Sony Pictures Imageworks Arnold

CHRISTOPHER KULLA, Sony Pictures Imageworks
ALEJANDRO CONTY, Sony Pictures Imageworks
CLIFFORD STEIN, Sony Pictures Imageworks
LARRY GRITZ, Sony Pictures Imageworks

Fig. 1. Sony Imageworks has been using path tracing in production for over a decade: (a) *Monster House* (©2006 Columbia Pictures Industries, Inc. All rights reserved); (b) *Men in Black III* (©2012 Columbia Pictures Industries, Inc. All Rights Reserved.) (c) *Smurfs: The Lost Village* (©2017 Columbia Pictures Industries, Inc. and Sony Pictures Animation Inc. All rights reserved.)

Sony Imageworks' implementation of the Arnold renderer is a fork of the commercial product of the same name, which has evolved independently since around 2009. This paper focuses on the design choices that are unique to this version and have tailored the renderer to the specific requirements of film rendering at our studio. We detail our approach to subdivision surface tessellation, hair rendering, sampling and variance reduction techniques, as well as a description of our open source texturing and shading language components. We also discuss some ideas we once implemented but have since discarded to highlight the evolution of the software over the years.

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

General Terms: Graphics, Systems, Rendering

Additional Key Words and Phrases: Ray-Tracing, Rendering, Monte-Carlo, Path-Tracing

ACM Reference format:
Christopher Kulla, Alejandro Conty, Clifford Stein, and Larry Gritz. 2017. Sony Pictures Imageworks Arnold. *ACM Trans. Graph.* 9, 4, Article 39 (March 2017), 18 pages.
DOI: 0000001.0000001

1 INTRODUCTION

1.1 History

Sony Imageworks first experimented with path tracing during the production of the film *Monster House*. The creative vision for the film called for mimicking the look of stop motion miniatures, a look which had already been successfully explored in early shorts rendered by the Arnold renderer [Jensen et al. 2001]. Despite some limitations that were worked around for the film, the simplicity and robustness of path tracing indicated to the studio there was potential to revisit the basic architecture of a production renderer which had not evolved much since the seminal Reyes paper [Cook et al. 1987].

After an initial period of co-development with Solid Angle, we decided to pursue the evolution of the Arnold renderer independently from the commercially available product. This motivation is twofold. The first is simply pragmatic: software development in service of film production must be responsive to tight deadlines (less the film release date than internal deadlines determined by the production schedule). As an example, when we implemented heterogeneous volume rendering we did so knowing that it would be used in a particular sequence of a film (see Figure 1), and had to refine the implementation as artists were making use of our early prototype. Such rapid iteration is particularly problematic when these decisions impact API and ABI compatibility which commercial software must be beholden to on much longer time frames. The second motivation to fork from the commercial project is more strategic: by tailoring the renderer solely towards a single user base, we can make assumptions that may not be valid for all domains or all workflows. By contrast, a commercial product must retain some amount of flexibility to different ways of working and cannot optimize with only a single use case in mind.

While our renderer and its commercial sibling share a common ancestry, the simple fact that development priorities changed the timeline around various code refactors have rendered the codebases structurally incompatible despite many shared concepts.

This paper describes the current state of our system, particularly focusing on areas where we have diverged from the system described in the companion paper [Georgiev et al. 2018]. We will also catalog some of the lessons learned by giving examples of features we *removed* from our system.

1.2 Design Principles

The most important guiding principle in the design of our system has been *simplicity*. For the artist, this means removing as many unnecessary choices as possible, to let them focus on the creative choices they must make. For us as software developers, this has meant intentionally minimizing our feature set to the bare necessities, aggressively removing unused experimental code when it falls out of favor, and minimizing code duplication.

Perhaps as a direct consequence of this goal, we have always focused on ray tracing as the only building block in our system. Because the natural expression of geometric optics leads one to think in terms of rays of light bouncing around the scene, we felt all approximate representations of such effects (shadow maps, reflection maps, etc.) were simply removing the artist from what they really wanted to express.

1.3 System Overview

We divide the architecture of our system into three main areas to highlight how we meet the challenge of modern feature film production rendering: geometry, shading and integration (see Figure 2).

Geometry. The first and most important design choice in our renderer has been to focus on *in-core* ray tracing of massive scenes. The natural expression of path tracing leads to incoherent ray queries that can visit any part of the scene in random order. Observing that the amount of memory available on a modern workstation can comfortably store several hundred million polygons in RAM, we explicitly craft our data structures to represent as much unique detail as possible in as few bytes as possible. We also heavily leverage instancing as its implementation is well suited to ray tracing but also closely matches how we author large environments by re-using existing models. This design choice also has the benefit of keeping the geometric subsystem *orthogonal* to the rest of the system.

Shading. The role of the shading system is to efficiently manage large quantities of texture data (terabytes of texture data per frame is not uncommon) and efficiently execute shading networks that describe the basic material properties to the renderer. Unlike older rendering architectures, shaders are only able to specify the BSDF per shading point rather than be responsible for its evaluation.

Integration. Turning shaded points into pixel colors is the role of the *integrator* which implements the path tracing algorithm itself. While Monte Carlo methods are conceptually simple, judicious application of (multiple) importance sampling for all integrals and careful attention to the properties of sampling patterns are critical to overall efficiency.

2 GEOMETRY

Since the early days of computer graphics, a fair amount of standardization has occurred leading most surfaces to be rendered as subdivision surfaces [Catmull and Clark 1978], most hair and fur as b-splines, and particulate effects as either volumes [Wrenninge 2009] or collection of spheres.

Because we have chosen to focus on in-core tracing of geometry, we put a special emphasis on compressing geometry in memory.

2.1 Subdivision Surfaces

Displaced subdivision surfaces are probably the most common geometric primitive at Sony Imageworks. Therefore, we needed a fast and highly parallel subdivision engine to minimize time-to-first pixel as well as a compact storage representation of highly subdivided surfaces to reduce memory consumption. We opted to focus on tessellation as opposed to native ray tracing of patches for the simple reason that our geometric models are usually quite dense which makes storing higher-order patches *more* expensive than simply storing triangles. Moreover, methods based on higher-order patches typically fail in the presence of displacements requiring extra geometric representations and going against our principle of code simplicity and orthogonality. Our system also focuses on up-front tessellation which greatly simplifies the implementation compared to deferred or on-demand systems that try to interleave tessellation and rendering. We further motivate this choice in Section 7.4.

A number of criteria dictated the approach we take to subdivision. Firstly, the subdivision engine must be able to handle arbitrary n-gons. Even though quads are the primary modeling primitive at the facility, other polygons do creep in to the system for a variety of reasons. Secondly, we chose to ignore support for weight driven creases to maximize interoperability between software packages. We anticipate that as the rules of the OpenSubdiv library become more widely supported, we may need to revisit this decision, but ignoring this requirement significantly simplifies both the implementation and the data transport across the production pipeline. While creases can make models slightly more lightweight when used optimally, our modelers are quite adept at explicitly adding tight bevels to mimic the influence of creases without needing special rules in the subdivision code itself. Thirdly, we wanted a patch-based subdivision engine where patches could be processed in parallel independent of their neighbors. With these criteria in mind, we chose to evaluate all top-level regular quads with bicubic b-spline patches, and all "irregular" top-level n-gons with n Gregory Patches (one patch per sub-quad) with Loop's method [Loop et al. 2009]. Patches are processed in parallel independently of each other, and we control the writing of shared vertices by prioritizing based on patch ID number.

Subdivision is parallelized first by treating multiple objects in parallel and then within objects by processing patches in parallel. This is further discussed in Section 6.5.

2.1.1 Adaptive tessellation. By default, objects are tessellated adaptively based on screen-space heuristics.

In the case of non-instanced objects, we compute edge rates so that the following metrics are met: (1) patches with displacement are tessellated until some target number of polygons per pixel is reached – the default is four which roughly equates to a final primitive with a $\frac{1}{2}$ pixel edge length; and (2) patches without displacement are tessellated until a primitive's maximum deviation from the true limit surface, as measured in pixels, is reached. The last default is $\frac{1}{3}$. Faces which lie off-screen undergo no subdivision.

Skipping off-screen faces entirely may seem like a surprising choice, as one can easily envision cases in which off-screen geometry casts prominent shadows that could reveal the approximation. This

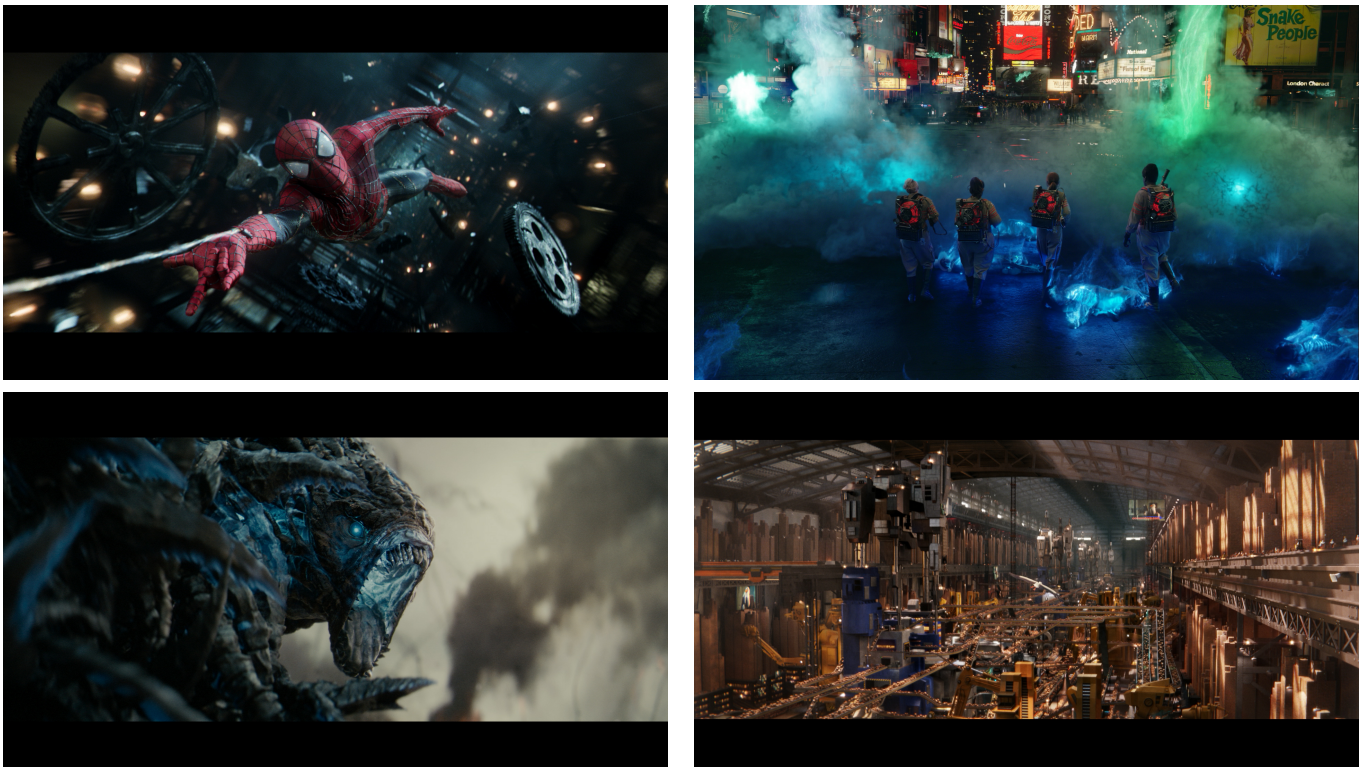

Fig. 2. Still frames from some recent productions. Modern film production rendering frequently involves full frame computer generated imagery of complex environments and characters. Credits from top-left to bottom-right: (a) *The Amazing Spiderman 2* ©Columbia Pictures; (b) *Ghostbusters* ©Columbia Pictures; (c) *Edge of Tomorrow* ©Warner Bros. Pictures; (d) *Storks* ©Warner Animation Group.

is a great example of where the needs of a film production renderer differ from that of a general purpose system serving more industries. At our studio, our input meshes are already quite dense, we rarely use very sharp shadows and set dressing typically will remove as much geometry outside the camera's view as possible to simplify the task of other departments. Taken together, this means that the cases that would cause our approximation to be visible never come up. We should note that we do allow 5% of padding beyond the image resolution before considering a patch off-screen, to prevent slow pans from revealing the transition between detailed tessellation and no tessellation. In the case of rapid camera pans, any transitions are masked by extreme motion blur and we have not experienced any artifacts from this situation either.

