RenderMan: An Advanced Path Tracing Architecture for Movie Rendering

PER CHRISTENSEN, JULIAN FONG, JONATHAN SHADE, WAYNE WOOTEN, BRENDEN SCHUBERT, ANDREW KENSLER, STEPHEN FRIEDMAN, CHARLIE KILPATRICK, CLIFF RAMSHAW, MARC BANNISTER, BRENTON RAYNER, JONATHAN BROUILLAT, and MAX LIANI, Pixar Animation Studios

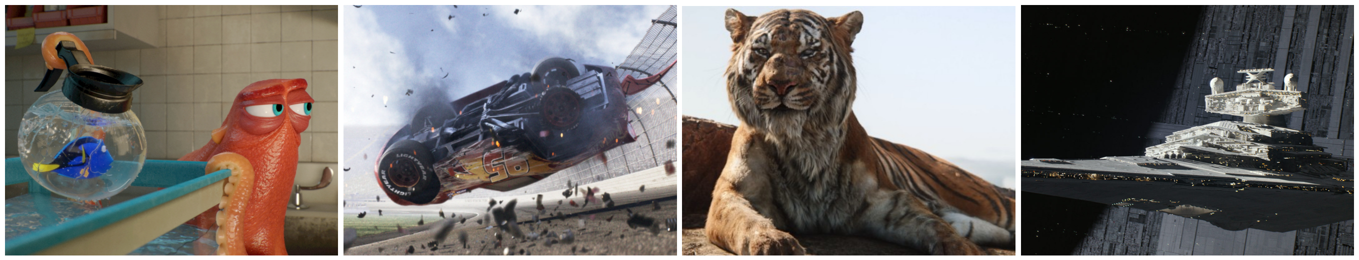


Fig. 1. Path-traced images rendered with RenderMan: Dory and Hank from *Finding Dory* (© 2016 Disney•Pixar). McQueen's crash in *Cars 3* (© 2017 Disney•Pixar). Shere Khan from Disney's *The Jungle Book* (© 2016 Disney). A destroyer and the Death Star from Lucasfilm's *Rogue One: A Star Wars Story* (© & ™ 2016 Lucasfilm Ltd. All rights reserved. Used under authorization.)

Pixar's RenderMan renderer is used to render all of Pixar's films, and by many film studios to render visual effects for live-action movies. RenderMan started as a scanline renderer based on the Reyes algorithm, and was extended over the years with ray tracing and several global illumination algorithms.

This paper describes the modern version of RenderMan, a new architecture for an extensible and programmable path tracer with many features that are essential to handle the fiercely complex scenes in movie production. Users can write their own materials using a bxdf interface, and their own light transport algorithms using an integrator interface – or they can use the materials and light transport algorithms provided with RenderMan. Complex geometry and textures are handled with efficient multi-resolution representations, with resolution chosen using path differentials. We trace rays and shade ray hit points in medium-sized groups, which provides the benefits of SIMD execution without excessive memory overhead or data streaming. The path-tracing architecture handles surface, subsurface, and volume scattering. We show examples of the use of path tracing, bidirectional path tracing, VCM, and UPBP light transport algorithms. We also describe our progressive rendering for interactive use and our adaptation of denoising techniques.

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

Additional Key Words and Phrases: Pixar, RenderMan, computer-generated images, visual effects, production rendering, complex scenes, path tracing, ray tracing, global illumination.

ACM Reference format:
Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. 2018. RenderMan: An Advanced Path Tracing Architecture for Movie Rendering. *ACM Trans. Graph.* 37, 3, Article 30 (July 2018), 21 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Author's addresses: 506 Second Ave, Seattle, WA and 1200 Park Ave, Emeryville, CA.

1 INTRODUCTION

Pixar's movies and short films are all rendered with RenderMan. The first computer-generated (CG) animated feature film, *Toy Story*, was rendered with an early version of RenderMan in 1995. The most recent Pixar movies – *Finding Dory*, *Cars 3*, and *Coco* – were rendered using RenderMan's modern path tracing architecture. The two left images in Figure 1 show high-quality rendering of two challenging CG movie scenes with many bounces of specular reflections and refractions, subsurface scattering, complex geometry, smoke, motion blur, etc.

At the same time, RenderMan is also a commercial product sold to other movie studios so they can render visual effects (VFX) for their movies. To date nearly 400 movies have used RenderMan for VFX; recent movies include *The Jungle Book*, *Rogue One*, *Fantastic Beasts*, and *Blade Runner 2049*. The two right images in Figure 1 show examples of realistic fur and a spaceship with high geometric complexity.

RenderMan has received several Academy Awards, and was the first software package to receive an Oscar® statuette (to Ed Catmull, Loren Carpenter, and Rob Cook in 2001 "for significant advancements to the field of motion picture rendering as exemplified in Pixar's RenderMan").

RenderMan was originally based on the Reyes scanline rendering algorithm [Cook et al. 1987], and was later augmented with ray tracing, subsurface scattering, point-based global illumination, distribution ray tracing with radiosity caching, and many other improvements. But we have recently rewritten RenderMan as a path tracer. This switch paved the way for better progressive and interactive rendering, higher efficiency on many-core machines, and was facilitated by the concurrent development of efficient denoising algorithms. Our path tracer was designed from the beginning to allow bidirectional path-tracing algorithms and efficient path tracing in volumes.

Path tracing was introduced to computer graphics [Kajiya 1986] at around the same time as Reyes was developed. It is amusing to note that, at the time, path tracing was considered to be an elegant but hopelessly impractical rendering algorithm, producing too noisy images (especially with motion blur and depth of field) and requiring the geometry of the entire scene to fit in memory (which is particularly problematic for displacement-mapped surfaces). Hence it was rarely used, and if so mostly for reference images of relatively simple static scenes. Fortunately these obstacles have been overcome through improved algorithms, better data management, and more powerful computers.

This paper first gives a brief overview of the history of RenderMan, and then describes the modern path-tracing architecture.

2 HISTORICAL BACKGROUND: YE OLDE RENDERMAN

RenderMan has utilized many generations of rendering algorithms. In this section we describe the original Reyes algorithm and the hybrid algorithms that resulted from extending Reyes with ray tracing, point-based approaches, and distribution ray tracing.

2.1 Reyes

Although ray tracing, distribution ray tracing, and path tracing were well known rendering techniques at the time, Cook et al. [1987] developed the Reyes algorithm to overcome some of their shortcomings, with particular emphasis on data locality, displaced geometry, and cheap noise-free edge antialiasing, motion blur, and depth-of-field effects. Many of Reyes' architectural decisions were made with an eye toward a scalable pipelined hardware implementation: it is a streaming feed-forward design motivated by compact memory footprint, coherence of shading calculations and texture access, and shading reuse.

The Reyes algorithm works as follows: The scene description is read in and high-level object descriptions are stored along with their bounding boxes. The image is rendered one image tile ("bucket") at a time. The surfaces of the objects visible in an image tile are split into smaller patches and the patches are tessellated ("diced") into micropolygon grids, with each micropolygon typically roughly the size of an image pixel. A displacement shader can be optionally run to move the grid vertices for added geometric detail. Then a surface shader is run on each grid vertex to calculate the color and opacity at that point – this can involve looking up texture map colors, computing illumination, looking up occlusion in shadow maps, and more. Finally, the color of each pixel is computed by stochastic point sampling (with a z-buffer) of the micropolygon grids. This is very efficient since it is a 2.5-dimensional problem with high data coherency.

Figure 2 shows images from the *Luxo, Jr.* short and the feature film *Toy Story*. Both were rendered with Reyes RenderMan.

With this algorithm, motion blur and depth of field are very cheap to compute: noise can be resolved by increasing the number of pixel samples without incurring any additional shading cost. The motion blurred images are not strictly correct since the shading only occurs at specific shutter times and the resulting colors are then interpolated for other sample times. However, these errors are usually small, and the benefits of cheap motion blur and depth of field outweigh the (usually) subtle artifacts.

Fig. 2. Luxo lamps from *Luxo, Jr.* (© 1986 Pixar) and Buzz and Woody from *Toy Story* (© 1995 Disney•Pixar).

Reyes handles geometric complexity by rendering one image tile (per CPU core) at a time, and only tessellating the surface patches visible in that tile. This way, surface patches are only tessellated when needed, and the micropolygon grids are deleted from memory as soon as they are no longer needed. Displacement-mapped surfaces do not incur any additional memory overhead. This particular aspect was instrumental in rendering the realistic dinosaurs in *Jurassic Park*: the skin scales and wrinkles (see Figure 3 (left)) could be represented with actual displaced surface geometry instead of being approximated with bump maps.

Fig. 3. T. Rex from *Jurassic Park* and the shiny metal Terminator from *Terminator 2* (© 1993 and 1991 ILM).

Likewise, huge amounts of texture data are dealt with by generating tiled MIP maps [Williams 1983] of all textures before rendering starts, only reading texture tiles at the appropriate MIP map level, and storing the texture tiles in a cache [Peachey 1990]. This makes it possible to efficiently render scenes with 100 times more texture data than main memory.

Another noteworthy aspect of Reyes RenderMan is the RenderMan Shading Language (RSL) by Hanrahan and Lawson [1990]: it is a simple but expressive C-like language that allows the user to program displacement, illumination, and surface shading. It provides a nice abstraction: a shader is written as if it computes displacement, illumination, or color and opacity at a single point, but it is executed in SIMD fashion over the vertices of an entire micropolygon grid. This automatically gives coherent access to textures and shadow maps.

Since Reyes has no facility to trace shadow rays, shadows must be computed using shadow maps [Reeves et al. 1987; Williams 1978]. Using shadow maps can be great for data locality, but requires the shadow map for each shadow-casting light source to be generated in a pre-pass. If there are hundreds of light sources with shadow maps this is cumbersome and the data locality is lost. Also, special tricks are needed to get realistic shadows from area lights (sharp shadow edges at contact points but increasingly smooth penumbra regions further away). Likewise, with Reyes, reflections have to be rendered using reflection maps – either a real image of some environment or a 360 degree image rendered in a separate pass. Figure 3 (right) shows reflections in a liquid metal Terminator T-1000 robot rendered with a reflection map.

Many more details on Reyes and its implementation in RenderMan can be found in the excellent books by Upstill [1990] and Apodaca and Gritz [2000].

2.2 Ray tracing

The motivation to add ray tracing [Whitted 1980] to RenderMan was to enable more realistic reflections (including self-interreflections), and to render detailed shadows in expansive scenes without shadow map resolution limitations or having to keep track of shadow maps for hundreds of lights in a scene. It also enabled us to render ambient occlusion [Landis 2002; Zhukov et al. 1998].

Ray tracing was used early-on for specular reflections and refractions in a glass bottle in *A Bugs Life*. This was done by having RenderMan call the Whitted-style ray tracer BMRT as a "ray server" [Apodaca and Gritz 2000; Gritz and Hahn 1996]. Only a subset of the scene could be ray traced since it had to fit in memory. Another issue was duplication of geometry since RenderMan and BMRT each kept their own representation of the geometry. Clearly it would be better to do ray tracing inside RenderMan itself, which led us to our first major re-architecting of RenderMan, turning it into a scanline-raytracer hybrid.

However, data access patterns for recursive ray tracing are more challenging than for Reyes: while camera rays, reflection rays from flat surfaces, and shadow rays to a small light are coherent, in general reflection or shadow rays can be traced in any direction at any time during rendering, so geometry and textures are accessed in rather random and unpredictable patterns. With ray tracing, we cannot discard a micropolygon grid when its image tile is done: it may be needed again for reflection or shadow rays from other image tiles.

We did some initial experiments with ray tracing directly against Bézier and NURBS patches (using iterative root finders), but found that ray tracing against tessellated patches was faster. Displacement-mapped surfaces also require tessellation. But too much memory was required if all tessellated surface patches were kept in memory. On the other hand, it would be too slow to tessellate and displace the micropolygon grids on-the-fly for every ray intersection test. In order to solve this problem we developed a multiresolution tessellation cache [Christensen et al. 2006, 2003], where tessellated (and displaced) surface patches were stored in the cache, and ray differentials [Igehy 1999] were used to select the appropriate resolution. For fast intersection tests we utilized SSE instructions to test one ray against four triangles (two quadrilaterals) in parallel. This scheme worked well even with incoherent rays, whereas testing four rays against one triangle [Wald et al. 2001] would be efficient only for coherent rays such as camera rays and mirror reflection rays from flat surfaces.

A similar multiresolution approach was also used for textures – although textures were simpler to implement since the already existing texture cache also worked really well for ray tracing with ray differentials.

This ray tracing functionality was used on the *Cars* movie for specular reflections, shadows, and ambient occlusion. Figure 4 (left) shows two bounces of ray-traced specular reflections in Doc Hudson's chrome bumper. Figure 4 (right) shows ray-traced refractions and reflections in wine glasses in a scene from *Ratatouille*. At that time, switching entirely to ray tracing – i.e. replacing Reyes rasterization with camera rays – was not an option since shading cost and noise in motion-blurred and depth-of-field images compared unfavorably with Reyes.

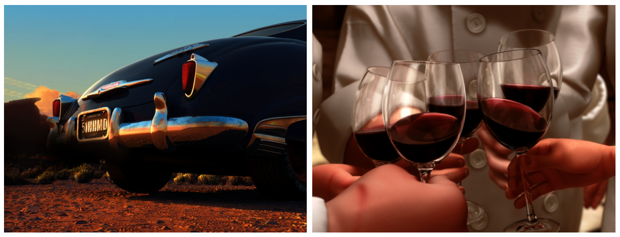


Fig. 4. Chrome bumper with specular reflection from *Cars* and wine glasses with refractions from *Ratatouille* (© 2006 and 2007 Disney•Pixar).

2.3 Distribution ray tracing

At the same time we added Whitted-style ray tracing, we also added distribution ray tracing [Cook et al. 1984] to compute global illumination (indirect diffuse illumination). But this did not get used in practice because it was too expensive to evaluate shaders at all the distribution ray hit points.

At Dreamworks, Tabellion and Lamorlette [2004] introduced a more efficient single-bounce global illumination method: In a pre-pass, they rendered direct illumination on all object surfaces and stored it in 2D texture maps. During rendering, they did a single level of distribution ray tracing, and looked up the color at the ray hit points in the 2D texture maps. They used irradiance gradients [Křivánek et al. 2008; Ward and Heckbert 1992] to interpolate the distribution ray tracing results, thereby reducing the number of rays significantly.

