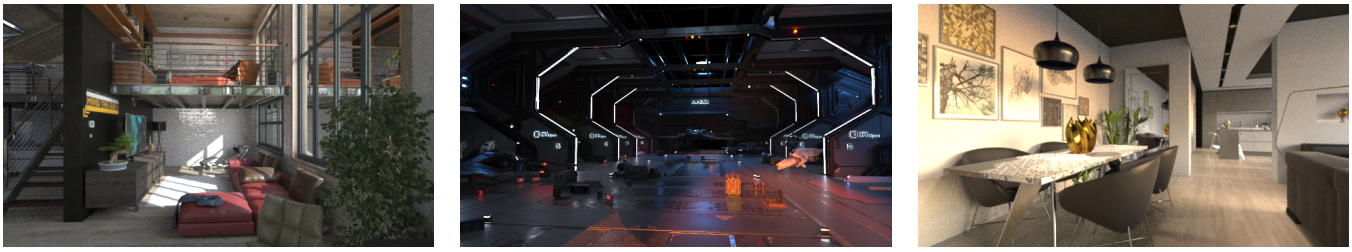
Geometry and Texture Streaming Architecture in Radeon™ ProRender

Atsushi Yoshimura
Advanced Micro Devices, Inc.
Japan

Sho Ikeda
Advanced Micro Devices, Inc.
Japan

Takahiro Harada
Advanced Micro Devices, Inc.
USA



(a) Loft **(b)** Hangar **(c)** Living room

(d) Classroom **(e)** Oasis **(f)** Reading room

Figure 1: Test scenes used to evaluate the texture streaming efficiency. These scenes are rendered at 1280×720 resolution by loading only 0.9% to 6.3% of the entire texture data thanks to the texture streaming.

ABSTRACT

Although light transport simulations generate realistic images, the realism of the synthesized images is limited by the input assets, such as geometries, which are specific attributes of objects, and textures. However, the memory size for the assets may not be large enough for rendering because the assets can be enormous as their resolution increases. Most significantly, the memory extensibility of a device on which compute units and memory are integrated, such as a GPU, could be low. Thus, rendering fine-detail assets with limited memory is a critical challenge for rendering software to achieve photo-realistic image synthesis. Radeon™ ProRender chooses on-demand data streaming for geometries and textures where the data transfer to the device is deferred until the rendering process requires the data. We call this streaming architecture. We choose this architecture because it has two benefits. First, it can reduce the total memory allocation as the rendering process may not require some parts of the assets. Second, it bounds the maximum memory allocation throughout the rendering by adaptively discarding the transferred assets.

This technical report explains the architecture and implementation of the geometry and texture streaming in Radeon™ ProRender.

email: { Atsushi.Yoshimura, Sho.Ikeda, Takahiro.Harada }@amd.com
Advanced Micro Devices, Inc. Technical Report No. 22-10-daf5, October 13, 2022.

KEYWORDS

ray tracing, global illumination, out-of-core rendering, texture streaming

1 INTRODUCTION

Detailed scene data, such as geometries and materials for light transport simulation, is essential for rendering a photo-realistic image. The geometries and materials have to be detailed enough to achieve the expected quality of the rendering. Content creation software has achieved a maximum computationally reachable level of detail in asset editing nowadays [Adobe 2022], [MAXON 2022]. However, such complexity often is presented by an enormous amount of data. The procedural process to generate assets can avoid data-size explosion. However, it is often baked as a classic discrete representation for the rendering process, such as polygons and pixels, because of performance and data portability requirements.

Dealing with huge amounts of data has long been a challenging task in rendering of complex images. To deal with huge assets, the Reyes image rendering architecture applied a sophisticated rasterization algorithm using data locality on the limited memory [Cook et al. 1987]. Shortly after that, ray-traced global illumination took the place of rasterization for film production because of the evolution of processors and memory. However, its incoherent access to memory causes a significant issue because the memory capacity is finite. A classical method but an effective way for dealing with

incoherent access on limited memory is to defer data transfer to the memory until the data is needed [Peachey 1990].

