Towards Ray Tracing Benchmark Using Video Games Data Extraction

Jakub Profota*
Supervised by: Jiří Bittner†

Department of Computer Graphics and Interaction
Czech Technical University in Prague
Prague / Czech Republic

Figure 1: A ray-traced scene from Cyberpunk 2077 (left) and its underlying ray tracing acceleration structure (right) as visualized by NVIDIA Nsight Graphics [29] frame capture tool.

Abstract

We propose a novel benchmarking concept for ray tracing and report initial results obtained by the underlying software framework. We aim to create an alternative to the traditional models used in research by leveraging assets and scenes from modern video games. We describe the methodology for extracting geometry by injecting a custom tool into a modern ray-traced game and present preliminary results. Should the respective rights holders agree and grant us explicit permission, we would be pleased to release the captured datasets that reflect current industry standards to the research community.

Keywords: Ray Tracing, Benchmark, Video Games, Data Extraction

1 Introduction

Recent years have witnessed a shift towards ray tracing in video games. Coupled with AI-driven techniques such as upscaling and frame generation, modern GPUs enable games to achieve real-time ray tracing performance. Sophisticated lighting previously restricted to offline rendering, such as dynamic global illumination, physically accurate soft shadows, and reflections, can now be realized without the approximate workarounds of rasterization.

Despite the computational demands, ray tracing offers an advantage in algorithmic elegance and simplicity. In contrast to complex rasterization techniques, ray tracing produces accurate advanced lighting effects with significantly reduced logical complexity. Modern graphics APIs, such as DirectX and Vulkan, use concise ray tracing abstractions that rely on programmable shaders.

The recent influx of ray-traced games [31] and the relative simplicity of the ray tracing pipeline raise a possibility that extracting data from games programmatically and on demand for benchmarking purposes is feasible. Modern AAA games are the culmination of hundreds of person-years of artistic labour, producing scenes of a scale and fidelity that researchers cannot replicate (see Figure 1). Additionaly, traditional ray tracing datasets used in research are meshes without duplicate geometry instantiation, a technique broadly used by graphics APIs.

Our paper provides the following contributions: it presents a brief overview of benchmarks and relevant metrics. It addresses the limitations of traditional benchmark datasets. It investigates existing tools and their data extraction capabilities, and details a methodology for extracting geometry by intercepting DirectX 12 API calls. Finally, some preliminary results are shown and discussed.

2 Related work

Meister et al. [22] conducted a survey covering the bounding volume hierarchy (BVH), the data structure used

*profojak@fel.cvut.cz
†bittner@fel.cvut.cz

by most offline and real-time ray tracing applications. The authors describe the foundational principles, state-of-the-art construction and traversal algorithms, specialized hardware, and industrial frameworks. They provide an overview of best practices, including a two-level BVH hierarchy well-suited to dynamic scenes with rigid-body transformations, as are typically found in video games. Such a technique is part of the DirectX 12, the most widely used graphics API for ray-traced games [35].

2.1 Metrics

Qualitative metrics predict the performance and guide BVH construction. A traditional metric is the surface area heuristic (SAH) [22], which represents the expected cost of tracing a random ray. SAH assumes rays are uniformly distributed, their origins are outside of the scene's bounding box, and the rays are not occluded. These assumptions are rather unrealistic, but work well in most cases.

Aila et al. [3] noted that SAH is not entirely accurate. The authors introduce two new metrics: end-point overlap and leaf count variability. When combined with SAH, these metrics can correlate better with observed ray tracing performance. Makarov [19] suggests optimizing top-level BVH (TLAS) in a two-level hierarchy by applying a global transformation that rotates the entire tree. This approach aims to axis-align as many bottom-level BVH (BLAS) instances as possible. Havran et al. [11] addressed the challenge of fairly comparing various ray-shooting algorithms. The authors reduce dependence on specific hardware, compilers, and implementation quality by introducing an evaluation methodology that features an ideal reference ray-shooting algorithm. They mandate the reporting of three subsets of properties: static, dynamic, and hardware and implementation-dependent.

2.2 Datasets

Lext et al. [15] created a dataset for animated ray tracing with three parametrically animated scenes. Applicable to static datasets as well, the authors highlight scenarios that can negatively impact the performance of ray-tracing algorithms. These challenges, called stresses, should be reflected in a benchmarking dataset to ensure its ongoing relevance and future difficulty. Examples of such stresses include a teapot in a stadium, overlaps in bounding volumes, and variations in object distribution.

