High-Performance Graphics 2025
A. Knoll and C. Peters
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 44 (*2025*), *Number 8*

RenderMan XPU: A Hybrid CPU+GPU Renderer for Interactive and Final-Frame Rendering

Per Christensen, Julian Fong, Charlie Kilpatrick, Francisco González, Srinath Ravichandran, Akshay Shah, Ethan Jaszewski, Stephen Friedman, James Burgess, Trina M. Roy, Tom Nettleship, Meghana Seshadri, and Susan Salituro

Pixar Animation Studios

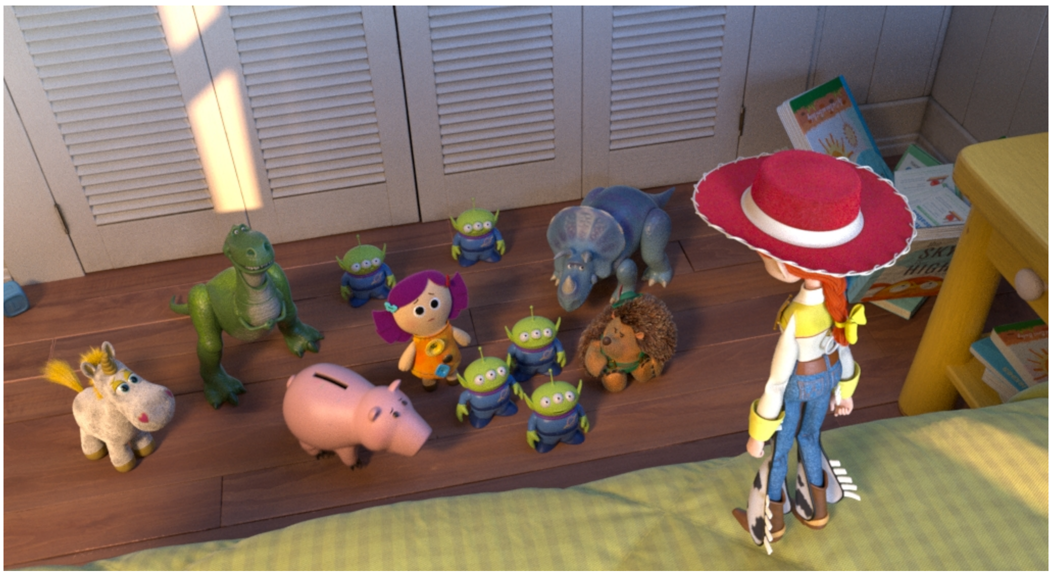

Figure 1: *Bonnie's room rendered at 1024×554 resolution with 1024 samples per pixel. Compared to the previous (CPU-only) version of RenderMan, this image renders 2.3 times faster on a CPU, 8.0 times faster on a GPU, and 9.9 times faster using both a CPU and a GPU. ©Disney/Pixar.*

Abstract

RenderMan XPU is a rewrite of Pixar's RenderMan renderer, designed to run on both CPUs and GPUs. Like its predecessor, it is a progressive path tracer, suitable for both interactive previews and high-quality final-frame rendering, but it utilizes modern hardware and software techniques to run significantly faster. Most source code is shared between the two platforms; code for materials (bxdfs) and light transport (integrators) is compiled with a C++ compiler for CPUs and a CUDA compiler for GPUs, with templating, specialization, and a few macros to handle syntax differences and parallel execution abstractions. The shaders that provide the material input values are written in OSL; we use LLVM so that the same OSL code will run on both types of hardware. Only the low-level ray tracing code and texture lookup and caching code is separate. Typical speedups over our previous renderer (for high-quality final-frame images) are 1.8× to 2.3× on CPUs, 5× to 10× on GPUs, and 6× to 15× on both.

1. Introduction

Pixar's RenderMan has been used to render high-quality animation and visual effects for hundreds of movies, and is also used for interactive previews. This paper describes RenderMan XPU, the newest generation of RenderMan. We call it "XPU" because it can run on CPUs, GPUs, or both. Our novel contribution is a renderer that can utilize heterogeneous hardware, provides the features necessary for final-frame rendering, and is fast enough for quick feedback during interactive preview work. Our main design goals are:

- Performance: To obtain high performance by running on heterogeneous hardware (CPUs and GPUs) and fully utilize their respective strengths with respect to compute power and memory size.
- Modular architecture: a) Several different integrators for shape visualization, texture and lighting layout, and full path tracing. b) Many different material models ranging in complexity from constant color to a complex material with more than 120 input parameters. c) Material parameters provided by texture maps and programmable shaders.
- Versatility and consistency: A *single* renderer designed for interactive previews (with quick feedback during design decisions), and also for final-frame movie rendering (with all the required advanced features), thereby ensuring predictability and consistent results.

Most of the XPU codebase is shared between CPUs and GPUs. This has two major benefits: we will always render the same perceptual image on the two platforms (modulo floating-point differences), and it is simpler to maintain and debug the codebase.

Materials (bxdfs) range from simple constant-colored emitters to complex, physically realistic (but artist controlled) many-layered materials with multiple specular lobes, non-Lambertian diffuse, iridescence, fuzz, subsurface scattering, and more. The material sample-generation and evaluation functions are written in a C++ subset, compiled with a C++ compiler for CPUs and a CUDA compiler for GPUs. We utilize templates, specialization, and a few macros to handle syntax differences and parallel execution abstractions that translate to explicit loops over shading points in C++, but to a single-point abstraction in CUDA.

Typically, the material parameter values (bxdf inputs) are provided by shaders. These shaders are written in OSL (Open Shading Language) [GSKC10]. The same OSL shader runs on both types of hardware: the OSL shading code is compiled with LLVM (low-level virtual machine) with runtime specialization for execution on CPUs and GPUs.

The only code that is totally separate for the two platforms is low-level ray tracing and texture cache lookups. We use BVH building and ray intersection similar to Embree [WWB*14] for CPUs and CUDA code as an alternative to OptiX [PBD*10] for GPUs. For texture tile caching, we initially implemented a solution similar to Garanzha et al. [GBPG11], but ultimately came up with a different approach.

RenderMan XPU is a complete rewrite. After initial prototyping and testing, this project started in earnest in 2017. We focused first on look development since the scenes are simpler (less geometry and simpler illumination) and require fewer features than final frames; our goal was to replace the in-house GPU renderer in the real-time shading preview tool Flow. More recently we have moved on to scenes with the full geometric and shading complexity of typical movie frames. When the same renderer is used both for previews and final rendering, the artists have confidence that the decisions they make in the interactive session will be faithfully carried through to the final frames.

Even though RenderMan XPU is a rewrite of the renderer, the functionality is the same as the previous version (RenderMan RIS) from a user point of view — with a few carefully chosen omissions — but much faster. All the advanced features needed for production rendering are present: motion blur, depth of field, global illumination, subsurface scattering, volumes, arbitrary output variables (AOVs), light path expressions (LPEs), light linking, adaptive sampling, checkpointing, deep output, and more; please see Christensen et al. [CFS*18] for more details. (One exception is that we still need to implement efficient light selection that is able to handle more than a few dozen light sources.)

With our new architecture, rendering is faster than the previous version — even on the same CPU hardware. The speedups compared to RenderMan RIS vary depending on geometric and shading complexity and many other factors, but for our benchmark scenes, we typically see speedups around 1.8× to 2.3× for CPU rendering, 5× to 10× for GPU rendering, and 6× to 15× for combined CPU+GPU rendering. In extreme cases, we have seen speedups of up to 18×. The CPU speedups — where XPU runs on the exact same hardware as RIS — are caused by better data access and execution coherency.

For now, RenderMan XPU only runs on CPUs from Intel and AMD and on GPUs with CUDA from Nvidia. But the architecture is designed to be flexible enough to support a wider variety of platforms; ongoing porting work targets Intel GPUs via SYCL, and we have compiled for Apple Metal as a proof of concept. CPUs usually have more memory than GPUs, so truly massive scenes can currently only render on CPUs. (However, future unified memory systems might change that.)

This paper starts with an overview of related work, then describes the architecture of the RenderMan XPU renderer and its key features and capabilities, followed by performance results, discussion and future work, and ends with a conclusion.

2. Background and Related Work

The most relevant related work is earlier versions of RenderMan, interactive rendering in movie production, and high-quality final-frame production rendering on CPUs and GPUs.

2.1. Earlier Versions of RenderMan

The first version of RenderMan was based on the Reyes algorithm [CCC87]. It was used to render the first CG animated feature film, Pixar's *Toy Story*, and hundreds of other movies — both animated movies and visual effects for live-action movies. More information about the origins of RenderMan can be found in e.g. the books by Upstill [Ups90] and Apodaca and Gritz [AG00]. We later augmented the Reyes algorithm with multithreading, ray-traced shadows and reflections, global illumination, level-of-detail tessellation, subsurface scattering, fast point-based approximations, and many other improvements [CFLB06; Chr08].

RenderMan RIS [CFS*18] was a partial rewrite of the renderer, where the front-end was switched from the Reyes architecture to ray casting, making it a full path tracer. Most of the low-level ray-tracing, tessellation and texturing code was carried over from the ray-tracing extensions to Reyes that we had introduced over the years. Fully embracing path tracing enabled better progressive and interactive rendering and higher efficiency on computers with many CPU cores.

2.2. Interactive Rendering

Pixar's Lpics [PVL*05] and ILM's Lightspeed [RKS*07] interactive relighting tools were designed to make light placement and adjustments more efficient. With Lpics, deep framebuffers were generated using RenderMan, containing the position, normal, surface color, etc. of the geometry in each pixel. Then (hand-written) simplified light shaders were run on a GPU for each light source that changed position or parameters. Due to the image-space framebuffer, Lpics was restricted to static scenes. The Lightspeed system was more general; it handled transparency, motion blur, depth of field, subsurface scattering, and indirect illumination (by precomputed links with weights between pixels). Lightspeed had automatic shader translation from RSL to Cg for GPU execution, thus avoiding a manual shader translation step. However, the automatic translator had to be updated when the RSL language was modified. Both tools eventually fell out of use because simplified shaders got out of sync with their full counterparts, and because the images they rendered were not predictive of the final render. In practical use, it is more convenient to have a single renderer for both previews and final frames, and a single set of shaders to maintain.

For the last 12+ years, Pixar's internal real-time shading tool Flow and its GPU viewport renderer RTP (Real-Time Previewer) [Nah13] have been extensively used by shading TDs (technical directors) on Pixar productions. RTP runs purely on GPUs and uses Nvidia's OptiX library [PBD*10] for all ray tracing. RTP is very fast and interactive. It supports CUDA-based shading, requiring separately maintained shaders and bxdfs. The user can change the viewpoint, illumination, bxdf parameters, textures, and shader networks. We have incorporated some RTP code and ideas into RenderMan XPU. The most recent iteration of Flow has RenderMan XPU as its default renderer, with RTP still available. It is our intent that XPU will eventually completely supplant RTP. Using RenderMan XPU for both interactive previews and final frames ensures consistency and no surprises.

2.3. CPU and GPU Production Rendering

MoonRay [LGXT17] is DreamWorks' vectorized CPU production renderer. It uses Intel's Embree library [WWB*14] for tracing rays, providing good SIMD utilization for single rays, ray packets, and ray streams. They also vectorize shaders and shading, texture lookups, and the integrator using Intel SIMD instructions. With 8-wide SIMD instructions, they got an average 1.3–2.3× speedup in overall rendering time for typical frames. Our system targets similar SIMD utilization on CPUs.