However, the data transfer to memory on the GPU is an expensive operation that intercepts the tasks. Thus, the memory management has to be specialized to the GPU. The images in Fig. 1 were rendered on a GPU without storing the entire scene data on the video memory. Such domain-specific memory management not only maximizes software performance but increases portability.

Radeon™ ProRender, a uni-directional path tracer on the GPU, chooses the deferred data transfer approach for geometry and texture memory management to maximize scalability against the details of the input assets [Advanced Micro Devices, Inc. 2020a]. The data required to render the scene is streamed to the video memory, which is kept as cache. The cached data could be evicted depending on the eviction policy of the cache. Our streaming architecture reduces the total allocation amount and controls the maximum allocation size of the cache adaptively to the system. As rendering on the GPU often requires allocating a large amount of temporal data for parallelization [Laine et al. 2013], the allocation controllability of the streaming approach also benefits optimizing the entire performance.

We first explain the design and implementation of the streaming architecture of the renderer. After describing the method, we discuss the statistics we captured from several test scenes. At last, we show that the streaming architecture makes it possible to render scenes by loading just a fraction of the entire texture data.

2 RELATED WORKS

Rendering a scene that does not fit into a system memory has always been a challenge in ray tracing and rasterization. Peachey discussed the inefficiency of loading all the texture data from the disk and proposed a tile-based texture caching system using a MIP map for the Reyes rendering pipeline [Cook et al. 1987; Peachey 1990]. Since then, there have been many studies on the topic. Pharr *et al.* proposed a path tracing system for loading texture and geometry on-demand from disk, where scene geometries are managed by a hierarchical grid acceleration structure, which is also used to reduce the cache miss by reordering the ray execution [Pharr et al. 1997]. Our work goes in the same direction but designs an on-demand loading and streaming architecture for texture and geometry implemented in a GPU renderer — Radeon™ ProRender.

3 STREAMING APPROACH

The streaming approach trades the scalability of the detailed input assets for performance because the memory management for streaming architecture intercepts the algorithm for the data transfer. Although the algorithm achieves the best performance with all assets stored on the GPU in most cases, streaming architecture may compensate for the overhead by minimizing transfer due to the memory access locality. Specifically, streaming architecture is an appropriate choice for Interactive Preview Rendering (IPR) in production because of a shorter time to get the first pixel by minimizing data transfers and preprocessing. Therefore, streaming architecture is beneficial depending on the use cases even in an environment with sufficient memory. Thus, streaming architecture on Radeon™ ProRender is designed to minimize the overhead and maximize these benefits in the workflows for two data types in domain-specific ways. The first is for geometries and Bounding Volume Hierarchies (BVHs), and the second is for textures. Memory access from the kernel on the GPU often has limitations, such as memory size and inaccessibility to persistent storage. Thus, the transfer is separately executed on the host code from the task that needs the data streaming. In the following, we will describe the details of our approaches.

3.1 Geometry Streaming

As geometries, including BVHs, are needed for ray casting and shading, our streaming approach manages geometries and their BVHs in Radeon™ ProRender. Multi-level BVH is widely used to deal with instancing and dynamic updates efficiently with a sacrifice of the performance compared to using a single BVH for all geometries. We use two-level BVH as we need flexibility. We choose bottom-level BVHs and their geometries as the granularity of streaming architecture because of its independence. The bottom-level BVHs are transferred to the memory with their geometries, such as vertices, indices, texture coordinates, etc., when the ray-cast traversal needs it. The top-level BVH is always stored in the memory because all rays visit the BVH. We also do not stream light-source geometries because it is frequently accessed for direct lighting calculation, which is known as next-event estimation for path tracing. It consumes some memory depending on the memory size taken by the light geometries, and we empirically assume that the light geometries take less memory than the others. In practice, we transfer all of the geometries and their BVHs as a preprocess when the geometries on a scene are small enough for the video memory to achieve the best performance; otherwise, the streaming architecture is used.

