Arnold: A Brute-Force Production Path Tracer

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, Solid Angle

Fig. 1. Arnold is a path-tracing renderer used for the production of photo-realistic and art-directed visual effects in feature films (left), commercials (middle), animated films (right), television series, music videos, game cinematics, motion graphics, and others. *Gravity ©2013 Warner Bros. Pictures, courtesy of Framestore; Racing Faces ©2016 Opel Motorsport, courtesy of The Mill; Captain Underpants ©2017 DreamWorks Animation.*

Arnold is a physically based renderer for feature-length animation and visual effects. Conceived in an era of complex multi-pass rasterization-based workflows struggling to keep up with growing demands for complexity and realism, Arnold was created to take on the challenge of making the simple and elegant approach of brute-force Monte Carlo path tracing practical for production rendering. Achieving this required building a robust piece of ray-tracing software that can ingest large amounts of geometry with detailed shading and lighting and produce images with high fidelity, while scaling well with the available memory and processing power.

Arnold's guiding principles are to expose as few controls as possible, provide rapid feedback to artists, and adapt to various production workflows. In this article, we describe its architecture with a focus on the design and implementation choices made during its evolutionary development to meet the aforementioned requirements and goals. Arnold's workhorse is a unidirectional path tracer that avoids the use of hard-to-manage and artifact-prone caching and sits on top of a ray-tracing engine optimized to shoot and shade billions of spatially incoherent rays throughout a scene. A comprehensive API provides the means to configure and extend the system's functionality, to describe a scene, render it, and save the results.

CCS Concepts: • **Computing methodologies → Ray tracing**;

Additional Key Words and Phrases: Rendering systems, production rendering, ray tracing, Monte Carlo, path tracing, global illumination

Authors' address: I. Georgiev, T. Ize, M. Farnsworth, R. Montoya-Vozmediano, A. King, B. van Lommel, A. Jimenez, O. Anson, S. Ogaki, E. Johnston, A. Herubel, D. Russell, F. Servant, and M. Fajardo, Solid Angle, Gran Via 51, 28013 Madrid, Spain.

https://doi.org/10.1145/3182160

ACM Reference format:
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 Trans. Graph.* 37, 3, Article 32 (July 2018), 12 pages.
https://doi.org/10.1145/3182160

1 INTRODUCTION

At a purely technical level, visual realism in computer-generated imagery boils down to two areas: (1) amount of scene detail, e.g., number of geometric primitives, amount of textures, variety of materials, and proceduralism, and (2) quality of the lighting simulation, e.g., soft shadows, reflections, refractions, and indirect light.

For decades, the prevailing production rendering systems were based on micropolygon rasterization (Cook et al. 1987). These could handle geometrically complex data sets but with poor lighting simulation quality. Ray tracing could achieve much higher quality, with accurate lighting effects, but was too costly except on simple scenes and was therefore not considered viable for production.

Over the years, increasing demands for realism forced the users of micropolygon rasterization to find ways of producing more complex, plausible lighting effects within the constraints imposed by that method. Adding features such as shadows or indirect lighting required the use of various precomputation passes to create shadow maps or diffuse inter-reflection caches. The intermediate results from these passes would then be carefully assembled into a final image during the shading of the rasterized camera-visible objects.

This approach resulted in a convoluted workflow revolving around the management of often complex dependencies between

various rendering passes and the storage of their intermediate results. Output was still limited in quality, subject to adjusting many parameters, and prone to temporal flickering due to the use of various types of caches and interpolation. This led to slow artist iteration on frames and entire shots. Moreover, many increasingly desirable effects, such as specular reflection, refraction, and accurate indirect lighting, could not be easily achieved. Renderers were augmented with ray-tracing functionality to selectively add such effects during the final shading (Christensen et al. 2006). Ray tracing was slowly gaining traction, but a simpler and more scalable approach was needed for artists to naturally get plausible lighting effects while iterating faster and with fewer errors.

1.1 Path Tracing

Stochastic path tracing (Kajiya 1986) provides an elegant approach to photo-realistic rendering that is far less subject to the issues experienced with rasterization. It naturally simulates all lighting effects, such as soft shadows, indirect illumination, glossy reflections, motion blur, depth of field, hair/fur, volumetrics. There is no requirement to split the rendering into passes with intermediate products to compute such effects—the user simply provides the scene and desired settings, and a final image is computed at once.

Beyond disentangling the rendering workflow, path tracing brings other advantages. It is relatively easy to control via a handful of sample-count parameters. The approximation error appears as fine-grained noise that is less visually objectionable than the "blotchy" artifacts of cache-based methods and is not susceptible to low-frequency flickering in animation. It also allows for progressive rendering, which in turn enables quick iteration over shading and lighting, and the amount of noise diminishes predictably as the sample counts are increased.

The advantages of path tracing were well known, but its longer render times and lower efficiency with important production features such as motion blur, complex geometry, and volumetrics appeared to be a weakness of its more brute-force approach. However, production workflows had become so complicated that artists were wasting significant time and effort attempting to do what path tracing could already easily do, though perhaps more slowly. It became compelling to try and tackle the weaknesses of path tracing while playing to its strengths, especially given that an hour of an artist's time can cost hundreds of times more than an hour of compute time.

1.2 The Arnold Philosophy

The development of Arnold was spurred by the need for a streamlined physically based production renderer that scales to film complexity. Arnold takes on the ambitious goal of efficiently rendering artifact-free images of dynamic scenes with massive complexity in-core, while at the same time simplifying the workflow, infrastructure requirements, and user experience. Path tracing naturally ticked most boxes.

Our guiding philosophy is that the renderer must be easy to use and be able to provide quick feedback to artists. Speed and memory efficiency are important for a production renderer, and throughout the article, we discuss some of the strategies we employ to achieve these goals. Perhaps less obvious is that simplicity is also important for saving both artist and compute time. Reducing the number of knobs and dials makes the renderer not only easier to use but also lowers the chance of a wrong setting being chosen that could result in slower or even unusable renders. These goals are met to a large degree with the switch from rasterization to path tracing, as discussed above. To further enable users to focus on their creative work, Arnold focuses on providing predictable performance and scalability with input complexity.

Unlike in-house renderers tailored to specific pipeline needs, a commercial renderer like Arnold has to integrate with the unique workflows of many facilities, each using different (often custom-made) tools and file formats. Arnold has therefore been conceived to be as much an application programming interface (API) as a rendering back-end. The API is used to incorporate Arnold into existing content creation tools and to cater to custom needs by extending its functionality via, e.g., shaders, file loaders, or entire end-user applications.

2 SYSTEM OVERVIEW

Arnold has been designed for both offline and interactive rendering, though its API allows users to implement their own ray tracing-based tools, such as rendering to textures (a.k.a. baking). Interactivity is not just limited to moving the camera but could be any other scene change. To reduce latency in interactive rendering, we avoid computationally expensive global optimizations, such as merging geometric objects into a single ray-tracing acceleration structure or precomputing shading or lighting.

2.1 Node Structure

The system is built on top of a programmable node-based structure. Every scene component, including geometry, lights, and cameras, is a node that is simply a type with a collection of named parameters. Nodes can be instanced multiple times and interconnected in networks, e.g., to form complex geometric and shading surface details. So-called "procedural" nodes can invoke user-provided code to, e.g., create child nodes that can represent the feathers on a bird or imported data from a custom file format. Node data can be serialized in a human-readable Arnold scene source (`.ass`) file format.

A flat list stores the top-level geometry nodes, such as triangle meshes, curve sets, and procedurals, each having its own transformation. This simple structure was chosen for the sake of efficiency of the rendering code and ray-tracing acceleration structures. It does not support grouped node transformations; this has not been a big limitation in practical workflows as scene translators have picked up the slack for flattening complex input hierarchies. When necessary, arbitrarily deep hierarchies inside Arnold can still be achieved via procedural node nesting, as we discuss in Section 3.5.4.

