The Design and Evolution of Disney's Hyperion Renderer

BRENT BURLEY, DAVID ADLER, MATT JEN-YUAN CHIANG, HANK DRISKILL, RALF HABEL, PATRICK KELLY, PETER KUTZ, YINING KARL LI, and DANIEL TEECE, Walt Disney Animation Studios

Fig. 1. Production frames from *Big Hero 6* (upper left), *Zootopia* (upper right), *Moana* (bottom left), and *Olaf's Frozen Adventure* (bottom right), all rendered using Disney's Hyperion Renderer.

Walt Disney Animation Studios has transitioned to path-traced global illumination as part of a progression of brute-force physically based rendering in the name of artist efficiency. To achieve this without compromising our geometric or shading complexity, we built our Hyperion renderer based on a novel architecture that extracts traversal and shading coherence from large, sorted ray batches. In this article, we describe our architecture and discuss our design decisions. We also explain how we are able to provide artistic control in a physically based renderer, and we demonstrate through case studies how we have benefited from having a proprietary renderer that can evolve with production needs.

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

Additional Key Words and Phrases: Production rendering, physically based rendering, path tracing

ACM Reference format:
Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. 2018. The Design and Evolution of Disney's Hyperion Renderer. *ACM Trans. Graph.* 37, 3, Article 33 (July 2018), 22 pages.
https://doi.org/10.1145/3182159

1 INTRODUCTION

Since the early 1990s, rendering of computer graphics (CG) imagery at Walt Disney Animation Studios had been accomplished using the Reyes algorithm (Cook et al. 1987) in Pixar's RenderMan. More recently, ray-traced global illumination (GI) promised artists significant productivity gains by providing more immediate feedback during rendering and removing the significant data management burden associated with shadow maps and point clouds. However, initial attempts to render our existing production scenes with ray-traced GI were unsuccessful; incoherent access of texture maps inhibited shading of indirect ray hits, and we had difficulty fitting our scenes in memory as required by existing ray-traced renderers.

To overcome these limitations, we created a new rendering architecture which traces and shades rays in large batches, first sorting each batch for geometric coherence during scene traversal, then sorting ray hits for texture coherence during shading (Eisenacher et al. 2013). This streaming architecture is at the heart of Hyperion, our proprietary renderer, and allows us to render scenes using ray-traced GI without compromising geometric

or shading detail. Starting with the production of *Big Hero 6*, Hyperion has been used to render all CG images in our studio (Figure 1).

In this article, we begin by recounting our motivation and philosophy for developing a new renderer in Section 2. We detail fundamental aspects of the system in Section 3. We present case studies of significant features that have evolved Hyperion's design based on specific production needs in Section 4. We then discuss how we make path tracing practical in production—how we reduce noise, debug problems, and provide artistic control within the confines of our physically based renderer—in Section 5. We finish by discussing tradeoffs and limitations of our design decisions with some suggestions for future directions in Section 6.

2 BACKGROUND

2.1 Why a New Renderer?

Historically, rendering at Walt Disney Animation Studios has evolved around an efficient look-development process based on a philosophy of "painted proceduralism." Texture maps form the foundation of our shading, offering advantages in both authoring and consumption, with typically dozens of unique texture layers applied to each surface. Textures are frequently adjusted and layered during shading using expressions (Selle et al. 2011), providing artistic flexibility and control within a consistent, predictable interface. While expressions can also be used for in-render pattern generation, they are more typically baked into texture maps to gain the benefits of high-quality filtering, intrinsic level-of-detail through mipmaps, and predictable cost.

This texture-heavy workflow has been advantageously employed on all of our productions for well over a decade. More recently, alongside the move to physically based rendering, adoption of techniques such as Point-Based Global Illumination (Christensen 2010) facilitated a richer look in rendered imagery, but at a cost. Artists were burdened with long iteration times, having to bake lighting into vast point cloud files before rendering. As an alternative, the immediacy of ray-traced global illumination promised increased artist efficiency, but came with other concerns. Incoherent texture access when shading secondary rays proved prohibitive with our texture maps. The requirement that the scene be fully resident in memory to enable ray tracing would also have forced us to scale back our scene complexity; something we could not impose in the name of reducing iteration time. For instance, attempting to render even a fraction of one city block in *Big Hero 6* in an existing ray-traced renderer exceeded the 64GB physical memory limit of our systems at the time, and we estimated our need to be perhaps an order of magnitude greater. In contrast, these are areas that the Reyes architecture handles well—by dicing and shading the scene one surface at a time, Reyes supports coherent texture access on geometry larger than available memory. These observations gave rise to the question: is it possible to combine the core strengths of Reyes with the promise of ray-traced global illumination? Specifically, could we perform GI rendering in a streaming manner, for example, making one pass over the scene for camera rays and one additional pass per indirect bounce?

2.2 Developing Hyperion

To explore the idea of "streaming GI," we implemented a number of experimental renderers. Initially we tried various biased approaches such as tracing "integrating cones" with the hope that we would need many fewer cones than rays. We also considered sharing radiance estimates between cones at intermediate hit points, but this approach made reasoning about the quality and correctness of the image difficult. Ultimately, we found Monte Carlo path tracing to be the simplest and most robust approach, but this meant we would have far more rays than expected, more than we could fit in memory. Because of this, we shifted our focus to maximizing traversal and shading coherence to minimize the number of times we visit each scene object.

In our final prototype (Eisenacher et al. 2013), sorted deferred path tracing using batches of up to 33 million rays provided enough shading coherence to allow efficient texturing even when all of the textures were reread from the network for each batch. We also demonstrated only a 2× performance penalty for *out-of-core* geometry, where the scene's geometry is too large to fit in main memory so objects are purged and reloaded from memory during ray intersection. The results of our prototype were compelling enough that we decided to write a renderer targeted at our then-current production, *Big Hero 6*. This gave us roughly 18 months to develop Hyperion, adding features such as motion blur, instancing, hair, subsurface scattering, volumes, and various artistic controls. Hyperion also needed features to enable artists and technical users to debug their scenes. And because we could not afford to render images to convergence, we needed to develop a robust denoising solution.

Our goals were ambitious but we had great advantages developing Hyperion during and specifically for *Big Hero 6*: we only needed to implement features that were critical to the production, and we always had a large amount of representative test data available to drive development. To mitigate risk we focused initially on features that had no workarounds, delaying features such as motion blur and volumes which could be handled with compositing techniques, though fortunately we completed these additional features in time. We initially maintained our previous rendering pipeline as a contingency plan, though this was abandoned during production because of the overhead incurred by maintaining two pipelines with feature parity, and artists had already become dependent on the benefits offered by the new renderer. We ultimately succeeded because of the deep commitment of the *Big Hero 6* production team.

2.3 Design Philosophy

Hyperion continues our migration toward physically based rendering (Burley 2012, 2015). Being an animation studio, our motivation for using phyiscally based rendering is primarily artist productivity rather than any explicit goal of increased realism. It promises rich, plausible results with a minimum of effort and provides a better starting point for artistic iteration than ad-hoc rendering. To meet art direction, which is never strictly photorealistic, we layer nonphysical artistic controls which we discuss in Section 5.3. And as each production produces new rendering challenges, we continue to add features and evolve the architecture.

To preserve our primary goal of artist productivity, in everything we do we aim to maintain the following philosophies:

— **Simplicity over flexibility.** We try not to burden users with non-artistic controls. For instance, we expose no ray bias options, no per-lobe or per-material sampling controls, and no choice of integration strategy. Even for artistic controls, we work to have as few as possible, and we review our controls periodically to see what we can remove.
— **General rather than specialized solutions.** We prefer solutions with robust behavior and predictable performance over optimizing for special cases.
— **Artist iteration over final frame efficiency.** Approaches that require manual tuning or time-consuming preprocessing inhibit iteration. Unbiased progressive rendering allows artists to iterate quickly on noisy images, ideally with final frames only needing to be rendered once.

The natural result of this philosophy has been a march toward increasingly brute-force solutions. As we describe throughout this article (in particular, in Sections 4.2, 4.4, and 4.5), we have been steadily replacing biased approximations with Monte Carlo solutions. In every case the motivation for doing so has been artist efficiency, and in every case the quality and consistency of our imagery also increased. Furthermore, having an in-house renderer has afforded us wide latitude in tailoring solutions specifically to the needs of our artists according to this philosophy.

3 SYSTEM DESCRIPTION

In this section, we present details about Hyperion's architecture. Hyperion's high-level processing is shown in Algorithm 1.

ALGORITHM 1: Hyperion renderer pseudocode

```
1  function Render
2      RunOneIteration // generates initial cache points
3      calculate photon-map probability mass functions
4      while not done do
5          RunOneIteration
6  function RunOneIteration
7      generate & trace photons
8      generate camera rays
9      ProcessRays
10     update adaptive sampler
11     update cache points
12     save image files & checkpoint
13 function ProcessRays
14     for each rayBatch do
15         // Traverse:
16         cones ← group and sort rayBatch rays into cones
17         hits ← traverse scene, intersecting cones & their rays against geometry
18         // Shade:
19         hitGroups ← sort and group hits by shader and object
20         for hitGroup in hitGroups do
21             sortedHits ← sort hitGroup's hits by face ID
22             for hit in sortedHits do
23                 // these steps splat to framebuffers & generate scattered rays
24                 integrate volumes
25                 shade surface
```

3.1 Iterations

To provide progressive feedback, our rendering is divided into *iterations*, each forming a complete, consistent image, with subsequent iterations adding additional samples per pixel (*SPP*) to converge toward the final image. We start at 4 or 16 SPP, doubling for successive iterations to a maximum of 256 SPP per iteration.

The intervals between iterations provide an opportunity to perform various operations:

— Compute variance estimates and adjust pixel budgets for adaptive sampling, discussed in Section 5.1.
— Refine the light-source importance cache, discussed in Section 4.1.1.
— Write frame buffers to disk, allowing artists to review and even perform compositing with the in-progress images during long-running renders.
— Record checkpoint information allowing the renderer to be restarted if interrupted.
— Refine photon maps, discussed in Section 4.3.1.
— Update the top-level hierarchy (Section 3.3) with tighter bounds for recently expanded procedurals.
— Record per-iteration statistics.

While some of these operations are only valid between iterations, updating data structures between iterations can also provide an efficiency benefit. During an iteration, most data structures are read-only, allowing lock-free sharing among threads. Between iterations, the structures are write-only, avoiding the need for read/write synchronization.

3.2 Coherent Ray Batches

Hyperion processes rays in batches to maximize coherence. We process one batch at a time, first intersecting all rays in the batch with the scene to generate hit points, and then shading the hit points to splat emitted radiance times path throughput into the frame buffer or generate new rays which are queued into future batches. We use ray batches to improve coherence in a number of ways:

— Primary rays are generated following a space-filling Z-order curve.
— All rays (primary and secondary) are organized by dominant semi-axis ($\pm X| \pm Y| \pm Z$) into separate batches, implying an approximate front-to-back traversal ordering for each batch.
— Before traversal, the rays in each batch are recursively sorted, first by origin, then time, then direction.
— Before shading, hit points are sorted by scene object and mesh face index.
— Ready batches are organized into a stack, with the most-recently filled batch therefore being the next batch to be processed. This produces a wide depth-first traversal that improves locality and keeps the number of pending batches in check.

We observe that both traversal and shading coherence improve with batch size and find 33 million rays to be the largest batch size practical in our system.

3.3 Scene Traversal

Our scenes are composed of *scene objects*, each of which references a *traceable* and a *shader*. Typical traceables are individual meshes or collections of curves, spherical particles, or instances.

Traceables intersect rays to produce ray hits, and each traceable is responsible for its own traversal and acceleration structure, usually some form of bounding volume hierarchy (BVH) optimized for its particular needs.

Achieving coherent ray traversal while maintaining efficient traversal speed for individual rays presents a significant challenge in global illumination rendering (Barringer and Akenine-Möller 2014). We separate our hierarchy into two levels:

—The *top-level hierarchy* intersects against the traceables' bounding boxes, determining which traceables a ray may hit. We trace 32-ray packets against this BVH.
—The *bottom-level hierarchy* intersects against a single traceable (e.g., its triangles). We trace individual rays against this BVH.

Because of the large number of rays within each batch, we are able to form coherent packets regardless of ray depth. For packet intersection, we find it advantageous to use bounding cones, avoiding the need to intersect the packet's individual rays with each bounding box; we find cone–box intersection to have similar cost to single-ray intersection, and we intersect multiple cones at once with SIMD. Additionally, we have found that large batch sizes are useful for extracting sets of coherent ray streams compatible for use with other packet traversal approaches, such as the one by Fuetterling et al. (2015).

We load scene objects on demand to avoid memory overhead for unreachable objects and to enable out-of-core rendering by allowing evicted objects to be reloaded. Packets are traversed through the hierarchy and queued at traceables that require loading. If the memory necessary to expand a traceable would cause the render process to exceed a prescribed memory limit, objects are evicted in approximate least-recently-used order. Eviction occurs in two stages, first shrinking traceables where possible, then fully evicting. An example of shrinking is discarding the internal node bounds of a BVH, which can easily be recovered from its leaf nodes.