At roughly the same time, we introduced a similar technique, but stored the textures in 3D point clouds and brick maps – a tiled 3D texture format that does not require the surfaces to be parameterized [Christensen and Batali 2004]. This was convenient for subdivision surfaces, implicit surfaces, and dense polygon meshes that do not have an inherent parameterization and hence are cumbersome to assign 2D textures to. Irradiance gradients were extra simple in our Reyes-based renderer: since all shading points on a micropolygon grid were known to be on the same object surface, interpolating irradiance gradients was trivial, and it was easy to know when interpolation was safe.

This technique was only used for global illumination in a few movies (none at Pixar). It was still slow since many rays needed to be traced – typically 64 rays per shading point – to avoid noisy results. It was, however, used for glossy reflections in metal pots and pans in *Ratatouille*.

2.4 Point-based approximations

Users had more success with our point-based methods. We first developed a point-based method for subsurface scattering, and later – inspired by Bunnell [2005] – point-based methods for ambient occlusion and approximate global illumination in collaboration with ILM and Sony [Christensen 2008].

Again, direct illumination was rendered in a pre-pass and stored ("baked") in point cloud file(s). Subsurface scattering, ambient occlusion, or global illumination could be computed in a stand-alone program (ptfilter, introduced in 2004) or by calling new point-based variants of functions in the RenderMan Shading Language. The computation organized the point clouds into a cluster hierarchy and computed the desired quantity. For subsurface scattering, the contributions from each point or cluster were simply multiplied by the diffusion profile and added together. For ambient occlusion and global illumination, a (very coarse) local hemisphere rasterization was necessary to reject contributions from points behind other points.

The advantages of the point-based methods over distribution ray tracing were that they were much faster (no ray tracing) and the results had no noise. The disadvantages were that they required a pre-pass to generate the direct illumination point clouds, managing the point cloud files was a hassle, and the results could have visible aliasing artifacts if the point clouds were too sparse. More than 60 movies were made using RenderMan's point-based techniques; Figure 5 shows two examples.

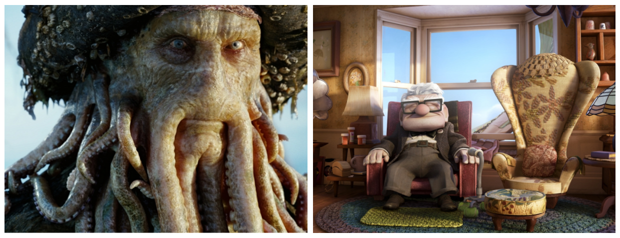


Fig. 5. Point-based subsurface scattering and global illumination on Davy Jones from *Pirates of the Caribbean: Dead Man's Chest* (© 2006 Disney/Jerry Bruckheimer) and in Carl's living room in *Up* (© 2009 Disney●Pixar).

2.5 Distribution ray tracing, take two

After using the point-based approaches for a few years, users grew tired of having pre-passes before rendering and having to store and manage point cloud files. This lead us to revisit distribution ray tracing, but this time with on-the-fly caching of view-independent shading results.

For this to work, the shaders had to be split into view-independent (diffuse) and view-dependent (specular) parts. We cached view-independent results in a so-called radiosity cache [Christensen et al. 2012]. (We chose this term because the cached results represent radiant exitance, a.k.a. radiosity.) This method reused expensive shading results – expensive because of textures and direct and indirect illumination calculations. The results were computed and cached for an entire micropolygon grid at a time, thus preserving data coherence. The view-dependent part of each shader was executed at every ray hit point.

The Pixar movie *Monsters University* was rendered with this technique. This was part of a push within Pixar for physically based rendering and a simplified lighting set-up with more global illumination and fewer fill lights. Figure 6 (left) shows a scene from *Monsters University* with global illumination computed this way.

This technique could also be used to generate somewhat fast previews if the distribution branching factors are dialed down – however, it is not as fast to first pixel as path tracing is, and it is not progressive: a noisy preview image does not progress to a final quality image. Nonetheless, many of the physically-based techniques and optimizations developed in this setting served as inspiration for our development of the modern path-traced RenderMan, and its material and integrator plug-ins.

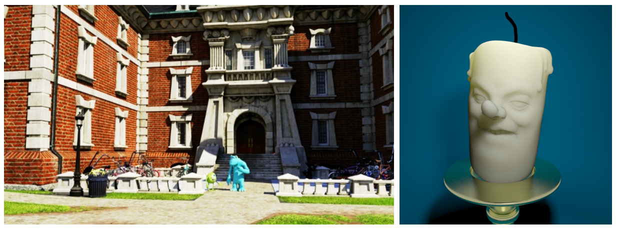


Fig. 6. Distribution ray tracing with radiosity caching: *Monsters University* scene with global illumination (© 2012 Disney●Pixar) and subsurface scattering on a candle face (modeled by Chris Harvey).

To avoid point clouds entirely, we also developed a distribution ray tracing version of subsurface scattering (in 2011). A spherical distribution of rays was shot from a point below the object surface, the illumination on the outside of the surface was computed or looked up in the radiosity cache at those ray hit points, and each outside illumination value was multiplied by the subsurface scattering diffusion profile. (This was later improved with importance sampling using the diffusion profile, multiple importance sampling, etc.) Ray tracing was remarkably efficient in this application since rays could have a fairly short maximum distance and only needed to be intersection tested against the object they were shot from. The candle in Figure 6 (right) was rendered with this method. A similar method was described by King et al. [2013].

2.6 Volumes

The original Reyes algorithm did not consider volumes, thus requiring RSL hackery – see Hanson's chapter in Apodaca and Gritz [2000]. But over the years, we extended RenderMan to incorporate volumes by doing the 3D equivalent of the Reyes surface split, tessellate ("dice"), and shade: subdivide the volume, voxelize it into microcubes, compute color and transparency at the voxel grid vertices, and render the microcubes. A similar approach was used in the Houdini renderer [Clinton and Elendt 2009].

2.7 Re-evaluating Reyes

Even though the Reyes algorithm has been highly successful over the years, and longer-lived than the original inventors imagined, in retrospect it also has some weaknesses:

1) The efficiency of the algorithm breaks down when the input geometry is extremely dense: if each input surface patch is a lot smaller than a pixel then at most one pixel sample will hit its bounding box (during scanline rendering). Once that hit has been found, the patch is tessellated into a single quadrilateral (or triangle, depending on the geometry type), the four (or three) vertices are shaded, and the results interpolated. But it is wasteful to shade multiple vertices just to obtain one pixel sample. (Note that this waste would have been even worse with the original Reyes strategy of shading all objects within the viewing frustum without determining whether some surfaces are entirely hidden behind other surfaces.)

2) Reyes cannot exploit instancing: different instances (with shared geometry and textures) can have different illumination, and hence shaded colors cannot be shared between instances. In this respect, it is as if the instances were separate objects.

3) Volumes, and particularly multiple bounces of light in volumes, are challenging to render within reasonable memory and time. This is because decoupling lighting frequency from microvoxel size is hard, and volumes typically have low frequency (lighting) content.

4) We found it hard in practice to scale our Reyes-based architecture to run at optimal efficiency beyond 16 threads. Even with an advanced architecture where locks are associated with geometry, it is hard to balance the needs of shared geometry being used by multiple threads vs. out-of-order writes to the z-buffer.

3 MODERN RENDERMAN ARCHITECTURE

Over the last few years, we have rewritten RenderMan as a path tracer. As mentioned earlier, the modern version of RenderMan has been used to render the Pixar movies *Finding Dory*, *Cars 3*, and *Coco*, as well as recent movies by other studios such as *Ant-Man*, *Terminator Genisys*, *The Jungle Book*, *Rogue One*, *Fantastic Beasts*, and *Blade Runner 2049*.

The reasons for the switch to path tracing are that it is a unified and flexible algorithm, it is well suited for progressive rendering, it is faster to first pixel, it is a single-pass algorithm (unlike the point-based approaches that require pre-passes to generate point clouds), and geometric complexity can often be handled through object instancing. Also, the Achilles' heels of path tracing – noisy images and slow convergence – have been addressed by new and effective denoisers and improved sampling. Path tracing requires more memory than pure Reyes rendering, but with all the various caches we had added to RenderMan over the years, the difference is less dramatic.

Like the original Reyes RenderMan, we chose to make our new renderer extensible and programmable. The plug-in architecture allows users to write their own materials ("bxdfs"), rendering algorithms ("integrators"), patterns, displacements, lights, light filters, camera projections, sample filters, and display (pixel) filters. The plug-in architecture makes it easy for users to express new material types, experiment with new light transport algorithms, try different strategies for volume integration, etc. On the other hand, we have found it necessary to protect the renderer from garbage results (such as NaNs and inf) being returned from plug-ins.

RSL surface shaders had a mixture of texture generation/lookups, ray generation, and color and opacity calculation. The new interface has a more clear distinction between surface properties and the rendering algorithm. This division was inspired by the PBRT book by Pharr et al. [2017]. The new interface was designed from the onset to support bi-directional path tracing and path-traced volumes.

We trace groups of rays together, and shade groups of ray hit points on surfaces with the same material (i.e. same bxdf and same input connections – a "bxdf instance") together. After a group of rays have been traced, the ray hits are sorted into shade groups with the same material. For each shade group, the integrator typically calls a bxdf constructor which causes the bxdf's inputs (usually a pattern network) to be evaluated. Given these input values, the bxdf can then be called to generate and evaluate sample directions, calculate opacity, etc.

Our trace and shade groups typically consist of up to a few hundred rays and ray hit points, respectively. We do not collect entire wavefronts of rays and ray hits (millions of rays and ray hits) since that would use huge amounts of memory or require streaming the data to disk. Instead, we have found that keeping relatively small groups often give the benefits of coherent tracing and shading without a large memory overhead. (A recent paper by Áfra et al. [2016] came to the same conclusion.) It should be mentioned, though, that in a typical production scene, the size of the trace and shade groups degenerate to just a single ray or shade point beyond four ray bounces. If we used larger groups of camera rays, that point of degeneration could presumably be pushed further out.

We still use RIB (RenderMan Interface Bytestream) format to describe scenes. RIB has survived because it is reasonable as an archival format: for more than 25 years it has been able to describe all the geometry formats we have needed for rendering. It has been able to describe the linkages between lights and geometry as well (although not the lights and shaders themselves). Unfortunately RIB is also inherently a streaming representation, and is ill suited for the needs of multithreaded geometry creation, as we will describe in Section 9.

4 PLUG-INS

RenderMan allows plug-ins for materials, rendering algorithms, patterns, and more.

4.1 Material interface: bxdfs

Surface materials are specified by physically-based bxdfs and by texture patterns to control the parameters of the bxdfs. "Bxdf" in our context also encompasses phase functions for volumes.

The two main API functions specifying a bxdf are EvaluateSample() and GenerateSample(). Both functions operate on arrays of data to reduce call overhead and increase data locality. EvaluateSample() takes as input arrays of incident and exitant directions, and returns an array of rgb bxdf values (each value being the ratio of reflected radiance in the exitant direction over differential irradiance from the incident direction) and two arrays of probability density function (pdf) values – one for forward scattering and one for backward scattering. GenerateSample() takes as input an array of incident directions and generates an array of "random" exitant directions along with arrays of rgb bxdf and pdf values for those directions. There is also a GetOpacity() function that returns opacity values for shadows.

Since we keep groups of ray hits on the same material together in shade groups, the material's generate, evaluate, and opacity functions can be called for entire groups in a single function call. This enables C++ compiler optimizations such as vectorization, loop unrolling, etc. in these functions.

Sadly, gone are the days where a novice TD could quickly learn to write an RSL surface shader. These days, specifying a material is much more complicated: writing a bxdf requires knowledge of C++, optics, probability theory, sampling, pdfs, etc. Luckily, many good resources and publications are available today for shading TDs to become more proficient in the face of this new challenge.

Users can write their own materials (bxdfs) or simply use one of the materials provided with RenderMan. We provide simple specific materials such as Lambertian diffuse, perfect mirror, and skin with subsurface scattering. We also provide a material based on Disney's bxdf model [Burley 2015], and a general-purpose "über"-material developed by Pixar's studio tools illumination group [Hery et al. 2017b]. (The idea is to provide the material used in Pixar movie productions to external RenderMan customers.) There is also a bxdf specifically for hair and fur [Marschner et al. 2003; Pekelis et al. 2015].

4.2 Light transport interface: integrators

Integrators are light transport plug-ins. They get a group of camera rays as input, and can call bxdf and light source API functions to generate and evaluate samples. Integrators also call API functions called GetNearestHits() and GetTransmission() – provided by the renderer – to do the actual ray tracing. There are two GetNearestHits() functions: one that returns shade groups for the ray hit points, and one that traces ray probes that only return geometric information about the ray hits (useful for e.g. subsurface scattering). GetTransmission() returns the transmittance between two points, and is used to query visibility for direct illumination computations and to connect eye and light paths in bidirectional path tracing. Examples of integrators are described in Sections 10 and 11.

Volumes and subsurface scattering require special handling. If an object has an associated interior integrator (for volumes, subsurface scattering, or both), the GetNearestHits() function for that interior integrator is called. The interior integrator can be as simple as tracing rays and reducing their transmission weights by Beer's law, or can trace many bounces of scattering in the volume. In the end, no matter how complex the interior integrator is, it returns ray hit points (shade groups) and associated weights to the main integrator.

The renderer provides generate and evaluate functions to help sampling of light sources for direct illumination calculation (next event estimation). This part of the renderer also handles shadow calculation, i.e. it traces transmission rays to determine whether the selected light source position is (fully or partially) blocked by a solid or semitransparent object or volume.

4.3 Pattern networks

Pattern networks can compute varying material parameters for scattering or transmission. Patterns can access texture maps or compute results procedurally, and can be combined in networks of compute nodes. Evaluating pattern networks for entire shading groups at a time reduces overhead and gives improved data locality for texture map look-ups.

RenderMan supports C++ pattern plug-ins as well as two procedural languages to generate patterns: SeExpr from Disney [Berlin et al. 2011] and Open Shading Language (OSL) from Sony [Gritz et al. 2010]. Both are open source. As opposed to RSL, which was designed to implement entire shaders, these new shading languages are based on the concept of networks of shading patterns. We found that these shading languages are not quite as performant as our C++ pattern plug-ins, mainly due to execution vectorization of C++, but they are more modular and easily transferrable between renderers and between studios working on the same project. Some very desirable properties of OSL are that parameters are only evaluated on demand, as well as a variety of run-time optimizations that OSL performs on the shader code, including static branch elimination, constant folding, caching and reuse of shared expressions, and more. There is also work in progress to vectorize OSL for more efficient execution.