Early work on rendering on both CPUs and GPUs includes Nvidia's Gelato renderer [WGER05], which used a Reyes rasterization approach on the GPU and multiple rendering passes for motion blur and depth of field.

Weta's PantaRay [PFAH10] used GPUs to precompute ambient occlusion in complex production scenes. The results could then be efficiently convolved with environment maps for quick image-based lighting and relighting.

Several commercial production renderers have a GPU version, for example Arnold [GIF*18], V-ray, and Redshift. Similarly, Weta Digital have an in-house CPU renderer, Manuka [FHL*18], and a separate GPU renderer, Gazebo, and Disney have their in-house renderer, Hyperion [BAC*18], and a separate GPU pre-viz renderer. Those are separate pieces of software, and the GPU renderers do not have the full feature set of the original CPU renderers.

Nvidia's Iray renderer [KWR*17], Sony's Spear renderer [SHE*24], and the Karma XPU renderer from SideFX [Sid25] have many architectural similarities with RenderMan XPU. Iray shares our philosophy of writing kernel code abstracted in a hardware-agnostic way and sharing it as much as possible between CPUs and GPUs. Many implementation choices in Spear — like selective just-in-time OSL compilation and string hashing — are the same as ours. Among the differences are that Spear ended up using megakernels (with "micro-jittering" for better execution coherence), and that Spear is not designed to run on CPUs and GPUs simultaneously. Like RenderMan XPU, Karma XPU can render on both CPUs and GPUs; however, Karma XPU is described not as a replacement for the Karma CPU renderer, but as a high-performance alternative with a limited feature set.

We highly recommend reading the excellent Spear paper to compare and contrast with our approach. We also recommend reading the latest version of the PBRT book [PJH23], which has an in-depth chapter about the porting of PBRT to a GPU wavefront path tracer (similar to our choice of wavefront over megakernel). PBRT obtained speedups from 8× to 37× for GPU vs CPU rendering of a representative scene, depending on the chosen hardware.

3. Architecture and Design Principles

The key abilities that guided our design are: high-performance rendering on heterogeneous hardware, a modular architecture for flexibility and scalability, and advanced rendering features for high-quality images. In this section we describe our design goals in more detail, and present the architecture we created in order to meet those goals.

3.1. Design Goals

The design goals for XPU are different from RIS. Our design decisions were guided at the outset by the goal of fast parallel execution on both CPUs and GPUs, allowing the same renderer to produce equivalent results on existing CPU-based render farms, with the ability to easily swap in GPUs to accelerate rendering for interactive or offline renders when appropriate. The focus on data-parallel design is pervasive throughout XPU, and it is largely responsible for the improved CPU performance we see with XPU compared to RIS, combined with hardware-specific tuning of the work scheduler. Optimized stream compaction and sorting for more coherent data accesses, parallel execution over wavefront kernels, and effectively non-divergent vectorized operations (SIMD or SIMT) are all important considerations in order to obtain high performance on CPU or GPU hardware platforms.

Equally important top-level goals are memory efficiency, optimizing time-to-first-decision, interactivity, and an emphasis on artist-focused design. We mention a few aspects of each of these high-level design goals here.

Memory efficiency is a primary design goal to ensure that production scenes can be rendered within the more limited memory resources of GPUs. The already compact data structures in RIS were streamlined and compressed further for XPU so that both CPU and GPU renders scale up to the memory requirements of production scenes. We also perform some of the scene load and startup work on the CPU (in parallel), and only transfer the final compressed data structures to GPU memory for rendering. As part of this process, any complex data structures are converted into more GPU-friendly array-based data layouts. This approach often results in the GPU portion of the rendering work requiring significantly less overall GPU memory, compared to the overall total CPU system memory used by a CPU-only render.

Time-to-first-decision is a metric that measures how quickly an artist is able to make meaningful decisions impacting the lighting, materials, or other aspects of their work. Whereas RenderMan historically has prioritized speed and memory usage of overall final frame rendering, with startup time more of a secondary goal, for XPU we designed the renderer also to optimize the startup and load time. In practice, this meant choosing variable bit-rate compression algorithms for data structures, compressing spatially coherent sections of the data in parallel at load-time, and ensuring that we make use of asynchronous data transfers of the compressed data structures from CPU to GPU. This allows for concurrent startup-time processing while the data is streamed into GPU memory for rendering on the GPU. Other aspects of load and start-up time in XPU are better optimized, parallelized and generally streamlined compared to RIS.

Finally, XPU is designed from the outset for interactivity with a focus on improving the artist experience. This means that the time it takes to respond to scene edits (e.g., time-to-decision), updating progressive refinement of renders efficiently, choosing an appropriate batch size when performing interactive renders, integration of high-quality interactive denoising, and other aspects that impact the user experience are all very important.

3.2. Wavefront Path Tracing

XPU uses a wavefront-based design based on tracing a "wave" of rays over the scene [LKA13; PJH23], as opposed to a design based on megakernels (where individual ray paths are processed in isolation). In our performance testing, we found that working with packets of rays allows for many significant performance advantages compared to megakernels: better use of cache hierarchies; improved load balancing across parallel work units; the ability to sort rays for spatial and directional coherency; and sorting ray hits by material properties for shading coherency. The improved coherency from sorting pays dividends in several areas, including improved texture cache access, reduced SIMD/SIMT divergence, and more opportunities for vectorization. Given our focus on optimizing heterogenous workloads, these improvements are realized on both CPU and GPU hardware architectures. While Shader Execution Reordering (SER) permits some limited coherency on Nvidia hardware, the architecture of RenderMan XPU is designed to exploit a high degree of coherency on any hardware, CPU or GPU, and thus wavefront path tracing is an important aspect of the architecture.

The unidirectional path tracing integrator in XPU works in stages over the packets of rays. First, we initialize the framebuffers and camera rays. Next we perform a loop over the ray depth: for each depth, we trace the rays (either initial camera rays or the next indirect bounce of rays). We consider the emission and direct illumination at all shading points, which involves shader evaluation, and performing multiple importance sampling (MIS) between bxdfs and lights. We make use of queues to buffer work between kernels, including a shadow queue for deferring work over the shadow rays. Colors are written into the framebuffer after the shadow trace kernel. Partially or fully transparent geometry adds more complexity due to the need to mix shading and tracing. Our approach uses an upfront determination of a set of *material hints* that indicate whether geometry is trivially opaque, or requires shading to determine transparency or homogeneous volumetric extinction. When tracing path or shadow rays, any hits on nearest geometry not trivially opaque are temporarily retained in an additional buffer. Once all rays have been fired, these retained hits are shaded to calculate transmittance. For path rays, the transmittance is used to stochastically terminate the ray; for shadow rays, the transmittance is multiplied into the shadowing term. In either case, the renderer next looks for more non-opaque geometry and we restart the process by shooting continuation rays. Figure 2 shows a diagram of the path tracing kernels.

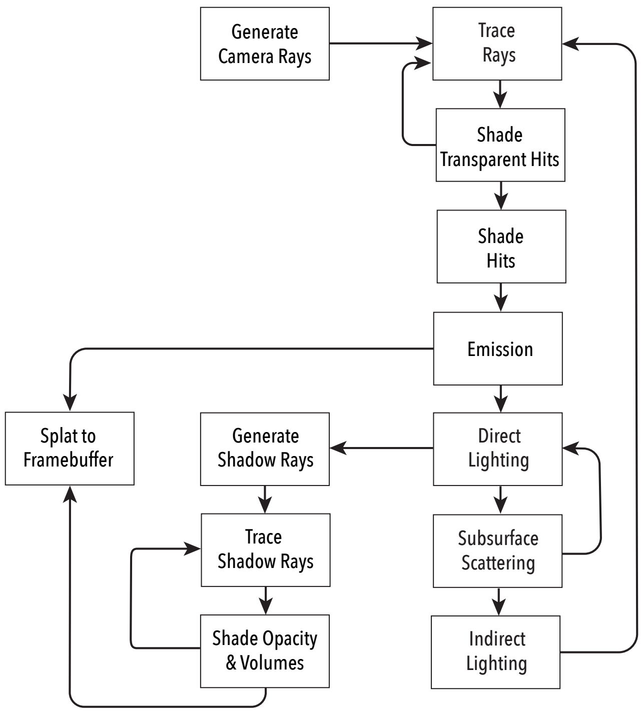

Figure 2: *Kernels in RenderMan XPU*

3.3. Vectorization

We organize data into data-parallel GPU-friendly data structures and compile the code with both the Intel C++ compiler (SIMD) and Nvidia CUDA compiler (SIMT) to vectorize the code. We've worked with engineers at Intel to best take advantage of a mix of the auto-vectorizer and explicit OpenMP SIMD pragma directives with their compiler, ultimately optimizing both OSL and various kernels for up to 16-wide SIMD (AVX512). The Nvidia compiler produces SIMT code which we profile to evaluate performance and look for divergent code paths. We make use of templating to ensure a single code path that the auto-vectorizer is able to exploit for vectorization, and in several cases, we break up code paths that were divergent (thus defeating SIMT on the GPU or the auto-vectorizer according to the vectorization report), so that we instead perform the work in multiple passes, where each individual code path is highly vectorized. In addition, we also do explicit vectorization using intrinsics in a few places such as the ray tracing kernels.

3.4. Ray Tracing

The bounding volume hierarchy (BVH) in XPU is organized into two levels: one set of low-level BVHs over each geometry primitive, and a separate top-level BVH over the geometry instances. The multi-level BVH permits fast scene edits, because when the geometry of one or a few primitives is modified, only their local BVH needs updating, and then the relatively small top-level BVH over the geometry instances can be quickly rebuilt. Nesting of instances is supported in XPU, and this works via a separate BVH traversal per level of instancing in order to form an instance path from the root instances to the leaf instances within the nested instancing hierarchy.

XPU sorts traced rays by spatial location and direction: the rays are binned based on ray origin and direction, with the origin bins taking priority over the directional bins when forming the sort key. Sorting improves coherency and reduces divergent paths of execution during ray traversal. For GPU ray-tracing code paths, we make use of per-geometry traversal kernels, and separate kernels for motion blur versus non-motion blurred traversal paths, nested instancing, and for volumes, and each combination thereof. These code paths permit efficient traversal, in particular on the GPU because it helps minimize divergent code paths and register usage, and increases occupancy.

Ray hits are also sorted — by geometry type, pattern network, and bxdf binding. This sort maximizes the batch size for the kernels comprising shading: primvar interpolation (which differs by geometry type), and OSL pattern networks. It also maximizes the efficiency of subsequent bxdf sample generation and evaluation in the path tracer. Prior to shading, we use stream compaction to group the ray hits for coherent shading kernel execution, and kernels are launched in separate streams to maximize available GPU resources.

In production, high numbers of trace subsets are used — between 70 to 80 on average, up to 450 maximum — for selectivity in ray tracing when it comes to visibility, shadowing, and direct light linking. While in theory trace subsets could be handled by creating separate root level BVHs, this incurs a significant memory and time cost because subsets often overlap their membership. While having one visibility or mask bit per actively used subset would be ideal, in practice we can still use fewer bits (such as 24 or 32 bits) effectively by mapping multiple subsets to the same bit in the mask, and then performing a final full filtering only at the leaf level during BVH traversal. This works well as long as we don't try to map too many subsets onto the same bit; in practice, trying to map 450 active subsets onto 8 hardware mask bits provides very limited value for ray culling during traversal, and more hardware mask bits (e.g. 32 bits) would be necessary to alleviate this issue. On the GPU, the overhead for context switching between RT cores and general-purpose execution units means that calling custom intersectors to cull based on subset membership during the inner loop of traversal can significantly impact performance, hence the importance of direct hardware support for more mask bits.