Instanced objects pose a particular difficulty to adaptive subdivision. Some instances might lie close to the camera requiring high levels of tessellation, whereas others might lie far in the distance, or even off-screen, requiring little to no subdivision. We attempt to find the "worst case" for each face among all the instances and tessellate it appropriately. The general idea is to estimate the area of a pixel projected into each instance's object-space. This object-level "tolerance" is used to approximate suitable edge-length or limit surface deviation measurements for the displaced, and non-displaced, cases respectively. Because computing this object-space error on a per-face basis for each instance can be a prohibitively expensive operation we use the following simplification which works well in practice. We first set the edge-rates for all faces in the reference object to zero (ie. no tessellation). Then for each instance, we analyze its position in screen-space and process the faces as follows: if the instance lies fully in-camera, then compute an object-level tolerance based on the instance's bounding box and update all the edge-rates if higher levels of tessellation are required than what is currently stored; if the instance lies fully off-camera, then do nothing; if the instance straddles the edge of the screen, then analyze an object-level tolerance for each on-screen face and update accordingly. This per-face analysis can result in significant memory savings as shown in Table 1.

2.1.2 Compact storage. Another goal of our subdivision engine was to find a compact representation for highly-tessellated objects. When tessellating patches, the renderer always outputs $n \times m$ patches of quads, and we are careful to arrange the vertices with an implicit ordering so that no connectivity information is needed. Connectivity is only required for stitching triangles which join patches with different edge-rates. We are careful to store shared edges and vertices exactly once as for most of the scene the tessellation rate per patch can be quite low (1×1 and 2×2 patches are quite common). As the tessellation rates increase in an object, the relative cost of connectivity information decreases. This is illustrated in Table 2. We allow arbitrary edge rates (not just powers of two) which means

Fig. 3. A collection of production test scenes to evaluate tessellation of subdivision surfaces. Credits from top-left to bottom-right: (a) *The Emoji Movie* ©Sony Pictures Animation; (b,c) *The Smurfs Movie* ©Sony Pictures Animation; (d) *Smallfoot* ©Warner Animation Group; (e,f,g) *Alice Through the Looking Glass* ©Disney; (h) *The Meg* ©Warner Bros. Pictures.

Scene	Bounding Box		Per-Patch		
	Mem	Unique tris	Mem	Unique tris	Savings
CG Classroom	3.49GB	82.8M	2.98GB	68.0M	15%
CG Garden	22.2GB	658.5M	4.31GB	123.0M	80%
CG Forest	8.89GB	291.6M	1.14GB	28.1M	87%
CG Hostel	3.84GB	117.9M	3.84GB	117.9M	0%
VFX Gears	173.7GB	6.04B	9.92GB	272.1M	94%
VFX Castle	17.3GB	573.4M	7.41GB	225.1M	57%
VFX Coral Reef	*	*	29.0GB	755.8M	*
VFX Garden	20.7GB	674.6M	11.1GB	330.0M	46%

Table 1. Comparing the results of our per-patch analysis versus bounding box analysis for subdividing instances for a variety of scenes (see Figure 3). The coral reef scene runs out of memory on a 256GB machine using bounding box analysis. The memory field includes all geometric data, connectivity information, per-vertex shading data, and BVH storage.

Scene	Input	Final	Regular Storage		Compact Storage	
	patches	tris	Mem	bytes/tri	Mem.	bytes/tri
CG Classroom	7.5M	68.0M	3.34GB	52.72	2.98GB	47.02
CG Garden	2.2M	123.0M	5.40GB	47.11	4.31GB	37.58
CG Forest	7.4M	28.1M	1.18GB	44.95	1.14GB	43.48
CG Hostel	29.2M	117.9M	4.14GB	37.69	3.84GB	35.01
VFX Gears	43.5M	272.1M	11.38GB	44.93	9.92GB	39.13
VFX Castle	26.8M	225.1M	8.76GB	41.80	7.41GB	35.34
VFX Coral Reef	25.9M	755.8M	35.74GB	50.79	29.0GB	41.24
VFX Garden	56.7M	330.0M	12.84GB	41.79	11.1GB	36.27

Table 2. Comparing a straightforward packing of triangle meshes against our more compact storage of patch tessellations. The exact amount of savings varies depending on the amount of per vertex data needed, the presence of motion blur as well as the exact dicing rates.

that a naive implementation of grid indexing would require modulo and division operations. We optimize this step with multiplicative inverses as these operations remain relatively slow even on modern CPUs. We do not attempt to emulate the fractional tessellation approach supported in some hardware rendering APIs [Moreton 2001]. Fractional tessellation requires rounding to either odd or even rates, which appeared to reduce the adaptivity slightly in our experiments.

2.1.3 Displacement. Displacement occurs during the subdivision process. Interior vertices are displaced a single time whereas shared vertices are displaced n times, where n is the number of faces with which that vertex is shared, and averaged due to the ambiguity introduced by having a vertex attached to different *uniform* (or per-face) user-data. Displacement shading differentials, such as $\frac{dP}{du}$ and $\frac{dP}{dv}$, are determined by the tessellation rates so that texture look-ups are filtered appropriately.

2.2 Curves

The next most common geometric primitive is for hair and fur. We represent these as b-spline cubic curves. The b-spline basis is particularly advantageous because most vertices are shared between segments when producing long strands. Therefore a curve of n cubic segments only requires $n + 3$ vertices. In contrast other equivalent bases like beziers require at least $3n + 1$ vertices. This simple observation already dramatically reduces the memory requirements compared to more naive implementations.

Our initial implementation of the ray intersection test for curves followed the work of Nakamaru and Ohno [2002]. However, motivated by the typical segment densities we observed in practice, we have found that hardcoding the division of segments into 8 linear pieces is faster, is more amenable to vectorization and does not introduce any visual artifacts [Woop et al. 2014].

Despite this optimization, the intersection test for curves has a much longer setup than for triangles. Hair strands can also frequently end up oriented diagonally, in which case a simple axis aligned bounding box is not sufficient to isolate segments in the BVH, forcing even more primitive tests to be done. This issue can be remedied by extending the classic axis aligned BVH with oriented

bounds [Woop et al. 2014], but this requires a substantially more complex build process, and more code only for a single primitive type. We instead use a much simpler solution, which is to simply pack oriented bounds in the leaves of a regular BVH and test several of them in parallel. This test is comparable in cost to traversing one level of a BVH, and actually allows *reducing* the depth of the BVH as it becomes more feasible to intersect multiple curves at once. Since curves are typically quite thin, in practice only one or two segments per BVH leaf will need to be fully intersection tested, with most segments getting rejected early.

From a historical perspective, efficient ray tracing of curves was one of the key advancements that convinced us path tracing was viable for film production.

2.3 Particles

Particles are most frequently used to represent media that isn't quite dense enough or slightly too coarse to be treated as a volume, such as sand, snow or sparks. Ray tracing of spheres is very well understood, however to support efficient ray tracing of many millions of spheres we use a similar mechanism to the curves and pack the particles directly into BVH leaves in small clusters so they can be intersection tested with vector instructions.

Because particles are typically very small (close to subpixel in size), we had to pay special attention to the numerical properties of the sphere intersection test. Simply using a robust quadratic solver [Goldberg 1991] is not sufficient because the precision of the polynomial coefficients themselves is an issue, particularly the constant term $c = \Delta P^2 - r^2$ which is subject to catastrophic cancellation when $\Delta P^2 \gg r^2$ (the sphere is distant relative to its radius).

Accurate shading of emissive particles (common for sparks or fly-away embers) has required us to extend our shading data to support motion blur. Each particle will have changing temperature data (or simply a pre-determined color) attached on every sub-frame. For visually pleasing motion blurred trails, this data must be interpolated on every shade according to the current ray time. Prior to this, artists would frequently convert said curves into small line segments which did not produce the desired effect when combined with camera motion blur or non-uniform shutters.

2.4 Volumes

Finally, volumes are represented as sparse two level grids [Wrenninge 2009]. We opted for this representation over the more sophisticated OpenVDB structure for ease of implementation and also because we found that for dense simulations, the extra sparsity afforded by OpenVDB does not pay off. Deformation motion blur in volumes is supported through velocity fields [Kim and Ko 2007]. We look forward to supporting temporal volumes [Wrenninge 2016], implemented in version 2.0 of the Field3D library, as a more robust solution.

As volumes interact differently than surfaces with light, it is important to capture the *interval* along the ray rather than a single intersection point. While there are stochastic intersection schemes that can unify the processing of surfaces and volumes [Morley et al. 2006], doing so *couples* the ray intersection logic with the light integration strategy. By deferring decisions about where to sample volumes to later stages, we retain flexibility over the types of sampling techniques we can implement. That being said, we believe the need to efficiently handle higher-order bounces [Kutz. et al. 2017] may require us to reevaluate this design decision in the near future.

2.5 Instancing

All geometric primitives in the renderer can be instanced to quickly replicate identical models. This feature is heavily utilized at our studio and is an integral part of how we can efficiently model very large environments through re-use of simple components.

Implementing instancing in a ray tracer is straightforward and memory efficient as only rays need to be transformed on the fly rather than triangles. Aside from the special considerations to adaptive tessellation described in Section 2.1.1, the only perhaps surprising design decision we have made is to focus exclusively on *single-level* instancing. This means that in our renderer, an instance may only refer to a single model, never a collection of models. This decision is directly related to how scenes are organized within our pipeline. We found it was very common for our scenegraphs to be structured into logical rather than spatial groupings that, if reflected in the acceleration structures, led to poor performance. By forcing the entire scene to be flattened to two levels, we greatly reduce these performance pitfalls and also greatly simplify the implementation.

We note that multi-level instancing is obviously more efficient in some scenarios. For example an object comprised of 100 meshes will require creating 100 instances each time it needs to be duplicated in our system. So far, we have been able to live with this overhead by minimizing the size of a single instance.

2.6 Acceleration Structures

Our system relies exclusively on bounding volume hierarchies [Kay and Kajiya 1986] for accelerating ray intersection tests among many primitives. Several factors motivate this choice:

- ease of implementation: construction and traversal are conceptually very simple, meaning that even a high performance implementation can still be easily understood
- easy to extend to motion blur: our implementation has a unified builder for both static and motion blur cases
- predictable memory usage: because each primitive is referred to only once, the storage cost is linear
- each primitive is only tested once: this greatly simplifies the handling of transparency in shadows because shading operations can be performed immediately during traversal

The later point is worth elaborating on. While BVHs have been extended to include spatial splits [Stich et al. 2009], we have explicitly chosen *not* to adopt this technique to retain the property that each primitive is stored exactly once. Because our scenes are mostly composed of subpixel sized triangles, we have not found the splitting of large triangles to be an important requirement for good performance.