4.4 Other plug-ins

RenderMan also supports plug-ins for surface displacement (to add geometric detail to objects), light sources (rectangular, disk, sphere, distant, environment, sky, mesh), light filters (barn door, blocker, cookie, gobo, color ramp, portals, etc), camera projections (for e.g. IMAX projections, virtual reality, or extreme U-shaped screens at amusement park rides), pixel sample filtering (for e.g. firefly reduction), and display filtering (to modify pixel colors for e.g. tone mapping).

5 HANDLING COMPLEX SCENES

A production-strength renderer such as RenderMan is a complex data management system that must be able to handle massive amounts of geometry, textures, and light sources, and utilize parallel compute resources. The geometry and texture input can be many times larger than the main memory of the computer, and therefore requires careful management: read on demand, caching, etc. We sometimes say tongue-in-cheek that rendering a movie image is simply extreme data compression, and that the output image is almost just a by-product of all this complex data management.

Scene complexity grows every year; for Pixar's most recent movie, *Coco*, the average memory use was 35 GB per frame, with 10% of the frames using more than 64 GB, and a few using up to 120 GB. Rendering a typical frame required shading of hundreds of millions of ray hit points.

5.1 Complex geometry

Movies have huge amounts of geometry in each image, and the surfaces are often displacement-mapped to add even more geometric complexity. For *Coco*, there were often on the order of 10 to 100 million (instanced and un-instanced) objects in each scene. Figure 7 shows a scene with about 20 million objects.

We split polygon meshes consisting of hundreds (or millions) of polygons into smaller meshes in order to create a BVH with smaller leaf nodes. The individual faces of polygon meshes are not diced

Fig. 7. Complex geometry in *Coco*: 20 million objects (© 2017 Disney•Pixar).

(unless the polygon faces are large on screen and displacement-mapped). As is common, we store the polygon meshes internally as strips with shared vertex positions and compact indices.

Other than polygon meshes, the most popular surface representations are subdivision surfaces [Catmull and Clark 1978; Loop 1987] and NURBS patches [Martin et al. 2000; Piegl and Tiller 1997]. (The subdivision surfaces can have sharp and semi-sharp creases, and NURBS patches can have trim curves.) For those types of geometry, we distinguish between surfaces that are large on screen and those that are small on screen.

Large surfaces are split into patches and then diced (and displaced) on demand and stored in a multi-resolution cache. Based on path differentials we may choose fine, medium, or coarse tessellations in the tessellation cache for intersection tests. We automatically determine the finest tessellation – roughly one micropolygon per pixel by default (but this is adjustable, of course). This gives seamless tessellation changes when zooming in or out without over- or under-tessellation. If the subdivision level for subdivision surfaces was determined manually it would often be too coarse (giving visibly facetted edges) or too fine (consuming too much memory).

Surfaces that are small on screen (for example over-modeled objects) do not get split into smaller patches, and the tessellation of each surface is very coarse. For subdivision surfaces, the limit surface can be evaluated at the vertices of the subdivision base mesh and the surface converted to a compressed polygon mesh. Then the original base mesh can be deleted since no subdivision is required anymore. The same applies to NURBS patches: the control mesh can be converted to a compressed polygon mesh.

Surfaces outside the viewing frustum only require a tessellation accuracy needed for reflections in visible surfaces or for casting shadows onto visible surfaces. So the further away from the viewing frustum the surface is, the coarser the geometry tessellation can be.

In the future, it would be convenient for users if RenderMan could simplify over-modeled geometry (for the given camera position) on read. Such simplification must maintain the overall shape reasonably well. However, feature-preserving geometric simplification is a difficult problem – usually done in specialized stand-alone programs – so this would likely be a challenge to implement properly and efficiently.

5.2 Complex textures

Textures are either computed procedurally or read from texture maps.

Procedural textures have the advantage that they require very little memory. They must be written such that they are efficient to evaluate and must take sample frequencies and the Nyquist limit into account in order to produce a texture with the right amount of detail: too little detail and the rendered image will look too smooth; too much detail and there will be more sampling noise than necessary.

The texture tile cache of Peachey [1990] is still alive and well. Since Peachey's original version, two major improvements have been implemented: multithreaded look-ups, and use of ray differentials to make the same approach work for ray tracing. It is astonishing that the same caching scheme – with only a few updates – is still viable nearly 30 years later! Typical scenes from *Coco* had tens of thousands of texture maps, with several gigabytes of texture reads.

We also offer a similar approach for 3D textures: a tiled, MIP mapped format called brick map [Christensen and Batali 2004]. A brick map is an octree of bricks where each brick contains $8 \times 8 \times 8$ voxels of texture data; bricks are cached just like 2D texture tiles. (Brick maps can also be used as geometric primitives with built-in level of detail since the voxels have implicit or explicit geometry information in addition to the stored texture data.)

Another popular texture format supported by RenderMan is Ptex [Burley and Lacewell 2008]. It is a single texture file per surface mesh, with a MIP mapped texture map for each face of the mesh. One limitation of the Ptex format is that the coarsest representation is 1 texel per mesh face, collected in a table in the file header. This varying-size representation is not ideal for caching. For texture look-ups on heavily over-modeled subdivision surfaces, it would be beneficial to have even coarser texels – each covering several faces – since that would access less data and also reduce noise due to the prefiltered sample values. We believe it would be interesting to extend Ptex along the lines suggested by Cook [2007] in his Htex proposal which includes coarser averages.

5.3 Complex illumination

Figure 8 shows a scene from *Coco* with 8 million point light sources. For efficient illumination calculations in scenes with many light sources, we can only afford to sample a subset of the lights at each shading point. It can be challenging to efficiently select which light sources to sample and/or trace shadow rays to. (For distribution ray tracing, many rays were traced from each shading point, so complex light selection strategies could be better amortized. But for path tracing we usually only trace a few rays per shading point, so the light selection has to be very fast.)

Fig. 8. Complex illumination in *Coco*: 8 million lights (© 2017 Disney•Pixar).

There are a couple of classic approaches to light selection. Ward [1991] sorted lights according to their potential contribution at each shading point. He only traced shadow rays to lights above a threshold and used coarse averages for the rest. Shirley et al. [1996] developed a strategy where for each region of space the light sources were divided into bright and dim. At a given shading point, all the bright lights were sampled, but only a few of the dim lights.

One improvement is to cluster the lights for improved efficiency. A second improvement is multiple importance sampling [Veach and Guibas 1995]: take some samples in directions that are important for the bxdf, and some samples at important light source positions, and combine the results of these samples. An even better approach (unless the material is purely diffuse) is joint importance sampling: take the bxdf into account when selecting light sources. We create a cone of directions toward the light (or cluster of lights) and sample the bxdf for directions within that cone to get an estimate. These samples are multiplied by estimates of the light illumination. This is then combined, via multiple importance sampling, with samples that are based purely on the bxdf.

Light filters (which are plug-ins) can modify the illumination from a light; examples are cookies, barn doors, and ramps (non-physical fade-in and fade-out with distance). Most of these light filter types can be taken into account when selecting the light sources (although the position on the light hasn't been chosen yet so we cannot evaluate the filter precisely).

Another complication in practice is light linking – a non-physical feature where some lights only illuminate certain objects. (Light linking is only applied to direct illumination, not indirect.) In this case we have to avoid selecting lights that do not illuminate the object we're trying to calculate direct illumination on. One could perhaps use rejection sampling for this, but we have chosen to build a separate cluster hierarchy for every combination of light linking, thus incurring a memory increase but very little increase in run-time.

We have experimented with learning algorithms [Dahm and Keller 2017; Lafortune and Willems 1995; Müller et al. 2017] to learn which light sources are occluded at which parts of the scene during rendering. This is still work in progress.

5.4 Many-core execution

A key to efficient rendering of complex scenes is parallel execution. The oft-repeated claim that ray tracing (and path tracing) are "embarrasingly parallel" algorithms is a myth. Certainly, each pixel can be rendered independently in parallel, and if all data (geometry, textures, shaders, etc.) fit in memory and is already loaded and divided into appropriately-sized tasks with even load balancing, then close to ideal speed-ups can be gained with minimal effort. However, in practice, it takes a lot of thought and design considerations to obtain good speed-ups for complex scenes.

We aim to always develop and test our renderer on computers that are one generation ahead of the computers our customers have in their render farms. When 4 CPU cores were the norm, we would ensure nearly ideal speed-ups on 8 cores, etc. This has been a successful strategy thanks to early access to Intel prototype computers which has ensured that we could reorganize computations, optimize code, and eliminate bottlenecks before they could get in the way of full efficiency for our customers. Currently we optimize for configurations with 44–72 cores. We usually see an additional 1.2–1.5 times speed-up when running two hardware threads (hyperthreads) on each core. We have recently switched from a home-grown thread scheduler to using Intel's TBB (thread building blocks) library.

6 RAY TRAVERSAL AND INTERSECTION TESTS

This section describes our implementation choices for surface tessellation, ray acceleration data structures, ray intersection tests, floating-point accuracy, motion blur, and path differentials.

6.1 Tessellation

Surface tessellation depends on the screen size of the surface: the default target is that a micropolygon should be roughly the size of a pixel. We support two types of tessellation: *full* tessellation of the entire surface as soon as a ray hits its initial bounding box, and *individual* tessellation of surface patches as needed. Our full tessellation algorithms ensure watertight tessellation: adjacent surface patches are tessellated with matching vertices along the shared edges (even for displaced surfaces). However, for some types of surfaces, we use individual tessellations which only consider one surface patch at a time. Those tessellations do not necessarily match along edges. (We could easily match tessellation rates along edges, but that would be futile unless post-displacement vertex positions could be forced to be consistent for neighbor grids.) Mismatched edges can give pinholes (ray misses) and false self-intersections unless handled carefully.

For object instances, the tessellation's target micropolygon sizes can be specified in world space, determined by the screen size of the master object, or determined by the screen size of the instance that is closest to the camera.

6.2 Bounding volume hierarchies

The surface patches of the scene are organized into a bounding volume hierarchy (BVH) [Kay and Kajiya 1986] which is a tree with bounding boxes at all nodes and actual geometry at the leaves. The BVH is built asynchronously on-the-fly as rays descend through it, causing large surface patches to be split into smaller patches (if needed) when a ray hits the bounding box of a patch. Procedural on-demand loading of geometry causes asynchronous rebuilds of parts of the BVH tree. Groups of objects with different visibility flags and trace subsets are stored in separate trees. For polygon meshes, a BVH leaf node currently consists of 4 or 8 triangles or quadrilaterals (depending on the geometry type). For other surface types, a BVH leaf node contains a tessellated surface patch in the multi-resolution geometry cache.

6.3 Rays and intersection tests

We trace ray bundles of up to 64 rays through the BVH when the rays are coherent, and mask off rays or split the bundle when they are not coherent. When the traversal reaches a leaf node, we intersection test one ray at a time with 4 or 8 triangles or quadrilaterals. In the spirit of Intel's Embree library [Wald et al. 2014], our implementation has an abstraction layer that keeps low-level vectorization details and instruction intrinsics (and changes) hidden from most developers.

6.3.1 Instances. Instances of objects are represented with just a transformation matrix, a material ID, and a pointer to the master object. Instances can have different material than the master, but must have the same displacement. Figure 9 shows examples of instancing from the short movie *Piper*; each sand grain instance has a different color and amount of refraction.

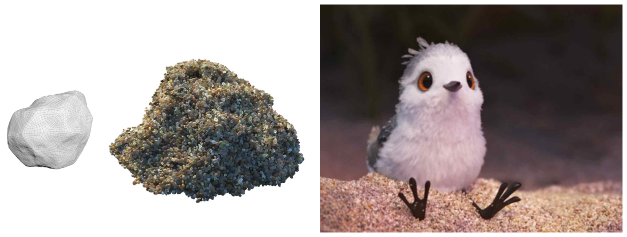


Fig. 9. Object instancing: a grain of sand consisting of 5,000 polygons, 50,000 instances of 20 sand grains, and millions of instanced sand grains (© 2016 Disney•Pixar).

6.3.2 Curves. Curves are used to represent hair and fur – see the tiger in Figure 1. Hair and fur geometry is usually very dense, so efficiently representing and ray tracing against curves is important. There are two main types of curves: flat "ribbons" with an orientation determined by a normal (for e.g. blades of grass), and cylindrical "round" curves (for hair and fur). The basis of the curves can be linear, Bézier, B-spline, or Catmull-Rom – allowing the user or modeling package to define curves in a way that is most convenient for them.

Groups of curves with a shared material are defined as a single object. The groups of curves can be split into smaller groups ("curvelets") and also split length-wise. The latter is important for efficient BVHs for very long hair – in extreme cases hair strands can stretch across the entire screen (what we affectionately call "the Rapunzel problem").

Groups of curves are organized in a bounding volume hierarchy with a mix of axis-aligned and curve-aligned bounding boxes. Our ray-curvelet intersection test splits curvelets on-the-fly depending on the ray radius. Then each curvelet segment is intersection tested using vectorized instructions to test one ray against 4 or 8 segments simultaneously. Related curve intersection methods have been described by Nakamaru and Ohno [2002] and Woop et al. [2014].

6.4 Trace bias and numerical accuracy

To avoid false self-intersections we offset the ray origins by a small amount – this is a standard ray-tracing trick [Woo et al. 1996]. The offset amount is called "trace bias" and is automatically computed by RenderMan. In our setting, two types of trace bias are needed internally:

1) For individually tessellated patches, the bias has to be large enough to overcome mismatched tessellations and displacements along patch edges. The size of this bias is proportional to the size of the micropolygons.

2) Bias to overcome numerical precision. On geometry very far from the coordinate system origin, we want the bias to be large enough that when the ray origin is offset by the bias, the bias doesn't get "drowned out" by the floating-point magnitude of the pre-bias ray origin.

When an entire surface is tessellated (and optionally displaced) at the same time, we can easily "weld" the grid vertices back together along patch edges (even after displacement). This eliminates the need for the first type of bias. However, enforcing matching tessellation along shared patch edges for individually tessellated patches would require e.g. the irregular tessellations described by Fisher et al. [2009] and some creative thinking for surfaces with displacement shaders.

As future work, we would like to implement individual watertight tessellations with post-displacement welding; that would allow us to reduce trace bias to an absolute minimum as determined by floating-point accuracy [Ize 2013; Pharr et al. 2017].

6.5 Motion blur and depth of field

To render motion blur, each camera ray is assigned a time within the shutter interval. The times for camera rays for each pixel are stratified to reduce noise: the ray times are stratified in 1D, and the combined pixel positions and times are stratified in 3D. Non-camera rays inherit the time of their parent ray.

Each node of the bounding volume hierarchy contains a bounding box for shutter open time and one for shutter close time. (Or, more generally, $s + 1$ bounding boxes for s shutter intervals.) The leaf nodes contain tessellated micropolygon grid positions for each $(s+1)$ shutter interval boundary. For ray intersection tests, the bounding boxes are interpolated according to the ray's time. For deforming geometry, the tessellated micropolygon grid positions in the leaf nodes are also interpolated before intersection testing.