2.2 Initialization and Delayed Node Processing

Once all nodes have been created, either via the API or by code in procedural nodes, the bounds of all geometry nodes are computed (bottom-up) and a ray-tracing acceleration structure, a bounding volume hierarchy (BVH), is built over top-level ones. Individual nodes have separate BVHs built over their own contents, i.e.,

geometric primitives or, for procedurals, child nodes. However, the construction of each such BVH, as well as any involved geometry processing, is delayed until a ray later intersects the node's bounds.

There are two main benefits to delayed node processing. One is that if no ray ever hits an object, we avoid both the time and memory overhead of processing its corresponding node. The processing includes polygon/curve subdivision, smoothing, displacement, and the BVH construction over the final geometric primitives, as detailed in Section 3. The savings can be substantial when the object has subdivision applied. The other benefit of this approach is that it can reduce the time to first pixel in an interactive progressive rendering session, allowing some pixels to be quickly rendered and displayed before all objects in the scene have been processed. Such a scenario is especially common in large outdoor environments. In fact, it is not unusual for some object processing to still happen on the second or third progressive rendering iteration, since up to that point relatively few rays have been traced through the scene.

There are also downsides to delayed processing. One is increased code complexity, especially with regard to parallelism and making sure threads do not block if they trace rays that hit an object being processed. Another is that it makes optimizations across the hierarchy of BVHs more difficult, as we discuss later in Section 3.4.

Until very recently, the execution of procedural-node code used to also be delayed until a ray intersected the node's bounds. This was changed for two reasons. One was to open the door for optimizations across the entire node hierarchy, such as BVH construction or the tessellation of a mesh w.r.t. the closest distance to the camera among all of its instances. Another was to unburden the user from having to specify bounds for the procedural, needed for top-level BVH construction. Computing such bounds would require extra pre-processing and was not always easy to do accurately ahead of time, e.g., for procedurally generated or archived, i.e. disk-dumped, geometry. As a result, the bounds were often too loose or too tight, resulting in inefficiencies or an incorrect BVH, respectively. While slightly increasing the time to first pixel, abandoning delayed procedural expansion resulted in an overall performance improvement, in addition to the added user convenience.

2.3 Ray and Shader Processing

Arnold's ray tracing kernel processes rays and shades their corresponding hit points one at a time, without using ray packets. We still benefit from SIMD parallelism during BVH traversal (Wald et al. 2008) and by intersecting both multiple primitives with a ray or a single primitive with a ray. We would likely achieve higher performance if we used large ray packets when rays are coherent, e.g., for rays emanating from a small-aperture camera (Boulos et al. 2007; Reshetov et al. 2005). However, most path-tracing rays do not fit this criterion. According to our experience on production renders, the percentage of coherent rays goes down as scene complexity, number of bounces, and usage of motion blur and depth-of-field all increase. This makes it difficult to justify pursuing ray packets as they only accelerate a small fraction of all rays.

Batching similar rays or shaders together helps in finding more coherence. This is especially attractive for SIMD shading and texturing, even with single-ray traversal (Áfra et al. 2016; Eisenacher et al. 2013; Lee et al. 2017). While batch shading, in particular, is an interesting avenue we would like to explore, we must be mindful that incorporating such a strategy in Arnold would likely not work with preexisting C++ shaders, require a new more restrictive shading API, and burden its users with writing vectorized shaders. Nevertheless, with the adoption of higher-level shading languages (see Section 4.6), batch processing may become more practical via automatic shader vectorization (Gritz et al. 2010; Lee et al. 2017).

Fig. 2. The Knowhere station in the film *Guardians of the Galaxy* comprised 1.2 billion unique triangles that fit within 30GB of memory in Arnold. *©2014 Marvel Studios/Walt Disney Pictures, courtesy of Framestore.*

Arnold does not support out-of-core geometry (Pharr et al. 1997; Son and Yoon 2017; Budge et al. 2009). This self-imposed restriction allows for a simpler and easier-to-improve renderer, which in turn lets us focus our effort on the more common case of in-core scenes.

3 GEOMETRY

Feature films require high levels of geometric complexity from a variety of primitive types. Arnold provides polygon meshes, curves, points, volumes, and implicit surfaces for these needs. It is optimized to efficiently store hundreds of millions of primitives and accelerate ray intersections against them (see Figure 2).

3.1 Mesh Processing

The first ray that intersects the bounds of a geometry node triggers the execution of the node's associated geometry processing steps that culminate with the BVH construction. For polygon mesh nodes, these steps are subdivision, displacement, and normal adjustment.

3.1.1 Subdivision. Arnold supports Catmull-Clark and linear subdivision. It can work on cages with arbitrary n-gons and non-manifold topology, which frequently occur in production meshes. The system will also subdivide any user data associated with polygons and vertices. Different boundary interpolation modes and infinitely sharp and soft creases on vertices and edges are also supported. Creases do not add too much code complexity thanks to template specialization for subdivision rules and a simple approximation for neighboring limit normals. Subdivided surfaces are compatible with the equivalent OpenSubdiv (2017) definitions—a hard requirement for some users who rely on a common language to share cages between providers. The final tessellation

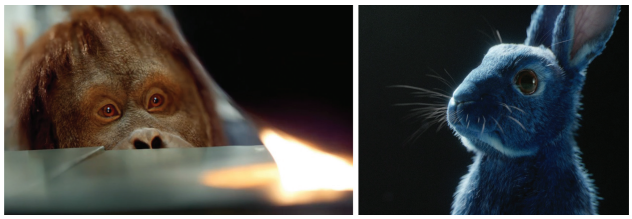

Fig. 3. Curve primitives are most commonly used to render hair and fur. *Maya ©2015 SSE, courtesy of The Mill; Follow The Rabbit ©2017 O_2, courtesy of The Mill.*

corresponds to uniform or adaptive per-patch subdivision. Users may also choose to skip subdivision for patches lying outside the view/dicing camera frustum, as these are unlikely to noticeably contribute to the final image.

3.1.2 Displacement and Autobump. Arnold's displacement stage can only move existing vertices, therefore the subdivision's tessellation should provide enough resolution to capture the user's desired profile shape and overall smoothness. Any additional high-frequency displacement detail that cannot (or is impractical to be) represented by actual geometry is captured via a surface normal adjustment feature that we call *autobump*. During shading, we run the displacement shader three times for each shading point to determine its would-be displaced location and tangent plane. The shading normal is then tilted to capture that high-frequency detail. This does not exactly match the shading of the real displaced geometry, but is a good approximation in practice. The drawbacks of the approach are very similar to those of normal mapping.

The autobump feature allows for saving memory by lowering the subdivision level, but it comes at the cost of more expensive shading due to the extra evaluations of the displacement shader. As memory capacities grow, it becomes increasingly likely to have mesh densities approaching one triangle per pixel, and consequently the reliance on autobump will probably wane.

3.2 Curves and Points

Thin primitives are a difficult case for ray tracers, due to potential aliasing resulting from insufficient ray sampling density (Figure 3). Under animation, this aliasing manifests as buzzing as the primitives scintillate into and out of existence. To alleviate this, we use a RenderMan-inspired (Cook et al. 2007) *minimum pixel width* parameter, which prevents aliasing by (1) expanding the width of curves and points so that they cover at least the specified distance across a pixel, and (2) compensating for this expansion by making the curves proportionately more transparent. This alteration produces a slightly different look, therefore the decision whether to use this feature has to be made early on in the look development cycle. Toggling the feature can be difficult once the look of an object has been approved. Another drawback is a performance hit due to the increased number of semi-transparent intersections along rays. We address this issue via stochastic opacity, as detailed in Section 5.5.