The construction of the BVHs for the scene happens in parallel before rendering begins. Similar to subdivision, we start by building BVHs for different objects in parallel before letting multiple threads join a single build process for leftover objects. This is further detailed in Section 6.5.

The biggest weakness of BVHs are that they can degenerate to the performance of a linear list in situations where objects are highly overlapped. We do occasionally run into this problem. In one extreme example, a degenerate simulation collapsed a portion of a heavy triangle mesh in such a way that several million triangles were jumbled into a tiny region of space, leading a single pixel being more expensive than rest of the frame. A similar effect is sometimes seen at the object level of the hierarchy, though usually to a much lesser extent. We look forward to experimenting with re-braiding as a solution to this problem [Benthin et al. 2017].

2.7 Motion Blur

As mentioned above, BVHs are one of the few acceleration structures that naturally extend to motion blur. Instead of a single bounding box per level in the tree, we store a pair of boxes each bounding an extremity of the linear motion path. Interpolating this pair during traversal produces tight bounds which leads to efficient traversal quite comparable to tracing through non-motion blurred geometry. For multi-segment motion blur we simply create multiple trees, each covering a consecutive set of keys.

For most animated feature films we perform motion blur "backwards" (from the previous frame to the current frame) resulting in mostly single segment blur. However in the case of live action visual effects, matchmoved geometry is keyed from the current frame, making "centered" motion blur match more naturally to the plate. This results in 3 keys of motion (or two segments) corresponding to the previous, current and next frames. Because this is a very common case at our studio, we have also specialized this case to only require a single tree. This saves a substantial amount of memory and acceleration structure build time compared to building a tree per segment.

Recent work [Woop et al. 2017] has focused on optimizing motion blurred acceleration structures beyond the simple cases discussed so far. This is a very promising area of future improvement. When a single object requires a large number of motion segments, we must conservatively either increase the number of keys of the top level acceleration structure, or revert to non-motion blurred acceleration structures where a single box encompasses all times. So far, however, the relative rarity of multi-segment blur has not made this a high priority.

3 SHADING

Shading is computation of the material properties of each point on a surface or light source, or in a volume. In our conception, it does not encompass the gathering and integration of light, but merely the description of the scattering properties of the surface, which may include any patterning that causes the scattering or emission properties of a surface or volume to be spatially varying. Displacement shading [Cook 1984] is also supported for modification of the shape of geometric primitives (see Section 2.1.3), but the remainder of this section will focus on shading to determine material properties.

3.1 Shading Language: OSL

Rather than hardcoding a strictly built-in set of functionality, our renderer supports user-extensible, programmable shading. We designed and implemented our own domain-specific language for this purpose: Open Shading Language, or OSL [Gritz et al. 2010]. OSL is syntactically related to prior shading languages such as the RenderMan Shading Language [Hanrahan and Lawson 1990] and its many descendants, but makes a number of design improvements to meet the specific needs of a modern, physically-based ray tracer.

3.1.1 Shader groups. OSL materials are *shader groups* in the form of a directed acyclic graph. Breaking up materials into smaller chunks promotes re-use of carefully written components, and lets end-users compose and extend basic networks in a very flexible manner. The nodes of the graph are individual separately-compiled shaders, and the edges are connections from the outputs of shader nodes to input parameters of other shader nodes. A shader input that is not connected to an upstream layer may instead have a runtime-determined instance parameter value, or be an interpolated geometric variable (such as vertex colors or normals or a positional lookup of a volumetric field texture), or a default determined by the shader. The individual shader nodes are authored and compiled ahead of time, but the full specification of the shader graph that describes a material — including the *instance parameter values* and edge connections of the graph — may be specified at runtime.

The easy composability of the shader nodes leads to material networks that can be extremely complex in practice, often totaling into the dozens to hundreds of shader nodes per material, and there may be hundreds to thousands of materials in a scene.

3.1.2 Runtime optimization. To tame this complexity, OSL implements an aggressive runtime optimization after the shader graph is fully specified. Full knowledge of the instance values and connections allows for many optimizations that would not be possible when examining the nodes individually prior to rendering. To begin, all parameters taking instance values are from this point treated as constants (they will not change during the rendering of the frame), and the shaders undergo a series of compiler optimizations such as constant folding (e.g., $\texttt{Const}_A \times \texttt{Const}_B \Rightarrow \texttt{const}$, $\texttt{A} + 0 \Rightarrow \texttt{A}$), transformations of `if/else` statements with foldable conditions into unconditional execution or eliminating the "dead code," tracking which variables alias each other (e.g., if `A` and `B` can be deduced to hold the same value, even if not a known constant, then `A-B` can be replaced with 0).

Optimizations are propagated across the connections between nodes so that, for example, if an upstream node can be found to store a constant in an output, the separate node that is the recipient of that output can treat its corresponding input as a constant. Dead code elimination also takes into consideration connections — an *unconnected* output (or an output connected to a downstream node that, after optimization, never needs that input) can be eliminated, including all calculations that lead exclusively to computing the unused output. In the course of optimization, some nodes in the graph optimize away completely and are replaced by a direct connection from their upstream to downstream nodes. In the example of Figure 4, a pre-optimized total of 2991 shader groups totaling 280

million instructions and using 161 million symbols (including temporaries) was reduced by runtime optimization to a total of 2.7 million operations and 1.9 million symbols (a 99% reduction). We have found that our complex OSL shader networks execute substantially faster than the equivalent networks expressed as separately-compiled C++ functions.

3.1.3 Just-In-Time Compilation. After the "runtime optimization" described above is completed, we use the LLVM Compiler Framework [Lattner and Adve 2004] for runtime JIT compilation of our optimized shader network into executable machine code for x86_64.[1] After generating LLVM intermediate representation, we also perform various LLVM optimization passes on the code before final JIT compilation to machine code.

The final code that we compile is constructed to implement lazy evaluation of the shader nodes within a material network; execution begins with the "root" node (typically the layer at the end, where the final BSDF closure is assembled), and upstream nodes are executed the first time one of their outputs is actually needed (thus, whole sections of the graph may not execute on every point, depending on dynamic conditions). Additionally, since we are generating the final machine code at runtime and have the opportunity to inject extra code automatically, a number of debugging modes are possible, such as automatically detecting any computation that leads to a NaN value, use of uninitialized variables, or range checking on arrays, with an error message that pinpoints the specific line in the shader source code where the error occurred.

3.2 Derivatives

In order to filter texture lookups (as well as aid in frequency clamping and other anti-aliasing techniques for procedural texturing), it is important to be able to have correct screen-space derivatives of any numeric quantities computed in the shader for texture lookup positions. We use the method of dual arithmetic [Piponi 2004] for automatic differentiation.[2] In summary, a numeric variable carries not only its value v, but also two infinitesimals representing $\partial v/\partial x$ and $\partial v/\partial y$, and all the usual math operators and functions (including everything in OSL's standard library) are overloaded with a version that supports derivatives. OSL performs a data flow analysis to determine the set of symbols that actually need differentials. Only a few operations need to know derivatives (such as the coordinate inputs to `texture()` calls), so only computations that lead ultimately to these "derivative sinks" need to compute derivatives. In practice, this analysis causes only between 5–10% of symbols to carry and compute derivatives, the remainder being ordinary scalar operations. Because this is all handled as part of the code generation at JIT time, they are not part of the OSL shader source code and there is never any need for shader authors to be aware of how derivatives are computed or which variables require them.

Using analytic derivatives is also superior to finite differencing schemes in that they produce accurate answers for discontinuous functions (like absolute value or modulo) and inside conditionals (including loops). For texture lookups, analytic derivatives are particularly advantageous in that a single lookup is sufficient to perform bump mapping. Likewise, expensive functions such as procedural noises can compute their gradient with many fewer computations than finite differencing. For incoherent ray tracing (which does not tend to shade entire tesselated grids at once), trying to compute derivatives via finite differences tends to require inventing and shading "auxiliary" points [Gritz and Hahn 1996] that serve no purpose except for aiding the derivatives.

3.3 Material Closures

Traditional shading systems and languages tended to focus on computing a color value that represented the exitant radiance from a surface in a particular view direction. Computing this concrete value necessitated gathering the incoming radiance from the scene (i.e., sampling, and recursively ray tracing) and integrating, including evaluating the BSDFs. In addition to most of the shading systems and languages typically having a less-than-solid grasp on the units and dimensionality of the quantities involved, they tended to embed the logic for sampling, integration, and ray tracing into the shaders themselves, making it very hard to separate the description of the materials from the implementation of the calculations for light propagation. This made the task of shader authoring unnecessarily complicated, and made it very hard for the renderer authors to alter its algorithms independently of the shader library.

The design of OSL sought to correct this by eschewing the calculation of view-dependent "final colors." Instead, the main output of shaders is a *material closure*. Borrowing the nomenclature from programming language theory, a closure is a functional object, along with any bound contextual state and concrete parameter values, that may be later evaluated, sampled, or examined in other ways by the renderer. A similar design was employed in PBRT [Pharr et al. 2016], where the *material* class returns instances of *BSDF* objects. Our implementation of closures is a weighted linear combination of BSDF lobes.[3] The specific set of primitive BSDF lobes and their implementations is supplied by the renderer.

The two primary things that a renderer may do with a returned closure are:

- **evaluate**: supplying particular view and light vectors, return the result of the BSDF for those directions.
- **sample**: supplying just a single view vector, it can importance-sample the BSDF to choose an appropriate outgoing direction to ray trace or sample a light source.

Because the shader is sending the renderer only the closure, without evaluating it, the shader itself does not trace rays, determine samples, integrate light, or need access to any renderer internals that perform those tasks. With the renderer in control of the sampling, choice of ray tracing strategy, and integration methods, shaders are less cluttered and shader writers unburdened of undue complexity, allowing them to concentrate on more interesting procedural shading tasks. Shader library overhauls (or even recompiles) are not

[1] At the time of publication, there are active projects to extend OSL's LLVM-based code generation to PTX for execution on GPUs, and batch shading aimed at SIMD execution using AVX-512 instruction set.

[2] Our full implementation of automatic differentiation may be found in the `dual.h` header within the OSL open source project, is header-only with few dependencies to other parts of the OSL code base, and can easily be extracted for use in other projects.

[3] OSL's closures are even more flexible than merely holding BSDF combinations, however, and renderers may allow primitive closures that set AOV values, designate the surface as a holdout matte, or trigger other behaviors from the renderer.

Fig. 4. Jungle scene from *Kingsman: Golden Circle*. © 2017, 20th Century Fox. All rights reserved.

necessary to accommodate innovations in sampling, integration, or ray tracing.

3.4 Texturing: OpenImageIO

The amount of external texture required for production scenes can be immense. Figure 4 required 324 billion texture queries, drawing from a set of 8109 texture files containing a total of 174 GB of texture data. This is not considered a very texture-heavy scene; we have seen many scenes that reference 1–2 TB of texture. In addition to handling massive texture data sets with grace, we must ensure high quality texture appearance to meet the needs of production. The texture system implementation is part of the open source OpenImageIO project [Gritz et al. 2008].

