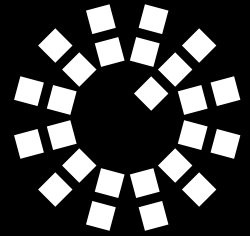


SONY PICTURES

imageworks

25TH ANNIVERSARY

Arnold at Sony Pictures Imageworks

From Monster House to Smurfs: the Lost Village

Christopher Kulla

SIGGRAPH 2017

Thank you for the introduction.

My name is Chris Kulla and I will be talking about our use of the Arnold renderer at Sony Imageworks.

The talk was titled “Monster House to Smurfs: The Lost Village” in the course notes, but actually ...

Arnold at Sony Pictures Imageworks

From Monster House to The Emoji Movie

Christopher Kulla

SIGGRAPH 2017

...we've released one more animated film since then. The Emoji Movie came out last weekend.

I'd like to first clarify why there is a second Arnold talk in this course.

Like Marcos mentioned this morning, Imageworks was one of the early adopters of Arnold and we also participated in its development because of a source code licensing agreement.

But since about 2009 the renderer was forked and the two copies have actually been evolving independently. So I'll be talking about why that is and what we've been doing in our version.

Monster House (2004-2006)

Arnold introduced as an experiment:

- Global Illumination
- Ray Traced Shadows
- Noise as the only artifact

But let's start from the begining. Arnold was brought into our facility back in 2004, specifically for the film Monster House. It really was a bit of an experiment. We were attracted by the types of images the renderer could produce and they lined up really well with the creative goals for the movie.

Global illumination and ray traced soft shadows really helped ground the characters in the environment and gave them a clay-like tangible feel.

Another benefit was that the renderer had very few controls and just one kind of artifact. That simplicity of path tracing is obviously something we've heard a lot about in this course.

Monster House (2004-2006)

Arnold introduced as an experiment:

- Global Illumination
- Ray Traced Shadows
- Noise as the only artifact

Compromises:

- Motion blur disabled
- No hair primitive
- Less displacement
- Noise (mimics film grain?)

But adopting this technology so early wasn't without compromises. Luckily all these things happened to fit the look of the movie. Motion blur was slow because of the acceleration structures we used, so we just turned it off. There was no hair primitive, so the characters had what we call "helmet hair". But luckily the film was going for a stop-motion look, so the film-makers just embraced those limitations.

We also didn't do as much displacement as we wanted, which was mostly because of memory limits.

And we also found out noise isn't so easy to get rid of! But we kind of pretended that it looked a bit like film grain.

Monster House Clip

Just to illustrate what I said, let me play a quick clip from the start of the film to remind you how it looked.

Monster House was released in 2006 and to our knowledge it was the first animated feature film to be rendered using path tracing. Even though this film was released more than 10 years ago, you can still see some of the hallmarks of global illumination. There is basically a single light acting as the sun in this shot, and yet we have very rich bounce lighting. With shadow maps, this kind of long traveling shot would have taken lots of manual effort to get right.

Of course some of the limitations we had to deal with are visible too.

Monster House (2004-2006)

Despite those compromises, we proved the renderer had potential!

- Eliminated pass management
- More consistent results from shot to shot
- Memory use was steadily going down

But despite these compromises we made, this production really showed us the potential for path tracing. Again, lots of this has been covered by the other speakers, but:

We got rid of all kinds of extra passes like shadow maps, reflection maps, occlusion maps, and so on.

Bounced lighting did a lot for us, so we could place fewer lights and fewer shot specific lights which helped with consistency between artists.

And even though memory usage was an issue, the general trend was pointed downwards, which was encouraging.

Cloudy With A Chance Of Meatballs (2007-2009)

Production team had just finished Surf's Up:

- Shadow map management was not scaling
- Hard to ensure consistency between artists
- Point clouds for GI would introduce one more set of dependencies
- Arnold was comparing very favorably to ray tracing in PRman

So the next movie to evaluate Arnold was Cloudy with a Chance of Meatballs, which started production around 2007.

The production team for this film had just wrapped up the movie Surf's Up - which was a great looking film, but had faced all of the issues I just mentioned. They had done some testing the new pointcloud GI approaches that were starting to appear but felt like it would be moving us in the wrong direction.

But actually the biggest reason Arnold was appealing was by how much it was outperforming the version of Prman we were using at the time. And because we had source code access to Arnold, we knew we could push it even further.

So we decided to try Arnold again...

Cloudy With A Chance Of Meatballs (2007-2009)

Address the compromises!

- Motion Blur
- Hair
- Memory Usage

…but there was still a bit of hesitation. For this movie, we didn't feel like we could get away with the same compromises as Monster House.

The biggest things to address were: motion blur, hair and of course memory usage.

This isn't even a complete list. We actually revisited lots of other features like subsurface scattering, the output driver system, the texture system and lots of other small things …

But for today I'll touch on these three topics.

Cloudy With A Chance Of Meatballs (2007-2009)

Acceleration Structure Rewrite:

- Bounding Volume Hierarchies
- Predictable memory usage
- Easy to extend to motion blur (with predictable performance)

Up until this point, Arnold had been using uniform grids. The implementation actually had some cool low level tricks, but didn't perform well when primitives were motion blurred. And in some cases even non-motion blurred geometry could be slow or use a lot of memory.

