# Conservative Ray Batching using Geometry Proxies

M. Molenaar[1] and E. Eisemann[1]

[1]Delft University of Technology

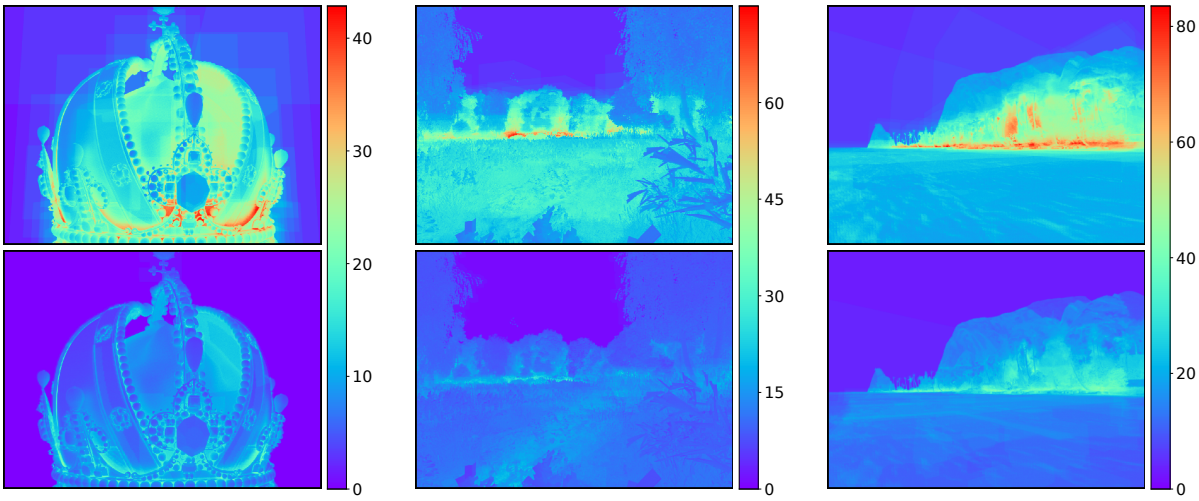


**Figure 1:** *Our method reduces reduces disk bandwidth by minimizing the number of batching points that need to be loaded. The images above show for each test scene (left to right: crown, landscape, island) the average number of batching points visited per sample in unidirectional path tracing. The top row shows regular batched ray traversal, the bottom row shows batched ray traversal with our method.*

**Abstract**

*We present a method for improving batched ray traversal as was presented by Pharr et al. [PKGH97]. We propose to use conservative proxy geometry to more accurately determine whether a ray has a possibility of hitting any geometry that is stored on disk. This prevents unnecessary disk loads and thus reduces the disk bandwidth.*

**CCS Concepts**

• ***Computing methodologies*** → *Ray tracing; Visibility;*

## 1. Introduction

Rendering large scenes is an ongoing challenge in computer graphics, specifically, simulating global illumination for scenes that do not fit in system memory. Monte Carlo light simulation can result in strongly incoherent memory access patterns when traversing an acceleration structure, hereby, reducing performance. These issues are exaggerated in out-of-core traversal since modern solid state drives only provide a fraction of the bandwidth that system memory does.

In this paper, we will discuss and improve upon batched ray traversal, which was first introduced in [PKGH97]. Batched ray traversal is a technique that improves performance of in-core, out-of-core, and distributed rendering by making access patterns more coherent. We propose a possibility to avoid batching rays during traversal, if we can predict that they will not intersect the actual geometry, which reduces the disk bandwidth. We show that this consistently outperforms regular batched ray traversal in terms of memory bandwidth.

## 2. Related work

The research field of large scene visualisation can be broken up into two categories. The first aims to minimise the memory require-

ments by reducing geometric complexity. This comes at the cost of a (slight) degradation in image quality. Alternatively, carefully designed renderers are able to visualise scenes at their full geometric complexity by caching data to disk. The challenge in this second category is to reduce disk traffic and hide the latency associated with disk access.