Opening any ray tracing research paper often reveals a benchmark involving some of the commonly used test scenes. Armadillo, Happy Buddha, Dragon, Stanford Bunny [14], Erato [20], Crown [32], and similar models are finely tessellated objects. However, object-centric scenes are not ideal representatives of typical ray-tracing workloads, as most rays escape the scene after the first bounce. Fairy [8], Landscape [32], or Vegetation have widely varying triangle sizes and shapes of plants. Some objects, such as Hairball [20] or Sheep, are explicitly designed to stress-test algorithms against triangle edge cases.

Conference, Sibenik, Crytek Sponza [20], and similar architectural models are widely used, but lack spatial extent, spanning only a few tens of meters. Larger models, such as Powerplant and Rungholt [20], lack the hierarchical level of detail typical of video games. Aila et al. [3] considered San Miguel [32], which combines architecture with detailed vegetation, to be the best representative of a realistic model, but later released Bistro [17] (see Figure 2) may now be the best.

Figure 2: Amazon Lumberyard Bistro [17].

Some production-ready scenes, such as Caldera [2] and Moana Island [36], are available. The datasets are massive in spatial extent and storage size of tens of GBs, making them quite challenging to work with. Caldera (see Figure 3) features multiple levels of detail meshes, which can be used to simulate a game-like scene. The main issue is, however, that datasets like these are very uncommon.

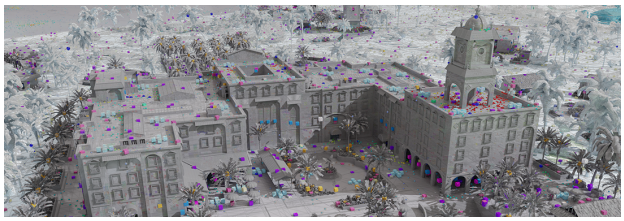

Figure 3: Activision Caldera [2].

We investigated open-source game reimplementations and demo game engine projects. The reimplementations provide access to the code [6] [30], but game assets remain protected by copyright and are thus not distributed. Demos for software like Godot [9] and Blender [7] come with permissive licenses and could be usable. In contrast, Unity [37] and Unreal [10] engines provide game-like demos under strict end-user license agreements that forbid the use of their assets outside the respective engines.

2.3 Benchmarks

Vinkler et al. [38] published a GPU performance comparison of BVHs and kd-trees, the previous ray tracing data structure of choice. By profiling the execution of GPU kernels, the authors show that kd-tree traversal tends to be more memory-limited due to higher data transfer volumes

and lower cache hit ratios. BVH has a predictable memory footprint, whereas kd-tree cannot estimate the number of triangle references in advance. The authors report ten times as many triangle references and longer build times for kd-trees than for BVHs on traditional benchmark models. Kd-trees also experienced a higher rate of divergent branching, hindering their parallel processing efficiency.

Meister et al. [21] conducted an empirical comparison of popular BVH variants to evaluate performance in GPU ray tracing. They assessed a range of popular construction algorithms, along with various enhancements such as spatial splitting, ray reordering, and wide BVHs. The authors propose a global optimization method utilizing simulated annealing to move closer to a theoretically optimal BVH.

Wald et al. [40] documented the shift of ray tracing from a strictly offline method to a viable interactive technique. The authors argue that ray tracing provides superior scalability compared to rasterization, with render times increasing logarithmically rather than linearly with a scene complexity. Their work details the use of ray packets and CPU-level SIMD execution. Wald et al. [39] noted that the main bottleneck in interactive ray tracing transitioned from intersection tests to the per-frame maintenance of acceleration structures. They evaluate the trade-offs between rebuilding BVHs from scratch and updating them.

Liu et al. [16] created LumiBench, a benchmark suite for hardware ray tracing. It offers a wide range of scenes and shaders that simulate various types of rays, including shadow and ambient occlusion rays. The solution features a software simulation framework to test hardware architectural designs. The suite reports hardware-related metrics, such as DRAM utilization, cache hits, ray tracing unit utilization, or SIMT efficiency.

3 GPU Inspection Tools

We investigated existing frame-capture tools and their data extraction capabilities. The most suitable tool for the task is the open-source GFXReconstruct (GFX) [18], a collection of graphics API capture and replay tools from LunarG. The tool lacks a graphical user interface, but supports capturing both Vulkan and DirectX ray tracing. GFX relies on the user to copy the interception dynamic-link libraries (DLLs) into a game directory, which we find inferior to programmatically injecting them. There is some DLL injection capability, but is undocumented. The authors do not accept new API hook pull requests, such as NVIDIA's NVAPI [28] used by some ray-traced games, so we would have to maintain a fork. Nevertheless, we consider GFX a fantastic effort and keep it as a reference.