In the beginning of the XPU project, we implemented both CUDA and OptiX code paths, and profiled memory usage, BVH build times, ray traversal times, and other metrics. At that time, although the raw OptiX ray traversal speed was impressive for some scenes, our CUDA version offered faster BVH build times and used less memory than OptiX 7. Notable issues included lack of RT hardware acceleration for motion blur in many cases, limited hardware visibility mask bits, and only some of the base geometry types were accelerated by the RT hardware cores. In the most realistic production scenarios, this required use of custom intersectors, which shifted much of the work away from the RT cores. Profiling and testing also indicated significant overhead for context switching between the RT core units and execution of custom intersectors, which meant we were unable to realize improved metrics with OptiX on most Pixar production scenes. We also found look differences for curves between OptiX and the RT hardware intersectors compared to our C++ and CUDA versions. Based on this early evaluation, we moved forward with the CUDA implementation.

Since then, newer versions of OptiX and RT hardware have substantially improved BVH build times and memory usage, ray traversal is faster, shader execution reordering was added, and more code paths are fully hardware-accelerated, including additional motion blur acceleration. We are encouraged by the progress and look forward to re-evaluating the latest versions of OptiX and RT hardware cores.

3.5. Tessellation and Displacement

Subdivision surfaces [CC78] and polygon meshes are the most important input geometry types in modern rendering. Our approach in XPU is to completely tessellate subdivision surfaces to *micropolygons* as early as possible in the rendering pipeline. Along similar lines, large polygons that have displacement are also handled by tessellating into smaller micropolygons, running displacement, and retaining only the displaced result. This way, XPU can focus on optimized (micro)polygon handling. As a consequence of this approach, all tessellation and displacement is completed before any rays are traced into the scene.

The transformation from high-level meshes to micropolygons occurs in multiple stages. The transformation in each stage is irreversible, a decision made to optimize memory since we do not need to keep around multiple representations of the geometry in order to repeat a transformation in the future. A downside of this approach is that when a geometry primitive edit is made, almost the entire pipeline and all of its transformation stages needs to be fully re-executed (but only for that primitive; only instances of this one primitive need to be updated in the global BVH). While this delays the time to first pixel compared to a rendering architecture that lazily tessellates geometry, it allows the renderer to compute optimal GPU-friendly representations for the data, as well as bounding volume hierarchies over large meshes. We also do not incur this pipeline cost on other common types of edits, such as transformation or material changes.

Currently all tessellation and displacement is done on the CPU since we have not yet implemented a task manager to efficiently distribute these tasks to both CPUs and GPUs.

3.6. Light Transport

Light transport is simulated with an integrator that implements a unidirectional Monte Carlo path tracing algorithm [Kaj86; CFS*18; PJH23], iterating over light reflection and refraction bounces "backwards" from the camera to the light sources. Each path is terminated when the ray exits the scene, by Russian roulette [AK90], or when a maximum depth has been reached. Each material (bxdf) has a sample generation and a sample evaluation function. The light sources are sampled explicitly at each bounce (aka. "next-event estimation"). In scenes with many lights, a subset of lights is chosen for each shading point in each iteration, and a sample point is stochastically chosen on each of those lights. The bxdf and light samples are combined with multiple importance sampling (MIS) [VG95]. As an additional noise reduction technique, we select more light source candidates in each iteration than we intend to illuminate the surface, compute their illumination and evaluate their bxdf response, and then pick the best combination using resampled importance sampling [TCE05]. That combination then becomes the illumination we push onto the shadow ray queue.

Subsurface scattering (sss) is an interesting wrinkle in our wavefront approach as seen in Figure 2. We compute direct illumination at all the original hit points, and then select which points should have subsurface scattering in that iteration (based on the relative weights of subsurface scattering vs. other lobes). At each point with subsurface scattering, we compute a new shading point by tracing sss rays; at those new shading points we update the path throughput and compute the (diffuse) direct illumination there. Now we have a mix of non-sss points with their original position and sss ray hit points with updated positions. We can then continue with the next bounce of indirect illumination from those mixed positions.

Modern versions of RenderMan are physically based at heart, but there are many ways to manipulate the light transport in non-physical ways — all in the interest of art directability and creativity. XPU supports plug-in light filters bound to light sources that can be used to arbitrarily modify direct lighting contributions. Light and shadow linking may indicate that lights should only illuminate or be shadowed by some of the objects in the scene. Trace visibility can be used to make objects invisible to the camera, to reflection/refraction rays, or to shadow rays. Trace subsets can be used for more fine-grained visibility controls, like making only certain other objects visible in the reflection from some object.

XPU also has simpler integrators for fast visualization of object shapes and shading, for example showing surface patches, surface normals, textures, or ambient occlusion. Figure 3 shows Woody's head rendered to visualize the subdivision mesh, and rendered with full path tracing and subsurface scattering. RenderMan XPU also supports non-photorealistic styles such as pen-and-ink, charcoal, cross-hatching, contour lines, and many more.

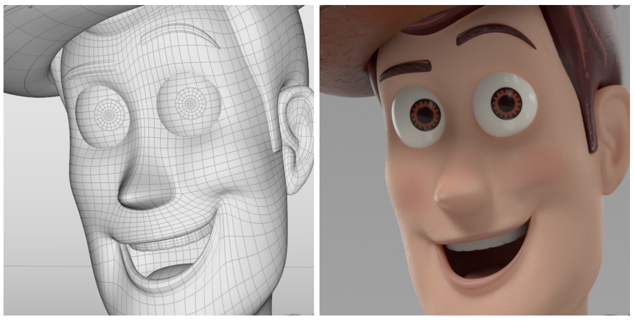
Figure 3: *Woody's head rendered with two different integrators. ©Disney/Pixar.*

3.7. Virtual Device Architecture

We handle *task management* on heterogeneous devices using a virtual device architecture. This allows us to abstract away implementation details such as work sizes and context management to the devices on which the execution happens.

A central part of this abstraction is the usage of a scheduler that determines the amount of work (pixel integration) sent to the devices. The GPU, being a large parallel machine, can only work efficiently when its working set is very large. In XPU, the GPU working set is 500k elements, whereas the per CPU thread working set size is 1024 elements (32×32 pixels). This maximum size was chosen empirically since the amount of state required to maintain for each working element grows almost linearly (dependent on AOV count, etc.), the limiting factor being the GPU memory capacity. In XPU, the pixels to be integrated are coalesced into buckets that span a rectangular region of the screen. This was deliberately chosen to improve the primary ray coherence (ray traversal and shading), which can have a huge impact on render times. However, as the number of bounces increases, the amount of extractable coherence decreases. In the core render loop, there are very few points of synchronization between the devices. The virtual devices (and their respective worker threads) operate independently. One of the synchronization points is framebuffer accumulation and we employ per-bucket locks to make sure different parts of the screen can be accumulated independently of one another.

4. Key Features and Advanced Capabilities

4.1. Surfaces, Curves, Points, and Volumes

The geometry representation in XPU is split between geometry primitives ("prototypes"), and instances of those prototypes. "Primitive variables", or *primvars*, include vertex positions, normals, and any other input data used for shader variation over the surface; they represent the bulk of the input geometry data and are bound to the prototypes. Materials and a small amount of per-instance override data are bound to instances.

Geometry prototypes and instances are processed in separate pipelines prior to any ray tracing. As shown in Figure 4, various stages of the pipeline are executed depending on actions performed by the user, and are run in parallel on different prototypes or instances. For instances, an important stage is creating a unique material binding across all instances of a prototype. The binding contains a mapping from primvar names requested by the shader to integer primvar offsets in the geometry prototype. The mapping avoids string processing of primvar names during shader execution, as the offset can be used directly by the primvar interpolation kernels prior to shading. The bindings can be quickly regenerated for any edits to the materials themselves; because we retain all primvars in the geometry, any other existing material bindings are unaffected because their offsets do not change. These mappings allow us to achieve a key goal: the ability to quickly edit materials on geometry without having to re-execute the geometry pipeline.

Geometry primitives in XPU are represented by surfaces, curves, points, and volumes. We will now describe these and their processing pipeline.

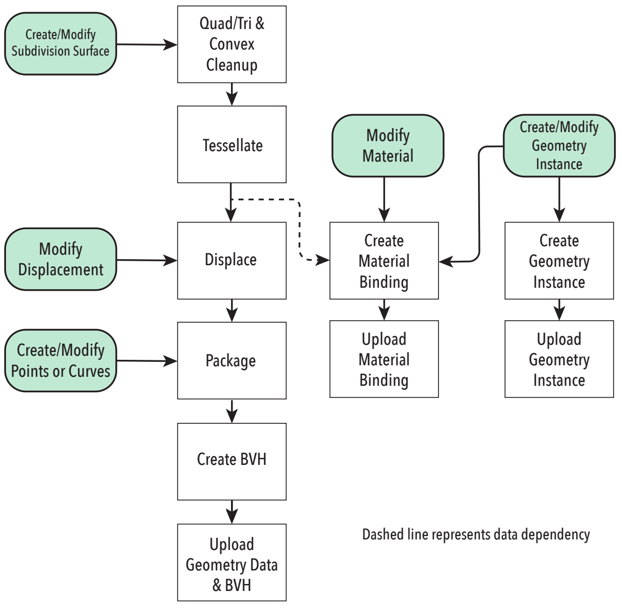
Create/Modify Subdivision Surface
Quad/Tri & Convex Cleanup
Tessellate
Modify Material
Create/Modify Geometry Instance
Modify Displacement
Displace
Create Material Binding
Create Geometry Instance
Create/Modify Points or Curves
Package
Upload Material Binding
Upload Geometry Instance
Create BVH
Upload Geometry Data & BVH
Dashed line represents data dependency

Figure 4: *XPU Geometry Processing Pipeline*

4.1.1. Surfaces

For surfaces, XPU supports subdivision surfaces and polygon meshes; we have chosen not to support NURBS, bicubic patches, and quadrics, but in the future we could support these via tessellation to polygon meshes.

XPU devotes much effort to processing subdivision surfaces, from high-level geometry to final ray-tracing ready micropolygons. This occurs over multiple discrete stages. The first stages include processing to ensure that any incoming data is well formed: holes in faces are eliminated, concave faces are converted to convex, and any faces with five or more sides are converted into quads and triangles. Any bad floating point data (from e.g. degenerate input polygons or NaN particle simulation data) is filtered at this stage. Other data filtering may occur here; in particular, deformation motion blur (motion blur described by movement of individual vertices through time) is supported compactly by retaining only multiple values of the vertex positions, with shared topology. XPU does not support deformation motion blur of arbitrary primvars, as this situation rarely occurs in production rendering. Therefore, if there are values of primvars (other than positions) that change in time, we only keep the values from the first time step.