Walt et al. [WDS05] have shown a distributed ray-tracing system that is able to produce images at interactive rates by replacing data that is not in memory by a low resolution proxy. A similar concept for GPU out-of-core rendering can be used for volumetric data sets [CNLE09,GMIG08,CNSE10]. Offline rendering solutions also explored replacing geometry by proxies until a scene fits in memory [PFHA10]. Similarly, Yoon et al. [YLM06] replace geometry by oriented planes based on a screen-space error function.

Out-of-core rendering often relies on a paging system to move data between disk and system memory [CE97, WDS05]. Alternatively, application controlled data movement may provide more room for domain specific optimizations; Christensen et al. [CLF*03] use application-controlled caching of surface tessellation to aid out-of-core traversal performance, while Wald et al. [WSB01] utilise a two-level acceleration structure hierarchy to manage data movement in a distributed system.

A common limitation in these works far is that incoherent rays are difficult to handle efficiently, due to their incoherent memory access patterns, which are particularly pronounced in Monte-Carlo global-illumination solutions. To combat ray divergence, breadth-first traversal, ray stream traversal and ray reordering schemes have been proposed. Breadth-first traversal of an acceleration structure by a collection of rays ensures that each node is touched at most once [WGBK07, GR08, Tsa09, RGD09], but an "early-out" is impossible, as no front-to-back traversal can be ensured. Ray stream traversal techniques [BAM14, FLPE15] solve this issue at the cost of potentially visiting nodes multiple times. Finally, global ray reordering schemes [MBK*10, ENSB13] sort rays before acceleration structure traversal in an effort to improve the resulting memory access patterns.

Batched ray traversal [PKGH97] can increase memory coherence for improved performance. Here, rays are batched (queued) at the lower elements of a two-level hierarchy, which represents nearby geometry. When enough rays have been batched, batching points are loaded into memory and the associated (bottom-level) acceleration structure is traversed by the batched rays. Batched ray traversal can also be effective for in-core rendering by reducing CPU cache misses [NFLM07] and different acceleration structures have been used for this purpose [Bik12, Gas16].

Our contribution improves ray-handling at the top-level acceleration structure. We propose to store (in system memory) proxy geometry at each leaf node (batching point). Rays that do not intersect the proxy can immediately continue their traversal to the next batching point, without having to load and access the second-level structure.

## 3. Algorithm Overview

Here, we give a more detailed overview of our approach. We first discuss batched ray traversal and our specific implementation. Then we present the memory-efficient geometric proxy, which allows us to quickly and conservatively restrict the amount of rays that require access to the lower-level acceleration structure. In an out-of-core system, this solution reduces disk access and the memory bandwidth.

### 3.1. Batched Ray Traversal

Batched ray traversal is designed to improve memory coherency by grouping rays passing through the same region of space. The space is divided by creating large clusters of primitives. Each of these clusters forms a batching point for rays to be queued, i.e., rays passing through this region will be accumulated. Each batching point has an acceleration structure such that rays can be quickly intersected against the geometry. In out-of-core rendering systems, such as ours, geometry is stored on disk and is loaded when needed.

The rendering process is started by spawning an initial set of camera rays and inserting them into the first batching point they intersect. The scheduler is then responsible for continuously selecting batching points to load and traverse. We use a simple scheduler which always selects the batching point with the most rays. This ensures that the disk bandwidth is amortised over many rays. When a light path ends, a new camera ray is automatically spawned such that the number of rays in the system remains constant. If a ray needs to continue its traversal of the scene after visiting a batching point (e.g., it missed the geometry inside) then it is batched at the next batching point along its path.

The top-level acceleration structure needs to be able to efficiently restart traversal, while storing as little state per ray as possible. We have chosen for a 4-wide BVH following Gasparian et al. [Gas16]. Geometry intersections at batching points are implemented using the Embree ray tracing kernels. The acceleration structures are (re)build when needed and are stored in a LRU (least recently used) cache.

### 3.2. Proxy Geometry

The major novelty of our solution is to store for each batching point some form of proxy geometry that is both memory efficient and conservative. When traversal of the top-level structure reaches a batching point, the ray is first intersected against this approximation; A ray is only batched if it intersects, otherwise traversal of the top-level acceleration structure continues.