Curves can be stored in piece-wise cubic Bézier, Catmull-Rom, B-spline, and linear formats. During intersection testing, we use basis matrices to allow the existing control points to be intersected as Bézier curves. Curves are split into segments based on the curve basis; these segments are then stored in a regular axis-aligned BVH. Because long diagonal curves are common, we also bound each with an oriented cylinder, which can more effectively cull ray misses. If a ray intersects the cylinder, then we do a comparatively more expensive ray-curve intersection test (Nakamaru and Ohno 2002). All these steps are done in SIMD, such as testing four cylinders at a time. Point primitives are also intersected four at a time using SIMD.

3.3 Volumes and Implicit Surfaces

Detailed fluid-simulation effects like pyrotechnics and water are typically described by volumetric data that is highly non-uniform. Arnold has both a volume integrator and an implicit surface solver, and to improve their efficiency we require volume plug-in nodes to provide intervals along a given ray where there is valid or interesting data. Tight ray bounds can significantly speed up the volume integration and the convergence of the numerical implicit solver. By requiring the plug-ins to calculate ray intervals on the fly, no additional acceleration structures nor caches need to be built beyond the native representation. Hierarchical volume structures such as OpenVDB already provide methods for quickly determining valid-data intervals (Museth 2013). During the development of our OpenVDB plug-in for Arnold, we added support for ray intersections and gathering of ray extents through leaf cells, which we contributed to OpenVDB as its first full implementation of ray tracing.

3.4 Ray Acceleration Structures

When Arnold was introduced, uniform grids were seen as fast to build and traverse. For instance, it was around the same time when the first interactive ray tracer was presented, and it was based on grids (Parker et al. 1999). At the time, SIMD was not used for ray tracing, motion-burred geometry was a rarity, and scenes were simple enough that the "teapot-in-a-stadium" problem[1] could often be easily avoided in a one- or two-level grid.

Since then, SIMD was shown to be widely beneficial for ray tracing, including ray casting in grids (Wald et al. 2006), but was never shown to be practical for tracing incoherent rays through a grid. Motion blurred objects and complex scenes were becoming more important for production rendering, which are hard to support in a grid. Once it was shown how BVH structures could lift these limitations (Wald et al. 2007, 2008), Arnold switched completely over to using such structures. At present it employs a 4-wide BVH (Wald et al. 2008) built using a binned surface area heuristic (SAH) method (Popov et al. 2006). Rays are traversed serially through the BVH using our robust SIMD traversal algorithm to ensure that valid intersections are not missed during traversal due to numerical precision errors (Ize 2013).

3.4.1 Parallel On-Demand Construction. The first ray that intersects the bounds of a geometry node initiates its BVH construction. If, in the mean time, other rays hit the node, then their threads will join this effort instead of simply blocking in anticipation for

[1] A failure case for acceleration structures based on uniform space subdivision: A grid built over a physically large scene will have its cells so large that a comparatively small but geometrically complex object can end up being fully contained within a single cell.

the construction to finish. Despite this added complexity, our parallel construction is fast. On a dual E5-2697 V3 CPU it can build the popular 7.2M-triangle Stanford Asian Dragon model in 0.31s (23M tri/s) on its 56 logical cores and in 5.6s on a single core (1.3M tri/s). This is close to linear scaling when factoring in the Intel's Turbo Boost 3.6GHz of the single core build versus the 2.6GHz when using all cores. Furthermore, the parallel build running with one thread does not have any significant time overhead compared to a dedicated serial-build implementation. This is important, since it is common for many small objects to end up being built by just a single thread.

In addition to time efficiency, memory efficiency during construction is also important. Even a brief spike in memory usage during construction can be enough to exceed a system or user-imposed memory quota and cause Arnold to be abruptly terminated. This means we can only use parallel BVH construction strategies that do not consume significant amounts of memory, especially at high thread counts. Our parallel construction only has a small constant-sized memory overhead over a dedicated serial implementation.

3.4.2 Tree Quality. Every geometry node in Arnold has its own separate BVH, which is a consequence of the on-demand processing. The resulting multi-level BVH, which can be arbitrarily deep due to nested procedurals (see Section 3.5.4), allows for fast time to first pixel. However, it also results in overall slower renders compared to building a single BVH over the entire scene geometry. Testing on the bathroom scene from Figure 4, a single BVH over the entire geometry, combined with other aggressive BVH and triangle intersection optimizations, halved our traversal and intersection time. However, the total render time improved modestly by just 9%, because only about 20% of it was spent on ray traversal and intersection tests. Interestingly, 26% was spent on sampling the two light sources in the scene, excluding the shadow ray casting. A further 10% was spent on pre-shading computations, i.e., ray differentials, normals.; 9% on the relatively simple texturing; and the rest of the time was split up among various other areas. While not fully representative of an actual production scenario, this test shows that render times today are not dominated by ray tracing, and therefore the returns from accelerating it further are diminishing.

The improvement from BVH merging will be larger for input scenes where the individual BVHs overlap substantially; such scenes are common in practice. Unfortunately, inserting the node's contents into a global BVH on-demand is challenging to do correctly and efficiently in a thread-safe manner. Despite this challenge, we would like to eventually support this so that users do not need to concern themselves with overlapping objects.

3.5 Memory Efficiency

Our choice of in-core path tracing requires all scene geometry to fit in memory. We ameliorate this through the use of various techniques for compressing data and eliminating redundancies as well as by choosing a balance between render time and memory usage. For instance, we forgo the speed improvement that comes with spatial splitting in BVHs (Popov et al. 2009; Stich et al. 2009), because we feel it does not justify the potential memory increase. Furthermore, since BVH performance is often not the dominant cost in our renderer, it is easier to turn down BVH optimizations that increase memory.

Fig. 4. This scene comprises 152 meshes totaling 402K triangles and takes 9% longer to render with Arnold's multi-level BVH (two-level in this case) than with a single flattened BVH.

3.5.1 Array Compression. We employ both lossless and lossy compression of data arrays. For example, meshes with fewer than 256 vertices could have their indices exactly stored each in just an 8-bit integer. Additionally, shading normals can be packed from 12 bytes (3 floats) into 4 bytes each with negligible loss of accuracy (Engelhardt and Dachsbacher 2008).

3.5.2 Polygon Representation. Storing quads instead of triangles as base primitives, when applicable, offers substantial memory savings in both meshes and BVHs. For instance, if all pairs of triangles in a mesh can be converted to quads, then the number of primitives is halved, which in turn can approximately halve the number of nodes in the BVH built for that mesh.

3.5.3 Instancing. Another common method for reducing memory footprint is through instancing of geometry nodes to allow for compact description of massive scenes containing vegetation, crowds, debris, cities, and so on. While the traditional implementation only allows overriding transformations and materials, we include nearly all non-geometry object attributes. For example, we support overriding UVs, face color, and vertex colors per instance. This flexibility makes it easier for artists to add interesting variation while keeping the scene in core.

3.5.4 Procedural Nesting. The child nodes of a procedural can themselves be procedurals to form a potentially deep node hierarchy. Additionally, an Arnold `.ass` file can be referenced in another `.ass` file as a procedural, i.e., as a native form of on-disk cache of scene data. When used in this manner, repeated references of the same `.ass` file will be automatically created as instances of the corresponding procedural node to save memory. This was used to great effect in films with massive environments, such as the ring-shaped space station in *Elysium* (Figure 5). The ring comprises four instanced quadrants, which in turn recursively contain groups of instanced trees and houses with substantial variation getting down to the level of a single object. The geometric structure of the station was effectively represented by a complex directed acyclic graph of `.ass` files referencing each other.