RenderDoc [34] is a well-known open-source tool. While it captures DirectX 12, it does not support the DXR ray tracing feature [27]. The tool provides a scriptable API usable for extracting geometry. However, the geometry obtained from the rasterization pipeline lacks useful data, such as information about BLAS instances in the DirectX two-level hierarchy ray tracing model. In fact, when inspecting a frame capture of the rasterization pipeline, we observed chunks of meshes being rasterized, while a ray tracing pipeline processes the entire model.

NVIDIA Nsight Graphics (Nsight) [29] is a proprietary tool that captures both the DirectX 12 and Vulkan ray tracing pipelines. The tool allows users to interactively inspect the acceleration structure (see Figure 4) and browse vertex and index buffers via user interface. However, it does not support exports. The format of the capture is proprietary and intended for use within the capture tool. Interestingly, the tool supports shader debugging, but the host system requires two NVIDIA GPUs. We have not tested that.

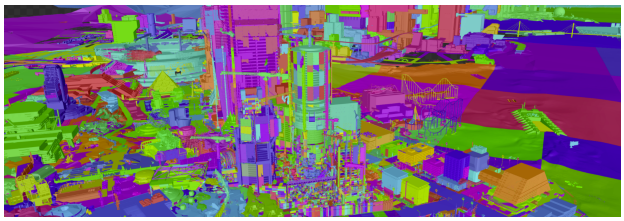

Figure 4: An interactive view of ray tracing acceleration structure of Cyberpunk 2077 as visualized by Nsight [29].

In contrast to Nsight, AMD Radeon Raytracing Analyzer (RRA) [4] displays the BVH tree's actual hierarchy. To our knowledge, the tool does not support extracting data, but we did not verify that ourselves as the machines we are using are not equipped with AMD GPUs.

Intel Graphics Performance Analyzer (GPA) [12] is a solution that has been recently deprecated. We did not have much success with capturing ray-traced games.

Microsoft PIX [24] is a performance and debugging tool designed for DirectX 12. To use this tool to inspect the application, the game must link to a debug DLL provided by Microsoft. Interestingly, one of the games we tested included this DLL in its files, which enabled a successful frame capture. Unfortunately, we were unable to access an interactive view of the BVH due to PIX crashes.

Table 1 summarizes tool capabilities. One intriguing feature of Nsight and PIX is their ability to generate a C++

Frame Capture	DR	VR	UI	C++	API	Availability
GFX	✓	✓	✗	✓	✓	Open-source
RenderDoc	✗	✗	✓	✗	✓	Open-source
Nsight	✓	✓	✓	✓	✗	Proprietary
RRA	✓	✓	✓	✗	✗	Proprietary
GPA	✗	✗	✓	✗	✗	Deprecated
PIX	✓	✗	✓	✓	✗	Proprietary

Table 1: An overview of frame capture software. The table shows whether the tools support the DirectX 12 (DR) and Vulkan (VR) ray tracing pipeline, they use a graphical user interface (GUI), they support exporting a standalone C++ project from the frame capture, and whether they support scripting (API). Their availability is also listed.

project, allowing users to compile and run a single frame of the captured application. GFX marks this feature as experimental. All the data, including geometry, is saved into a multi-gigabyte binary file, which is accessed from one of the numerous auto-generated source files. The code spans tens of thousands of lines. We did not study the code thoroughly because it is somewhat obfuscated.

4 Implementation

We are building a tool that injects API hooks into a ray-traced game to track and export the geometry used to build the ray-tracing acceleration structures. Our solution targets 64-bit Windows games using the DirectX 12 graphics API and NVAPI, which extends DirectX 12 with, for example, additional geometry primitives. While it is feasible to play ray-traced games using Vulkan, either natively or through DirectX translation layer [5], possibly even on Linux [33], the DirectX 12 and Windows remain the dominant platform for games.

4.1 Hook Injection

The tool launches a suspended game and injects a payload DLL that hooks `LoadLibrary` and `CreateProcess` Win32 [25] API calls. The game is then resumed, entering its `main` function. Whenever the game starts a new process, unless blacklisted like a game launcher or Steam overlay, the payload gets reinjected. When a DLL is loaded through `LoadLibrary`, the hooked variant checks whether the library is of interest, i.e., it is `d3d12.dll`, `dxgi.dll`, or `nvapi.dll`, and intercepts corresponding top-most API calls (see Figure 5). Subsequent hooks are injected lazily whenever a parent API call is executed, i.e., `ID3D12CommandList` methods are hooked when `CreateCommandList` method creates a command list (see Figure 6).