3.1.1 Preprocess. We build BVHs for all the scene geometries and then for its top-level BVH before we the rendering starts, to alleviate stalls due to interruptions for BVH constructions during ray traversal. Subdivision and displacement are also applied as preprocessing for the same reason. We keep the BVHs built on the host memory for the streaming. The BVHs are also kept on persistent storage to avoid redundant BVH constructions on subsequent use. The index buffer for a mesh is also preprocessed and compressed. These data are also kept on the host memory during the rendering. The memory size for geometry streaming is allocated proportionally to the device's memory size. This allocation also can be limited by users for co-operating applications on the device.

3.1.2 Streaming. A ray traversal is done by an iterative algorithm. An iteration mainly consists of traversal and transfer of geometries and their BVHs. Intersections against unavailable bottom-level BVHs on the memory are skipped. Instead, Requests for those geometries are placed on a request buffer. The requested geometries are transferred for the next iterations accordingly, and the ray continues the traversal with the BVHs. However, note that a ray traversal does not exit on a hit of an object which is not available on the video memory. It continues and intersects against geometries that their BVHs are cached. The ray traverses the scene until the end, and the hit distance is updated if a hit is found. Therefore, bottom-level BVHs we already processed in the previous iterations do not need to be intersected in the subsequent iterations. To keep

track of the state of an object, a bit flag is set for each geometry to mark whether we have already traversed the geometry or not. Thus, we never transfer geometries that their BVHs more than once in a ray cast. Fig. 2 is an illustration of the iterations. Each iteration starts from top-level BVH traversal keeping the closest hit instead of saving the ray state and restoring on the next kernel execution, as done in [Harada 2016], because of the complexity and global memory traffic coming with the approach.

A ray could intersect many geometries which are not resident yet. In that case, it could request to transfer many geometries, which results in unnecessary transfers as the ray may find an intersection point at the very first geometry. Therefore, we restrict the number of requests a ray can put in a single iteration to one. We could take another strategy: let a single ray request more than one request in a single intersection kernel execution. This would increase the number of geometries we transfer but can reduce the number of iterations we do. We set the number of requests to one to prevent cache overflow, which would be computationally expensive.

The memory for geometry streaming is sequentially filled by the geometries and their BVHs. The filled memory is flushed when it runs out of memory. The memory size has to be larger than or equal to the size of the largest geometry, as we use bottom-level BVHs and their geometries as the granularity of the transfer, because the finer granularity is computationally expensive and it is difficult to pay off its complexity. The hardware ray intersection implemented on RDNA2 architecture [Advanced Micro Devices, Inc. 2020b] is not used for our geometry streaming because of the memory footprint. Instead, we use compressed BVH node data, as done in [Ylitie et al. 2017], to reduce BVH's memory traffic. In this work, we set a single geometry as the data transfer unit, but we could use other chunking strategies, e.g., split a geometry into uniform 256 MB blocks. The latter would simplify the cache management and let us use a more sophisticated cache eviction logic. However, it would add some complexity to the traversal code. At present, we only set a flag value for the offset for the bottom-level BVH if the BVH is not cached; thus, no additional logic is added, which keeps our traversal code easier. Exploring different chunking strategies is future work.

Usually, shading requires many attributes, such as shading normal, texture coordinates, and user-defined attributes of the geometry at each shading point. These attributes are computed based on their hit points after ray-traversal iterations because the points are not determined during the iterations. We transfer all the data associated with geometries during the iterations to reduce the number of kernel interruptions due to the data transfers for shading. Accordingly, these attributes are available without data transfer after the iterations as long as the hit geometries are not evicted. However, we use an iterative structure similar to the ray traversal in case the hit geometries are evicted, which does not happen often.

3.2 Texture Streaming

We use a small fixed-size chunk as the granularity of the texture streaming because of two reasons. First, the texture buffer is trivially split into small chunks in contrast to the BVHs of the geometres. Second, texture access is not as random as the memory access of BVH on traversal. We choose the set associative cache to control cache evictions. Each entry on the cache has a time stamp to prioritize the lately accessed entry because of the locality of access. A cache entry is evicted based on the prioritization when the set is full. In this work, we used 8 sets and 512B cache line size. The memory for texture streaming is statically allocated proportionally to the device memory size same as geometry streaming.