While the main hierarchy only performs packet traversal of batched radiance-querying rays, shaders may immediately trace individual rays for other purposes. One such purpose is subsurface scattering, which is integrated locally and immediately within the shader. Rays are also immediately traced for probability density function (PDF) evaluation of emissive geometry sampling (discussed in Section 4.1.2). In such cases, only a user-defined subset of the scene is traced, most often corresponding to an individual mesh, and traversal coherence follows naturally from shading coherence.

The way an artist or production pipeline might organize a scene may lead to an inefficient top-level hierarchy construction with large amounts of object overlap. For example, the base mesh for an entire island may be a single scene object, overlapping the millions of objects that populate the island. Worse, each type of instanced object, such as each species of plant, might have a traceable representing a collection of instances covering the entire island. The extreme overlap in these cases defeats the logarithmic cost advantage of BVH traversal potentially making such scenes unrenderable.

To avoid excessive overlap in our top-level hierarchy, we split large objects, including both individual meshes and instance collections, into smaller parts. Rather than forming a separate scene object for each part, we create *entry points* into each object's internal BVH and insert references to these entry points into the top-level hierarchy. This system is generally similar to and was developed concurrently with recent partial BVH rebraiding techniques (Benthin et al. 2017), and entry points are an interchangeable term with *BRefs* from partial re-braiding. Much like in partial re-braiding, our entry point system is essentially building the top-level BVH over sub-trees of the object BVHs instead of directly over object bounds. Instead of building entry point logic directly into the top-level BVH builder, we use a two-step process where object BVHs are built first, then entry points are calculated, and then the top-level BVH is built over entry points using the same exact BVH builder as the object BVHs. Our termination criteria for calculating entry points incorporates two main factors: the sum of an SAH-like heuristic measure for candidate entry points versus the sum for the parent node of the candidates, and the solid angle of each candidate entry point relative to the camera.

3.4 Surface Shading

After traversal of each ray batch, we group the resulting hit points by scene object for shading. Importantly, hit points for instanced objects are grouped and shaded as if they were a single object. We further sort the hit points for shading coherence within each object; for meshes we sort by face, providing maximal coherence for per-face textures (Burley and Lacewell 2008).

Shaders compute surface scattering, enqueuing new radiance-gathering rays into a future ray batch. Shaders also add emitted radiance, multiplied by path throughput, to the framebuffers. As is standard practice, scattered rays are generated using multiple importance sampling to combine BSDF and light sampling techniques (Veach 1997). However, rather than tracing transmittance-only rays for light samples, as is done with traditional next-event estimation, all scattered rays gather radiance. More recently though, we no longer allow rays generated via light sampling to gather indirect radiance for reasons discussed in Section 4.5.

Any particular shader may be bound to multiple objects. To enable asynchronous shading of these objects, we create a local copy of the shader for each object. Local copies allow shaders to modify their state during shading without the overhead of locking or other synchronization. This mechanism also allows us to arbitrarily mutate the shader per object. For instance, a lighting artist may attach *object modifiers* to various scene objects to override shader parameters without the burden of creating and managing persistent copies of the shaders. Concurrent threads preferably shade different objects, but when there remain no other objects to be shaded, we create additional clones of each shader to allow multiple threads to shade a single object.

Our prototype required no texture cache, as we typically have sufficient shading coherence to amortize file access cost. However, we now employ one, mainly to limit network I/O load. Texture data access routinely exceeds our 6GB cache size, yet texture lookups typically account for less than 5% of render time with I/O representing only a small fraction of that.

Texture filtering is based on the *ray diameter*, the diameter of the ray interpreted as a cone with apex angle based on scattering PDFs (Amanatides 1984; Nguyen 2007).

Our shaders also access additional data such as per-vertex attributes and point clouds. Point clouds can be used for arbitrary purposes, often to manipulate shading with animated effects, but one type that is accessed on every shading point is the importance cache used for light selection, discussed in Section 4.1.1. All of these data sources benefit from the coherence provided by our sorted shading. In addition to improving data coherence, sorted shading minimizes shader cloning and other costs related to context switching between shading objects.

3.5 Volume Integration

After surface hit points are determined and sorted but before they are shaded, we compute volume transmittance, in-scattering, and emission along each ray segment. In-scattered rays are enqueued into a future ray batch, and are identical to surface-scattered rays.

Because volume integration occurs as part of sorted deferred shading, our volume data also benefits from increased coherence. However, being tied to surface integration, this approach is only practical for a small number of bounces. We discuss how we recently addressed the need to render arbitrary numbers of volume bounces in Section 4.5.

4 CASE STUDIES IN PRODUCTION-DRIVEN DEVELOPMENT

In this section, we present selected cases of development in Hyperion driven by production needs.

4.1 Sampling Complex Lighting Environments

Complex lighting, both direct and indirect, presents a variety of challenges. It is difficult to predict the specific lighting scenarios that will arise in the middle of production, and even more difficult to design and implement systems that are sufficiently and automatically robust to all of them.

Big Hero 6 contained a city with hundreds of thousands of lights, refractive globes around small bright light sources, a bedroom lit by sunlight filtered through window slats, and a dusty warehouse with visible rays of sunlight entering through small skylights. *Moana* contained massive smoke plumes lit by lava and ocean floors lit by refracted sunlight.

Sometimes it is tempting to provide artists with additional sampling controls so they can manually tune the renderer to handle a particular scenario. However, this approach can backfire. For example, in *Big Hero 6* we provided "window light samplers" that could be placed in the windows of interior spaces to draw rays out of them and reach the light sources on the outside. These turned out to be very difficult to place correctly and to tune, and, when combined with existing light samplers for outdoor lights, could result in oversampling those lights and undersampling the interior lights. We have found that automatic sampling solutions are usually preferable when possible.

4.1.1 Cache Points. It is not unusual for movie scenes to contain huge numbers of light sources. *Big Hero 6* featured nighttime city scenes that contained as many as a half million small, bright lights, including directional sources such as street lights and car headlights, as pictured in Figure 2. Our original approach of looping over every light at every shading point to build a probability distribution for light selection became unusably slow in such situations.

Fig. 2. Scenes in *Big Hero 6* featured anywhere from hundreds of lights (top), to thousands of lights (middle), to hundreds of thousands of lights (bottom). Our cache-points system allows for robust and efficient light sampling in all of these scenarios.

We experimented with hierarchical tree data structures for light selection but found them difficult to make useful due to certain edge cases and varied light types, in particular those with narrow, highly directional IES profiles. We realized that the lighting did not vary enough between nearby shading points to warrant generating a brand new probability distribution for each one, even in scenes with many lights. We explored a number of possible ways of caching and reusing light-selection information, and ultimately designed and implemented a system we call *cache points*.

We generate an initial set of 100,000 cache points randomly distributed within individual scene object bounding boxes, followed by a coarse render iteration to place cache points at a random subset of path vertices, pruning points that fall within a target radius of a previous point. Additional points are added during subsequent rendering iterations in a similar manner. To populate each new cache point, we loop over each light, estimate its potential contribution to six oriented planes as well as an omnidirectional receiver centered at the point. Then, for each of these seven distributions we store a list of just enough lights to account for 97% of the energy

reaching the point (limited to at least 4 and at most 256 lights per distribution) (Shirley et al. 1996). To account for rapid changes in illumination between cache points, we put lights that are very close to the cache point in a separate list. We also aggregate information from each cache point's neighborhood such that only nearest-point lookup is needed during light sampling. The total size of the cache point database is typically only a few MB. The time to generate the cache points is negligible and masked by the time required to load and process the scene objects hit during the cache point iteration.

To further improve sampling, we maintain a per-light visibility estimate at each cache point, conceptually similarly to the approach of Georgiev et al. (2012). We track the number of samples directed to each light and compare this with the number of samples that reach the light and receive a contribution. This information is later used to reduce sampling of lights that were initially thought to be important but which actually do not contribute. This system significantly reduces noise in, for example, interior scenes lit by sunlight entering through small windows, in which the mostly occluded sun would otherwise draw samples away from the other light sources.

To generate a light sample from a cache point, we blend together multiple direction-specific distributions based on the surface normal at the shading point, incorporate the nearby lights and the learned visibility information described above, and then select a light from the resultant probability distribution. Lights that do not appear in the distributions tend to be undersampled and produce *fireflies*, isolated anomalously bright pixels. We limit the impact of fireflies using clamping and denoising (Section 5.1.2).

4.1.2 Emissive Geometry. Hyperion was designed so that any object could be a source of light. In particular, the emission of any surface could be driven by expressions and texture maps. While we initially directly sampled a few explicit light types, we did not have a way of identifying general objects as light sources a priori, determining the distribution of emission over their surfaces, or sampling them in any way other than hitting them by chance with BSDF samples. This was further complicated by lazy loading—we could not know the distribution of emission over a surface that we had not yet loaded.

However, late in the production of *Moana*, we encountered a situation where the lack of emissive geometry sampling was a serious problem. A hundred shots in the movie featured the lava monster Te Kā whose emissive body was the main light source in an otherwise dark environment (Figure 3). The concentrated and bright streams of lava on Te Kā's body were illuminating plumes of smoke that permeated the scene, and, as a further complication, there were also lightning bolts within the smoke. To converge to a usable noise level, it was projected that each frame would take an average of 450 core hours to render.

Before this point, we never had a pressing need to implement emissive geometry sampling, but now we needed to come up with a solution in a short amount of time. We decided to try reusing our emissive-volume sampler for the emissive surfaces, thinking that this would be a straightforward and effective approach. The first step was to identify the emissive objects, which was easy because emission already had to be enabled explicitly in each shader. We

Fig. 3. A production frame from *Moana* that required emissive geometry sampling, since the primary source of illumination in the scene is the large character "Te Kā" made of bright lava.

loaded these objects non-lazily when the renderer was launched, and for each object looped over each triangle of its subdivided mesh, evaluated the emission expression at the center of the triangle, and stored the triangle and its power in a list. Then, for each object, we built a volumetric grid, rasterized the relevant triangle bounding boxes into it, and used the grid as a sampler. This approach worked, but making it practical proved difficult. Higher grid resolutions resulted in better sampling but were very memory intensive and slow to sample. And, as the grid structure was not time-varying, moving triangles occupied far too much space when rasterized and destroyed the sampling quality. We needed a different approach.

We decided to try building a new light sampler from the ground up. This light sampler would store the deforming emissive triangles themselves and sample them directly. After identifying an object's emissive triangles as before, we would pass them into this new light sampler, build a probability distribution over them for sampling, and build a time-varying BVH over them for efficient PDF evaluation. Building a view-dependent probability distribution over the triangles for each shading point was too computationally expensive for all but the simplest meshes, so we constructed a fixed probability distribution based solely on the triangle powers and drew samples from it in constant time using the alias method (Vose 1991). We were initially concerned that the power-based sampling would result in a poor distribution of samples, but in practice it performed well. The triangle-based approach was a significant implementation effort but in the end produced cleaner code without the heuristics necessary for the rasterization approach, and dramatically better sampling quality for animated meshes with motion blur in particular.

Ultimately the improved sampling allowed us to render the problematic Te Kā sequences twice as fast as before with less noise. Independent improvements to denoising further improved the render times. In general, our emissive geometry sampling has proven to be highly beneficial when used appropriately, although in some cases the benefits do need to be weighed against the loss of lazy loading and the expense of evaluating potentially heavy emission expressions for potentially large numbers of triangles. So far, we only build a single light sampler per single triangle mesh, where each light sampler is a linear cumulative density function (CDF) over all of the triangles in the mesh. In practice, we have not yet encountered a case large enough to necessitate spatially splitting the triangle list and building multiple CDFs per mesh, but in the

event that we would need to do so, we envision splitting meshes using the entry points system described in Section 3.3 and building a linear CDF over all of the triangles in a given entry point for all entry points, instead of in the entire object.

4.1.3 Path Simplification. For a caustic path to be discovered by a unidirectional path tracer starting from the camera, a ray has to be scattered from a diffuse surface in such a way that it happens to bounce off a shiny smooth surface and hit a small, bright light source. Without a priori knowledge of the distribution of indirect light in a scene, this sequence of events is highly unlikely; therefore, caustics are not sampled efficiently without more complex techniques such as bidirectional path tracing (Veach 1997). As the surface interactions leading up to the final specular surface become more diffuse, bundles of camera rays become less coherent and less likely to consistently reach the light source using BSDF sampling alone.

During the production of *Big Hero 6*, we faced the problem of caustics causing fireflies strewn across images. The caustic patterns themselves were not particularly important artistically in these cases, so rather than devoting a large amount of resources to implementing a light-transport algorithm that can sample them well, we attempted to get rid of the caustics with the smallest possible overall impact on images. One of the approaches we tested was outlier rejection (DeCoro et al. 2010), but this required a significant amount of memory, produced different results for different SPP values, and resulted in significant energy loss for low SPP values.

