



MoonRay

SIGGRAPH 2017

Hello, my name is Brian Green. I am the technical lead for Rendering at DreamWorks Animation. I am very happy to introduce you to MoonRay today.

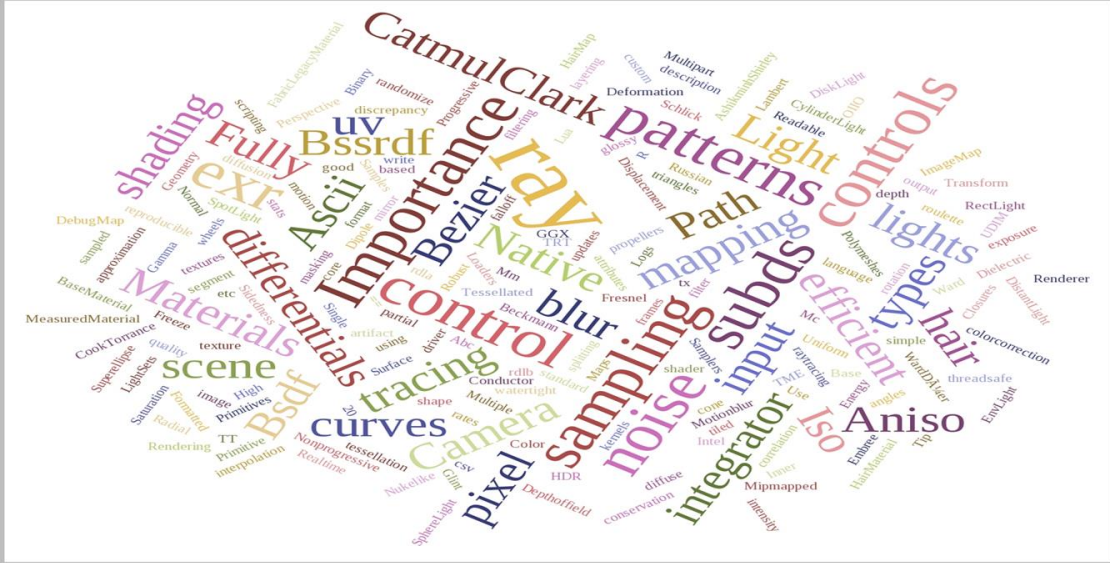


Objectives

- Introduce MoonRay
- Focus on Unique Features
 - Cloud Deployment and Parallelization via Arras
 - Diagnostic Material AOV syntax
 - Developing Shaders
 - Vectorization!

My topics for this session include a brief introduction to moonray followed by focused descriptions of several unique features of the renderer. These include distributed rendering, a discussion of a new aov syntax, development of shaders using ISPC., and finally, I'll finish up with a detailed look at our approach to vectorization, which is Moonray's most defining feature to date.

What is MoonRay?



So what is Moonray? As a real quick approach to answering this question, here is a calculated word frequency Image, from our documentation.



What is MoonRay?

MoonRay is our new high-end production rendering system

Monte-Carlo Ray Tracer (MCRT)

100% ray-traced, no pre-passes

Easy to use, very fast iterations

Integrated into various tools

Maya, Katana, In-house lighting tools, MotionBuilder

Arras cloud computation framework

Provide rendering as a service

Leverage distributed rendering

Moonray is our new high-end production rendering system. It replaces “Moonlight”, our deep-framebuffer, micro-polygon rasterization system.

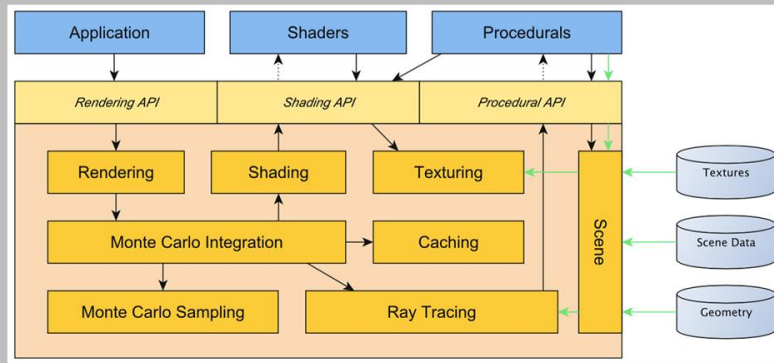
It is a monte-carlo raytracer. 100% ray-traced, no pre-passes. It is easy to use and provides artists with very fast iterations. It is integrated into wide variety of tools such as Maya, Katana, in-house lighting tools, and MotionBuilder.

Moonray is provided as a service to clients via our in-house cloud framework “Arras”. Not only does this simplify application integration, but it also allows moonray to take advantage of massive machine scale distributed rendering.



Modern Architecture

Render / Integrator / Closure / Shader



Leverage, collaborate with, and customize open source components: Embree, OpenImageIO, OpenSubdiv, OpenVDB, etc.

We completely wrote Moonray from scratch, leveraging state of the art open source components where appropriate. No studio legacy code was used. The architecture is cleanly divided across three different APIs: The rendering API for clients to initiate rendering, the shading API for the development of pluggable shaders, and the procedural API for the development of geometry generators. Embree is our ray-intersection engine. We use OpenImageIO to generically handle different image file formats. We follow a design that is very similar to that as described in PBRT, where we have a renderer that implements integration algorithms on top of closures that are produced by material shaders.



High Performance

"Keep all the lanes of all the cores of all the machines busy all the time with meaningful work"

Scalability up to real-time rendering

Multi-machine, Multi-threading, Vectorization

A new, clean architecture, designed from the ground up to never lock and efficiently distribute work

Trace and shade billions of rays per image

Thin interfaces across renderer components

No data-structure redundancy

Data Oriented Design

CPU is fast, memory access latency is slow

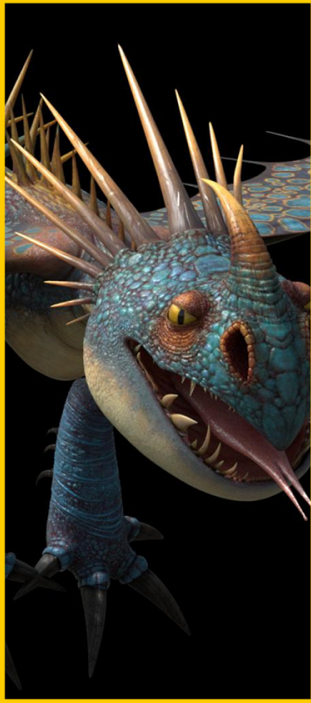
Careful data layout

Memory coherent processing in batches

SIMD / SPMD vectorization end-to-end

All renderers have personalities. "Keeping all the lanes" is our mantra and our personality.

