Pacific Graphics (2022)
Y. Yang, A. D. Parakkat, B. Deng, and S.T. Noh (Editors) *Short Paper*

Reconstructing Bounding Volume Hierarchies from Memory Traces of Ray Tracers

Max von Buelow[1], Tobias Stensbeck[1], Volker Knauthe[1], Stefan Guthe[1] and Dieter W. Fellner[1,2]

[1]Technical University of Darmstadt, Germany
[2]Fraunhofer IGD, Germany & Graz University of Technology, Institute of Computer Graphics and Knowledge Visualization, Austria

Abstract

The ongoing race to improve computer graphics leads to more complex GPU hardware and ray tracing techniques whose internal functionality is sometimes hidden to the user. Bounding volume hierarchies and their construction are an important performance aspect of such ray tracing implementations. We propose a novel approach that utilizes binary instrumentation to collect memory traces and then uses them to extract the bounding volume hierarchy (BVH) by analyzing access patters. Our reconstruction allows combining memory traces captured from multiple ray tracing views independently, increasing the reconstruction result. It reaches accuracies of 30 % *to* 45 % *when comparing against the ground-truth BVH used for ray tracing a single view on a simple scene with one object. With multiple views it is even possible to reconstruct the whole BVH, while we already achieve* 98 % *with just seven views. Because our approach is largely independent of the data structures used internally, these accurate reconstructions serve as a first step into estimation of unknown construction techniques of ray tracing implementations.*

CCS Concepts

• ***Software and its engineering*** → ***Software reverse engineering;*** • ***Computing methodologies*** → ***Ray tracing;*** • ***Theory of computation*** → *Program analysis;*

1. Introduction

Advanced computer graphics require sophisticated GPU hardware and increasingly complex ray tracing techniques. These techniques include the efficient construction of bounding volume hierarchies (BVH) that are used as a key performance accelerator in ray tracing [AKL13]. Unfortunately, not all of these implementations are freely available, making debugging and further research difficult.

To obtain the memory addresses such systems reference, we need to inject instrumentation code into a target ray tracing application. This can be achieved conveniently via binary instrumentation tools, which enables us to collect these desired memory traces for a specific rendering view. Memory traces are recordings of every memory access performed by a target application. Analyzing them allows us to obtain detailed information, e.g., about the memory subsystem and structures that are used internally. We therefore developed a process to reconstruct the BVH directly from the trace data. Our approach first generates an annotated graph with cost metrics that is later refined into a BVH tree structure. The reconstructed BVH can then be used for further analysis or profiling purposes.

Our approach is largely independent of the data layout used by ray-tracers and enables first steps for semi-automatic reverse engineering of BVHs. The pipeline of our process can be seen in fig. 1.

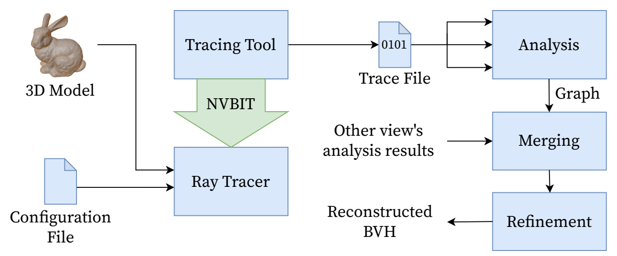


Figure 1: *Overview of our exemplary ray tracer system architecture and our tracing and visualization processes.*

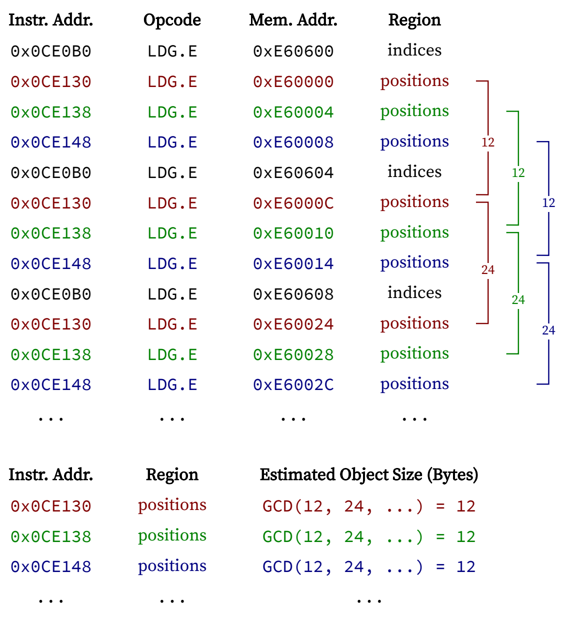
In summary our contributions are:

- A process for reconstructing the BVH from a memory trace.
- The observation that relaxing the constraints on the connections between nodes from a binary tree to a general directed graph enables a simpler and more robust reconstruction.

2. Related Work

Although reverse engineering is an actively researched topic in the security community, the field of computer graphics needs further investigation. While basic characteristics of graphics hardware

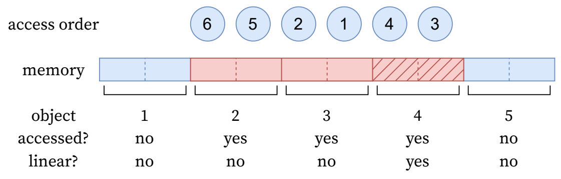
DOI: 10.2312/pg.20221243



Instr. Addr.	Opcode	Mem. Addr.	Region
0x0CE0B0	LDG.E	0xE60600	indices
0x0CE130	LDG.E	0xE60000	positions
0x0CE138	LDG.E	0xE60004	positions
0x0CE148	LDG.E	0xE60008	positions
0x0CE0B0	LDG.E	0xE60604	indices
0x0CE130	LDG.E	0xE6000C	positions
0x0CE138	LDG.E	0xE60010	positions
0x0CE148	LDG.E	0xE60014	positions
0x0CE0B0	LDG.E	0xE60608	indices
0x0CE130	LDG.E	0xE60024	positions
0x0CE138	LDG.E	0xE60028	positions
0x0CE148	LDG.E	0xE6002C	positions
...	...	...	...

Instr. Addr.	Region	Estimated Object Size (Bytes)
0x0CE130	positions	GCD(12, 24, ...) = 12
0x0CE138	positions	GCD(12, 24, ...) = 12
0x0CE148	positions	GCD(12, 24, ...) = 12
...	...	...

Figure 2: *Exemplary trace of SASS memory load instructions generated by executing a kernel that loads a vertex from a indexed triangle list. The relevant address differences are marked in the memory trace as well.*



object	1	2	3	4	5
accessed?	no	yes	yes	yes	no
linear?	no	no	no	yes	no

Figure 3: *Demonstration of the linear access analysis. Each subdivision in the memory row represents a structure consisting of two words. Words are accessed in the order shown in circles. The last two rows describe the results of the linear access analysis. Red boxes represent accessed structures and linearly accessed structures have a hatched background.*

were explored by MEI and CHU [MC17], the security community usually aims to understand systems from a higher level perspective by analyzing malicious software as well as finding potential security issues in proprietary software [TFSS11]. In particular, XU, RAY, SUBRAMANYAN, and MALIK [XRSM17] use memory traces in order to train a neural network for judging if a program is malicious or not. From a software systems perspective, GU, WU, LI, and CHEN [GWLC20] use memory traces for analyzing memory patterns if they are regular or not. CLEARY, GORMAN, VERBEEK, et al. [CGV*13] use memory traces in order to reconstruct memory contents of target applications efficiently at arbitrary program states for debugging and security purposes. Von BUELOW, RIEMANN, GUTHE, and FELLNER [vBRGF22] visualize GPU memory traces of simulated cache profiling values of ray tracers. The recently published NVIDIA binary instrumentation tool (NVBIT) [VSNK19] enables aforementioned analysis of arbitrary GPU binaries on an assembly level during run time. In this work, however, we further focus on the bounding volume hierarchies (BVH) of ray tracers [App68; Whi80]. Reconstruction of BVHs requires knowledge of their initial construction methods [MB90; WBKP08; GS87]. Our overall approach of reconstructing BVH data from multiple views can be seen as coarsely related to multi-view stereo algorithms [SCD*06] that similarly take low-dimensional data from multiple views and reconstruct their common 3D scene.