We realized that if the final surface were more diffuse, direct light sampling would make reaching the light more likely, and that noise could thus be greatly reduced by increasing the roughness of surfaces in secondary bounces to permit useful direct light sampling (Kaplanyan and Dachsbacher 2013). We did not want to eliminate all indirect specular effects, however, so we designed a heuristic that increased the roughness proportional to the ray diameter. The "wider" and more "diffuse" a ray became, the more we would increase the roughness of subsequent bounces, thus eliminating difficult caustics without completely eliminating interesting directional lighting effects like glossy–glossy transport.

Refraction presented another challenge. Increasing the roughness of refractive surfaces was effective at eliminating noise, but drastically changed the image in some cases. For example, in scenes with sunlight entering a room through a glass window and casting hard shadows, increasing the roughness of the window blurred the shadows and greatly changed the character of the lighting of the scene. We quickly realized that a better solution for handling smooth refraction was to go straight through these surfaces without bending. For this to produce good results, we adjusted the reflection coefficient such that rays passing straight through a window undergo the same amount of reflection from the front and back surfaces as their original bent counterparts, and rays passing out of a refractive object such as a pool do not experience total internal reflection, which would otherwise prevent rays leaving the object from covering the whole hemisphere. In particular, we achieve these properties by replacing the relative index of refraction with its multiplicative inverse when a non-bending ray exits the object.

Path simplification is our name for this set of features that modify BSDF properties in later bounces in order to eliminate difficult caustics and thus reduce noise. Path simplification is always enabled unless we want to render a "ground truth" image for reference or debugging.

4.2 Fur Rendering in *Zootopia*

We knew *Zootopia* would require a city full of appealing animals, with appearances designed to match those of their real-life counterparts, and authoring such a large variety of characters (ultimately, 157 unique characters of 64 different species) was a daunting challenge. Our previous shading model (Marschner et al. 2003; Zinke et al. 2008) was easy to control, but lacked energy conservation. This was sufficient for *Big Hero 6* as most of the characters had dark hair, but *Zootopia* had several characters with nearly white fur, and our multiple scattering approximation (Zinke et al. 2008) was making the fur look dark and dirty. Also, the shading model was fundamentally a human hair model which had no ability to differentiate the various species needed for *Zootopia*.

In addition to the authoring and shading challenges, *Zootopia* presented geometric challenges for the scale of the fur in terms of both memory to hold the geometry and render time to traverse so many curves. There were up to 5,000 characters in a shot, with typical characters having 2 to 10 million hairs, and with 9 curve segments per hair on average and as many as 100 segments per curve for characters with particularly long or curly fur such as the sheep (see Figure 4 for examples).

4.2.1 Shading. We began with the energy-conserving hair shading model of d'Eon et al. (2011), but used a near-field formulation to avoid the expensive numerical integration over the fiber width. We found that brute-force multiple scattering with this model gave us the softness and richness we were looking for, and the additional azimuthal roughness control provided species differentiation, but we needed to further improve the efficiency and controllability. Through the use of a novel fourth lobe and closed-form logistic azimuthal distribution, we were able to gain enough efficiency to make brute-force multiple-scattering practical. For controllability, we derived single-scattering parameters from the artist-specified multiple-scattering color by inverting the results of a numerical scattering simulation. Our final model has six intuitive parameters (Chiang et al. 2016a) and has since been used on all our animal and human characters in production.

4.2.2 Geometry and Traversal. We represent hair fibers as cubic B-spline curves with widths that vary along the length of the curve. Compared to pre-tessellated curves, splines are memory efficient and provide high quality at any zoom level. Unlike commonly used Bézier representations (Nakamaru and Ohno 2002; Woop et al. 2014), which require four CVs for every segment, the B-spline representation requires only one CV per additional segment after the initial one. To further reduce curve memory, we quantize control point positions to 16-bit fixed-point numbers relative to the bounding box of the containing set of curves. We store curve segments in a BVH, which can be even more memory intensive than storing the control points. To resolve this, we quantize bounding

Fig. 4. Various scenes with fur in *Zootopia*: high albedo fur requiring accurate multiple scattering (top), characters with complex color variation using our intuitive color parameterization (middle), and crowds with lots of furry characters requiring efficient geometric representations and efficient traversal (bottom).

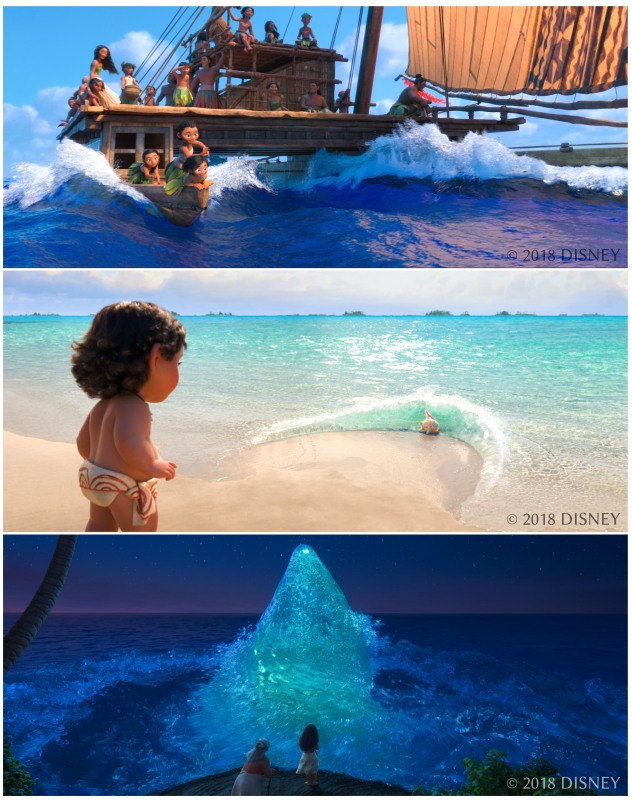


Fig. 5. Different types of water in *Moana*: ocean with boat wakes (top), shoreline with open ocean and character interaction (middle), and art-directed sentient water (bottom).

box positions to 8-bit fixed-point numbers relative to their parent bounding box as described by Mahovsky (2005).

For ray-segment intersection, we follow the method of Nakamaru and Ohno (2002), projecting each curve segment to a plane perpendicular to the ray, recursively bisecting the segment until an error tolerance is reached, and then performing linear-segment intersection. At intersection time, we convert the 3D B-spline segment to 2D cubic polynomial form within the ray's intersection plane. The cubic polynomial form allows more efficient position and derivative evaluation by avoiding the need to repeatedly evaluate B-spline or Bézier basis functions. Rather than using a fixed recursion depth precomputed from a specified error tolerance (Nakamaru and Ohno 2002), which could result in an arbitrarily high number of recursions even for distant curves, we measure the error at each traversal step and compare with the ray diameter. We estimate this error as the sum of the difference between each tangent vector and the linear-segment approximation. Segments that are nearly straight, distant from camera, or intersected with wide-diameter indirect rays require minimal recursion. In our typical production scenes, we find that the vast majority of ray segment intersections require no recursion with our error metric.

It is known that dense, diagonally oriented curve sets can perform poorly due to overlapping segment bounds (Woop et al. 2014). Our approach is to automatically split long curve segments, adding each sub-segment separately to the BVH. While this works well for long, thin curve segments, it does not help with short, fat ones as splitting will only worsen the overlap; artists simply avoid this case. For instance, when artists decimate distant curve density for rendering efficiency, and compensate by increasing curve widths, they are also careful to decimate the CV density along the curves to preserve the curve segment length-to-width aspect ratio.

4.3 Ocean Rendering in *Moana*

Moana contained by far the largest and most complex water scenes our studio has ever produced. Water was present in almost all shots, including vast ocean, boat wakes, splashes, shorelines, walls of water, and highly art-directed sentient water (see Figure 5). During *Moana*, a dedicated pipeline (Garcia et al. 2016) was developed for producing single large water meshes, generated from a level-set compositing graph (Palmer et al. 2017) consisting of any number of inputs, such as procedural ocean surfaces or voxelized level sets.

In this system, the mesh resolution is defined by the camera frustum and off-screen geometry is kept at a low resolution, resulting in meshes of an average size of around 2 gigabytes. This approach avoids treating the different types of water as separate elements. As an example, the miles away ocean at the horizon and close-up spray or bubbles are a single mesh. Due to the level-set definition

of the water surface, the resulting geometry is always watertight and can be used as the boundary for the homogeneous volume defining the scattering optical properties of the ocean water. The volumetric appearance of the water is fully path-traced with physically correct scattering parameters.

We intended to include foam into the water rendering setup so that we could blend seamlessly between clear water, turbid bubble-filled water, and foam, but the research and development necessary to deliver a production-ready system was too time-consuming. Therefore, foam is modeled using standard volumes and particles as separate elements.

4.3.1 Photon-Mapped Caustics. Refractive caustics through clear water are a defining attribute of tropical shoreline water appearance, creating intricate patterns on the submerged sand as the waves churn (see Figure 5, middle). Additionally, reflective caustics cause subtle illumination on shoreline rocks and other nearby objects such as boats on the open ocean. Incorporating these lighting effects into the shots in a realistic way was key in defining the look of the water and increasing the visual complexity.

Rendering caustics is highly inefficient in a purely forward path tracer like Hyperion and they do not converge in a reasonable amount of time. While certain bidirectional techniques can resolve such difficult features, we could not change the inherently unidirectional nature of Hyperion's streaming architecture just for rendering caustics. Projective methods were deemed too labor-intensive to do well on a large number of difficult shots. To address realistic caustics, we implemented a limited photon-mapping (Jensen 1996) subsystem that only renders ideal specular caustics, only interacting with the water surface for reflection and refraction.

In many cases, the photon maps needed to span large areas such as a shoreline or large sections of shallow ocean seabed in underwater shots. We addressed this by adaptively distributing photons. We start by distributing one set of photons uniformly over the scene. Then, during the same initial iteration that distributes cache points, we gather these photons, but instead of writing their contributions to the image, we record their contributions into a grid on the plane from which they were emitted. This grid is then used to build a probability mass function (PMF) over the plane, which is used to distribute a new set of photons before each render iteration. This system automatically results in a higher density of photons within the camera frustum, near the camera, and on visible parts of objects, as illustrated in Figure 6. This approach is currently implemented only for narrow, infinitely far away lights—since the emission directions are largely parallel, it is sufficient to build a spatial PMF over the emitting plane. However, extending this approach to work with diffusely emitting, finite lights should be possible as well. For full generality, instead of building a PMF on a plane, one could be built in primary sample space.

Our photon system does not support volume caustics, which were instead modeled with cucoloris lights scattering inside the ocean volume.

While the system is not capable of doing much more than specular water caustics and similar effects, it was largely automatic and was employed in almost all shots with shorelines and boats. The large number of shots containing water justified developing a full subsystem for this specialized purpose. The limited scope allowed us to optimize toward the specialized use case in *Moana*. Later extensions of this system to partially support area lights also allowed artists to find limited applications toward caustics from ice in *Olaf's Frozen Adventure*.

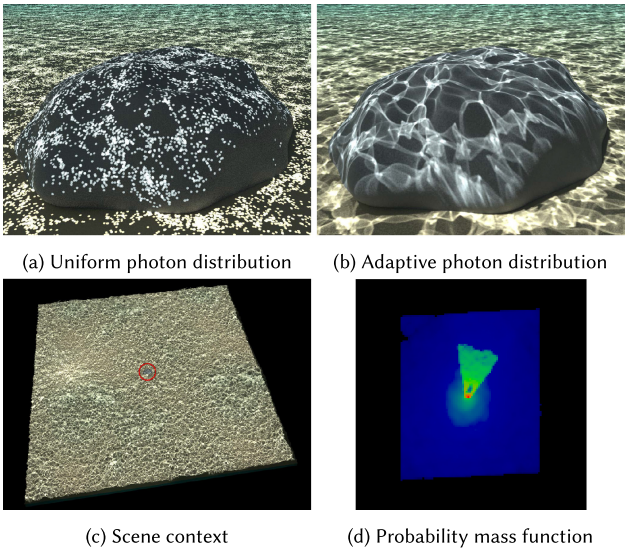


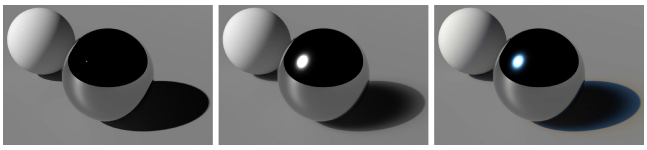
Fig. 6. An example of adaptive photon distribution. (a) and (b) show a close-up of a small rock in a large environment rendered using the same number of photons without and with adaptive photon distribution, respectively. The context of the close-up is shown in (c) with a red circle around the rock, and the probability mass function used for adaptive photon distribution is shown in (d) with the square water surface, camera frustum, region around the camera, rock, and area behind the rock all clearly visible.