The first significant transformation phase is *tessellation*, where subdivision surfaces or large displaced polygonal faces are turned into a set of smaller micropolygons. Note that XPU does not use a multi-resolution geometry cache [CFLB06]. In recent years, we have found many overmodeled assets from production, with each subdivision face smaller than a pixel and requiring tessellation only to a few micropolygons. Hence, it is generally more efficient to commit to an irreversible transformation of the subdivision mesh to a fully tessellated polygonal representation. Even with additional storage for the limit surface analytic tangent vectors and normals, with compression this is more efficient in memory than retaining a subdivision mesh boundary representation that allows for arbitrary re-tessellation. For subdivision surfaces, we leverage OpenSubdiv [Pix23] to compute the limit surface quantities of the position data as well as all primvars.

The target size for tessellation is determined by a number of user selectable factors, with the default strategy being to compute micropolygons whose dimensions satisfy a projected screen size, measured in pixels. Many scenes contain large amounts of off-screen geometry. In order to reduce memory consumption when such geometry is close to the camera's near clipping plane, we aggressively under-tessellate such geometry if possible, especially if it is not over-modeled in the first place. This may lead to artifacts if the off-screen geometry is reflected back into camera view, but in practice objectionable artifacts are rare and the under-tessellation can be dialed back on a case-by-case basis. In cases where temporal stability of tessellation is necessary (especially when objects move in and out of camera), we have a user setting to determine the rate using a spherical instead of a planar projection; this is both robust and reasonably memory efficient.

It is important in production rendering to avoid artifacts that may be incurred by tessellation. Watertightness of tessellated meshes avoids pinholes in the alpha channel and errors tracking nested dielectrics. In order to avoid T-junctions on geometry caused by dicing parametric patches into rectangular collections of micropolygons, we use a variant of the DiagSplit algorithm [FFB*09].

The next significant transformation is *displacement*, which uses OSL pattern networks to compute a new displaced position for every vertex on the tessellated mesh. As inputs to the pattern network may depend on data that varies even on a single shared vertex (e.g., the face normals on a cube are different on the vertices of the adjacent shared edges), an extra averaging step is required to ensure that no cracks occur in the displaced results, and that the final mesh is stable through animation. As a nod to the importance of interactive editing of displacement, we retain the undisplaced vertex positions and normals even after displacement, which allows us to rerun the displacement upon shader edit without having to rerun the tessellation stage.

The next transformation is a *packaging* stage, which uses multiple techniques to compact the geometry data into a compact, efficient form. After packaging, the data is considered read-only and ready to be uploaded from host to device. Several compression techniques are used, which tend to favor speed of compression over high compression ratios. Buffers containing floating point primvar data are sorted and duplicate values removed. The sorted floating point data requires the use of integer indirection indices to look up the data during ray tracing; these indices are converted into offsets, which simplifies encoding using variable bit depths. Our encoder achieves compression ratios of 3-5×, but comes at moderate runtime compute cost: random access requires around 24 instructions and 3 memory reads. Normals are compressed into 4 bytes using octahedral mapping [CDE*14]. All buffers, including both the floating point data buffers and the variable bit-encoded index buffers, are globally deduplicated.

Figure 5 shows an occlusion render of the train station set from Pixar's *Coco*. This scene is highly detailed: the 53,874 subdivision meshes and 2,251 polygon meshes in the instance prototype definitions tessellate to 72.4 million micropolygons. Without packaging, the uncompressed geometry data for this scene would require 21.2 GB of memory. With compression and deduplication, the required memory shrinks to 6.2 GB. Even with the additional overhead of BVHs and other buffers, this scene fits well within the memory of a 16 GB GPU.

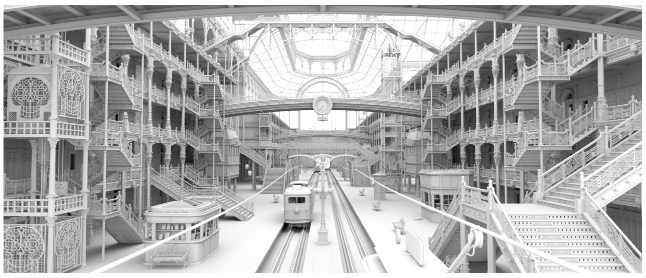

Figure 5: *Train station from the movie* Coco. *©Disney/Pixar.*

The final stage for the geometry processing is building a *per-primitive BVH*. The internal nodes of the polymesh BVH follow that of the main BVH, except that we specialize the representation of the internal nodes based on the number of vertices in the polymesh. The indices to the primitives stored within the BVH node use the smallest possible bit depth. For meshes with upwards of a million vertices, we quantize the bounding box offsets using 1 byte float per dimension in order to save a considerable amount of memory, trading off some compute expense. The BVHs built for use on the CPU and the GPU are identical.

4.1.2. Curves and Points

Objects like fur, hair, and grass are represented by a curves geometry primitive. Key to this primitive is the assumption that each individual curve has no variation in primvar values across its width — the only useful parameterization is over the length of the curve.

Curves are represented as collections of linear or cubic segments with per-vertex widths. If optional normals are supplied, the curves are treated as oriented flat ribbons (like grass blades); otherwise, they are treated as round cylinders. In XPU, we have chosen not to tessellate curves into micropolygons; instead, we trace rays directly against the linear or cubic splines [NO02]. This is somewhat at odds with our previous observation that production geometry tends to be overmodeled; it is often the case that short fur is modeled with tens of control points packed into strands shorter than a few pixels. However, we have also observed that hair can run to the opposite extreme: long flowing tresses can be efficiently represented with just a few cubic control points rather than many small tessellated micropolygons. For now, we have chosen to focus on optimizing an untessellated curves geometry representation in XPU.

Since we support neither tessellation nor displacement of curves, the only significant transformation stages in its pipeline are the packaging and BVH build stages. The compression techniques in the packaging stage follow that of the polymesh primitive, with one additional step: for curves using the Bézier basis, we compress four control points into the space of three by converting the two middle control points into tangent vectors and encoding them in half precision.

Because curves are not tessellated into smaller, approximately square micropolygons, the BVH for a curves primitive can be more complicated than for a polymesh in order to achieve acceptable performance. We also recognize that a curves primitive may contain thousands or even millions of individual strands. To balance these competing goals, the curves BVH uses a non-axis aligned, object oriented bounding box similar to Woop et al. [WBW*14], but with a SIMD-friendly quantized quaternion encoding: each child's orientation relative to the axis aligned bounding box is encoded in a quaternion quantized to 8 bits. The bound offsets are also quantized to a single byte per dimension. Figure 6 shows Dorothea from Pixar's *Soul*. All curves (including her hair, peach fuzz, and garments) were rendered with the Chiang hair bxdf [CBTB16]. Her hair has 29,297 individual strands with an average of 150.6 vertices per strand. The BVH built for just her hair rendered by itself at 1024×1024 pixel resolution has 2,078,260 nodes; using full precision BVH nodes would require 437 MB, whereas the compact quantized representation only uses 246 MB.

Particle effects are rendered using a dedicated points primitive, which efficiently scales to millions or even billions of individual points. Each point is represented by a position and a radius representing a sphere, along with any per-point primvars. We assume there is no variation of primvars on the spheres and that there is no useful parametric space for shading on the surface. Since no tessellation or displacement occurs, the package and BVH build stages are the only significant steps. The package stage follows that of polymeshes. For large numbers of points (currently over a million), we create a BVH whose nodes contain 8-bit quantized bounding

Figure 6: *Dorothea from* Soul. *©Disney/Pixar.*

boxes. The ray tracing intersector for points is a simple ray versus sphere test.

4.1.3. Volumes

Homogeneous volumetric effects that can be considered part of a material's light scattering are handled as relatively trivial special cases by the ray tracing kernels. *Heterogeneous* volumetric effects such as smoke, fire and clouds require further handling. Volume primitives modeling these effects are described using a bounding box and an optional associated filename. In production, volumetric fields are generally written to disk in an independent file format. For XPU, we have chosen to support only the OpenVDB format [Mus13], due to its ubiquitousness in industry and because we can take advantage of NanoVDB [Mus21] to efficiently interpolate data on both CPUs and GPUs. For each volume primitive, the main transformation stage is where the OpenVDB file is read from disk and converted to NanoVDB representation before being uploaded to the device.

Ray tracing of heterogeneous volumes departs from the handling of other primitives described in section 3.4. Modern volume rendering techniques rely heavily on Delta tracking methods [WMHL65] and derived variants such as residual ratio tracking [NSJ14]. As described in Fong et al. [FHWK17], we have found it more efficient to use a BVH fully aware of volume metadata used by tracking methods, in particular the extrema of the extinction coefficient. We use the *aggregate volumes* method, which creates an octree over all volume primitives, with each node containing metadata. The implementation in XPU follows that of RenderMan RIS, incorporating recent developments and extensions: we assume the filter width is always zero, eliminating any dynamic updates of the octree during ray tracing, and we extend the octree to handle transformation motion blur and visibility [Fon23].

After all volume primitives have been supplied to the scene, a separate pre-rendering stage is scheduled where the aggregate octree is built over all volumes. If the extinction is derived solely from the VDB file, we can compute the extrema directly from NanoVDB; in other cases where shading contributes to the extinction, we shade the volumes at voxels created at a user-defined frequency on both the CPU and GPU. The completed octree is uploaded to the device in a pointer-less, index-only representation, and remains static for the duration of the render. Each octree node contains extinction metadata over all volumes overlapping the bounding box plus an integer count and an integer offset into the list of volume primitives.

During ray tracing, the aggregate volume BVH is considered after all other geometry primitives have been traced. Rays traversing the octree nodes will pause in a node when tracking requires testing for interaction with volumes. Every volume listed in that node will require OSL shading to compute density, which also requires interpolation of inputs via NanoVDB. The density of all volumes is summed and used to decide whether the stop is a real interaction or a fictitious one to be ignored. Once tracking has been completed, and an interaction with the aggregate volume has been returned to the path tracer as a ray hit, the volume is treated as any other primitive in the system, with the only extra considerations being that the primvar interpolation will involve NanoVDB, and the bxdf will involve a phase function such as a double-lobed Henyey-Greenstein.

4.2. Programmable Shading with OSL

One cornerstone of a feature-film production renderer that provides the ability to be used across visual effects and animation is flexibility. One of the ways XPU provides this flexibility is through programmable shading. Based on the widespread use of Open Shading Language [GSKC10] in feature animation at Pixar, it was a logical candidate for supporting that programmability in XPU.

4.2.1. OSL everywhere

OSL uses a just-in-time (JIT) compilation strategy for shading networks that is built on top of the LLVM [LA04] compiler framework. Fortunately, LLVM has back-ends that can target both CPUs and GPUs, and OSL has been extended to support both of these back-ends, allowing authors to write the programmable shaders once in OSL, which then run in both places via the run-time compile. This helps XPU meet its goal of having the CPU and the GPU produce equivalent pixels, as both are running the same original code. While RIS supports both C++ and OSL for material pattern shading networks, XPU solely supports OSL patterns, and also extends support beyond RIS to sample and display filters. Going forward, XPU will continue to broaden the places that can be controlled with programmable shading via OSL, such as in driving the camera model, non-physical light and shadow filtering, etc.

4.2.2. Strings

Optimizing for specific data types such as strings can be important for rendering performance on the GPU. Strings are widely used for texture file names, names for settings, use of LPEs, access to AOVs, etc. Because we support only assignment and comparison of strings during ray traversal and shading, we can compare either pointers to the strings or hashes of the string values, rather than performing full string comparisons.