So we implemented what is now basically the industry standard: the bounding volume hierarchy. The big advantage of this structure is that the memory usage is very predictable and its really easy to extend to motion blur with predictable performance.

Cloudy With A Chance Of Meatballs (2007-2009)

Acceleration Structure Rewrite:

- Bounding Volume Hierarchies
- Predictable memory usage
- Easy to extend to motion blur (with predictable performance)

Our implementation:

- Shared for all primitives (C++ templates)
- No spatial splits (primitives expected to be small)

These days you can get a very high quality implementation of this in the Embree library from Intel.

Our implementation was done earlier, but has similar characteristics. For instance we used C++ templates to get code that's optimized per primitive without having to duplicate any code. We just have one BVH builder and two traversal kernels (for motion blur on or off).

Another thing worth noting is that we didn't bother implementing spatial splits. It can help if you have very large triangles but most of our scenes have tiny triangles.

Memory Usage

Details matter!

- Before: 4.5Gb
- After: 900Mb

Not just geometry:

- Meshes: 70Mb
- Accels: 100Mb
- Strings: 350Mb

Moving to BVHs definitely helped memory usage, but lots of other details matter. This city scene at one point was over 4Gb, whereas now it renders in less than 1Gb.

What I want to stress here, is that none of this comes from geometric complexity. The scene uses instancing so the amount of mesh data is tiny. On the other hand, the scene references lots of textures. And our lighting tool likes to give objects really long names.

So we actually have 5 times more string data than mesh data. And that's after implementing string de-duplication. So we got lots of huge memory improvements by doing these kinds of optimizations and really focusing on the details throughout the system.

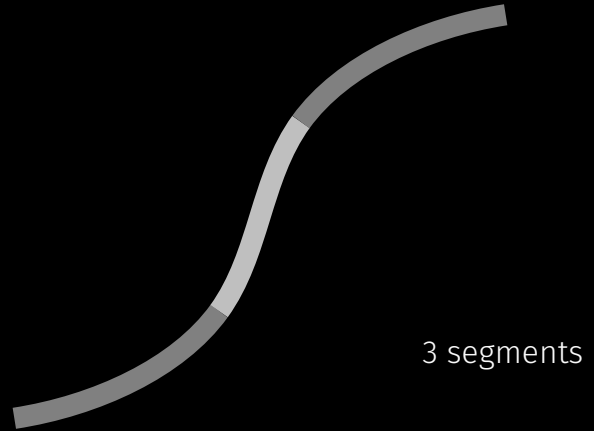
Hair Rendering

Next I'd like to talk about how we implemented support for hair.

We had this furry character in the movie, but also used hair strands for all the other characters.

Curve Primitive

Use cubic segments to reduce memory usage

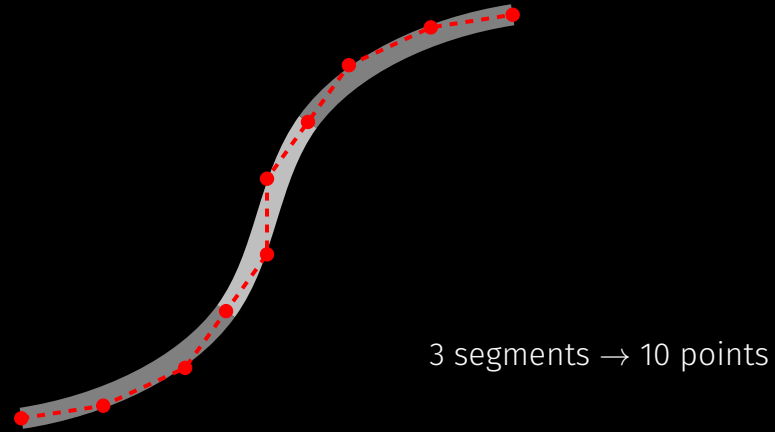


3 segments

We chose to represent hairs as cubic curves to reduce memory usage and avoid the need for tessellation. During Monster House we had some prototypes that use linear curves for grass and when we tried this on hair we saw it wasn't going to scale well.

Curve Primitive

Bezier curves: $3n + 1$ points for n segments (redundant representation!)

