

Digesting the Elephant — Experiences with Interactive Production Quality Path Tracing of the Moana Island Scene

Ingo Wald^{1,2} Bruce Cherniak¹ Will Usher^{3,1} Carson Brownlee¹ Attila Áfra¹ Johannes Günther¹ Jefferson Amstutz¹
 Tim Rowley¹ Valerio Pascucci³ Chris R. Johnson³ Jim Jeffers¹

(¹)Intel Corp (²)now at NVIDIA (³)SCI Institute, University of Utah

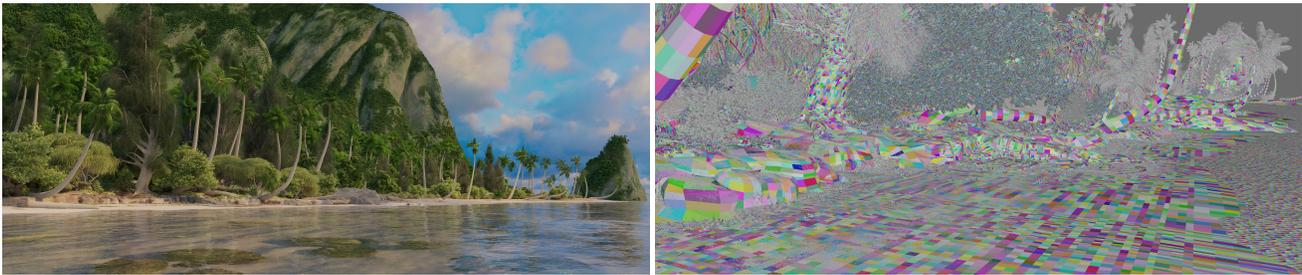


Fig. 1. Left: An overview of the Moana Island Scene (39.3 million instances, 261.1 million unique quads, and 82.4 billion instanced quads). Right: A close-up of the beach, with each primitive colored individually to try to convey the detail of this model. In this paper, we describe the steps taken to allow us to render this model—in its entirety, without simplification, and with a high-quality path tracer—at interactive frame rates.

Abstract

New algorithmic and hardware developments over the past two decades have enabled interactive ray tracing of small to modest sized scenes, and are finding growing popularity in scientific visualization and games. However, interactive ray tracing has not been as widely explored in the context of production film rendering, where challenges due to the complexity of the models and, from a practical standpoint, their unavailability to the wider research community, have posed significant challenges. The recent release of the Disney Moana Island Scene has made one such model available to the community for experimentation. In this paper, we detail the challenges posed by this scene to an interactive ray tracer, and the solutions we have employed and developed to enable interactive path tracing of the scene with full geometric and shading detail, with the goal of providing insight and guidance to other researchers.

1 INTRODUCTION

Over the past two decades rendering technology has seen an astonishingly fast shift towards ray tracing. While only a short time ago production movie rendering predominantly used micro-polygon rasterization, specifically Reyes [Cook et al. 1987], today virtually all production renderers use some form of ray tracing [Burley et al. 2018; Christensen et al. 2018; Fascione et al. 2018; Georgiev et al. 2018; Keller et al. 2015; Kulla et al. 2018; Pharr 2018a]. In the past few years even real-time rendering applications have begun to incorporate ray tracing effects [4A Games 2019; Electronic Arts 2019a,b]. This move toward ray tracing is driven by a number of factors: ray tracing supports physically based lighting effects, which improves artist usability and removes the need for approximations; scales well with scene complexity; and maps well to thread parallelism. With regard to the last two factors, significant improvements have been made over the past two decades in both software and hardware to enable real-time ray tracing.

The push towards interactive ray tracing is most visible today in scientific visualization applications [Parker et al. 2010; Wald et al. 2017] and games [4A Games 2019; Electronic Arts 2019a,b], where compelling ray traced images can be rendered at moderate to real-time framerates on current CPUs and GPUs, depending on the model size and shading complexity. However, production movie rendering has largely remained an offline process, taking minutes to hours per-frame. Artist tools typically use rasterization for interactive rendering, resulting in a disconnect between the lighting and material models used in the modeling tools and the final renders. This disparity in ray tracing performance is a result of the radically different rendering demands of interactive applications and film. Interactive applications are driven by hard real-time requirements, and leverage a range of approximations or simplified models to produce convincing imagery with just a few samples per-pixel; however, production renderers are driven by quality demands and prefer generality and true realism, even if these require hundreds to thousands of samples per pixel.

The demands placed on the ray tracer further diverge when considering the content each is tasked with rendering. While game assets are modeled with a specific real-time frame budget in mind, this is not the case for film, where assets are re-used to save artist time or always modeled at high-quality. For example, if a scene in a movie needs some shells on a beach and a shell model is available, it is easiest to re-use this asset, even if it was originally created for a close-up shot and consists millions of triangles and high-resolution textures, while the new use case may be for hundreds to thousands of sub-pixel objects. Similarly, assets tend to be modeled at high-quality regardless of their initial planned size on screen, as the shot may change to include a close-up of the asset.

As a result, movie content is often detailed to a degree that those outside of film may view as extreme, with many thousands of triangles projecting to each pixel [Fascione et al. 2018], tens to hundreds

of gigabytes of textures, and orders of magnitude more *instances* than a typical game has *triangles*. To quote Matt Pharr’s experience [Pharr 2018b] in getting PBRT to render Disney’s Moana Island Scene, dealing with such content is akin to “swallowing an elephant”, a surprisingly fitting description.

In this paper we follow Pharr’s example, but with the explicit goal of enabling *interactive* rendering, without compromises in model complexity or shading detail. We describe our experience in both “swallowing the elephant”, i.e., loading and rendering it at all; and “digesting” it, i.e., rendering it at interactive rates. We detail the challenges faced in terms of geometric variety and complexity, model size, texturing and shading complexity, and performance; and how we tackled these challenges to enable interactive rendering. Specifically, this paper aims to:

- Detail the challenges posed by production film assets to an interactive ray tracer;
- Present our solutions for tackling these challenges to enable interactive rendering;
- Summarize what is possible today for interactive rendering of production assets, and briefly discuss future challenges.

2 PAPER OVERVIEW

The goal of this paper is to detail the challenges encountered and solutions developed in our efforts to enable interactive rendering of the Moana Island Scene—exactly as provided by Disney, without simplification and with production-style path traced image quality.

In Section 3 we give a brief overview of related work in production rendering and to the Moana Island Scene, but defer specific related work discussion to the relevant technical subsections throughout Section 5. To properly motivate the trade-offs and design decisions made in our approach, it is imperative to first convey the challenges posed by the Moana Island Scene. Though the amount of detail in a modern movie shot is clear even to those outside of graphics, few researchers outside of the studios have had the chance to work with production content. As such, the exact kind—and in particular scale—of the challenges posed by production content is often not well understood in the broader research community. To properly convey these challenges we spend Section 4 describing the scene, and the variety and scale of challenges it poses to a ray tracer. Though other production content may pose a somewhat different set of challenges, the Moana Island Scene provides a good proxy for other such content.