3.2.1 Preprocess. The compression, MIP map texture generation, and color space transformation of textures are done as preprocess. We use block compression to support random access to the texture. The block size is 16 pixels - 4 by 4. We compress textures using pixel similarity on the tile. The pixels are described as representative values and differences from the values. Although it is lossy and fixed rate compression, the compression rate is 1:2. We do not do aggressive compression to keep the compression artifact small. This is a variation of delta color compression [Brennan 2016]. To avoid repeated texture processing, the pre-processed textures are saved on the persistent storage which is read on the subsequent use.

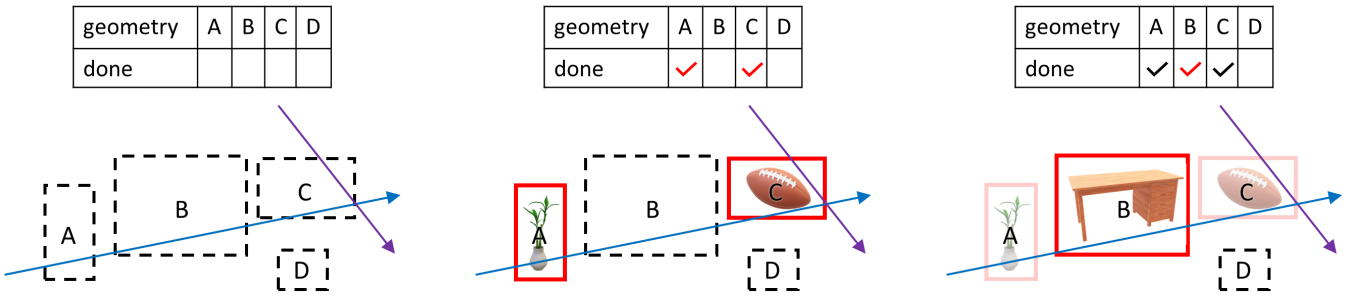
Uploading all the textures to video memory before rendering starts is better in terms of rendering performance when the total texture size is smaller than because it reduces the number of intercepts for streaming. When all the texture data is transferred to video memory, the overhead of our approach is just a single indirection using the value stored in the translation lookaside buffer.

3.2.2 Streaming. Our shader execution logic is implemented as a stack machine as described in [Fujieda and Harada 2022]. The required texels are determined during the evaluation process of the shader. MIP map let us use the texels filtered with the right footprint which avoids texture aliasing and improve the memory access locality. The shader evaluation and texture transfer can be alternatively repeated until all required textures for the shading process is ready similar to geometry streaming. The evaluation issues requests for missing texture chunks and the data is transferred based on the requests. We do not stream the coarsest level of the MIP map but it is always kept on the memory in order to use it as an approximated value when the required chunk is unavailable on the cache. Thus, the number of iterations can be used as a tweakable parameter depending on the accuracy requirement. We use smaller iterations for the interactive sessions to minimize the streaming overhead and achieve better interactivity. Although smaller iterations may cause an over-blurred texture mapping, the bias gradually decreases during the interactive sessions because of the subsequent samples after a fallback reads the texel at the right resolution.

4 RESULTS

The methods described in this paper are implemented in Radeon™ ProRender using OpenCL™ and C++. All the evaluations are performed with an AMD Ryzen™ 9 5950X CPU and a Radeon™ RX 6900XT GPU with 16 GB video memory.

We first evaluated the amount of data transfer and its performance with the geometry streaming using a scene we call W (we cannot describe the details as this is the customer's data). The scene is an outdoor scene where there are buildings and trees. The scene is rendered with and without the geometry streaming. Fig. 4 shows the GPU memory consumption during a frame rendering.



(a) Initial status **(b) First iteration** **(c) Second Iteration**

Figure 2: Iterations of geometry streaming. (a) All geometries A, B, C, and D are not loaded at the very first iteration. (b) A and C are loaded and traversed as the rays hit their bounding boxes first. Then, A and C are marked as done. (c) B is loaded and traversed if the blue ray goes through A. A and C are skipped as these geometries are already marked as done. D is not processed but iterations are finished because there is no more hit by any rays.