We use Microsoft Detours [26] instrumentation to inject the hooks. Because graphics API DLLs are loaded at runtime, the addresses of the API calls are not known before

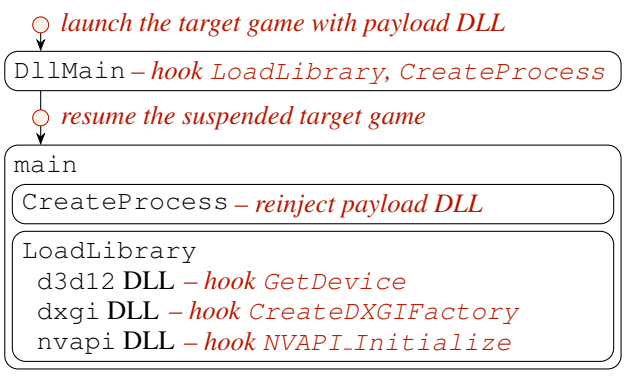

Figure 5: Schematic of top-most API hook injections.

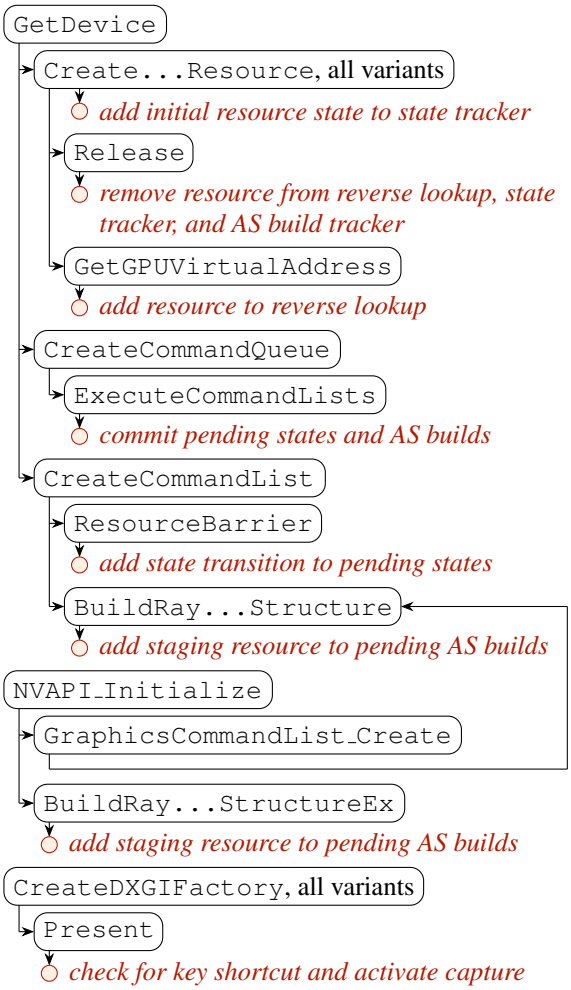

Figure 6: Schematic chain of interceptions, hooked lazily from the top-most API calls, and the purpose of the hooks.

the game's execution. For free-standing functions, the addresses are obtained using Win32's `GetProcAddress`. To hook DirectX 12 interfaces, it is necessary to retrieve addresses of member methods from the virtual method table (VTable) of instantiated objects that implement those interfaces. We obtain the VTable indices by parsing `d3d12.h` and all the `dxgi.h` header files. NVAPI DLL does not export individual API calls, so we disassemble the library binary provided under the MIT license [28] to obtain the required hard-coded addresses. All NVAPI API calls are free-standing functions. While the DirectX 12 VTable indices are stable throughout versions, we are unsure about NVAPI addresses, so we provide the necessary extraction scripts for users to mitigate any issues.

4.2 Resource Tracking

Plenty of game data, such as geometry, resides in buffers allocated on the GPU. In DirectX 12, these are called resources. The `ID3D12Resource` interface is used, for example, to free the buffer via `Release` or to obtain

the resource's D3D12_GPU_VIRTUAL_ADDRESS. GPU memory is managed explicitly by the user. For example, to build an acceleration structure for ray tracing using the BuildRaytracingAccelerationStructure API call, the user must allocate a resource large enough to hold the resulting acceleration structure and pass its GPU address together with the GPU addresses of resources of vertex and index buffers as parameters.

Unfortunately, it is not possible to obtain the resource from the GPU address via the API. Moreover, resources may be aliased, i.e., a resource may be allocated to hold additional resources, thus multiple resources may share the same GPU address. We maintain a reverse lookup by hooking GetGPUVirtualAddress. We store a list of ID3D12Resource and their sizes in a sorted map, keyed by GPU address, to get a fitting resource.