We have chosen to approximate the geometry as a binary volume, as very efficient storage solutions exist that have been used in several real-time applications [KSA13, SKOA14, DKB*16, KRB*16]. Such a volume handles even open geometry naturally but we need to ensure the solution is conservative. The construction of the proxy for each batch point works as follows. In a preprocess, the related geometry is conservatively voxelized into a tightly fitting voxel grid. This binary grid is converted to a Sparse Voxel Octree (SVO). To reduce the memory usage, these SVOs are converted to Sparse Voxel Directed Acyclic Graphs (SVDAG) [KSA13] with the difference that not only duplicate subtrees within an SVO are removed but also duplicate subtrees across all of the SVOs. Each duplication is replaced by a single instance. In the resulting structures nodes may have multiple parents; hence they are not trees but

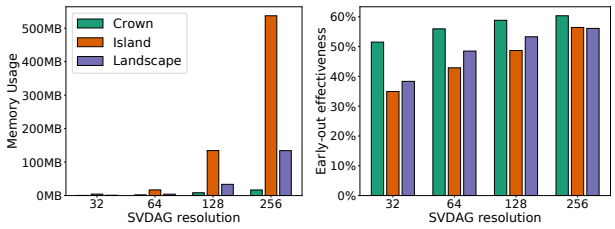


**Figure 2:** *Total SVDAG memory usage (left) and the effectiveness of our system (right) at different voxel grid resolutions.*

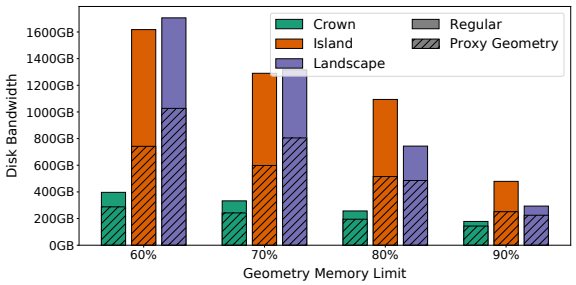


**Figure 3:** *The total disk bandwidth at different geometry memory limits with and without our technique. When proxy geometry intersection testing was enabled, the memory limit was adjusted to compensate for the memory used by the SVDAGs.*

directed acyclic graphs. Further, SVOs from one batch point can virtually share children with another batch point. These SVDAGs may be traversed by any existing SVO traversal algorithm; We use the traversal algorithm presented by Laine et al. [LK11].

## 4. Results & discussion

To test this concept, we have implemented a basic out-of-core path tracer using batched ray traversal as described above. The scenes that were used for testing are the crown model, the landscape scene and the Moana Island scene. The crown model was artificially subdivided 5 times to increase its primitive count. For the Island scene we only render the triangle/quad control meshes, ignoring other geometric primitives. All scenes were rendered using a homogeneous Lambert material since our focus lies on traversal performance only.

**Table 1:** *Triangle counts of the tested scenes.*

| | Crown | Landscape | Island |
|---|---|---|---|
| Unique | 860,272,002 | 25,947,395 | 142,771,030 |
| Instanced | 860,272,002 | 4,330,336,849 | 31,443,289,446 |
| Per batching point | 10,000,000 | 20,000,000 | 25,000,000 |

Selecting the appropriate batching point size is important and non-trivial. More (smaller) batching points increases the memory overhead of the batching system as well as requiring more memory to store the SVDAGs. The minimum number of triangles per batching point was empirically chosen for each scene, see Table 1. This results in 132, 335 and 1984 batching points for the crown, landscape and Island scenes respectively. Our code is available at https://github.com/mathijs727/pandora.

### 4.1. Voxel grid resolution

The resolution of the voxel grids from which the SVDAGs are generated impacts the memory usage, computational overhead, and effectiveness, see Figure 2. As expected, memory usage of the SVDAGs scales cubic with respect to the voxel grid resolution. The large discrepancies in memory usage between the scenes can be attributed to the different number of batching points (and thus SVDAGs) per scene.