4.3.2 Halo Light. One of the challenges of rendering water was that a physically correct sun caused very strong harsh sparkling highlights which in many cases were noisy but also were undesired for the look of the movie. Increasing the roughness of the water shader uniformly blurred out the features and was not a satisfactory solution to the problem. Instead, the sunlight was softened by providing artistic control over a halo around the sun, emulating the Mie-scattering halo around the sun's disk.

Following our paradigm of simplicity, instead of giving full control over the halo with an expression, a two-parameter model was devised that determines the shape of the halo. An important attribute of the model is that the overall light intensity is constant, avoiding any intensity changes while tuning the parameters. While the halo light has been specifically developed to address the look of water, it found its way into the standard toolset of lighters for defining the shape and color of highlights, shadow terminators, and shadow borders, as illustrated in Figure 7.

4.4 Unifying Subsurface Scattering, from Snow to Skin

For *Big Hero 6*, we used Normalized Diffusion (Burley 2015) as our subsurface-scattering solution. It matches the Monte Carlo reference well, and is efficient and easy to control. However, like other diffusion approximations, it suffers from artifacts caused by



(a) No halo (b) Regular halo (c) Tinted halo

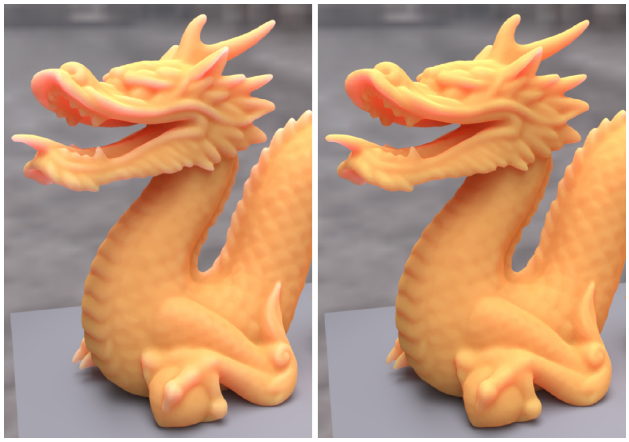
Fig. 7. A simple scene lit by a regular sun light (left), a sun light with a user-defined halo (center), and a sun light with a tinted halo (right). The halo can help shape the highlights, shadow terminators, and shadowed borders.

Fig. 8. A production frame from *Frozen Fever* with characters using path-traced subsurface-scattering snow.

the *semi-infinite-slab* assumption. Such artifacts were particularly problematic in *Frozen Fever* where the snow monster, Marshmallow, is made of geometrically complex interpenetrating ice and snow which exhibited both undesirable darkening in thin regions as well as excessive light penetrating into crevices. Previously, with Reyes rendering in *Frozen*, such snow ice integration was accomplished with ad hoc solutions that were labor-intensive, producing not entirely satisfactory results, and inapplicable in Hyperion regardless.

4.4.1 Path-Traced Snow. For *Frozen Fever* we implemented a volumetric path-tracing solution for the snow which integrated with ice seamlessly and avoided the artifacts artists previously struggled with. To make path-traced snow viable for production rendering, as in Figure 8, we made a few simplifications. We treated snow as a completely homogeneous volume with monochromatic scattering. For rendering efficiency, we assumed index-matched diffusely transmitting interfaces. Also, since snow generally has uniform optical properties throughout, though unintuitive, we had artists control snow directly using volumetric single-scattering parameters.

While successful on a smaller scale, the fact that path-traced subsurface scattering faithfully preserves all geometric details posed surprising challenges rendering the larger snowscapes in *Olaf's Frozen Adventure*. For instance, snow skirts around trees were often not closed and floated above the ground, and snow clumps on top of other surfaces such as roofs interpenetrated, creating unwanted shadowing. To address geometric issues, we implemented multi-object interface counting to determine when we are inside the snow volume as well as a probe-ray heuristic that attempts to handle open and transparent surfaces, but ultimately artists needed to be more mindful of geometric correctness.



(a) IOR 1 (b) IOR 1.4

Fig. 9. The Stanford dragon rendered with IOR 1 in (a) shows overly bright thin edges due to the lack of internal reflection. It can be fixed by using a more reasonable IOR value, in this case 1.4 in (b). Dragon model courtesy of the Stanford Computer Graphics Laboratory.

We also faced challenges relating to the plausibility of the scattering parameters. In animated films, exaggerated light bleeding into shadow is often desired and achieved by having higher scattering distances. However, unlike diffusion, this conflicts with the fact that path-traced subsurface scattering accurately represents the thickness of objects. For example, the layer of snow covering rooftops in *Olaf's Frozen Adventure* initially appeared to be too dark since too much of the light passed through to the dark roof. In these cases, we relied on artists to use painted masks to reduce volumetric scattering in areas appearing too thin.

Computational efficiency was an additional challenge. Path-traced subsurface scattering in snow using a plausibly high albedo, that is, very close to one, requires simulating a very high number of scattering events, which can be prohibitive. We found that rendering such high albedo could be made practical using shell transport (Müller et al. 2016) to take larger steps within the interior of snow volumes, though in practice our productions have found it sufficient to use slightly reduced albedo values for snow in conjunction with higher scattering distances.

4.4.2 Path-Traced Skin. With the success of path-traced snow, we wondered whether we could get similar benefits for other materials such as skin. The first challenge is that skin is not homogeneous and it is not intuitive for artists to directly specify single-scattering parameters. As before with Normalized Diffusion, our artists paint skin albedo maps. Analogous to our fur-color reparameterization, we derive single-scattering parameters from our Normalized Diffusion parameterization. Since our initial implementation (Chiang et al. 2016b), we have discovered that assuming index-matched interfaces for internal scattering leads to overly bright thin edges due to the lack of internal reflection as shown in Figure 9. However, with the introduction of internal reflection, the average path length is increased and so is the amount of volumetric absorption, making our reparameterization dependent on the index of refraction (IOR). We found that assuming an IOR of

Fig. 10. A production render from *Ralph Breaks the Internet: Wreck-It Ralph 2* with characters using path-traced subsurface-scattering skin.

1.4 (only for calculating the fraction of energy internally reflected) reasonably represents most materials, and thus we rederived our single-scattering parameters for this assumption:

$$\alpha = 1 - e^{-11.43A+15.38A^2-13.91A^3},$$
$$s = 4.012 - 15.21A + 32.34A^2 - 34.68A^3 + 13.91A^4,$$
$$\sigma_t = 1/(ds),$$

where the artist-specified surface color A and scattering distance d are internally converted to the single-scattering albedo α and extinction coefficient σ_t for rendering.

Ideally, an accurate skin model should be layered and heterogeneous, though for artistic controllability we continue the widely used non-physical approach of modeling skin as a spatially varying homogeneous volume (Jensen et al. 2001). Modeling skin this way with chromatic scattering can produce excessive coloration noise, compounded by the fact that highly saturated color bleeding is often exaggerated in animated films. We minimize this coloration noise by sampling each color channel proportional to its single-scattering albedo (Chiang et al. 2016b) and we find the resulting rendering efficiency surprisingly similar to diffusion.

By matching our Normalized Diffusion parameterization, we were able to re-render existing characters, and artists generally preferred the path-traced result as it enhanced details that are often blurred away by diffusion. However, to work around the artifacts and limitations of diffusion, the characters had been modeled with exaggerated creases and wrinkles and used hand-painted scattering masks. For this reason, existing productions continued to use diffusion, but path-traced subsurface scattering is now being used in all current productions on all materials from snow to skin (see Figure 10).

4.5 Rendering Cloudscapes

The volume rendering system developed during *Big Hero 6* embraces the sorted-deferred philosophy by using splitting and providing scattered rays to Hyperion's batching system, rather than immediately redirecting the incoming ray upon scattering. Upon encountering a surface intersection, before shading, Hyperion estimates the transmittance using residual ratio tracking (Novák et al. 2014) and builds a probability density function for sampling scattering distances. The PDF draws its form from the transmittance estimate, the volume extinction, and a fluence estimate gathered from the cache points. Multiple importance sampling is performed between these contributions. Figure 11 (left) illustrates the splitting and PDF building. A more detailed description of the approach can be found in the course notes by Fong et. al (2017). The fluence estimate generated from the cache points has a similar effect on the sampling as the equiangular importance sampling of lights (Kulla and Fajardo 2012): the PDF is large where a large in-scattering contribution is expected. While generating a fluence estimate from the cache points is more flexible than equiangular importance sampling, it is also more expensive due to the searches for the nearest cache point.

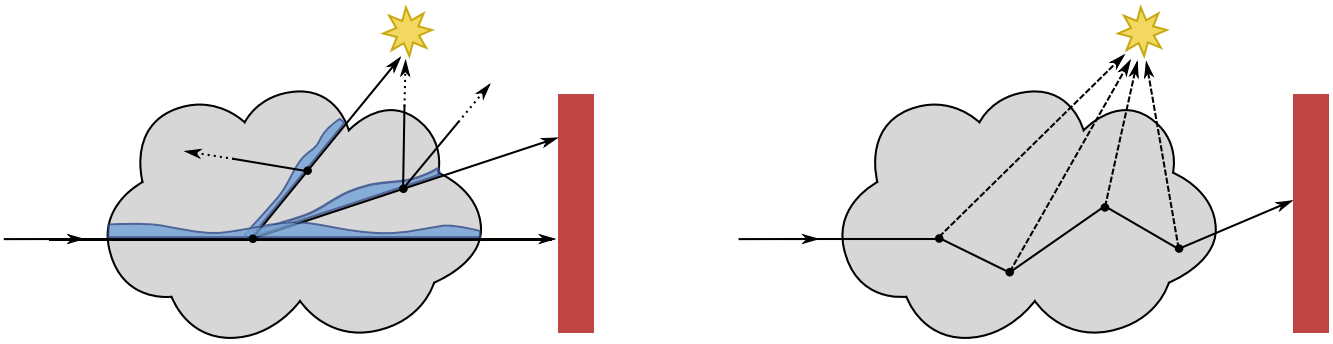
To amortize the computational cost associated with expensive PDF construction, the PDF is sampled multiple times and splitting is employed to generate multiple in-scattering rays, which are enqueued into the ray-batch system.

By using a large splitting factor, this strategy is efficient at rendering low-order scattering, producing high-quality estimates at a high cost per sample. Unfortunately, recursive invocation becomes prohibitively expensive for high-order scattering, such as in clouds and other high-albedo volumes.

4.5.1 Tracking-Based Volumes. While higher order scattering can be approximated with various tricks, we found that consistent and realistic cloud appearance was achievable only by integrating all of the high-order scattering, up to hundreds or even thousands of bounces.

To achieve the high-order scattering, we decided to use lighter-weight tracking algorithms which do not build a PDF explicitly as the previous volume system did. In the tracking approach, the ray scatters and changes direction inside the volume, in contrast to being fully traced to the next surface intersection and spawning child rays along the way (see Figure 11). The resulting estimates are of much lower quality than in the previous strategy for direct illumination and transmittance, but also come at much lower cost and therefore allow many more bounces to be calculated. A more detailed description of the difference of both strategies is given in Fong et al. (2017).

Standard delta tracking, as a monochromatic estimator, limited our use of colored volumes. In our recent publication (Kutz et al. 2017) we introduced a *spectral tracking* algorithm that removes the restriction of delta tracking being limited to monochromatic extinction, as well as a *decomposition tracking* algorithm that offers the same sampling quality as delta tracking with fewer volume lookups. Leveraging the mathematical framework underlying our new tracking algorithms (Galtier et al. 2013), we have also been able to extend our trackers to gather heterogeneous volumetric emission at multiple points along each ray—we can gather emission at all collision locations by setting the absorption/emission probability to one regardless of the scattering and null-collision probabilities. To perform empty space detection and deliver the



(a) Old system (b) New system

Fig. 11. In our old volume system (left), before shading a surface hit, a detailed PDF (blue) is built along the ray segment encoding transmittance, extinction, and a fluence estimate generated from the cache points. In-scattering samples are drawn and enqueued into a future ray batch. In our new volume system (right), a ray enters the volume and scatters many times based solely on the local scattering properties of the volume. Shadow rays (dashed) are used in conjunction with next-event estimation to importance-sample direct lighting contribution and enqueued into a future ray batch only after completely exiting the volume.

volume parameter bounds for the tracking, we use an adaptively built octree using the partitioning metric of Yue et al. (2011).

4.5.2 Similarity Relation. As highly forward-scattering media, clouds present difficult performance challenges. Anisotropic scattering distributions increase noise and require more samples to adequately converge, and short scatter distances require more rays to be traced.