These are a few of our guiding principles for development. Our goal is to achieve scalability up to real-time rendering leveraging all of the available hardware. Trace and shade billions of rays which implies thin interfaces and no data structure redundancy. We embraced and applied "Data Oriented Design" rather than Object Oriented Design in many places. This is a methodology that first grew in the games industry, but we applied it with great success in Moonray.



High Performance

"Keep all the lanes of all the cores of all the machines busy all the time with meaningful work"

Fast Render Prep

- Parallel loading and update of scene objects
- Optimized BVH construction

Optimized texture sampling on top of OpenImageIO

Bundled / wavefront path tracing

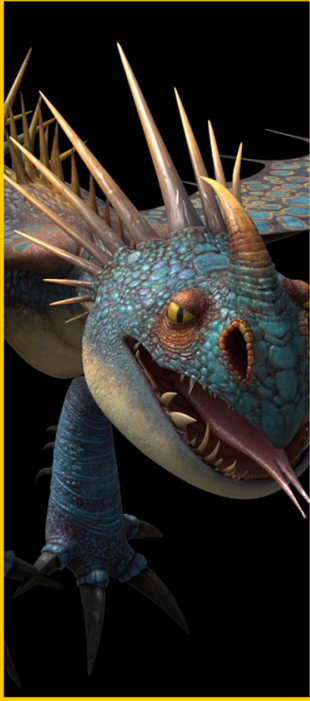
- Embree ray-packet or ray-streaming
- Most components vectorized with the Intel® SPMD program compiler

Broadly speaking, we divide the moonray process into render prep and mcrt rendering. While most of our performance focus has been the mcrt rendering time, we have not forgotten about "render prep", which can be quite expensive in many production contexts. This includes parallel loading and updating of scene objects, along with optimized bvh construction. We have written optimized texture sampling code on top of OIIO and make use of bundled/wavefront path tracing using either embree's ray-packet or ray-streaming APIs. Most components are vectorized with ISPC.



Distributed Rendering

I'd now like to briefly discuss the “embarrassingly parallel” aspects of parallelization and how we uniquely approach this problem.



Multi-threading

Divide Frame into 8x8 tiles

Create an order for how those tiles are processed

Place tiles in work queue

Idle threads pull work from this work queue. There is no central scheduler.

Progressive rendering by making the unit of work a partial pass of a tile

No Locks!

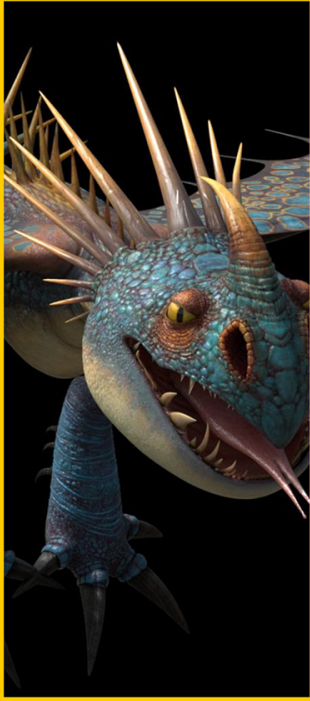
- Frame Buffer updated with atomics
- Filter Importance sampling
- Memory allocation from memory pools

Compared to vectorization (which will be discussed later), multi-threading is quite straight forward.

First we divide the frame into 8x8 tiles and create a user configurable order for those tiles.

The order can be top-down, bottom-up, morton, spiral, etc...

No Locks. The frame buffer is updated with atomics. We use filter importance sampling as opposed to traditional post pixel filters to minimize pixel write contention between threads, and all memory allocation is handled via pre-allocated memory pools.



Distributed Rendering

Each Machine is given an ID

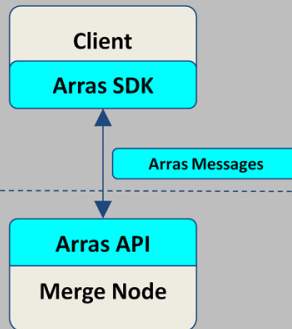
The same tile generation scheme is run on each node

But only a subset of those tiles are run on any particular machine

Results are sent to a single merge node

Extension to distributed rendering is also straight-forward

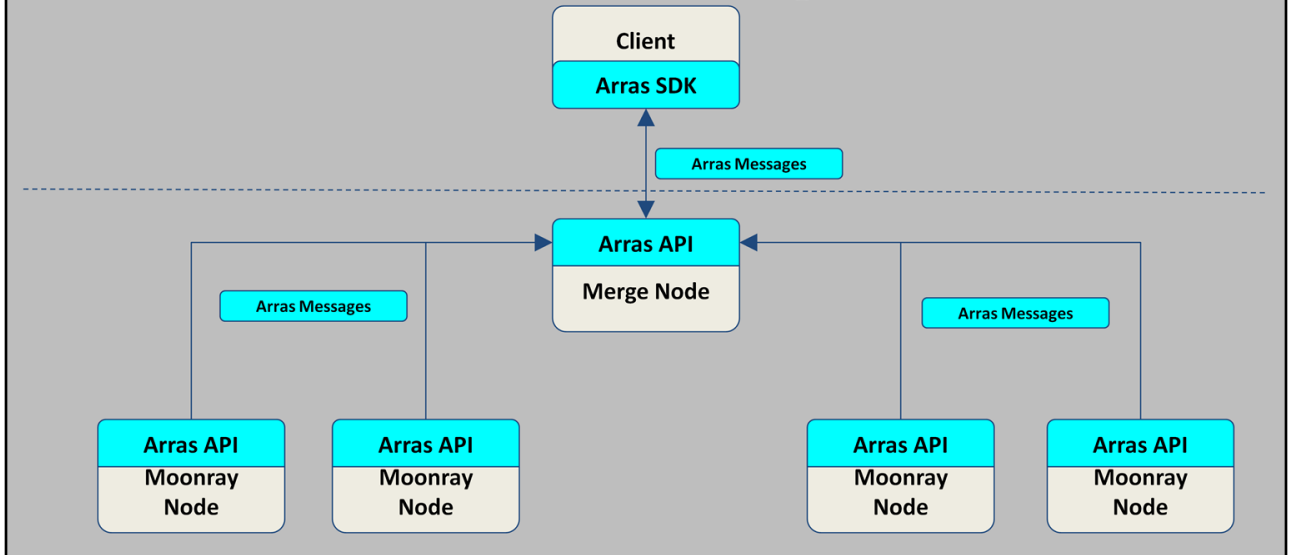
Distributed Rendering with Arras



Arras is our cloud based framework that we use for distributing rendering tasks to a cluster or cloud. The client programs make use of an extremely thin and portable Arras SDK to connect to an arras rendering session. In the cluster there is a single moonray merge node which is responsible for sending final rendering results back to the client.

