DirectX Raytracing: Performance of updating acceleration structures

Alexander Wester
Blekinge Institute of Technology
Email: alwe15@student.bth.se

Fredrik Junede
Blekinge Institute of Technology
Email: frju15@student.bth.se

Henrik Johansson
Blekinge Institute of Technology
Email: hejc15@student.bth.se

I. Introduction

Real-time ray tracing for games is currently gaining popularity among many people. This started when Nvidia released their RTX series graphics cards which have dedicated hardware for accelerating ray tracing. There are still many parts to performing real-time raytracing and the new hardware mainly accelerates the scene traversal using a bounding volume hierarchy (BVH). Our hypothesis is that there are still important parts that will be slow even when using the new hardware. Specifically using the DirectX Raytracing (DXR) and dynamic models which creates the need for frequent updates to the internal ray tracing acceleration structures. Skeletal animations updated on the CPU are used to illustrate this frequent need to update and timings are measured.

According to Dunn, an Nvidia employed engineer, a developer needs to know when to rebuild and when to update their bottom layer acceleration structures (BLAS) [1]. As such, it is expected that building BLASes will be slower than updating them in place. Furthermore, it is anticipated that these updates and rebuilds will take a large chunk of time in the ray tracing pipeline [1]. As Dunn further mentions that it is not required to update the top layer acceleration structure (TLAS) but to rather rebuild it for convenience it is expected that this part of the pipeline will not take much time from the execution.

During the measurements a few different independent variables were in focus. Firstly, how long it takes to build the top layer acceleration structure. Secondly, how long all of the bottom layer acceleration structures take to build in milliseconds.

II. Implementation Methodology

The application was designed to fully test our hypothesis by using only ray tracing on a dynamic scene. Every pixel on the screen dispatched a ray in full resolution. The acceleration structure build times are measured using GPU query timers [2]. These measured the time it took the GPU to execute the individual tasks, therefore the frame times are not relevant. Building acceleration structures was tested with a variation of the flags *PERFORM_UPDATE*, *ALLOW_UPDATE*, *FAST_BUILD* and *FAST_TRACE*. All The flags written above in italic has the internal prefix of *D3D12_RAYTRACING_ACCELERATION_-STRUCTURE_BUILD_FLAG_* in code.

A. Animations and vertex buffer updates

Skeletal animations were used to create the dynamic scene, one model and one animation were used to create all the duplicate models in the scene. One vertex buffer were made for each object. The models and the vertex buffers were updated once per frame.

B. Testing setup

The hardware used in the tests was a Nvidia RTX 2060 Founders Edition graphics card with 6GB of VRAM.

When timing the TLAS and BLAS updates the application was set up to add 10 objects every 500 frames. Each object is the same model containing 7313 vertices. The different parts of the ray tracing pipeline were timed and averaged during these 500 frames. Each test was done in the same viewing angle as seen in fig. 1.

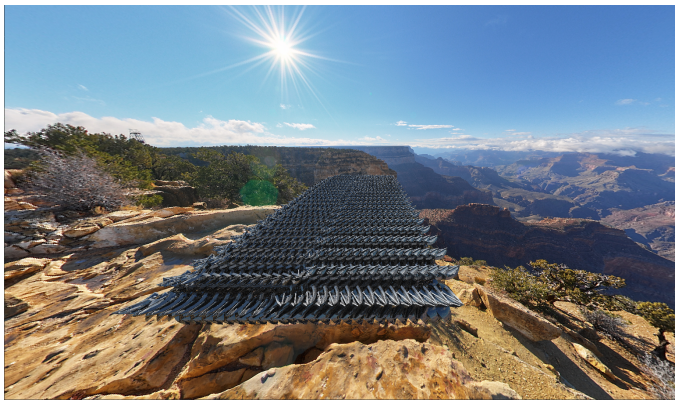

Fig. 1. Test scene with 500 objects

C. Acceleration Structures (AS)

The acceleration structures is what contains all the geometry in the game world that later be intersection-tested against when rays are dispatched. DXR has two different types of acceleration structures, top level and bottom level.

1) Top Level Acceleration Structure: The TLAS can be seen as an acceleration structure over acceleration structures [3]. It contains underlying bottom level acceleration structures. A TLAS can instantiate the same BLAS multiple times, although this is not used in the tests performed. The implemented TLAS contains only one instance of each BLAS

because the focus is on testing BLAS update times. Instancing should otherwise be used where possible for performance improvements.