Data	Usage
Polygon meshes	4,015 MB
Indices	2,031 MB
Vertices	1,266 MB
Normals	441 MB
UV coordinates	180 MB
BVHs	2,528 MB
Node overhead	800 MB
Instance overhead	456 MB
Total	**8,784 MB**

Fig. 5. The production version of the *Elysium* station (left, middle) comprised 5 trillion triangles thanks to the use of multi-level instancing of `.ass` files and Arnold's space-efficient geometry structures. The incomplete version of the scene we have, with no textures and only 4 trillion (144 million unique) triangles, requires about 9GB of RAM to render with the current Arnold version (5.0.2). A partial memory-usage breakdown is shown on the right. We suspect that with textures (studio used 4GB of texture cache) and the additional missing data (max. 2GB), it would require at most 15GB. *©2013 CTMG, courtesy of Whiskytree.*

4 RENDERING

Predictably scaling rendering performance and image fidelity with increasing complexity is important as it helps artists find a good balance between quality and resource usage. To that end, preventing image artifacts, especially flickering in animation, and extreme noise in difficult lighting configurations is also of particular importance. We aim to optimize Arnold's scalability under the hood without exposing undue complexity to the user. However, avoiding pathological performance in corner cases sometimes requires user intervention.

4.1 Path Tracing

Arnold's rendering subsystem is based on brute-force path tracing from the camera with direct lighting computation (Kajiya 1986), a.k.a. next-event estimation, and optional splitting at the first visible surface and/or medium. Beyond controls for adjusting sample counts and number of bounces, few details about the process are exposed. The rule of thumb for reducing noise today is to simply increase the number of camera samples (paths per pixel, etc.) and keep the splitting factors (diffuse, glossy, etc.) at one. Tweaking individual splitting factors was more productive in the past when few lighting effects and short paths were used.

We adhere to this simple algorithm, because its basics are easy for users to understand and therefore control, and it performs well in the most common lighting configurations encountered in production, such as open environments and character-centered shots. However, this is also where our two goals of speed and simplicity can also conflict. For instance, caching-based algorithms, such as irradiance caching (Ward et al. 1988) and photon mapping (Jensen 2001), do often allow for faster renders. However, they are also prone to temporally flickering image artifacts that require additional tweaking to ameliorate, or their performance and memory footprint may not adequately scale in complex scenes and on many-core hardware. Sophisticated bidirectional methods are generally more robust to varying lighting configurations than unidirectional path tracing (Georgiev et al. 2012; Hachisuka et al. 2012), but in the vast majority of production scenes they result in slower renders and also complicate the computation of ray differentials w.r.t. the camera. Markov chain methods (Hachisuka et al. 2014; Jakob and Marschner 2012) have similar issues and in addition produce correlations in the pixel estimates that are highly prone to flickering in animation. Adding these as optional modes in Arnold would go against our goal of simplicity and could paradoxically result in slower renders. An alternative production solution worth investigating is unidirectional path guiding (Müller et al. 2017). Such techniques, however, involve caching, which adds code complexity, especially with motion blur, and is prone to flickering in animation.

4.2 Aggressive Variance Reduction

While unidirectional path tracing is not generally prone to artifacts, it can suffer from extreme noise under strong and focused indirect light. Some experienced users know how to work around certain special cases. For example, they will avoid placing light sources close to geometry and will model a recessed luminaire by using an invisible planar light source at the opening. Alternatively, they will clamp the illumination range of such sources to some minimum distance. However, more general cases involving multi-bounce light transport and/or glossy surfaces require more general approaches. One technique we employ to avoid noise from rare but intense light paths is to clamp the total as well as only the indirect illumination contribution of every camera sample to a user-set threshold.

A more elaborate noise suppression approach is to take advantage of the obscuring effects of light bouncing at diffuse or rough glossy surfaces. High-frequency illumination details often cannot be seen through such bounces but remain very difficult to simulate with a path tracer. Arnold by default employs techniques that avoid having to deal with such paths in a way that tries to be as energy preserving as possible. This includes adaptively increasing surface roughness (a.k.a. path space regularization (Kaplanyan and Dachsbacher 2013)) and skipping specular caustic paths. Surface roughness is clamped to a minimum value dependent on the maximum roughness seen along the path. This blurs caustics but can substantially reduce the variance of indirect lighting while retaining the overall effect.

Fig. 6. Rendering this flyover of London with a million light sources at about 1.5 hours/frame on a 2013-era computer was made possible by using the low-light threshold and our light BVH, which allowed for sampling roughly 50 lights per shading point. *©2013 Company Pictures/Stormdog.*

4.3 Light Source Culling

For efficiency reasons, Arnold ignores light sources whose expected contribution to the point being shaded is below some user-set threshold. While estimating the contribution of point lights is trivial, area lights are more complicated because of geometric factors and potentially textured emission. While we could do the estimation by sampling the light for every shading point, a much more efficient and scalable approach is to instead compute an axis-aligned bounding box for the region of influence of every light and then ignore those whose bounding boxes do not contain the point being shaded. We create a BVH over these boxes that we traverse to quickly find at each shading point the list of significantly contributing lights, which we then sample for direct illumination.

The culling technique works well for most scenes and allows scaling to many lights. For instance, Figure 6 shows a nighttime scene of London illuminated by a million lights, where most shading points required sampling of only around 10 to 50 lights, with a few small regions requiring around a hundred.

Our technique can, however, completely fail in certain situations where lights have little individual influence but combined are significant, such as if a TV screen has each pixel modeled as a separate light. These failure cases can often be solved by combining the individual lights into a larger textured area light. A single light is also faster and easier to sample. Still, it would be preferable if artists did not have to work around these corner cases. Stochastic light selection is a promising approach that allows for a fixed light-sample budget per shading point (Conty and Kulla 2017) in contrast to our current approach of sampling each significant light at least once.

4.4 Motion Blur

Arnold mainly targets rendering of dynamic production scenes, for which correct and efficient rendering of motion blur is essential. We support two types of motion with an arbitrary number of keys, i.e., time samples, per node. Transformation motion is specified by an affine matrix, while in deformation motion the shape of the object, e.g., its mesh vertices and normals, changes over time.

4.4.1 Deformation Blur. For meshes with deformation motion, we build one BVH over every pair of motion keys. Each pair gives the start and end primitive bounding boxes that when linearly interpolated during BVH traversal will contain the moving primitive.

4.4.2 Volumetric Blur. For grid-based volumetric data, such as OpenVDB, we implement Eulerian motion blur (Kim and Ko 2007). Each time a data channel is sampled, the velocity channel is queried first at the sampling location and once again along that velocity at the time of shutter open. We then use this estimated velocity to backtrack a position where the data channel is finally queried. Note that this requires the simulation to have sufficient padding of the velocity channel.

To not miss any potential ray intersections with data in motion, the active cells of the volumetric data are conservatively dilated by the maximum velocity times the duration of the frame. Being the result of a simulation, the velocity channel can be very noisy and often contains locally very high values. This can make the conservative dilation too large and explode the render times of motion-blurred volumes. To mitigate the issue, we have added an outlier filter to remove extreme velocities. This keeps the bounds expansion within control in the presence of erroneous velocity values, with correspondingly smaller ray intervals through the volume that must be integrated and sampled (see Section 3.3).

A future improvement would be to dilate the bounds more intelligently. Eulerian motion blur is costly as it needs two additional lookups to estimate the velocity. Temporally unstructured volumes could make volume sampling more efficient (Wrenninge 2016); however, they require pre-processing and a non-standard data structure currently unsupported in most simulation packages.

4.5 Textures

Especially in film, it is not uncommon for a given frame to reference an entire terabyte of uncompressed floating-point texture maps with 8K or even 16K resolution each. Some of these textures, e.g., for displacement of vertices, may only need to be used once and so need not persist in RAM for the entire duration of rendering. However, most other types of textures are made for object shading and will be accessed repeatedly and at random due to the incoherent nature of path tracing. This necessitates either storing such textures in RAM or rescheduling the texture evaluation (Eisenacher et al. 2013). Even without limitations in RAM size, simply fetching all this data from a local or networked disk, even in compressed form, could easily take an hour, or much more if the file servers or network are saturated with texture requests from the render farm.