In addition to enabling distributed rendering, this architecture makes it extremely simple to integrate MoonRay in a wide variety of client applications. Much easier than if the entire core rendering libraries needed integration with the client application.

Distributed Rendering with Arras



Multiple moonray nodes communicate via messages and the arras computation api.

Distributed Rendering with Arras



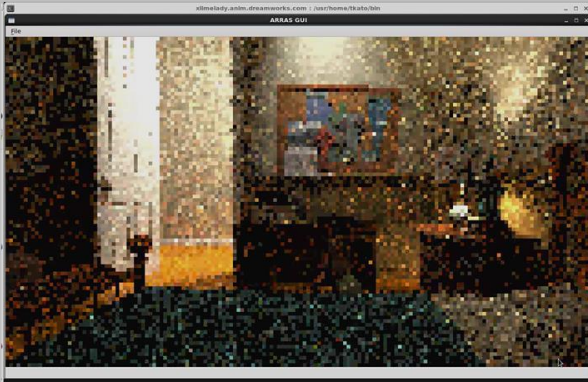
This architecture, while seemingly simple has produced linear speed-ups on clusters of 30 machines.

<PgDn pause PgDn to start both videos>

This is an example of what a 9.5x performance improvement feels like to an artist. In both windows, the artist is using a simple arras client program called "arras_gui". This program allows the artist to manipulate his camera view and request progressive rendering results with a fixed frame rate (24 fps in this case). Note that both programs produce exactly the same frame rate. The frame buffer is snapshotted every 24th of a second in both cases. But the quality of the result is what should be paid attention to. On the left, the artist is connected to a 2 machine arras session, on the right a 19 machine arras session. Both sessions contain identical 32 core machines.

Note that progressive rendering makes the 2x32 case look pretty good. This is one of the powerful aspects of mcrt progressive rendering.

Distributed Rendering with Arras



2x32



19x32

This is an example of what a 9.5x performance improvement feels like to an artist. In both windows, the artist is using a simple arras client program called “arras_gui”. This program allows the artist to manipulate his camera view and request progressive rendering results with a fixed frame rate (24 fps in this case). Note that both programs produce exactly the same frame rate. The frame buffer is snapshotted every 24th of a second in both cases. But the quality of the result is what should be paid attention to. On the left, the artist is connected to a 2 machine arras session, on the right a 19 machine arras session. Both sessions contain identical 32 core machines.

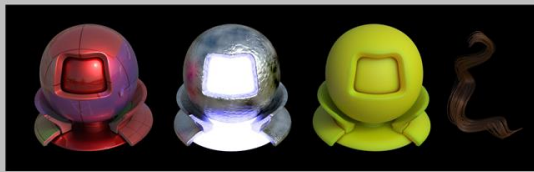
Note that progressive rendering makes the 2x32 case look pretty good. This is one of the powerful aspects of mcrt progressive rendering.



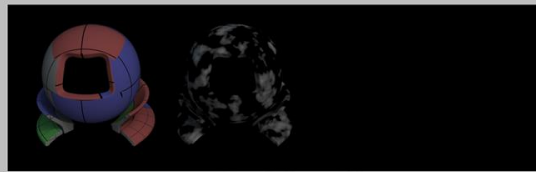
AOVs

Every production renderer worth its salt needs to support a deep and sophisticated aov system. Moonray is not an exception in this respect.

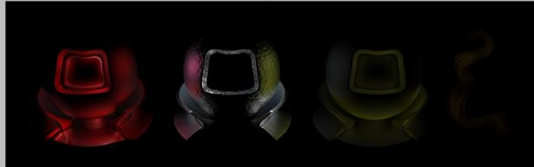
Light Path Expressions



Beauty



Direct Diffuse = CDL



Indirect Glossy = $C \langle RG \rangle [DSG] + L$



Direct Glossy = CGL



Indirect Mirror = $C \langle RG \rangle [DSG] + L$



Direct Mirror = CSL

The emerging standard for AOVs in an MCRT renderer is Light path expressions. While LPEs do a great job of telling you which paths to include in a particular AOV, they say nothing about what values should actually be included in that AOV. For Moonray, all light path expression based AOVs are associated with radiance values.

State AOVs



Beauty



Normal



Texture st



dPdu



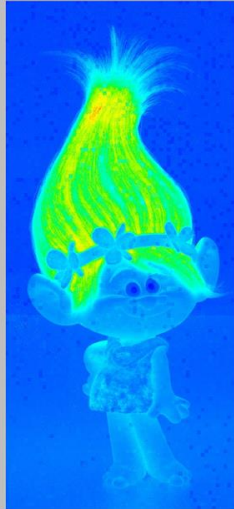
dPdv

We of course support very basic outputs, which describe the differential geometry at the primary ray intersection point. We call these “State AOVs”

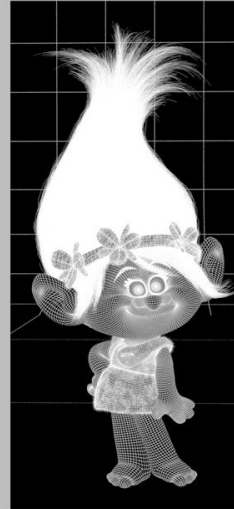
Heat Map / Wireframe



Beauty



Heat Map



Wireframe

We also support custom outputs such as HeatMap (time per pixel) and wireframe which is used for tessellation diagnostic purposes.

Material AOVs



Beauty



Diffuse Color



Spec Hair Color



Spec Surface Color

...

What we have termed “Material AOVs” are probably the most unique AOV feature in Moonray. Material AOVs provide detailed diagnostics about the materials and are extremely helpful to surfacing and lighting artists when verifying material correctness and standards conformance. Examples of various color aovs are shown in the slide.



Material AOV Syntax

Provide a diagnostic view of a material property. E.g.

Roughness

Emission

Color

Does not include any information that is influenced by actual scene lighting

Occlusion is not considered

Material AOVs are used for diagnostic purposes. Our material shaders produce parameterized multi-lobe bsdf objects. We have developed a material aov syntax (intentionally similar to LPE syntax) that allows an artist to extract important bits of information about this parameterization. This syntax complements light path expressions, which concern themselves with how a ray travels through the scene. Material AOV syntax focuses on extracting properties of a bsdf at an intersection.