3. BVH Reconstruction from Memory Traces

We use NVBIT to inject our memory tracing function in order to extract byte addresses of every load and store instruction.

Due to the embarrassingly parallel nature of GPU ray tracing, memory accesses of different warps get interleaved, despite being independent from one another. To make further analysis more straight forward, we initially group memory accesses according to their global warp indices.

As analyses operate on structure-wise indices instead of bare memory addresses, it is important to know the size of the involved structures. However, internal data structures vary between different ray tracing implementations, making dynamic estimation of their size mandatory. We developed two heuristics that in a combination estimate sizes of internal structures reliably. The first heuristic is based on the observation that many instructions reference memory with a constant intra-structure offset. If the same instruction executes multiple times on different indices (see fig. 2), addresses of resulting memory accesses will differ by a multiple of the structure size. Our analyzer uses this observation by collecting enough references from the same instruction address and computing the greatest common divisor in order to derive the final structure size. However, cases where this heuristic fails are loop unrolling and when instructions are only executed once per thread. Therefore, our second heuristic is based on the observation that loading a small structure (e.g. a vertex) results in multiple consecutive memory accesses on a chunk of memory. We analyze the extent of memory accessed by groups of accesses to the same allocation. If the majority of reported groups have the same size, we assume it is the actual structure size.

Given the resulting list of *indices* of BVH nodes, bounding boxes, vertex positions, etc., we proceed with our analyses steps.

3.1. Analyses

The following describes individual analyses that were used. Analyses can be executed independently from one another ensuring a modular application design. Their main purpose is to provide the metrics about potential dependencies in memory that are then used to build the desired graph structure.

Linear Access Analysis The linear access analysis checks for each accessed structure of an allocation if it was accessed in a linear fashion. An example for this analysis are leaf nodes that usually store a list of triangles. Figure 3 shows an example.

Inter-Allocation Link Analysis The allocation link analysis attempts to capture associations between two different buffers. Some buffers simply store indices (implicitly or explicitly) to another buffer, like indexed face lists, making them therefore dependent on each other. We capture links by storing the set of subsequently accessed indices in other buffers.

Consecutive Access Analysis The Consecutive Access Analysis finds dependencies *inside* a single allocation, by recording successively accessed indices. This way, we can capture hierarchical dependencies for BVH trees that are recorded in the same buffer. We use plausibility score counters for further estimation of the probability of a potential link.

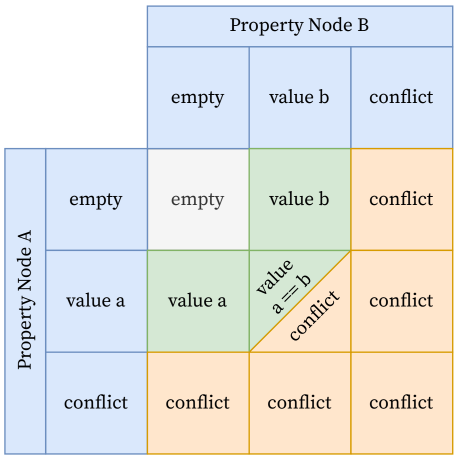
Stack Analysis The Consecutive Access Analysis suffers from false linkage due to irregular traversal of children nodes originating from stack-based traversal. The Stack Analysis analysis is specialized to stack-based BVH traversals and aims to further refine the quality of the reconstructed BVH by keeping track of push and pop operations onto the ray tracing stack. As stack operations are usually the only source for local memory utilization, our reconstruction algorithm therefore simply tracks local memory load and store operations.

3.2. Reconstruction

Our proposed BVH reconstruction pipeline combines the information collected by various analyses into a tree of nodes, which are classified as either inner nodes or leaf nodes. For inner nodes, the collected information includes the memory location of the bounding box as well as potential child nodes. For leaf nodes, an index into the face list as well as the number of triangles is stored.