In Section 5 we then describe our approach to handling these challenges, and discuss the individual components of our final system, with a focus on those which did not work out of the box, or had to be added specifically for this work. In Section 6 we present performance results of our final system, and end with a brief discussion and summary (Section 7).

3 RELATED WORK

Early versions of the Moana Island Scene were made available to researchers as early as late 2017, and the first public version released in mid 2018. An illuminating summary of some of the challenges in dealing with the Moana Island Scene is available online through Matt Pharr’s series of blog posts [Pharr 2018b] on “Swallowing the

Elephant”, which discusses the challenges encountered in getting the model loaded and rendered offline using PBRT [Pharr et al. 2016].

In this paper we give a further in-depth discussion of the model and the challenges it poses to a ray tracer, with a focus on interactive ray tracing. We build our interactive renderer on top of Embree [Wald et al. 2014] and OSPRay [Wald et al. 2017], for ray traversal and rendering, along with Disney Animation’s *Ptex* library [Burley and Laceywell 2008] and *principled* BSDF [Burley 2012, 2015], for texturing and shading.

Though the general inaccessibility of production assets outside of the movie studios means that dealing with such assets is largely uncharted territory for interactive rendering research, production renderers deal with such content on a daily basis. An excellent survey of production renderers was recently presented in the ACM Transactions on Graphics Special Issue on Production Rendering [Bala 2018], which includes in-depth descriptions of some of today’s most prominent production renderers: Autodesk’s Arnold [Georgiev et al. 2018] and Sony’s Arnold [Kulla et al. 2018], Disney’s Hyperion [Burley et al. 2018], Weta’s Manuka [Fascione et al. 2018], and Pixar’s RenderMan [Christensen et al. 2018]. Many of these production renderers have added some support for interactive model rendering (and in some cases, editing) over the past few years, typically through some form of progressive re-rendering of lower resolution images when a part of the scene or viewpoint changes.

4 THE CHALLENGE: THE MOANA ISLAND SCENE

The Moana Island Scene was publicly released in June 2018, and comes with an extensive whitepaper describing the asset [Tamstorf and Pritchett 2018], which mentions that the publicly released version is only an approximation of the original content (e.g., subdivision surfaces are represented by their base cages). The asset comes in two separate compressed archives, together comprising a total of 51 GB of compressed data (134 GB uncompressed). An additional archive specifying the animation data is also provided, containing 24 GB of compressed data (131 GB uncompressed).

Even without animation data the model is 134 GBs, split across roughly three components: a version of the model in a proprietary JSON-encoding, a version of the model converted to PBRT format, and textures which are shared between both versions. The amount of data needed to render the scene is less than the total 134 GBs, as only one of the JSON or PBRT versions are needed.

The scene contains 3749 Ptex textures, comprising a total of 41 GBs of texture data. Of these textures roughly 4% are used for displacement mapping, with the majority used for color mapping.

The JSON version of the scene contains 20 GB of JSON data across 165 files which specify the shading information and scene hierarchy, along with an additional 11 GB of Wavefront OBJ files, which specify the polygon meshes referenced by the JSON files. Virtually all polygons are quads, and correspond to the subdivision base cages used in the original scene.

The PBRT export of the scene contains 39 GB of data across 495 files. There are a variety of differences between the PBRT and JSON versions of the model, e.g., quad meshes are converted to PBRT triangle meshes, materials are slightly different, etc., (for more detail, see [Tamstorf and Pritchett 2018]); however, the overall

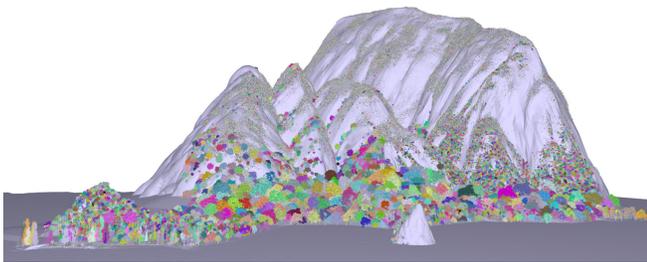


Fig. 2. An overview over the Moana Island Scene, with a pseudo-color “instance ID” shader to convey the large number of instances. Different object colors indicate different instances.

scene structure remains the same. For the remainder of this section we will refer to the PBRT version of the model; the JSON version should produce similar numbers, with the exception that every pair of triangles in the PBRT files corresponds to a single quad in the JSON’s OBJ files.

4.1 Statistical Data

In terms of shading data, there are 95 different PBRT materials in the scene, all using Disney’s principled BSDF [Burley 2015]. For many of these materials the diffuse component is modulated by the underlying shape’s Ptex texture. Finally, there is one textured environment light source and 23 quad-shaped key lights, which for a production asset is a rather modest number.

As for geometry, there are a total of 278 unique PBRT “objects” (i.e., before instantiation), with a total of 1.9 M unique PBRT “shapes” (mostly triangle meshes) which contain a total of 146 M unique triangles. After instantiation these correspond to 39 M instanced objects with 106 M shapes, containing 10 M curves and 164 billion instanced triangles. There are an additional 375 K “ribbon” curves (some round, some flat) with roughly 10 M cubic curve segments. These curves are primarily used for grass and some palm fronds in the scene. To emphasize the scale of the geometry in the scene, the Moana Island contains more geometry *instances* (over 100 M) than most scenes used in ray tracing research contain in final polygons (also see Figure 2).

4.2 Beyond Statistics

While the statistics behind the scene are already impressive, raw statistical information does not convey the true challenges posed by how this translates to the scene geometry. An adequate description is best provided through visual exploration in an interactive session (see supplemental video on YouTube ¹), and in text form will necessarily fall short. However, to at least convey some of these challenges: a frequently occurring feature in the scene are significantly overlapping or intersecting geometric shapes, either resulting from two physical copies of the same logical asset (in particular, trees), or finer detail levels added on top of coarser base geometry. The latter case is encountered on the water and island surface geometry, where a finer top surface was placed on top of coarser base one.

