Lazy Build of Acceleration Structures with Traversal Shaders

Won-Jong Lee
Intel Corporation

Gabor Liktor
Intel Corporation

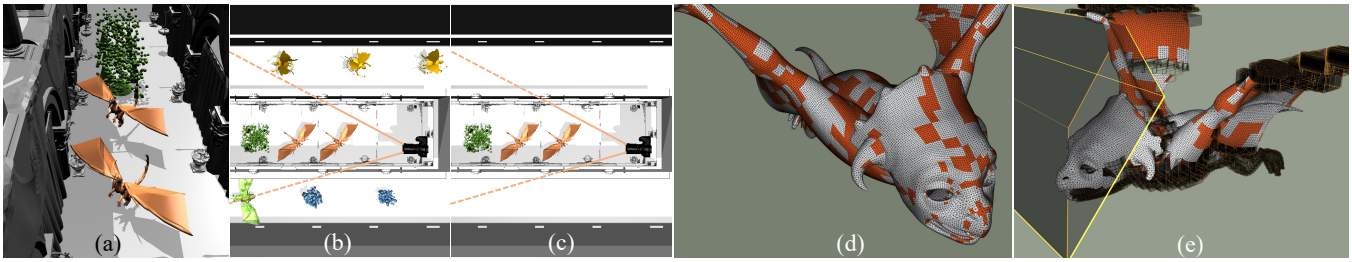

Figure 1: Our algorithm allows the AS builder to ignore geometry if no rays traverse their bounds. When rendering the Sponza scene with several dynamic objects (a), conventional renderers may update the AS of all dynamic objects (b), while our lazy build omits invisible ones including secondary rays (c). We can also extend the scope of the algorithm for procedural geometry, such as tessellation (d). We visualize the bounds of animated patches that were not even tessellated for the current frame (e).

ABSTRACT

Modern ray tracing APIs allow developers to easily build acceleration structures (AS) with various optimization techniques. However, the visibility-driven on-demand build can not be implemented with the current APIs due to the lack of flexibility during ray traversal. In this paper, we propose a new algorithm to lazily build ASes for real-time ray tracing with an extended programming model supporting flexible ray traversal. The core idea of our approach is a multi-pass build-traversal, which computes instance visibility and builds the visible ASes in different passes. This allows us to lazily build the entire AS only when necessary without hardware implication. Applying our algorithm to dynamic scenes, we demonstrate that the build cost is significantly reduced with minimal overhead.

CCS CONCEPTS

• **Computing methodologies** → **Ray tracing**.

KEYWORDS

ray tracing, programming model, BVH, acceleration structures

ACM Reference Format:
Won-Jong Lee and Gabor Liktor. 2020. Lazy Build of Acceleration Structures with Traversal Shaders. In *SIGGRAPH Asia 2020 Technical Communications (SA '20 Technical Communications), December 4–13, 2020, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3410700.3425430

1 INTRODUCTION

Modern graphics programming APIs [Khronos 2020; Microsoft 2018] have recently led to the rapid adoption of ray tracing in games. They abstract key elements of ray tracing, such as traversal and acceleration structures (AS), so that GPU vendors can optimize the implementation on their platforms. The scene representation is defined using a two-level hierarchy that comprises a single top-level acceleration structure (TLAS) that is built over instances of bottom level acceleration structures (BLAS). While game developers cannot implement their own AS construction method, they can select from rebuilding/updating the AS as a tradeoff between quality and build cost. The overhead of the AS construction and storage for the entire scene can be significant since ray tracing introduces indirect visibility which makes it nontrivial to cull invisible parts of the scene. This restricts the complexity of ray-traced dynamic geometry in games since the cost of updating their BLAS can become excessive.

Lazy build [Hunt et al. 2007] is one possible way to avoid redundant AS construction. The rendering of a frame starts with a coarse AS like a scene-graph or hierarchies of the previous frame, then progressively builds the newly required ASes for the objects that are hit by rays during traversal. Invisible objects can be effectively excluded from the construction process. However, this method cannot be easily implemented with the current APIs, because the higher-level (i.e. per-object) programmability essential to compute the instance visibility is not supported.