Our texture lookups use derivatives computed by the shader to determine anisotropic filtering which is accomplished via a weighted set of individual samples along the major axis of the filter ellipse. Texture samples within each MIP level are usually bilinearly interpolated, but any time a "magnification" is required (including intentionally blurred texture lookups and any time we are sampling from the highest-resolution level of the pyramid), we use bicubic interpolation in order to avoid visible linear artifacts, which are especially visible when using texture to determine displacement. Texture filtering is sped up by utilizing SIMD vector operations using SSE intrinsics, primarily by doing the filter math and sample weighting on all color channels at once (most texture files are 3 or 4 color channels).[4]

Texture Caching. We require textures to be stored in a tiled format, typically divided into 64x64 texel regions that can be read from disk individually, as well as to be MIP-mapped [Williams 1983]. A cache of texture tiles is maintained in memory, with tiles read from disk on demand. When the cache is full, unused tiles are evicted using a "clock" replacement policy. A second cache maintains the open file handles for textures, so that we do not exceed OS limits. In practice, we tend to have an in-memory cache of 1-2 GB for texture tiles, and 1000-10000 (depending on OS) open file handles. With these parameters, in practice we tend to only have around 10% redundant tile reads, thus limiting in-RAM storage of texture to 1%–10% of the full texture set with almost no discernible performance degradation. Because multiple threads may be accessing the texture cache at once, we ensure that the cache accesses are always thread-safe, and the caches are implemented internally as multiple "shards," or sub-caches based on some of the hash bits. The shards lock individually but do not block each other, so two threads accessing the top-level cache simultaneously are highly likely to need different shards and this way neither will wait for the other's lock.

Production and efficiency considerations. The caching efficiency depends on maintaining a certain amount of coherence to the texture accesses, a difficulty given the notoriously incoherent access pattens of ray traced global illumination. We address this problem by tracking the accumulated roughness of rays, and automatically blurring the texture lookups of diffuse and rough rays generally choosing a blur amount that ensures that the entire texture can reside in just a single tile. This ensures that the most incoherent rays tend to touch a minimal set of texture tiles and avoids thrashing the texture cache. It also has the benefit of reducing Monte Carlo noise by heavily pre-filtering the texture used for any shades of the very sparsely sampled diffuse rays.

The nature of production is that rushed artists often create scenes with substantial inefficiencies. Two situations we see frequently are large textures (8192x8192 is standard for hero assets) that are constant-colored, and multiple textures in the scene that are exact pixel-for-pixel duplicates of each other, but nonetheless are separate files. Recall that because we require all renderer input textures to be MIP-mapped and tiled, there is a separate offline step to convert

[4]We are also exploring the use of wider SIMD, such as AVX and AVX-512, by texturing batches of shade points simultaneously.

ordinary images into this form. The tool that performs this task (OpenImageIO's `maketx` utility) checks for constant-valued input, and in such cases outputs a greatly reduced resolution version as well as an annotation in the texture file's metadata, and also for all textures a computed hash of all the input pixel values is stored as metadata. At render time, textures marked as constant can avoid expensive filtering (and taking up space in the cache). Also, when a texture is first opened, its pixel value hash is compared to that of all other textures, and if a duplicate is found, further queries to that texture will be redirected to the first of its identical siblings. It is not unusual for complex scenes to have hundreds, or even thousands, of exact duplicate textures, so this strategy can eliminate a lot of redundant disk I/O.

3.5 BSDF Models

We have historically valued a rather free-form assembly of basic BSDFs by weighted summation. This approached was effective for a long time, but was unable to express structural layering (such as varnishes or clearcoats) where one BSDF affects the others in a view dependent manner. Moreover, it complicated energy conservation as many BSDF models reflect all energy at grazing angles due to Fresnel effects.

We have since transitioned to a more holistic description of surface shading that encapsulated the behavior of the three canonical surface types: conductors, opaque dielectrics (plastic) and transparent dielectrics (glass). This uber shader approach has allowed us to more carefully account for energy conservation (never creating energy from any angle) but also energy preservation (not losing energy unnecessarily). We refer the reader to our recent presentation of these ideas for more details [Hill et al. 2017].

3.6 Medium Tracking

Solid objects with non-opaque interfaces let light travel through their interior medium. This is the case for glass, liquids or shapes with subsurface scattering. A ray tracer needs to be aware of the medium a ray is traveling inside at all times if we want to be physically accurate. We have adopted ideas from nested dielectrics [Schmidt and Budge 2002] to keep track of which medium any given ray is inside of.

Artists assign medium priorities to resolve the conflict between overlapping objects. An ice cube floating on water would have high priority to exclude the water medium from it, for example.

Stacks are light-weight and the dynamic memory management can rely on per-thread memory pools. They are non-mutable small objects shared where needed and they all get released at the end of the random walk, so there is little overhead involved. From this tracking we also get the benefit of computing the right IOR between mediums, which is key for realistic looking underwater objects or liquids inside glass (see Figure 5).

Any conflicts in the medium stack update arising from improper modeling or numerical issues causing missing hits are simply ignored. We have done our best to craft the stack update rules to be as robust to all cases as possible. We refer to our recent presentation on this topic for more details, including how medium tracking interacts with volume primitives [Fong et al. 2017].

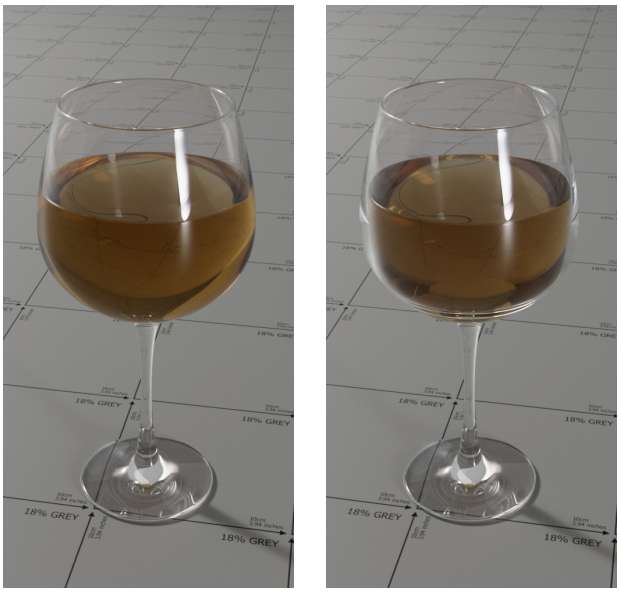

Fig. 5. Rendering of liquids in glass requires medium tracking to obtain a proper refraction event. On the left, the liquid is modeled slightly overlapping the glass, with extra intersection events filtered by our medium tracking system. On the right, the liquid is modeled slightly smaller than the glass and the resulting air gap produces an undesirable refraction.

4 INTEGRATION

4.1 Path Tracing

Arnold mainly uses unidirectional path tracing with next event estimation. This is a Monte Carlo integration technique that randomly traces paths from the camera to the scene, computing connections to the light sources at every intersection.

The simplicity of unidirectional path tracing is of important value from a software engineering point of view, but also because it is easy to explain to end users. It also happens to be the ideal sampling technique for the vast majority of production shots because lights are frequently placed to maximize their direct impact. Strongly indirect scenarios are usually avoided for the same reason they are avoided in real film production: the first bounce of light is simply easier to control. Path tracing does have its weaknesses however, mainly high variance when strong and localized indirect lighting appears (caustics or indoor lighting scenarios).

In practice there are a number of techniques to work around these issues, so these drawbacks are avoided. Filtering and softening caustics, together with intelligent transparent shadows help get the desired look in most cases. So far we have only had to resort to more advanced integration techniques like bidirectional path tracing (BDPT) in very specific scenarios.

The use of OSL shaders allows us to switch from one integrator to another with no asset modification. Unfortunately, the BDPT family of integrators (see Section 4.8) have too much overhead and often shoot too many unnecessary rays. This weakness is common to all applications of multiple importance sampling because they only weight multiple estimators after the fact instead of choosing the

right one a priori. As the number of techniques grows, the overall efficiency becomes lower, particularly if some sampling techniques have negligible contribution. For these reasons, simple path tracing continues to be the technique of choice for our production and research efforts.

4.2 Filter Importance Sampling

There are two popular methods to perform image filtering in a ray tracer. Samples may either be drawn uniformly and splatted to multiple pixels, weighted by the filter kernel, or the filter kernel can be importance sampled directly, warping the distribution of rays within each pixel to fit its profile [Ernst et al. 2006]. The former method has the advantage of reusing rays and therefore reduces variance with fewer samples, but it has a number of disadvantages:

- Pixel values and the integral they come from are not independent. This affects the nature of the noise, lowers the frequency and can confuse a denoising process later on.
- Sampling patterns need to be carefully crafted between pixels to avoid artifacts because samples are shared with neighboring pixels.[5]
- It increases code complexity because splatting across pixels introduces additional data-dependencies that complicate multi-threading.

These reasons motivated us to switch to Filter Importance Sampling (FIS) instead. This turns all image filtering into a simple average and simplifies the parallel implementation by making each pixel fully independent. This choice also makes adaptive sampling much easier to implement (see Section 4.7.5).

While numerical experiments suggested the convergence was worse than splatting, the perceptual nature of the noise was deemed much more pleasant by artists who appreciated the pixel-sized grain much more than the "blurry" noise that splatting can produce. Moreover, the pixel-to-pixel independence greatly improves the effectiveness of denoising algorithms [Sen et al. 2015] making the move to FIS a win overall.

4.3 MIS

While importance sampling of lights and BRDFs is well understood in isolation, sampling their product is much more difficult. We employ multiple importance sampling [Veach 1998] to improve the weighting between multiple estimators. We also apply this technique when sampling complex BSDFs made up of multiple lobes [Szécsi et al. 2003]. Each direction on the hemisphere can usually be generated by several BRDFs, but by weighting the samples according to all BSDF sampling probabilities, we can obtain much more robust estimators.

The math for this combination is very simple, although in practice, special care needs to be taken for singularities. This means perfectly smooth mirrors or point lights where the PDF reaches infinity. Arnold exercises the IEEE float algebra for these values to achieve an efficient implementation with minimal code branching and no special cases for singular BSDFs represented by delta distributions [6].

Beyond these cases, we also apply MIS to related problems of combining estimators for volume sampling [Kulla and Fajardo 2012] and BSSRDF [King et al. 2013] importance sampling.

4.4 Many Lights

Next event estimation in path tracing is a powerful variance reduction technique, but it does introduce the problem of efficiently deciding which lights are most important to perform the next event estimation on. When dealing with scenes with millions of light sources, the mere act of looping over them can be a severe performance overhead.

We extend the concept of acceleration structures for ray tracing to also improve the light selection process [Conty and Kulla 2017]. Starting from the assumption that we would need a BVH for efficiently tracing shadow rays towards scene lights for BSDF samples, we sought to make use of the same structure to also guide light importance sampling. The BVH can be viewed as a light clustering hierarchy where we define an importance measure at each node to decide how to traverse and sample the tree. However, for very big clusters (the upper levels of the tree), a meaningful importance is hard to define. We use a variance based heuristic to decide for each cluster if we should continue traversal into both children (thereby increasing the number of lights selected) or select a child stochastically for the remainder of the tree traversal (guaranteeing a single light will be chosen).

Although our technique cannot bound the number of chosen lights a priori, experience has shown that this greatly accelerates convergence. The subtree selection resembles the Lightcuts technique [Walter et al. 2005], but we make much smaller cuts (typically 1 to 20 lights) and perform unbiased importance sampling on the remainder of the tree instead of discretizing the light sources into points upfront and using cluster representatives.