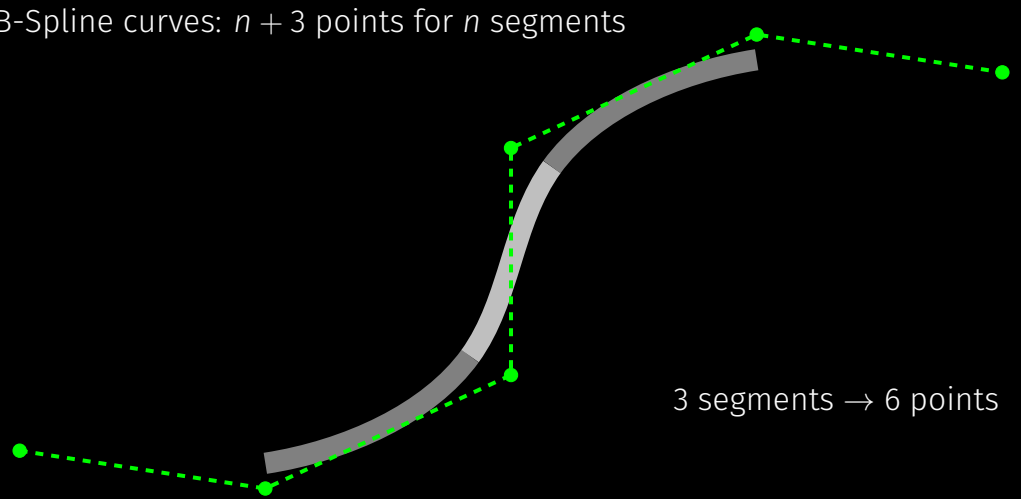


3 segments $\rightarrow$ 10 points

But the type of cubic curve also matters. Here I've drawn the bezier control points for this curve. You can see that for n segments I would need roughly 3 times as many points.

And those control points between segments need to be aligned just right for the whole curve to be smooth, so these extra points are really just encoding redundant information.

Curve Primitive



On the other hand, a b-spline representation just needs $O(n)$ points to define the exact same curve. There's also no redundancy at all when describing continuous curves. In fact, because points get re-used from segment to segment, they're more likely to stay in the CPU cache.

Our hair generation system at Imageworks already made b-splines by default. So this was a natural choice for us. But later on we actually got rid of all support other basis types and optimized the code assuming b-splines since thats all we ever use.

Curve Primitive

Improving performance:

- Compute tight AABB (Bezier cage is not minimal)

That covers efficient storage, but we did lots of other small things to speed up ray intersections.

First we make sure to compute the tightest possible bounding box for each segment. Using bezier control points is better than using the b-spline points like you probably saw in the previous slides but you can do even better by solving for the exact fit.

Curve Primitive

Improving performance:

- Compute tight AABB (Bezier cage is not minimal)
- Add oriented bounds in BVH leaves to help with “diagonal” curves

Even with a good bounding box though, hair segments can often end up oriented diagonally. In this case the performance of the BVH starts to suffer because lots of nodes overlap and hitting the bounding box is a bad predictor of actually hitting the segment itself.

So we calculate oriented bounds per segment as well.

But to keep the code simple, we just put them in the leaves of the tree. This is way easier to implement than oriented BVHs and also keeps our BVH code *shape agnostic* (both the traversal and the construction).

Curve Primitive

Improving performance:

- Compute tight AABB (Bezier cage is not minimal)
- Add oriented bounds in BVH leaves to help with “diagonal” curves
- Build shallow BVH and use SIMD to intersect several segments

We still have to deal with the bounding box overlap problem, but we mitigate this by just building shallower trees. Because our leaf level bounds are really cheap to intersect, we can pack several of them together and use SIMD instructions to test several at once against a ray.

This also has the side benefit of reducing memory usage even further.

Curve Primitive

Improving performance:

- Compute tight AABB (Bezier cage is not minimal)
- Add oriented bounds in BVH leaves to help with “diagonal” curves
- Build shallow BVH and use SIMD to intersect several segments
- Wider curves with transparency for better anti-aliasing

The last important optimization we did is not really related to the intersection test, but we nudge the width of the curve to be at least half a pixel wide. That makes the hairs easier to anti-alias - especially the tips.

These days the target width is closer to 10% of a pixel because we fire a lot more rays and limiting the amount of transparency is helpful. In fact, this particular trick might be obsolete soon.

Alice In Wonderland (2008-2010)





Hair rendering really got optimized during Alice in Wonderland because it had even more furry characters. For example the move to using SSE instructions happened for this movie.

I'll also point out that this was the first really big VFX project we used Arnold on. We had done a few smaller shows before, but this one had over a 1000 shots and had lots of all CG environments and characters.

Once we made it past this production - the renderer was pretty solid and our studio hasn't looked back since.

Where to go from here?

What challenges were left to address?

- Shading Architecture
- Reducing Variance
- Geometry Complexity

But of course we weren't done yet either. The core of the renderer was fairly mature at this point, and it could handle lots of data, but we wanted to push it further.

I've broken down the rest of the talk into three basic topics:

The Shading Architecture

Techniques for reducing Variance

And Geometric Complexity

Shading Architecture

I’ll start with the shading architecture, since this is probably the biggest change we did

Shading Architecture

Shaders written in C++:

- Texturing / Pattern Generation
- BxDFs
- Light Transport

Up until this point we were writing all of our shaders in C++. This included not just texturing and pattern generation but also the specific details of how materials react to light.

This means the shaders were responsible for looping over lights, firing secondary rays and so on.

Basically, the path tracing algorithm wasn't really a part of the renderer at all. It just so happened we had written our shaders that way.

Maybe this sounds a bit strange to some of you. Even the textbook on physically based rendering PBRT has the concept of an integrator outside the materials. But the Arnold API had evolved sort of by analogy to the Renderman shading language and inherited this limitation.

Shading Architecture

Shaders written in C++:

- Texturing / Pattern Generation
- BxDFs
- Light Transport

Roadblock to improvements:

- Difficult to change integration strategies
- No ability to batch or re-order rays
- Painful to correctly track derivatives

So of course this structure made is really hard to improve the renderer.