We typically choose to use a hash of the string contents. Because hashes do not change from run to run (unlike pointers), we can reuse previously cached JIT-ed and compiled code in many cases by using hashes. This ultimately yields a higher driver cache hit rate; see Stein et al. [SHE*24] for more details. We observed numerous collisions with 32-bit hashes and thus we extended this to 63-bit hashes with one additional bit reserved for a collision management scheme. We detect hash collisions and set the bit to 1 to indicate no collisions of the hash value, in which case the hash is used. However, when we detect a collision, this bit is set to 0, and then we resolve the collision by using a pointer to the string itself instead of the hash. Because the pointer is guaranteed to be unique, we'll always generate a correct picture. Thus, while a collision will result in an occasional driver cache miss, it will not produce incorrect results.

4.2.3. Optimizations for Interactive Look-Dev

One of the first deployments of XPU was for the internal look-development tool Flow. Flow is specialized for editing OSL shader networks, and initially could switch between RTP for fast preview and RenderMan RIS for final quality rendering. A benefit of introducing XPU was that it can directly use OSL for rendering instead of RTP's set of corresponding CUDA shaders. However, we soon ran into a drawback of this approach with interactive parameter editing. OSL's JIT model avoids the cost of unused shader functionality at runtime via dead code elimination and constant folding of input parameter values. This is great for optimizing final frame rendering where these values are not changing; however, in an interactive setting, it also means we must conservatively re-compile the whole shading network any time a parameter is changed or run the risk of incorrectly optimized shaders. Compounding the problem, OSL's batched extension to support SIMD code generation on the CPU and PTX (via LLVM) on the GPU greatly increase compile time.

We mitigated the compile time cost of every parameter edit by using an OSL feature that allows late-binding of parameters to geometric values in a buffer, instead of requiring them to be baked in during a compile. These buffer values can be easily changed by edits at run-time. Upon receiving a shader edit, the renderer will compare old and new pattern networks to decide whether it can proceed with a fast parameter-only edit, or requires a full re-compile due to changes in network topology. As a heuristic to avoid keeping *all* parameters live in the buffer and foregoing many optimizations, we keep only those parameters live that the user has explicitly set; all other inputs still at their default are compiled in, but will trigger a re-compile on first edit. This brought much needed interactivity into many edits and greatly improved the usability for interactive look-dev, particularly for the CPU. However, more work was needed on the GPU. Our GPU texturing system requires handles to textures that are constant at shader-compile time, and as previously mentioned, we only support limited operations on strings. Our shading relies on compile-time optimizations to transform dynamic manipulations of strings, used for flexibility and artist convenience, into constant run-time results. These optimizations are at odds with the need to keep things live for fast interaction, and complicated by the fact that *any* parameter could influence the compile time resolution of strings or texture handles; for example, a float parameter could be tested against a threshold to select between two texture filenames. In order to know which parameters are safe to keep editable, we added a dependency analysis pass to OSL to discover any parameters required to be constant in order for all strings and texture handles to be resolved to constants at compile time. This information is used to override requests by the renderer to keep those parameters live, as well as to maintain them as constant so that a re-compile will be required when they are edited.

These changes allowed for many shading edits on heavy assets to proceed in milliseconds instead of tens of seconds. With XPU, interactive editing has changed from clicking on a color and waiting for it to update to see what happened, to being able to interactively drag a slider during shader look development, getting us to the point where XPU can be used for similar use cases as RTP.

4.2.4. Closures

Closures are a way of communicating information from the OSL shader to the renderer about an implementation that needs to be made. For example, bxdf closures ask the renderer to provide some form of evaluation of the material. So-called "debug" closures are a way to pass a float/color value back to the renderer to write to the pixel buffer. Any closure defined by the renderer must be registered with the OSL shading system and must have id ≥ 0. (Id -1 and -2 are reserved for closure multiplication and addition, so that the shader writer can do simple combinations of closures.)

Closure pool allocation happens once during scene ingestion and expands in case there is not enough memory during rendering. We let closures pile up compactly in the pool of memory. Every shading point gets to store its list of closures in the order they are accessed. The count of closures per shading point is stored so that later when we need to process this tree of closures, we know exactly how many nodes will need to be unrolled into a stack (since recursively unrolling a tree on the GPU is not feasible).

We store an atomic counter to point to the back of the already initialized memory pool. During shader execution, we expect the memory pool to be filled about halfway. When closures need to be processed (to either place pixel values into input AOVs or to evaluate a material), and all the closures from the shader have been assigned a location, the renderer will unroll the closure tree that has formed. If the closure pool runs out of space to either allocate memory for a closure at shader execution time or while allocating space for stack traversal of the closure tree per shading point, we notify the renderer via status flags that we need to double the size of the entire closure memory pool and re-run the shader execution for that shader.

4.3. Materials (Bxdfs)

As mentioned in section 3.6, each bxdf is represented by a sample generate and sample evaluate function that can be called by the renderer. The bxdf functions are written as kernels that can be compiled for and executed on CPUs or GPUs.

4.3.1. Monolithic Bxdfs

There are ten stand-alone bxdfs inherited from RenderMan RIS: PxrConstant, PxrDiffuse, PxrDisney, PxrSurface, etc. PxrSurface is a very complex bxdf with several diffuse models, two specular lobes, multiple subsurface scattering types, fuzz, iridescence, glossy refraction, and more; in all, it has 10 lobes and 127 input variables. It has been used for many years at Pixar to represent all material types. Its main limitation is that the layering of the lobes is fixed; for example, fuzz is always on top of diffuse and specular. All the XPU bxdfs have the exact same parameters as in RIS.

4.3.2. MaterialX Lama and MaterialX PBS Nodes

ILM's original implementation of Lama shaders [Pix21] was written in C++ solely for execution on CPUs. We ported the Lama shaders to the XPU framework. This was facilitated by being able to re-use the same header files, classes for scattering distribution functions, etc., so we only had to write a thin layer of sample/evaluate functions on top. One problem we ran into is that Lama has several precomputed tables for microfacet multiscatter energy compensation [Tur17], and those tables were so large that they caused the CUDA compiler to grind to a halt. Fortunately, we were able to reduce the size of the tables — combined with selective remapping of the parameters — without any visible differences.

MaterialX [SS16] is the next generation of shared materials; it is open-source with contributions from many studios and intended to simplify interchange of assets between studios and renderers. ILM is currently implementing all the MaterialX Physically Based Shading base nodes in XPU.

4.3.3. Combiner Nodes

The MaterialX shader framework provides combiner nodes for mixing, adding, and layering base nodes. In RIS, these combiners are relatively straightforward: they have function pointers to their child bxdfs, which they call to sample/evaluate, and then combine the results. However, in XPU, this approach is not feasible due to the complexity of supporting both CPU and GPU execution.

To overcome this, we leveraged XPU's OSL closure support to implement a more flexible, efficient, and general solution. Instead of directly invoking bxdf functions, we defined built-in material closures for each material type (e.g., Lama and MaterialX nodes, and Pixar materials like PxrSurface): at build time, OSL shim nodes are generated alongside these built-in closures for all supported materials. The primary role of these shim nodes is to supply the appropriate parameters to their corresponding material closures. Then, during scene ingestion, when materials are instantiated, the shim nodes are seamlessly integrated into the shading networks. These OSL shading networks are executed as usual, and the resulting material closure DAGs (directed acyclic graphs) are stored in our closure pool (CPU and/or GPU depending on the device).

Finally, material evaluation and sampling follows two key strategies: for bxdf evaluation, we traverse the DAG of closures, accumulating contributions from each bxdf; for bxdf sampling, traversal is performed stochastically, selecting a bxdf to determine the outgoing direction. This approach enables efficient and flexible material evaluation across both CPU and GPU execution paths.

4.4. Texture Caching

The challenge of caching production texture data for rendering is even more pronounced for GPUs, where memory is more limited and expensive. We optimized our implementation of out-of-core texturing for different devices and rendering scenarios. This involved developing distinct texture caching strategies tailored to both the hardware architecture and to the specific types of textures used, such as regular and Ptex textures [BL08]. Our CPU and GPU texture caches differ in several key aspects:

- *Implementation:* The CPU texture cache is implemented in standard C++ and runs entirely on the host. In contrast, the GPU texture cache is implemented using CUDA device code and executes fully on the GPU, making use of CUDA constructs. This allows for efficient parallel processing with minimal host intervention.
- *Page Replacement Strategy:* The CPU cache employs a simple round-robin algorithm for page replacement, while the GPU cache utilizes an LRU (Least Recently Used) strategy to enhance page prediction and improve memory efficiency.
- *Data Granularity and Management:* The CPU cache operates at the tile level and supports tiles of varying sizes. In contrast, the GPU cache subdivides tiles into fixed-size data blocks, referred to as pages. This distinction is significant because using a fixed page size simplifies the parallel processing of pages and facilitates more efficient page replacement.
- *Multi-level Caching Strategy:* The GPU cache utilizes a small additional CPU-side cache to temporarily store tiles for which some, but not all, pages have been requested. Since the remaining pages of these tiles are likely to be accessed in subsequent iterations, retaining them in CPU memory helps minimize redundant disk reads — thus reducing latency and improving overall caching efficiency.
- *Performance Advantage:* In our experiments with production scenes, the GPU texture cache demonstrated up to 4× faster performance compared to the CPU texture cache, benefiting from its parallelized architecture and optimized page management.

The space required to store texture tiles in the cache is currently allocated upfront; this is an area we are working to optimize. However, both texture caches employ lazy allocation for their internal data structures. For the GPU cache, this is a key capability and a notable advantage over the approach proposed by Garanzha et al. [GBPG11], which assumes that the page table and various LRU buffers fit entirely within device memory. In complex production environments, the size of these data structures can easily exceed several gigabytes, making the approach impractical. By using lazy allocation, we have observed memory reductions of up to 80%, depending on the scene, significantly improving scalability without sacrificing performance.

An exception to the system described thus far is that Ptex textures are managed by the CPU texture cache even during GPU rendering. In this case, the texture cache state is transferred from host memory to device memory to ensure the required pages are available when needed. This behavior arises from the interaction of two factors: the fixed page size of the GPU cache and the high variability in tile sizes for Ptex textures. We found that the fixed page size was inefficient for handling the wide range of tile sizes, leading to suboptimal performance of the GPU cache — unless the page size was extremely small. However, using an extremely small page size resulted in an increase in the internal data structures of the GPU cache (the page table and LRU buffers), which introduced additional overhead and negated some of the performance benefits. We also explored using multiple GPU cache instances with different page sizes, but this only resulted in marginal performance gains compared to the simpler, more straightforward CPU cache. This remains an open area of research, and we continue to explore potential future optimizations.

4.5. Interactive Rendering

In an interactive session, artists frequently update the scene with quick material edits, camera pans, lighting modifications, etc. The renderer has to be interrupted on each such edit. The time it takes to register an interrupt, process the changes, and present new results to the screen can have a significant impact on the artist's experience. Our goal is to reduce the lag as much as possible. However, the CUDA GPU programming model employs a fork-join programming model and does not allow for a mid-kernel interruption, which can lead to large unbounded lags hurting the interactive experience. Hence we adopted a "progressive pixels" mode where, in the beginning, results of constrained work elements are splatted to larger regions of the frame buffer (up to a maximum of 16×16 pixels), and as subsequent work items are processed, the splat region sizes are reduced until we reach a single pixel. Only the CPU is employed in these first few iterations of rendering, so that interrupts can be registered and responded to quickly (while the GPU is idle). The working set size for the CPU is gradually increased (using powers of two) as the iterations progress and no interrupts are registered. Only then is the GPU brought in to help process the remaining work. The rationale behind this design is that as more time progresses between interrupts, the chances of registering a new interrupt decreases, and hence we can allow the GPU to start pulling in more work. The low-resolution samples are chosen from their final high-resolution sample pixel locations with high-resolution ray differentials in order not to pollute the texture cache with low-resolution MIP-map levels. We also adopt an update order that recursively applies Bayer ordered dithering to distribute the newest information in areas dominated by the oldest results, which allows the artist to more quickly see the important parts of the image.