To transfer the data to the CPU, we need to create a temporary resource on a readback heap, a type of GPU memory accessible by the CPU. We copy the data from the resource we obtained via reverse lookup into the temporary. However, before commencing the copy, the resource must be transitioned to a specific state, COPY_SOURCE, and then transitioned back to the previous state to avoid breaking internal DirectX state synchronization. Unfortunately, it is not possible to get the current state of a resource via the ID3D12Resource interface, so we need to track the states as follows:

Our tool hooks CreateCommittedResource and other resource creation API calls to track the initial resource state. ResourceBarrier is intercepted to track subsequent state changes. However, we cannot update the state globally in the ResourceBarrier hook, because calling this API only stages the resource state transition command to the command list, which is executed on the GPU later via ExecuteCommandLists. Thus, we also track pending local states for each command list separately and update the global resource state tracker only when executed.

4.3 Geometry Capture and Extraction

Top-level acceleration structure (TLAS) is built using the BuildRaytracingAccelerationStructure API call, same as the bottom-level acceleration structure (BLAS). TLAS instantiates previously built BLASes by specifying their GPU addresses, transformation matrices, and other data as function parameters, which we can read directly in the build call hook. Using reverse lookups, we can obtain BLAS resources. Unfortunately, we are unable to interpret the resource's data because the internal representation of the BVHs is undocumented and implementation-dependent. Interestingly, the BVH internals vary across GPUs but also between driver versions [13]. Thanks to RADV [1], the open-source Vulkan driver adopted by AMD, we can inspect the BVH node structure and the BVH construction implementation within compute shaders on AMD GPUs.

BLASes do not retain any references to the resources with geometry. Thus, the only way to access the geometry is to intercept the build call. BLASes are also self-contained, so we need to copy the data from the input resources into temporary staging buffers, since the resources may no longer exist after the build call. Current ray-traced games use a hybrid rendering pipeline: rasterizing all geometry, then computing advanced lighting effects via ray tracing. The geometry resources are compatible and shared between the two rendering stages, but we copy them to a temporary buffer anyway to comply with the DXR specification [27].

Most BLASes are built when the game loads. Each frame, TLAS, and a few BLASes are rebuilt, usually representing dynamic geometry like vegetation or animated characters. During regular gameplay, many BLASes are overwritten with new BLASes as the game scene around the player changes. We track these overwrites to garbage-collect staging geometry buffers for BLASes that no longer exist. As with the resource state tracker, we cannot delete overlapped BLASes in the hooked call. Instead, we track pending builds locally for each command list and commit them to the global acceleration structure tracker once the command lists are executed.

The tool starts in tracking mode. In the Present hook, we check for a shortcut key press to commence the capture. The tool switches to write mode, waits a single frame until the next Present call to allow all the pending TLAS and BLAS builds to be committed to the global tracker, and then writes the tracked data to disk. The process is straightforward: a single temporary command list is created, along with temporary CPU-readable resources for each staging geometry buffer. Copy commands are issued, and the command list gets executed. A binary file containing all the captured geometry is now on the disk, ready to be parsed into a readable format such as OBJ or OpenUSD by another command-line tool.

5 Results

We tested our tool on a PC equipped with NVIDIA GeForce RTX 3070 and AMD Ryzen 5 5600X with 32 GB of RAM. The frame rate averages 39 FPS in Cyberpunk 2077's built-in benchmark at maxed-out settings with path tracing, in a 1600×900 window without frame generation or upscaling. The GPU's 8 GB of DRAM are fully utilized, along with $\approx$500 MB of shared memory, which Windows automatically reserves on the CPU's RAM for the GPU's use. The game with the injected tool runs reasonably well even on the GPU with limited DRAM, averaging 12 FPS in the built-in benchmark. The shared memory GPU consumption rises to $\approx$1 GB. Since the DRAM is full and all geometry copies of newly built BLASes are forwarded to RAM, this is the worst-case scenario, but it still runs interactively. GPUs with ample DRAM experience less performance degradation.