Transformations of volume parameters that result in similar radiance distributions are referred to as similarity relations (or sometimes similarity theory) and allow volume parameters to be changed while keeping the large-scale appearance unchanged. For high-order scattering, we desire to reduce the highly anisotropic scattering toward more isotropic scattering. A first-order relationship between scattering coefficient μ_s, the mean cosine of the phase function g, reduced scattering coefficient μ_s^*, and reduced mean cosine g^* was derived by van de Hulst (1974):

$$\mu_s(1-g) = \mu_s^*(1-g^*).$$

Using the reduced scattering coefficient and mean cosine substantially improves render times by increasing scattering distances and reducing the total number of bounces, and it significantly reduces noise by making direct light sampling more effective. We find that smoothly interpolating between anisotropic and isotropic scattering based on bounce depth provides a nice balance between improved efficiency and accurately reproducing important optical effects. In particular, we currently linearly interpolate g^* from g to zero between the 5th and 20th bounces and then solve for μ_s^* using the aforementioned relationship. The lower and upper bounds of the extinction coefficient in a particular region of space, used by our trackers, also need to be interpolated appropriately.

4.5.3 Light Sampling Without Path Splitting. In our previous volume system, the path was split at each scattering event to combine light sampling and phase function sampling techniques. As with our surface scattering, each scattered ray was allowed to continue the path equally, gathering both direct and indirect radiance. While this strategy worked well for surface-surface and surface-volume interactions with low-albedo volumes, it potentially produced an unbounded path tree in a high-albedo volume. To prevent this, we required aggressive Russian roulette as well as an independently controlled volume-bounce limit to prevent ray explosion.

In the new volume system, to enable high-albedo volume integration with unbounded path length, we have eliminated path splitting. Instead, rays generated via light sampling are no longer allowed to undergo further scattering either in the volume or from a surface.

Before being dispatched into the ray-batch system, light sample rays are first traversed immediately through the volume to compute transmittance, using Russian roulette to prevent overwhelming the batch system with such rays in a dense volume. When a surface might penetrate the volume, immediate surface traversal is performed to limit the integration distance through the volume. Surface shading is still deferred through the batch system as before. Figure 11 outlines the differences of the two volume integration strategies, and Figure 12 shows equal-time renders of the same cloud using the two volume systems.

5 MAKING PATH TRACING PRACTICAL FOR PRODUCTION

In this section, we discuss how we process, troubleshoot, and art-direct rendered images.

5.1 Dealing with Monte Carlo Noise

Residual noise is a fundamental concern in a Monte Carlo renderer like Hyperion. We use several techniques to cope with noise, improving visual quality or saving render time by allowing us to reduce SPP.

5.1.1 Adaptive Sampling. Noise in a Monte Carlo render is distributed nonuniformly since the light paths and materials vary

(a) Old system (b) New system

Fig. 12. Equal-time comparison of a cloud rendered using our old *Big Hero 6*-era volume rendering system (left) and our new tracker-based system (right). The old system is limited to 254 bounces (somewhat low-order scattering for high-albedo volumes), yet still produces a noticeably noisier image than the new system. The new system produces a less noisy image while also supporting efficient high-order multiple scattering.

across the image. Rendering the entire image with a constant SPP would waste samples on the less noisy areas. Ideally we would want each pixel to use the minimum SPP necessary for that pixel.

Hyperion's adaptive sampler interprets the requested SPP as an overall sample budget rather than a per-pixel maximum. At the start of each iteration, samples are allocated to each pixel proportional to the pixel's variance (see Section 5.1.6). The variance estimates are first blurred because the estimate is noisy and pixels near a high-variance, difficult-to-sample pixel are likely also difficult to sample even when their variance is low. Since variance in motion-blurred images tends to smear along the direction of motion, this blur uses an anisotropic kernel based on the structure tensor of the variance image.

We have encountered several adaptive sampling challenges:

- **Fireflies:** To prevent the adaptive sampler from sending so many samples to fireflies that the rest of the image suffers, we clamp the variance estimate and variance drives the number of samples at a less than linear rate (though this may limit adaptivity).
- **Matte Render Passes:** These render passes output mattes and geometric information without rendering color. Since noise in rendered color cannot guide the adaptive sampler, it uses the variance of the mattes themselves.
- **Sample Budget for Excluded Objects:** Artists often partition the scene into separate render passes such as character and environment passes (see Section 5.3.4) where excluded or "held out" objects are rendered as black on camera hits. Ideally, separating objects into passes should not significantly affect the overall sample budget for the scene as held-out objects should require minimal samples. To achieve this, Hyperion's adaptive sampler uses a convergence threshold to detect such pixels, and once a pixel is deemed converged, its remaining SPP is removed from the render process's sample budget. This is a conservative threshold that most pixels never reach, intended solely to prevent sample budget inflation due to held-out objects.
- **Alpha Inconsistency Between Render Passes:** Separate render passes are composited simply by adding their color and alpha. Adaptive sampling can cause different SPP counts to be used for the same pixel in each pass, and thus different alpha sampling. This can cause compositing artifacts such as leakage along the edge of geometry since alphas do not sum to exactly one. We solve this with our *consistent alpha* feature, which applies adaptive sampling to color but renders alpha using the same number of samples for every pixel and render pass.
- **Subpixel Objects:** Isolated subpixel objects such as dust particles can be entirely missed if the convergence threshold considers the containing pixel "done" before the object is hit for the first time.
- **Adapt Sampling to Denoising:** We plan to integrate denoising into the adaptive sampler, similarly to the approach of Rousselle et al. (2012). This would drive adaptive sampling by the denoiser's estimate of post-denoising variance, saving render time on areas which denoise well, though it does require the renderer to run the denoiser after each iteration.
- **Adaptivity Limited by Iterations:** The adaptive sampler updates the sample allocation only at the end of iterations so it cannot fully equalize noise levels (e.g., a 128 SPP render had only three opportunities to adapt—after 16 SPP, 32 SPP, and 64 SPP). Reducing iteration sizes would allow more adaptivity, but smaller iterations would be less efficient (see Section 6.2).
- **Localized SPP Control:** Users have suggested adding user-specified per-object SPP controls but that would run counter to our philosophy of simplicity over flexibility. When noise problems arise, we prefer to investigate the underlying causes (e.g., suboptimal importance sampling in a particular shader) or address the problem more broadly (e.g., adaptive sampling).
- **Post-Render Compositing Adjustments:** Exposure adjustment in compositing (Section 5.3.8) changes the visibility of noise in different parts of the image, so the adaptive sampler's goal of spatially uniform noise levels is not optimal. Consequently, we discourage making extreme composite-time exposure adjustments.

5.1.2 Denoising. A major concern when we decided on Monte Carlo rendering and Hyperion for *Big Hero 6* was long render times. It would have been impossible to render the movie on time without post-render denoising.

Our denoiser is inspired by the work of Li et al. (2012) and Rousselle et al. (2013). See Figure 13 for an example from Moana. It spatially and temporally filters the rendered images using a spatially varying kernel, trying to smooth away noise without blurring the image's details.

Hyperion writes and the denoiser reads a "feature" image, which contains these layers:

- **Diffuse** and **specular:** the rendered color decomposed into separate "specular" and "diffuse" parts; radiance contributions are splatted into diffuse and specular framebuffers proportionally to the relative strengths of the diffuse and specular lobes of the first surface the camera ray hits.
- **Albedo:** the surface color (including texture)
- **Surface Normal Direction**
- **Depth**
- **Forward** and **backward motion vectors**
- **Variances** of diffuse, specular, albedo, surface normal, and depth (see Section 5.1.6)

To do temporal denoising, the denoiser reads the previous, current, and next frames (requiring the denoiser to run after all three frames are done rendering). The denoiser uses motion vectors to pre-warp the previous and next frames to the position of the current frame.

The denoiser does a weighted average of nearby pixels and adjacent frames, with a different non-symmetric kernel at each pixel. Weights are calculated to draw from nearby pixels according to the similarity of their color and features. The sensitivity of similarity is determined by the corresponding variances.

Fireflies have extremely high amplitude, which could cause a bright "smudge" on the image even after denoising. Our denoiser detects fireflies as pixels with much higher variance than their neighborhood's average variance. It fully removes those fireflies, replacing them with data from their neighborhood. This actually causes energy loss (darkening of the image) but it is preferable to preserving the energy as a bright smudge.

Our original denoiser excessively blurred the fine detail in hair and fur: the denoiser is cued by the surface-normal-direction feature, but that is not well-defined for thin curves like hair (whose normal varies across the subpixel diameter of a hair). On *Big Hero 6*, Hyperion wrote zeros to the "normal" feature; the denoiser detected that and limited the amount of filtering on hair to prevent blurring hair's details. This worked well enough since that show's characters mostly had dark hair.

Some characters in *Frozen Fever* and *Zootopia* had blonde hair and bright fur, where detail is more visible and noise is a bigger problem than on dark hair. The denoiser initially either kept too much noise or blurred details too much. We changed the hair shader to output hair's *tangent* direction into the feature image's "normal" layer. The tangent direction of hair is well-defined and stable across a pixel and it is well-correlated with visible features in the image. This let the denoiser effectively filter hair without over-blurring it.

On *Moana* we found that the denoiser blurred details in Te Kā's volcanic smoke clouds, which are rendered volumes. The problem was that volumes lacked the geometric features which normally cue the denoiser: surface normal direction, surface color, and depth. We solved this by making the volume renderer calculate normals based on the volume's gradient. This is especially useful for dense detailed volumes such as smoke clouds.

The denoiser is used on most production shots, and is run automatically unless disabled. The denoiser lets artists render at 1/4th to 1/8th as many samples per pixel as they would otherwise need, saving significant compute time and giving faster turnaround.

5.1.3 Manual In-Render Techniques. For compositing flexibility, artists usually split the scene up into separate render passes for different categories of elements, such as characters versus the environment. This also lets lighters choose the SPP which is best for each render pass instead of using the worst-case SPP for all the elements.

When a particular light is causing noise, artists may split that light out into a separate render pass. The pass often renders with less noise since no other lights draw light samples; all light samples are aimed at the problematic light. Hyperion's emission-category render outputs (Section 5.3.5) make it easy to tell if noise is due to a particular light.

5.1.4 Manual Post-Render Techniques. When an element looks the same over several frames, the lighting artist may render a single source frame at high SPP and use that *held frame* for several target frames in the composite.

Reprojection is a generalization of a held frame; it reuses a single high-SPP source frame for several target frames even if the element or camera is moving. The composite reprojects the source frame onto the element's surfaces of a target frame using that frame's camera transformation and geometry. Where a surface is visible in the target frame but not in the source frame, the composite outputs a *sliver render mask*, which Hyperion uses to render only the requested pixels.

On scenes with limited motion, we sometimes do motion-aware multiple-frame averaging in the composite, making sliver renders if necessary to fill in small areas of disocclusion.

Lighters sometimes apply defocus blur and motion blur in compositing, turning those effects off in the rendered scene and preventing associated noise.

5.1.5 Filter Importance Sampling. Early versions of Hyperion used a classic image reconstruction filter, which splatted each sample to a three-by-three pixel region of the framebuffer (weighted by the filter). This caused a sample's noise or firefly to affect multiple pixels, making denoising and firefly removal difficult. It also made variance estimation more difficult since noise is correlated among neighbors.

We solved those problems with filter importance sampling (*FIS*) (Ernst et al. 2006). We splat each sample to only one pixel, weight all samples equally, and choose sample locations relative to pixel centers with likelihoods proportional to the reconstruction filter's weight.

(a) Raw rendered image (b) Denoised rendered image

Fig. 13. Our denoising process takes as input an unconverged render (left, from *Moana*) and a number of feature buffers, and produces a final-quality image with minimal artifacting and smoothing (right). In this example, the input unconverged image was rendered with 128 SPP.

With FIS, noise is uncorrelated between pixels, and appears as higher frequency "salt and pepper" noise rather than "blurry" noise. Ernst claims that FIS reduces noise since samples no longer have implicit reconstruction weights, about which the light and surface sampling system would be unaware.

5.1.6 Variance Estimation. Hyperion estimates the variance of each pixel's rendered color and its diffuse and specular components to drive adaptive sampling, guide the denoiser, and report MSE statistics to the user. The naive approach would compute the variance of the individual samples' contributions to a pixel. But it would be difficult to compute sample variance in Hyperion since multiple bounces of a single camera sample are splatted onto the framebuffer at different times—we never have access to the total contribution of a single camera sample. Using sample variance would also bias the variance estimate since low-discrepancy samples are correlated rather than independent samples (Rousselle et al. 2012).

To avoid these difficulties, Hyperion estimates rendered color variance using a "two-buffer" approach (Dmitriev and Seidel 2004; Rousselle et al. 2012). Hyperion has a pair of "half-sample" framebuffers for color (and for specular and diffuse). Camera samples are partitioned into one or the other of these buffers in a randomized shuffled order. We prevent cache points from coupling correlated noise into both buffers by generating independent cache points for each buffer. At the end of an iteration we estimate the variance of each pixel from the difference of the two buffers. This estimate is noisy so the adaptive sampler and denoiser spatially filter it.

In addition to variance of diffuse and specular rendered color, the denoiser needs the variance of the albedo, normal, and depth features. These are estimated as a simple sample variance. This fits easily into our architecture since the values of these features are known on the first hit; we do not need to wait for results from subsequent bounces.