4.5.1 MIP-Mapping and Ray Differentials. As Pharr (2017) points out, most, if not all, film renderers attempt to get around the aforementioned issue by using ray differentials (Igehy 1999) to read in from disk tiles of texture data at the MIP map level required for properly displaying the texel without aliasing or over-blurring artifacts. This often allows for an order or two of magnitude reduction in the working set of texture data that must be read in from disk.

Ray differentials can be extended to account for surface roughness, e.g., by widening the ray footprint after glossy reflections (Suykens and Willems 2001). Unfortunately, such differentials can

be difficult to derive and costly to compute, especially with complex materials. Simpler approximations have therefore been proposed (Christensen et al. 2003). Arnold takes a middle ground and tracks the ray differentials of Igehy (1999), but not the additional partial derivatives of Suykens and Willems (2001), and instead widens the ray differentials according to a microfacet-inspired heuristic. More principled approaches have recently been proposed that we would like to investigate, such as that of (Belcour et al. 2017).

4.5.2 Texture Mapping. Arnold provides direct support for UV texture mapping, tile-based schemes such as UDIM, and projective textures. These parameterizations work well with MIP-mapping: A few texels from an appropriately chosen MIP map level can be reused over all the shading points within the ray footprint. For large enough footprints, these few texels could be shared by many triangles, or at the limit, by even the entire mesh. This can result in extremely efficient texture usage and thus rendering performance.

Generally, the highest-resolution MIP map reading will take place at objects directly seen from the camera. Such spatially coherent shading points are trivial to sample, e.g., via tile-based rendering, allowing successive lookups to often use the same texture tile. Once that image region is rendered, that high-resolution tile will likely never be used again and so can be quickly evicted from memory.

4.5.3 Texture Caching. The minimum required amount of texture data can still be too large to fit in the often small amount of unclaimed RAM, so we use the texture cache of OpenImageIO (2008) to further reduce memory usage. Generally a 2 to 4 GB cache, shared by all render threads, suffices. As Pharr (2017) shows, a well implemented texture cache can scale to many cores.

4.6 API and Shaders

Arnold provides a C++ API for populating and introspecting the scene as well as for implementing custom nodes via callbacks, such as volumes, procedurals, and most notably shaders. Over the years, users and third parties have leveraged the API to add capabilities often unforeseen by us. These include texture baking via specialized cameras that map texture space to camera space, as well as shader extensions that output extra data to facilitate post-render object-mask extraction.

Shaders may be written in C++ or Open Shading Language (OSL) (Gritz et al. 2010), where both can be intermixed within the same shader network. A C++ shader is free to return a (color) value computed in an arbitrary way: it can cast rays with custom payload and even integrate the incident light at the query point. The provided flexibility has been particularly useful in the past for users implementing features such as light portals, ray-marched heterogeneous media, and layered shading models (Langlands 2015). Unfortunately, exposing this level of flexibility has also had negative consequences: shader writers could go overboard and implement entire custom rendering engines within shaders, which we could not support easily. This also limited Arnold's potential as it became increasingly difficult to make rendering improvements without breaking the API.

Over time, we have added enough missing functionality to reduce the need for shaders to act as light integrators themselves.

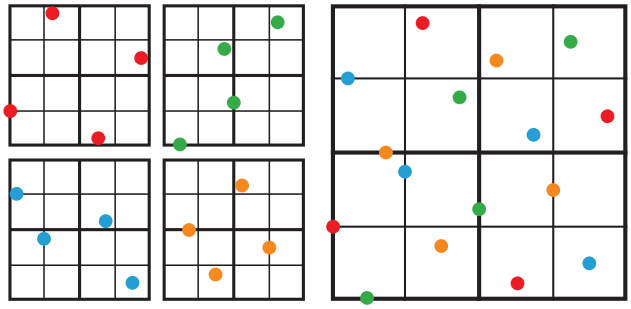

Fig. 7. Each of M^2 camera rays can split into N^2 paths using a CMJ pattern; here, $M = N = 2$. These M^2 patterns of size N^2 each (left) are constructed in a way that their union is itself CMJ pattern of size $(MN)^2$ (right).

Arnold now encourages users to write shaders that instead return a *closure*, which describes the bidirectional scattering distribution function (BSDF) at the point being shaded (Pharr et al. 2016). This paradigm reduces flexibility somewhat but also allows Arnold to improve its built-in algorithms and techniques such as multiple importance sampling (MIS) (Veach and Guibas 1995) transparently to shaders. The closure paradigm allows users to focus their shader development on what matters most to them anyway—the spatially varying object appearance. OSL offers a friendly syntax for writing such shaders. OSL shaders also often perform faster than their C++ counterparts, since their just-in-time compilation can optimize the code of entire complex shader networks.

5 THE IMPORTANCE OF SAMPLING

Improving sampling efficiency has been instrumental for increasing Arnold's rendering performance over the years. Thanks to the use of (better) importance sampling techniques, many noise-inducing components have substantially improved in quality transparently to users. Furthermore, turning certain deterministic evaluations to stochastic via techniques like Russian roulette (Arvo and Kirk 1990) has made rays cheaper and has allowed us to simulate more effects, to eliminate all illumination caching, and to use fewer approximations, thereby reducing memory usage and simplifying users' workflow.

5.1 Sampling Patterns

An efficient way to reduce noise is to use high-quality stratified sampling patterns. Arnold employs correlated multi-jittered (CMJ) patterns (Kensler 2013), which we have extended to two levels so that splitting paths at the first shading point does not introduce additional noise compared to tracing correspondingly more camera rays. That is, we make sure that the union of the CMJ splitting patterns for all camera rays in a pixel is also a CMJ pattern (Figure 7).

A single CMJ pattern can be used for every sampling decision inside a pixel, if individual decisions are properly decorrelated. We accomplish this by applying pseudo-random shuffling and Cranley-Patterson (CP) rotation (Cranley and Patterson 1976) to the samples.

We also use the same CMJ pattern for every pixel. When doing so, traditional wisdom dictates pixel decorrelation, e.g., via CP

rotation with a randomly chosen offset per pixel. However, we have found that by instead carefully *correlating* these CP offsets across pixels, the visual perception of the resulting noise distribution can improve substantially (Georgiev and Fajardo 2016).

For every pixel, for each sampling dimension, we can simply look up the pattern CP offset in a pre-computed blue-noise mask. However, using the same mask for every dimension would introduce correlations as all dimensions would offset the sampling pattern by the same amount. To avoid correlations, we pseudo-randomly shift the blue-noise mask in image space for each sampling dimension.

Unfortunately, randomly shifting the dither mask ruins the desirable blue-noise error distribution in cases when multiple sampling dimensions contribute significantly to the pixel variance. Ideally, a single high-dimensional mask should be used, but achieving high-quality blue-noise in more than a few dimensions is difficult (Reinert et al. 2015). Nevertheless, the technique is very cheap, and the cases where a single sampling dimension, e.g., motion blur or direct illumination, is the major noise contributor are surprisingly common.

5.2 Area Light Sources

Accurate illumination from area light sources has been a major selling point of path tracing for production rendering. Such sources are now so ubiquitous that any improvement in their sampling efficiency will benefit virtually every rendering job. For years, Arnold used the traditional approach of combining uniform light surface and BSDF sampling via MIS. We have subsequently developed techniques for uniformly sampling the solid angle subtended by quad- and disk-shaped luminaires (Guillén et al. 2017; Ureña et al. 2013). Apart from bringing valuable noise reduction, the use of these techniques has allowed us to skip BSDF sampling for diffuse surfaces and phase-function sampling for isotropic media, and thereby avoid MIS altogether in such cases for an additional speed-up. We have also adopted the technique of Conty and Kulla (2017) for approximate solid angle sampling of polygon mesh light sources.