For depth of field effects, the pixel positions are stratified in 2D, the lens positions for a pixel are stratified in 2D, and the combined pixel position and lens position samples are stratified in 4D (using precomputed sample tables as described in Section 7.1).

6.6 Path differentials

Igehy [1999] expressed ray differentials as four vectors: for a ray $\mathbf{r} = \mathbf{o} + t\mathbf{d}$ the vectors are $\frac{d\mathbf{o}}{du}$ and $\frac{d\mathbf{o}}{dv}$ for the origin and $\frac{d\mathbf{d}}{du}$ and $\frac{d\mathbf{d}}{dv}$ for change in direction per unit travelled.

For path tracing, rather than the full generality of Igehy's ray differentials, we have chosen much simpler quantities: one float for the ray radius at the ray origin, and one float for ray spread (expressing the change in radius per unit travelled). This is sufficient because each pixel is sampled many times. (A similar argument has been made by Pharr et al. [2017].)

For camera rays from a pin-hole camera we set the ray radius to zero at the origin and the ray spread to correspond to a quarter pixel by default. We have empirically found that there is no need to use a smaller spread (except for high-frequency bump mapping).

The radius r and spread s after a ray has travelled distance t are:

$$r_t = r_0 + t\, s_0; \quad s_t = s_0.$$

For reflection and refraction, the new ray's radius at its origin is the same as the incident ray's radius at the hit point. For specular reflection, the change in ray spread is determined by the ray radius and surface curvature:

$$r_0' = r_t; \quad s_0' = s_t + 2\,\kappa\,r_t$$

– where κ is the local mean surface curvature. The factor of 2 comes from differentiating the formula for mirror reflection direction [Igehy 1999]. For specular refraction the change in ray spread is determined by the ray radius, surface curvature, and the relative index of refraction. (We calculate curvature on non-displaced surfaces from analytical surface derivatives, and on displaced surfaces from the normals of neighboring micropolygons.)

For glossy and diffuse reflection (and transmission) we use an empirically determined heuristic based on the pdf of the chosen scattering direction, where lower pdf gives higher spread.

7 PROGRESSIVE AND ADAPTIVE RENDERING

Progressive and adaptive path tracing requires progressive sample sequences, good error measures, and the ability to store and resume in-progress images.

7.1 Sample sequences

Sample patterns can be divided into two categories: finite, unordered sample *sets* and infinite, ordered sample *sequences*. Every prefix of a progressive (a.k.a. hierarchical) sample sequence is well distributed.

For non-progressive, non-adaptive rendering, we only care about the final image, and could use finite sample sets such as jittered samples [Cook et al. 1984; Mitchell 1996], correlated multi-jittered samples [Kensler 2013], randomized versions of quasi-random sets such as the Hammersley [1960] or Larcher-Pillichshammer [2001] sets, or special sets with desirable spectral properties [Ostromoukhov et al. 2004]. But for progressive rendering the intermediate samples are also important. Figure 10 shows area light sampling with 100 samples from a set with 400 jittered samples vs. the first 100 samples from a progressive sample sequence. The advantage from using a sequence is evident. Likewise for adaptive sampling: we don't know how many samples we'll end up taking in each pixel, so we'd like the quality of the first samples to be as good as possible.

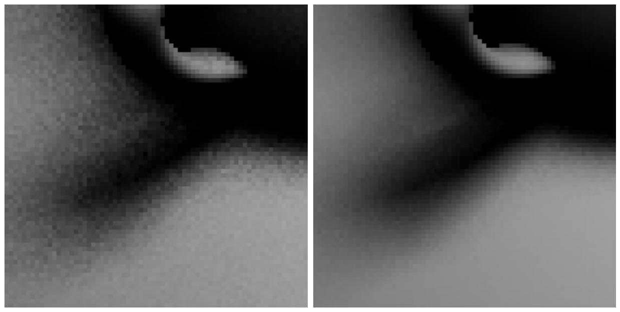


Fig. 10. Penumbra region with 100 samples per pixel: 100 samples from a stratified sample *set* with 400 samples vs. the first 100 samples from a progressive sample *sequence*.

Comparing the quality of sample sequences is a complex and multi-faceted task. Ranking according to star discrepancy is highly misleading [Dobkin et al. 1996; Mitchell 1992; Shirley 1991]. Instead, we have performed many tests by sampling functions and images. In the following we describe a few simple but representative tests of 2D and high-dimensional sample sequences.

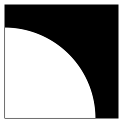
7.1.1 *2D sample sequences.*

Best candidates ("blue noise") sequences [Mitchell 1991; Reinert et al. 2015] have desirable spectral properties, but the same slow convergence rate as random samples. Ahmed et al. [2017] recently introduced an elegant way of generating blue noise sequences on a tiled lattice, thereby enforcing a regular jitter structure (stratification). These sequences have faster convergence rate than best candidates.

Randomized versions of quasi-random sequences such as the Halton [Fauré and Lemieux 2009; Halton 1964] or Sobol' [Joe and Kuo 2008; Sobol' 1967] sequences have fast convergence, are simple to use, and their samples can be generated very efficiently during rendering.

For an informative comparison of these 2D sequences, we use them to estimate the integral of a disk function,

$$f(x,y) = \begin{cases} 1 & \text{if } x^2 + y^2 < 2/\pi, \\ 0 & \text{otherwise,} \end{cases}$$



by sampling the function over the unit square. The disk radius was chosen such that the correct value of the integral is 0.5.

The plot in Figure 11 shows sampling error as function of the number of samples; each curve is the average of 10000 trials. The plot shows the error for sampling with uniform random, best candidates, Ahmed, Halton, and Sobol' sequences. The Halton sequence is randomized with rotations (toroidal shifts a.k.a. Cranley-Patterson rotations [Cranley and Patterson 1976]) and random digit scrambling; the Sobol' sequence is randomized with rotations, bit-wise xor [Kollig and Keller 2002], and Owen scrambling [Owen 1997]. For this discontinuous function, the error for uniform random and best candidates converges as $O(N^{-0.5})$, while the error for the Ahmed, Halton and Sobol' sequences converges roughly as $O(N^{-0.75})$.

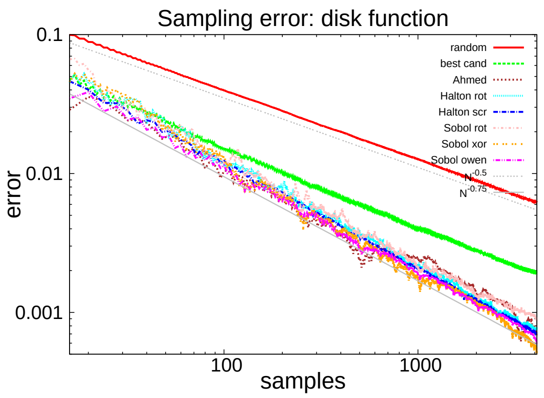


Fig. 11. Error for sampling the disk function with different sample sequences.

For sampling of smooth functions (for example a 2D Gaussian or a bilinear function $f(x,y) = xy$), the random and best candidates sequences still converge as $O(N^{-0.5})$. Ahmed sequences and nearly all combinations of quasi-random sequence and randomization converge as roughly $O(N^{-1})$. However, the 2D Sobol' sequence with Owen scrambling has a remarkably fast convergence: roughly $O(N^{-1.5})$ for power-of-two numbers of samples [Owen 1997]. So for sampling of two-dimensional functions, an Owen-scrambled Sobol' sequence is a really good choice.

7.1.2 High-dimensional sample sequences. For higher dimensions, it is tempting to just use high-dimensional quasi-random sequences. Unfortunately, the Halton and Sobol' sample sequences have increasingly poor distributions in pairs of higher dimensions – even with good randomization.

The plot in Figure 12 again shows sampling error for the 2D disk function. This time the error curves are for pairs of higher dimensions of the best quasi-random sample sequence we know of: Joe and Kuo's version of the Sobol' sequence, with Owen scrambling, as discussed above. The bottom (magenta) curve shows good convergence – roughly $O(N^{-0.75})$ as before – for the lowest two dimensions of the samples. But for higher pairs of dimensions, the error quickly gets worse. For some of the dimensions, the error is very erratic and the convergence is sporadic and unpredictable – sometimes the error is even as high as for random samples! This is not a particularly contrived example: it could be sampling of the shadow of a sphere from a square area light after a few bounces of narrow glossy reflection (with pixel positions, lens positions, time, and each glossy bounce "consuming" sample dimensions). We see similar errors for randomization using bit-wise xor, and even worse errors with rotations. There are similar problems with the Halton sequence, with other variations of the Sobol' sequence, and with sampling of other functions than a disk.

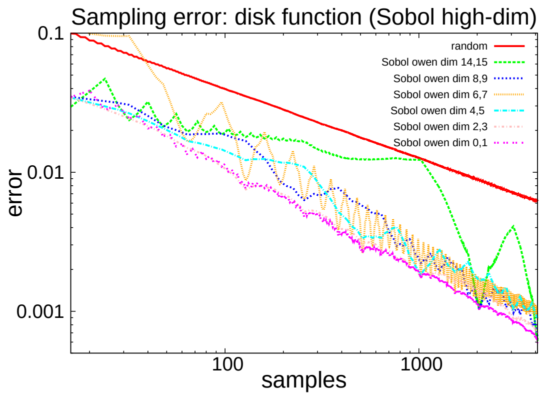


Fig. 12. Error for sampling the disk function with different pairs of dimensions from a Sobol' sequence. (Error for uniform random shown in red for reference.)

7.1.3 Progressive multi-jittered sample sequences. To avoid the convergence problems described above, we use carefully crafted stochastic samples. We start with 2D progressive multi-jittered (0,2) sample sequences ("pmj02") inspired by the progressive jittered sample sequence construction of Dippé and Wold [1985] and Kensler's correlated multi-jittered (cmj) sample sets [Kensler 2013]. Our pmj02 sequences are stratified in 1D and all 2D elementary intervals for power-of-two sample counts, i.e. they are (0,2)-sequences in base 2. Their error matches the best quasi-random sequence: $O(N^{-0.75})$ for discontinuous functions and the extraordinarily low $O(N^{-1.5})$ for smooth functions. The sequences are theoretically infinite, but in practice we truncate them at 4K samples in pre-generated tables (and resort to other means if more than 4K samples per pixel are needed in each dimension).

We then combine ("pad") the 2D pmj02 sequences by locally shuffling the order of sample values in all but the lowest pair of dimensions such that the samples are stratified in higher dimensions (as well as in the original 1 and 2 dimensions) – this gives lower sampling errors in 3D, 4D, etc. than randomly padding the 2D sequences together as in e.g. Cook et al. [1984] and Kolb et al. [1995]. Sampling a 2D function such as the disk with higher pairs of dimensions from one of these high-dimensional sequences will – by construction – have the same low error as with the first two dimensions since only the sample order was changed.

Nonetheless, the main point here is that none of the sample sequences we know of have a great distribution simultaneously in all pairs of dimensions and in all higher dimensions. Developing better high-dimensional sample sequences for progressive path tracing is an open area of research with many conflicting goals and constraints. See also the interesting "wish list" by Hickernell [2003].

7.1.4 Initial samples. For interactive rendering it is desirable that the first samples are as visually pleasing as possible. This can be accomplished by placing the initial samples using a best-candidates (blue-noise) distribution [Georgiev and Fajardo 2016] or using a single quasi-random sequence for the entire image [Grünschloß et al. 2010]. (However, the latter approach is prone to visual aliasing for some numbers of samples.)

We use a different method to obtain the same goal. The initial pixel sample positions are coordinated within groups of 4×4 pixels such that they are stratified in all elementary intervals: $[4, 0.25]$, $[2, 0.5]$, ..., $[0.25, 4]$. Same for lens samples for depth-of-field, etc. Given a desired initial sample position, we could select the sequence among our tables with the first sample closest to the desired position. Instead we pick a sequence at random and apply a bit-wise xor to move the initial sample to the desired position in the pixel, and then use the same xor bit pattern for all the following samples in that pixel.

7.2 Adaptive pixel sampling

Different areas of an image may converge at different speeds when rendering. For example, areas where the rays miss the geometry completely will trivially converge to the background color, and flat, well-lit parts also tend to converge quickly. On the other hand, phenomena such as penumbras, caustics, glossy surfaces, and moving objects may take much longer to converge. For this reason, it is useful for the renderer to be able to detect pixels that have converged and focus all its effort on the unconverged pixels. This is the job of the adaptive sampler.

In practice, this involves a tricky balance. If the adaptive sampler is too conservative and shoots mostly the same camera rays that a uniform, fixed sampler would have, then it provides little benefit; to be worth the effort it must save more in rendering time than its own computational overhead. On the other hand, if it is too aggressive then it may make the render quick but it could miss rare-but-important samples which will lead to flickering under animation.

One other concern that we had – specific to production film rendering – was memory use in the face of many additional outputs (arbitrary output variables, also known as AOVs, such as depth, motion, surface normal, etc.). Given that productions often render out dozens of AOVs for compositors to use, any implementation that required keeping detailed statistics for each pixel of each AOV could prove quite costly in framebuffer memory.

To avoid this, our initial implementation operated on a tile at a time and used a contrast-based approach inspired by Mitchell [1987]. Though the tile level proved too coarse, using contrast had some nice benefits – particularly its insensitivity to the absolute light exposure level. For production use, this means compositors can adjust the exposure on a rendered image at will and the noise level will remain relatively constant. It also avoids oversampling of blown-out pixels.

Subsequently, we changed the adaptive sampler to operate on individual pixels, keeping a counter of samples remaining to render for that pixel. If a sample outside the convergence criteria is detected for any channel, we reset the counter for that pixel and each of its neighbors. Resetting the neighbors as well greatly helps with relatively rare but important samples such as from bokeh.

As pointed out by Rousselle et al. [2013], standard statistical variance does not capture the improved convergence of high-quality sample sequences. Our default convergence criteria effectively asks what the relative difference between a pixel before and after adding the sample to it would be, and compares that difference with a user-specified threshold. This relative measure preserves the independence from the absolute exposure while also not incurring any memory costs beyond what we already needed for pixel filtering.

Over time, we have extended this algorithm. The convergence threshold can be adjusted per-object, allowing more samples on hero or problematic objects and fewer on backgrounds. Different channels may also have different convergence thresholds. User-created AOVs that track the estimated variance of a primary AOV can be used directly; this allows for a more traditional variance-based metric where the user explicitly opts-in to the extra memory cost. We also added a control to limit oversampling in very dark regions.