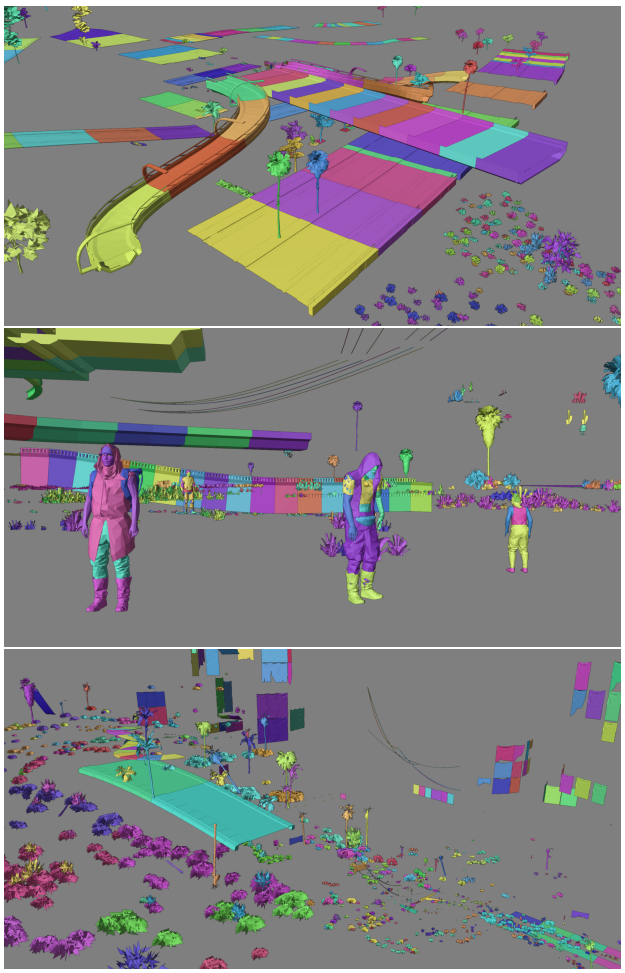

Figure 7: Partial geometry of the three captured scenes (see Table 2 and 3) as visualized in Blender, with 5.5% – 17% of all unique objects and 1.8% – 12.8% of all instances of objects exported to OBJ and shown. Each unique color refers to a unique BLAS. Bushes in the lower left corner of the third image highlight the usage of BLAS instantiation, with multiple objects having the same color.

Exporting to disk takes a few seconds. The game freezes and becomes unresponsive, then recovers and continues without issue. The dump file is a ≈500 MB binary, and partial scenes in Figure 7 are ≈ 65 to ≈ 125 MB OBJ files.

While we dump most of the geometry data as a binary blob, we have yet to interpret it in a separate parser tool. Figure 7 shows a portion of three game scenes, with parsed single-mesh BLASes for DirectX's `TRIANGLES` primitives, indexed by 4-byte indices and without the internal BLAS transformation matrix. Other primitives include NVAPI's `TRIANGLES_EX`, `OMM_TRIANGLES_EX`, and `DMM_TRIANGLES_EX`, and some non-triangles. BLASes can also hold multiple meshes, the indices can be in 2-byte format, and they may include a transformation matrix. Statistics of the partial exports are reported in Table 2.

The number of BLAS addresses in a TLAS varies widely across Cyberpunk 2077's scenes, ranging from

Geometry Metric	Scene 1	Scene 2	Scene 3
Unique objects	390	472	792
Total objects	677	3,810	2,460
Total vertices	792,767	932,937	1,312,385
Total triangles	650,225	614,522	1,259,128

Table 2: Statistics of partial OBJ exports of the three captured scenes from Cyberpunk 2077. A unique object is a BLAS with unique address, and multiple object instances correspond to multiple BLAS instances in the TLAS.

BLAS Metric	Scene 1	Scene 2	Scene 3
Total BLAS addresses	38,613	29,715	35,344
Unique BLAS addresses	6,878	4,677	4,637
Single BLAS instances	3,468	2,785	2,634
Multiple BLAS instances	3,410	1,892	2,003
Average BLAS instances	5.61	6.35	7.62
Maximum BLAS instances	511	788	925

Table 3: Statistics of ray tracing acceleration structures of the three captured scenes from Cyberpunk 2077.

fewer than ≈ 25,000 up to ≈ 42,000, judging from debug logs when running around in the game. We captured three scenes and examined them closely (see Table 3). Among all BLAS addresses referred to in the TLAS, only 13% – 18% are unique, meaning each BLAS is reused and instantiated on average 5.6 – 7.6 times. 50% – 60% of BLASes are instantiated only once, and the maximum number of instances per BLAS ranges from 511 to 925.

In Cyberpunk 2077's built-in benchmark, we tracked 6,730 to 7,280 unique acceleration structures per frame, a number higher than for the captured in-game scenes. During the benchmark run, there are ≈ 2.1 BLAS updates each frame on average, plus the TLAS. The number of updated BLASes ranges from 0 to 4 per frame. During regular gameplay, it can reach two digits as new levels of detail meshes are loaded and their respective BLASes are built.

6 Conclusion and Future Work

In this paper, we outline the rationale for creating a new dataset to benchmark ray tracing algorithms by documenting the geometry models currently used. We evaluate existing software tools for their capabilities to extract data from video games and discuss the technical challenges encountered during the development of our own solution, presenting some preliminary results.