Material Aovs provide ...



Material AOV Syntax

Selection + Property = Material Aov

A Bsdf closure consists of

- A set of parameterized Bsdf Lobes

 - Reflection or Transmission

 - Diffuse, Glossy, or Mirror

 - Optional Fresnel

A single Bssrdf lobe with an optional Fresnel

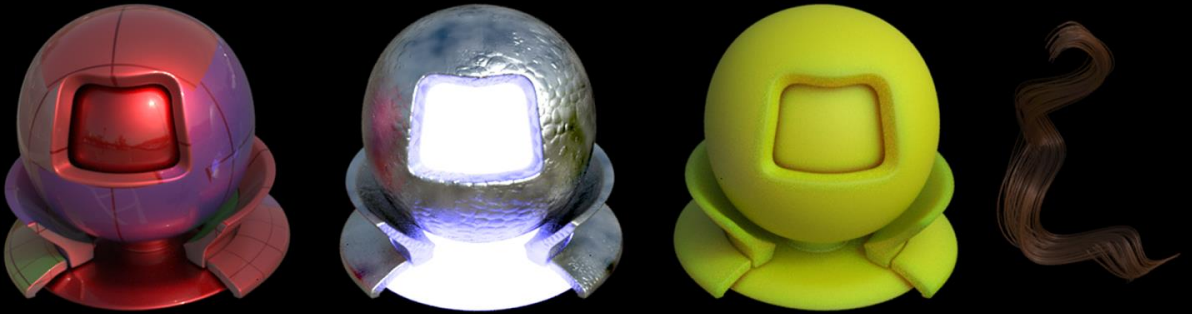
An Emission Color

Once we define the “property” we are interested, we must “select” the components of the multi-lobes bsdf that we are interested in....

A selection plus a property equals a Material AOV

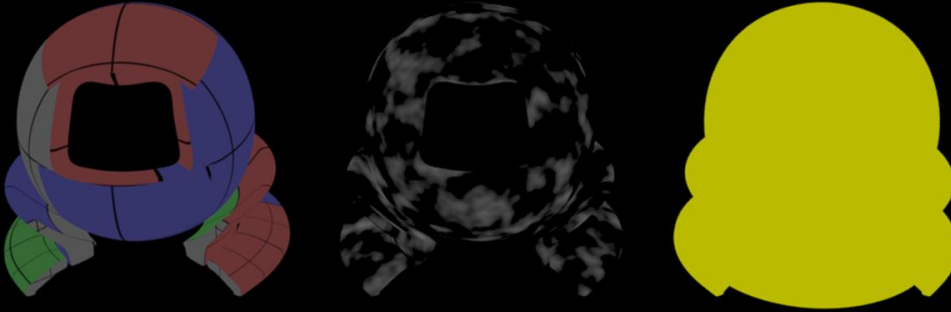
To review, a Bsdf closure consists of

Example - Beauty Render



Lets look at a few examples of the syntax in action, before defining it precisely....

Example - Diffuse and SS Color



Material AOV = DSS.color

To extract the diffuse and subsurface color, use the syntax DSS.color. Here the property is “color” and the selection is DSS.

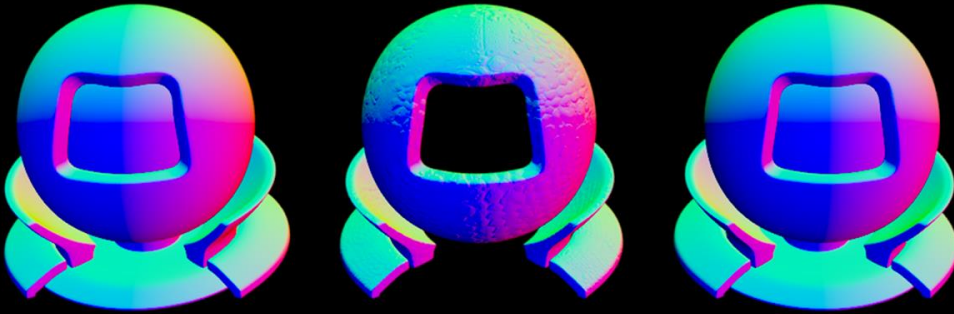
Example - Glossy Color



Material AOV = G.color

Similarly to extract the color of all glossy lobes, use G.color

Example - Normal



Material AOV = RTSS.normal

In this case, the property is “normal” and the selector is RTSS – read as all reflection, transmission, and subsurface lobes.

Example - Glossy and SS Fresnel Factor



Material AOV = `GSS.fresnel.factor`

Finally, we are examining the “factor” parameter for the fresnel objects associated with all glossy and subsurface lobes.

Syntax

`[(SS | R | T | D | G | M)+\..][fresnel\..]<property>`

Where:

R = reflection side, T = transmission side

D = diffuse lobe, G = glossy lobe, M = mirror lobe

SS = bssrdf

Fresnel means to select the lobe or bssrdf's fresnel

Formally....

Labels

Material shaders can assign labels to BsdF and Bssrdf lobes

Multiple labels are allowed per lobe/bssrdf

These can be used to further refine material aov selection.

Any selected lobe or bssrdf must match at least one label.

Labels in the syntax are “or” operations

One final bit of syntax, the integrates nicely with light path expressions are labels.

Material shaders can assign....

Lets look at a couple more examples.

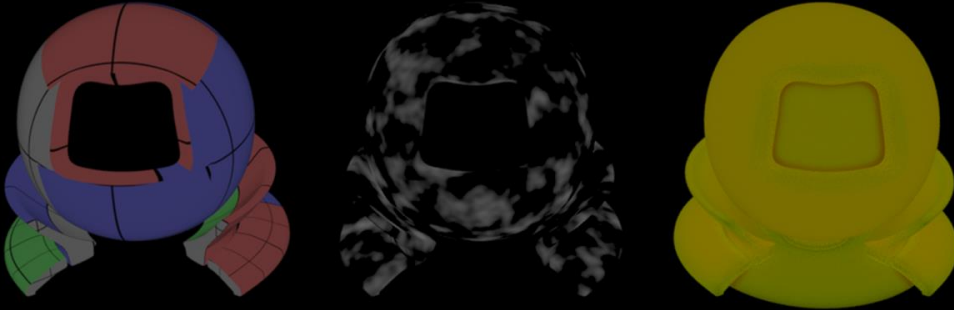
Example - Specular Albedo



Material AOV = `'specular'.albedo`

The shader writer has assigned the label “specular” to some of his lobes. To extract the albedo property of these lobes use ``specular`.albedo`

Example - Diffuse and Translucent Albedo



Material AOV = 'diffuse''translucency'.DSS.albedo

In this case, the shader writer has assigned "diffuse" and "translucency" to some of his lobes. Notice how we can scope the label match to just the diffuse and subsurface lobes by using the ".DSS" selector. Any non D or non SS lobe, even if it has the appropriate label, will not match this aov.