Note that the adaptive sampler is intentionally affected by sample filter plug-ins. This way it can be made to operate in other (e.g., perceptual or filmic) color spaces or to consider tone mapping. Savvy users can also programatically drive the adaptive sampler via special AOVs.

7.3 Progressive rendering and checkpointing

One of the primary reasons for choosing path tracing over distribution ray tracing as a light transport algorithm is that complete paths can be easily sampled and added to pixels in any order. Rather than sampling all of the paths through a pixel at once, RenderMan can visit each pixel many times, progressively refining it with additional samples. For interactive use, this allows artists to quickly see an initial noisy rendering which then refines over time.

For batch use, the renderer can periodically generate what we call checkpoint images. These are complete, standard images, viewable in any OpenEXR reader, that carry additional metadata that the renderer can use to resume where it left off if it is stopped and restarted. Checkpoint images can be written after some number of passes over an image, or after a set amount of time. The renderer can also write a checkpoint image and exit gracefully when a time limit is reached.

This has become an extremely valuable tool in production. A sequence of frames can be sent to the render farm with the constraint that each frame take no more than a given time between checkpoints. Within minutes, an artists can view the checkpoint images and be confident that they have no broken renders. After further refinement, the resulting partially converged image sequence is often good enough for approval in a daily review. If it passes, the renders can be resumed and allowed to fully finish. Time-slicing the renders using checkpoints ensures that no one goes to a daily review empty-handed.

On the downside, it should be mentioned that this strategy (one sample per pixel in each iteration) for progressive path tracing increases the total render time for the final image by 10-30% compared to non-progressive rendering. The problem is the scattered data accesses: most data (geometry, textures, etc.) is accessed in each iteration. In contrast, for non-progressive rendering, each thread keeps iterating over an image tile until it is finished, and only then moves to the next available tile, giving excellent data locality. In the future we want to implement power-of-two progressions, i.e. progressive rendering where the number of samples per pixel doubles in each iteration. We believe this would give the best of both: quick iterations initially for fast feedback, but more coherency in later iterations.

8 PATH SELECTION AND MANIPULATION

Light path expressions [Gritz et al. 2010] are used to distinguish between the different paths that light can take from the light sources to the eye. This is useful for noise reduction, compositing, and debugging: it can be utilized to change the intensity of certain light paths, or even delete some paths entirely. For example, caustic paths might contribute a lot of noise to path-traced images without any desirable visible contribution, so they can be explicitly omitted from the final images.

Light path expressions can be written concisely using Heckbert's regular expression notation [Heckbert 1990]: E denotes the eye (camera), L denotes a light source, D is diffuse reflection, and S is specular reflection. '|' is a choice between two paths, '*' means zero or more repeats, '+' means one or more repeats, and '[x]' means x is optional. For example, light directly from the light source to the eye is LE, and light reflected any number of times is $L(D|S)^*E$. Our extensions of this notation allow distinction between reflection and transmission, and between light sources and emissive geometry. We also allow expressions with individual light source names.

Light path expressions can be written to arbitrary output variables in the rendered image. For example, it can be useful to write not only the complete ("beauty") image but also the direct lighting, surface albedo, specular and diffuse reflections separately. Such separate channels are required as input to the denoiser described in Section 13.

Even though path tracing is based on physical simulation, it is possible to take many of the same liberties as with Reyes [Barzel 1997]: some lights might only illuminate certain objects, some objects may only cast shadows on certain other objects, some ray paths can be explicitly omitted, reflection ray directions can be "bent", some light paths can be enhanced or reduced, color bleeding from one particular object to another can be increased or decreased, caustics and secondary specular reflections can be deleted, brightened, or blurred, etc. Our path tracer provides support for such non-physical manipulation; this is implemented using light filters. The survey by Schmidt et al. [2016] and course notes by Hery et al. [2017a] provide more details of such light path expression editing tools.

9 INTERACTIVE RENDERING

Traditionally RenderMan has been used for "off-line" rendering of final-quality images. But we are currently pushing for more interactive use, for example during modeling, texturing, lighting, and animation.

One of the advantages of being a pure path tracer is that anything can be edited at any time: all shading and light transport is re-evaluated on every render. During interactive rendering, RenderMan is put into a progressive rendering mode that iterates over camera samples, showing a progressively refined (less noisy) image as each round of camera rays is traced. Updates to the renderer are typically driven by a scene translator embedded in an application such as Maya or Katana. When the user updates something in the scene, the renderer is paused, an edit sent, and the rendering restarted at the first iteration of the progressive integration. Light transport algorithms (integrators) and their parameters, materials (bxdfs), pattern networks, lights, light filters, display filters, and projection plug-ins may be edited or replaced. The camera and lights can be moved.

Conspicuously missing from the list above is the ability to move scene geometry. While RenderMan is no longer a hybrid renderer, the current version has retained an optimization put in place for the scanline renderer: geometry, including all per-vertex data, is stored in camera space. This was done to accelerate z-buffer hiding (since a camera-to-raster space transformation is trivial) and shading (which was also performed in camera space). In addition, the bounding volume hierarchies were built with the geometry from separate but overlapping objects intermingled. Given this design, moving geometry would be cumbersome and was thus not implemented. We are currently lifting this restriction by no longer storing geometry in camera space and by segregating the BVHs so that it is trivial to move, add, or delete objects or groups of objects. Figure 13 shows an example: a screen grab of an interactive scene editing session in Maya with RenderMan in the viewport. A similar interface exists for Katana.

The RenderMan Interface (RI) is a streaming API that provides a hierarchical scene description. This is well-suited to an efficient serialized representation of a static scene. However, it is not a natural fit for interactive scene editing for several reasons: naming scene components unambiguously is a challenge; bindings between objects (for example material to geometry) are implicit in the stream while

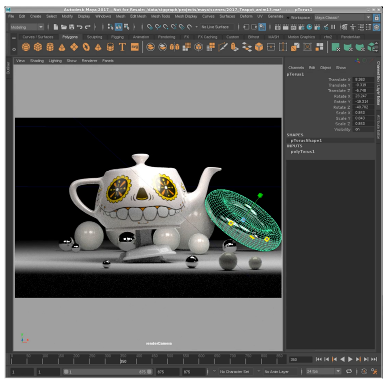


Fig. 13. Interactive session with RenderMan in the Maya viewport. The chrome torus (highlighted in green) is selected for geometric editing. (Maya integration by Katrin Bratland.)

an application that is editing a scene would prefer the binding to be explicit; not all applications that make use of interactive rendering need a hierarchical representation of a scene, but all would benefit from a direct, easy to modify representation.

To satisfy these goals we have introduced two new APIs: a light-weight scene graph, and a low-level interface to the renderer that we call Riley. The scene graph provides the core representation upon which we build translators from modeling and scene management applications. This is a natural fit for these applications. The scene graph can serialize a scene to RIB, or it can drive an interactive rendering through Riley. Riley maintains no hierarchical state, expecting all attributes and transforms to be flattened. Handles are associated with objects (e.g. geometry, materials, lights, cameras) in the API and bindings are explicit. The scene graph maintains handles associated with all of the scene objects it creates. Any change in the scene graph results in an edit to Riley that is direct and straightforward to express. Both the scene graph and Riley are thread-safe, providing high-performance multi-threaded export from modeling applications to the renderer.

Riley is a natural fit for a scene graph, but it is also well suited as the renderer interface for a hierarchical streaming scene description. We have ported our implementation of RI to Riley – both of our higher-level scene descriptions use Riley.

It should be mentioned that there has also been prior work on interactive renderers based on deep frame-buffers generated by RenderMan, for example Lpics [Pellacini et al. 2005], Lightspeed [Ragan-Kelley et al. 2007], and a product called Lumiere. Lpics ran on a GPU and used simplified versions of RSL shaders. It recomputed the illumination and shadows from each light individually if there was a change in that light. Lumiere supported shading and lighting edits, but relied on a deep buffer that used huge amounts of memory. Lpics and Lumiere were limited to static scenes – no change in geometry or viewpoint were possible – whereas Lightspeed allowed limited moving geometry. Another early push in the direction of interactive rendering was a very nice hack (what we call a "Stupid" RenderMan Trick) that turned the distribution ray tracing version of RenderMan into a progressive path tracer [Grabli et al. 2012].

10 INTEGRATORS FOR SIMPLE RENDERING

We provide several integrators that implement simple rendering algorithms that are useful during interactive modeling, scene layout, surface material and texture development, and other uses where realistic rendering is not desired or necessary.

The visualizer integrator can show the surface patch parameterization, surface normals, diffuse shading "illuminated" from the camera position (without shadows), material IDs, etc. This can optionally be combined with a grid overlay showing the patch boundaries. Figure 14 shows a few examples. This integrator is extremely quick since it only does direct shading: no rays are traced from the camera ray hit points. It can be used in Maya's viewport, and can be faster than a GPU rasterizer for complex scenes.

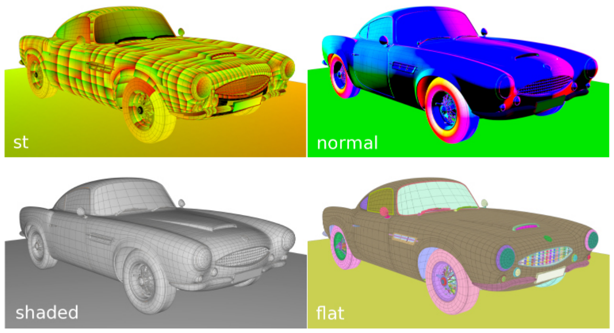


Fig. 14. Visualization of a car showing (s, t) parameters across surface patches, color representation of surface normals, diffuse reflection with grid overlay, and material IDs. (Images courtesy of Philippe Leprince.)

Figure 15 shows an example of ambient occlusion [Landis 2002; Zhukov et al. 1998]. This integrator shoots rays over the hemisphere above each camera ray hit point, but there is no shading at secondary hit points, no tertiary rays, and no direct illumination. Ambient occlusion rendering is very fast, shows surface details, and provides a good indication of where objects are located relative to each other.

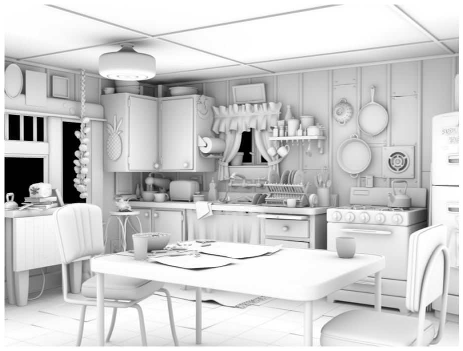


Fig. 15. Ambient occlusion in a kitchen scene. (Scene courtesy of Christina Faraj, Mike Altman, and Rosie Cole.)

LollipopShaders is developing an integrator for non-photorealistic cartoon, painterly, watercolor, and crosshatching looks [LollipopShaders 2018].

11 PATH TRACED RENDERING

The uni-directional path tracer is the integrator that is used most for movie rendering with RenderMan. Figure 16 shows a complex scene from *The Jungle Book* rendered with this integrator.

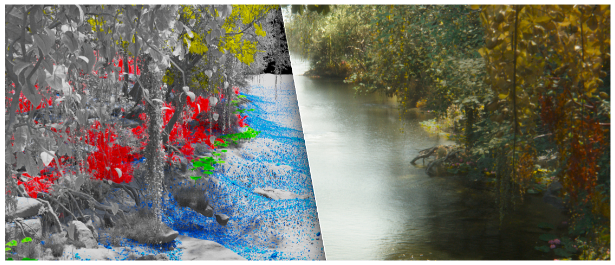


Fig. 16. Path traced scene from *The Jungle Book* movie. The left half shows a break-down of the procedurally generated geometry; the right half shows the final image. (© 2016 Disney/MPC. Image courtesy of MPC.)

11.1 Path tracing of surfaces

The path-tracing integrator calls pattern plug-ins to evaluate surface parameters at camera ray hit points, calls the renderer to compute direct illumination at those points (including tracing of shadow rays), calls the bxdf plug-ins to generate new rays, calls GetNearestHits() to trace those rays, receives new groups of ray hit points, and iterates again until no new rays remain or the maximum ray depth has been reached.

Although path tracing can be described elegantly as a recursive algorithm, our implementation is iterative and uses double-buffering (i.e. writing to one memory buffer while reading from an other, and swapping the buffers between iterations). This makes the working set of rays and shading data smaller, and independent of the trace depth. It does, however, mean that path splitting has to be limited; we only allow more than one indirect ray at camera ray hit points, not at deeper levels. (This restriction is in line with the classic path tracing techniques.)

As mentioned previously, we shade groups of shading points with the same material together, thus amortizing pattern evaluation overhead and maximizing texture lookup coherency. These shade groups often have up to a few hundred shading points for the first bounce, but less than that for deeper bounces.

Our path tracer uses standard techniques such as Russian roulette [Arvo and Kirk 1990] and explicit sampling of light sources (next-event estimation). In addition, it also has some fairly unique features: It is possible to set the visibility for each object – is it visible to camera, indirect, and/or transmission rays? Caustics can be turned off (to eliminate noise from undesired caustic light paths). At the first bounce, the number of bxdf samples and the number of direct illumination samples can be set higher than 1 (and at different values) – thus allowing fine-tuning of noise reduction. The maximum ray depth can be set separately for diffuse and specular. A typical use of this flexibility is a scene with several layers of glass: high maximum specular depth is needed, but low diffuse depth might be sufficient.

Figure 17 shows a path-traced image of a collection of RenderMan walking teapots with a *Coco*-themed paint scheme.

Fig. 17. *Coco*-themed teapots. (Image courtesy of Dylan Sisson.)

11.2 Subsurface scattering

Subsurface scattering in skin and other translucent materials is rendered with efficient local ray tracing. Subsurface scattering can use the dipole diffusion [Jensen and Buhler 2002; Jensen et al. 2001] or Burley normalized diffusion [Burley 2015; Christensen and Burley 2015] models. The dipole diffusion model has a very soft "waxy" or "cartoony" look, while normalized diffusion has a more correct sharper look suitable for realistic rendering of e.g. human skin. For these diffusion models, we compute subsurface scattering in a manner similar to that previously used for distribution ray tracing; mainly the ray branching factor is different.

When a ray hits a surface with subsurface scattering, we first trace "surface probe rays" using two sampling strategies: spherical and planar. The planar sampling strategy uses importance sampling based on the cumulative distribution function (cdf) of the diffusion profile – this is particularly simple for Burley's profile which has an analytic cdf that is easy to invert numerically. Figure 18 shows two examples of realistic VFX movie characters rendered with RenderMan's ray-traced diffusion subsurface scattering.