4.6. Implementation Details

This section is a collection of implementation details we found interesting and believe could be of interest to others. Many other XPU features are carried over directly from RenderMan RIS and not described here.

4.6.1. Floating-point Differences

Like any renderer running on both CPUs and GPUs, we must deal with slightly inconsistent floating-point results. Some pixel samples can be different due to a different order of floating-point operations, fused multiply-adds, etc. Most of these differences only show up in the least-significant bits of the floating-point values, and are not visible at all. But sometimes a tiny float difference can cause different lobe selection (e.g. reflection instead of refraction), or a different termination choice in Russian roulette — in that case, the pixel sample will have a visibly different color. In practice, a few pixels can differ visibly if "the same" image is rendered multiple times in CPU+GPU mode, but these differences are lost in the noise and disappear with sufficient samples. We take a pragmatic approach to this: both images are equally "correct", just different, and converge to the same result. For the image in Figure 1, the root-mean-square difference between a CPU-only and a GPU-only image is 0.27% and the structured similarity index (ssim) is 99.9978. We are used to such differences from our CPU-only RenderMan RIS renderer: we have seen similar image differences between different CPU vendors and operating systems, and even between debug and optimized builds using the same compiler. These differences have not been objectionable in practice.

4.6.2. OSL Sample and Display Filters

RenderMan allows post-processing workflows on sample and pixel buffers via a plugin infrastructure in the form of sample and display filters. The sample and display filter infrastructure vastly increases the domain of images that can be produced without subsequent compositing treatment; see Figure 7 for a stylized look (non-photoreal) example. In XPU, these filters are implemented in OSL. Sample filters describe transformations made to sample values before they are pixel filtered. They always operate on raw camera samples; their changes to values in the sample buffer become a permanent part of the final render. Operating at sample level offers a unique advantage: the ability to access ray hit information, including geometry primvars and attributes. The downside for this support is that camera ray and hit buffers need to be kept in memory until execution of the sample filter. Display filters are commonly used for color space transformations, to suppress infinite/NaN/negative colors, and for edge detection. They run on accumulated pixel-filtered results, and do not offer any hit information since operations are carried out at a pixel level. Any modifications made to the pixel buffer via display filters are always overwritten at every iteration.

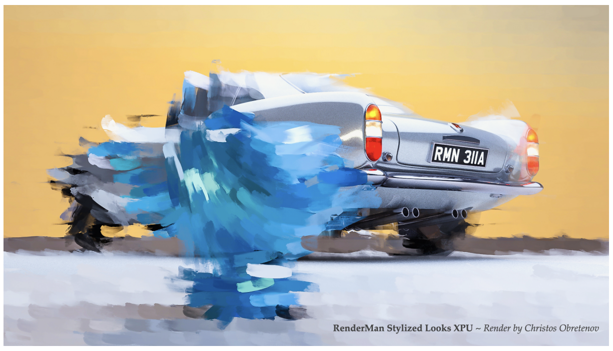

Figure 7: *Stylized image generated using sample and display filters. Rendered by Christos Obretenov. ©Disney/Pixar.*

Reading a pixel or sample value is done via the texture() function, where instead of passing in a filename as input, shader writers pass in AOV names; static AOV names are converted to indices by the renderer to optimize runtime costs. Writing transformed values back is done via closures provided by OSL. Display and sample filters also provide the ability to read a region of pixels in order to infer motion, objects, or edges. This may require looking up regions outside the current bucket (without locking the framebuffer), potentially reading data from a slightly different wavefront iteration.

Daisy-chaining of sample and display shaders is done by executing each shader one after the other. Every shader operates on a bucket of pixels either right before accumulating into the pixel buffer or after. For display filter chaining, XPU keeps an entire copy of the pixel buffer in order to be able to independently call shaders one after the other, instead of asking shader writers to pass the outputs of their shaders on to the next one. Our approach allows users to only worry about their own shader without having to pay attention to what might come before or after it. Due to the double buffering of the pixel buffer, shaders that use inputs from a region of pixels (edge detection filter, for example) also means that the region read from the pixel or sample buffer will always be accumulated pixels rather than the value from the previous shader in the chain. This is because a thread executing a particular shader on a bucket of pixels cannot guarantee that it can read from a pixel value that a different thread might be writing to, causing race conditions and possible garbage values being picked up.

4.6.3. Optimized Subsurface Scattering and Volumes on GPUs

Our XPU implementation of diffusion sss follows the RIS implementation closely. There is only one batch of sss rays, with shading at their hit points, so the execution coherency is good. The XPU implementation of brute-force Monte Carlo path-traced sss is more interesting. Here, a random sss path is followed, tracing one small step at a time, until it hits a surface or a maximum number of steps has been reached (typically 256). The path lengths differ wildly, leading to starvation on GPUs: many cores can sit idle because their path has terminated quickly, while a few cores keep generating new steps and tracing new rays. To ameliorate this problem, we use speculative paths: once less than half the cores are active, we generate multiple steps (scatters) along each path, and intersection test their sss rays together. Once three quarters of the rays have terminated, the remaining rays are speculatively replicated three times and each ray takes four steps, and so on. This gives better utilization of the GPU cores than alternating single trace and scatter steps when the GPU occupancy is thinning out.

We encountered similar issues with tracking methods for volumetric integration. The number of steps taken can vary wildly across rays in a wavefront, again leading to starved GPU cores. Ray-marching with an unbiased technique [KdPN21] can avoid this problem, but we found it less performant than tracking methods on the CPU with our well-tuned octree. We alleviate this problem on the GPU using a similar speculative path extension approach to sss: once half the rays have exited, at the beginning of the next tracking step we synchronize, sort and reorder current work, and duplicate remaining rays and their current tracking state. From this point on, each ray takes two tracking steps through the octree. The duplicated rays are offset by one step from their originals and inherit the same random number context. Similar when only one quarter of the rays remain, etc.

4.6.4. Statistics Output

The RenderMan statistics system underwent a complete overhaul during the early development of XPU. This new system is agnostic to its data source, supporting both RIS and XPU renderers, as well as other sources such as digital content creators (DCCs) and plugins. XPU relies exclusively on this updated system for its diagnostics. Previously, diagnostic data were only accessible through an end-of-render report. However, with interactive rendering becoming a crucial part of the daily workflow, there is now a need for interactive statistics that provide real-time feedback to technical directors and artists as changes are made. This new system is specifically designed for interactivity and extensibility. Instrumentation has been decoupled from analysis and presentation, ensuring minimal impact of instrumentation on the renderer's performance and maximum flexibility of data consumption.

Instrumentation payload data are collected into per-thread buffers, each restricted to a single writer and a single reader. This design allows for an efficient implementation of two types of buffers: a sampled buffer for metrics such as counters, and a reported event buffer. The sampled data buffer is non-atomic, enabling both reading and writing of payloads without the need for locking. This approach may result in slightly out-of-order data updates, which is acceptable given the frequency of updates compared to reads. The event buffer is implemented as an almost lock-free ring buffer. Events are reported by the renderer only if there is an interested listener.

For performance comparison and feature validation, we have a standard set of data collected for XPU that aligns with that of RIS. For example, system memory, time-to-first pixel/iteration, ray counts, and overall render time. XPU-specific stats include BVH build time, texture cache, and more extensive memory tracking. All data is available for extraction at any point during a render, and any subset of data can be gathered for a post-render JSON report.

Qt-based statistics viewing tools present real-time data from an active render, enabling users to track performance and troubleshoot issues on the fly. These live data stream widgets are integrated into each of our DCC bridge plugins, as well as a standalone viewing application.

Instrumentation in RenderMan is always enabled, but the reporting of metric data relies on the presence of diagnostic plugins known as "Listeners". These Listener plugins can collect varying amounts of diagnostic information according to user needs, ranging from a live stream of telemetry data to a post-render report detailing time and memory usage.

5. Results: Performance Evaluation

Most images in this section are rendered on a computer with an AMD Epyc 7763 processor with 31 cores running at 2.4 GHz with 128 GB memory, and an Nvidia RTX-A6000 Ampere graphics card with 10700 CUDA cores and 24 GB memory. Render times labeled 'xpucpu' are rendered using only the CPU cores, times marked 'xpugpu' are rendered only on the GPU, and 'xpu' means rendered on both (the pixels are a mix). All images are rendered with 1024 pixels wide resolution; the shown images are rendered with 'xpu'.

The reported XPU render times are without the initial tessellation and displacement phase. For RIS, this is a bit harder to disentangle since tessellation and displacement are done on-demand during rendering. So for RIS, we subtract the time for the first iteration and report the remaining time as the render time.

5.1. Raw Ray Tracing Speed

As a first test, we measure the "raw" ray tracing speed for a scene with moderate complexity and no bxdf. Woody is modeled with 74 subdivision surfaces with displacement and around 46,000 cubic curves. Figure 8 (left) shows Woody rendered for surface visualization with 256 samples per pixel. Since all rays are camera rays, this is a good test of the performance of coherent rays with very simple shading (mainly computation of smooth normals and a few shading parameters).

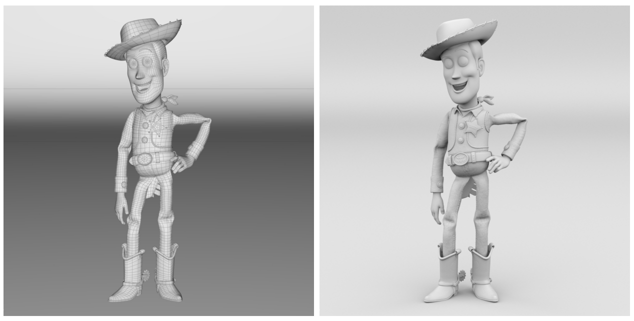

Figure 8: *Woody rendered with surface patches and ambient occlusion. ©Disney/Pixar.*

In the second test, Woody is rendered with ambient occlusion with 32 occlusion rays per camera ray hit point — see Figure 8 (right). This scenario is indicative of ray tracing speed for rays with no shading at all, only hit/miss.

Table 1: *Ray speed (Mrays/sec) and speedups for Woody*

scene	ris	xpucpu	xpugpu	xpu
viz	12	21 ∼ 1.8×	91 ∼ 7.5×	111 ∼ 9.1×
amb occ	21	41 ∼ 2.0×	176 ∼ 8.4×	218 ∼ 10×

The ray tracing speed and speedups for both tests are listed in Table 1. The CPU speedups of 1.8× and 2.0× compared to RIS — running on the exact same hardware — is due to better tuning to the cache hierarchy and improved coherency. The results also show that ray tracing is more than 4 times faster on our GPU than on our CPU for this scene.

5.2. Simple Scenes, Simple Shading

In this section we test different aspects of rendering using four scenes with simple geometry and simple shading. These images are rendered with 256 samples per pixel. The four scenes are shown in Figure 9, and the render times are listed in Table 2.