Syntax - with labels

`[('<Label>')+\.][(SS | R | T | D | G | M)+\.][fresnel\.]<property>`

Where <Label> is any string.

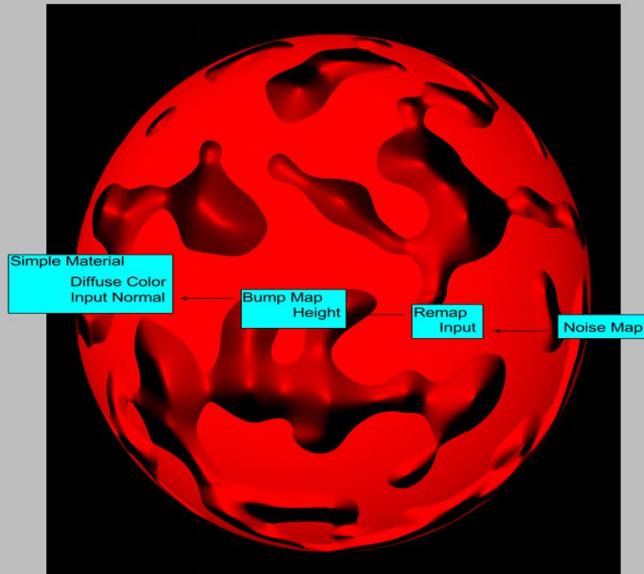
So the final syntax, with labels, is a simple and powerful way to extract information from a material.

ISPC Shading Framework



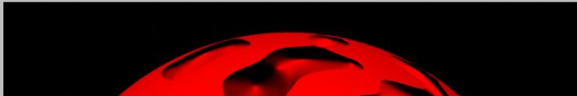
For my next topic, I will show a simple example of how we use ISPC to develop vectorized shaders

ISPC Shading Framework




A simple material with a diffuse color and input normal. The normal is bound to a bump map shader, which routes through a remap shader to a noise map.

ISPC Shading Framework

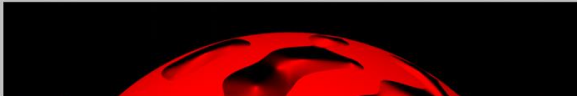


```
local noiseMap = NoiseMap("/Scene/surfacing/noiseMap") {  
    ["frequency multiplier"] = 5,  
}  
  
local remapMap = RemapMap("/Scene/surfacing/remap") {  
    ["input"] = bind(noiseMap),  
}  
  
local bumpMap = BumpMap("/Scene/surfacing/BumpMap") {  
    ["height"] = bind(remapMap)  
}  
  
local simpleMtl = SimpleMaterial("/Scene/surfacing/SimpleMtl") {  
    ["diffuse color"] = Rgb(1, 0, 0),  
    ["input normal"] = bind(bumpMap)  
}
```




Scene configuration can be specified using lua (the rdl - ascii format, clap, clap). At the bottom of the code is an instance of a simpleMtl. It has two attributes. A diffuse color and an input normal. The color is red, the normal is bound to a bump map shader, which means it can vary over the surface. The Bump map shader has a single attribute "height". Which I want to be between 0 and 1, so I Bound this to a remap shader, which is itself bound to a noise map.

ISPC Shading Framework



```
local noiseMap = NoiseMap("/Scene/surfacing/noiseMap") {  
    ["frequency multiplier"] = 5,  
}  
  
local remapMap = RemapMap("/Scene/surfacing/remap") {  
    ["input"] = bind(noiseMap),  
}  
  
local bumpMap = BumpMap("/Scene/surfacing/BumpMap") {  
    ["height"] = bind(remapMap)  
}  
  
local simpleMtl = SimpleMaterial("/Scene/surfacing/SimpleMtl") {  
    ["diffuse color"] = Rgb(1, 0, 0),  
    ["input normal"] = bind(bumpMap)  
}
```



The bind function is how attributes of one shader are wired to the output of another. Binding information is known only at run-time and is a property of the scene setup.

ISPC Shading Framework

```
{
  "name": "SimpleMaterial",
  "type": "Material",
  "attributes": {
    "attrDiffuseColor": {
      "name": "diffuse color",
      "type": "Rgb",
      "default": "Rgb(1.0f, 1.0f, 1.0f)",
      "flags": "FLAGS_BINDABLE"
    },
    "attrInputNormal": {
      "name": "input normal",
      "type": "Vec3f",
      "default": "Vec3f(0.0f, 0.0f, 1.0f)",
      "flags": "FLAGS_BINDABLE"
    }
  },
  "labels": {
    "aovDiffuse": "diffuse"
  }
}
```

```
{
  "name": "BumpMap",
  "type": "Map",
  "attributes": {
    "attrHeight": {
      "name": "height",
      "type": "Float",
      "default": "1.0f",
      "flags": "FLAGS_BINDABLE"
    }
  }
}
```

Shader attributes are specified using json files. Code generation from these json files is an integral part of our system. Not only does it remove the need for substantial amounts of boiler-plate coding, but more importantly it allows for code generation that will provide syntactic help to enable just-in-time (JIT) compilation of network connections.

In the slide, I have provide the attribute json descriptions for the SimpleMaterial and BumpMap shader.



ISPC Shading Framework

ISPC = Intel® SPMD Program Compiler

SPMD = Single Program Multiple Data

C-like with “varying” and “uniform” keyword extensions

Automatic masking

Now lets take a step back. Actual shading functions are written in ISPC.

ISPC = Intel...

In our shading system, each vector lane represents an intersection point to be shaded. For example, on AVX2 hardware, 8 different intersection points are passed into the shade function by the renderer. ISPC greatly simplifies the process of writing vectorized code. It Eliminates the extremely unproductive need for intrinsics programming and explicit masking. I think this would be inappropriate for most production shader developers, which would make adoption of our vectorized renderer extremely problematic.