Another common feature in the scene are large variations in the tessellation density of nearby objects (Figure 3), either due to

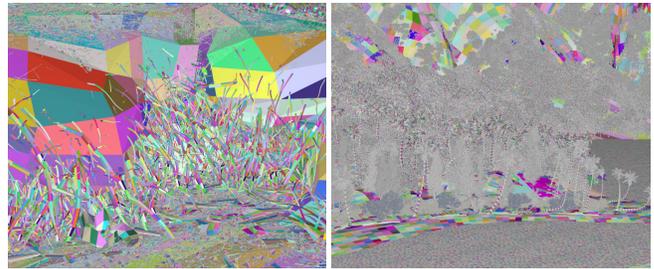


Fig. 3. Near and far views of the Moana Island, with a pseudo-color “primitive ID” shader to convey the highly varying geometric density. Left: Detailed individual twig, pebble and seed models sit on the coarser island terrain mesh. Right: The ocean surface near the land is highly tessellated, with the beach covered with the sub-pixel twigs, pebbles and seeds seen in the left.

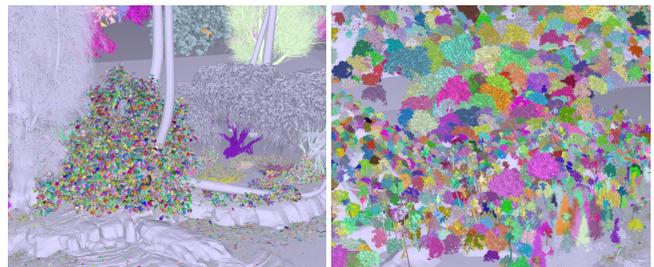


Fig. 4. The size of instances and large amount of overlap among them pose significant challenges to the ray tracer’s top-level BVH. Left: Individual leaves of some bushes are instantiated, and overlap significantly with each other and geometry from other instances. Right: The large number of trees instantiated through the scene overlap significantly.

the relative sizes of the objects, e.g., 1k+ triangle twigs on a beach tessellated according to a mile-sized island, or due to some view- and curvature-adaptive tessellation of the water. The water surface alone is a few million triangles, with some areas using almost millimeter-scale tessellation. Some of these scale differences come from instantiation (e.g., twigs, grains of sand), while some (e.g., the water) are in the base geometry. Large and tiny geometric primitives often overlap and, combined with the large number of long, thin polygons (e.g., branches, roots), this poses a significant challenge to the ray tracer’s acceleration structure.

The distribution of instances in the scene poses a similarly challenging situation. Some instances are large, both physically and in number of primitives, such as some trees which consist of several million triangles. As the entire geometry for the tree is in a single BVH, these are simple cases for the ray tracer. However, other instances are tiny, consisting of a single leaf or flower bud made of a few dozen triangles. These small instances are then used thousands of times, e.g., to place leaves on a bush, and significantly overlap each other and other scene geometry (see Figure 4). Although each instance is small, the extreme amount of overlap effectively disables the top-level BVH’s ability to separate these objects. Moreover, many instanced objects in the scene (e.g., trees) contain a large amount of empty space, thus there is a high chance they will have to be traversed by the ray tracer, but a low chance of intersecting them, leading to a significant increase in traversal cost (Figure 4).

¹<https://youtu.be/JHyC7DE3mJ4>

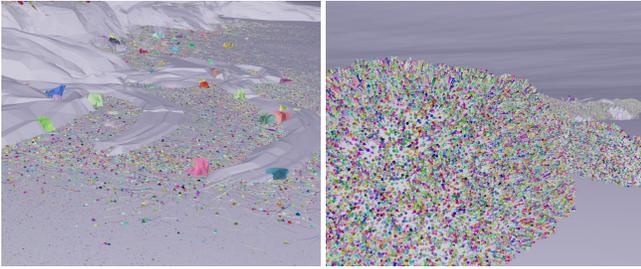


Fig. 5. Pebbles, twigs, and flowers (left); and coral antlers (right); are modeled at high detail and instanced, creating a massive amount of detailed sub-pixel and off-camera geometry.

Beyond the challenges the scene poses to a ray tracer, the overall level of geometric detail in the model is difficult to convey. For example, there are millions of object instances modeled at high detail which for anything other than a close-up shot will be sub-pixel, off-camera, or occluded in the final frame (see Figure 5). For those outside of production rendering it may be tempting to dismiss such data as “extreme” or “overmodeled”. In practice, such assets are a natural consequence of a workflow which prioritizes artist time, asset re-use, and realism, where scenes are assembled from many smaller high-quality assets and through content generation tools such as Disney’s Bonsai [Keim et al. 2016].

5 DIGESTING THE ELEPHANT

From the beginning, our goal was to enable interactive visual exploration of the Moana Island Scene described in the previous section, at full geometric and shading quality. Though we were initially confident that Embree [Wald et al. 2014] and OSPRay [Wald et al. 2017] could handle such content out of the box, we found this to not always be the case. The challenges we encountered in this work can be roughly grouped into the following five categories: data wrangling (i.e., dealing with dozens of gigabytes of input data); geometric complexity and variety (i.e., non-triangular primitives); the use of Ptex textures throughout the model; OSPRay’s path tracer’s inability to handle the model’s shading demands (i.e., the principled material); and the need to achieve interactive performance.

5.1 Data Wrangling

An often overlooked but crucial component when working with large scenes is the ability to load the data at all, in a reasonable time and memory budget. While not an issue for smaller scenes, in the case of a production scene at the scale of the Moana Island, just loading the data poses a real challenge. An excellent discussion of some of these challenges can also be found in Matt Pharr’s “Swallowing the Elephant” series of blog posts [Pharr 2018b], which covers the challenges of loading the Moana Island Scene into PBRT.

Given both the JSON and PBRT versions of the asset², we conducted some early experiments using our open-source PBRT parser, though quickly ran into its limitations. Attempts to load the original JSON format directly proved similarly challenging. Though

²We were graciously provided some early versions before the public release, the latest release of the asset also includes scripts to export an Embree XML version.

the whitepaper [Tamstorf and Pritchett 2018] provides some documentation about the JSON format, the nature of JSON as a purely syntactical encoding makes it challenging to identify semantic relationships between entities across the large number of files and directories. This effort was further complicated by the fact that there appears to be a form of multi-level instancing used in the JSON version, though it was not always clear which directories used instances in which way. As just reading the 20 GBs of JSON data could take minutes, with little to verify the parser’s output, debugging the JSON parser turned into a major issue.

Thus we returned to the PBRT version, and developed a working PBRT parser using the public PBRT v3 scenes for verification. Armed with this parser we returned to the Moana Island Scene, and began testing on subsets of the scene by manually editing the root `island.pbrt` to remove objects the parser initially did not support (e.g., curves) as they were gradually added in.

While our PBRT parser was now able to load the scene, parsing larger and larger subsets of the scene quickly led to the parsing time becoming a huge bottleneck. As PBRT is an ASCII format, a significant amount of time was spent performing billions of `fscanf`s to read the geometry data. To alleviate this issue, we developed an internal binary file format, BIFF³, which maintained the exact same structure as the original PBRT files, but stored the geometry data in a binary format which could be read directly with `fread` instead. After converting the scene to BIFF, we were able to parse the model (without textures) in just 15s, compared to 65 minutes using our PBRT parser. The total scene load and setup time to create the geometries, build the BVHs and so on takes approximately 6 minutes.