In this paper, we propose *multi-pass lazy build* (MPLB), an algorithm for real-time ray tracing that resolves this problem with an extended programming model called *traversal shader* [Lee et al. 2019]. It allows us to track the instance-level traversal during each ray dispatch and selectively build BLASes for only the potentially visible geometry at render time. Akin to some adaptive sampling techniques, MPLB may require multiple ray dispatches over the same set of pixels to relaunch rays to previously unbuilt parts of the scene, but we show several heuristics that can minimize this overhead, such as the assumption of frame-to-frame coherence and rasterized primary visibility. We demonstrate MPLB on multiple dynamic geometry use cases. Based on experimental data, we argue that it can be a prominent AS build mechanism by showing a significant reduction in build complexity compared to one-time builders with only a marginal increase in traversal cost on average.

SA '20 Technical Communications, December 4–13, 2020, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8080-5/20/11…$15.00
https://doi.org/10.1145/3410700.3425430

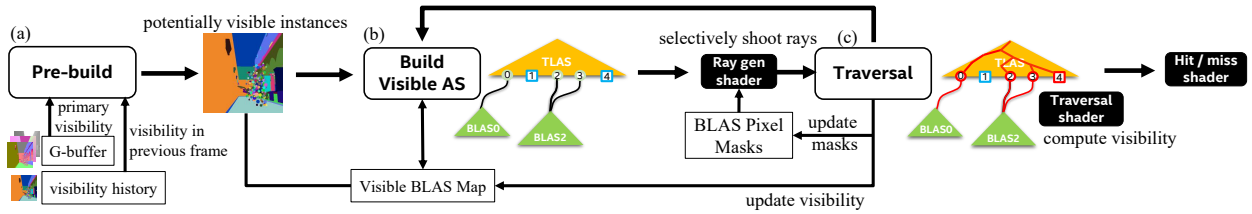

Figure 2: Overview of our algorithm. It begins with an initial hierarchy from the primary and historical visibilities. The BLASes are selectively built for the only visible instances based on the visibilities computed in traversal shader during traversal.

2 RELATED WORK

Handling dynamic objects is a very important in modern games. GPU vendors are optimizing the build algorithm for their platforms and exposing them according to the API interface. With this interface, game developers are making various efforts to reduce the construction time, using methods like *instance culling*, *asynchronous compute* [Schmid 2019], or *build throttling* [Choi 2020]. However, none of the above techniques can support an on-demand lazy build algorithm constructing an AS based on object visibility.

Lazy build is a prominent technique that can reduce the construction cost because only the necessary AS is built on demand. Since Hunt et al. [2007] first proposed the algorithm, some production renderers [Benthin et al. 2015; Georgiev et al. 2018] have introduced this feature in production. One downside is that it should queue rays hitting the object on which AS is being built. Hence, it would require complex hardware and software changes in the current real-time solutions. Moreover, it cannot be implemented with the existing APIs due to the lack of flexibility during traversal.

The traversal shader is a new programmable stage extending the current real-time ray tracing APIs that allows the dynamic modification of ray traversal on the instance level [Lee et al. 2019]. It can receive the parameters for the current ray and instance (e.g transformation), and access all the standard shader resources, enabling applications such as stochastic LoD and multi-level instancing. In this paper, we leverage traversal shaders to implement a novel lazy build algorithm for real-time ray tracing.

3 MULTI-PASS LAZY BUILD

Fig. 2 is an overview of our MPLB algorithm. The BLASes are selectively built over the potentially visible instances in the AS build step (Fig. 2(b)) and the instance visibility is updated during the ray traversal step (Fig. 2(c)). Unlike the previous implementations [Benthin et al. 2015; Georgiev et al. 2018], MPLB forms multiple passes in order to avoid complicated ray scheduling. The idea is analogous to recent texture-space shading approaches [Hillesland and Yang 2016; Microsoft 2019], where visibility-driven marking of texels is used to avoid redundant shading before the final rendering.