ISPC Shading Framework

```
static void
shade(const uniform Shadable *      uniform me,
      uniform ShadingTLState *uniform tls,
      const varying State          &state,
      varying Closure *            uniform closure)
{
    // evaluate our input normal - no need for derivatives
    const Vec3f N = evalNormalInput(me, tls, state);

    // evaluate our diffuse color - no need for derivatives
    const Color diffuseColor = evalAttrDiffuseColor(me, tls, state);

    // add the lobe
    if (!isBlack(diffuseColor)) {
        Closure_addLambertBsdfLobe(closure, tls, state,
                                  /* scale = */ diffuseColor,
                                  /* normal = */ N,
                                  /* labels = */ aovDiffuse);
    }
}
```

Here is the ISPC code for our SimpleMaterial. This is a root material node and is responsible for filling in the output closure parameter.

Passed as input to the function is a varying state parameter. Each lane of State represents a unique intersection point to shade. The Shadable and ShadingTLState parameters are uniform objects that provide handles to the scene data object and utility functions such as thread-local memory pools that are used to allocate and build up closures.

ISPC Shading Framework

```
static void
shade(const uniform Shadable *      uniform me,
      uniform ShadingTLState *uniform tls,
      const varying State          &state,
      varying Closure *            uniform closure)
{
    // evaluate our input normal - no need for derivatives
    const Vec3f N = evalNormalInput(me, tls, state);

    // evaluate our diffuse color - no need for derivatives
    const Color diffuseColor = evalAttrDiffuseColor(me, tls, state);

    // add the lobe
    if (!isBlack(diffuseColor)) {
        Closure_addLambertBsdfLobe(closure, tls, state,
                                   /* scale = */ diffuseColor,
                                   /* normal = */ N,
                                   /* labels = */ aovDiffuse);
    }
}
```

These hilit functions are auto-generated by our code generator and are the locations where connections between shaders take place. The first function is responsible for evaluating the input normal attribute, which in our case is bound to a bump map shader. The second function is responsible for evaluating the diffuse color attribute. In our case, this is just a simple constant color.

Only at run-time do we know if a shader is bound or not,

It is also worth noting that neither of these results require bound shaders to produce derivatives.

ISPC Shading Framework

```
static varying Color
sample(const uniform Map *          uniform map,
       uniform ShadingTLState *    uniform tls,
       const varying State &      state)
{
    // we'll compute 4 bumped positions:
    // 3--2
    // 1 1
    // 0--1
    // where 0 is the current shade location and
    // 1-3 are computed based on dpdx and dpdy
    // then we'll compute normals for each 0->1
    // and 0->(1+1) fan and return an average

    // compute bumped P
    // todo: d[PN][dxy] in state would be useful
    Vec3f p[4];
    const Vec3f P = state.mP;
    const Vec3f N = state.mN;
    const Vec3f dPds = state.mdPds;
    const Vec3f dPdt = state.mdPdt;
    const float dsdx = state.mdSdx;
    const float dsdy = state.mdSdy;
    const float dtdx = state.mdTdx;
    const float dtdy = state.mdTdy;
    const Dual h = dEvalAttrHeight(map, tls, state);
    const float H = eval(h);
    const float dHdx = dDx(h);
    const float dHdy = dDy(h);
    p[0] = P + N * H;
    p[1] = P + dPds * dsdx + dPdt * dtdx + N * (H + dHdx);
    p[2] = P + dPds * (dsdx + dsdy) + dPdt * (dtdx + dtdy) + N * (H + dHdx + dHdy);
    p[3] = P + dPds * dsdy + dPdt * dtdy + N * (H + dHdy);

    // compute normals
    Vec3f n[2];
    n[0] = normal(p[0], p[1], p[2]);
    n[1] = normal(p[0], p[2], p[3]);

    // average them to compute the final bumped normal
    Vec3f bump = normalize(n[0] + n[1]);

    return result;
}
```

This is my bump map shader. I'm not the best shader writer in the world, but it works. Remember all of the types without the keyword "uniform" are wide types.

ISPC Shading Framework

```
static varying Color
sample(const uniform Map *      uniform map,
       const uniform ShadingTLState * uniform tls,
       const varying State &    state)
{
    // we'll compute 4 bumped positions:
    // 3--2
    // 1 1
    // 0--1
    // where 0 is the current shade location and
    // 1-3 are computed based on dpdx and dpdy
    // then we'll compute normals for each 0->1
    // and 0->(1+1) fan and return an average

    // compute bumped P
    // todo: d[PNI][dxy] in state would be useful
    Vec3f p[4];
    const Vec3f P = state.mP;
    const Vec3f N = state.mN;
    const Vec3f dPds = state.mdPds;
    const Vec3f dPdt = state.mdPdt;
    const float dsdx = state.mdSdx;
    const float dsdy = state.mdSdy;
    const float dtdx = state.mdTdx;
    const float dtdy = state.mdTdy;
    const Dual h = dEvalAttrHeight(map, tls, state);
    const float H = eval(h);
    const float dHdx = dDx(h);
    const float dHdy = dDy(h);
    p[0] = P + N * H;
    p[1] = P + dPds * dsdx + dPdt * dtdx + N * (H + dHdx);
    p[2] = P + dPds * (dsdx + dsdy) + dPdt * (dtdx + dtdy) + N * (H + dHdx + dHdy);
    p[3] = P + dPds * dsdy + dPdt * dtdy + N * (H + dHdy);

    // compute normals
    Vec3f n[2];
    n[0] = normal(p[0], p[1], p[2]);
    n[1] = normal(p[0], p[2], p[3]);

    // average them to compute the final bumped normal
    Vec3f bump = normalize(n[0] + n[1]);

    return result;
}
```

The hilit function is where the bump map shader evaluates its height map attribute. In our case this is bound to a remap shader.

Note that this evaluation requires the child shader to produce derivatives. We make use of Dual algebra and autodiff for this purpose. Again note that the remap shader does not now if it must produce derivatives or not until runtime, when it is bound to a shader that needs them.



ISPC Shading Framework

JIT = Just In Time Compilation

LLVM defines a bitcode format and a run-time API to manipulate and JIT compile to machine code

```
ispc shader.ispc -emit-llvm -o shader_ispc.bc
```

Load needed .bc files into LLVM module.

Replace defined (but not declared) functions with run-time information

Run -O3 level optimizers ... Profit!

LLVM plays an important role in our shader run-time....

LLVM defines a bitcode format and a run-time API to manipulate and JIT compile to machine code.

When a shader is built, we use the ispc compiler to produce llvm bitcode files from the individual shader code.

At run-time we load needed bitcode files into an LLVM module.