Construction The construction phase creates an annotated directed graph containing score values from previous analyses steps. This graph will later be refined into a BVH tree. Therefore, our reconstruction pipeline first creates a node for each accessed element in the nodes buffer and identifies them with their index inside the buffer. Then, in order to find out if nodes are leaves, we apply the inter-allocation link analysis between nodes and faces, followed by a linear access analysis onto the faces for finding the list of faces that correspond to a leaf node. Most importantly, our reconstruction now creates edges between nodes forming the desired graph. Therefore, it uses either the consecutive access analysis, or the stack analysis, if desired. For each edge, we store the scores for further cycle removal steps. Finally, for completeness reasons, we also use the inter-allocation link analysis in order to annotate corresponding bounding box structures for each inner node.

Merging When capturing memory traces for several ray tracing views, multiple individual graphs will be produced that need to be merged for further processing. However, as reconstructed annotations for a node in the reconstruction graph may differ, we introduce a permanent *conflict* state as visualized in fig. 4 for each property that will be used if the same node has different properties in both graphs to be merged. Otherwise, if a property of a node does not exist, it is marked as *empty* denoting that it might get filled in the future.



		Property Node B		
		empty	value b	conflict
Property Node A	empty	empty	value b	conflict
	value a	value a	value a = b / conflict	conflict
	conflict	conflict	conflict	conflict

Figure 4: *Result of merging two properties with different states. If both properties have the same value, the result is that value. If they have different values, the result is a conflict.*

This ensures that merging is commutative and associative. For example, if instead of a conflict we would set the state back to empty if the two values did not match, then the following equality and therefore associativity would break (where a and b are values and $\circ$ is the merge operation):

$$(a \circ b) \circ b = a \circ (b \circ b) \tag{1}$$

$$\text{empty} \circ b = a \circ b \tag{2}$$

$$b = \text{empty} \quad \lightning \tag{3}$$

Refinement In the final refinement phase, our proposed pipeline determines the final type of each node and reduces the graph structure into a tree by removing dependencies. As a first step, it defines inner nodes and leaves. A node in the graph becomes a leaf if our algorithm annotated an index to the face buffer. Otherwise, if it has edges to other nodes, it is marked as a parent node. If both cases are not true, the type remains unknown.

The pipeline then determines the root by computing the global access rate onto each BVH node: The root node receives most accesses, as it needs to be traversed for each ray. The same access rate metric is used for determining the traversal order of the directed graph for the analogous reason.

During traversal, our pipeline uses the score value in order to calculate the probability of an edge between nodes to be a plausible edge in the actual BVH by dividing the score of each edge by the number by the sum of all outgoing edges. We then use the two edges with the highest probability in order to create the desired binary relation of our initial BVH. Due to ambiguities while reconstructing node relationships using the consecutive access analysis, unconnected sub-trees may occur. Our reconstruction algorithm can automatically remove such trees that are not connected to the root.

4. Results

For our experiments we used the Bunny, Dragon and Happy Buddha models from the Stanford 3D Scanning Repository as well as the Dabrovic Sponza and Living Room models from the McGuire Computer Graphics Archive. The Dabrovic Sponza and Living Room models provide full environments instead of individual objects.

For evaluating results, we implemented a single hit-point ray tracer on a GPU that uses basic work distribution optimizations from AILA and LAINE [AL09] with a BVH layout similar to the work of WALD, SLUSALLEK, BENTHIN, and WAGNER [WSBW01]. Implementing an own ray tracer enables us the possibility to compare against the ground-truth BVH. While our ray tracer only sends out primary rays, ray tracers with secondary rays will also work in our approach directly as BVH traversal remains independent between primary and secondary rays and our approach handles all *trace* commands separately.

In the following, we compare our reconstructed BVH against the original BVH that was used in our reference ray tracer and state the proportion and location of fully correct reconstructed nodes. A node is marked as correct if all reconstructed data items (section 3.2) match.