The tool targets a small subset of the graphics API calls and is thus quite performant. Given that the geometry is passed to BVH build calls in a format suitable for rasterization, we intend to visualize the TLAS by creating a new window to render it alongside the actual game in real time. Our goal is to release not only the meshes but also the two-level BVH hierarchy to support research on hierarchical

optimization. To our knowledge, such a dataset does not exist. While we are currently focusing on a single frame, we also plan to export geometry across multiple frames to support research in animated ray tracing.

6.1 Rays and Hits Capture

The next step involves attempting to transpile and emulate the shaders to obtain rays and their corresponding hits. We plan to augment the geometry datasets with this metadata to enable deterministic ray tracing benchmarks. Theoretically, emulation would also allow for the extraction of lights, which could be added to the geometry without the rays, producing yet another variant of the dataset.

The DXR [27] pipeline is invoked similarly to rasterization and compute: set a pipeline state with the shaders on a command list via `SetPipelineState()`, then call `DispatchRays()`. The dispatch invokes a grid of ray-generation threads, where each knows its location in the grid and generates arbitrary rays via `TraceRay()` HLSL intrinsic function. Ray has an origin, direction, and parametric interval in which intersections may occur. A ray is accompanied by a user-defined payload that can be modified as the ray interacts with geometry and is also visible to the caller of `TraceRay()` upon its return. The intrinsic function supports ray flags to override transparency, culling, and early-out behavior. Geometry instances in BLAS contain a user-defined mask, which is ANDed with the `TraceRay()` argument to consider that geometry for intersection.

Any hit and closest hit shaders are part of a hit group. Each geometry refers to a hit group to provide the code to execute. Ray generation and miss shaders are shared. Applications control when the shader compilation occurs. The initial HLSL shader compile to DXIL [23] format can be done offline without any knowledge of the hardware driver. When `CreateStateObject()` creates a DXR pipeline, the DXIL is compiled and optimized to the GPU machine code of the host system. We can hook the call to extract the DXIL, which is well-defined and based on LLVM IR.

`TraceRay()` HLSL call is part of a compiled shader executed on the GPU, making it impossible to intercept in the same way as CPU-side API calls like DirectX. Curiously, the HLSL standard defines `print` function that would straightforwardly solve the problem. However, we do not know any GPU supporting this feature. We investigated reusing the user-defined ray payload by programmatically editing the DXIL to extract ray information. We ruled this option out because we cannot determine the semantic meaning of specific payload bytes, nor can we predict whether subsequent calculations within the ray tracing pipeline depend on them. Simply appending more bytes to the payload is also not feasible, since the payload size is registered via an API call prior to pipeline execution and potentially checked by the application if not tampered with. Thus, we are considering the software emulation.

6.2 Licensing

We plan to contact the relevant rights holders to ask for their permission to release the captured datasets. We are aware of the issues related to licensing, artistic intellectual properties, and other concerns, and we understand the possible reasons for the absence of such data in existing datasets. Activision, the creator of Caldera [2], stated that transforming proprietary assets into a shareable format demands considerable effort. We sincerely appreciate the diligence involved. We also believe that by taking the initiative to extract and convert video game data into a usable format ourselves, instead of relying on companies to allocate their resources for this task, we enhance the likelihood of releasing similarly complex datasets.

Even if we do not secure permissions, our work remains relevant and beneficial. Researchers could use the tool to capture data for their own internal benchmarking. Real-time TLAS visualization in a separate window, and the potential visualization of rays and hits could aid game developers during debugging. No existing frame-capture tool currently supports that. In the long run, it would be beneficial to establish an online hub where researchers could download the datasets and upload their algorithms and benchmark results for others to use, rather than relying on survey papers and re-engineering existing algorithms.

Acknowledgments

This work was supported by the Czech Science Foundation under project GA26-23713S and by the Grant Agency of the Czech Technical University in Prague, project No. SGS25/150/OHK3/3T/13.

References

[1] Mesa 3D. Radv. https://docs.mesa3d.org/drivers/radv.html. Accessed: 2026-03-05.

[2] Activision. Caldera. https://github.com/Activision/caldera, 2024.

[3] Timo Aila, Tero Karras, and Samuli Laine. On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 101–107. ACM, 2013.

[4] AMD. Amd radeon raytracing analyzer. https://gpuopen.com/radeon-raytracing-analyzer/.

[5] Hans-Kristian Arntzen. Vkd3d-proton. https://github.com/HansKristian-Work/vkd3d-proton.

[6] Timothee Besset. doom3.gpl: Doom 3 gpl source release. https://github.com/TTimo/doom3.gpl, 2010.