5.2 Geometry Types, Complexity, and Memory Consumption

With our parser now able to load the triangle meshes in the scene, we loaded these into OSPRay and Embree to render them. Though the model also contains 3 M Bézier curve segments, these are clearly dwarfed by the 164 billion triangles (after instantiation), and thus we began with just the triangle data. Our initial tests rendered the meshes colored by primitive, geometry, or instance ID using OSPRay’s built in debug renderers (see Figures 2-5).

Though this approach worked reasonably well from a rendering standpoint, it required a significant amount of memory just for the geometry. Upon closer investigation, we found several ways of reducing the memory consumed by the geometry. First, as PBRT does not support quad primitives, the JSON-to-PBRT converter exported each quad as a pair of triangles, along with filler texture coordinates and additional data arrays to remap the triangle IDs to quad IDs for Ptex texturing. Removing these arrays and computing these values on the fly provided a significant reduction in memory use. Second, we initially instantiated PBRT *geometries* rather than *objects*. While this was not a problem for other PBRT models, on the Moana Island this approach significantly increased the number of instances in the top-level BVH, from around 39 M to over 100 M.

³A variant of the BIFF format and our PBRT parser are available on GitHub, at <https://github.com/ingowald/pbrt-parser/>

Correcting the parser to instantiate PBRT objects instead provided a further reduction in memory use.

Quads. Having already partially reverted the quad to triangle-pair conversion by computing the texture coordinates and quad IDs on the fly, the obvious next step to reduce memory use further was to completely revert the tessellation, and render quads directly. From the content side this was straightforward, as a visual inspection of the PBRT files revealed that every pair of triangles formed a quad, and thus could simply be merged back together when converting the data to BIFF.

On the rendering side, however, this proved more challenging. While Embree had recently added a quad mesh primitive in version 3, OSPRay was still on Embree 2, and did not support quads. As OSPRay was initially developed for scientific visualization, where quads are uncommon, this had not previously been an issue. Adding support for rendering quads in OSPRay required upgrading OSPRay to Embree 3, which, due to changes in how user geometry work between Embree 2 and 3, and their extensive use throughout OSPRay, was a significant effort. After upgrading to Embree 3 we added a `QuadMesh` geometry to OSPRay which directly mapped to Embree’s `QuadMesh`. Migrating to quads immediately halves the number of geometric primitives, which also reduces the number of BVH nodes, and thus build time and memory use. Upgrading to Embree 3 also provided some additional upgrades to the underlying BVH as well, improving performance further.

Curves. As with the quads, we initially skipped loading the Bézier curves in the scene as Embree 2, and thus OSPRay, did not support them. Although OSPRay did have support for stream line geometries, this geometry was designed for visualization applications and did not support smooth cubic curves nor flat ribbon style curves, making it unsuitable for representing grass and palm fronds. However, after upgrading to Embree 3 we were able to use the new cubic curve types which were recently added to Embree independently from this work. Similar to adding quad support, all that had to be done was add a new `Curves` geometry to OSPRay, which used Embree’s curve primitive internally.

As a result of our efforts to reduce memory consumption when loading and rendering the scene the full model can be rendered on a workstation with 128 GB of RAM. As Embree and OSPRay both allow for zero-copy sharing of the vertex, index, and other data arrays with the application, passing the data to the renderer requires no additional memory use; though Embree will require some additional space to store the BVHs. After constructing the BVHs the viewer requires a total of 100 GB to hold the geometry and acceleration structures, and reaches a peak memory use of 104 GB during the BVH build.

5.3 Ptex

Texture data is used heavily in production rendering, and the Moana Island Scene is no exception: the diffuse component of nearly every primitive in the scene comes from a Ptex texture. Although OSPRay already supported textures, it only supported *image* textures, however Ptex is a *geometry* based texture format baked on top of the underlying meshes [Burley and Laceywell 2008]. Not only does this

mean there is no reasonable way these textures could be converted to 2D images for use in OSPRay, but that OSPRay’s entire view of how textures can be applied to geometry—which was inherently based on image textures—would have to change.

Previously, a texture in OSPRay would be given just the 2D UV coordinates to be sampled and return back the computed color. However, in the case of a Ptex texture, we also need the primitive ID (in Ptex terms, the “face ID”) to find the correct texture to sample, and the barycentric coordinates of the primitive. Modifying OSPRay to pass this data as well was straightforward: rather than passing just the UV texture coordinates to the texture, we pass it the full intersection information, which includes the primitive ID and barycentric coordinates. However, all of OSPRay’s rendering, shading, and texturing code is written in ISPC, which operates on multiple shade points in SIMD. While we now had the right data to pass to Ptex for each sample, the library does not provide a SIMD interface which can be called directly from ISPC.

To call back into the Ptex library we wrote a C-callable shim function which could be called from ISPC with the data for a single sample. In ISPC we then serialize the SIMD execution over the active vector lanes using ISPC’s `foreach_active` construct, and call our shim with the corresponding sample for the lane. While this does lose the advantage of ISPC’s vectorization during texture lookups, in a path tracer it is likely that different vector lanes will sample different textures, impacting SIMD utilization regardless. Further investigation into the potential for an ISPC version of Ptex which can take advantage of SIMD for texture lookups remains an interesting direction for future work.

On the C++ side of the Ptex texture object we use a `PtexCache` to load and cache the texture data, which is shared across all textures in the scene. The cache helps reduce memory use as the required texture data is loaded on demand, though comes at the cost of poor performance when first starting the viewer as frequently used textures are first loaded into the cache.

5.4 Shading

Prior to this work, OSPRay had integrated a reasonably full-featured path tracer. Though OSPRay was primarily designed for scientific visualization, users’ needs beyond classical sci-vis had resulted in this path tracer evolving to support various material types (e.g., glass, metal, plastic), different light sources and area lights, performance optimizations for importance sampling, and progressive refinement [Wald et al. 2017]. However, our hope that this path tracer would meet the needs of the Moana Island Scene out of the box was disappointed. The Moana Island Scene exclusively uses the Disney principled BSDF, which could not be well approximated by the existing materials in the path tracer. Proper support for the principled BSDF was crucial to achieving the correct look for the scene (see Figure 6), and thus we have implemented a slightly modified and improved version of the Disney BSDF in ISPC for OSPRay. One notable difference compared to the original version of the BSDF is that we have made it both energy conserving *and* preserving [Hill et al. 2017].