4.1. Single-View Reconstruction

Table 1 shows how many correct nodes of the BVH were reconstructed using our approach by various configurations on one single rendering view. We also list the number of accessed nodes based on which parts of the nodes allocation were accessed in the trace due to implicit culling of BVH-based ray tracing. As there is no information about the other nodes in the memory trace, the number of accessed nodes provides an upper bound of what can possibly be reconstructed by our method. Additionally, we exemplarily visualize a reconstructed BVH of the Bunny in fig. 5. A general observation on our approach is that the more general consecutive access analysis comes very close to the results of the specialized stack analysis. The succeeding pruning step makes the measured reconstruction result worse to a small magnitude, which can be explained by our comparison metric not taking the traversal into account and also counting unconnected nodes as correct as long as their successors are equal.

For the Bunny model, we were able to reconstruct 44 % to 49 % of the BVH, which corresponds to approximately half of the model being visible with one representative view. The Dragon and Happy Buddha models have slightly lower reconstruction scores. Here, the chosen resolution of 256 × 256 pixels was arguably too small given the resolution of the models, which we further discuss later in this section. The Living Room model has the worst reconstruction scores despite having less triangles than the Dragon or Happy Buddha models. We attribute this to the geometry being partially outside of the camera frame or occluded by itself, leading to large parts of the BVH never being accessed. The Dabrovic Sponza is a much simpler model and does not have this particular problem. Similarly, in the zoomed-in render of the Bunny model, less of the model is visible in the frame leading to less accessed nodes in the BVH and therefore to a decrease in the reconstruction rate.

Table 1: *Number of fully-correct nodes by type. The first row (with the model name) shows the statistics from the original BVH. The second row shows how many nodes were accessed at all in the trace. Then, we report how many nodes were correctly reconstructed by using either the stack analysis (SA) or consecutive access analysis (CAA) for the reconstruction. Finally, we also show the results of the consecutive access analysis after pruning the tree. Results for the stack analysis do not change after pruning.*

	Parents	Leaves	Total
Bunny	100.0 %	100.0 %	100.0 %
Accessed	57.5 %	48.0 %	52.7 %
SA	49.3 %	48.0 %	48.6 %
CAA	45.0 %	48.0 %	46.5 %
CAA pruned	43.8 %	43.8 %	43.8 %
Dragon	100.0 %	100.0 %	100.0 %
Accessed	46.4 %	35.7 %	41.0 %
SA	36.8 %	35.7 %	36.2 %
CAA	32.1 %	35.7 %	33.9 %
CAA pruned	31.7 %	31.5 %	31.6 %
Happy Buddha	100.0 %	100.0 %	100.0 %
Accessed	44.6 %	29.2 %	36.9 %
SA	30.4 %	29.2 %	29.8 %
CAA	24.6 %	29.2 %	26.9 %
CAA pruned	24.0 %	24.2 %	24.1 %
Dabrovic Sponza	100.0 %	100.0 %	100.0 %
Accessed	60.2 %	46.3 %	53.3 %
SA	47.2 %	46.3 %	46.8 %
CAA	41.9 %	46.3 %	44.1 %
CAA pruned	39.3 %	39.8 %	39.5 %
Living Room	100.0 %	100.0 %	100.0 %
Accessed	19.1 %	10.8 %	15.0 %
SA	11.4 %	10.8 %	11.1 %
CAA	9.6 %	10.8 %	10.2 %
CAA pruned	8.5 %	8.3 %	8.4 %
Bunny (zoom)	100.0 %	100.0 %	100.0 %
Accessed	32.8 %	26.8 %	29.8 %
SA	27.1 %	26.8 %	26.9 %
CAA	26.0 %	26.8 %	26.4 %
CAA pruned	25.8 %	25.7 %	25.7 %

Figure 6 shows our reconstruction result for the Bunny and Dragon models at individual resolutions. The Dragon model has approximate twelve times as many triangles as the Bunny model, therefore it requires bigger images to reconstruct a significant part of the BVH, as otherwise not all visible nodes are accessed due to the fact that the primary rays are spread too far apart. The curves for the different models seem to be shifted by approximately two magnitudes in resolution corresponding to 16 times the number of pixels. After a certain threshold, increasing the resolution does not improve the reconstruction results by much. As discussed above, the reconstruction is limited by which nodes are accessed, and increasing the resolution does not lead to more nodes being accessed if the nodes are culled because of visibility. In general, modest resolutions are sufficient for reconstructing even high-resolution models, however, the sampling frequency of the rays has to be big enough to match the sampling rate of the mesh [Nyq02].