It was really hard to change integration strategies, because we had redundant logic across lots of different shaders. Something as simple as russian roulette couldn't be easily added because individual rays weren't even aware of their overall weight.

We couldn't think about any kind of batching or re-ordering of rays because they were being traced from inside the materials and had to return values right away.

Even just correctly tracking derivatives for proper texture filtering was something we had to do manually and that was hard to get right in all cases.

Shading Architecture

Shaders written in C++:

- Texturing / Pattern Generation
- BxDFs
- Light Transport

Roadblock to improvements:

- Difficult to change integration strategies
- No ability to batch or re-order rays
- Painful to correctly track derivatives

We wanted a smoother experience for shader writers *and* a more decoupled architecture for future improvements.

So we knew we wanted to do something. We wanted shader writers to worry less about technical details and we also wanted a decoupled architecture to able to make bigger changes.

Open Shading Language

Design a domain specific language!

- Decouple shading from integration
  - Remove lighting calculations
  - Shaders return “closures”
- Track derivatives automatically
- Composable through shading networks
- Open Source!

That’s where Open Shading Language comes in. Its our domain specific language to write shaders. But unlike previous shading languages, we made sure not to put anything into the language related to integration. Shaders just return “closures” which are usually BSDFs but also cover things like BSSRDFs, emission, or even holdouts.

We designed the runtime so it could track derivatives automatically. It can also compose individual shaders into larger networks and manages the execution to avoid redundant evaluations.

We also decided from the start the project would open source! A few other studios we talked to mentioned they were interested, so we decided to develop the project in the open.

Open Shading Language

- First runtime was an interpreter
- Packet tracing to assemble batches
- Half of the renderer had to be rewritten
- All our shaders rewritten in OSL

The first OSL runtime was actually an interpreter. The plan was to simultaneously switch to tracing rays in packets and assemble batches of points to feed the interpreter.

We had an implementation of packet tracing ready to use, but to really integrate OSL we had to rewrite at least half of the renderer. And of course rewrite all our shaders in OSL.

So this was a big project, but it was also really exciting because we were finally changing things we had been stuck on for a long time.

Forking the renderer (around 2009)

Solid Angle was being established to commercialize the renderer.

- Rewrite was a risky architecture change
- We forked the code base!
- Retained IP sharing agreements
- Our push to OSL made codebases diverge quickly

Smart decision for Solid Angle because …

Now around the same time we were starting this big integration of OSL into the renderer, Marcos who you heard from this morning, was establishing Solid Angle to commercialize the renderer more broadly. This was part of his agreement with Imageworks all along. We had a license of the code but he was still able to sell it to other clients.

But right as he was looking to stabilize things and staff up his team to work on third party applications…And we were talking about this huge rewrite.

So this is when we actually decided to fork the code base. We kept in place all our IP sharing agreements, but the renderers started to diverge almost immediately.

And in retrospect, this was probably a smart decision for Solid Angle because…

Renderer got much slower! (2x at best)

- Interpreter overhead for small batches
- Packetized integrator was very complex
- Undid all benefits of tracing ray bundles

...this big rewrite wound up making the renderer way slower. We worked on this for a long time, but the fastest it ever got was roughly half the speed of what we started from.

This was due to a number of factors. First the interpreter was pretty good at executing large batches, but had lots of overhead when the batches were small. Then the integrator code to maintain packets through light loops and indirect rays was really complicated and spent a lot of time just shuffling data around. So both of these things ended up undoing the benefits of packet tracing.

Now we heard today from other projects that have been much more successful at vectorization, so I don't want to say it was a bad architecture - it was really more about our specific implementation choices.

Renderer got much slower! (2x at best)

- Interpreter overhead for small batches
- Packetized integrator was very complex
- Undid all benefits of tracing ray bundles

Rewrote the renderer *again*!

- Reverted to single ray
- LLVM based runtime
- Renderer finally *faster*! (+30%)
- Shaders stayed the same!

So we rewrote everything again! We went back to a simpler single ray system with an integrator that closer to how the shaders had been written.

We also replaced the interpreter with LLVM to generate native code. And all this finally pushed us to the point where the renderer was finally faster! Only by about 30%, but that's before we started to tackle all the changes we wanted to make to the rest of the code.

And the great thing was, the shaders didn't have to change at all. This was already a big validation of our design. We were making major changes to the renderer without changing anything in the shaders.

Reducing Variance

So after this big rewrite of how shaders were executed, we could finally get back to improving the integrator.

Now in reality some of these things happened simultaneously, but conceptually OSL was the enabling factor.

Advancing the Integrator

OSL allowed us to tackle many improvements to the integrator:

- Multiple Importance Sampling applied everywhere
- Russian roulette for deeper bounces
- Integrated volume shading*
- Ray-traced SSS
- Many variance reduction tricks (clamping, blurring, …)

Shaders did not need to change at all!
Improvements could be done mid-production with little risk.

*See “Production Volume Rendering” course

First we were able to implement multiple importance sampling a lot more consistently. Our C++ shaders did MIS too but the implementation was a bit clunky because it had to use callbacks between the shader and the renderer. With everything unified the code was cleaner and we could do things like MIS between BSDF lobes (our shaders tend to have lots of lobes).

