tris)		
Primary	No LOD	10.3	127.0
	LOD-D	5.0	53.7
	LOD-S	6.4	72.9
	Change(D) (%)	**-51.5**	**-57.7**
	Change(S) (%)	**-37.9**	**-42.6**
Primary + AO	No LOD	10.1	62.9
	LOD	4.1	25.4
	Change (%)	**-59.4**	**-59.6**
	Dragon (871K tris)		
Primary	No LOD	10.0	138.0
	LOD-D	4.9	57.8
	LOD-S	6.2	79.8
	Change(D) (%)	**-51.0**	**-58.1**
	Change(S) (%)	**-38.0**	**-42.2**
Primary + Indirect	No LOD	9.8	167.8
	LOD	3.3	49.1
	Change (%)	**-66.3**	**-70.7**
	BMW (384K tris)		
Primary	No LOD	2.1	28.5
	LOD-D	1.6	19.1
	LOD-S	2.1	27.8
	Change(D) (%)	**-23.8**	**-33.2**
	Change(S) (%)	0.0	**-2.5**
Primary + AO	No LOD	0.7	5.0
	LOD	0.2	1.7
	Change (%)	**-71.4**	**-66.7**
Primary + Indirect	No LOD	0.8	32.7
	LOD	0.1	3.7
	Change (%)	**-87.5**	**-88.7**

6 DISCUSSION AND FUTURE WORK

Our programming model offers flexible scene management on the cost of additional shader invocations during ray traversal. However, users can choose to selectively apply programmable instancing to parts of the scene with complex geometry where the reduction in memory bandwidth should outweigh the additional cost of shading. Moreover, traversal shader invocations are likely to be less frequent than some existing shader stages, for example, any-hit shaders that are invoked for each alpha-textured primitive.

As a limitation, programmable instances also preclude some acceleration structure optimization techniques like re-braiding [Benthin et al. 2017] that require access to the instance acceleration structures during build.

A complete understanding of the performance implications of our programming model requires a more thorough analysis of these tradeoffs, which we plan to investigate in the future.

ACKNOWLEDGMENTS

We thank Intel's Advanced Rendering Technology team. We also thank David Blythe and Chuck Lingle for supporting this research. The Buddha and Dragon scenes are courtesy of Stanford 3D scanning repository. The BMW scene was modeled by Fred C. M'ule Jr., the Tree model was modeled by Reza Shams and the Sphereflake was created by Eric Haines. We are grateful for these scenes. We also appreciate the anonymous reviewers for their insightful comments.

REFERENCES

Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. 2018. *Real-Time Rendering 4th Edition.* A K Peters/CRC Press.

Carsten Benthin, Sven Woop, Ingo Wald, and Attila T. Áfra. 2017. Improved Two-level BVHs Using Partial Re-braiding. In *Proceedings of High Performance Graphics.* ACM.

Per Christensen, David Laur, Julian Fong, Wayne Wooten, and Dana Batali. 2003. Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. *Computer Graphics Forum,* 22, 3 (2003), 543–552.

Andreas Dietrich, Gerd Marmitt, and Philipp Slusallek. 2006. Terrain Guided Multi-Level Instancing of Highly Complex Plant Populations. In *IEEE Symposium on Interactive Ray Tracing.* 169–176.

Peter Djeu, Warren Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark. 2011. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. *ACM Transactions on Graphics,* 30, 5 (2011), 115:1–115:26.

Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramón Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, Adrien Herubel, Declan Russell, Frédéric Servant, and Marcos Fajardo. 2018. Arnold: A Brute-Force Production Path Tracer. *ACM Transactions on Graphics,* 37, 3 (2018).

Eric Haines. 2019. Standard Procedural Databases. http://www.realtimerendering.com/resources/SPD/

Yong He, Tim Foley, and Kayvon Fatahalian. 2016. A System for Rapid Exploration of Shader Optimization Choices. *ACM Trans. Graph.* 35, 4 (July 2016), 112:1–112:12.

Microsoft. 2018. D3D12 Ray Tracing Functional Spec. http://forums.directxtech.com/index.php?topic=5860.0

Martin Mittring. 2007. Finding Next Gen - CryEngine 2. SIGGRAPH Advanced Real-Time Rendering in 3D Graphics and Games Course.

Matthias Nießner, Charles Loop, Mark Meyer, and Tony Derose. 2012. Feature-Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces. *ACM Transactions on Graphics,* 31, 1 (2012), 6:1–6:11.

NVIDIA. 2018. Turing GPU Architecture, Technical Whitepaper.

Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. 2010. PantaRay: Fast Ray-Traced Occlusion Caching of Massive Scenes. *ACM Transactions on Graphics,* 29, 4 (2010), Article No. 37.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics,* 29, 4 (2010), 66:1–66:13.

Eric Tabellion. 2010. Ray Tracing vs. Point-based GI for Animated Films. In *ACM SIGGRAPH 2010 Course: Global Illumination Across Industries.*

Natalya Tatarchuck. 2019. How Do We Harness Film Rendering Techniques into Game Ray Tracing Rendering. In *ACM SIGGRAPH 2019 Course: Open Problems in Real-Time Rendering.*

Robert Ulichney. 1993. The void-and-cluster method for dither array generation. In *IS&T/SPIE Symposium on Electronic Imaging and Science*, Vol. 1913. 332–343.

Karthik Vaidyanathan, Carsten Benthin, and Sven Woop. 2019. Wide BVH Traversal with a Short Stack. In *Proceedings of High Performance Graphics.*

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics,* 33, 4 (2014), 143:1–143:8.