2) Bottom Level Acceleration Structures: The geometry in each BLAS is defined using vertex and index buffers, the same buffers are also bound to the local hit group for shader access. All geometry inside a single BLAS is transformed with the same transformation matrix, therefor each BLAS in this implementation is one mesh. The connections between top level and bottom levels can be seen in figure 2.

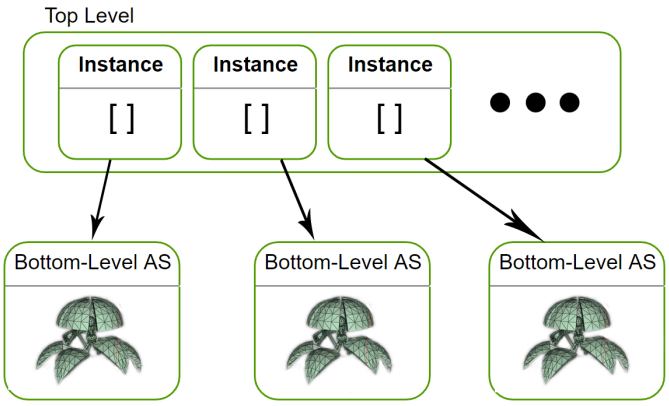

Fig. 2. Acceleration structure setup

III. Result

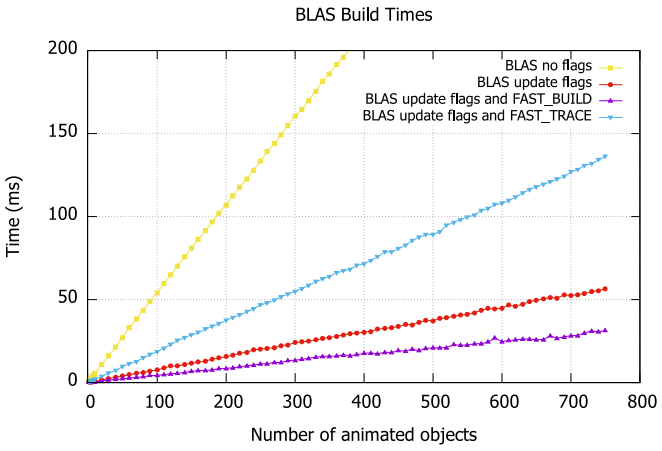

Fig. 3. Timing results for building BLASes

The results show that the most expensive part of the raytracing is rebuilding bottom layer acceleration structures. The time it takes grows linearly with the number of objects. Using the flags *ALLOW_UPDATE* and *PERFORM_UPDATE* for BLAS builds that only moves existing vertices (no removal or insertions) speeds up performance with a factor of 2.9x to 12.8x, depending on which other flags are set. Since the data grows linearly, an average update time per object can be calculated. Using only the update flags, the BLAS updates took on average 0.076ms/object. The fastest results where reported using the update flags together with *FAST_BUILD* with an average of 0.042ms/object. Switching out *FAST_BUILD* with *FAST_TRACE* increases the average update time to 0.181ms/object. Nvidia recommends that developers optimize their AS build/updates to take at most 2ms [1]. Table I shows how many of the object used in this test could build/update with the different flags within the recommended time.

TABLE I
AVERAGE TIME PER OBJECT

Flags	Time (ms)	Max objects within 2ms
Update | FAST_BUILD	0.042	47
Update	0.076	26
Update | FAST_TRACE	0.181	11
NONE	0.52	3

Update = PERFORM_UPDATE | ALLOW_UPDATE

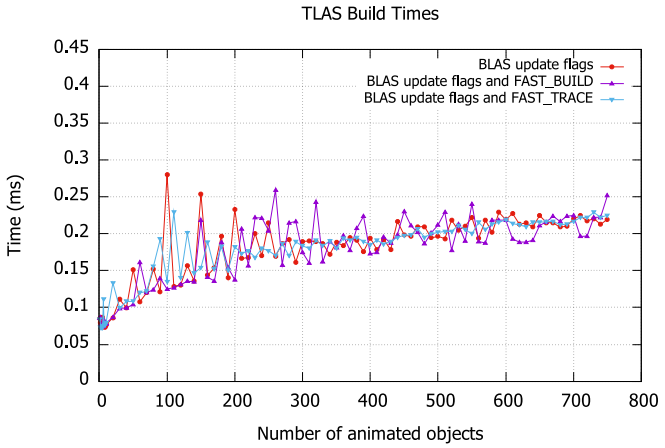

Fig. 4. Timing results for building the TLAS

The top level acceleration structure was rebuilt every frame. Results show that using different flags when building the BLAS does not change TLAS update times.