The core of MPLB is the Build-Traversal loop. First, we build the BLASes for empty instances that were marked as potentially visible in the previous pass. In the second stage, we reshoot the rays to the unfinished pixels, where we use the traversal shader to either record more potentially visible empty instances or complete the pixel. The number of incomplete pixels decreases after each iteration until there are no rays left that traversed an empty instance.

Our MPLB algorithm is based on a hybrid rendering using GPU rasterizer and ray tracing hardware together, which are widely utilized in modern ray tracing games. This is because when creating a G-buffer, the primary visibility of all instances in the scene is easily obtained (Figure 2(a)). Hence, the MPLB algorithm takes the advantages of hybrid rendering and it easily builds the initial AS. Before the first iteration, we mark potentially visible instances in this pre-build heuristic, that we discuss in Sec. 3.3.

Listing 1: Visibility Traversal Shader.

```
1  RWStructuredBuffer<vblas> visibleBlasMap[] ..;
2  RWStructuredBuffer<pmask> pixelMasks[] ..;
3
4  [shader("traversal")]
5  void myVisibilityShader(in RayPayload rp) {
6    uint2 index = DispatchRaysIndex();
7    uint2 size = DispatchRaysDimensions();
8
9    UpdateVisibility(visibleBlasMap, InstanceID(), true);
10
11   // Control BLAS traversal with updating pixel mask
12   RaytracingAccelerationStructure myAccStruct;
13   bool isInstanceEmpty = IsEmptyInstance();
14   if (isInstanceEmpty) {
15     UpdateMask(pixelMasks, index.y*size.x + index.x, false);
16     rp.trav_valid = false;
17     return;
18   }
19   else if (!isInstanceEmpty && !rp.trav_valid)
20     return;
21   else {
22     myAccStruct = FetchBLAS(InstanceID());
23     RayDesc xformedRay = {...};
24     // Set the next level instance and shader table offset
25     SetInstance(myAccStruct, xformedRay, hitShaderOffset);
26   }
27 }
```

3.1 Traversal control and visibility compute

MPLB algorithm takes full advantage of the traversal shader. Listing 1 is the abstracted HLSL code of our traversal shader described with some intrinsic- and user-functions. To record instance visibility, we defined a dedicated data structure called Visible BLAS Map (VBM) (Fig. 3) commonly used in AS builders and traversal shaders as shown in Fig. 2. The VBM contains a flag indicating the BLAS visibility to which each instance refers and two flags indicating whether the BLAS has already been built. In addition, we added a boolean flag, `trav_valid`, to ray payload to keep track of traversal status, which can be used for checking if the ray has encountered an empty instance so far.

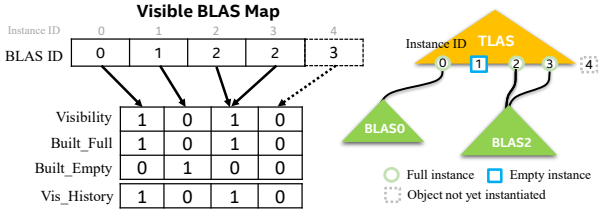

Figure 3: Structure of the Visibility BLAS Map with an example scene hierarchy.

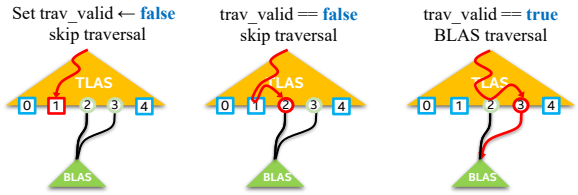

Figure 4: Control of the BLAS traversal based on the instance type and traversal validity.