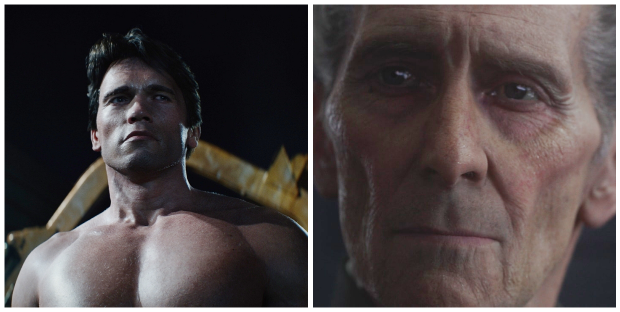


Fig. 18. Subsurface scattering: a young Arnold Schwarzenegger from *Terminator Genisys* (© 2015 MPC) and Grand Moff Tarkin from *Rogue One: A Star Wars Story* (© & ™ 2016 Lucasfilm Ltd. All rights reserved. Used under authorization.)

Recently, Pixar's in-house tools illumination team implemented a brute-force path-traced random-walk Monte Carlo approach that simulates subsurface scattering as scattering in a dense homogeneous volume [Chiang et al. 2016; Křivánek and d'Eon 2014; Meng et al. 2016; Wrenninge et al. 2017]. It was fairly simple to integrate this new method with RenderMan using our API for interior integrators. Figure 19 (left) shows an ant; the thin legs and antennae are prime examples of cases where the infinite slab assumption built into the diffusion methods fails but this brute-force technique shines. Figure 19 (right) shows an example with realistic skin.

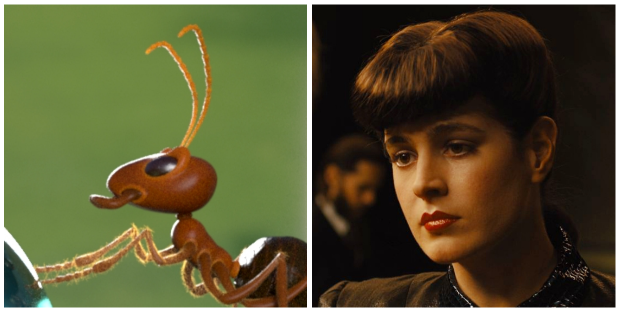


Fig. 19. Path-traced subsurface scattering: ant (original model by Sunny Chopra; additional modeling and rendering by Chu Tang) and Rachael from *Blade Runner 2049* (© 2017 Alcon Entertainment, LLC., Warner Bros. Entertainment Inc., and Columbia Pictures Industries, Inc. All rights reserved. Image courtesy of MPC.)

11.3 Path tracing of volumes

Rendering of volumes is more challenging than surfaces: direct and indirect illumination has to be computed at many points within the 3D volume to obtain a correct and noise-free color for each pixel.

Volumes are specified by volume density (extinction coefficients), albedo, and scattering phase function. Volumes can have homogeneous or heterogeneous density; for heterogeneous volumes the density data can be procedurally generated, or read from OpenVDB [Museth 2013] or Field3D files. These files are typically generated from simulations, for example cloud simulations. Since the volumes often come from separate simulations they can overlap or be nested within each other. The phase function can be isotropic or anisotropic. Volumes can have zero scattering (just attenuation with Beer's law), single scattering, or multiple scattering.

Path tracing of multiple scattering in homogeneous volumes is quite simple: repeatedly choose a random scattering distance (with exponentially decreasing probability), and at that distance choose between absorption or scattering. If scattering is chosen, the (isotropic or anisotropic) phase function generates a new scattering direction.

However, path tracing of heterogeneous volumes is unfortunately more complicated. Our emphasis is on performance for multiple bounces – which is important for e.g. clouds. Figure 20 shows a cloud rendered with single and multiple scattering. The simplest method to render heterogeneous volumes is using ray marching, but ray marching is too inefficient for multiple bounces using path tracing.

Delta tracking (also known as Woodcock tracking) [Raab et al. 2006; Woodcock et al. 1965] is an optimization that introduces a fictional medium that homogenizes the total collision density. With delta tracking there are three collision types: absorption, scattering,

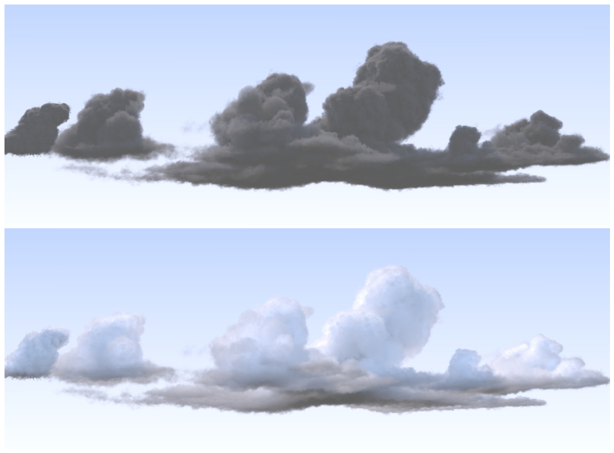


Fig. 20. Cloud with single-scattering (top) and multi-scattering (bottom). The cloud has anisotropic scattering: Henyey-Greenstein phase function.

and null collision; if null collision is chosen, we simply choose a new random distance in the same direction. Delta tracking requires an upper limit on the density of the volume. Residual ratio tracking [Novák et al. 2014] is a further optimization that uses a coarse transmittance as a control variate to reduce the number of look-ups of the volume density.

When volumes overlap – for example a bank of clouds that each has been simulated individually – the density of the combined volume is simply the sum of the densities of the overlapping volumes. For scattering, we probabilistically choose one of the volumes (based on the relative densities) and use the albedo and phase function of that volume. RenderMan keeps track of the volumes that a ray enters and exits, and integrates over all volumes covering a region. Figure 21 (left) shows two overlapping homogeneous volumes.

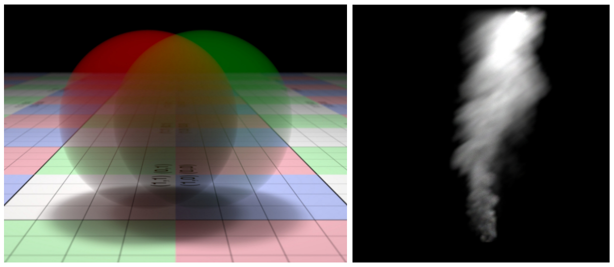


Fig. 21. Left: two overlapping volumes with different density and albedo. Right: volume with motion blur.

For volumes with deformation motion blur, we represent the density in each voxel as an approximate function over time within the shutter interval [Wrenninge 2016]. Figure 21 (right) shows a billowing smoke volume. Path tracing of volumes is covered in much more detail in the course notes by Fong et al. [2017].

12 OTHER PATH TRACING METHODS

As mentioned previously, the modern RenderMan architecture was designed from the outset with bidirectional path tracing and VCM in mind.

12.1 Bidirectional path tracing

With bidirectional path tracing [Lafortune and Willems 1993; Veach and Guibas 1994], paths are traced both from the camera and from the light sources, and then connected. Bidirectional path tracing is advantageous for scenes dominated by indirect illumination or with strong caustics.

In each iteration, we trace one light path and one eye path per pixel, and connect the vertices of the two paths with transmission rays (giving either full, partial, or no light contribution for each connection).

Figure 22 shows a scene where the illumination is provided by a sconce light directed upwards. Nearly all illumination in this scene is indirect, and unidirectional path tracing has a hard time finding the light source – hence Figure 22 (left) is very noisy even though it has 2048 samples per pixel. With bidirectional path tracing the light paths automatically originate at the light source, so there is no problem finding the light source: Figure 22 (right) is very clean. Each iteration of bidirectional path tracing takes longer than unidirectional; the right image has 700 samples per pixel and took the same time to render as the left image.

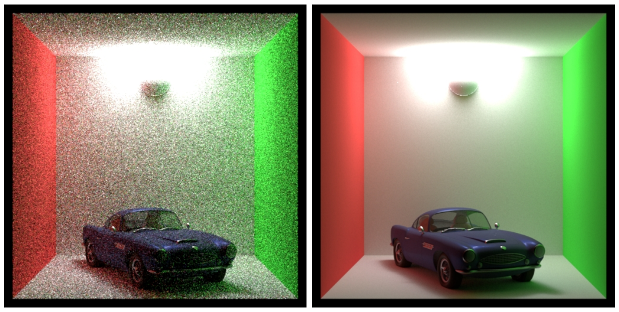


Fig. 22. Car scene rendered with unidirectional and bidirectional path tracing in the same amount of time.

There is an extra bonus when using bidirectional path tracing: surface shader results (which are often very expensive to compute due to many texture map look-ups, procedural texture evaluations, complex pattern networks, etc.) are reused for several light transport paths when light paths and camera paths are combined.

A practical problem with bidirectional path tracing is that bxdfs have to be written such that they are reciprocal; this requires careful consideration and debugging. Another problem we have seen in complex production scenes is that incoherent texture access patterns for the light paths can cause texture cache thrashing. Unfortunately path differentials are not as useful as for unidirectional path tracing: even when a light path undergoes diffuse reflection (or specular reflection from a highly curved surface) we can not necessarily assign a large differential to it – the light path could subsequently hit a surface point right in front of the eye. Even the work on covariance tracing [Belcour et al. 2015] for bidirectional path tracing only specifies the optimal texture filter sizes after a light path has been traced and connected to an eye path; it does not solve the question of optimal filter sizes *during* light path tracing. We believe this is an important area to improve in the future.

12.2 VCM and UPBP

We also have an integrator implementing vertex connection and merging (VCM) – also known as unified path sampling (UPS) – [Georgiev et al. 2012; Hachisuka et al. 2012]. This rendering algorithm is a combination of bidirectional path tracing and progressive photon mapping. It is particularly advantageous for rendering of caustics and reflections of caustics.

We believe that a future "grand unified rendering algorithm" will contain some elements of (progressive) photon mapping, so we made sure to design the plug-in interface such that it can handle this.

As for bidirectional path tracing, we trace one light path and one camera path per pixel in each iteration. In addition, we store the vertices of the light paths in a photon map. We also look up photons within a radius – this is the merge part of the algorithm. The initial photon look-up radius at a point is proportional to the path differential for a (hypothetical) camera ray hitting that point, and shrinks by a constant factor in each iteration. The light paths for direct illumination, bidirectional path connections, and photon map lookups ("path merges") are combined with multiple importance sampling.

Incremental rendering and only tracing one light path per image pixel in each iteration keeps photon map memory usage reasonable. We use two photon maps: one is being written asynchronously by multiple threads (multiple image tiles being rendered simultaneously), while the other is being read for merges. This avoids any read-write conflicts. The two photon maps are swapped between iterations.

Figure 23 shows a tricky caustic image. On the left is an area light source emitting light based on a texture. A Fresnel lens in the middle focuses the emitted light. The right wall is diffuse and shows a caustic which is a refocused version of the light source texture. The crispness of the caustic is mostly due to the progressive photon mapping part of VCM.

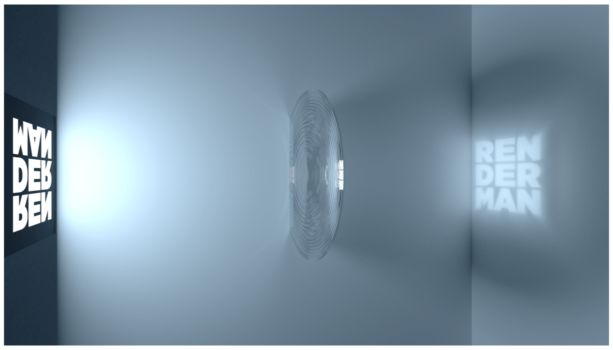


Fig. 23. Fresnel lens focusing effect rendered with VCM.

VCM was used in a separate rendering pass for underwater caustics on *Finding Dory*. VCM has also been used to render realistic caustics in human eyes. But it is probably fair to say that VCM is not widely used in production yet.

Křivánek et al. [2014] extended the VCM idea to volumes with a rendering algorithm called 'unified points, beams, and paths' (UPBP). Most light paths through volumes are stored as photon points (as before), but some are stored as photon beams. Light in the volumes is rendered with a combination of the photon points and beams – points are best in dense volumes while beams are best in sparse. Figure 24 shows an updated version of a classic *Luxo Jr.* scene, with added homogeneous volume and glass sphere, rendered with our experimental UPBP integrator. (The UPBP integrator is fully featured, but currently does not handle heterogeneous volumes as efficiently as our VCM integrator.)

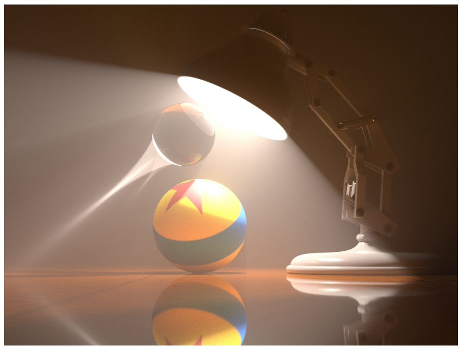


Fig. 24. Luxo lamp with volume and glass sphere rendered with UPBP. Note the volume caustic under the glass sphere and the reflection of the volume caustic in the varnished table top. (Image credit: Brian Savery and Martin Šik.)

12.3 Metropolis

Metropolis rendering algorithms [Kelemen et al. 2002; Veach and Guibas 1997] excel at finding difficult paths from light sources to the eye, for example illumination through a slightly-ajar door or a caustic path involving narrow glossy materials. When an important light path has been found, mutations of it are explored to discover similar light paths.

Unfortunately, the Metropolis algorithm has some practical shortcomings: 1) Because it focuses on important paths there are structured artifacts in the images. 2) Temporal instability – sudden "pops" when important paths are discovered during progressive rendering. 3) Metropolis rendering needs a "training phase" before actual rendering starts, which is not ideal for interactive use.

We are hopeful that the latest (and future) Metropolis rendering research papers – see for example Manzi et al. [2014], Pantaleoni [2017], and Bitterli et al. [2018] will address some of these shortcomings and make Metropolis a useful rendering algorithm for movie production. When this happens, RenderMan will support a Metropolis integrator.

13 DENOISING

Reducing noise in an image by increasing the number of samples per pixel has diminishing returns [Mitchell 1996], even with good variance reduction techniques. For this reason, RenderMan ships with a denoising filter to try to eliminate remaining image noise. To our knowledge, it was the first commercial renderer to do so. Figure 25 shows denoising on a frame from *Finding Dory*.