IV. Analysis

A. Synchronization between queues

Letting each queue wait for the other makes for not so great parallelism as it simply stops execution until a previous queue has finished executing its commands as seen in fig. 5. These waits are however required in the current implementation as the different queues accesses and processes the same resources, having them do this concurrently would result in memory race hazards.

Another possibility to improving parallelism would be to split up the vertex buffer updates and BLAS updates into multiple chunks, one being run asynchronously from another. Each chunk would contain one set of vertex buffers and one set of BLASes and as such the parallelism should be increased as

the copy queue should be able to be ran asynchronously from the compute queue.

Fig. 5. Execution of a frame

B. Synchronization between CPU and GPU

Making sure the GPU does not have to wait for updates being done on the CPU can be essential to acquiring high FPS count, i.e. keeping the GPU busy. In this implementation this is not done as looking at FPS was not an interesting measurement for the hypothesis since focus is on the performance of the different individual parts of the ray tracing pipeline. This is an apparent bottleneck and skinning should be done on the GPU, and the GPU should never have to wait for the CPU unless it is of absolute necessity [4].

V. Conclusion

A. BLAS build times

The results has shown that using different flags varies the BLAS build times by many factors. It is important to use the update flags and *FAST_TRACE* where possible, therefor developers needs to keep track of which geometry will be static, updated rarely or updated often. When to use the different flags and what developers need to think about is well documented in a post by Nvidia about best practices in ray tracing [1].

B. Hybrid solutions

While looking at the different results it shows that ray tracing still has some ways to go, but that it has a place in the future of real-time rendering. As mentioned previously, building bottom level acceleration structures can be very expensive, but it is required when the full frame of a dynamic scene is ray traced. This is where hybrid solutions come in where the majority of the rendering can be done using rasterization and ray tracing can be used only on specific parts where the technology shines. Major areas where ray tracing can produce more realistic results are reflections [5], global illumination [6] and shadows [7]. Using ray tracing only for specific parts means that the AS will not have to be updated every frame, and only parts of the scene could be updated at a time. This approach can be seen in Metro Exodus [8].

C. Grade

As we began our project we decided right away that we were going to go for grade A, and we believe we've reached that goal.

1) Requirements: In rasterization we implemented multi threaded command recording for drawing out meshes. It splits up all the objects into 4 different groups where each group is handled by one thread each, recording all commands in separate lists that are later sent to and executed on the direct command queue in the main thread.

By updating our animations on the CPU we make sure that we can make use of the copy queue. As we have implemented ray tracing using D3D12's DXR it made perfect sense for us to use the compute queue since operations like this is where the compute queue excels [1]. At the end of each frame we use the direct queue in order to copy the results generated by the compute queue and present them on the screen.

References

[1] A. Dunn, "Tips and tricks: Ray tracing best practices," Mar 2019, accessed: 2019-03-26. [Online]. Available: https://devblogs.nvidia.com/rtx-best-practices/

[2] Microsoft, "Timing," May 2018, accessed: 2019-03-25. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/direct3d12/timing#timestamp-calibration

[3] M.-K. Lefrançois and P. Gautron, "Dx12 raytracing tutorial - part 1," Sep 2018, accessed: 2019-03-25. [Online]. Available: https://developer.nvidia.com/rtx/raytracing/dxr/DX12-Raytracing-tutorial-Part-1

[4] L. Kavan, S. Collins, J. Žára, and C. O'Sullivan, "Skinning with dual quaternions," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. ACM, 2007, pp. 39–46.

[5] A. Burnes, "Atomic heart adds nvidia rtx real-time ray tracing – see the stunning results in our exclusive video," Aug 2018, accessed: 2019-03-26. [Online]. Available: https://www.nvidia.com/en-us/geforce/news/atomic-heart-rtx-ray-tracing/

[6] ——, "Metro exodus enhanced with nvidia rtx ray traced effects – see them in action in our exclusive tech video," Aug 2018, accessed: 2019-03-26. [Online]. Available: https://www.nvidia.com/en-us/geforce/news/metro-exodus-rtx-ray-traced-global-illumination-ambient-occlusion/

[7] ——, "Shadow of the tomb raider updates adds ray-traced shadows and dlss," Mar 2019, accessed: 2019-03-26. [Online]. Available: https://www.nvidia.com/en-us/geforce/news/shadow-of-the-tomb-raider-nvidia-rtx-update-out-now/

[8] S. Petersson, "Triangelplockaren, metro exodus, part 1 (dxr)," Feb 2019, accessed: 2019-03-26. [Online]. Available: https://www.youtube.com/watch?v=SgKeV0yF25Y

Appendix A
Source code

https://github.com/Piratkopia13/DV2551_Project_DXR