In its final version the path tracer can largely provide everything the Moana Island Scene requires, achieving the desired look at



Fig. 6. Top: Rendering with the simplified material model OSPRay used before this project. Bottom: The same, with the Disney principled material model that we added for this project. Note the incorrect colors on the Ironwood tree and the disappearance of the ocean in the top image.



Fig. 7. The material model and path tracer support high-quality path tracing effects with progressive refinement, allowing interactive rendering of the Moana Island Scene with full geometric and shading quality. To provide interactive frame rates, we take one sample per-pixel each frame and accumulate these frames over time to refine the image.

reasonable efficiency (see Figures 1 and 7). As with any path tracer there is potential for even better sampling, importance sampling, filtering, etc. In particular, scenes with many more light sources than the Moana Island would likely require additional support for improved sampling strategies.

5.5 Performance

OSPRay achieves interactive performance when rendering the Moana Island Scene by leveraging a set of high-performance libraries, code, and components. To leverage the SIMD capabilities on a single core OSPRay uses Embree [Wald et al. 2014], which is vectorized internally, for ray intersections, and implements the remainder of the renderer, material, and texture sampling code in ISPC [Pharr and Mark 2012].

On a single machine (or node in HPC terminology) OSPRay uses Intel’s Thread Building Blocks (TBB) for multi-threading, which provides utilities for parallel for loops and asynchronous tasks. Work is distributed among multiple threads by parallelizing the rendering task over the tiles of the image, and processing them in a TBB parallel for loop. Each thread then traces a packet of rays in SIMD, using Embree to traverse the ray packet, and ISPC kernels to shade the packet’s rays in parallel. The most similar production film renderer to OSPRay is MoonRay [Lee et al. 2017], which uses larger ray streams instead of SIMD-width packets to further improve memory coherence.

To efficiently distribute rendering work and communicate across multiple nodes on a cluster, OSPRay leverages a Distributed Frame-Buffer [Usher et al. 2019] and MPI. Finally, to provide a high-quality image at low sample counts OSPRay uses Intel’s Open Image Denoise library [Intel 2019] for post-process denoising.

ISPC. ISPC is a compiler for a C-like language for writing single-program multiple data (SPMD) kernels which are executed in SIMD on the CPU’s vector lanes. The code is written as a serial program which at runtime is executed in parallel, with a program instance run per-CPU vector lane in a model roughly similar to GLSL, HLSL, CUDA, and OpenCL. The group of program instances running on a vector unit is referred to as a “gang”. In contrast to GPU programming languages, ISPC runs on the CPU in the same memory space as the calling program, and can share pointers with the “host” program or even call back into the host code. Directly sharing pointers with the rendering kernels is especially valuable for large scenes which already struggle to fit in memory, as this removes the need to make a copy of the data to pass to the compute device.

ISPC’s support for calling back into the host code directly is useful for introducing vectorization into existing large codebases and interfacing with non-vectorized code, without requiring a complete re-write. As discussed in Section 5.3, in this work we leveraged this capability to allow our rendering and shading code written in ISPC to use the Ptex library for texturing. Although the program gang must be serialized to call out to the serial host code, this enables interoperability with existing code for texturing, on demand model loading, etc., which would either be difficult or impossible to port directly to ISPC.

Moreover, ISPC allows for easily writing portable vectorized code, which is highly desirable when deploying renderers across a wide range of hardware. This portability is achieved by compiling a multi-target binary, which includes specialized code paths for each backend supported by ISPC. At runtime ISPC will then pick the correct code to run from this binary for the target architecture. The portability and performance provided by ISPC have made it our language of choice for implementing the core kernels of OSPRay,

with higher-level scene setup, multi-threading, and multi-node code implemented in C++.

Compared to alternatives for achieving vectorization on CPUs, e.g. compiler pragmas, OpenCL, OpenMP, etc., we have found ISPC to provide better and more reliable performance. A key drawback of auto-vectorization and compiler pragmas is that they can easily break when control flow diverges, and revert to fully scalar code. In contrast, ISPC is explicitly a SPMD on SIMD model and will still vectorize the code, though at the cost of introducing control flow masking to handle possible divergence within a gang. Although diverging control flow in ISPC comes with a cost, it is far more desirable to pay this cost and keep the code vectorized than to fall back to completely scalar code in most cases.

However, some care must be taken when writing high-performance code in ISPC. While a complete discussion of performance considerations is beyond the scope of this paper⁴, we discuss a few which are directly applicable to the task of path tracing large, complex models. To minimize control flow masking and allow greater use of scalar registers, we recommended to use `uniform` variables wherever applicable. A `uniform` variable in ISPC is one which is the same across all vector lanes, and can be placed in a scalar register. Moreover, when a branch depends on a `uniform` variable the control flow within a gang is known to not diverge, allowing the compiler to avoid emitting control flow masking instructions.

We also recommend to use ISPC in 32-bit addressing mode. In this mode, all pointers used in ISPC kernels map to 32-bit offsets relative to a `uniform` 64-bit pointer. This allows the compiler to use faster 32-bit address computations and scatter/gather intrinsics, leading to significant performance gains. However, when rendering large data sets 32-bit offsets may be insufficient to access large data arrays of geometry or texture data. In such cases we treat the single array as multiple subarrays, each indexable by 32-bit offsets from a 64-bit pointer, and use ISPC’s `foreach_unique` execution construct to iterate over the unique subarrays being accessed by each program instance. We found that even on the Moana Island Scene, there were no objects large enough to require emulating 64-bit addressing in this manner.

The Distributed FrameBuffer. To efficiently distribute rendering work among nodes and combine the partial results produced by each node, OSPRay uses a Distributed FrameBuffer [Usher et al. 2019]. The Distributed FrameBuffer (DFB) is a general framework for executing image compositing and processing tasks for distributed renderers through a distributed, asynchronous tile processing pipeline. Along with standard image- and data-distributed rendering, the DFB supports more advanced configurations where scene data can be partially replicated among nodes, or some of the scene fully replicated and combined with distributed geometry.

A “distributed renderer” implemented using the DFB consists of a render, responsible for producing image tiles, and a tile operation, which combines the tiles received for some image tile into a single final tile. A tile operation can be a simple averaging to combine multiple samples, or alpha-blended depth-compositing, e.g., for data-distributed rendering. After the tile operation is run, additional post-processing tasks can be performed, e.g., tone-mapping. The DFB



Fig. 8. A crop of the Shot view, with and without denoising at one sample per-pixel. At the start of the progressive accumulation the denoiser provides a significant improvement in image quality, even at very low sample rates.

distributes the execution of the tile operations and post-processing tasks among the processes by assigning tile owners to run tasks for each tile in round-robin order among the nodes.