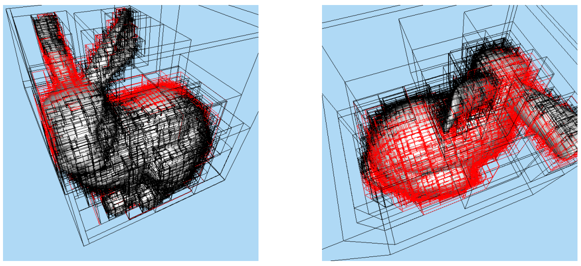


Figure 5: *Visualization (front and back) showing the reconstructed BVH of the Bunny mesh. The bounding box locations are extracted from the memory contents. Fully-correct nodes are shown in black; other nodes are shown in red.*

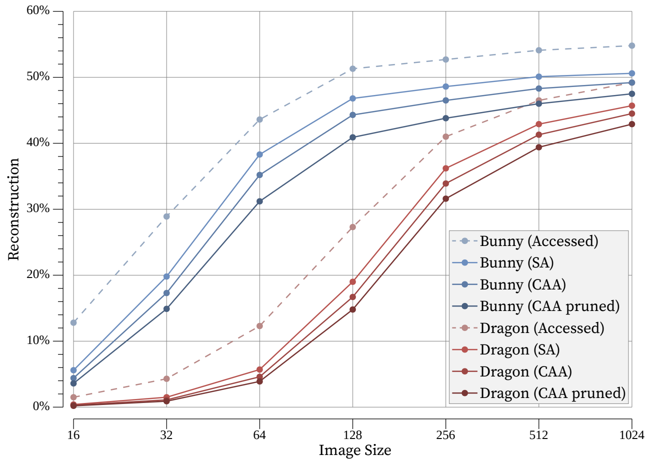


Figure 6: *Percentage of correctly reconstructed nodes depending on image size.*

4.2. Multi-View Reconstruction

As seen in the previous section, the BVH can be reconstructed only partially from a single view. Because we reached reconstruction rates of 100 % with few views easily on the Bunny, we focus here on reconstructing the Dragon model, which turned out to be more difficult due to the model's intricate geometry. Table 2 lists the corresponding reconstruction results. Using the best camera configuration from the Bunny model resulted in the reconstruction of only 85.5 % of the BVH of the Dragon model. Even using a higher resolution of 512×512 pixels and with four additional cameras that were manually placed to capture regions missing in the reconstruction, we were only able increase the reconstruction to approximately 98 %. Figure 7 shows the location of the incorrect or missing nodes on the model, which are mostly found in concave regions in the mouth, the hind leg and the bottom of the model, despite the additional cameras explicitly targeting the mouth and hind leg. For this model, the higher triangle count causes the bounding boxes of leaf nodes to be smaller in relation to the model, exacerbating the culling effect.

Table 2: *Multi-view reconstruction results for the Dragon model with different numbers of cameras.*