We conservatively update the visibility in the traversal shader because all traversed instances are potentially visible to the current ray. Hence, the first task is to set the `visibility` flag as True for the corresponding BLAS of the current instance. It also sets the `vis_history` flag as True to reuse it in the next frame (line 9). Next, the traversal destination is determined based on the status of the current instance. This is classified into three cases (Fig. 4):

- **Empty instance**: We reset the pixel mask for reshooting rays in the next pass (line 15), then invalidate the current traversal by setting `trav_valid` flag in ray payload (line 16). Finally we ignores the current instance and continues traversal in the higher-level AS by simply returning.
- **Full instance and invalid traversal**: The current instance has a built BLAS, but the ray has encountered an empty instance so far. Because the ray will be eventually shot again to the current pixel, we can skip the BLAS traversal (line 20).
- **Full instance and valid traversal**: Since the ray normally traversed the AS without empty instances, it fetches the BLAS of the current instance and continues the traversal.

If the ray maintains validity until the end of the traversal, the ray will normally invoke and execute the closest-hit or miss shader. Otherwise, those shaders return control without executing their code and finish the current pass, which prevents the overheads of hardware ray traversal and shader launching for secondary rays. In the next pass, the rays are shot again only to the pixel having the "False" mask, and a valid traversal for those pixels is attempted.

3.2 Build AS for visible instances

We build the BLASes of the instances or create empty instances, depending on the visibility flag of the VBM. The potentially visible instance normally constructs the BLAS (BUILD_FULL), and the invisible instance computes only the bounding box of the geometry and packs it in the leaf node of TLAS (BUILD_EMPTY). We also refer to the other two flags, indicating whether a BUILD_FULL or BUILD_EMPTY action was already performed for the current object. By checking these flags, we can avoid duplicate actions in the different iterations of the Build-Traverse loop.

Once the BLAS build process for the objects is finished, the final AS is constructed by building the TLAS over these BLASes. The TLAS is rebuilt only in the first pass and refitted in the rest of the passes because the bounding boxes of all objects could be already set up in the first pass. See the pseudocode and description of the detailed build process in the supplemental material.

3.3 Pre-build heuristics and masking

Our algorithm forms multiple passes, which makes it sometimes redundantly shoot rays for the same pixel. This is because the current pass should make up for the invalid traversal in the previous pass. This can lead to redundant hardware ray traversal and shader invocations. However, we limit this overhead only to the pixels corresponding to invalid traversal by applying a pixel mask.

Moreover, we propose two heuristics to identify potentially visible BLASes (and build them) even before the first ray is traversed (Fig. 2(a)). Using the G-buffer we can mark directly visible instances that are likely to be traversed by primary rays. Furthermore, we assume that there is a significant amount of frame-to-frame coherence, therefore we also pre-build the BLASes of instances traversed in the previous frame. These two heuristics can greatly reduce the number of Build-Traverse iterations.

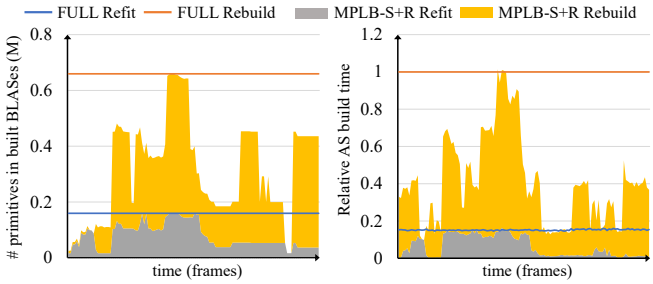

Figure 5: Comparison of the build cost between our MPLB and the conventional full build (stacked plot).

4 APPLICATIONS AND EVALUATION

In order to evaluate our MPLB algorithm, we use the framework of Lee et al. [2019] that is a user-space implementation of an extended programming model and a software ray tracing runtime. This framework also includes a GPU AS builder that is implemented with OpenCL 2.0 based on SAH and refit algorithm [Wald 2007]. We also implemented two dynamic scene scenarios requiring AS rebuild/refit in every frame, which practically happens in games.

4.1 Fully deformable and skinned animation

The first application consists of seven dynamic models (182 objects, 3094 instances, 656K triangles) in the Sponza model (377 objects/instances, 262K triangles) as shown in Fig. 1(a-c). Each dynamic model can be a skinned animation or fully deformable and its BLAS is constructed with a refit or rebuild method respectively. This scene has various occlusion cases through a camera animation.