We also use this technique to deal with *mesh-lights* which we view as collections of triangular sources. In this case, we disable the splitting heuristic so we never choose more than one triangle per sample. Because our scheme takes into account power, distance and orientation, it does not waste any samples on portions of the mesh that are facing away from the point to be shaded.

Introducing this acceleration structure has freed artists to more liberally place lights across environments without worrying about managing shadow ray budgets or having to use tricks such as artificially limiting decay regions. Our technique has the advantage of being relatively memory efficient in that it re-uses a BVH that would likely be needed for ray tracing anyway. On the other hand, it is unable to take into account any information not included in the tree such as occlusion or light blockers leading us to occasionally waste samples on lights that make no contribution. This remains an interesting avenue for future work.

[5] Ernst et al. [2006] claim that FIS produces *less* noise for a given sample budget than splatting samples, but this effect is actually due to the poor pixel-to-pixel correlations of the particular sampling pattern they chose. In our experiments, the convergence of splatting was always numerically superior due to the fact that splatting samples raises the overall sample count per pixel.

[6] The `testrender` program in the OSL codebase contains an example implementation of this approach.

4.5 Volume Importance Sampling

The scattering of light from volumes behaves very similarly to scattering from surfaces with the added problem dimension of choosing the scattering point along the ray before choosing the scattering direction.

We have particularly focused on efficient sampling of *direct* lighting in volumes for which we have developed two novel techniques [Kulla and Fajardo 2012].

To reduce variance from light sources inside participating media, we direct more samples closer to the light via *equiangular* sampling. This has the effect of canceling the $1/r^2$ weak singularity from the lighting integral and substantially reduces noise.

To improve the performance of *heterogeneous* media, we maintain a description of the volumetric properties along the ray in order to be able to quickly evaluate the transmittance to any point and place samples appropriately along it as well. This caching step means we can evaluate volume shaders at a different rate from lighting calculations which we refer to as *decoupled ray-marching* to contrast it with classical ray-marching which evaluated both volume properties and lighting at the same rate [Perlin and Hoffert 1989].

4.6 Subsurface Scattering

We provide two different approaches to render subsurface scattering. The first is a BSSRDF model [King et al. 2013] which simply blurs illumination across the surface and thus ignores the actual paths that light could have taken below the surface. The other is a brute-force Monte Carlo method which leverages our volume sampling infrastructure.

The BSSRDF approach is faster as it only requires a few probe rays to approximate the effect of long scattering paths below the surface. This works well if the mean free path is quite short, but fails at capturing the proper appearance over longer scatter distances as the model makes the assumption the surface is locally flat. As the surface under the scattering radius starts to deviate from this assumption, this model can start gaining energy due to improper normalization.

On the other hand, the volumetric approach computes a much more realistic response, regardless of volume parameters, but has greater cost because it is evaluated as a volumetric effect. To keep the results predictable for users (and reduce the number of special cases in our code) we have fully unified volume rendering and brute force subsurface scattering. This means that placing light sources inside an object will produce the expected result and that subsurface objects can be nested within one another freely. Medium-tracking (see Section 3.6) is essential here to help define volumetric regions from potentially overlapping geometry. Even though this method is volumetric, we still rely on the surface shader run at the entry point to define the volumetric parameters.

4.7 Variance reduction

4.7.1 Path Intensity Clamping. Our first line of defense against unwanted noise is to clamp any contribution above a certain threshold. We apply this to any path that has bounced at least once, and clamp with a lower threshold for high roughness paths (that have a large angular spread) versus low roughness paths (that mostly follow deterministic paths).

Despite its simplicity, this technique can easily reduce the amount of noise in many common scenarios such as light sources against a wall where the hot spot on the wall itself would normally manifest itself as strong indirect noise. Naturally, the overall energy can be much lower if compared to a ground truth solution, but because this optimization is enabled by default, most artists never compare to the true solution. Early work by Rushmeier and Ward [1994] attempted to spread the clamped energy to nearby pixels, but this can lead to low frequency temporal artifacts as small fireflies tend to move around between frames. Recovering this missing energy in a meaningful and temporally stable way is definitely an important avenue for future work.

4.7.2 Roughness Clamping. Kaplanyan et al. [2013] present a modification of the light transport measure to remove singularities that cause certain light transport paths to be impossible to sample in some algorithms. The main observation was that blurring the singularities in the integration domain makes them easier to integrate.

We use a similar technique to smooth out strong caustic contributions, by clamping the roughness at any path vertex to be at least equal to the largest roughness seen along the path so far. This avoids diffuse-glossy or glossy-glossy noise at the expense of smoothing out caustic effects.

4.7.3 Caustic Avoidance. Together with the softened caustics, we use an additional shortcut to let light reach the interior of glass objects. We approximate this effect by ignoring the refraction at the glass boundary for shadow rays, allowing direct lighting to pass through the boundary. For many scenarios such as car windows, bottles or glasses, the thickness of the glass volume is such that the double refraction is well approximated by this simple approximation.

We avoid double counting the contribution by an MIS inspired weighting [Hill et al. 2017] between the two possible paths: the correct one that reaches the light source by indirect sampling, and the approximate one that reaches the light by ignoring refraction.

We rely on this scheme for all cases involving glass such as car interiors, glass bottles but also brute-force subsurface scattering, where getting light to affect the medium through the interface is important to a correct appearance.

4.7.4 Stochastic Path Rendering. There is a special case of multiple hits that happen on a straight line. A ray can cross multiple transparent surfaces and volume segments. Computing light contribution at all these intersections introduces wide splits in the ray tree resulting in an exponential increase in the amount of lighting calculations to be performed. This is in fact the same issue that makes brute force classical ray marching impractical for recursive ray tracing (see Section 4.5).

We found that we can track the contribution decay along the ray and the albedo of all intersected elements to build a CDF for their importance. Random candidates for lighting get chosen from this distribution, allowing us to follow a single path from a ray instead of many, preventing the ray tree from growing exponentially.

In fact, this step can be extended across bounces and we can sample lighting over very long paths sparsely to improve efficiency. We have found that it is worth always sampling lighting on the camera ray (once across multiple transparent hits) while the higher-order bounces can be merged if the total path length becomes large (more than 16 bounces in our current implementation). This is particularly helpful for subsurface scattering which may require many internal bounces that all make similar contributions to the final pixel.

4.7.5 Adaptive Sampling. The noise distribution within the frame is rarely uniform; there are always areas of the image that require more samples than others. To this end, we allow the pixel variance to drive additional sampling to noisier pixels. In this context we use the variance of the *mean* (the pixel color) which is a value that decreases as the number of samples grows, as opposed to the variance of the *samples*. The central limit theorem ensures that in the limit, the distribution of values the mean takes on approaches a normal distribution, making the variance a quantity we can meaningfully reason about.

The render operates in multiple passes over the image. The initial passes serve to establish a baseline for the variance and ensure very small features are not missed. Beyond this point, only pixels exceeding an error threshold receive more samples. The render stops whenever all pixels pass the error threshold or a maximum sampling rate is reached.

Because the variance itself is a random variable, looking at a single pixel in isolation introduces bias. Our solution is to use the maximum error over a small window of pixels to make our decision. While this is still theoretically biased, we have not observed any issues from this in practice.

The error estimate is based on the standard deviation σ and the mean μ. We apply a perceptual tone mapping curve to the values $\mu + \sigma$ and $\mu - \sigma$. This lets us estimate the perceptual rather than absolute error which avoids over-sampling highlights or under-sampling darker regions.

Introducing adaptive sampling to the renderer has generally allowed artists to worry much less about fine-tuning individual sampling controls, however it has had some perhaps surprising workflow implications. For example a user error like a light partially intersecting a wall will introduce high variance that the adaptive sampler will try to compensate for. With a fixed number of samples per pixel these errors appeared as extra variance in the final image which could be diagnosed, while with adaptive sampling the error might only manifest as extra long render times which may be harder to trace back to the source. Another perhaps surprising consequence of the image-level adaptive sampling has been that raising sample counts can sometimes *speed up* a render, by reducing the amount of variance between primary rays and therefore requiring fewer passes overall.

4.8 Advanced Integrators

Since we use view-independent OSL shaders, we have freedom to change the integration algorithm easily. We have implemented basic BDPT [Veach 1998] with recursive MIS weighting [Antwerpen 2011]. But we also have variants using primary space Metropolis light transport [Kelemen et al. 2002], multiplexed MLT [Hachisuka et al. 2014], and Vertex Connection and Merging (VCM) [Georgiev et al. 2012].

These methods allow us to produce complex effects in some very particular shots where they are really required, but also give us the ability to compare our unidirectional path tracing shortcuts discussed in Section 4.7 to the ground truth.

4.8.1 MCM. One of the most difficult type of paths to sample are specular-diffuse-specular paths like in underwater caustics. The VCM algorithm was designed to handle these cases, but it can be further improved by using Metropolis methods to guide the paths. A basic implementation of Metropolis techniques often leads to unstable results in animation due to correlation between paths and the sudden discovery of new features. To mitigate this problem, we combine a number of techniques, including Multi-Stage Metropolis light transport [Hoberock and Hart 2010]. This helps equalize the contributions over the image plane, while still improving the convergence of the underlying light transport method (BDPT or VCM).

We call our hybrid approach Metropolis Connecting and Merging (MCM). In this integrator we store photons in a pass-based fashion like the VCM algorithm, except the photons come from the camera paths instead of light paths (we call them *sensor photons*).

The advantage is that we can apply Metropolis sampling to the light paths, which accumulate not only by connections but also by merges with the sensor photons. This helps greatly focus light particles in the interesting areas, reducing variance from the original VCM algorithm. This is our most robust integrator to date. We briefly note that many other researchers have independently explored the combination of these ideas. The work of Šik et al. [2016] is most conceptually similar to ours, though we have not attempted a direct comparison to their implementation details.

4.8.2 Production usage. MCM (and sometimes our simpler MLT if photon merges are not required) is our alternative integrator of choice for complex lighting situations. But rendering with a separate integrator is far from ideal for the artists. The preference so far has been to generate additional passes isolating a particular effect like caustics to be combined later on in compositing rather than trying to render all passes with the alternative integrator.

Bidirectional integrators are robust but have the disadvantage of considering many techniques that end up being weighted away by MIS. Also, the texture access pattern is much less coherent which can lead to cache thrashing. Unlike with path tracing, where successive bounces can reduce texture filtering fidelity, light paths can connect to the camera at any point and must retain roughly pixel sized ray differentials. Finally, some simple artistic controls such as tagging objects not to cast shadows while still being illuminated by a light are difficult to support in a bidirectional context.

5 SAMPLING PATTERNS

The performance of Monte Carlo sampling depends critically on the spatial properties of the input random numbers. It can be shown, for instance, that stratified sampling is always superior to purely random numbers. Recent research [Subr et al. 2016] has tried to further explore the link between variance and the power spectrum of the input point set. The most directly applicable consequence

of this research is the observation that variance is proportional to the product of the power spectrum of the integrand and sampling pattern. Sampling patterns whose radially averaged power spectrum falls to 0 at low frequencies are much more effective at lowering variance than those that do not, regardless of the actual shape of the integrand's power spectrum (which is unknown but usually contains some low frequency component).

Arnold has historically expressed all sampling controls as perfect squares. When a user requests sampling level n, we use n^2 samples. This facilitates the generation of sample sets that are well stratified, particularly when implementing path *splitting* where an initial set of n^2 camera rays is further refined into m^2 rays each, producing a total of $(nm)^2$ rays for the pixel. By generating a larger pattern for those secondary rays (for instance by multi-jittering [Shirley and Wang 1994]) it is possible to guarantee a good distribution both within the pixel, and within each subset of size m^2.