5.2 Analyzing/Understanding Render Processes

While one of the commonly stated advantages of path tracing is simplicity, a complete full-featured system for production rendering inevitably comes with a certain level of complexity. To allow artists and technical staff to get the most out of the software, the system needs to be comprehensible, user-friendly, and as transparent as possible. Most commonly, a user will want to ask where the processor time is being spent, what is expensive in terms of memory use, or why the rendered image does not meet their expectation.

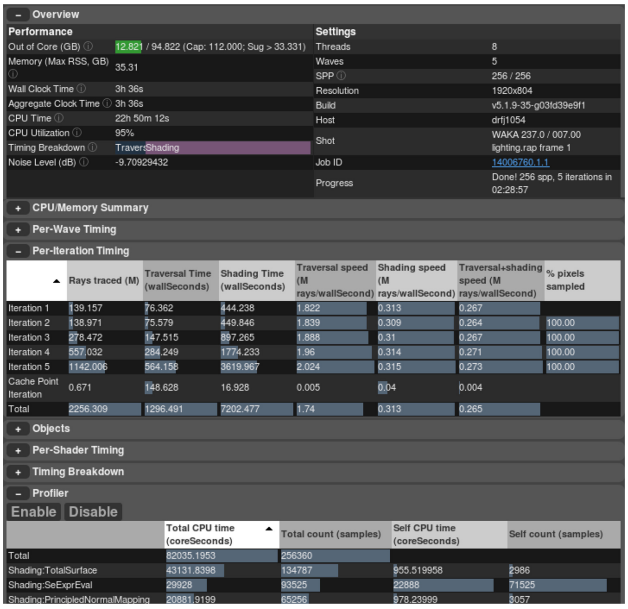


Fig. 14. Hyperion's statistics viewer organizes, formats, sorts, and searches render statistics.

In addition to a typical render output log, Hyperion provides a comprehensive statistics reporting system. Statistics are available both for in-progress renders and after the processes finish, in JSON statistics files and as metadata in OpenEXR images.

Hyperion's interactive statistics viewer (Figure 14) is a JavaScript/web presentation of render statistics. This is available in Hyperion's interactive render viewer, in a stand-alone program, or in a web page served by Hyperion's internal web server or an intranet web server.

Due to the scale of production scenes, the statistics content itself can become overwhelming—"information overload" kicks in

quickly. The breakdown and formatting of statistics into categories with sortable columns makes large amounts of data much more digestible. An example would be viewing a table of scene objects and sorting by expense, whether that is memory footprint or shading time. Once a particularly expensive shader has been identified, live profiling can be enabled to identify hotspots at a finer level of granularity. The output statistics can also be consumed by automated processes—we are able to identify scene objects that make a negligible contribution to the final image as candidates for culling, by finding surfaces hit by a very small number of indirect rays.

Other useful tools for render analysis are a path inspector, also integrated into our interactive viewer, diagnostic render outputs and shaders, and a variety of options for geometry export. A developer might wish to write out displaced tessellated meshes, cache points, or bounding volume hierarchies for debugging purposes when investigating support issues from artists.

Where possible, we try to associate errors with a scene object and/or an image location. During shading, assertions can throw exceptions that are then caught by a custom handler, an example being attempted normalization of a zero-length vector. The corresponding pixel is then marked with an "X" using negative values. This allows the markers to be easily removed from the image, and provides prominent visual indicators of problematic areas in addition to lists of errors or warnings by type.

In terms of errors and warnings, the general philosophy for Hyperion has seen a shift during development. Initially, problems that stem from the input scene were not just flagged but handled in a manner that forces fixing of the data. Over time, this strictness has moved more toward resilience—the system is attempting to produce the best result it can with what it is given, while still producing localized reporting to encourage user correction of scene data.

5.3 Artistic Control in a Physically Based Renderer

While Hyperion is a powerful path tracer with an array of features that make creating beautiful photorealistic imagery a straightforward process, artists at Walt Disney Animation Studios are rarely asked to just do that. Every frame of the film is a carefully crafted work of art, aspiring to the vision described by the directors, art directors, production designers, and lighting leadership. Artists are constantly challenged to hit decidedly non-physical goals within a limited amount of time and with finite compute resources. This section explains how artists use various tools, both in and outside of the renderer, to achieve art direction and efficiently craft the final frame.

5.3.1 Transport Modifiers and Exclusive Lights. Sometimes accurate global illumination may not achieve the desired artistic effect. Certain objects may bounce too much or too little indirect light into the environment. Or the director may give a note specific to just one element while approving the rest of the scene's lighting. In these and other cases, the light transport must be altered in a localized way.

Using *transport modifiers*, a lighter can scale the light transport between two elements (objects or light sources), or groups of elements, within a scene. For example, an artist could define a transport modifier to cut the energy between a particular object and light by half. When a ray hits the object, the renderer records the interaction by attaching a tag to outgoing rays. If a ray from the object reaches the light, the system recognizes that a relationship exists, and cuts the throughput by half for that ray. But if the ray hits anything else, the transport modifier is disregarded.

A variant of transport modifiers can be used to localize lights to particular elements. Lights marked as *exclusive lights* contribute only to objects that have a defined relationship with the light. An artist may add a rim light exclusive to a character, for instance, because there is a certain appeal to a slightly backlit character even when there is no physical basis for it, and because it helps separate the character from their environment to focus attention on them. Global lights can be locally altered in a similar way. For example, the artist may widen the key light just for a character to create softer shadows on the character than the environment. To do this, the artist could use a transport modifier to eliminate the global key light's effect on the character and add a widened key light exclusive to the character.

Transport modifiers are a powerful tool for artists to control light transport in non-physical ways in their scenes, but have limitations as well. Control between a few small groups of objects, as described above, is simple to describe. However, to control light transport between many large groups of objects, the data-management burden begins to grow, especially when complex inclusion/exclusion relationships are necessary.

5.3.2 Region Grads. Similar to Pixar's *Rods* (Hery et al. 2016), we allow artists to adjust light contributions within volumetrically gradated shapes or regions within the scene. Additionally, our *region grads* can be used in arbitrary ways within material shader expressions. For instance, a *geometry mask* expression can reference a region grad to cut away part or all of a surface, for example, to let light show through a hole in an off-screen object, or aspects of surface reflectance can be modulated, such as reducing the specular response. Region grads thus give artists great flexibility and locality in addressing director's notes.

For rendering efficiency, region grads are implemented as part of shading rather than as a volumetric effect. Light shaders evaluate the region grad using the origin of the ray, allowing the light's contribution to be modulated at the lit surface, for instance to accentuate the shadowing. Surface shaders evaluate the region grad using the surface hit point, with the result able to be used for any part of the material shading.

5.3.3 Shadow Shaping. Lighters often want very specific control over light and shadow in achieving the art direction of their scene. They craft custom IES profiles (Illumination Engineering Society of North America 1991) to control the emission of a light, and use cucoloris images to craft specific light and shadow shapes in the image. This can be a very labor-intensive process, depending on the specifics of the control they are trying to achieve, but is used often either to fake a particular shadow that is not in the scene, or replace a shadow from the environment that is affecting the character in an unappealing way. Note that replacing a shadow requires two render passes, one to render the object casting the unwanted shadow, and another with that object removed and the replacement shadow being cast onto the other objects in the scene.

For a cucoloris effect, an artist could simply place a desired shadow shape in the path of the light, as in a practical spotlight, but the shape would be blurred out unless the spotlight's rays were perfectly focused (which is never the case). Instead, our light shader projects the cucoloris image to the lit surface, using an artist-defined projection, and then multiplies the cucoloris into the light's overall contribution at each surface point. While non-physical, this gives artists the control they need.

5.3.4 Render Passes. The image generated by the straightforward execution of Hyperion is rarely the final image used in a film, even using all of the controls described above. Artists separate and reassemble various outputs of the render process to achieve a given art direction in a given amount of time.

Render passes are completely separate renders for different components of the image generated by artists. Characters are often broken out into their own pass, for example, but the artist may also break out particular foreground objects or objects they know they will want to manipulate after rendering. There are two main reasons for this, and both are factors in almost every case. Render passes give artistic control over some facet of the image, and also allow an artist to iterate without having to re-render the entire frame each time.

In a global illumination renderer, each render pass must still process the entire scene for accurate indirect light transport. This is not as inefficient as it seems. For instance, in a character pass, the environment is "matted out," rendering black on camera hits and contributing only to indirect lighting. Though there is still additional overhead with each pass, the total number of pixels that compute full GI is roughly the same as a single-pass render with no matting.

There are cases, however, where the artist deliberately excludes objects from some passes, for control or for render management. A particularly heavy volume pass, for instance, may be rendered in a separate pass at a lower resolution or with fewer ray bounces, and left out of other passes to speed them up, even accepting that the effects of the light transport through the volume on those other passes may have to be faked.

5.3.5 Render Outputs and Emission Categories. Within a single render pass, the resultant image is sometimes important, but more often artists identify components of that image that they want to assemble artistically. While rendering a character, an artist may want to manipulate the hair separately from the character, for example, using a matte generated from the hair. Or they may want to output contributions from specific light groups into separate images for post-render rebalancing.

Various types of *render output* images allow artists to gather the building blocks they will want to reassemble in creative ways when compositing the final image, all rendered in a single run. For instance, each light can be assigned an *emission category*, with individual categories producing separate render outputs.

5.3.6 Path Classifications. An artist may want to not only change the effect of a given collection of lights on a character's hair, for instance, but may want to dial the effect in its various forms. The direct diffuse illumination from that collection of lights may be fine, but the specular may be too harsh. Or the direct illumination from that collection of lights may be fine, but the indirect diffuse contribution of those lights bouncing off a nearby surface onto the character may be too bright, or not saturated enough. In such cases, *path classifications* allow an emission category to be broken down into its constituent types of light paths. Additionally, path classifications can be applied to transport modifiers to localize the modification to a portion of the light transport.

Path classifications are similar to "light path expressions" (Heckbert 1990), but differ in key ways. A fixed set of path classifications, *light source*, *direct specular*, *indirect specular*, *direct diffuse*, *indirect diffuse*, *caustic*, and *volumetric*, provides a simple, consistent interface to artists that directly addresses their needs. Having a fixed set also allows efficient implementation, requiring only a few bits on each ray to represent the current path type, and a trivial state machine to update the path type with each scattering event. Importantly, path classifications, like emission categories in general, are energy conserving by design—because the classifications are mutually exclusive, and because each light can only be assigned to one emission category, any given ray contributes to only one render output. Artists can trust that if they add all the emission render outputs together, they will get identical results to the non-decomposed image.

5.3.7 Mattes and Deep Mattes. Artists use mattes they generate from renders, as well as those they craft from the results of renders using custom tools in the compositing engine. This gives them further flexibility to break apart render outputs into smaller components, to nuance some particular part of an image. Eye mattes are very common because there is often considerable art direction to make the face and especially the eyes appealing.

Deep mattes provide additional flexibility beyond ordinary "flat" image mattes. A deep matte contains various attributes such as mesh name, element name, material name, and so forth, that may be keyed off of to derive various flat image mattes at composite time. Unlike more general deep output images, these are quite efficient. They are generated using a low-SPP utility pass requiring only camera hits, and samples within each pixel are collapsed to the smallest unique set of attributes. The attributes themselves are also stored as integers, using the same hash function as in the compositor to allow artists to select by name.

5.3.8 Putting it all Together. In-render tools like transport mods and region grads are powerful but require re-rendering to make adjustments, and sometimes due to resource contention or schedule pressure, re-rendering can be a prohibitive requirement. For this reason, artists tend to over-generate render outputs to give as much flexibility as possible in responding to director notes on their scene. A final composite graph will have hundreds of nodes in it, reassembling the path classifications of the emission categories of the many render passes that were broken out from the final image, crafting a final image that achieves the artistic goals on an often demanding schedule. Each artist finds a balance between the artistic control these hundreds of distinct images can give, with the gigabytes or even terabytes of disk space they can take up and the slowness that can ensue in interacting with so many large files in the final composite graph. Over the course of a film, aggressive disk management has to occur to keep artists from filling the space

(a) Raw rendered image (b) Final frame

Fig. 15. Artistic manipulation of a rendered image. Physically based path tracing produces beautiful imagery, which is further enhanced and modified by artists to meet art direction needs. Ultimately, physical accuracy is only an important starting point for meeting aesthetic requirements. Note how in going from the raw render to the final frame, the artist has reduced the sunlit highlight from young Moana's face, lowered the exposure of the foam on the water, brought out additional detail beneath the water surface, and added haze to distant cliffs in the background, among other things.