We start with a Cornell box with two polymesh teapots (one made of chrome, the other glass), rendered with 10 bounces of global illumination. It should be noted that the speedups of 4.6× on CPUs and 14.2× on GPU are higher than usual and not typical.

The second test is a test of hair rendering. The geometry consists of 3128 long curves, shaded with a Chiang hair bxdf [CBTB16]. The hair is rendered with 4 bounces of global illumination. The combined speedup is 15.1×, and the XPU image quality is actually better than RIS (the curves are smoother).

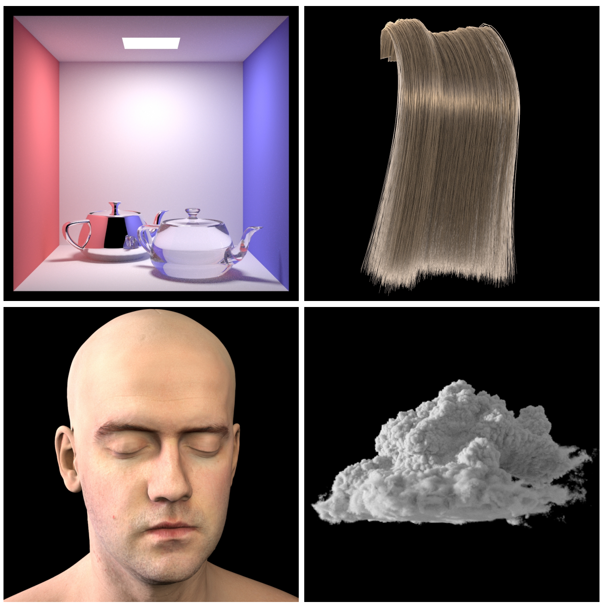

Figure 9: *Simple scenes. Head geometry and textures kindly provided by Infinite Realities. Cloud from the Moana data set [Dis18].*

Table 2: *Render times and speedups for simple scenes*

scene	ris	xpucpu	xpugpu	xpu
box	255s	56s ∼ 4.6×	18s ∼ 14.2×	14s ∼ 18.2×
hair	393s	181s ∼ 2.2×	31s ∼ 12.7×	26s ∼ 15.1×
head	139s	68s ∼ 2.0×	15s ∼ 9.3×	12s ∼ 11.6×
cloud	493s	192s ∼ 2.6×	146s ∼ 3.4×	81s ∼ 6.1×

The next test is subsurface scattering on a human head. The head is a single subdivision surface with displacement and an albedo texture, with direct illumination from two light sources. The head is rendered with brute-force path-traced subsurface scattering using realistic parameters for pale skin; the combined speedup is 11.6×.

The last test in this section is a test of volume rendering: a cloud rendered with 10 bounces of global illumination. The cloud density is defined by a VDB data set. The CPU speedup of 2.6× is good; however, the GPU speedup of 3.4× is a bit disappointing. We hope to further optimize our volume rendering implementation for GPU execution. In the meantime, this provides a good case for combined CPU+GPU rendering.

5.3. Character Shading Tests

Now let's look at shading tests for five characters from the *Toy Story* movies. They are illuminated by a dome light source and have 4 bounces of global illumination. The images in Figure 10 are rendered with 64 samples per pixel; images like these are often used for texturing and material approvals and feedback.

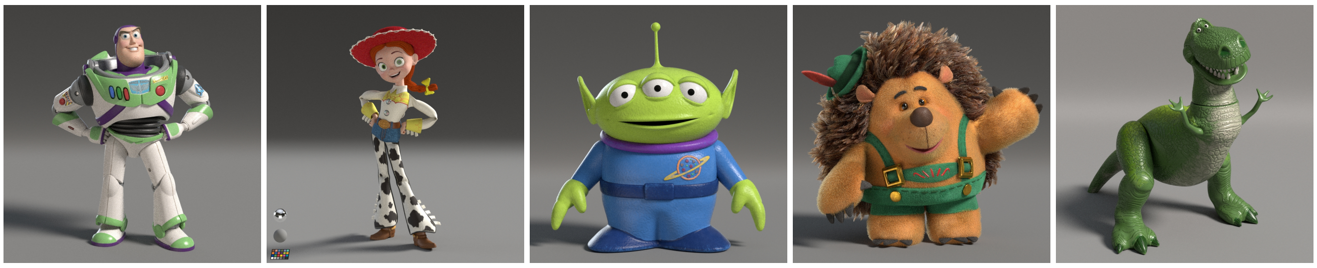

Figure 10: *Five* Toy Story *characters. ©Disney/Pixar.*

Table 3 shows the render times and speedups for rendering these images with RIS and the three different variants of XPU. In summary, the speedups are 1.8-2.3× for CPU, 4.9-9.7× for GPU, and 5.3-11.3× for combined CPU+GPU.

Table 3: *Render times and speedups for toys*

character	ris	xpucpu	xpugpu	xpu
Buzz	78s	34s ∼ 2.3×	11s ∼ 7.1×	10s ∼ 7.8×
Jessie	79s	42s ∼ 1.9×	16s ∼ 4.9×	15s ∼ 5.3×
Alien	139s	67s ∼ 2.1×	17s ∼ 8.2×	17s ∼ 8.2×
P.pants	224s	122s ∼ 1.8×	23s ∼ 9.7×	21s ∼ 10.7×
Rex	68s	36s ∼ 1.9×	7s ∼ 9.7×	6s ∼ 11.3×

5.4. A Complex Production Scene

In this section we test a more complex scene: Bonnie's room with furniture, 13 toys, complex shading networks, many textures, 15 light sources, subsurface scattering, and 4 bounces of global illumination. This scene renders using 14.3 GB peak memory in RIS, 15.4 GB for CPU, and 16.8 GB for GPU. Render times and speedups for various sample counts are listed in Table 4, and the image rendered with 1024 samples per pixel is shown in Figure 1.

Table 4: *Render times and speedups for room*

spp	ris	xpucpu	xpugpu	xpu
4	25s	7.9s ∼ 3.2×	3.4s ∼ 7.4×	3.3s ∼ 7.6×
16	99s	39s ∼ 2.5×	14s ∼ 7.1×	12s ∼ 8.3×
64	379s	164s ∼ 2.3×	53s ∼ 7.2×	44s ∼ 8.6×
256	1571s	653s ∼ 2.4×	198s ∼ 7.9×	161s ∼ 9.8×
1024	6115s	2630s ∼ 2.3×	768s ∼ 8.0×	615s ∼ 9.9×

5.5. Scaling on More Powerful Hardware

We also tested Bonnie's room on a more powerful computer with two AMD Epyc 9654 CPUs with a total of 192 cores and hyperthreading, running at 2.4 GHz with 527 GB total memory.

For RIS, the speedups are good with 192 threads (4.8× faster than our base machine with 31 cores), but rendering actually gets slower with more than 200 threads, indicating poor hyperthreading utilization at very high thread counts.

With XPU running on both CPUs, we measured a speedup of 8.5× relative to our base machine: each core is roughly 20% faster, there are about 6 times more cores, and hyperthreading ekes out another 20% speedup. We consider that scaling result to be very satisfying. The same machine also has two Nvidia RTX-6000 Ada cards with 18200 CUDA cores and 48 GB each. With both GPU cards, the speedup relative to our base machine (with one Ampere card) is 3.0×. CPU rendering on 192 cores is almost as fast as two GPUs.

5.6. Interactive Rendering

RenderMan XPU can run interactively inside Maya, Katana, and Houdini. Figure 11 shows an interactive session in the Katana viewport. This is a snapshot at 4 samples per pixel, with the interactive denoiser [VRM*18; ZMV*21] enabled. Note that even with only 4 samples per pixel, the quality is good enough that a TD can make informed design decisions regarding geometry, shading, textures, and lighting. The render time — including writing of 32 output AOV channels, running multiple pixel and display filters, and interactive denoising — is 5 seconds with XPU.

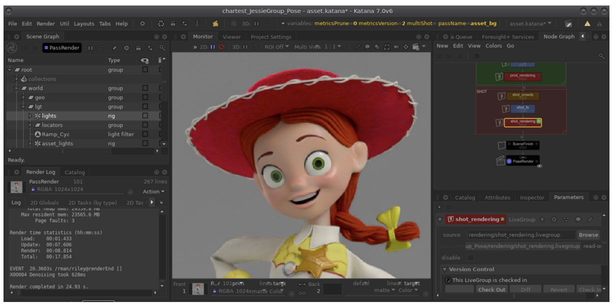

Figure 11: *Jessie rendered inside the Katana viewport. ©Disney/Pixar.*

The supplemental material contains a recording of a session with XPU in Flow, interactively rendering Jessie with camera tumble, material selection, and parameter changes.

6. Discussion and Future Work

We believe these results validate our design decisions for RenderMan XPU. The combination of CPU and GPU power utilizes all the available resources on modern computers.

As expected, rendering on a GPU is typically several times faster than on a CPU. So why bother with CPU rendering at all? 1) Most production rendering farms consist primarily of CPUs. 2) All the scenes shown here fit within the GPU memory of a typical artists' workstation, but larger scenes can be rendered purely on the CPU without changes to the rendered image. 3) The extra boost from adding a CPU varies from nearly nothing when GPU performance is dominant (the alien) to almost twice the speed (the Moana cloud). We like that speedup when it is available. 4) CPUs are more nimble than GPUs when shaders need to be recompiled during an interactive session.

The most important missing feature in XPU is the ability to efficiently handle hundreds or thousands of lights. In RIS, we have fairly sophisticated light clustering, estimation, and selection algorithms, and we need to implement something similar in XPU. Other features from RIS that we would like to re-implement are bidirectional path tracing, VCM, and manifold walking.

There are many avenues for further speedups. We hope that with all our XPU shading code written with SIMD in mind, using vectorization and wider vector instructions will give a further 2.0× to 2.5× speedup for shading on CPUs. Implementing tessellation and displacement shading on GPUs should give a good speedup of the pre-processing stage. As mentioned earlier, we would also like to further optimize volume rendering on GPUs.

7. Conclusion

RenderMan XPU is a single renderer with a dual purpose: interactive rendering for fast feedback, and off-line rendering of movie-quality final frames. It utilizes heterogeneous hardware for optimal use of available compute power and memory capacity, within tools and workflows that artists are used to. It has a modular and flexible architecture, and source code is shared as much as possible across different types of hardware.

Writing a new version of RenderMan has been — and still is — a very large project. Taking RenderMan XPU from a simple proof-of-concept to a fully featured production renderer has only been possible through the effort of a dedicated team over many years. RenderMan XPU is still a work in progress, but we are excited that artists have started using it in production. We feel that we are very close to reaching feature parity with RenderMan RIS, with significant improvements in rendering speed.