We also could finally add Russian roulette to be able to trace to deeper bounces.

We did bigger projects also like integrating volume shading. Hopefully you were able to catch the course on Sunday where I discussed the details of that system.

Advancing the Integrator

OSL allowed us to tackle many improvements to the integrator:

- Multiple Importance Sampling applied everywhere
- Russian roulette for deeper bounces
- Integrated volume shading
- Ray-traced SSS*
- Many variance reduction tricks (clamping, blurring, …)

Shaders did not need to change at all!
Improvements could be done mid-production with little risk.

*See “Physically Based Shading” course

We also switched from a pointcloud based implementation of subsurface to a ray traced approach. We presented that work at Siggraph 2013. And since then we’ve gone further and just unified subsurface with volume rendering. I talked about some of this in the Physically Based Shading course from Sunday.

And of course there are lots of other little details that help reduce variance like clamping some values or blurring roughness values based on the ones seen so far. We used these tricks before, but by putting them in the integrator the implementation was a lot more consistent.

And again, through all these improvements - none of the shaders ever had to change. This meant we could do some of these changes in the middle of production.

Research on Advanced Integrators

- VCM/UPS (and subsets like BDPT)
- MCMC methods
- Combination of both is very robust!
- Very useful for reference solutions
- Used occasionally for caustic passes

Path tracing (+tricks) still hard to beat for production cases …

We also started doing some research on fancier algorithms. We implemented papers like the VCM or UPS algorithm and the various subsets it covers. We also played with markov chain methods that help explore difficult cases.

Actually that combination of metropolis and VCM is really robust. Of our fancy integrators - it can probably handle the most cases. And it means we can compare the path tracer tricks to the right ground truth solution.

On the other hand, path tracing with tricks is still hard to beat. As we heard from other presenters today - bidirectional methods have a lot of overhead and it doesn't always pay off for the types of scenes we have.

A typical frame contains many *locally* difficult problems

Just to illustrate what I mean, here is a frame from The Emoji Movie. It has lots of things going on of varying difficulty.

For example nearly every character has volumetric subsurface, we have some that are motion blurred, there's some blurry refractions in the table…

…but lots of the image is relatively *easy* to render too.

Because we are using some tricks to simplify the *really* hard problems like caustics, we don't have any major fireflies. And yet some areas still take longer to converge than others.

Uniform 64 paths/pixel

So let me zoom into some small areas here to show the type of noise we are getting with plain path tracing. A uniform budget of rays actually does reasonably well, but the amount of noise isn't quite constant. Some area like the floor look pretty good, but the subsurface on the characters or some of the shadowed regions are noisier.

Sorry these slides are probably only going to make sense if you are sitting in the front...

Image Space Techniques - Adaptive Sampling

Adaptive 36 to 256 paths/pixel

A very simple and yet very effective solution is to target samples adaptively in image space. That lets us spend extra time only where its needed and get a more uniform level of noise across the image.

Image Space Techniques - Denoising

Adaptive + Denoise

And in fact - because adaptive helps equalize the noise level across the frame, it also makes it easier to denoise.

Disney really pioneered the use of these techniques in production, and we've really come to embrace these methods too because they're very effective in removing that last bit of noise.

And because modern denoising methods are guided by feature images from the render, they can retain detail much better. Now unless you are sitting in the front, this probably is going to be hard to see, but be sure to look at the slides online later on.

Image Space Techniques - Details

Adaptive Sampling:

- Minimum sampling level to set baseline variance
- Variance of samples $\neq$ Variance of the mean
- Measure error in tone-mapped space for better perceptual behavior
- Dilate error in 3×3 pixels to explore small details

Just a few details about these techniques. Adaptive sampling starts by taking a fixed number of samples to get a baseline of variance.

When I say variance of course I mean the variance of the mean of the samples, not the variance of the samples themselves. This is really important. The variance of the samples doesn't mean much because the samples come from some unknown distribution. On the other hand, the mean will have a normal distribution by the central limit theorem.

Then that variance gets turned into an error measure by going through tone mapping so that we don't over-sample highlights or under sample darker regions.

We also dilate the error in 3 by 3 blocks to avoid missing any small details.

Image Space Techniques - Details

Adaptive Sampling:

- Minimum sampling level to set baseline variance
- Variance of samples $\neq$ Variance of the mean
- Measure error in tone-mapped space for better perceptual behavior
- Dilate error in 3×3 pixels to explore small details

Image space denoise:

- Blur $\rightarrow$ 0 as Variance $\rightarrow$ 0 to ensure consistency
- Minimize number of feature inputs (just color and normals)
- Implement as Nuke plugin for more flexibility
- Multi-threading + SIMD: 16 seconds/frame

For image space denoising - the most important property is that the amount of blur should go to 0 as the variance goes to 0. This makes it a consistent technique because converged pixels won't be modified at all.

We've also tried to minimize the number of feature inputs we use. Our denoiser just uses color and normals. If we had to split diffuse and specular it would mean splitting per light outputs as well.

We tend to output lots of images from each render, so we didn't want denoising to grow this number too much. In fact we let artists denoise in Nuke so that we don't process any images they end up not using.