Because our code generator generated the functions that define the connections between shaders, replacing them is a simple matter of search and replace using the LLVM API. Evaluation functions for attributes not bound to shaders (such as the diffuse color in our example) are just replaced with simple constants. Evaluation functions that are bound to shaders are replaced with inline code.

Once the replacements are in place, we run the O3 level optimizer. Good things then happen such as constants being folded, and dead code (such as unused derivative computations) are allided.

Vectorization

Lee et al. *Vectorized Production Path Tracing*. HPG '17



For my final topic, I'd like to talk about what is undoubtedly the single most defining and unique feature of Moonray – Vectorization. This work was presented earlier in the week at HPG by my good friend and colleague Mark Lee.

Vectorized Path Tracing

Now I am going to describe how we've mapped a scalar, depth first, path tracing algorithm into a corresponding vectorized version.

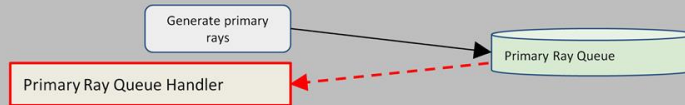
A very big distinction is that in the process, we've had to transform the algorithm to traverse each ray tree in a breadth first fashion as opposed to depth first.

Vectorized Path Tracing



We start off as before by generating primary rays.
Instead of directly intersecting these rays with the scene geometry, we add them to a primary ray queue.

Vectorized Path Tracing

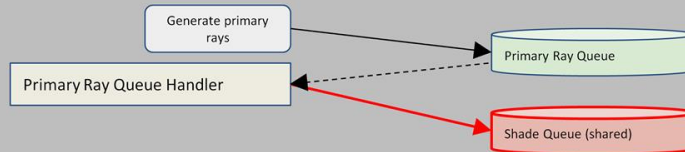


When the number of queued primary rays hits its pre-configured queue size limit we pass them onto the queue's associated handler.

The handler sends the rays into embree for ray / geometry intersection.

These queued primary rays will be very coherent and so we gain some performance benefit by passing them into embree in batches.

Vectorized Path Tracing



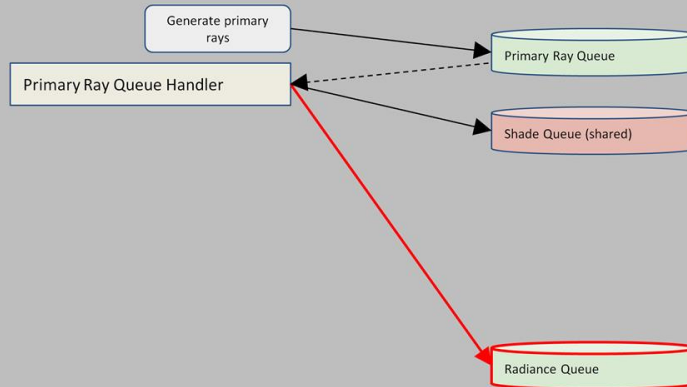
Intersecting each ray with a scene geometry returns hit point information as well as surface shader information.

We pre-allocate a separate queue per material instance.

This enables us to minimize code flow divergence since all entries in a shade queue are guaranteed to execute the same shader code which will be invoked with the same shader parameters.

Like other queues in the system, a shader queue is only flushed after it fills up.

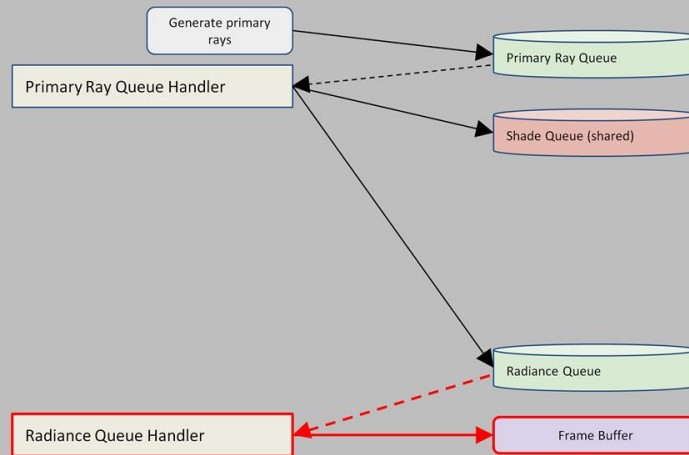
Vectorized Path Tracing



Another possibility is that a primary ray doesn't actually intersect any scene geometry at all.

In that case we test if it hit any lights and add the light contribution to the radiance queue.

Vectorized Path Tracing



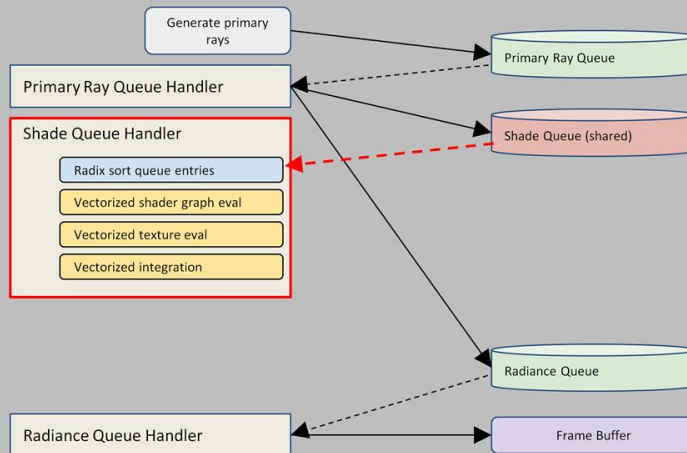
There is a potential scaling problem when dealing with tens or hundreds of threads all potentially trying to write radiance samples into a shared frame buffer.

This is true despite not locking and only using atomic operations to update radiances.