Increasing the resolution quickly has a diminishing effect on the number of times that we do not need to descend to the second-level hierarchy. Scenes with larger batching points such as the Island scene seem to benefit more from high resolution voxel grids. The additional geometric details caused by higher primitive counts require a higher SVDAG resolution to accurately approximate.

### 4.2. Proxy Geometry

To test the effectiveness of our system as a whole, we rendered all scenes with 128 samples per pixel at different memory budgets with a voxel grid resolution of $128^3$. Figure 3 shows the reduction in total disk bandwidth when our solution is enabled. It significantly reduces the total disk bandwidth required to render the island and landscape scenes. The impact on disk bandwidth for the crown scene is modest. We suspect that this is caused by it's simplistic shape which only has a high primitive count because of the artificial subdivision.

### 4.3. Discussion

Our addition of a conservative test on the upper hierarchy level for batched ray traversal works well for all the scenes and memory limits that were tested. Every time a ray is batched it delays its traversal, requiring a disk read to continue. By reducing the amount of times that rays are batched we improve the flow of rays through the system. An issue with batched ray traversal is that rays can get stuck at infrequently visited batching points. Only when nearing the end of a render will these batching points be loaded and traversed. Path tracing may lead to many bounces causing long delays when finishing a render. By adding our mechanism we can prevent that straggler rays cause batching points to be loaded every time. In our limited tests, discarding these straggler rays did not reduce the effectiveness of our method.

Our solution works well for batched ray traversal but could also be applied to other traversal schemes. For out-of-core traversal where the discrepancy between processing power and disk bandwidth is high, the overhead of SVDAG traversal is negligible. We are unsure whether this approach could also work for in-core traversal and we leave this up to future work.

The hit points found by intersecting against the proxy geometry could also be used to guide the bottom-level acceleration structure traversal. For example, rays could be sorted based on their expected hit points as was suggested by Moon et al. [MBK*10].

Additionally, our method could work well for shadow rays. Here,

we would voxelize geometry conservatively in two manners; one as described above, the other as an inner voxelization that conservatively predicts a valid intersection. Testing against this inner voxelization would reveal whether a ray can conservatively be stopped. If not, we test it against the other voxelization and no intersection implies that the ray can traverse further, while an intersection implies that we need to batch the ray.

## 5. Conclusion

The novel idea presented in this paper is to batch rays only when an intersection with a (conservative) proxy geometry occurred. Hereby, we reduce disk bandwidth. To illustrate this principle, we have built an out-of-core renderer that implements batched ray traversal and relies on a voxel-based representation that is efficient in terms of memory.

Our results confirm that this technique can indeed improve performance of batched ray traversal. We also believe the same concept can be applied more broadly for other traversal schemes but we leave this for future work.

## 6. Acknowledgements

This work is part of the research program "LED it be 50number P13-20), which is funded by the Netherlands Organisation for Scientific Research (NWO) and supported by LTO Glaskracht, Philips, Nunhems, WUR Greenhouse Horticulture. We would like to thank Walt Disney Animation Studios for contribution the Moana Island scene to the research community.

## References

[BAM14] BARRINGER R., AKENINE-MÖLLER T.: Dynamic ray stream traversal. *ACM Trans. Graph. 33*, 4 (July 2014), 151:1–151:9. doi:10.1145/2601097.2601222. 2

[Bik12] BIKKER J.: Improving data locality for efficient in-core path tracing. In *Computer Graphics Forum* (2012), vol. 31, Wiley Online Library, pp. 1936–1947. 2

[CE97] COX M., ELLSWORTH D.: Application-controlled demand paging for out-of-core visualization. In *Visualization'97., Proceedings* (1997), IEEE, pp. 235–244. 2

[CLF*03] CHRISTENSEN P. H., LAUR D. M., FONG J., WOOTEN W. L., BATALI D.: Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. *Computer Graphics Forum 22*, 3 (2003), 543–552. doi:10.1111/1467-8659.t01-1-00702. 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, ACM, pp. 15–22. doi:10.1145/1507149.1507152. 2

[CNSE10] CRASSIN C., NEYRET F., SAINZ M., EISEMANN E.: *Gpu pro*. AK Peters, 2010, inbook X.3 Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels, pp. 643–676. 2