		Parents	Leaves	Total
Dragon		100.0 %	100.0 %	100.0 %
2 views	Accessed	80.3 %	67.4 %	73.9 %
256^2 pixels	SA	68.7 %	67.4 %	68.0 %
	CAA	60.4 %	67.4 %	63.9 %
	CAA pruned	59.4 %	59.6 %	59.5 %
3 views	Accessed	86.2 %	76.3 %	81.2 %
256^2 pixels	SA	77.2 %	76.3 %	76.8 %
	CAA	71.8 %	76.3 %	74.1 %
	CAA pruned	71.3 %	71.1 %	71.2 %
5 views	Accessed	91.3 %	84.0 %	87.7 %
256^2 pixels	SA	84.7 %	84.0 %	84.4 %
	CAA	80.4 %	84.0 %	82.2 %
	CAA pruned	80.1 %	80.0 %	80.1 %
5 views	Accessed	95.4 %	89.0 %	92.2 %
256^2 pixels	SA	89.5 %	89.0 %	89.2 %
	CAA	85.9 %	89.0 %	87.4 %
	CAA pruned	85.6 %	85.5 %	85.5 %
5 views	Accessed	96.9 %	94.3 %	95.6 %
512^2 pixels	SA	94.6 %	94.3 %	94.4 %
	CAA	93.1 %	94.3 %	93.7 %
	CAA pruned	92.8 %	92.8 %	92.8 %
9 views	Accessed	98.6 %	97.8 %	98.2 %
512^2 pixels	SA	97.9 %	97.8 %	97.8 %
	CAA	97.4 %	97.8 %	97.6 %
	CAA pruned	97.3 %	97.3 %	97.3 %

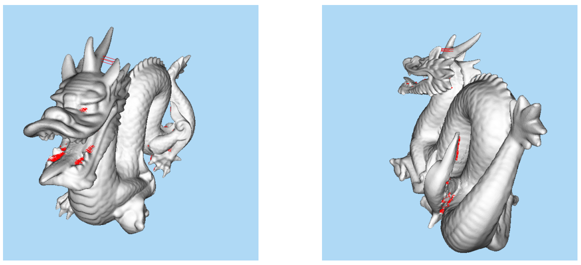


Figure 7: *Incorrect or missing nodes in in the reconstruction of the BVH of the Dragon model from 9 views. Nodes are visualized as their corresponding bounding box.*

5. Conclusion

Bounding volume hierarchies are an important technique in ray tracing acceleration. Reconstruction of bounding volume hierarchies from existing ray tracing binaries can be a promising technique for further analysis of used techniques.

In this paper, we presented an approach that automatically extracts memory references from a ray tracing program during runtime. These memory traces are then used to perform a set of independent analysis steps that encode their results in a graph structure with annotated plausibility scores. Our approach uses this graph in order to reconstruct the original BVH. In order to increase accuracy, memory traces recorded from multiple ray tracing viewing perspectives can automatically be combined during reconstruction.

Our results show that we can reach accuracies of 30 % to 45 % when comparing against the actually rendered bounding volume hierarchy for one view. With multiple views, we reach 98 % and theoretically 100 % are possible when positioning cameras carefully in a way that every triangle can be seen in the combination of all views.

We conclude, that it is possible to reconstruct the structure of a BVH just from the list of memory reference without knowing its actual contents in main memory. It might serve as a first step into estimation of concealed construction techniques of ray tracing implementations.

Limitations and Future Work While this short paper already implements an important part of reverse engineering BVH algorithms, there are still unsolved parts. Our approach relies on memory traces that are extracted from GPU ray tracers. Unfortunately, it is not trivially possible to extract memory traces of hardware-accelerated ray traces which use complex assembly instructions that perform the whole BVH traversal. As these assembly instructions do not have memory reference operands, memory tracing is not possible this way and possibly other techniques have to be involved.

In the future, we would like to use the extracted BVH in order to make estimations about the construction algorithm, specifically the used heuristic for splitting volumes. Additionally, we would like to evaluate our approach on further ray tracing implementations. Last, we think that global optimization methods on our graphs with annotated plausibility scores can lead to improved results.

Source Code The source code for this paper is available at `https://github.com/GreenLightning/MemTracer`.

Acknowledgements

Part of the research in this paper was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 407714161. We thank the anonymous reviewers whose comments helped improve this manuscript.

References

[AKL13] Aila, Timo, Karras, Tero, and Laine, Samuli. “On quality metrics of bounding volume hierarchies”. *Proceedings of the 5th High-Performance Graphics Conference - HPG ’13*. ACM Press, 2013. doi: 10.1145/2492045.2492056 1.

[AL09] Aila, Timo and Laine, Samuli. “Understanding the efficiency of ray traversal on GPUs”. *Proceedings of the 1st ACM conference on High Performance Graphics - HPG ’09*. ACM Press, 2009. doi: 10.1145/1572769.1572792 4.