5.3 Participating Media

Rendering participating media has always been challenging for production renderers, even when only simulating single scattering, and Arnold has been no exception. To address this, we have developed two techniques for importance sampling the direct illumination from a light source along a given ray (Kulla and Fajardo 2012).

Equiangular sampling chooses a propagation distance along the ray proportionally to the inverse squared energy falloff of a given point on a light source. This technique can substantially reduce variance for rays passing close to the light source.

Decoupled ray marching traverses the medium and builds a table that is subsequently used to importance sample distances along the ray proportionally the product of the transmittance and the medium scattering coefficient. This allows us to decouple the medium and light sampling rates from one another in contrast to the traditional approach (Perlin and Hoffert 1989) and also to ignore medium regions that do not scatter light, i.e., with zero scattering coefficient.

These two techniques importance sample different terms in the single-scattering light contribution and thus complement each other. We therefore combine them via MIS.

Our heterogeneous media implementation relies on ray marching, which is biased (Raab et al. 2008) but can in practice work better than state-of-the-art unbiased tracking techniques (Kutz et al. 2017). First, it samples the medium with a stratified regular pattern, yielding a less noisy transmittance estimate. Second, while increasing the marching step size (to make ray computation cheaper) increases the bias, the variance of a stochastic tracker can explode exponentially when increasing its expected step size. Ray marching thus fails more gracefully. Unfortunately, though, the bias cannot be reduced by simply firing more rays, and the step size has to be hand-tuned per volume as we have no heuristic for doing that automatically. Devising more efficient unbiased techniques is therefore important.

5.4 Subsurface Scattering

Arnold's first subsurface scattering (SSS) implementation was based on diffusion point clouds (Jensen and Buhler 2002), which worked well with few translucent objects that were close to the camera but had several major drawbacks. Every translucent object, and its every instance, required a separate cache. This approach does not scale well: Even with a parallel implementation, the cache construction time could be substantial, especially with many translucent objects far away from the camera. Moreover, the cache density was dependent solely on the SSS profile width, i.e., mean free path, and not on the object's visual importance. Point density had to be hand-tuned by the user as it was not possible for Arnold to automatically deduce shader-mapped profile widths. Too-low density would lead to artifacts, and too-high density could increase memory usage substantially for no visual improvement. An "SSS-in-a-stadium" problem[2] arose on large objects with a narrow profiles, which could only be dealt with by the user painstakingly tweaking the SSS settings to find an acceptable memory-versus-appearance trade-off. The point clouds also increased the complexity of pre-render object processing, were prone to flickering in animation, and suffered from inaccurate motion blur as they were computed for one time instant and assumed constant illumination across motion.

The key to solving these issues was the realization that the SSS illumination blurring integral could also be estimated on-the-fly during rendering. To this end, we developed a method for sampling surface points in the vicinity of a given shading point by importance sampling the SSS profile (King et al. 2013). This drop-in replacement for point clouds streamlined SSS rendering without changing the look. It solved the SSS-in-a-stadium problem by allowing arbitrary profile widths and only paying for SSS computations for rays that actually hit translucent objects. It also made it possible to render dynamic scenes made entirely of such objects, as showcased in Figure 8. We currently use the SSS profile of Christensen and Burley (2015).

The simple diffusion-based illumination blurring approach suffers form lack of energy conservation when the SSS profile is large

[2]This is a pun on the notorious "teapot-in-a-stadium" problem mentioned in Section 3.4.

Fig. 8. Everything in this commercial is made of milk, which would not have been possible with our previous point-cloud based SSS solution due to the low-frequency noise introduced by the mesh topology changing per frame. *Milk ©2013 Czar.be, courtesy of Nozon.*

compared to the size of the geometry features, i.e., in optically thin object regions. This is because SSS profiles are normalized under the assumption of (at least locally) flat surface, which is almost never the case in practice. The issue is exacerbated when paths bounce multiple times off translucent objects, which we are forced to clamp. To address this, we have recently implemented a much more accurate method that performs a volumetric random walk inside the object, very similar to that of Wrenninge et al. (2017). While often slower than diffusion, this method can be faster in the cases where the former struggles most—optically thin objects. We still provide the diffusion-based option for cases where it is faster or easier for artists to derive the look they desire.

5.5 Hair Scattering

Rendering hair is important in modern production but used to be very expensive in Arnold when hairs were semi-transparent, either for artistic reasons or because of the minimum-pixel-width setting (see Section 3.2). To make illumination computations on individual hairs cheaper, we previously reduced the number of rays by simulating only direct and one-bounce diffuse indirect illumination. The direct illumination was cached for each hair segment and interpolated along it for indirect rays. This could speed up rendering by over 3×. Unfortunately, caching did not work well for specular illumination, which had to be skipped altogether. As a result, the output was not realistic-looking but rather flat and cartoony.

To cheaply approximate the speculars, users tried various approaches, such as dual-scattering (Zinke et al. 2008). Similar to diffusion-based SSS, such approximations broke energy conservation and could not be applied recursively for multi-bounce lighting. Achieving a realistic look was difficult, especially in light-colored hair. The diffuse cache also suffered from the same issues as SSS point clouds: additional memory usage, animation flickering, inaccurate motion blur, and increased start-up delay in progressive rendering. We had to find a way to remove the cache and afford tracing large numbers of rays through hair efficiently.

The key to making hair rays cheaper was the addition of stochastic transparency, which avoids the shading and lighting of every single transparent hair along a ray by randomly treating hairs as fully opaque or fully transparent based on their opacity. This technique increases noise but also efficiency, allowing for a greater

Fig. 9. An illustration of varying the hair darkness by changing the total amount of melanin (top row) and the hair redness by changing the ratio of pheomelanin to eumelanin (bottom row). Note that lighter hair is more expensive to render as it requires simulating more light bounces.

sample budget. It allowed us to remove the hair cache and to simulate more light bounces. This in turn enabled the adoption of modern realistic hair scattering models (d'Eon et al. 2011; Zinke and Weber 2007). These allow for accurate rendering of blond hair, which requires many specular bounces (Figure 9, top left). In practice, users are not always after realism, however, and a cartoony look or simpler animal fur is sometimes desired, for which we still provide a cheaper legacy hybrid model (Kajiya and Kay 1989; Marschner et al. 2003).

6 COLOR AND OUTPUT

To fit various imaging and post-production needs, Arnold provides color management control over the input, output, and rendering stages, as well as customizable render data outputs.

6.1 Color Spaces

Adding effects such as physical sky illumination, blackbody radiation, melanin absorption (Figure 9), or dispersion to an RGB renderer like Arnold requires converting between spectral distributions and RGB coefficients. Such conversion requires knowing the chromaticities of the rendering RGB color space (Fairchild 2005). We recently enabled users to specify these chromaticities and also added input and output image color conversion via SynColor and OpenColorIO (2010). This in turn also allowed users with specific color needs to work in the space of their choice.

The recent adoption of wide-gamut formats highlights a deficiency of the RGB rendering done by Arnold compared to true spectral rendering. Light transport is spectral in nature, and approximations like multiplying an RGB albedo with RGB irradiance are typically acceptable in narrow-gamut spaces like sRGB. However, for higher-saturation colors in wide-gamut spaces such as ACEScg (Duiker et al. 2015) or Rec. 2020 (Sugawara et al. 2014), these approximations yield increasingly inaccurate results and color shifts (Meng et al. 2015; Ward and Eydelberg-Vileshin 2002). Thus far, users working in wide-gamut spaces have been able to compensate for such shifts during look development and lighting.

Fig. 10. A scene rendered using 64 camera rays per pixel, with one deep surface sample and multiple ray-marched deep volume samples per ray. With the default quality setting, our deep compression method reduces the storage for every pixel to between 0.5% and 7.5% of the samples produced by the renderer, as visualized in the false-color inset image. The size of the 960×540 32-bit-per-channel deep EXR file reduces from 3.92GB to 153MB (4% of the raw input). The flat "beauty" EXR output is 8.1MB large.