5.1 Magic Shuffling

While the fast generation of high quality sampling patterns in 2D is well studied, the extension to multi-dimensional settings is harder and must contend with the curse of dimensionality. An alternative approach to using high dimensional points is to re-use shuffled independent realizations of low dimensional patterns. In doing so, one must pay special attention not to introduce unwanted correlation between the dimension pairs that are not directly constructed.

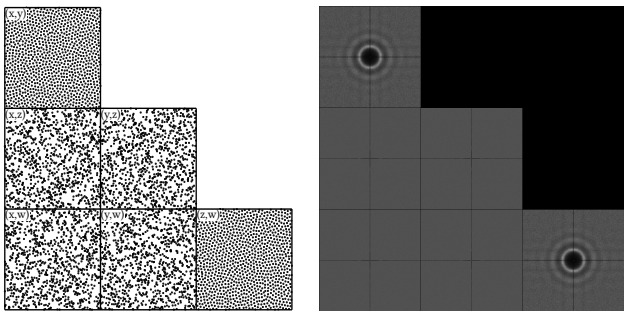

Fig. 6. We visualize two instances of blue noise points (x, y) and (z, w) as might be used for pixel sampling and BSDF sampling for example. We also visualize the resulting cross-dimension sampling patterns, for instance the x coordinate of the subpixel against the z random component for BSDF sampling. Here we randomly shuffle the order of the second set of points, which means the cross dimensions look like completely random point sets.

The simplest way to decorrelate pairs of dimensions is to randomly shuffle the points between them. This has the effect of randomizing the non-primary pairs of dimensions, letting them revert to plain random sampling (see Figure 6).

We have discovered that a small modification to this shuffling step can produce much better results, assuming that the input points were generated on a grid. With this method, the non-primary pairs now revert to stratified sampling, without breaking the properties of the original points (as they are merely enumerated in a different order). We outline the pseudocode of this approach in Figure 7. We have dubbed this technique the "magic" shuffle as we do not have any formal proof of the reason for its success. The basic intuition

```
int magic_shuffle(int i, int n, int seed) {
  // shuffle rows and columns
  int rx = rand_permute(i % n, n, hash(seed, 1));
  int ry = rand_permute(i / n, n, hash(seed, 2));
  // shuffle diagonals to stratify
  int sx = rand_permute((rx + ry) % n, n, hash(seed, 3));
  int sy = rand_permute((ry + sx) % n, n, hash(seed, 4));
  return sy * n + sx;
}
```

Fig. 7. Magic shuffling creates a permutation of a sample index $i \in [0, n^2)$. The seed parameter should change per pixel and with each depth. The `rand_permute` function procedurally indexes a random permutation of length n from a given seed.

that guided us was that we wanted to exchange rows and columns of the input pattern, followed by diagonals of the pattern. To our surprise, the spectral properties of the resulting shuffled points look very much like that of stratified point set, despite the fact that we have not changed the coordinates, only the relative enumeration from one pattern to the other. That being said, only 3 of the 4 possible cross-pattern dimensions become stratified, which is explained by the slight asymmetry in the last step of the shuffling. To date, we have not been able to "fix" this issue, and therefore we need to pay special attention to which pair of dimensions receives the less well shuffled set.

The pseudo-code in Figure 7 can be further generalized to optimize the path splitting case where an input point set of size $(nm)^2$ is partitioned into n^2 well distributed subsets of size m^2 each. It can also be generalized to shuffle more than 2 pairs of 2D samples, resulting in good distributions across more dimensions. Naturally, each time a shuffle is performed, the optimal properties only appear relative to the input index. Therefore we had to pay special attention to which pairs of samples received the more optimal points. Finally, it is worth noting that the shuffling is not necessarily restricted to 2D input points. It can work for specially crafted 3D or higher-order base points that need to be decorrelated from each other, as long as the first two dimensions can be enumerated on a grid.

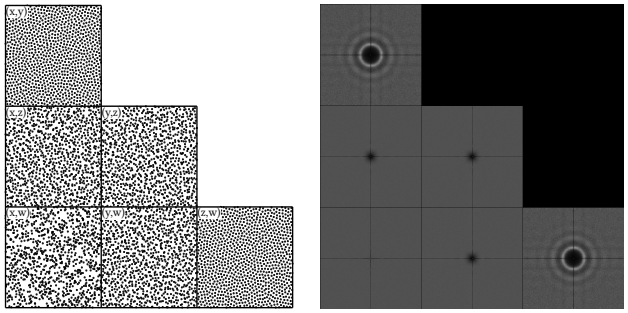

Fig. 8. Our magic shuffling improves the distribution of the other pairs of points by giving them similar properties to stratified points, without changing the two input point sets at all. Only the (x, w) pair remains randomly shuffled.

5.2 Progressive Shuffling

As discussed in Section 4.7.5, we have recently transitioned to rendering in a more continuous way which precludes the upfront optimization of sampling patterns for particular rates.

As such, we have had to abandon the techniques outlined above in favor of progressively enumerable patterns (recent work by Ahmed et al. [2017] presents a particularly elegant construction). We have faced the same challenge as with fixed-rate samplers though, and have found that replicating high quality low-dimensional patterns in higher dimensions is more successful than trying to generate very high dimensional patterns directly.

To ensure there is no correlation between pairs of dimensions, we can no longer rely on a straightforward random shuffle as the number of points is not known. Instead we perform the shuffling *progressively*. We observe that since our sampling controls are still perfect squares, each increment in the global number of samples per pixel provides a larger and larger budget of samples. Therefore we simply shuffle our patterns as follows: when moving from n^2 points to $(n+1)^2$ points, we shuffle the $2n+1$ new samples amongst themselves. Using a different permutation for each depth ensures a good quality scrambling while minimizing the deviation from the overall sequence. As we reach each perfect square marker, we have exhaustively taken all samples from each stream, simply in different orders. We consider each such marker of n^2 samples a *pass* and only present the image to the user on such boundaries.

The combination of FIS and progressive sampling has enabled us to implement adaptive sampling in a very straightforward way. While both changes were slight regressions on their own, the ability to "keep going" on any pixel has been worth it overall.

6 PIPELINE AND TOOLING

Production renderers do not exist in a vacuum. One of the particular advantages of writing an in-house renderer is the ability to tailor its interfaces to match that of the rest of the production pipeline.

6.1 kick

Arnold has a simple command-line frontend tool called `kick` which is used to read in `.ass`[7] scene files and to display the render in a simple X11 window. Over the years, we added numerous debugging features such as the interactive selection of shading modes, controls for interactively moving the camera or lights, overriding of parameters, simple turntable controls, etc. It is also very common for developers to write `.ass` files for fast edit-build-render iterations when debugging. Arnold's regression test suite consists of `.ass` files rendered by `kick`.

The `.ass` file format is intentionally very simple. It consists of a straightforward serialization of the nodes that comprise the scene. We provide an example in Figure 9. To support debugging of very large scenes, we also support a binary version of this file format (`.abs` files[8]) which is even more optimized. We continue to use the plain ASCII flavor for longer term archival of scenes however, as it is more convenient to edit.

[7] A "backronym" standing for Arnold Scene Source.
[8] A "backronym" standing for Arnold Binary Scene

```
persp_camera {
 position 0 0 20
 look_at 0 0 0
 up 0 1 0
 fov 13
}
sphere {
 center -1 0 0
 radius 1.0
 shader red
}
sphere {
 center 1 -0.2 2
 radius 0.8
 shader basic
}
plane  {
 point 0 -1 0
 normal 0 1 0
 shader basic
}

standard { name basic }
standard { name red color 0.5 0.1 0.1 roughness 0.4 }

skydome_light {
  color 0.7 0.8 0.9 affect_camera on
}
physical_distant_light {
  color 0.9 0.7 0.5 direction -1 -2 -1 angle 2 exposure 3
}
```

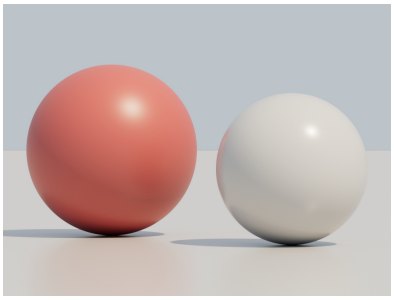

Fig. 9. A sample `.ass` scene description, and the resulting image. The file format directly mimics how the renderer represents objects in memory.

6.2 Katana

Katana is a lighting tool built around a very flexible scenegraph processing architecture which allows lazy processing of very large scenes. Our look development and lighting artists primarily interact with the renderer through this tool.

We take advantage of Katana's flexibility to replace some features that would otherwise belong in the rendering interface. For instance, because Katana natively represents a scenegraph with powerful inheritance mechanisms, the renderer only needs to have a much simpler node-based architecture without any form of inheritance. Also, because Katana can lazily load and procedurally define geometry, we do not need to have procedural geometry creation tools in the renderer itself. This means render time generated geometry like fur, grass or even crowds can be processed on the fly as Katana translates the scene to the renderer. Unlike systems which have opaque render time generation procedurals, the fact that ours are described through the lighting tool itself greatly simplifies the manipulation of said geometry for artists who are able to inspect it directly.

6.3 Live Rendering

Artist productivity is greatly enhanced when working in a "live rendering" environment where attributes such as position and orientation of objects and lights can be freely manipulated without having to restart the render. To facilitate this workflow, we have made a few changes to simplify this interaction paradigm from the API level. The renderer can be launched in a special "server" mode

which accepts all attribute changes from the host application, and schedules their application inside the renderer.

To avoid race conditions such as tracing against partially built acceleration structures, attribute changes submitted to the renderer while a render is in progress are stored in a change queue. This act of writing an attribute into the change queue will trigger a signal to the renderer to abort any in-progress render. Once the render has stopped, all outstanding changes are applied, any affected spatial hierarchies are rebuilt, and the render is restarted. No other special considerations have been made, and the "live" renderer is no different from the final frame renderer. This has allowed artists to heavily rely on this functionality and we make every effort to support a maximum number of scene edits in this mode.

Arnold is robust to nearly all feature changes such as the addition or modification of geometry, lights, and shading networks.

6.4 Checkpointing

There are several scenarios that require the ability to abort renders in progress without losing the progress made so far. For example, migrating jobs from slower to faster cores as dictated by production driven priority changes, or quickly taking machines offline for maintenance or upgrades, even if they are in the middle of rendering frames.

The required data files are simply the full float EXR image files, which can be loaded back in memory without data loss. Some special filters may need some auxiliary data: for example a closest hit filter may need to record the Z value of the current pixel for comparison against future samples. We also keep an image with sample count and variance information for the adaptive sampling convergence measure. Even with dozens of simultaneous image outputs (for example isolating per-light contributions, or geometric data like normals and Z), we can flush all needed data to disk in a couple of seconds. This data is saved between passes of the adaptive sampling code (see Section 4.7.5) which ensures that less than a pass worth of data can be lost. We throttle this rate if the render is going quickly to avoid doing too much I/O as well.

Because our sampling patterns are progressive, we can keep providing new samples from any stopping condition. The same checkpointing feature therefore also allows resuming renders with higher fidelity.

6.5 Threading

While it is generally easy to assign large independent tasks, such as building a BVH or subdividing a mesh, to different threads for coarse-grained parallelism, there is often at least one outlying task that takes significantly longer than any others which causes the remaining threads to idle and until it is complete. To ameliorate this waste, all parallel algorithms in Arnold support *cooperative* multi-threading, where any thread can assist any other thread with its current task. These concepts are illustrated in Figure 10.