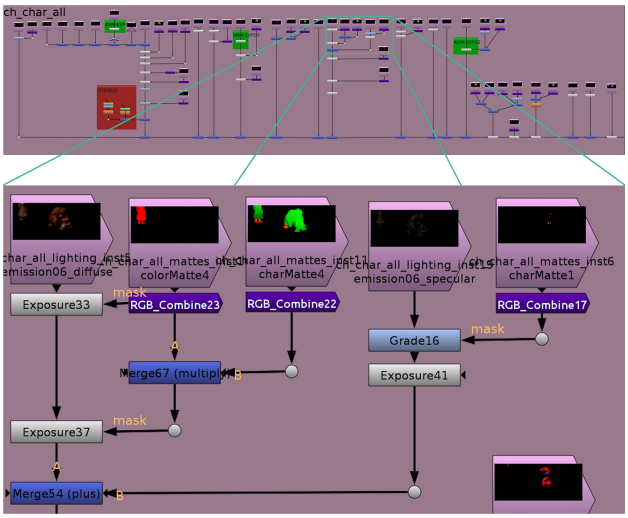


Fig. 16. Compositing graph of just the character layer of Figure 15, showing the dozens of render outputs the artist uses to assemble the final frame.

they have, because of this flexibility. See Figure 16 for a portion of the compositing graph for the image shown in Figure 15.

There are limitations to this workflow. Just as digitally increasing the brightness of a real photograph can reveal noise in underexposed areas, pushing a render output too far in the reassembly process can amplify noise or other artifacts. This is usually a sign that the scene should be modified and re-rendered to generate something closer to the desired result. When schedule pressure or resource contention will not allow this, the artist is forced to massage the existing render outputs into something usable. Sometimes the most expedient option is to just paint over the final image assembly, and there may be times where doing so is the only option, where the artist may have not broken out all of the layers that they needed, or the render process may be misbehaving in some subtle way that is spotted only in the final frames, and re-rendering is not possible.

In the shot shown in Figure 15, the artist used many of the techniques discussed in this section to achieve the desired art direction. As was frequently true on *Moana*, the artist used separate path classifications in the composite to tune the water color and reflectivity of the water surface. The water also reflected too much light onto the characters at various points in the shot and this reflected light was an unpleasant hue. While the artist was able to use a transport modifier on Pua the pig, this did not work as well for Moana or Chief Tui. As those characters moved away from the water, they became too dark, which meant additional lights were needed. The artist added additional exclusive lights that only illuminated the characters and then used a combination of mattes and rotos to blend the various emission categories from the original light and the exclusive lights. This enabled them to achieve a seamless lighting transition as Moana is picked up out of the boat. Finally, the background cliff was an expensive and noisy render, which also had several art-direction notes. To save time, the artist only rendered the first and last frames and projected a blend of these two frames back onto the geometry in the composite. This allowed the artist to make several paint fixes and adjustments using the mattes and rotos that would have been very difficult to achieve in a single rendered image.

While all these manipulations may seem contrary to the notion of a physically based renderer, our goal is not academic perfection, but rather to make beautiful images of compelling worlds that feel tangible.

6 DISCUSSION AND LIMITATIONS

6.1 Production Impact

Hyperion has made a significant impact on our filmmaking by improving artist efficiency, raising the visual quality and consistency, and allowing our film complexity to scale significantly beyond what we were previously able to achieve. However, there are still challenges. Achieving art direction on an interior illuminated entirely from outside can be difficult. Finding the right tradeoff between artistic controllability versus the visual richness provided by strong multi-bounce indirect illumination can be challenging. Hyperion can render a massive amount of geometry, so of course our films have taken that as a challenge to put more geometric complexity on screen. Render times can be prohibitive under schedule pressure, warranting additional non-rendering solutions to meet

art direction. Certain forms of light transport, such as through complex volumes, continue to be expensive.

There are also challenges outside the creation of final frames. Hyperion is a powerful diagnostic tool, and gives many insights that upstream departments can use. Technical animation artists use renders to judge whether hair penetration is noticeable and needs fixing. Animators can judge a performance in the context of the final lighting to understand what to reduce or accentuate. These use cases come at a cost, though. Compute resources are at a premium (especially under deadline pressure), so launching renders for diagnostic purposes poses a bandwidth challenge as every department is rushing to finish its work. To combat this bandwidth constraint, upstream departments often launch renders at lower samples-per-pixel. Lower-quality renders help alleviate compute resource contention, but has at times allowed artifacts to slip through that higher samples-per-pixel renders would have shown. Therefore, trading off compute time against success at catching problems becomes a balancing act throughout the last few months of production.

Overall, having a powerful in-house renderer and a team of dedicated people evolving it to meet new challenges has been a huge success. Having full control over Hyperion's implementation allows us to provide targeted, deeply integrated solutions, and gives us agility to quickly respond to production problems as they arise.

6.2 Ray Batches: Interactivity versus Throughput, Batch Starvation, and Path-Length Limitations

Hyperion achieves the highest throughput when it has full batches. The end of an iteration always includes some number of partially filled batches with accordingly degraded coherence. Reduced sized batches are less of a problem for final quality renders, where a few inefficient batches out of hundreds per iteration are barely noticeable. However, for more immediate feedback in interactive sessions, Hyperion renders the first iteration in progressive fractions, starting with as few as 4 SPP and 1/256th of the pixels. In this initial interactive progression, the batches are far from being filled.

Also, rendering with long paths can produce a string of many nearly empty batches toward the end of an iteration when only a few paths remain. In addition to reduced coherence, such small batches can starve the system from having enough work to keep all the cores busy. Even with full batches, the batch system has a dip in core utilization as synchronization is required when the system switches back and forth between traversal and shading phases. For these reasons, our current architecture has diminished scaling beyond 16 cores for typical scenes, and would not likely be able to utilize the hundreds of cores anticipated in future computing platforms.

To address these issues, we are investigating ways to overlap traversal and shading, to use dynamic queues instead of fixed batches, and/or to overlap iterations; recent work such as that of Lee et al. (2017) provide encouraging validation for more queue-centric architectures.

6.3 Coherent Traversal and Out-of-Core Efficiency

Out-of-core rendering is fundamentally a tradeoff between ray intersection bandwidth and accessing data from disk; since accessing data from disk is by definition slower than accessing data in memory or in cache, efficiency has to be regained by increasing the number of rays that can be processed per out-of-core data access and decreasing the total number of out-of-core data accesses required.

A simple approach to achieve maximum coherence within a single thread would be to trace all of the rays in a batch against one scene object at a time, and proceed through scene objects front-to-back through the scene. This approach could be naively parallelized by intersecting all of the rays against the current scene object in parallel. However, this strategy has two major shortcomings: first, the system has a synchronization point at each scene object, and second, waiting for a scene object to load blocks all threads.

Instead, in our current cone-based traversal system, we split traversal into multiple queues at each top-level-hierarchy node, with each thread processing a different queue. Since multiple threads can operate on different queues for different scene objects at any given time, object loading does not block all parallel threads. To alleviate the situation where all the rays hit the same object at the same time (which often occurs in the first batch where nothing has been loaded yet), we also multi-thread traceable initialization steps such as tessellation and displacement.

The problem with "spreading out" the traversal in this way is that rays that miss an object must then be queued at the next object in the traversal order, but that object may have already been processed in another thread. Visiting each object multiple times per batch impacts our out-of-core efficiency by increasing the number of times traceables need to be reloaded. Additionally, when reloading a traceable, we reprocess it from scratch (reading the mesh description from disk, tessellating, displacing, and building a BVH) making it difficult to amortize the cost even with optimal coherence.

Our need for out-of-core geometry has been somewhat mitigated by increasing available memory, more extensive use of instancing, and by optimizing the memory footprint of geometric data structures. Nonetheless, we are experimenting with per-object ray queues which defer processing rays for a scene object to accumulate as many rays as possible and amortize the load cost, and we are also investigating ways to pre-generate and cache our ready-to-trace geometry to reduce the load cost.

6.4 Packet versus Single-Ray Traversal

For a packet of rays, our packet traversal is significantly faster than traversing each ray individually due to the efficiency of our cone–box intersection. Additionally, our queuing overhead is reduced through the use of packets, requiring up to 32× fewer items in each queue. However, when coherence degrades with small batches, the packets' cones become very wide and lose their culling power in the top-level hierarchy. This is an occasional inefficiency we have lived with, though we are considering disabling packets or perhaps switching from packets to individual rays dynamically when we detect that the packet is much larger than a given BVH node.

6.5 Managing Geometric Complexity

Even with the steady growth in computational and memory capacities, production demand for increased geometric complexity

continues unabated. This growth is only partly due to increasing visual complexity. Artists also use more detail than strictly necessary because there is often a productivity benefit for doing so. Tight production schedules leave little or no time for optimizing over-modeled assets or authoring low-detail versions. Artists also often find it easier to use finely detailed geometry to control reflectance even when the details are not distinctly visible to the camera; examples of this are flyaway cloth fibers, vellus hairs on characters' skin, grains of sand or ice on the ground, or dust particles on surfaces.

The sheer amount of geometry in our renders has forced compromises. We would prefer to always tessellate surfaces to a subpixel size (something we could routinely do with Reyes), but usually choose not to in order to conserve memory. Though our tessellation rate is usually quite fine, it can still occasionally be coarse enough to cause visible pops when tessellation changes with slow moving cameras. When necessary, these artifacts can be addressed by fixing tessellation from a static representative camera position. The lack of subpixel tessellation also requires the use of *shading normals* to give the impression of the missing subpixel detail, but shading normals are far from ideal, adding significant cost by requiring displacement derivative evaluation for every shading point (as opposed to simple displacement evaluation during tessellation), neglecting shadowing between fine surface features, and introducing artifacts when the shading normal differs greatly from the geometric surface.

6.6 Attempts at Automatic Level-of-Detail

Level-of-detail (*LOD*) approaches can reduce memory usage, render time, noise, or all three. Given that artists seldom have time to manage manual LOD solutions, automatic solutions are appealing. However, though we have made a couple of attempts at automatic level-of-detail in Hyperion, we have yet to find an acceptable approach.

Our initial LOD attempt in Hyperion, inspired by the R-LOD work by Yoon et al. (2006), fit planes to internal BVH nodes approximating leaf geometry. We terminated our traversal when the ray footprint was larger than the BVH node. Initial problems with rays leaking through geometry led to the use of bounding slabs instead of approximating planes. Artifacts were still a problem but the biggest issue was that we observed neither speed nor memory gains. The potential savings of shortening the depth of traversal were offset by the additional traversal complexity.

Our next attempt was inspired by the Sparse Voxel Octree (*SVO*) work of Laine and Karras (2010). Similar to our previous attempt, we fit bounding slabs to the contents of octree nodes and also fit directional normal distributions. With this system, we could aggregate arbitrary geometry with memory reductions of 10–100× for distant objects. Unfortunately, we found that aggregating some surfaces produced objectionable artifacts and artists would need to enable SVOs selectively with explicit LOD groups. Further inhibiting adoption was the fact that SVOs were in general slower than the non-LOD geometry and also added significant startup time to the render process. While these issues may be surmountable, the lack of robustness and need for artist management led us to abandon the approach.

One component of both LOD approaches that was deployed into production was a simplified shading model. Our "shading LOD" precomputed and stored parameters of our BSDF model at triangle vertices. As before, when the ray footprint was larger than the triangle being shaded, shading LOD would be used instead of the full shader evaluation. While automatic and mostly successful, it increased rather than reduced memory use given that it only used the shading LOD for some rays. Worse, it sometimes slowed the overall rendering iteration since BSDF-parameter evaluation on start-up happens on every triangle vertex, even on distant geometry where triangles are subpixel. There were also visible artifacts that required disabling the shading LOD for some surfaces. While this was not by itself an excessive burden, the fact that it cast doubt on the soundness of the renderer's output was problematic. Any artifact in the image immediately drew the question "Did you try disabling shading LOD?" Ultimately, the benefit did not justify the cost and this feature was removed.

Perhaps the most significant problem caused by our lack of an effective automatic LOD solution is noise due to subpixel geometric variance. A scene may render in a reasonable amount of time and within the memory budget, but subpixel geometric detail may still introduce excessive noise. And while memory and time can be increased to a point, such noise often does not converge in any reasonable amount of time.

6.7 Per-Face versus Single-Image Textures

We use the open source Ptex library for all of our texture storage, caching, and filtering of both per-face and "single-image" textures (where a single-image texture is just a traditional texture map stored in a Ptex file containing only one face). Ptex was initially developed primarily to address I/O bottlenecks due to the large number of texture files previously required for highly detailed texturing (Burley and Lacewell 2008). The per-face textures offered by Ptex also enable significant workflow efficiency, avoiding the need to manually assign UVs or decompose surfaces into multiple texture files to manage texture resolution. For these reasons, per-face textures form the bulk of our texture data, but single-image textures are still used, usually for quickly covering an object with a tiled projection when a unique texture has not been authored or is not needed. Using the Ptex format for both allows unified cache management and filtering, and though Hyperion's sorted shading is intended to provide maximal coherence for per-face texture access, single-image textures also benefit.