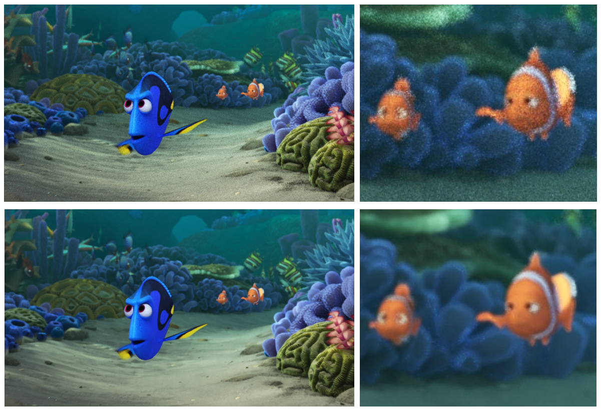


Fig. 25. A frame from *Finding Dory* before (top) and after (bottom) denoising. The right column shows a close-up of each version. (© 2016 Disney•Pixar.)

The denoising filter has been a subject of intense collaboration between Fabrice Rousselle (now at Disney Research), the Hyperion team at Walt Disney Animation Studios, Pixar's research group, our team, and Intel's Technical Computing Engineering team.

One of the simplest approaches to denoising is the bilateral filter [Tomasi and Manduchi 1998]. Like Gaussian blurring, this replaces each pixel with a weighted blend of its neighbors with the weights based on spatial similarity. However, it also adjusts the weights by color similarity in order to preserve edges. The cross bilateral filter [Eisemann and Durand 2004; Petschnigg et al. 2004] extends this to smooth one image based on the color similarities in a second image; this can also be thought of as a bilateral filter on a multi-channel image. The non-local means (NLM) filter [Buades et al. 2005] replaces the pixel-to-pixel color similarity of the bilateral filter with a patch-to-patch neighborhood similarity measurement.

Rousselle's approach [Rousselle et al. 2013], on which our denoiser is based, uses NLM with additional "features" produced by the renderer such as the normal, depth, and texture albedo to help preserve edges. This NLM filtering is done at multiple scales and the best filters for each pixel are chosen via Stein's unbiased risk estimator (SURE) using estimates from the renderer of the variance in each pixel. These are smoothly blended to produce the final image.

The denoiser has several improvements on Rousselle's original approach. For one, it normally divides out the texture albedo and alpha before filtering and then remultiplies them after. Most of the noise in the renders comes from direct and indirect illumination, and removing the texture like this provides more opportunities for denoising the illumination. However, it does require the albedo channel to be effectively noise free since any noise there will be kept.

The denoising filter also filters the diffuse and specular illumination separately. One of our extensions was to allow it to work on even finer subdivisions of the illumination – for example, separate primary specular and clearcoat specular. In RenderMan these subdivisions can be produced using light path expressions (LPEs). Filtering these components of the illumination separately and then combining them produces better results than simply filtering the final beauty pass.

The integration into RenderMan involved several additions to the renderer. For one, the denoiser requires that each pixel be statistically independent of its neighboring pixels. Otherwise, it may interpret noise splatted across several pixels as signal and mistakenly enhance it. Consequently, we implemented pixel filter importance sampling [Ernst et al. 2006] as an option in RenderMan. We also added the ability to compute variance estimates (both directly and from the difference of two half-buffers) and to estimate 2D motion vectors.

These motion vectors are used for cross-frame denoising. Given a sequence of frames, the denoiser will use the motion vectors to warp the frames immediately before and after the current one that it is filtering to try to align them. Then the NLM filters compare patches in the neighborhood from all three frames to blend them together. In animation this helps to reduce distracting flickering.

Denoising remains an area of ongoing research both at Disney and Pixar. Small speckles and glints from e.g. car paint or water waves can be difficult to distinguish from noise. The algorithm has been extended to thick volumes, but wispy, thin volumes remain challenging. More details about modern denoising techniques can be found in the papers and course notes of Rousselle et al. [2013], Zimmer et al. [2015], and Zwicker et al. [2015]. Recently, there have also been experiments in applying machine learning to denoising [Bako et al. 2017; Chaitanya et al. 2017].

Currently denoising is done separately after rendering, but it would be better to denoise images during rendering; this would provide images with less noise for faster decisions during progressive rendering.

14 CURRENT AND FUTURE WORK

In addition to the future work discussed throughout this article (for example geometric simplification inside the renderer, texture formats that better handle overmodeled surfaces, improved trace bias, better sample sequences, and on-the-fly denoising), there are several more topics of current and future work:

- Fully utilizing vectorized instructions is a big challenge, more so than traditional scaling and multi-threading. Thanks to our batched shading architecture, we have many opportunities to efficiently execute some of the costly parts of our code. Vectorization of scene data ingestion and parts of the raytracer are more challenging, in particular when targeting modern instruction-set architectures like 16-wide vectorized instructions (AVX512). Consideration must also be given to users with older hardware: we have to support the lowest common denominator of the hardware in use by our customers.

- We are currently implementing a large subset of RenderMan on GPUs using Cuda. The goal is to provide a super fast rendering solution that utilizes both CPUs and GPUs; hence we call this project "RenderMan XPU".

- Rendering in color spaces beyond RGB would be interesting – see for example the work by Wilkie et al. [2014]. The RenderMan specification has always allowed more than 3 color bands, but the RenderMan renderer has not. Implementing more color bands would allow rendering realistic images of dispersion effects such as prisms and colored sparkles in gemstones. We have not had much demand for this feature from movie production, but have received requests for it from customers in product visualization (gems, crystals, etc.).

- Learning algorithms for both direct and indirect illumination are promising areas of investigation. For indirect illumination, the dominant directions of incident light can be counted in fixed bins [Jensen 1995], represented with adaptive accuracy with Gaussian functions [Vorba et al. 2014], or with adaptive bins in space and directions [Dahm and Keller 2017; Lafortune and Willems 1995; Müller et al. 2017]. With this knowledge, the indirect illumination can be sampled better. For direct illumination, one could adaptively learn which light sources are blocked from certain parts of the scene (and hence can be sampled more sparsely).

- Larger shade groups may be beneficial for even better data locality. However, memory traffic and sorting of the shade groups become bottlenecks if the groups are too large. It would be interesting to determine the optimal size of shade groups and trace groups. See Áfra et al. [2016] for examples of such tests in a slightly different setting.

15 CONCLUSION

RenderMan has been substantially reinvented to support physically-based shading and light transport. RenderMan is now a pure path tracer with an extensible plug-in architecture. There are plug-ins for light transport algorithms, materials, patterns, displacements, lights, light filters, camera projections, and sample and pixel filtering.

RenderMan development is guided by a mission that has two parts: to be a production-ready renderer that handles immense complexity, and to be an extensible platform for studios to develop their own rendering solutions.

We are currently in a push toward more interactive uses of RenderMan. Whereas RenderMan traditionally has been targeting only final-frame movie-quality rendering, we are now also targeting interactive rendering during object modeling and texturing, scene layout, lighting, animation, etc. This has required us to focus more on start-up time and optimizations of time to first pixel and time to first frame (with one sample per pixel).

We encourage any interested reader to download a free non-commercial version of RenderMan at renderman.pixar.com.

ACKNOWLEDGMENTS

In addition to the authors of this paper, many other people have contributed to the development of RenderMan.

The Reyes algorithm that was the foundation for the original RenderMan renderer was developed by Rob Cook, Loren Carpenter, and Ed Catmull. The path tracing architecture of the modern RenderMan renderer was designed by Dana Batali and Brian Smits (plus some of the authors of this paper) with inspiration from the PBRT book.

RenderMan development over the past 30 years has been lead in turn by Rob Cook, Jeff Mock, Mickey Mantle, Tony Apodaca, Dana Batali, and Allan Poore. We thank them for their vision and guidance. Other significant RenderMan contributors through the years include Tom Porter, Pat Hanrahan, Darwyn Peachey, Dan McCoy, Mark VandeWettering, Bill Reeves, Tom Duff, Steve Upstill, Jim Lawson, Sam Leffler, David Salesin, Eric Veach, Tom Lokovic, Larry Gritz, Craig Kolb, Matt Pharr, Tien Truong, Reid Gershbein, David Laur, Brad West, Jamie Hecker, Mark Leone, Susan Fisher Fong, Luca Fascione, George Harker, Danielle An, Aaron Lefohn, Kayvon Fatahalian, Martin Šik, and many others.

Thanks to our RenderMan tools, infrastructure, and support team: Katrin Bratland, James Burgess, Ian Hsieh, Trina Roy, Philippe Leprince, Jeff Varga, Sarah Hutchinson, and Adrian Bell, and to our marketing and documentation team: David Laur, Wendy Wirthlin, Dylan Sisson, David Hackett, Mark Manca, Leif Pedersen, and Carly Riley.

Also huge thanks to Pixar's studio tools illumination group: Christophe Hery, Ryusuke Villemin, Magnus Wrenninge, Florian Hecht, and others. They have contributed production-strength plug-ins (materials and integrators), as well as many suggestions and discussions along the way.

Many thanks to the directors of Pixar's movies for always pushing RenderMan, in particular, to John Lasseter for insisting on ray-traced reflections in the first *Cars* movie. Likewise, Jean-Claude Kalache, Bill Reeves, and Christophe Hery made a concerted effort toward more physical realism and global illumination on *The Blue Umbrella* and *Monsters University*. Also many thanks to Pixar's tech leadership, including Ed Catmull, Steve May, and Guido Quaroni.

Thanks to Matt Pharr for organizing this special issue of TOG and providing excellent feedback on early drafts of our paper. Thanks to Stéphane Grabli, Tony Apodaca, and Dan McCoy for suggestions to improve the accuracy and readability of our writing, and to Renee Tam for *Coco* statistics. Thanks to Disney, Lucasfilm, MPC, and others for allowing us to use their beautiful images.

And finally: our greatest thanks to our brilliant, dedicated, and talented users, who have suggested, tested, and pushed our rendering algorithms to their limits in pursuit of the perfect image.

REFERENCES

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

Abdalla Ahmed, Till Niese, Hui Huang, and Oliver Deussen. 2017. An adaptive point sampler on a regular lattice. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 36, 4, Article 138 (2017).

Anthony Apodaca and Larry Gritz. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

James Arvo and David Kirk. 1990. Particle transport and image synthesis. *Computer Graphics (Proceedings of SIGGRAPH)* 24, 4 (1990), 63–66.

Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony DeRose, and Fabrice Rousselle. 2017. Kernel-predicting convolutional networks for denoising Monte Carlo renderings. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 36, 4, Article 97 (2017).

Ronen Barzel. 1997. Lighting controls for computer cinematography. *Journal of Graphics Tools* 2, 1 (1997), 1–20.

Laurent Belcour, Ling-Qi Yan, Ravi Ramamoorthi, and Derek Nowrouzezahrai. 2015. *Antialiasing complex global illumination effects in path-space*. Technical Report 1375. University of Montreal.

Janet Berlin, Brent Burley, Lawrence Chai, Andrew Selle, Dan Teese, and Tom Thompson. 2011. SeExpr. (2011). www.disneyanimation.com/technology/seexpr.html.

Benedikt Bitterli, Wenzel Jakob, Jan Novák, and Wojciech Jarosz. 2018. Reversible jump Metropolis light transport using inverse mappings. *ACM Transactions on Graphics* 37, 1 (2018).

Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. 2005. A review of image denoising algorithms, with a new one. *Multiscale Modeling & Simulation* 4, 2 (2005), 490–530.

Michael Bunnell. 2005. Dynamic ambient occlusion and indirect lighting. In *GPU Gems 2*, Matt Pharr (Ed.). Addison-Wesley, 223–233.

Brent Burley. 2015. Extending the Disney BRDF to a BSDF with integrated subsurface scattering. In *'Physically Based Shading in Theory and Practice' SIGGRAPH Course*.

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

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

Chakravarty Chaitanya, Anton Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 36, 4, Article 98 (2017).

Matt Jen-Yuan Chiang, Peter Kutz, and Brent Burley. 2016. Practical and controllable subsurface scattering for production path tracing. In *SIGGRAPH Tech Talks*.

Per Christensen. 2008. *Point-based approximate color bleeding*. Technical Report 08-01. Pixar Animation Studios.

Per Christensen and Dana Batali. 2004. An irradiance atlas for global illumination in complex production scenes. *Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering)* (2004), 133–141.

Per Christensen and Brent Burley. 2015. *Approximate reflectance profiles for efficient subsurface scattering*. Technical Report 15-04. Pixar Animation Studios.

Per Christensen, Julian Fong, David Laur, and Dana Batali. 2006. Ray tracing for the movie 'Cars'. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*. 1–6.

Per Christensen, George Harker, Jonathan Shade, Brenden Schubert, and Dana Batali. 2012. Multiresolution radiosity caching for global illumination in movies. In *SIGGRAPH Tech Talks*.

Per Christensen, David Laur, Julian Fong, Wayne Wooten, and Dana Batali. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum (Proceedings of Eurographics)* 22, 3 (2003), 543–552.

Andrew Clinton and Mark Elendt. 2009. Rendering volumes with microvoxels. In *SIGGRAPH Tech Talks*.

Robert Cook. 2007. *3D paint baking proposal*. Technical Report 07-16. Pixar Animation Studios.

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

Robert Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed ray tracing. *Computer Graphics (Proceedings of SIGGRAPH)* 18, 3 (1984), 137–145.

R. Cranley and T. Patterson. 1976. Randomization of number theoretic methods for multiple integration. *SIAM Journal on Numerical Analysis* 13, 6 (1976), 904–914.

Ken Dahm and Alexander Keller. 2017. Learning light transport the reinforced way. In *SIGGRAPH Tech Talks*.

Mark Dippé and Erling Wold. 1985. Antialiasing through stochastic sampling. *Computer Graphics (Proceedings of SIGGRAPH)* 19, 3 (1985), 69–78.

David Dobkin, David Eppstein, and Don Mitchell. 1996. Computing the discrepancy with applications to supersampling patterns. *ACM Transactions on Graphics* 15, 4 (1996), 354–376.

Elmar Eisemann and Frédo Durand. 2004. Flash photography enhancement via intrinsic relighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 23, 3 (2004), 673–678.

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

Henri Fauré and Christiane Lemieux. 2009. Generalized Halton sequences in 2008: a comparative study. *ACM Transactions on Modeling and Computer Simulation* 19, 4, Article 15 (2009).

Matthew Fisher, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, William Mark, and Pat Hanrahan. 2009. DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 28, 5, Article 150 (2009).

Julian Fong, Ralf Habel, Magnus Wrenninge, and Christopher Kulla. 2017. Production Volume Rendering. In *SIGGRAPH Courses*.