[7] Blender and community. Demo files. https://www.blender.org/download/demo-files/.

[8] Scientific Computing and The University of Utah Imaging Institute. The utah 3d animation repository. https://www.sci.utah.edu/~wald/animrep/.

[9] Godot Engine. Third person shooter (tps) demo. https://godotengine.org/asset-library/asset/2710.

[10] Unreal Engine. Sample game projects. https://dev.epicgames.com/documentation/en-us/unreal-engine/sample-game-projects-for-unreal-engine.

[11] Vlastimil Havran and Werner Purgathofer. Comparison methodology for ray shooting algorithms. Technical Report TR-186-2-00-20, Institute of Computer Graphics and Algorithms, TU Wien, 2000.

[12] Intel. Intel graphics performance analyzers. https://www.intel.com/content/www/us/en/developer/tools/graphics-performance-analyzers/overview.html.

[13] Arseny Kapoulkine. Measuring acceleration structures. https://zeux.io/2025/03/31/measuring-acceleration-structures/, March 2015.

[14] Stanford Computer Graphics Laboratory. The stanford 3d scanning repository. https://graphics.stanford.edu/data/3Dscanrep/.

[15] Jonas Lext, Ulf Assarsson, and Tomas Moller. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications*, 21:22–31, 2001.

[16] Lufei Liu, Mohammadreza Saed, and Yuan Hsi Chou et al. Lumibench: A benchmark suite for hardware ray tracing. In *2023 IEEE International Symposium on Workload Characterization*, pages 1–14, 2023.

[17] Amazon Lumberyard. Amazon lumberyard bistro, open research content arhive (orca). http://developer.nvidia.com/orca/amazon-lumberyard-bistro, 2017.

[18] LunarG. Gfx reconstruct: Graphics api capture and replay tools for reconstructing graphics application behavior. https://github.com/LunarG/gfxreconstruct.

[19] Evgeny Makarov. Practical tips for optimizing ray tracing. https://developer.nvidia.com/blog/practical-tips-for-optimizing-ray-tracing/, 2023.

[20] Morgan McGuire. Computer graphics archive. https://casual-effects.com/data, July 2017.

[21] Daniel Meister and Jiří Bittner. Performance comparison of bounding volume hierarchies for gpu ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 11(4):1–19, 10 2022.

[22] Daniel Meister, Shinji Ogaki, and Carsten Benthin et al. A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum*, 40(2):683–712, June 2021.

[23] Microsoft. Directx intermediate language. https://github.com/Microsoft/DirectXShaderCompiler/blob/main/docs/DXIL.rst.

[24] Microsoft. Pix on windows. https://devblogs.microsoft.com/pix/.

[25] Microsoft. Programming reference for the win32 api. https://learn.microsoft.com/en-us/windows/win32/api/.

[26] Microsoft. Detours. https://www.microsoft.com/en-us/research/project/detours/, January 2002.

[27] Microsoft. Directx raytracing (dxr) functional spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html, June 2025.

[28] NVIDIA. Nvapi. https://github.com/NVIDIA/nvapi.

[29] NVIDIA. Nvidia nsight graphics. https://developer.nvidia.com/nsight-graphics. Accessed: 2026-03-01.

[30] NVIDIA. Q2rtx: Nvidia's implementation of rtx ray-tracing in quake ii. https://github.com/NVIDIA/Q2RTX, 2025.

[31] PCGamingWiki. List of games that support ray tracing. https://www.pcgamingwiki.com/wiki/List_of_games_that_support_ray_tracing.

[32] Matt Pharr. Example scenes for pbrt-v4. https://github.com/mmp/pbrt-v4-scenes, 2021.

[33] ProtonDB. Protondb. https://www.protondb.com.

[34] RenderDoc. Renderdoc (v1.43). https://renderdoc.org, February 2026.

[35] Steam. Steam hardware & software survey: January 2026. https://store.steampowered.com/hwsurvey, January 2026.

[36] Walt Disney Animation Studios. Moana island scene. https://www.disneyanimation.com/resources/moana-island-scene/, 2018.

[37] Unity Technologies. Demos and sample projects. https://unity.com/demos.

[38] Marek Vinkler, Vlastimil Havran, and Jiří Bittner. Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing. *Computer Graphics Forum*, 35(8):68–79, November 2015.

[39] Ingo Wald and William R. Mark et al. State of the art in ray tracing animated scenes. *Computer Graphics Forum*, 28(6):1691–1722, 2009.

[40] Ingo Wald and Philipp Slusallek. State of the art in interactive ray tracing. In *Eurographics 2001 - STARs*. Eurographics Association, 2001.