Fig. 5 compares the build cost (the number of BLAS primitives and relative build timing measured on Intel UHD 630 GPU) of conventional AS build (FULL) and our MPLB during rendering 120 frames (See the supplemental video). We tested four types of secondary rays of shadow (S), shadow and reflection (S+R), ambient occlusion (AO), and two-bounce diffuse indirect rays (GI). These graphs show the result of S+R (See the supplemental material for others). Our MPLB algorithm can significantly reduce the number of primitives compared to the FULL since it excludes invisible objects from the construction process, which directly reflects on the build timing as shown in the left graph. Depending on the ray type, it achieves 3.4x (S), 2.4x (S+R), 2.8x (AO), and 3.1x (GI) reduction in build time respectively. As the number of bounces increases, the number of instances hit by rays also increases, which makes the gains slightly decrease. Nevertheless, our MPLB could build AS significantly faster (3.0x) than the FULL on average.

Table 1: Comparison of the average traversal cost per frame.

		S	S + R	AO	GI
#loops	MPLB	**1.2975**	**1.3483**	**1.1175**	**1.1500**
#rays	FULL	4,100,025	6,727,248	32,800,201	32,800,201
	MPLB	4,100,081	6,728,008	32,800,436	32,800,260
	ratio (%)	**0.0014**	**0.0113**	**0.0007**	**0.0002**
#travs	FULL	49,882,319	88,106,269	499,413,431	494,696,645
	MPLB	49,078,435	87,702,958	509,369,463	512,708,300
	ratio (%)	**-1.6380**	**-0.4599**	**1.9546**	**3.5130**

To analyze the overhead of our algorithm, we measured the number of generated rays, traversal steps, and the iteration counts of the Build-Traverse loop (See Table 1). The average number of iterations is very low (1.1–1.3), which means the MPLB could mostly complete within a single iteration with our pre-build heuristics. Hence, the re-shooting ratio was very low (0.0007–0.0113%) and the increased traversal steps are also minimal (-1.6–3.5%). Interestingly, this could even be reduced in some cases (S, S+R), this is because the initial AS that has only visible instances could also reduce some traversal steps, which ratio overwhelmed the overhead ratio.

4.2 View-Dependent Adaptive Tessellation

In our second application, we use MPLB to avoid the costly evaluation of invisible procedural geometry. We render subdivision surfaces approximated with Gregory patches and integer tessellation [Loop and Schaefer 2008]. Our goal is not state-of-the-art tessellation, but to demonstrate a visibility-driven framework.

The bounds of each patch can be evaluated cheaply based on their control points. During the Build-Traversal loop, we mark potentially visible patches, tessellate them based on the projected edge lengths, and update their BLAS. We only tessellate patches with changed tessellation factors since the previous frame (using a simple hashing scheme), for the rest we update the vertex positions and refit the BLAS. In Fig. 1 we highlight the re-tessellated patches.

The `DragonBaby` model consists of 8154 patches, and for simplicity, each has its own BLAS. The application is fully ray-traced and does not rely on the G-buffer, but uses historic visibility to reduce the overhead of lazy-build. Each patch is dynamically tessellated due to animation. We have an extreme depth complexity of empty BLASes, so marking each of them visible along the rays would result in a large amount of redundant BLASes updates. We address this by limiting the number of newly marked empty instances per pixel, and terminating the traversal upon this threshold. This creates a trade-off between the redundant builds and the MPLB iterations, we have chosen two empirically for our demo.

Fig. 6 shows the build complexity and traversal cost during a camera animation (see the supplemental video). As the camera gets extremely close to certain patches (e.g. the wings) the number of tessellated triangles fluctuates, but on average, lazy build generates about 76% of the triangles compared to frustum culling. The overhead of additional rays is marginal, since most pixels are finalized after the first pass, but the average number of build iterations is significantly higher compared to the first application due to the high depth complexity of the empty instances.