[App68] Appel, Arthur. “Some techniques for shading machine renderings of solids”. *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. AFIPS ’68 (Spring). ACM Press, 1968. doi: 10.1145/1468075.1468082 2.

[CGV*13] Cleary, Brendan, Gorman, Patrick, Verbeek, Eric, et al. “Reconstructing program memory state from multi-gigabyte instruction traces to support interactive analysis”. *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, Oct. 2013, 42–51. doi: 10.1109/wcre.2013.6671279 2.

[GS87] Goldsmith, Jeffrey and Salmon, John. “Automatic Creation of Object Hierarchies for Ray Tracing”. *IEEE Computer Graphics and Applications* 7.5 (May 1987), 14–20. doi: 10.1109/mcg.1987.276983 2.

[GWLC20] Gu, Yongbin, Wu, Wenxuan, Li, Yunfan, and Chen, Lizhong. “UVMBench: A Comprehensive Benchmark Suite for Researching Unified Virtual Memory in GPUs”. (2020). doi: 10.48550/ARXIV.2007.09822. Pre-published 2.

[MB90] MacDonald, J. David and Booth, Kellogg S. “Heuristics for ray tracing using space subdivision”. *The Visual Computer* 6.3 (May 1990), 153–166. doi: 10.1007/bf01911006 2.

[MC17] Mei, Xinxin and Chu, Xiaowen. “Dissecting GPU Memory Hierarchy Through Microbenchmarking”. *IEEE Transactions on Parallel and Distributed Systems* 28.1 (Jan. 2017), 72–86. doi: 10.1109/tpds.2016.2549523 2.

[Nyq02] Nyquist, H. “Certain topics in telegraph transmission theory”. *Proceedings of the IEEE* 90.2 (2002), 280–305. doi: 10.1109/5.989875 4.

[SCD*06] Seitz, S.M., Curless, B., Diebel, J., et al. “A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms”. *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1 (CVPR’06)*. IEEE, 2006, 519–528. doi: 10.1109/cvpr.2006.19 2.

[TFSS11] Treude, Christoph, Filho, Fernando Figueira, Storey, Margaret-Anne, and Salois, Martin. “An Exploratory Study of Software Reverse Engineering in a Security Context”. *2011 18th Working Conference on Reverse Engineering*. IEEE, Oct. 2011, 184–188. doi: 10.1109/wcre.2011.30 2.

[vBRGF22] Von Buelow, Max, Riemann, Kai, Guthe, Stefan, and Fellner, Dieter W. “Profiling and Visualizing GPU Memory Access and Cache Behavior of Ray Tracers”. *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2022, 7–17. doi: 10.2312/PGV.20221061 2.

[VSNK19] Villa, Oreste, Stephenson, Mark, Nellans, David, and Keckler, Stephen W. “NVBit. A Dynamic Binary Instrumentation Framework for NVIDIA GPUs”. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. ACM, Oct. 2019. doi: 10.1145/3352460.3358307 2.

[WBKP08] Walter, Bruce, Bala, Kavita, Kulkarni, Milind, and Pingali, Keshav. “Fast agglomerative clustering for rendering”. *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, Aug. 2008, 81–86. doi: 10.1109/rt.2008.4634626 2.

[Whi80] Whitted, Turner. “An improved illumination model for shaded display”. *Communications of the ACM* 23.6 (June 1980), 343–349. doi: 10.1145/358876.358882 2.

[WSBW01] Wald, Ingo, Slusallek, Philipp, Benthin, Carsten, and Wagner, Markus. “Interactive Rendering with Coherent Ray Tracing”. *Computer Graphics Forum* 20.3 (Sept. 2001), 153–165. doi: 10.1111/1467-8659.00508 4.

[XRSM17] Xu, Zhixing, Ray, Sayak, Subramanyan, Pramod, and Malik, Sharad. “Malware detection using machine learning based analysis of virtual memory access patterns”. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, Mar. 2017, 169–174. doi: 10.23919/date.2017.7926977 2.