Scene preparation is divided into several different stages, such as processing geometry or lights. Associated with each stage is a large queue of tasks requiring attention, such as subdividing objects, building their BVHs, or creating light importance tables. Each thread will take the next available task in the queue and work on it to completion. When there are no more tasks available, threads will make a second pass through the queue and work collaboratively on any remaining "in-flight" tasks.

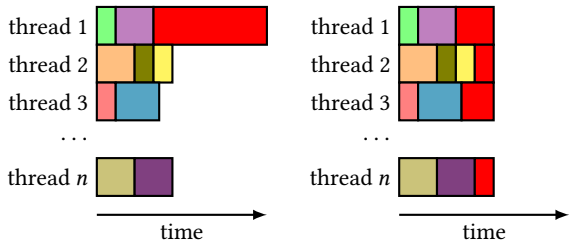

Fig. 10. On the left, we show how parallelism only across high level tasks illustrates how very long tasks can ruin the efficiency of the system. On the right, the outlying task is further worked on cooperatively by all threads which improves efficiency.

7 DISCARDED IDEAS

To retain code simplicity, we do our best to *remove* experiments that have failed or features that have been supplanted by others.

7.1 Hair and SSS Illumination Caching

For many years, we relied on caching techniques to speed up the calculation of subsurface scattering or hair multiple scattering. However, as we discovered improved importance sampling techniques for these cases, the need for such caches was removed.

Our implementation of these types of caches were transparent to the artist. They never required manually setting up passes between jobs and instead were populated on-the-fly during rendering. But they still had a number of drawbacks: they could grow quite large in memory, had subtle artifacts and tuning parameters the artists had to be conscious of. Moreover, compared to proper importance sampling, caching was simply not competitive in speed due to more complicated inter-thread synchronization.

7.2 Batch shading interpreter

In our first working implementation of OSL, the shaders were evaluated in the renderer in classic Reyes style: shading batches of points, interpreting one shader instruction at a time in lock-step, with each instruction executing on all batch points. We even made several improvements over earlier systems, such as *dynamically* allowing variables to change from "uniform" to "varying" and back, depending on what was assigned to them (uniform calculations are done once per batches, versus varying calculations being performed for every point).

But we found that while this batched interpreted approach had worked well for Reyes-style renderers that naturally generated large coherent grids of points to shade with the same material, the performance was inadequate for production rendering in the context of a path tracer that struggled to find enough shading coherence for secondary rays, limiting how much we could amortize interpreter overhead over large batches. The necessity of batches also greatly complicated the ray tracing architecture.

We switched to the simpler single-point-at-a-time shading, and changed from an interpreter to LLVM-based JIT to machine code

(thus, no interpreter overhead), which not only resulted in much faster shader evaluation, but also allowed us to greatly simplify the ray tracing code base by removing the need to trace batches of rays. (Though some of these ideas may yet return, as we continue to explore streaming of rays and batched shading – though this time, utilizing hardware SIMD rather than an interpreter.)

7.3 Light Path Expressions

An interesting fallout of our push to OSL was a rethinking of how shading AOVs[9] should be handled by a modern path tracer. Our facility relies on AOVs for a number of things: decomposing the image into individual components (such as direct and indirect diffuse, specular, sss and volume scattering), breaking up the image by (groups of) light sources, and also computing masks of individual features (individual characters in crowds or shading regions like skin, eyes, etc.). These allow more flexibility in how the image can be manipulated after the render is done in compositing.

Previous shading languages (and our own C API) burdened shader writers with maintaining a coherent array of output colors that split the image into these meaningful components. As the importance of indirect lighting grew, tracking this efficiently throughout all shaders became a major source of complexity. We did not wish to burden the OSL language with said complexity. Instead, drawing inspiration from Paul Heckbert's early work on classifying light paths ([Heckbert 1990]) we designed a regular expression engine that could compactly encode the desired set of AOVs. This *Light Path Expression* (LPE) engine was initially outside of the OSL project and part of the renderer, but we published a description of its capabilities alongside the OSL documentation to suggest a possible way that AOVs could be supported orthogonally to the language. Because the engine constructs a state machine from the user supplied expressions, only 32 bits of state per ray is required, making the runtime cost relatively low. However there were some unexpected consequences that made us ultimately reconsider the use of light path expressions.

The biggest downside was that LPEs constrained some sampling decisions like the ability to merge diffuse and specular lobes at deeper bounces. We also discovered that very few artists (and in some cases, even ourselves) could assemble a coherent set of LPEs that would be guaranteed to add back up to the main output without double counting any contributions. For this reason, we reverted back to a hand-written set of rules, maintained by the renderer. At the moment, all AOV splitting logic resides in a single function, where a chain of `if/else` conditions ensures that no contribution can ever be double-counted.

The core idea of LPEs, on the other hand, proved to be popular, and by community request we ended up including the matching engine as part of the OSL project around the same time we removed support for it from our renderer. To our knowledge, several of the implementations of LPEs in commercial products have directly made use of this code.

[9] "Arbitrary Output Variables" is jargon inherited from the implementation in RenderMan, where extra shading components were simply additional variables output by a shader. Our renderer does not represent these this way, but the acronym has stuck nonetheless.

In retrospect, this small feature ends up being a great case study in the difference between in-house vs. commercial renderers. A commercial renderer cannot envision all possible use cases a customer might face, and must provide very powerful mechanisms to override built-in behavior. On the other hand, in-house renderers cater to a much narrower set of users facing known problems. When looking at the problems that LPEs solve, it turns out that very few really required the full generality they offer, and we therefore benefited from keeping our code simpler.

7.4 Deferred Tessellation

Lazy evaluation of subdivision and tessellation in hopes that we might save time and memory by only subdividing those objects that are hit by rays is a false promise. Our experience has been that when global illumination is being computed, nearly every object in the scene will be seen by at least one ray.

While we did support deferred subdivision and displacement for a long time, we removed this ability for several reasons. During the first few passes, many rays were waiting on the same objects. This meant the slightly less efficient inter-object cooperative parallelism was used when in fact working on multiple objects independently could be more beneficial (see Section 6.5). Also, displaced geometry required some padding to account for the eventual movement of the not-yet displaced vertices. This last point was a particular problem because this padding is very hard to specify optimally: if the bounds were too tight, geometry could appear clipped in the render; and if the padding was too loose, then we would suffer performance degradation. With up-front subdivision and tessellation, we can compute exact bounds for subdivided and displaced geometry prior to building the acceleration structures. Moreover, the resulting code is simpler and more *orthogonal*.

7.5 Geometric Procedurals

As discussed in Section 6.2, geometric procedurals which were once described by a custom plugin type to the renderer itself are now handled by our *lighting tool* where they are much easier to inspect for artists.

Interestingly, the Katana lighting tool itself was heavily based around the idea of lazy expansion through procedurals. Because Katana's architecture is lazy (nothing is computed until requested) it is possible to traverse the scenegraph only a few levels at a time, waiting for rays to intersect the bounds of that region before expanding the scene further.

This workflow turned out to be detrimental to a ray tracer for similar reasons to lazy tessellation. Additionally, artists typically organize the scene for ease of logical navigation, not spatial efficiency. For instance a city scene may group all trees, all stop signs and all benches together. Naively walking the hierarchy would produce pathologically overlapping bounding boxes that would be very inefficient to traverse. By walking the entire scene upfront, we have greatly simplified the renderer implementation as well as the Katana renderer plugin, which no longer needs to retain state about the scene traversal during rendering. Katana's ability to evaluate the scenegraph is still critical however as it ensures we can stream through the scene representation and pass it along to the renderer without having duplicated storage.

8 CONCLUSION

Arnold has been in use at Imageworks for over 13 years. During this time, it has undergone many dramatic changes. This document represents a snapshot of our thinking on a variety of topics, but we do not believe we are "done" with any aspect of the renderer.

As exemplified by the other systems described in this issue, the rendering landscape is quite competitive and our studio continues to invest in pushing the boundaries of image quality and performance. To this end, we continually investigate new hardware architectures (wider SIMD on CPUs, evolving GPU capabilities, etc.). We recently have been pushing beyond Katana as the only rendering interface within our studio and are looking to integrate the renderer more directly into other applications to provide higher fidelity rendering or preview to other departments such as modeling, texture painting, layout and animation. We have found that for dense environments for example, a ray traced view can be faster than hardware accelerated ones (some commercially available applications like Clarrise or Keyshot have made the same observation).

In terms of image quality, we believe the biggest remaining challenge is overcoming the needs for the variance reduction "tricks" described in section 4.7. Despite our implementation and occasional use of bidirectional methods and MCMC techniques, their limitations seem difficult to overcome. We are encouraged by recent advancements in path guiding [Bus and Boubekeur 2017; Müller et al. 2017; Vorba et al. 2014] which suggest that path tracing may be sufficient for accurate rendering of complex light paths, provided the paths can be directed intelligently enough. Our initial experiments in this area are quite promising.

ACKNOWLEDGMENTS

We would like to thank the many members of `arnold-dev` over the years: Scot Shinderman, Rene Limberger and Solomon Boulos as well as Eduardo Bustillo, Hiro Miyoshi, Kishore Mulchandani and Aner Ben-Artzi.

We are indebted to Jesse Andrewartha and the `arnold-help` team for patiently optimizing scenes in the heat of production, evangelizing best practices within the studio, and bringing new artists up to speed.

We would like to thank Marcos Fajardo for allowing us to be a part of the Arnold experience, and we would also like to acknowledge the rest of the team at Solid Angle SL with whom we still discuss and exchange ideas on how we can improve our renderers.

We would also like to thank Francisco de Jesus, Rob Bredow, Barbara Ford, Bill Villareal and George Joblove for the early bets they made on bringing Arnold to the facility and embracing it facility-wide.

Finally, the authors would like to thank the many artists who for over a decade have contributed to the evolution of this renderer and used it to create amazing images.

REFERENCES

Abdalla G. M. Ahmed, Till Niese, Hui Huang, and Oliver Deussen. 2017. An Adaptive Point Sampler on a Regular Lattice. *ACM Trans. Graph.* 36, 4, Article 138 (July 2017), 13 pages. DOI: http://dx.doi.org/10.1145/3072959.3073588

Dietger Van Antwerpen. 2011. Recursive MIS Computation for Streaming BDPT on the GPU. (2011).