5 DISCUSSION AND FUTURE WORK

We believe that the primary overhead of MPLB comes from the need to relaunch the computation of pixels that encountered an empty instance. This can be removed if the ray tracing hardware provided

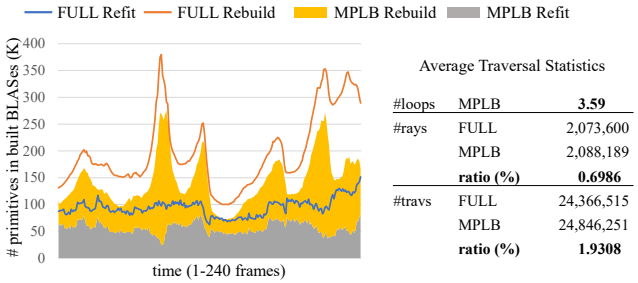

Figure 6: Build and traversal cost of one primary ray per pixel. For full build we used frustum culling for a fair comparison, but this is only feasible for primary rays.

the user with more control over its internal memory management. The user could allocate a pool of memory for storing the context information whenever a ray encounters an empty instance during traversal. In the next pass, the user could restore the context and directly resume traversal from the corresponding point.

One of the limitations of MPLB is that the dependencies in the Build-Traverse loop make it difficult to schedule both tasks asynchronously and utilize the hardware during the BLAS updates. This may be resolved by dividing the screen into tiles and overlapping tasks on a per-tile basis. A more thorough investigation of scheduling would require a closer implementation to the hardware than our current functional simulation.

ACKNOWLEDGMENTS

We thank Karthik Vaidyanathan and members of the Advanced Rendering Team for their insightful feedback, and Sven Woop for his help with the Embree tessellation code. We thank David Blythe and Charles Lingle for supporting this research.

The assets `MARBLES` and `FAIRY` are courtesy of the Utah 3D animation repository, and `EXPLODING-DRAGON`, `CLOTH-BALL`, `BREAKING-LION`, `N-BODY` are courtesy of UNC dynamic scene benchmarks. `BLACK-DRAGON` was modeled by Dennis Haupt and the `SPONZA` scene was modeled by Marko Dabrovic. The `DragonBaby` model is from the Sintel movie by the Blender Foundation.

REFERENCES

C. Benthin, S. Woop, M. Nießner, K. Selgrad, and I. Wald. 2015. Efficient Ray Tracing of Subdivision Surfaces Using Tessellation Caching. In *Proceedings of High-Performance Graphics*. 5–12.

J. Choi. 2020. Ray Traced Reflections in 'Wolfenstein: Youngblood'. GDC.

I. Georgiev, T. Ize, M. Farnsworth, R. Montoya-Vozmediano, A. King, B. V. Lommel, A. Jimenez, O. Anson, S. Ogaki, E. Johnston, A. Herubel, D. Russell, F. Servant, and M. Fajardo. 2018. Arnold: A Brute-Force Production Path Tracer. *ACM Transactions on Graphics*, 37, 3 (2018).

K. Hillesland and J. Yang. 2016. Texel Shading. In *Eurographics 2016 – Short Papers*.

W. Hunt, W. R. Mark, and D. Fussell. 2007. Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*. 47–54. https://doi.org/10.1109/RT.2007.4342590

Khronos. 2020. https://www.khronos.org/blog/ray-tracing-in-vulkan

W.-J. Lee, G. Liktor, and K. Vaidyanathan. 2019. Flexible Ray Traversal with an Extended Programming Model. In *ACM SIGGRAPH Asia 2019, Technical Brief*. 17–20.

Charles Loop and Scott Schaefer. 2008. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Transactions on Graphics*, 27, 1 (2008), 8:1–8:11.

Microsoft. 2018. DirectX Raytracing (DXR) Functional Spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

Microsoft. 2019. DirectX Sampler Feedback Functional Spec. https://microsoft.github.io/DirectX-Specs/d3d/SamplerFeedback.html

J. Schmid. 2019. It Just Works: Ray-Traced Reflections in 'Battlefield V'. GDC.

I. Wald. 2007. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*. IEEE Computer Society, 33–40.