Iliyan Georgiev and Marcos Fajardo. 2016. Blue-noise dithered sampling. In *SIGGRAPH Tech Talks*.

Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. 2012. Light transport simulation with vertex connection and merging. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 31, 6, Article 192 (2012).

Stéphane Grabli, Stephan Steinbach, and Mike King. 2012. Ratgather: how to turn RenderMan into a progressive path tracer. (2012). (Presented at the RenderMan User Group meeting).

Larry Gritz and James Hahn. 1996. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools* 1, 3 (1996), 29–47.

Larry Gritz, Clifford Stein, Chris Kulla, and Alejandro Conty. 2010. Open shading language. In *SIGGRAPH Tech Talks*.

Leonhard Grünschloß, Matthias Raab, and Alexander Keller. 2010. Enumerating quasi-Monte Carlo point sequences in elementary intervals. In *Proceedings of Monte Carlo and Quasi-Monte Carlo Methods*. 399–408.

Toshiya Hachisuka, Jacopo Pantaleoni, and Henrik Wann Jensen. 2012. A path space extension for robust light transport simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 31, 6, Article 191 (2012).

John Halton. 1964. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM* 7, 12 (1964), 701–702.

John Hammersley. 1960. Monte Carlo methods for solving multivariable problems. *Annals of the New York Academy of Sciences* 86 (1960), 844–874.

Pat Hanrahan and Jim Lawson. 1990. A language for shading and lighting calculations. *Computer Graphics (Proceedings of SIGGRAPH)* 24, 4 (1990), 289–298.

Paul Heckbert. 1990. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (Proceedings of SIGGRAPH)* 24, 4 (1990), 145–154.

Christophe Hery, Ryusuke Villemin, and Anton Kaplanyan. 2017a. Emeryville: where all the fun light transports happen. In *'Path Tracing in Production: Part 2' SIGGRAPH Course*.

Christophe Hery, Ryusuke Villemin, and Junyi Ling. 2017b. Pixar's foundation for materials. In *'Physically Based Shading' SIGGRAPH Course*.

Fred Hickernell. 2003. My dream quadrature rule. *Journal of Complexity (Oberwolfach special issue)* 19, 3 (2003), 420–427.

Homan Igehy. 1999. Tracing ray differentials. *Proceedings of SIGGRAPH* (1999), 179–186.

Thiago Ize. 2013. Robust BVH ray traversal. *Journal of Computer Graphics Techniques* 2, 2 (2013), 12–27.

Henrik Wann Jensen. 1995. Importance driven path tracing using the photon map. In *Rendering Techniques (Proceedings of the Eurographics Workshop on Rendering)*. 326–335.

Henrik Wann Jensen and Juan Buhler. 2002. A rapid hierarchical rendering technique for translucent materials. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 21, 3 (2002), 576–581.

Henrik Wann Jensen, Steve Marschner, Marc Levoy, and Pat Hanrahan. 2001. A practical model for subsurface light transport. *Proceedings of SIGGRAPH* (2001), 511–518.

Stephen Joe and Frances Kuo. 2008. Constructing Sobol' sequences with better two-dimensional projections. *SIAM Journal on Scientific Computation* 30, 5 (2008), 2635–2654.

Jim Kajiya. 1986. The rendering equation. *Computer Graphics (Proceedings of SIGGRAPH)* 20, 4 (1986), 143–150.

Timothy Kay and James Kajiya. 1986. Ray tracing complex scenes. *Computer Graphics (Proceedings of SIGGRAPH)* 20, 4 (1986), 269–278.

Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. 2002. A simple and robust mutation strategy for the Metropolis light transport algorithm. *Computer Graphics Forum (Proceedings of Eurographics)* 21, 3 (2002), 531–540.

Andrew Kensler. 2013. *Correlated multi-jittered sampling*. Technical Report 13-01. Pixar Animation Studios.

Alan King, Christopher Kulla, Alejandro Conty, and Marcos Fajardo. 2013. BSSRDF importance sampling. In *SIGGRAPH Tech Talks*.

Craig Kolb, Don Mitchell, and Pat Hanrahan. 1995. A realistic camera model for computer graphics. *Proceedings of SIGGRAPH* (1995), 317–324.

Thomas Kollig and Alexander Keller. 2002. Efficient multidimensional sampling. *Computer Graphics Forum (Proceedings of Eurographics)* 21, 3 (2002), 557–563.

Jaroslav Křivánek and Eugene d'Eon. 2014. A zero-variance-based sampling scheme for Monte Carlo subsurface scattering. In *SIGGRAPH Tech Talks*.

Jaroslav Křivánek, Pascal Gautron, Greg Ward, Henrik Wann Jensen, Eric Tabellion, and Per Christensen. 2008. Practical Global Illumination with Irradiance Caching. In *SIGGRAPH Courses*.

Jaroslav Křivánek, Iliyan Georgiev, Toshiya Hachisuka, Petr Vévoda, Martin Šik, Derek Nowrouzezahrai, and Wojciech Jarosz. 2014. Unifying points, beams, and paths in volumetric light transport simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 33, 4, Article 103 (2014).

Eric Lafortune and Yves Willems. 1993. Bi-directional path tracing. In *Proceedings of Compugraphics*. 145–153.

Eric Lafortune and Yves Willems. 1995. A 5D tree to reduce the variance of Monte Carlo ray tracing. *Rendering Techniques (Proceedings of the Eurographics Workshop on Rendering)* (1995), 11–20.

Hayden Landis. 2002. Production-ready global illumination. In *'RenderMan in Production' SIGGRAPH Course*. 87–102.

Gerhard Larcher and Friedrich Pillichshammer. 2001. Walsh series analysis of the L_2-discrepancy of symmetrisized point sets. *Monatshefte für Mathematik* 132 (April 2001), 1–18.

LollipopShaders. 2018. (2018). www.lollipopshaders.com.

Charles Loop. 1987. *Smooth Subdivision Surfaces Based on Triangles*. Master's thesis. University of Utah.

Marco Manzi, Fabrice Rousselle, Markus Kettunen, Jaakko Lehtinen, and Matthias Zwicker. 2014. Improved sampling for gradient-domain Metropolis light transport.

ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 33, 6, Article 178 (2014).

Stephen Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, and Pat Hanrahan. 2003. Light scattering from human hair fibers. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 22, 3 (2003), 780–791.

William Martin, Elaine Cohen, Russel Fish, and Peter Shirley. 2000. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools* 5, 1 (2000), 27–52.

Johannes Meng, Johannes Hanika, and Carsten Dachsbacher. 2016. Improving the Dwivedi sampling scheme. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering)* 35, 4 (2016), 37–44.

Don Mitchell. 1987. Generating antialiased images at low sampling densities. *Computer Graphics (Proceedings of SIGGRAPH)* 21, 4 (1987), 65–72.

Don Mitchell. 1991. Spectrally optimal sampling for distribution ray tracing. *Computer Graphics (Proceedings of SIGGRAPH)* 25, 4 (1991), 157–164.

Don Mitchell. 1992. Ray tracing and irregularities in distribution. *Proceedings of the Eurographics Workshop on Rendering* (1992), 61–69.

Don Mitchell. 1996. Consequences of stratified sampling in graphics. *Proceedings of SIGGRAPH* (1996), 277–280.

Thomas Müller, Markus Gross, and Jan Novák. 2017. Practical path guiding for efficient light-transport simulation. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering)* 36, 4 (2017), 91–100.

Ken Museth. 2013. VDB: high-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics* 32, 3, Article 27 (2013).

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

Jan Novák, Andrew Selle, and Wojciech Jarosz. 2014. Residual ratio tracking for estimating attenuation in participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 33, 6, Article 179 (2014).

Victor Ostromoukhov, Charles Donohue, and Pierre-Marc Jodoin. 2004. Fast hierarchical importance sampling with blue noise properties. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 23, 3 (2004), 488–495.

Art Owen. 1997. Monte Carlo variance of scrambled net quadrature. *SIAM Journal on Numerical Analysis* 34 (1997), 1884–1910.

Jacopo Pantaleoni. 2017. Charted Metropolis light transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 36, 4, Article 75 (2017).

Darwyn Peachey. 1990. *Texture on demand.* Technical Report 217. Pixar Animation Studios.

Leonid Pekelis, Christophe Hery, Ryusuke Villemin, and Junyi Ling. 2015. *A data-driven light scattering model for hair.* Technical Report 15-02. Pixar Animation Studios.

Fabio Pellacini, Kiril Vidimče, Aaron Lefohn, Alex Mohr, Mark Leone, and John Warren. 2005. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 24, 3 (2005), 464–470.

Georg Petschnigg, Richard Szeliski, Maneesh Agrawala, Michael Cohen, Hugues Hoppe, and Kentaro Toyama. 2004. Flash photography enhancement via intrinsic relighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 23, 3 (2004), 664–672.

Matt Phárr, Wenzel Jakob, and Greg Humphreys. 2017. *Physically Based Rendering: From Theory to Implementation* (3rd ed.). Morgan Kaufmann.

Les Piegl and Wayne Tiller. 1997. *The NURBS Book.* Springer-Verlag.

Matthias Raab, Daniel Seibert, and Alexander Keller. 2006. Unbiased global illumination with participating media. In *Proceedings of Monte Carlo and Quasi-Monte Carlo Methods.* 591–606.

Jonathan Ragan-Kelley, Charlie Kilpatrick, Brian Smith, Doug Epps, Paul Green, Christophe Hery, and Frédo Durand. 2007. The Lightspeed automatic interactive lighting preview system. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 26, 3, Article 25 (2007).

William Reeves, David Salesin, and Robert Cook. 1987. Rendering antialiased shadows with depth maps. *Computer Graphics (Proceedings of SIGGRAPH)* 21, 4 (1987), 283–291.

Bernhard Reinert, Tobias Ritschel, Hans-Peter Seidel, and Iliyan Georgiev. 2015. Projective blue-noise sampling. *Computer Graphics Forum* 35, 1 (2015), 285–295.

Fabrice Rousselle, Marco Manzi, and Matthias Zwicker. 2013. Robust denoising using feature and color information. *Computer Graphics Forum (Proceedings of Pacific Graphics)* 32, 7 (2013), 121–130.

Thorsten-Walther Schmidt, Fabio Pellacini, Derek Nowrouzezahrai, Wojciech Jarosz, and Carsten Dachsbacher. 2016. State of the art in artistic editing of appearance, lighting, and material. *Computer Graphics Forum (Proceedings of Eurographics)* 35, 1 (2016), 216–233.

Peter Shirley. 1991. Discrepancy as a quality measure for sample distributions. *Proceedings of Eurographics* (1991), 183–193.

Peter Shirley, Changyaw Wang, and Kurt Zimmerman. 1996. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics* 15, 1 (1996), 1–36.

Ilya Sobol'. 1967. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics* 7, 4 (1967), 86–112.

Eric Tabellion and Arnauld Lamorlette. 2004. An approximate global illumination system for computer generated films. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 23, 3 (2004), 469–476.

Carlo Tomasi and Roberto Manduchi. 1998. Bilateral filtering for gray and color images. In *Proceedings of the International Conference on Computer Vision.* 839–846.

Steve Upstill. 1990. *The RenderMan Companion.* Addison Wesley.

Eric Veach and Leonidas Guibas. 1994. Bidirectional estimators for light transport. In *Proceedings of the Eurographics Workshop on Rendering.* 147–162.

Eric Veach and Leonidas Guibas. 1995. Optimally combining sampling techniques for Monte Carlo rendering. *Proceedings of SIGGRAPH* (1995), 419–428.

Eric Veach and Leonidas Guibas. 1997. Metropolis light transport. *Proceedings of SIGGRAPH* (1997), 65–76.

Jiři Vorba, Ondřej Karlik, Martin Šik, Tobias Ritschel, and Jaroslav Křivánek. 2014. On-line learning of parameteric mixture models for light transport simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 33, 4, Article 101 (2014).

Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of Eurographics)* 20, 3 (2001), 153–164.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory Johnson, and Manfred Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 33, 4, Article 143 (2014).

Gregory Ward. 1991. Adaptive shadow testing for ray tracing. In *Proceedings of the Eurographics Workshop on Rendering.* 11–20.

Gregory Ward and Paul Heckbert. 1992. Irradiance gradients. In *Proceedings of the Eurographics Workshop on Rendering.* 85–98.

Turner Whitted. 1980. An improved illumination model for shaded display. *Communications of the ACM* 23, 6 (1980), 343–349.

Alexander Wilkie, Sehera Nawaz, Marc Droske, Andrea Weidlich, and Johannes Hanika. 2014. Hero wavelength spectral sampling. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering)* 33, 4 (2014), 123–131.

Lance Williams. 1978. Casting curved shadows on curved surfaces. *Computer Graphics (Proceedings of SIGGRAPH)* 12, 3 (1978), 270–274.

Lance Williams. 1983. Pyramidal parametrics. *Computer Graphics (Proceedings of SIGGRAPH)* 17, 3 (1983), 1–11.

Andrew Woo, Andrew Pearce, and Marc Ouellette. 1996. It's not really a rendering bug, you see ... *IEEE Computer Graphics and Applications* 16, 5 (1996), 21–25.

E.R. Woodcock, T. Murphy, P. Hemmings, and T. Longworth. 1965. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Proceedings of Applications of Computing Methods to Reactor Problems.* Argonne National Laboratory, 557–579.

Sven Woop, Carsten Benthin, Ingo Wald, Gregory Johnson, and Eric Tabellion. 2014. Exploiting local orientation similarity for efficient ray traversal of hair and fur. In *Proceedings of High Performance Graphics.* 41–49.

Magnus Wrenninge. 2016. Efficient rendering of volumetric motion blur using temporally unstructured volumes. *Journal of Computer Graphics Techniques* 5, 1 (2016), 1–34.

Magnus Wrenninge, Ryusuke Villemin, and Christophe Hery. 2017. *Path traced subsurface scattering using anisotropic phase functions and non-exponential free flights.* Technical Report 17-07. Pixar Animation Studios.

Sergei Zhukov, Andrei Iones, and Gregorij Kronin. 1998. An ambient light illumination model. In *Rendering Techniques (Proceedings of the Eurographics Workshop on Rendering).* 45–55.

Henning Zimmer, Fabrice Rousselle, Wenzel Jakob, Oliver Wang, David Adler, Wojciech Jarosz, Olga Sorkine-Hornung, and Alexander Sorkine-Hornung. 2015. Path-space motion estimation and decomposition for robust animation filtering. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering)* 34, 4 (2015), 131–142.

Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and Sung-Eui Yoon. 2015. Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. *Computer Graphics Forum (Proceedings of Eurographics)* 34, 2 (2015), 667–681.