The denoising is reasonably fast. Our current algorithm is roughly 16 seconds for 2k frame...

Image Space Techniques - Details

Adaptive Sampling:

- Minimum sampling level to set baseline variance
- Variance of samples $\neq$ Variance of the mean
- Measure error in tone-mapped space for better perceptual behavior
- Dilate error in 3×3 pixels to explore small details

Image space denoise:

- Blur $\rightarrow$ 0 as Variance $\rightarrow$ 0 to ensure consistency
- Minimize number of feature inputs (just color and normals)
- Implement as Nuke plugin for more flexibility
- Multi-threading + SIMD: 86 seconds/frame (5 frame temporal denoise)

...and if we do the temporal version that blurs across 5 frames it goes up to just over a minute. So just slightly over 5x slower. That's because the algorithm is very cache friendly and you get some non-linear behavior when you fall out of cache.

We can probably get this down even further, maybe with a GPU version. This is still a hot topic for research and lots of interesting new papers are still coming out on this.

Geometric Complexity

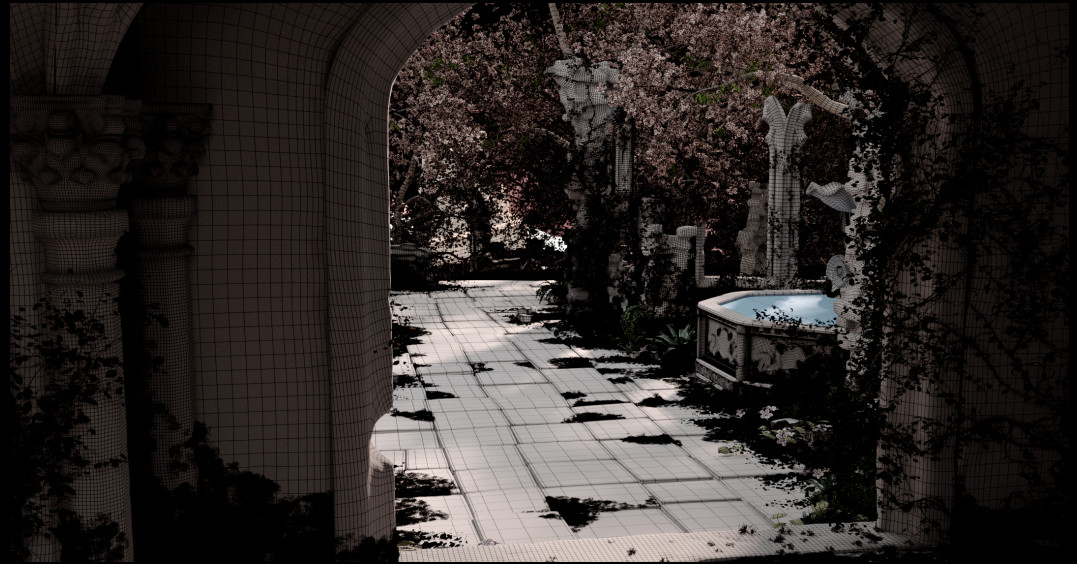
Finally I just want to talk about geometric complexity a bit.

An average scene: 56M subdiv patches, 86M polygons, 8M instances

Here's an environment I took from our performance regression suite. Its fairly average in terms of complexity.

The first thing to point out is that we have lots of subdivision surfaces. In fact this scene is a bit of an outlier because it also plain polygons. You can't see the entire environment here, there's also a large portion thats outside the frame.

Before tessellation: 260M unique triangles, 35.8B instanced triangles, 11Gb



Here is a wireframe *before* we do any tessellation. We already have over 200M triangles, a lot more if you count the instancing.

The memory footprint, including all the acceleration structures is about 11Gb.

After tessellation: 566M unique triangles, 39.5B instanced triangles, 21Gb

Now turning on tessellation for those subdiv patches raises that total to about half a billion unique triangles and memory went up by slightly less than half - even though we've tessellated everything down to sub-pixel size.

Subdivision Surfaces

- Base meshes typically very dense
- Target ½ pixel edge length or distance to limit surface
- Optimize level 0 limit-surface projection for off-screen / distant cases

The most important observation we made is that our base meshes are always very dense. So in reality, subdividing even to a half pixel target doesn't require too much work.

That's one reason we've focused on upfront tessellation as opposed to a dynamic solution. If we wanted to subdivide on the fly we'd have to maintain more information than the final tessellated copy takes.

In fact, because our base meshes are so dense, the case we've optimized the most is the level 0 case where we just do limit surface projection without adding any new vertices (even in the irregular regions).

Subdivision Surfaces - Storage

- Decompose meshes into patches, evaluate and store as patches
- Store shared edges/vertices *once* (helps at low tessellation levels)
- Topology requires very little memory
- Explicit connectivity only for stitching triangles

When we do need to subdivide, we take a patch based approach. We set edge rates based on the camera and then turn each patch into a small grid. But in many cases the tessellation needed is not that high (like 2x2 or 4x4). If we just stored patches as independent grids we'd be wasting memory on the edges.