To render the Moana Island Scene in parallel on multiple nodes we use the image-parallel renderer in OSPRay. The image-parallel renderer works similar to other master-worker rendering architectures previously used in, e.g., Manta [Bigler et al. 2006] and OpenRT [Wald et al. 2002]. This renderer is exposed through OSPRay’s `MPIOffloadDevice`, which transparently distributes the scene data to a set of worker processes running on the compute nodes. These workers then render the scene using the image-parallel distributed renderer implemented with the DFB. The rendering work is distributed among the nodes by assigning the image tiles round-robin to each node to provide even work distribution for most scenes. Each node then renders its assigned tiles in parallel using multiple threads. At the end of the frame the final tiles are gathered onto the head node to display the image.

Denoising. Beyond increasing compute power and improved efficiency, arguably one of the biggest breakthroughs in recent years for interactive path tracing performance was the introduction of denoising techniques [Bako et al. 2017; Bitterli et al. 2016; Chaitanya et al. 2017; Mara et al. 2017; Schied et al. 2017]. Recent techniques based on machine learning have been shown to be fast and capable of high quality images with few samples per-pixel, see e.g., Chaitanya et al. [2017] or Bako et al. [2017]. However, current approaches are not without limitations when applied to interactive rendering of production content. Real-time denoising techniques (e.g., [Chaitanya et al. 2017; Mara et al. 2017; Schied et al. 2017]) can provide smooth images with just a sample per-pixel, but are too approximate for production; while denoising techniques for production rendering (e.g., [Bako et al. 2017; Bitterli et al. 2016]) can provide better quality, they require a higher initial sampling rate, more image features, and do not run in real-time.

In this work we use Intel’s Open Image Denoise library [Intel 2019], which is a fast CPU implementation of a denoiser in the spirit of Chaitanya et al. [2017]. To preserve as much image detail as possible we provide not only color but albedo and normal buffers as well to the denoiser. While at very low sample rates we do observe visible blurring artifacts, in particular on trees, bushes, and reflections, these are preferable to the unfiltered image (see Figure 8). As additional samples are accumulated over time and the image

⁴See the ISPC performance guide: <http://ispc.github.io/perfguide.html>



Fig. 9. The camera viewpoints used in our benchmarks. The Beach camera (left) contains a mix of geometries filling roughly the entire image and a range of materials, and averages 252ms per-frame. The Palms camera (center) consists of diffuse materials, with a mix of large and sub-pixel geometry, and averages 183ms per-frame. The Dunes camera (right) consists of diffuse materials with a range of geometry covering the entire image, and averages 339ms per-frame.

converges, the better image quality provided to the denoiser results in better handling of these fine detail features.

When run in parallel on multiple nodes the denoising is performed on the head node after the image is rendered. This allows the denoiser to access the entire image at once, but clearly poses a scalability issue for large images or expensive denoisers. To avoid the workers remaining idle while the head node is denoising the frame, we run the denoising in parallel to the rendering. When a frame is finished we begin denoising it on the head node and immediately start rendering the next frame on the workers. Distributing the denoising work to be run in parallel across the workers through the DFB would improve performance for large images and expensive denoisers, though would require the workers to perform some form of neighboring pixel exchange to provide the required data for the denoiser. How this exchange can be done efficiently remains an interesting follow-on effort.

5.6 USD Moana

Pixar’s Universal Scene Description format, USD, was designed for fast loading, rendering, and collaborative editing of large-scale production assets [Poh et al. 2018] which makes it a promising standardized alternative to using PBRT or our custom binary format. Originally an internal scene graph format used by Pixar in production, USD was recently released as an open source project, and has subsequently seen a wide adoption across other studios and even use within the game industry [Blevins and Murray 2018]. Due to the lack of publicly available USD datasets for the wider community to develop and test new techniques with, Disney began work on an additional version of the Moana Island Scene converted to USD. As of this writing, the USD version of the dataset remains a work in progress, and does not yet match the full scale or correctness of the publicly released JSON or PBRT versions of the model. It does, however, present a widely used file format that exhibits significantly improved loading times over pbrt and supports subdivision surfaces which were missing in the pbrt conversion. We loaded and rendered the USD Moana Island Scene using the OSPRay backend of the Hydra rendering layer in USD, HdOSPRay⁵ (see Figure 10). The lack of detail missing from only using the subdivision cages, a refinement level of 0, compared to using a tessellation rate of 8 can be seen in Fig. 11.

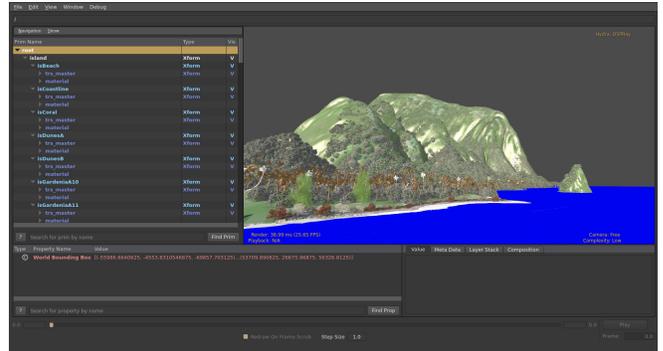


Fig. 10. A rendering of the work in progress USD Moana Island Scene using HdOSPRay.

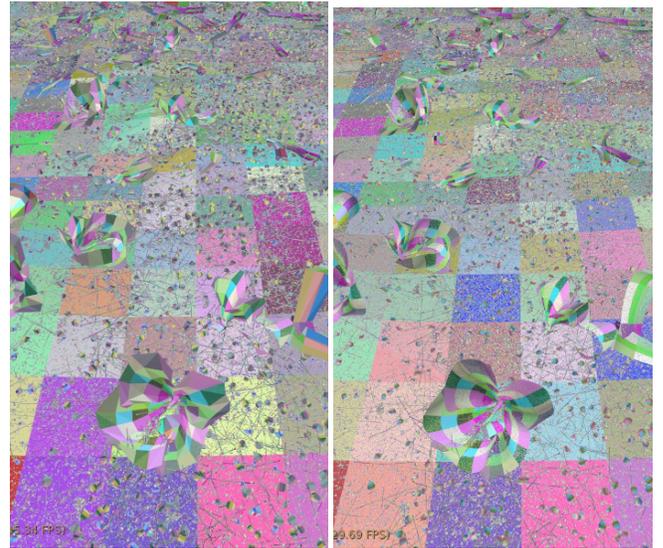


Fig. 11. Renderings of flowers using HdOSPRay and subdivision surfaces with tessellation rates of 0, left, and 8, right.