Per-face and single-image textures differ in how they handle large filter widths, for example, resulting from the wide footprint of indirect diffuse rays. A single-image texture can be pre-filtered down to a single texel making texture access cost negligible for sufficiently wide filter footprints. In contrast, per-face Ptex textures are pre-filtered only down to a single texel for each face. This limits overblurring but increases the data access required for indirect shading. Fortunately, the increased data access is mitigated by a special constant-per-face texture which is stored as a contiguous block at the front of the Ptex file and covers the entire mapped mesh.

It's worth pointing out that many, perhaps even most of our textures are masks used for blending between material layers. These

masks often have large regions of constant color, typically black or white. For these mask textures, per-face Ptex textures are particularly advantageous as the filter is aware of constant faces, reading these directly from the aforementioned constant-per-face texture without any filter kernel overhead required.

6.8 Difficulty of Experimenting with Alternative Sampling Techniques

Hyperion was designed to enable efficient unidirectional path tracing from the camera. As a result, there are now many assumptions in the code and architectural constraints that sometimes increase complexity, reduce efficiency, and make experimenting with alternative sampling techniques difficult. In particular, our integration code is currently highly decentralized and built into the shading system.

As an example, instead of being implemented in a single place, our light sampling code is implemented partially in individual shaders, and partially in some shared infrastructure. Furthermore, until recently we did not distinguish from rays created using BSDF sampling and rays created using light sampling. As a result, we had to have a relatively complex Russian-roulette system that worked for all rays and we could not implement features, such as eye caustics, that required separate rays for separate lights, each refracted differently.

As another example, our photon-mapping system is spread across several different areas, ranging from custom, dedicated photon-tracing passes between iterations to photon-gathering code implemented inside of our shaders. Ideally, we would like to have a dedicated integrator to handle photon-mapping calculations.

7 CONCLUSION

We have presented the architecture and development of Disney's Hyperion Renderer, along with a number of challenges encountered during the course of three feature films. We have also analyzed and discussed various strengths and shortcomings in Hyperion's architecture. Hyperion has led to significant improvements in our filmmaking process and continues to evolve to meet the studio's needs for rendering new phenomena, increasing artist efficiency, and ultimately creating films of the highest quality possible.

ACKNOWLEDGMENTS

Many people besides the authors of this article contributed to the development of Disney's Hyperion Renderer. The original architecture for Hyperion was proposed by Brent Burley, and developed by Christian Eisenacher, Gregory Nichols, and Andrew Selle. Significant Hyperion development was also performed by Benedikt Bitterli, Simon Kallweit, Gabor Liktor, Ulrich Muller, Jan Novák, Ben Spencer, and Serge Sretschinsky.

We are grateful for the many in-depth discussions with Disney Research, Pixar Research, and the RenderMan development team which influenced the development of Hyperion, especially Per Christensen, Julian Fong, Christophe Hery, Wojciech Jarosz, Marios Papas, Thomas Müller, Fabrice Rouselle, Rasmus Tamstorf, Ryusuke Villemin, and Magnus Wrenninge.

We are thankful for leadership and management support from Sean Jenkins, Darren Robinson, Rajesh Sharma, and Chuck Tappan; project management from Andrew Fisher and Tami Valdes; and quality engineering from Doug Lesan and Lisa Young.

More than anything, we would like to thank the many artists, technical directors, and production supervisors who motivated Hyperion, participated in its development, and contributed to its success, especially *Big Hero 6* leadership for taking a leap of faith. We would also like to thank the pipeline software development team, technology department, and studio leadership for supporting our efforts.

REFERENCES

John Amanatides. 1984. Ray tracing with cones. *Computer Graphics (Proc. of SIGGRAPH)* 18, 3 (Jan. 1984), 129–135. DOI: http://dx.doi.org/10.1145/964965.808589

Rasmus Barringer and Tomas Akenine-Möller. 2014. Dynamic ray stream traversal. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 33, 4 (July 2014), 151:1–151:10.

Carsten Benthin, Sven Woop, Ingo Wald, and Attila T. Áfra. 2017. Improved two-level BVHs using partial re-braiding. In *Proc. of HPG*. 7:1–7:8.

Brent Burley. 2012. Physically based shading at Disney. *SIGGRAPH 2012 Course Notes: Practical Physically-Based Shading in Film and Game Production.*

Brent Burley. 2015. Extending Disney's physically based BRDF with integrated subsurface scattering. *SIGGRAPH 2015 Course Notes: Physically Based Shading in Theory and Practice.*

Brent Burley and Dylan Lacewell. 2008. Ptex: Per-face texture mapping for production rendering. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)* 27, 4 (June 2008), 1155–1164.

Matt Jen-Yuan Chiang, Benedikt Bitterli, Chuck Tappan, and Brent Burley. 2016a. A practical and controllable hair and fur model for production path tracing. *Computer Graphics Forum (Proc. of Eurographics)* 35, 2 (May 2016), 275–283.

Matt Jen-Yuan Chiang, Peter Kutz, and Brent Burley. 2016b. Practical and controllable subsurface scattering for production path tracing. In *ACM SIGGRAPH 2016 Talks*. 49:1–49:2.

Per H. Christensen. 2010. Point-based global illumination for movie production. *SIGGRAPH 2010 Course Notes: Global Illumination Across Industries.*

Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes image rendering architecture. *Computer Graphics (Proc. of SIGGRAPH)* 21, 4 (July 1987), 95–102.

Christopher DeCoro, Tim Weyrich, and Szymon Rusinkiewicz. 2010. Density-based outlier rejection in Monte Carlo rendering. *Computer Graphics Forum (Proc. Pacific Graphics)* 29, 7 (Sept. 2010).

Eugene d'Eon, Guillaume Francois, Martin Hill, Joe Letteri, and Jean-Marie Aubry. 2011. An energy-conserving hair reflectance model. In *Proceedings of the 22nd Eurographics Conference on Rendering (EGSR'11)*. Eurographics Association, 1181–1187. DOI: http://dx.doi.org/10.1111/j.1467-8659.2011.01976.x

Kirill Dmitriev and Hans-Peter Seidel. 2004. Progressive path tracing with lightweight local error estimation. In *Proceedings of Vision, Modeling, and Visualization.*

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted deferred shading for production path tracing. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)* 32, 4 (June 2013), 125–132. DOI: http://dx.doi.org/10.1111/cgf.12158

Manfred Ernst, Günther Greiner, and Marc Stamminger. 2006. Filter importance sampling. In *Proceedings of the Symposium on Interactive Ray Tracing*. IEEE Computer Society, 125–132.

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, Article 2, 79 pages. DOI: http://dx.doi.org/10.1145/3084873.3084907

Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert. 2015. Efficient ray tracing kernels for modern CPU architectures. *Journal of Computer Graphics Techniques* 4, 4 (Dec. 2015), 91–111.

M. Galtier, S. Blanco, C. Caliot, C. Coustet, J. Dauchet, M. El Hafi, V. Eymet, R. Fournier, J. Gautrais, A. Khuong, B. Piaud, and G. Terrée. 2013. Integral formulation of null-collision Monte Carlo algorithms. *Journal of Quantitative Spectroscopy and Radiative Transfer* 125 (April 2013), 57–68. DOI: http://dx.doi.org/10.1016/j.jqsrt.2013.04.001

Jonathan Garcia, Sara Drakeley, Sean Palmer, Erin Ramos, David Hutchins, Ralf Habel, and Alexey Stomakhin. 2016. Rigging the oceans of Disney's "Moana." In *SIGGRAPH ASIA 2016 Technical Briefs (SA'16)*. ACM, New York, Article 30, 4 pages. DOI: http://dx.doi.org/10.1145/3005358.3005379

Iliyan Georgiev, Jaroslav Křivánek, Stefan Popov, and Philipp Slusallek. 2012. Importance caching for complex illumination. *Computer Graphics Forum (Proc. of Eurographics)* 31, 3 (May 2012), 701–710.

Paul S. Heckbert. 1990. Adaptive radiosity textures for bidirectional ray tracing. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 24, 4 (Aug. 1990), 145–154.

Christophe Hery, Ryusuke Villemin, and Florian Hecht. 2016. Towards bidirectional path tracing at pixar. *SIGGRAPH 2016 Course Notes: Physically Based Shading in Theory and Practice*.

Illumination Engineering Society of North America. 1991. *IES Standard File Format for Electronic Transfer of Photometric Data and Related Information*.

Henrik Wann Jensen. 1996. Global illumination using photon maps. In *Proceedings of Eurographics Workshop on Rendering Techniques*. 21–30.

Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. 2001. A practical model for subsurface light transport. In *Prococeedings of SIGGRAPH'01 (Annual Conference Series)*. ACM, New York, 511–518.

Anton S. Kaplanyan and Carsten Dachsbacher. 2013. Path space regularization for holistic and robust light transport. *Computer Graphics Forum (Proc. of Eurographics)* 32, 2 (2013), 63–72.

C. Kulla and M. Fajardo. 2012. Importance sampling techniques for path tracing in participating media. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)* 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 SIGGRAPH 2017)* 36, 4 (2017). DOI: http://dx.doi.org/10.1145/3072959.3073665

Samuli Laine and Tero Karras. 2010. Efficient sparse voxel octrees. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*. ACM, New York, 55–63.

Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized production path tracing. In *Proceedings of HPG*. 10:1–10:11.

Tzu-Mao Li, Yu-Ting Wu, and Yung-Yu Chuang. 2012. SURE-based optimization for adaptive sampling and reconstruction. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 31, 6 (Nov. 2012), Article 194, 9 pages. DOI: http://dx.doi.org/10.1145/2366145.2366213

Jeffrey A. Mahovsky. 2005. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. Ph.D. dissertation. The University of Calgary, Calgary, Alberta, Canada.

Stephen R. Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, and Pat Hanrahan. 2003. Light scattering from human hair fibers. *ACM Tranactions on Graphics* 22, 3 (July 2003), 780–791. DOI: http://dx.doi.org/10.1145/882262.882345

Thomas Müller, Marios Papas, Markus Gross, Wojciech Jarosz, and Jan Novák. 2016. Efficient rendering of heterogeneous polydisperse granular media. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 35, 6 (Nov. 2016), 168:1–168:14.

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

Hubert Nguyen. 2007. *Gpu Gems 3, Ch. 20* (1st ed.). Addison-Wesley Professional.

Jan Novák, Andrew Selle, and Wojciech Jarosz. 2014. Residual ratio tracking for estimating attenuation in participating media. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 33, 6 (Nov. 2014), 179:1–179:11. DOI: http://dx.doi.org/10.1145/2661229.2661292

Sean Palmer, Jonathan Garcia, Patrick Kelly, and Ralf Habel. 2017. The ocean and water pipeline of Disney's "Moana." In *ACM SIGGRAPH 2017 Talks*.

Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. 2012. Adaptive rendering with non-local means filtering. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 31, 6 (Nov. 2012), 195:1–195:11. DOI: http://dx.doi.org/10.1145/2366145.2366214

Fabrice Rousselle, Marco Manzi, and Matthias Zwicker. 2013. Robust denoising using feature and color information. *Computer Graphics Forum (Proc. of Pacific Graphics)* 32, 7 (2013), 121–130. DOI: http://dx.doi.org/10.1111/cgf.12219

Andy Selle, Janet Berlin, and Brent Burley. 2011. SeExpr. Retrieved May 22, 2018 from https://www.disneyanimation.com/technology/seexpr.html.

Peter Shirley, Changyaw Wang, and Kurt Zimmerman. 1996. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics* 15, 1 (Jan. 1996), 1–36. DOI: http://dx.doi.org/10.1145/226150.226151

H. C. van de Hulst. 1974. The spherical albedo of a planet covered with a homogeneous cloud layer. *Astronomy and Astrophysics* 35 (Oct. 1974), 209–214.

Eric Veach. 1997. *Robust Monte Carlo Methods for Light Transport Simulation*. Ph.D. dissertation. Stanford University, Stanford, CA.

Michael D. Vose. 1991. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on Software Engineering* 17, 9 (Sept. 1991), 972–975. DOI: http://dx.doi.org/10.1109/32.92917

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 HPG*. 41–49.

Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. 2006. R-LODs: Fast LOD-based ray tracing of massive models. *Visual Computer* 22, 9 (Sept. 2006), 772–784. DOI: http://dx.doi.org/10.1007/s00371-006-0062-y

Yonghao Yue, Kei Iwasaki, Bing-Yu Chen, Yoshinori Dobashi, and Tomoyuki Nishita. 2011. Toward optimal space partitioning for unbiased, adaptive free path sampling of inhomogeneous participating media. *Computer Graphics Forum* 30, 7 (2011), 1911–1919. DOI: http://dx.doi.org/10.1111/j.1467-8659.2011.02049.x

Arno Zinke, Cem Yuksel, Andreas Weber, and John Keyser. 2008. Dual scattering approximation for fast multiple scattering in hair. *ACM Transactions on Graphics* 27, 3 (Aug. 2008), Article 32, 10 pages. DOI: http://dx.doi.org/10.1145/1360612.1360631

Received November 2017; accepted January 2018