Carsten Benthin, Sven Woop, Ingo Wald, and Attila T. Áfra. 2017. Improved Two-level BVHs Using Partial Re-braiding. In *Proceedings of High Performance Graphics (HPG '17)*. ACM, New York, NY, USA, Article 7, 8 pages. DOI: http://dx.doi.org/10.1145/3105762.3105776

Norbert Bus and Tamy Boubekeur. 2017. Double Hierarchies for Directional Importance Sampling in Monte Carlo Rendering. *Journal of Computer Graphics Techniques (JCGT)* 6, 3 (28 August 2017), 25–37. http://jcgt.org/published/0006/03/02

E. Catmull and J. Clark. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design* 10, 6 (November 1978), 350–355.

Alejandro Conty and Christopher Kulla. 2017. Importance Sampling of Many Lights with Adaptive Tree Splitting. In *ACM SIGGRAPH 2017 Talks (SIGGRAPH '17)*. ACM, New York, NY, USA, Article 33, 2 pages. DOI: http://dx.doi.org/10.1145/3084363.3085028

Robert L. Cook. 1984. Shade Trees. *SIGGRAPH Comput. Graph.* 18, 3 (Jan. 1984), 223–231. DOI: http://dx.doi.org/10.1145/964965.808602

Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes Image Rendering Architecture. *SIGGRAPH Comput. Graph.* 21, 4 (Aug. 1987), 95–102. DOI: http://dx.doi.org/10.1145/37402.37414

M. Ernst, M. Stamminger, and G. Greiner. 2006. Filter Importance Sampling. In *2006 IEEE Symposium on Interactive Ray Tracing*. 125–132. DOI: http://dx.doi.org/10.1109/RT.2006.280223

Julian Fong, Magnus Wrenninge, Christopher Kulla, and Ralf Habel. 2017. Production Volume Rendering: SIGGRAPH 2017 Course. In *ACM SIGGRAPH 2017 Courses (SIGGRAPH '17)*. ACM, New York, NY, USA, Article 2, 79 pages. DOI: http://dx.doi.org/10.1145/3084873.3084907

Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramon Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, Adrien Herubel, Declan Russell, Frederic Servant, and Marcos Fajardo. 2018. Arnold: A Brute-force Production Path Tracer. *ACM Transaction on Graphics* (2018).

Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. 2012. Light Transport Simulation with Vertex Connection and Merging. *ACM Trans. Graph.* 31, 6, Article 192 (Nov. 2012), 10 pages. DOI: http://dx.doi.org/10.1145/2366145.2366211

David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.* 23, 1 (March 1991), 5–48. DOI: http://dx.doi.org/10.1145/103162.103163

Larry Gritz and others. 2008. OpenImageIO. (2008). https://github.com/OpenImageIO/oiio

Larry Gritz and James K. Hahn. 1996. BMRT: A Global Illumination Implementation of the RenderMan Standard. *Journal of Graphics Tools* 1, 3 (1996), 29–47. DOI: http://dx.doi.org/10.1080/10867651.1996.10487462

Larry Gritz, Clifford Stein, Chris Kulla, and Alejandro Conty. 2010. Open Shading Language. In *ACM SIGGRAPH 2010 Talks (SIGGRAPH '10)*. ACM, New York, NY, USA, Article 33, 1 pages. DOI: http://dx.doi.org/10.1145/1837026.1837070 https://github.com/imageworks/OpenShadingLanguage.

Toshiya Hachisuka, Anton S. Kaplanyan, and Carsten Dachsbacher. 2014. Multiplexed Metropolis Light Transport. *ACM Trans. Graph.* 33, 4, Article 100 (July 2014), 10 pages. DOI: http://dx.doi.org/10.1145/2601097.2601138

Pat Hanrahan and Jim Lawson. 1990. A Language for Shading and Lighting Calculations. *SIGGRAPH Comput. Graph.* 24, 4 (Sept. 1990), 289–298. DOI: http://dx.doi.org/10.1145/97880.97911

Paul S. Heckbert. 1990. Adaptive Radiosity Textures for Bidirectional Ray Tracing. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '90)*. ACM, New York, NY, USA, 145–154. DOI: http://dx.doi.org/10.1145/97879.97895

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)*. ACM, New York, NY, USA, Article 7, 8 pages. DOI: http://dx.doi.org/10.1145/3084873.3084893

Jared Hoberock and John C. Hart. 2010. Arbitrary Importance Functions for Metropolis Light Transport. *Comput. Graph. Forum* 29, 6 (2010), 1993–2003.

Henrik Wann Jensen, James Arvo, Marcos Fajardo, Pat Hanrahan, Don Mitchell, Matt Pharr, and Peter Shirley. 2001. State of the Art in Monte Carlo Ray Tracing for Realistic Image Synthesis. *Siggraph Courses* (2001).

Anton S. Kaplanyan and Carsten Dachsbacher. 2013. Path Space Regularization for Holistic and Robust Light Transport. *Computer Graphics Forum (Proc. of Eurographics 2013)* 32, 2 (2013), 63–72.

Timothy L. Kay and James T. Kajiya. 1986. Ray Tracing Complex Scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*. ACM, New York, NY, USA, 269–278. DOI: http://dx.doi.org/10.1145/15922.15916

C. Kelemen, L. Szirmay-Kalos, G. Antal, and F. Csonka. 2002. A simple and robust mutation strategy for the metropolis light transport algorithm. In *Computer Graphics Forum*, Vol. 21. 531–540.

Doyub Kim and Hyeong-Seok Ko. 2007. Eulerian Motion Blur. In *Proceedings of the Third Eurographics Conference on Natural Phenomena (NPH'07)*. Eurographics Association,

Aire-la-Ville, Switzerland, Switzerland, 39–46. DOI: http://dx.doi.org/10.2312/NPH/NPH07/039-046

Alan King, Christopher Kulla, Alejandro Conty, and Marcos Fajardo. 2013. BSSRDF Importance Sampling. In *ACM SIGGRAPH 2013 Talks (SIGGRAPH '13)*. ACM, New York, NY, USA, Article 48, 1 pages. DOI: http://dx.doi.org/10.1145/2504459.2504520

Christopher Kulla and Marcos Fajardo. 2012. Importance Sampling Techniques for Path Tracing in Participating Media. *Comput. Graph. Forum* 31, 4 (June 2012), 1519–1528. DOI: http://dx.doi.org/10.1111/j.1467-8659.2012.03148.x

Peter Kutz., Ralf Habel, Yining Karl Li, and Jan Novák. 2017. Spectral and Decomposition Tracking for Rendering Heterogeneous Volumes. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2017)* 36, 4, Article 111 (July 2017). DOI: http://dx.doi.org/10.1145/3072959.3073665

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673 http://llvm.org.

Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. 2009. Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. In *ACM SIGGRAPH Asia 2009 Papers (SIGGRAPH Asia '09)*. ACM, New York, NY, USA, Article 151, 9 pages. DOI: http://dx.doi.org/10.1145/1661412.1618497

Henry Moreton. 2001. Watertight Tessellation Using Forward Differencing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS '01)*. ACM, New York, NY, USA, 25–32. DOI: http://dx.doi.org/10.1145/383507.383520

R. Keith Morley, Solomon Boulos, Jared Johnson, David Edwards, Peter Shirley, Michael Ashikhmin, and Simon Premože. 2006. Image Synthesis Using Adjoint Photons. In *Proceedings of Graphics Interface 2006 (GI '06)*. Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 179–186. http://dl.acm.org/citation.cfm?id=1143079.1143109

Thomas Müller, Markus Gross, and Jan Novák. 2017. Practical Path Guiding for Efficient Light-Transport Simulation. *Computer Graphics Forum* 36, 4 (June 2017), 91–100. DOI: http://dx.doi.org/10.1111/cgf.13227

Koji Nakamaru and Yoshio Ohno. 2002. Ray Tracing for Curves Primitive. In *WSCG*.

K. Perlin and E. M. Hoffert. 1989. Hypertexture. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '89)*. ACM, New York, NY, USA, 253–262. DOI: http://dx.doi.org/10.1145/74333.74359

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering: From Theory to Implementation* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Dan Piponi. 2004. Automatic Differentiation, C++ Templates, and Photogrammetry. *Journal of Graphics Tools* 9, 4 (2004), 41–55.

Holly E. Rushmeier and Gregory J. Ward. 1994. Energy Preserving Non-linear Filters. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '94)*. ACM, New York, NY, USA, 131–138. DOI: http://dx.doi.org/10.1145/192161.192189

Charles M. Schmidt and Brian Budge. 2002. Simple Nested Dielectrics in Ray Traced Images. *Journal of Graphics Tools* 7, 2 (2002), 1–8. DOI: http://dx.doi.org/10.1080/10867651.2002.10487555 arXiv:http://dx.doi.org/10.1080/10867651.2002.10487555

Pradeep Sen, Matthias Zwicker, Fabrice Rousselle, Sung-Eui Yoon, and Nima Khademi Kalantari. 2015. Denoising Your Monte Carlo Renders: Recent Advances in Image-space Adaptive Sampling and Reconstruction. In *ACM SIGGRAPH 2015 Courses (SIGGRAPH '15)*. ACM, New York, NY, USA, Article 11, 255 pages. DOI: http://dx.doi.org/10.1145/2776880.2792740

Kenneth Chiu Peter Shirley and Changyaw Wang. 1994. Multi-jittered sampling. *Graphics gems IV* 4 (1994), 370.

Martin Stich, Heiko Friedrich, and Andreas Dietrich. 2009. Spatial Splits in Bounding Volume Hierarchies. In *Proc. High-Performance Graphics 2009*.

Kartic Subr, Gurprit Singh, and Wojciech Jarosz. 2016. Fourier Analysis of Numerical Integration in Monte Carlo Rendering: Theory and Practice. In *ACM SIGGRAPH Courses*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/2897826.2927356

László Szécsi, László Szirmay-Kalos, and Csaba Kelemen. 2003. Variance Reduction for Russian-roulette. In *The 11-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2003, WSCG 2003, in co-operation with EUROGRAPHICS and IFIP working group 5.10 on Computer Graphics and Virtual Worlds, University of West Bohemia, Campus Bory, Plzen-Bory, Czech Republic, February 3-7, 2003*. http://wscg.zcu.cz/wscg2003/Papers_2003/C29.pdf

Eric Veach. 1998. *Robust Monte Carlo Methods for Light Transport Simulation*. Ph.D. Dissertation. Stanford, CA, USA. Advisor(s) Guibas, Leonidas J. AAI9837162.

Jiří Vorba, Ondřej Karlík, Martin Šik, Tobias Ritschel, and Jaroslav Křivánek. 2014. On-line Learning of Parametric Mixture Models for Light Transport Simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33, 4 (aug 2014).

Martin Šik, Hisanari Otsu, Toshiya Hachisuka, and Jaroslav Křivánek. 2016. Robust Light Transport Simulation via Metropolised Bidirectional Estimators. *ACM Trans. Graph.* 35, 6, Article 245 (Nov. 2016), 12 pages. DOI: http://dx.doi.org/10.1145/2980179.2982411

Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. 2005. Lightcuts: A Scalable Approach to Illumination. *ACM Trans. Graph.* 24, 3 (July 2005), 1098–1107. DOI: http://dx.doi.org/10.1145/1073204.1073318

Lance Williams. 1983. Pyramidal Parametrics. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '83)*. ACM, New York, NY, USA, 1–11. DOI: http://dx.doi.org/10.1145/800059.801126

Sven Woop, Attila T. Áfra, and Carsten Benthin. 2017. STBVH: A Spatial-temporal BVH for Efficient Multi-segment Motion Blur. In *Proceedings of High Performance Graphics (HPG '17)*. ACM, New York, NY, USA, Article 8, 8 pages. DOI: http://dx.doi.org/10.1145/3105762.3105779

Sven Woop, Carsten Benthin, Ingo Wald, Gregory S. Johnson, and Eric Tabellion. 2014. Exploiting Local Orientation Similarity for Efficient Ray Traversal of Hair and Fur. In *Proceedings of High Performance Graphics (HPG '14)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 41–49. http://dl.acm.org/citation.cfm?id=2980009.2980014

Magnus Wrenninge. 2009. Field3D. (2009). https://github.com/imageworks/Field3D

Magnus Wrenninge. 2016. Efficient Rendering of Volumetric Motion Blur Using Temporally Unstructured Volumes. *Journal of Computer Graphics Techniques (JCGT)* 5, 1 (31 January 2016), 1–34. http://jcgt.org/published/0005/01/01/