6 RESULTS

We evaluate the performance of our renderer using the predefined Shot, Beach, Palms, and Dunes camera positions provided in the PBRT scene file. The camera positions chosen cover various configurations in terms of the directly visible geometries and materials, covering a range of cost per-pixel (see Figures 1 and 9)

⁵<https://github.com/ospray/hdospray>

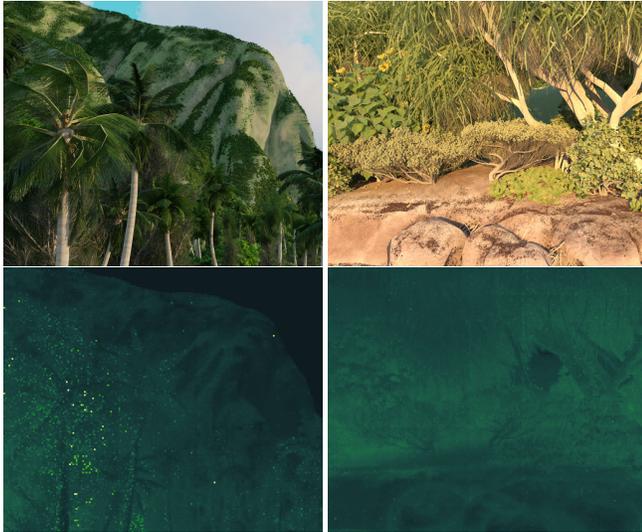


Fig. 12. Crops of two exemplary views on the cost-per-pixel spectrum. Left: The Palms view is cheap on average with some hot spots, and achieves 5.46 FPS and 39.38 Mray/s. Right: The Dunes view is far more expensive on average, and achieves 2.95 FPS at 36.32 Mray/s. Bottom: The same, colored by cost per-pixel from dark (low) to light (high). Both images use the same heat-map scale.

Table 1. Compute time breakdowns for each of the benchmarked views. We find that the majority of time is spent in Embree performing ray traversal and intersection tests, with sampling and shading BRDFs taking the bulk of the remaining time. Surprisingly little time is spent in Ptex, which is likely attributable to the texture caching performed by the library.

Component	Shot	Beach	Palms	Dunes
Embree (T. & I.)	67.28%	70.17%	71.20%	74.99%
PostIsec	7.10%	5.96%	6.99%	5.13%
Ptex	2.26%	2.89%	1.25%	2.27%
Sample & Shade	22.74%	20.45%	19.87%	17.21%
Other	0.62%	0.53%	0.69%	0.39%

The benchmarks are rendered at the film aspect ratio at a resolution of 1536×644 and a maximum path depth of five. We use Ptex’s caching system to manage loading textures, and configure it to allow for an unlimited amount of cache memory and 100 open files. The Ptex cache does take some period to warm up, we found that the first 25 to 30 frames take much longer than subsequent frames, with the first few frames taking orders of magnitude longer as frequently accessed textures are loaded into the cache. To benchmark the renderer after this warm up period we use the first 64 frames as warm up frames, and measure performance over the next 64. Our benchmarks are run using nine Intel Skylake Xeon nodes on the Texas Advanced Computing Center’s Stampede2 system, with one head node and eight worker nodes. Each node has two Intel Xeon Platinum 8160 processors and 192GB of DDR4 RAM.

In terms of overall rendering performance, we find that eight worker nodes are sufficient to provide interactive rendering. For each camera position, we measured the average ray tracing time to

be: 207ms for Shot (36.95 Mray/s), 252ms for Beach (35.29 Mray/s), 183ms for Palms (39.38 Mray/s), and 339ms for Dunes (36.32 Mray/s). The image denoising cost depends only on the number of pixels being processed, and takes on average 130ms per-frame across all the benchmarked views.

The ray tracing times correspond to larger differences in the number of rays actually processed and shaded per-frame, due to the differences in the scenes being rendered. In the Shot and Palms views a large portion of the image only sees the background, and in the Shot view additional camera rays reflect off the water surface into the background. In contrast, the Beach and Dunes views are largely filled with dense geometry with smooth materials, resulting in a large number of diffuse bounces and thus rays traced (also see Figure 12). For example, the average number of rays traced per-pixel on the Palms scene is just 7.29, while the Dunes traces 12.45 per-pixel, corresponding to the lowest and highest average rays per-pixel across the benchmarks, respectively.

To determine where time is spent within the renderer, we break down the total compute time for each view by component in Table 1. Across all scenes we observe that the majority of time (67-75%) is spent in Embree, tracing rays and intersecting geometry, with the second largest amount of time spent sampling and shading BRDFs. The material model currently used in OSPRay’s path tracer returns a set of BRDFs from the material; and with potentially different materials hit by each ray in a packet and different sets of BRDFs returned by these materials, the shading code can become quite expensive and nearly serialized for a packet. The PostIsec measure the time spent computing the properties needed to shade the BRDF, namely the surface normals, texture coordinates, and so on.

The result we found the most surprising was how little time was spent in Ptex after the warm up period. After the bulk of texture data which is needed for the scene has been loaded into the cache, the time spent sampling textures drops significantly. Even in the Beach view, where a large portion of the scene is visible, Ptex lookups only account for 2.89% of the total compute time for a frame. We do note that this is not the case during the warm up frames, especially when data is first being read from disk and cached. During the warm up period the Ptex and Sample & Shade components together take up the bulk of compute time, up to 75% in some cases, as required texture data is fetched and cached.

7 DISCUSSION

In this paper we have presented the challenges encountered and solutions developed to achieve interactive rendering performance on the Moana Island Scene. Such production scenes present a significant challenge to interactive rendering, from loading the data at all, to rendering it interactively at full quality. As production scenes of this scale are typically not available to the broader research community, we hope that by presenting our experiences and difficulties in working with this asset, this paper can provide guidance to other researchers beginning to work with the Moana Island Scene or other similar production assets.

With tools and renderers capable of interactive rendering at full geometric and shading quality on production scenes, artists will be

able to iterate more quickly on modeling and design of film assets. To this end, we are working on integrating the rendering system presented with production tools, and developing native support for USD through HdOSPRay. As modeling tools for film become more interactive, it is interesting to consider whether this faster feedback loop between artist changes and results will change the underlying assets, or the films themselves. Given the ability to truly explore scenes interactively, directors may frame shots differently, or artists be better able to adjust lighting and materials to achieve the desired look.

Finally, while film resolutions are increasing, the rate at which geometric and texture complexity is increasing far outpaces it. When the renderer is only able to parallelize the rendering work over the pixels and samples in an image, there is an inherent limit on the amount of parallelism which can be extracted. To this end, it may be valuable to consider data-distributed rendering of such assets, where subregions of the data are assigned to different nodes, and rays or data moved as needed during rendering. A data-distributed approach may also allow for GPU-based interactive rendering of such production scenes, where memory is more constrained than on a CPU.