Ultimately, though, physically accurate results require spectral rendering.

6.2 Deep Output

Apart from "beauty" images, Arnold supports rendering arbitrary output variables (AOVs), i.e., auxiliary images of partial shading and illumination data used for debugging, denoising, and post-editing. Custom AOVs can be produced via programmable shaders and light path expressions (Gritz et al. 2010; Heckbert 1990). Rendering 30+ AOVs per frame is not uncommon in production, and Arnold is optimized to efficiently accumulate an arbitrary number of them during path tracing. Post-render tasks such as compositing and depth-of-field effects also benefit from so-called "deep" AOVs, i.e., with multiple values per pixel at different distances from the camera. Minimizing disk footprint is then important as the size of a single raw deep EXR file (OpenEXR 2014) can easily go over 100GB.

To keep deep AOV disk usage manageable, the number of stored samples for every pixel can be compressed (Lokovic and Veach 2000). Our method greedily merges pairs of samples with similar depths based on a user-set error tolerance. This gives best results when the entire set of input samples is available at once, after finishing the pixel. With many AOVs and high sampling rates, the scheme is prone to high memory usage. To avoid caching all input volumetric samples, we compress them on-the-fly earlier, during ray marching. The resulting output file size scales strongly sublinearly with the number of input samples per pixel and depends mostly on the depth complexity of the visible parts of the scene, as illustrated in Figure 10.

7 CONCLUSIONS

We have found that there is no panacea for rendering performance—it is the sum of many optimizations and user workflow improvements that have made path tracing a practical and robust solution for production. We have presented in this article a snapshot of the techniques we use to achieve this.

Key strategies have been to improve user productivity by eliminating pipeline steps, intermediate caches, and extraneous controls. This enables us to optimize performance without introducing complexity to users. We judiciously trade performance for usability improvements and streamlined predictability in both memory and time usage. We seek to minimize the time to present the first pixels to the user during interactive rendering sessions, rather than focus solely on final-frame performance.

Scalability to many cores is an area in which Arnold excels, whether tracing rays, calculating shading, or sampling textures or volumes. Our goal, largely met, has been to scale linearly with the number of cores no matter the type of visual complexity being generated. As hardware becomes faster and capable of tracing more rays per second, and as new algorithms emerge, we expect that Arnold will evolve to become even simpler to use, along with delivering higher quality and interactivity.

ACKNOWLEDGMENTS

A number of people have contributed to the improvement of Arnold in addition to the authors of this article. In particular, we are grateful for the contributions from Solid Angle team members past and present: Luis Armengol, Sebastién Blaineau-Ortega, Stephen Blair, Ivan DeWolf, Ben Fischler, Pedro Gómez, Lee Griggs, Örn Gunnarsson, Julian Hodgson, Peter Horvath, Stefano Jannuzzo, Anders Langlands, Jorge Louzao, Pál Mezei, Borja Morales, Jamie Portsmouth, Yannick Puech, Ashley Retallack, and Xo Wang. We also thank key members of the Autodesk team: Håkan "Zap" Andersson, Cynthia Beauchemin, Eric Bourque, Niklas Harrysson, and Patrick Hodoul.

We thank contributing members from the sister Arnold team at Sony Pictures Imageworks: Solomon Boulos, Alejandro Conty, Larry Gritz, Chris Kulla, Rene Limberger, and Clifford Stein.

We thank the editors of the journal, especially Matt Pharr, for the detailed and constructive feedback on the drafts of this article.

We express our appreciation for the many past and present studios and individual users of Arnold for their feedback and amazing work. There are simply too many to list, but user contributions and advocacy have been integral to the success of Arnold.

The scene from Figure 4 is from http://amscenes.com/16-bath.html. The tank model from Figure 10 is courtesy of Andreas Bystrom.

REFERENCES

Attila T. Áfra, Carsten Benthin, Ingo Wald, and Jacob Munkberg. 2016. Local shading coherence extraction for SIMD-efficient path tracing on CPUs. In *Proceedings of the Conference on High Performance Graphics*. 119–128.

James Arvo and David Kirk. 1990. Particle transport and image synthesis. *SIGGRAPH Comput. Graph.* 24, 4 (1990), 63–66.

Laurent Belcour, Ling-Qi Yan, Ravi Ramamoorthi, and Derek Nowrouzezahrai. 2017. Antialiasing complex global illumination effects in path-space. *ACM Trans. Graph.* 36, 1 (2017), 9:1–9:13.

Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. 2007. Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007*. 177–184.

Brian Budge, Tony Bernardin, Jeff A. Stuart, Shubhabrata Sengupta, Kenneth I. Joy, and John D. Owens. 2009. Out-of-core data management for path tracing on hybrid resources. *Comput. Graph. Forum* 28, 2 (2009), 385–396.

Per H. Christensen and Brent Burley. 2015. *Approximate Reflectance Profiles for Efficient Subsurface Scattering*. Pixar Technical Memo #15.04. Pixar.

P. H. Christensen, J. Fong, D. M. Laur, and D. Batali. 2006. Ray tracing for the movie *Cars*. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 1–6.

Per H. Christensen, David M. Laur, Julia Fong, Wayne L. Wooten, and Dana Batali. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Comput. Graph. Forum* 22, 3 (2003), 543–552.