(a) Restaurant

(b) Plants

Figure 3: Test scenes used to evaluate the texture streaming efficiency. These scenes are rendered at 1280×720 resolution with only 0.9 to 6.3% of the entire texture data thanks to the texture streaming architecture.

	Texture size	Texture count	Used cache size
Loft	2257 MB	304	43.17 MB
Hanger	1440 MB	143	9.86 MB
Living room	99 MB	213	10.93 MB
Classroom	206 MB	69	13.43 MB
Oasis	621 MB	874	6.53 MB
Reading room	2139 MB	145	20.00 MB
Restaurant	730 MB	116	25.12 MB
Plants	1948 MB	51	38.89 MB

Table 1: Statistics on the test scenes. The amount of texture size we needed to allocate if they are uploaded to the GPU when the streaming approach is not taken. Number of textures for each scene, and the actual texture data paged in to render these scenes.

The rendering starts with a small amount of GPU memory usage for the geometries (316 MB) at 0% as the geometries are almost not transferred until needed. The amount increases and saturates around 2.7 GB during the rendering. If we turn off the geometry streaming and upload all the data on the GPU, it consumes 3.4 GB. Thus we can see that we could render the scene with 79% of the memory compared to the case where geometry streaming is not used. However, it comes with a cost. It adds complexity to the rendering pipeline and starts many sessions of GPU synchronization and GPU memory writing and reading. This rendering finished in 34 s without streaming but took 55 s with streaming. Thus, the runtime overhead is not negligible.

Next, we tested texture streaming. Test scenes shown in Fig. 1 and Fig. 3 were rendered with 1 GB texture cache on the video memory. The entire texture data size of these scenes ranges from 206 MB to 2.3 GB, as shown in Table. 1. The texture data size paged into the cache is also shown in the table. Even for the Loft scene, which has 2.3 GB texture data, it did render with only 43.17 MB cache usage, which is 1.9% of the entire data. This is because it computes the appropriate MIP level and only requested data is transferred to video memory. From these experiments, we can see that texture streaming effectively reduces memory usage for the texture.

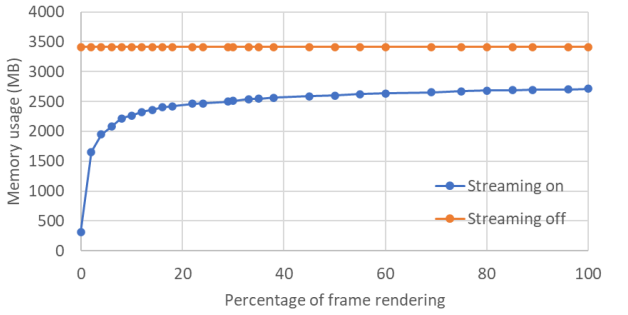


Figure 4: Memory consumption during the rendering with and without streaming. Rendering times were 55 s and 34 s respectively, 1.59x longer than with streaming on Radeon™ RX 6900XT.

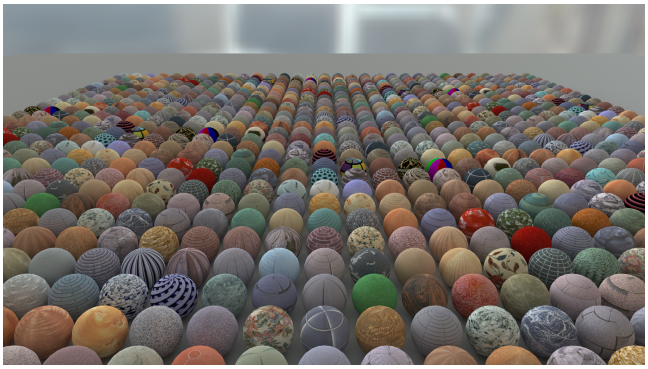


Figure 5: Stress test with a synthetic scene that contains 962 unique 2048x2048 textures.

We also evaluated the relationship between the rendering resolution and the cache usage by rendering the same scene in different resolutions. As the resolution of the rendering increases, the memory traffic gets larger as a finer MIP level is used. We used a synthetic scene with 3592 MB for textures by 962 unique 2048x2048 textures from Pixar One Twenty Eight texture pack, as shown in Fig. 5. The texture cache usage with three resolutions is shown in Fig. 6. Although the memory traffic adaptively increases depending on the resolution, almost all the transfer is avoided thanks to the streaming architecture, even on the 2160 p resolution. This result emphasizes the benefits of our streaming architecture.