ACKNOWLEDGMENTS

The authors would like to thank Disney, and in particular Rasmus Tamstorf, for making the Moana Island Scene publicly available, for granting access to early versions and for assistance with the data. The authors thank the Texas Advanced Computing Center (TACC) at the University of Texas at Austin for providing HPC resources that have contributed to the results reported in this paper.

REFERENCES

- 4A Games. 2019. Metro Exodus.
- Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novak, Alex Harvill, Pradeep Sen, Tony DeRose, and Fabrice Rousselle. 2017. "Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings". *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2017)* 36, 4 (2017).
- Kavita Bala (Ed.). 2018. Special Issue On Production Rendering and Regular Papers. *ACM Transactions on Graphics* 37, 3 (2018).
- James Bigler, Abe Stephens, and Steven G. Parker. 2006. Design for Parallel Interactive Ray Tracing Systems. In *2006 IEEE Symposium on Interactive Ray Tracing*.
- Benedikt Bitterli, Fabrice Rousselle, Bochang Moon, José A. Iglesias-Guitián, David Adler, Kenny Mitchell, Wojciech Jarosz, and Jan Novák. 2016. Nonlinearly Weighted First-order Regression for Denoising Monte Carlo Renderings. *Computer Graphics Forum (Proceedings of EGSR)* 35, 4 (2016).
- Alan Blevins and Mike Murray. 2018. Zero to USD in 80 Days: Transitioning Feature Production to Universal Scene Description at Dreamworks. In *ACM SIGGRAPH 2018 Talks (SIGGRAPH '18)*. ACM, New York, NY, USA, Article 53, 2 pages.
- Brent Burley. 2012. Physically-based Shading at Disney. In *SIGGRAPH 2012 Course Notes "Practical Physically Based Shading in Film and Game Production"*.
- Brent Burley. 2015. Extending the Disney BRDF to a BSDF with Integrated Subsurface Scattering. In *SIGGRAPH 2015 Course Notes "Physically Based Shading in Theory and Practice"*.
- Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. 2018. The Design and Evolution of Disney's Hyperion Renderer. *ACM Transactions on Graphics* 37, 3, Article 33 (July 2018), 22 pages.
- Brent Burley and Dylan Laceywell. 2008. Ptex: Per-Face Texture Mapping for Production Rendering. *Computer Graphics Forum* 27, 4 (2008), 1155–1164.
- Chakravarty R. Alla Chaitanya, Anton Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. "Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder". *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2017)* 36, 4 (2017).
- Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. 2018. RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. *ACM Transactions on Graphics* 37, 3, Article 30 (Aug. 2018), 21 pages.
- Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes Image Rendering Architecture. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*.
- Electronic Arts. 2019a. Battlefield V.
- Electronic Arts. 2019b. Project PICA PICA.
- Luca Fascione, Johannes Hanika, Mark Leone, Marc Droske, Jorge Schwarzhaupt, Tomáš Davidovič, Andrea Weidlich, and Johannes Meng. 2018. Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production. *ACM Transactions on Graphics* 37, 3, Article 31 (Aug. 2018), 18 pages.
- Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramón Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, Adrien Herubel, Declan Russell, Frédéric Servant, and Marcos Fajardo. 2018. Arnold: A Brute-Force Production Path Tracer. *ACM Transactions on Graphics* 37, 3, Article 32 (Aug. 2018), 12 pages.
- Stephen Hill, Stephen McAuley, Alejandro Conty, Michal Drobot, Eric Heitz, Christophe Hery, Christopher Kulla, Jon Lanz, Junyi Ling, Nathan Walster, Feng Xie, Adam Micciulla, and Ryusuke Villemin. 2017. Physically Based Shading in Theory and Practice. In *ACM SIGGRAPH 2017 Courses (SIGGRAPH '17)*.
- Intel. 2019. Intel Open Image Denoise. <https://openimagedenoise.github.io>.
- Hans Keim, Maryann Simmons, Daniel Teece, Jared Reisweber, and Sara Drakeley. 2016. Art-directable Procedural Vegetation in Disney's Zootopia. In *ACM SIGGRAPH 2016 Talks (SIGGRAPH '16)*.
- Alex Keller, Luca Fascione, Marcos Fajardo, Iliyan Georgiev, Per Christensen, Johannes Hanika, Christian Eisenacher, and Gregory Nichols. 2015. The Path Tracing Revolution in the Movie Industry. In *ACM SIGGRAPH 2015 Courses (SIGGRAPH '15)*.
- Christopher Kulla, Alejandro Conty, Clifford Stein, and Larry Gritz. 2018. Sony Pictures Imageworks Arnold. *ACM Transactions on Graphics* 37, 3, Article 29 (Aug. 2018), 18 pages.
- Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized Production Path Tracing. In *Proceedings of High Performance Graphics (HPG '17)*.
- Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. 2017. An Efficient Denoising Algorithm for Global Illumination. In *Proceedings of High Performance Graphics (HPG '17)*.
- Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*.
- Matt Pharr. 2018a. Guest Editor's Introduction: Special Issue on Production Rendering. *ACM Transactions on Graphics* 37, 3, Article 28 (July 2018), 4 pages.
- Matt Pharr. 2018b. Swallowing the Elephant. <https://pharr.org/matt/blog/2018/07/16/moana-island-pbrt-all.html>.
- Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. Physically Based Rendering: From Theory to Implementation (3rd ed.). (2016), 1200.
- Matt Pharr and Bill Mark. 2012. ISPC: A SPMD Compiler for High-Performance CPU Programming. In *Proceedings of Innovative Parallel Computing (inPar)*.
- Kiki Poh, Michael Kilgore, Tom Wichitscripornkul, and Gary Monheit. 2018. Using USD Shading to Provide the "Extra" Touch on Incredibles2. In *ACM SIGGRAPH 2018 Talks (SIGGRAPH '18)*.
- Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal Variance-guided Filtering: Real-time Reconstruction for Path-traced Global Illumination. In *Proceedings of High Performance Graphics (HPG '17)*.
- Rasmus Tamstorf and Heather Pritchett. 2018. Moana Island Scene. <http://datasets.disneyanimation.com/moanaislandscene/island-README-v1.1.pdf>.
- Will Usher, Ingo Wald, Jefferson Amstutz, Johannes Günther, Carson Brownlee, and Valerio Pascucci. 2019. Scalable Ray Tracing Using the Distributed FrameBuffer. *Computer Graphics Forum* (2019).
- Ingo Wald, Carsten Benthin, and Philipp Slusallek. 2002. *A Flexible and Scalable Rendering Engine for Interactive 3D Graphics*. Technical Report. Saarland University.
- Ingo Wald, Greg P. Johnson, Jefferson Amstutz, Carson Brownlee, Aaron Knoll, Jim Jeffers, Johannes Günther, and Paul Navrátil. 2017. OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics* (2017).
- Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* 33 (2014).