Alejandro Conty and Christopher Kulla. 2017. Importance sampling of many lights with adaptive tree splitting. In *Proceedings of the ACM SIGGRAPH 2017 Talks*. 33:1–33:2.
Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes image rendering architecture. *SIGGRAPH Comput. Graph.* 21, 4 (1987), 95–102.
Robert L. Cook, John Halstead, Maxwell Planck, and David Ryu. 2007. Stochastic simplification of aggregate detail. *ACM Trans. Graph.* 26, 3 (2007), Article 79.
R. Cranley and T. N. L. Patterson. 1976. Randomization of number theoretic methods for multiple integration. *SIAM J. Numer. Anal.* 13, 6 (1976), 904–914.
Eugene d'Eon, Guillaume Francois, Martin Hill, Joe Letteri, and Jean-Marie Aubry. 2011. An energy-conserving hair reflectance model. *Comput. Graph. Forum* 30, 4 (2011), 1181–1187.
Haarm-Pieter Duiker, Alexander Forsythe, Scott Dyer, Ray Feeney, Will McCown, Jim Houston, Andy Maltz, and Doug Walker. 2015. ACEScg: A common color encoding for visual effects applications. In *Proceedings of the 2015 Symposium on Digital Production*. 53.
Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted deferred shading for production path tracing. *Comput. Graph. Forum* 32, 4 (2013), 125–132.
Thomas Engelhardt and Carsten Dachsbacher. 2008. Octahedron environment maps. In *Proceedings of the Vision, Modeling, and Visualization Conference*. 383–388.
M. D. Fairchild. 2005. *Color Appearance Models*. Wiley.
Iliyan Georgiev and Marcos Fajardo. 2016. Blue-noise dithered sampling. In *Proceedings of the ACM SIGGRAPH 2016 Talks*. 35:1–35:1.
Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. 2012. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.* 31, 6 (2012), 192:1–192:10.
Larry Gritz, Clifford Stein, Chris Kulla, and Alejandro Conty. 2010. Open Shading Language. In *Proceedings of the ACM SIGGRAPH 2010 Talks*. 33:1.
Ibón Guillén, Carlos Ureña, Alan King, Marcos Fajardo, Iliyan Georgiev, Jorge López-Moreno, and Adrian Jarabo. 2017. Area-preserving parameterizations for spherical ellipses. *Comput. Graph. Forum* 36, 4 (2017), 179–187.
Toshiya Hachisuka, Anton S. Kaplanyan, and Carsten Dachsbacher. 2014. Multiplexed metropolis light transport. *ACM Trans. Graph.* 33, 4 (2014), 100:1–100:10.
Toshiya Hachisuka, Jacopo Pantaleoni, and Henrik Wann Jensen. 2012. A path space extension for robust light transport simulation. *ACM Trans. Graph.* 31, 6 (2012), 191:1–191:10.
Paul S. Heckbert. 1990. Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph.* 24, 4 (1990), 145–154.
Homan Igehy. 1999. Tracing ray differentials. In *Proceedings of SIGGRAPH 1999*. 179–186.
Thiago Ize. 2013. Robust BVH ray traversal. *J. Comput. Graph. Techn.* 2, 2 (2013), 12–27.
Wenzel Jakob and Steve Marschner. 2012. Manifold exploration: A Markov chain Monte Carlo technique for rendering scenes with difficult specular transport. *ACM Trans. Graph.* 31, 4 (2012), 58:1–58:13.
Henrik Wann Jensen. 2001. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd.
Henrik Wann Jensen and Juan Buhler. 2002. A rapid hierarchical rendering technique for translucent materials. *ACM Trans. Graph.* 21, 3 (2002), 576–581.
James T. Kajiya. 1986. The rendering equation. *SIGGRAPH Comput. Graph.* 20, 4 (1986), 143–150.
James T. Kajiya and Timothy L. Kay. 1989. Rendering fur with three dimensional textures. *SIGGRAPH Comput. Graph.* 23, 3 (1989), 271–280.
Anton S. Kaplanyan and Carsten Dachsbacher. 2013. Path space regularization for holistic and robust light transport. *Comput. Graph. Forum* 32, 2 (2013), 63–72.
Andrew Kensler. 2013. *Correlated Multi-Jittered Sampling*. Technical Memo #13-01. Pixar.
Doyub Kim and Hyeong-Seok Ko. 2007. Eulerian motion blur. In *Proceedings of the 3rd Eurographics Conference on Natural Phenomena*. 39–46.
Alan King, Christopher D. Kulla, Alejandro Conty, and Marcos Fajardo. 2013. BSSRDF importance sampling. In *Proceedings of the ACM SIGGRAPH 2013 Talks*. 48:1.
Christopher Kulla and Marcos Fajardo. 2012. Importance sampling techniques for path tracing in participating media. *Comput. Graph. Forum* 31, 4 (2012), 1519–1528.
Peter Kutz, Ralf Habel, Yining Karl Li, and Jan Novák. 2017. Spectral and decomposition tracking for rendering heterogeneous volumes. *ACM Trans. Graph.* 36, 4 (2017), 111:1–111:16.
Anders Langlands. 2015. alShaders. Retrieved from http://www.anderslanglands.com.
Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized production path tracing. In *Proceedings of the Conference on High Performance Graphics*. 10:1–10:11.
Tom Lokovic and Eric Veach. 2000. Deep shadow maps. In *Proceedings of SIGGRAPH 2000*. 385–392.
Stephen R. Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, and Pat Hanrahan. 2003. Light scattering from human hair fibers. *ACM Trans. Graph.* 22, 3 (2003), 780–791.
Johannes Meng, Florian Simon, Johannes Hanika, and Carsten Dachsbacher. 2015. Physically meaningful rendering using tristimulus colours. In *Proceedings of the 26th Eurographics Symposium on Rendering*. 31–40.
Thomas Müller, Markus Gross, and Jan Novák. 2017. Practical path guiding for efficient light-transport simulation. *Comput. Graph. Forum* 36, 4 (2017), 91–100.
Ken Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.* 32, 3 (2013), 27:1–27:22.
Koji Nakamaru and Yoshio Ohno. 2002. Ray tracing for curves primitive. *J. WSCG* 10, 1–2 (2002).
OIIO. 2008. OpenImageIO: A Library for Reading and Writing Images. Retrieved from http://openimageio.org.
OpenColorIO. 2010. Home Page. Retrieved from http://opencolorio.org.
OpenEXR. 2014. Home Page. Retrieved from http://www.openexr.com.
OpenSubdiv. 2017. Home Page. Retrieved from http://graphics.pixar.com/opensubdiv.
Steven G. Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian E. Smits, and Charles D. Hansen. 1999. Interactive ray tracing. In *Proceedings of the Conference on Interactive 3D Graphics*. 119–126.
K. Perlin and E. M. Hoffert. 1989. Hypertexture. *SIGGRAPH Comput. Graph.* 23, 3 (1989), 253–262.
Matt Pharr. 2017. The Implementation of a Scalable Texture Cache. Retrieved from http://www.pbrt.org/texcache.pdf.
Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering: From Theory to Implementation* (3rd ed.). Morgan Kaufmann.
Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of SIGGRAPH 1997*. 101–108.
Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. 2009. Object partitioning considered harmful: Space subdivision for BVHs. In *Proceedings of the Conference on High Performance Graphics*. 15–22.
Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. 2006. Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. 89–94.
Matthias Raab, Daniel Seibert, and Alexander Keller. 2008. Unbiased global illumination with participating media. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Springer, 591–606.
Bernhard Reinert, Tobias Ritschel, Hans-Peter Seidel, and Iliyan Georgiev. 2015. Projective blue-noise sampling. *Comput. Graph. Forum* 35, 1 (2015), 285–295.
Alexander Reshetov, Alexei Soupikov, and Jim Hurley. 2005. Multi-level ray tracing algorithm. *ACM Trans. Graph.* 24, 3 (2005), 1176–1185.
Myungbae Son and Sung-Eui Yoon. 2017. Timeline scheduling for out-of-core ray batching. In *Proceedings of the Conference on High Performance Graphics*. 11:1–11:10.
Martin Stich, Heiko Friedrich, and Andreas Dietrich. 2009. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics*. 7–13.
M. Sugawara, S. Y. Choi, and D. Wood. 2014. Ultra-high-definition television (Rec. ITU-R BT.2020): A generational leap in the evolution of television. *IEEE Signal Process. Mag.* 31, 3 (2014), 170–174.
Frank Suykens and Yves Willems. 2001. Path differentials and applications. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*. 257–268.
Carlos Ureña, Marcos Fajardo, and Alan King. 2013. An area-preserving parametrization for spherical rectangles. *Comput. Graph. Forum* 32, 4 (2013), 59–66.
Eric Veach and Leonidas J. Guibas. 1995. Optimally combining sampling techniques for Monte Carlo rendering. In *Proceedings of SIGGRAPH 1995*. 419–428.
Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting rid of packets—Efficient SIMD single-ray traversal using multi-branching BVHs. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing*. 49–58.
Ingo Wald, Solomon Boulos, and Peter Shirley. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.* 26, 1 (2007), 1–18.
Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. 2006. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.* 25, 3 (2006), 485–493.
Greg Ward and Elena Eydelberg-Vileshin. 2002. Picture perfect RGB rendering using spectral prefiltering and sharp color primaries. In *Proceedings of the Eurographics Workshop on Rendering*. 117–124.
Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. 1988. A ray tracing solution for diffuse interreflection. *SIGGRAPH Comput. Graph.* 22, 4 (1988), 85–92.
Magnus Wrenninge. 2016. Efficient rendering of volumetric motion blur using temporally unstructured volumes. *J. Comput. Graph. Techn.* 5, 1 (2016), 1–34.
Magnus Wrenninge, Ryusuke Villemin, and Christophe Hery. 2017. *Path Traced Subsurface Scattering Using Anisotropic Phase Functions and Non-Exponential Free Flights*. Technical Memo #17-07. Pixar.
Arno Zinke and Andreas Weber. 2007. Light scattering from filaments. *IEEE Trans. Visual. Comput. Graph.* 13, 2 (2007), 342–356.
Arno Zinke, Cem Yuksel, Andreas Weber, and John Keyser. 2008. Dual scattering approximation for fast multiple scattering in hair. *ACM Trans. Graph.* 27, 3 (2008), 32:1–32:10.

Received December 2017; revised January 2018; accepted January 2018