So our data structures actually share edges and corners of patches and only stores them once. For the inside of the grids the topology is implicit, so we just have explicit topology for the stitching triangles between patches.

Subdivision Surfaces - Instancing

Most large environments are heavily instanced:

- Adaptive tessellation still important
- Dice for worst case (measured in screen space)
- Estimate required edge rates per patch, per instance
- Use early-outs to speed up common cases

I said we set edge rates based on the camera, but we also use a lot of instancing. So we've had to make adaptive tessellation work in that case as well. The basic idea is just measure the edge rate on each patch under the transform of each instance.

But this isn't going to work because we have millions of patches getting instanced millions of times. It turns out we can make this practical with a few simple heuristics...

Subdivision Surfaces - Instancing

Most large environments are heavily instanced:

- Adaptive tessellation still important
- Dice for worst case (measured in screen space)
- Estimate required edge rates per patch, per instance
- Use early-outs to speed up common cases
  - Skip fully off-screen

We can just skip any instances that fully off-screen since they don't need any subdivision

Subdivision Surfaces - Instancing

Most large environments are heavily instanced:

- Adaptive tessellation still important
- Dice for worst case (measured in screen space)
- Estimate required edge rates per patch, per instance
- Use early-outs to speed up common cases
  - Skip fully off-screen
  - Skip fully on-screen (distance to bbox)

For any instance that is fully on-screen we can do a conservative estimate using the distance to the bounding box

Subdivision Surfaces - Instancing

Most large environments are heavily instanced:

- Adaptive tessellation still important
- Dice for worst case (measured in screen space)
- Estimate required edge rates per patch, per instance
- Use early-outs to speed up common cases
  - Skip fully off-screen
  - Skip fully on-screen (distance to bbox)
  - Loop only when bbox crosses frustum

So we just need to do the brute force approach for those objects that cross the frustum. And this is usually a small fraction of the scene - so it hasn't been a bottleneck.

Since we've put these heuristics in place, the artists really haven't had to think about tessellation at all, and our artists are finally able to use displacement mapping as much as they want.

Subdivision Surfaces - Multi-Threading

- Tessellate and build acceleration structures before rendering
- Exact bounds for displaced surfaces without user intervention
- All instances known during tessellation
- Allows perfect threading (object parallel → patch parallel)

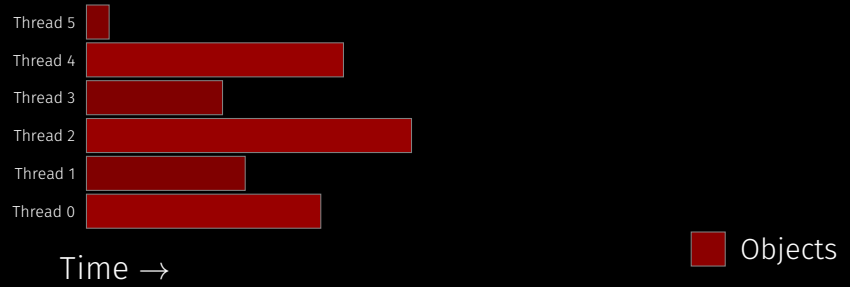
The last aspect I want to mention is multi-threading.

We do all the tessellation and acceleration structure building before tracing any rays. That means we always get correct bounds for displaced surfaces and we know about all the instances which lets us do adaptive tessellation like I just described.

It also means threading can be “perfect” in the sense that we can transition from object parallelism to patch parallelism when we run out of objects.

Subdivision Surfaces - Multi-Threading

- Tessellate and build acceleration structures before rendering
- Exact bounds for displaced surfaces without user intervention
- All instances known during tessellation
- Allows perfect threading (object parallel → patch parallel)

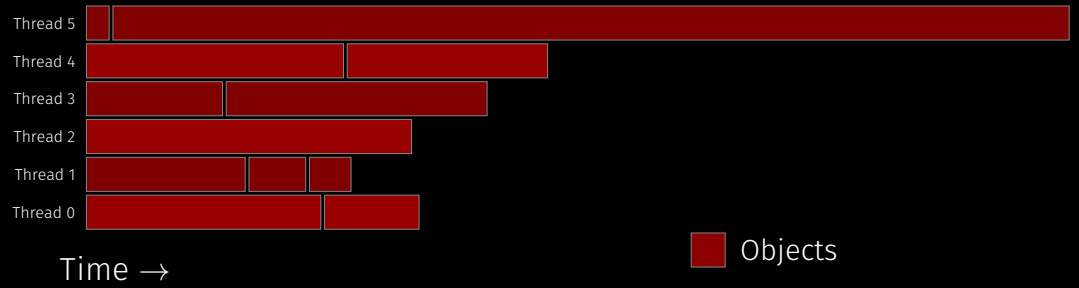


Let me just illustrate this with a timeline.

Each thread starts by working on different objects in parallel.

Subdivision Surfaces - Multi-Threading

- Tessellate and build acceleration structures before rendering
- Exact bounds for displaced surfaces without user intervention
- All instances known during tessellation
- Allows perfect threading (object parallel → patch parallel)



But of course different objects take different amounts of time. So one unlucky thread, like thread 5 here, could wind up working on an object that's much bigger than the rest.