Acknowledgments

We would like to thank everyone involved in the development and testing of RenderMan XPU. Huge thanks to former RenderMan XPU core developers Marc Bannister, Max Liani, and Cliff Ramshaw for their invaluable contributions. Also thanks to Jonathan Shade, Andrew Kensler, Lucy Wilkes, and Sarah Invernizzi for their core contributions. RenderMan would be useless without our fabulous bridge and support team: Katrin Bratland, Ian Hsieh, Sarah Hutchinson, Sarah Forcier, Eugene Riecansky, Damien Maupu, Peter Ellerington, and Christos Obretenov. Thanks also to our sales and QA team: Dylan Sisson, Wendy Wirthlin, Leif Pedersen, and Katya Ferguson. And finally, many thanks to the RenderMan management team: Mark Manca, Adrian Bell, David Laur, Oliver Meiseberg, and Steve May; Mark has been particularly helpful with test scenes and use cases. We would also like to thank our former manager Allan Poore for greenlighting the XPU project. We are also grateful for help, support, and inspiration from outside the RenderMan team. Pixar's RTP effort was lead by Mark Meyer, Danny Nahmias, and Florian Hecht. Studio testers for *Toy Story 5* include Bill Reeves, Jean-Claude Kalache, Athena Xenakis, Michael Kilgore, Jeremy Heintz, Chris Burrows, Gus Dizon, and Renee Tam. Gus recorded the interactive session in the supplemental material. Rick Sayre provided insightful comments. Also many thanks to our external testers and users at studios big and small for providing feedback, suggestions, and bug reports. Intel developers Alex Wells and Charles Congdon help analyze and optimize our RenderMan XPU core code for multiple platforms. Nvidia's Eric Enderton and David Hart have provided much guidance for GPU specific issues. ILM provided the Lama source code for RIS that we have adapted for XPU, and ILM's Jonathan Stone is writing MaterialX shaders for XPU. Disney Research developed the denoiser we use for both interactive and final-frame denoising — special thanks to Gerhard Röthlin.

References

[AG00] Apodaca, Anthony and Gritz, Larry. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000 2.

[AK90] Arvo, James and Kirk, David. "Particle transport and image synthesis". *Computer Graphics (Proc. SIGGRAPH)* 24.4 (1990), 63–66 6.

[BAC*18] Burley, Brent, Adler, David, Chiang, Matt Jen-Yuan, et al. "The design and evolution of Disney's Hyperion renderer". *ACM Transactions on Graphics* 37.3 (2018) 3.

[BL08] Burley, Brent and Lacewell, Dylan. "Ptex: per-face texture mapping for production rendering". *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering)* 27.4 (2008), 1155–1164 11.

[CBTB16] Chiang, Matt Jen-Yuan, Bitterli, Benedikt, Tappan, Chuck, and Burley, Brent. "A practical and controllable hair and fur model for production rendering". *Computer Graphics Forum (Proc. Eurographics)* 35.2 (2016), 275–283 8, 14.

[CC78] Catmull, Edwin and Clark, James. "Recursively generated B-spline surfaces on arbitrary topological meshes". *Computer Aided Design* 10.6 (1978), 350–355 5.

[CCC87] Cook, Robert, Carpenter, Loren, and Catmull, Edwin. "The Reyes image rendering architecture". *Computer Graphics (Proc. SIGGRAPH)* 21.4 (1987), 95–102 2.

[CDE*14] Cigolle, Zina, Donow, Sam, Evangelakos, Daniel, et al. "Survey of efficient representations for independent unit vectors". *Journal of Computer Graphics Techniques* 3.2 (2014) 8.

[CFLB06] Christensen, Per, Fong, Julian, Laur, David, and Batali, Dana. "Ray tracing for the movie 'Cars'". *Proc. IEEE Symposium on Interactive Ray Tracing*. 2006, 1–6 2, 7.

[CFS*18] Christensen, Per, Fong, Julian, Shade, Jonathan, et al. "RenderMan: an advanced path tracing architecture for movie rendering". *ACM Transactions on Graphics* 37.3 (2018) 2, 6.

[Chr08] Christensen, Per. *Point-based approximate color bleeding*. Tech. rep. 08-01. Pixar Animation Studios, 2008 2.

[Dis18] Disney Animation. *Moana cloud data set*. 2018. URL: https://www.disneyanimation.com/data-sets/ 14.

[FFB*09] Fisher, Matthew, Fatahalian, Kayvon, Boulos, Solomon, et al. "DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering". *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* 28.5 (2009) 7.

[FHL*18] Fascione, Luca, Hanika, Johannes, Leone, Mark, et al. "Manuka: A batch-shading architecture for spectral path tracing in movie production". *ACM Transactions on Graphics* 37.3 (2018) 3.

[FHWK17] Fong, Julian, Habel, Ralf, Wrenninge, Magnus, and Kulla, Christopher. "Production Volume Rendering". *SIGGRAPH Courses*. 2017 9.

[Fon23] Fong, Julian. "Volume rendering for Pixar's Elemental". *SIGGRAPH Tech Talks*. 2023 9.

[GBPG11] Garanzha, Kirill, Bely, Alexander, Premoze, Simon, and Galaktionov, Vladimir. "Out-of-core GPU ray tracing of complex scenes". *SIGGRAPH Tech Talks*. 2011 2, 11.

[GIF*18] Georgiev, Iliyan, Ize, Thiago, Farnsworth, Mike, et al. "Arnold: A brute-force production path tracer". *ACM Transactions on Graphics* 37.3 (2018) 3.

[GSKC10] Gritz, Larry, Stein, Clifford, Kulla, Chris, and Conty, Alejandro. "Open Shading Language". *SIGGRAPH Tech Talks*. 2010 2, 9.

[Kaj86] Kajiya, Jim. "The rendering equation". *Computer Graphics (Proc. SIGGRAPH)* 20.4 (1986), 143–150 6.

[KdPN21] Kettunen, Markus, d'Eon, Eugene, Pantaleoni, Jacopo, and Novák, Jan. "An unbiased ray-marching transmittance estimator". *ACM Transactions on Graphics (Proc. SIGGRAPH)* 40.4 (Aug. 2021) 13.

[KWR*17] Keller, Alexander, Wächter, Carsten, Raab, Matthias, et al. *The Iray light transport simulation and rendering system*. Tech. rep. Nvidia, 2017 3.

[LA04] Lattner, Chris and Adve, Vikram. "LLVM: A compilation framework for lifelong program analysis and transformation". *Proc. International Symposium on Code Generation and Optimization (CGO)*. 2004, 75–88 9.

[LGXT17] Lee, Mark, Green, Brian, Xie, Feng, and Tabellion, Eric. "Vectorized production path tracing". *Proc. High Performance Graphics*. 2017 3.

[LKA13] Laine, Samuli, Karras, Tero, and Aila, Timo. "Megakernels considered harmful: wavefront path tracing on GPUs". *Proc. High Performance Graphics*. 2013, 137–143 4.

[Mus13] Museth, Ken. "VDB: high-resolution sparse volumes with dynamic topology". *ACM Transactions on Graphics* 32.3 (2013) 9.

[Mus21] Museth, Ken. "NanoVDB: a GPU-friendly and portable VDB data structure for real-time rendering and simulation". *SIGGRAPH Tech Talks*. 2021 9.

[Nah13] Nahmias, Jean-Daniel. *Using Nvidia OptiX for lighting preview in a Katana-based production pipeline*. Tech talk at Nvidia Visual Computing Theater at SIGGRAPH. 2013. URL: https://www.youtube.com/watch?v=LACmRpMYOak&t=7s 3.

[NO02] Nakamaru, Koji and Ohno, Yoshio. "Ray tracing for curves primitive". *Journal of WSCG* 10 (2002), 311–316 8.

[NSJ14] Novák, Jan, Selle, Andrew, and Jarosz, Wojciech. "Residual ratio tracking for estimating attenuation in participating media". *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* 33.6 (2014) 9.

[PBD*10] Parker, Steven, Bigler, James, Dietrich, Andreas, et al. "OptiX: a general purpose ray tracing engine". *ACM Transactions on Graphics (Proc. SIGGRAPH)* 29.4 (2010) 2, 3.

[PFAH10] Pantaleoni, Jacopo, Fascione, Luca, Aila, Timo, and Hill, Martin. "PantaRay: fast ray-traced occlusion caching of massive scenes". *ACM Transactions on Graphics (Proc. SIGGRAPH)* 29.4 (2010) 3.

[Pix21] Pixar. *MaterialX Lama*. 2021. URL: https://rmanwiki-26.pixar.com/space/REN26/19661457/MaterialX+Lama 11.

[Pix23] Pixar. *OpenSubdiv*. Pixar, 2023. URL: https://graphics.pixar.com/opensubdiv/docs/intro.html 7.

[PJH23] Pharr, Matt, Jakob, Wenzel, and Humphreys, Greg. *Physically Based Rendering: From Theory to Implementation*. 4th edition. MIT Press, 2023 3, 4, 6.

[PVL*05] Pellacini, Fabio, Vidimče, Kiril, Lefohn, Aaron, et al. "Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography". *ACM Transactions on Graphics (Proc. SIGGRAPH)* 24.3 (2005), 464–470 3.

[RKS*07] Ragan-Kelley, Jonathan, Kilpatrick, Charlie, Smith, Brian, et al. "The Lightspeed automatic interactive lighting preview system". *ACM Transactions on Graphics (Proc. SIGGRAPH)* 26.3 (2007) 3.

[SHE*24] Stein, Clifford, Hellmuth, Chris, Estevez, Alejandro Conty, et al. "Spear: across the streaming multiprocessors — porting a production renderer to the GPU". *Proc. Digital Production Symposium (DigiPro)*. 2024 3, 10.

[Sid25] SideFX. *Karma XPU*. 2025. URL: https://www.sidefx.com/docs/houdini/solaris/karma_xpu.html 3.

[SS16] Smythe, Doug and Stone, Jonathan. *MaterialX: an open standard for network-based CG object looks*. 2016. URL: https://www.materialx.org 11.

[TCE05] Talbot, Justin, Cline, David, and Egbert, Parris. "Importance resampling for global illumination". *Proc. Eurographics Symposium on Rendering*. 2005, 139–146 6.

[Tur17] Turquin, Emmanuel. *Practical multiple scattering compensation for microfacet models*. Tech. rep. Industrial Light & Magic, 2017 11.

[Ups90] Upstill, Steve. *The RenderMan Companion*. Addison Wesley, 1990 2.

[VG95] Veach, Eric and Guibas, Leonidas. "Optimally combining sampling techniques for Monte Carlo rendering". *Computer Graphics (Proc. SIGGRAPH)* (1995), 419–428 6.

[VRM*18] Vogels, Thijs, Rousselle, Fabrice, Mcwilliams, Brian, et al. "Denoising with kernel prediction and asymmetric loss functions". *ACM Transactions on Graphics (Proc. SIGGRAPH)* 37.4 (2018) 15.

[WBW*14] Woop, Sven, Benthin, Carsten, Wald, Ingo, et al. "Exploiting local orientation similarity for efficient ray traversal of hair and fur". *Proc. High Performance Graphics*. 2014, 41–49 8.

[WGER05] Wexler, Daniel, Gritz, Larry, Enderton, Eric, and Rice, Jonathan. "GPU-accelerated high-quality hidden surface removal". *Proc. Graphics Hardware*. 2005, 7–14 3.

[WMHL65] Woodcock, E., Murphy, T., Hemmings, P., and Longworth, T. "Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry". *Applications of Computing Methods to Reactor Problems*. Argonne National Laboratory. 1965 9.

[WWB*14] Wald, Ingo, Woop, Sven, Benthin, Carsten, et al. "Embree: a kernel framework for efficient CPU ray tracing". *ACM Transactions on Graphics (Proc. SIGGRAPH)* 33.4 (2014) 2, 3.

[ZMV*21] Zhang, Xianyao, Manzi, Marco, Vogels, Thijs, et al. "Deep compositional denoising for high-quality Monte Carlo rendering". *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering)* 40.4 (2021) 15.