[DKB*16] DADO B., KOL T. R., BAUSZAT P., THIERY J.-M., EISEMANN E.: Geometry and attribute compression for voxel scenes. In *Computer Graphics Forum* (2016), vol. 35, Wiley Online Library, pp. 397–407. 2

[ENSB13] EISENACHER C., NICHOLS G., SELLE A., BURLEY B.: Sorted deferred shading for production path tracing. In *Computer Graphics Forum* (2013), vol. 32, Wiley Online Library, pp. 125–132. 2

[FLPE15] FUETTERLING V., LOJEWSKI C., PFREUNDT F.-J., EBERT A.: Efficient ray tracing kernels for modern cpu architectures. *Journal of Computer Graphics Techniques (JCGT) 4*, 4 (2015). 2

[Gas16] GASPARIAN T.: *Fast Divergent Ray Traversal by Batching Rays in a BVH*. Master's thesis, Utrecht University, 2016. 2

[GMIG08] GOBBETTI E., MARTON F., IGLESIAS GUITIÁN J. A.: A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7 (Jul 2008), 797–806. doi:10.1007/s00371-008-0261-9. 2

[GR08] GRIBBLE C. P., RAMANI K.: Coherent ray tracing via stream filtering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (2008), IEEE, pp. 59–66. 2

[KRB*16] KÄMPE V., RASMUSON S., BILLETER M., SINTORN E., ASSARSSON U.: Exploiting coherence in time-varying voxel data. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2016), ACM, pp. 15–21. 2

[KSA13] KÄMPE V., SINTORN E., ASSARSSON U.: High resolution sparse voxel dags. *ACM Trans. Graph. 32*, 4 (July 2013), 101:1–101:13. doi:10.1145/2461912.2462024. 2

[LK11] LAINE S., KARRAS T.: Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (2011), 1048–1059. 3

[MBK*10] MOON B., BYUN Y., KIM T.-J., CLAUDIO P., KIM H.-S., BAN Y.-J., NAM S. W., YOON S.-E.: Cache-oblivious ray reordering. *ACM Trans. Graph. 29*, 3 (July 2010), 28:1–28:10. doi:10.1145/1805964.1805972. 2, 3

[NFLM07] NAVRATIL P. A., FUSSELL D. S., LIN C., MARK W. R.: Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on* (2007), IEEE, pp. 95–104. 2

[PFHA10] PANTALEONI J., FASCIONE L., HILL M., AILA T.: Pantaray: Fast ray-traced occlusion caching of massive scenes. *ACM Trans. Graph. 29*, 4 (July 2010), 37:1–37:10. doi:10.1145/1778765.1778774. 2

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 101–108. doi:10.1145/258734.258791. 1, 2

[RGD09] RAMANI K., GRIBBLE C. P., DAVIS A.: Streamray: a stream filtering architecture for coherent ray tracing. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 325–336. 2

[SKOA14] SINTORN E., KÄMPE V., OLSSON O., ASSARSSON U.: Compact precomputed voxelized shadows. *ACM Trans. Graph. 33*, 4 (July 2014), 150:1–150:8. doi:10.1145/2601097.2601221. 2

[Tsa09] TSAKOK J. A.: Faster incoherent rays: Multi-bvh ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 151–158. doi:10.1145/1572769.1572793. 2

[WDS05] WALD I., DIETRICH A., SLUSALLEK P.: An interactive out-of-core rendering framework for visualizing massively complex models. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM. doi:10.1145/1198555.1198756. 2

[WGBK07] WALD I., GRIBBLE C. P., BOULOS S., KENSLER A.: Simd ray stream tracing-simd ray traversal with generalized ray packets and on-the-fly re-ordering. *Informe Técnico, SCI Institute* (2007). 2

[WSB01] WALD I., SLUSALLEK P., BENTHIN C.: Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001* (Vienna, 2001), Gortler S. J., Myszkowski K., (Eds.), Springer Vienna, pp. 277–288. 2

[YLM06] YOON S.-E., LAUTERBACH C., MANOCHA D.: R-lods: fast lod-based ray tracing of massive models. *The Visual Computer 22*, 9-11 (2006), 772–784. 2