So just object level parallelism isn't enough, it can leave a lot of threads idle. This isn't rare at all, there's usually at least one hero asset in the frame that needs more tessellation or displacement that the rest.

Subdivision Surfaces - Multi-Threading

- Tessellate and build acceleration structures before rendering
- Exact bounds for displaced surfaces without user intervention
- All instances known during tessellation
- Allows perfect threading (object parallel → patch parallel)

So we actually allow threads that don't have any new object to grab to join the threads that haven't finished yet. This way all the cores stay busy and we can finish much faster.

This isn't just for subdiv either, we handle BVH building and a few other cases like this too.

The good thing about starting with object parallelism is that it doesn't have any locking at all. So it scales almost linearly. The patch parallel part also scales really well but only for objects that have enough patches. So its good to keep it for the end.

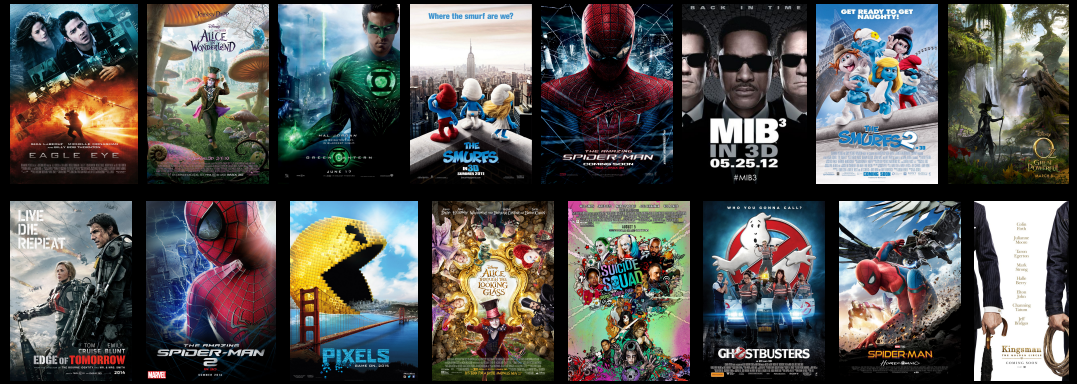
Conclusion

That covers most of what I wanted to talk about today. Let me quickly conclude by saying...

The Last 13 Years ... 10 Animated Films

...we've been using this renderer for over 13 years now. That includes 10 animated films...

The Last 13 Years ... 20+ VFX Films



...and more than 20 visual effects films - this isn't the complete list.

Conclusion

We have come a long way since Monster House:

- Some features presumed impractical → Feature Complete
- Subdivision tessellation manually managed → Automatic
- Variance reduction tricks manually enabled → Automatic
- Noise hunting → Adaptive + Denoise

So we've come a really long way since our original "experiment" on Monster House.

At the time we just assumed some features were totally impractical. Now the renderer is feature complete.

Subdivision tessellation used to be something artists manually managed. Now the renderer handles it automatically.

Variance reduction was something we had to do somewhat manually, by tagging objects with special flags or tuning modes in the shader. Now all those tricks handled automatically by the integrator.

And finally, the combination of adaptive sampling and denoising is really starting to change how we approach the debugging of noisy frames.

Remaining Challenges

We have come a long way and yet …

- High albedo media (snow, clouds, white fur) still challenging
- Some variance reduction tricks break physicality
- Keeping up with hardware (high core counts, wide SIMD, GPUs)
- Keeping up with the competition!

But I don't want to give the impression that we are done yet. There's still a lot of interesting challenges.

A big one is dealing with cases that need really deep bounces. Things like snow, clouds or white fur. We are actually working on a project right now that has all three of those at the same time.

Also, some of the variance reduction tricks we use break the physics of the light simulation a bit. Tricks like clamping or approximate caustics work great but they don't look as good as doing the real thing. So we're always looking for better solutions to those problems.

Remaining Challenges

We have come a long way and yet …

- High albedo media (snow, clouds, white fur) still challenging
- Some variance reduction tricks break physicality
- Keeping up with hardware (high core counts, wide SIMD, GPUs)
- Keeping up with the competition!

And the hardware we run on keeps changing too. CPUs are getting more and more cores and wider SIMD and making sure we take full advantage of that power means re-evaluating all our code constantly. I suspect we'll be revisiting vectorization again in the near future.

And finally - like we've seen in this course, the whole industry has really embraced path tracing and is doing really amazing work. So we have to keep pushing to keep up with everyone.

Acknowledgments

Imageworks Rendering Team:

- Cliff Stein
- Alex Conty
- Larry Gritz
- Jesse Andrewartha

And all our artists!

I just want to acknowledge my co-workers on this project: Cliff has been with us since Monster House, and recently worked on tessellation and denoising.

Alex, who implemented all the bidirectional integrators and adaptive sampling. Hopefully you also had a chance to catch his talk on Monday about many light sampling.

Larry who wrote most of our open source components: OpenImageIO and OSL.

And Jesse who is a our head of render efficiency and has helped artists really make the most of the renderer throughout its evolution.

And of course all our amazing artists - let me actually play our reel from last year's projects so we can see what the renderer is capable of today...

Thank you for your attention, and I'll be happy to take any questions.