5 CONCLUSION

We introduced our streaming architecture for geometry and textures in Radeon™ ProRender.

The streaming architecture optimized in a domain-specific way effectively reduces memory traffic. Despite the overhead, its scalability for the memory size constraint benefits production. However, a non-negligible overhead is observed with geometry streaming on our measurement. We use the two types of workflow for geometry — with and without streaming architecture — because of the performance requirements. A more significant reduction of the

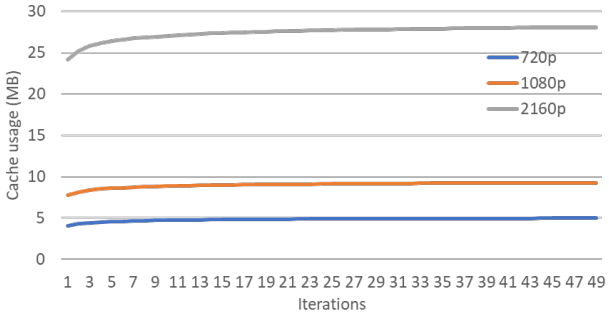


Figure 6: Texture cache usage to render the scene shown in Fig. 5 with rendering resolution of 1280x720 (720 p), 1920x1080 (1080 p), and 3840x2160 (2160 p). The cache usage is saturated at around 20 samples in this example.

overhead on streaming architecture is still ongoing research, and we possibly need more fine-grained communication between the host and GPU. Also, preprocessing, such as MIP-map building and BVH construction, could be deferred, similarly to the data transfer. We would like to further investigate lazy and adaptive streaming architecture.

ACKNOWLEDGMENTS

We would like to thank the team members who have been involved in the development of Radeon™ ProRender and its integration into many digital content creation tools. We thank Oleksandr Kupriyanchuk for his valuable feedback on this paper.

REFERENCES

Adobe. 2022. Substance 3D. https://www.adobe.com/products/substance3d/3d-augmented-reality.html

Advanced Micro Devices, Inc. 2020a. Radeon™ ProRender 2.0. https://gpuopen.com/radeon-pro-render/

Advanced Micro Devices, Inc. 2020b. "RDNA 2" Instruction Set Architecture Reference Guide. https://developer.amd.com/wp-content/resources/RDNA2_Shader_ISA_November2020.pdf.

Chris Brennan. 2016. Getting the Most Out of Delta Color Compression. https://gpuopen.com/learn/dcc-overview/.

Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes Image Rendering Architecture. In Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87). Association for Computing Machinery, New York, NY, USA, 95–102. https://doi.org/10.1145/37401.37414

Shin Fujieda and Takahiro Harada. 2022. Progressive Material Cache. In SIGGRAPH Asia 2022 Technical Communications (SA '22). Association for Computing Machinery, New York, NY, USA.

Takahiro Harada. 2016. A Framework to Transform In-Core GPU Algorithms to out-of-Core Algorithms. In Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (Redmond, Washington) (I3D '16). Association for Computing Machinery, New York, NY, USA, 179–180. https://doi.org/10.1145/

2856400.2876011

Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In Proceedings of the 5th High-Performance Graphics Conference (Anaheim, California) (HPG '13). Association for Computing Machinery, New York, NY, USA, 137–143. https://doi.org/10.1145/2492045.2492060

MAXON. 2022. Zbrush. https://www.maxon.net/en/zbrush

Darwyn Peachey. 1990. Texture on Demand. Animation Studios Technical Memo #217.

Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering Complex Scenes with Memory-Coherent Ray Tracing. In Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97). ACM Press/Addison-Wesley Publishing Co., USA, 101–108. https://doi.org/10.1145/258734.258791

Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient Incoherent Ray Traversal on GPUs through Compressed Wide BVHs. In Proceedings of High Performance Graphics (Los Angeles, California) (HPG '17). Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3105762.3105773