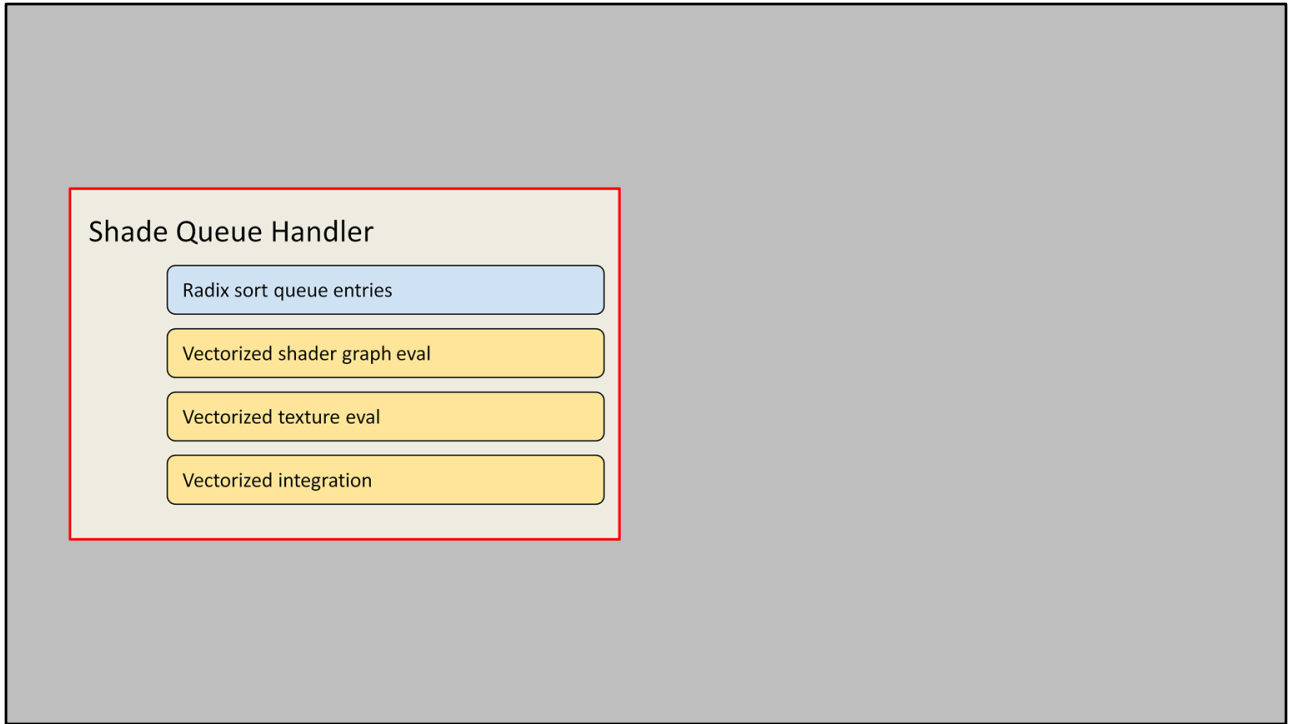
To alleviate this we first sort radiance queue entries by the pixel they belong to, and accumulate each belonging to the same pixel locally before attempting the atomic update.

The reason why radiance queues exist in our architecture are purely for scalability reasons.

Vectorized Path Tracing



So let's see what happens when a shade queue fills and needs to be flushed. Remember that all queues contain references to AOS data structures. The central idea behind the one big gray block here, which does all the shading, texturing, and integration work, is that AOS to SOA is not free, and so we want to only perform that transformation once. At this point in the pipeline is where we essentially switch over to fully vectorized execution. Let's zoom in to see some more details of what's happening in here.



This is the zoomed in view of the shade queue handler.

Shade Queue Handler

Radix sort queue entries

Transform AOS entries to SOA

Vectorized shader graph eval

Vectorized texture eval

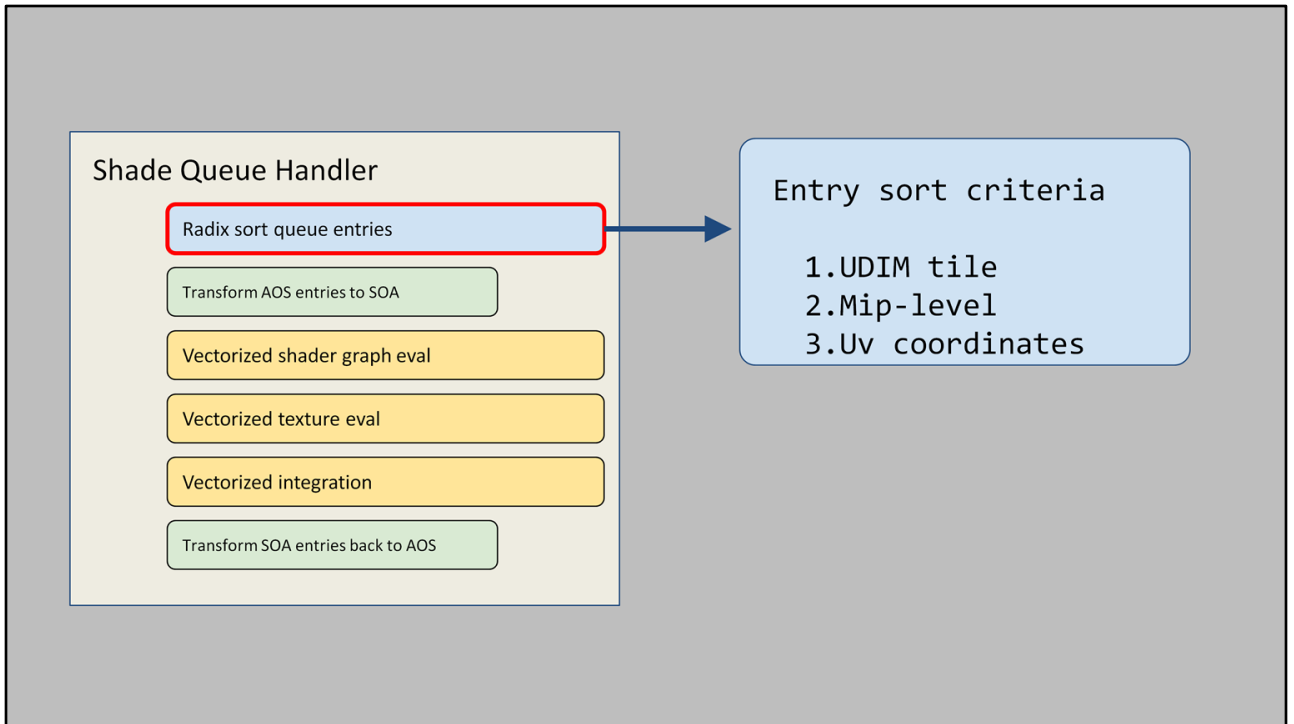
Vectorized integration

Transform SOA entries back to AOS

Now that it's expanded there are 2 other blocks we can add to fill in some extra details.

You can see that all the explicitly vectorized blocks, namely shading, texturing, and integration, are sandwiched between a pair of green blocks.

These green blocks are responsible for transforming the data back and forth between AOS and SOA, which we only want to do once per ray segment.



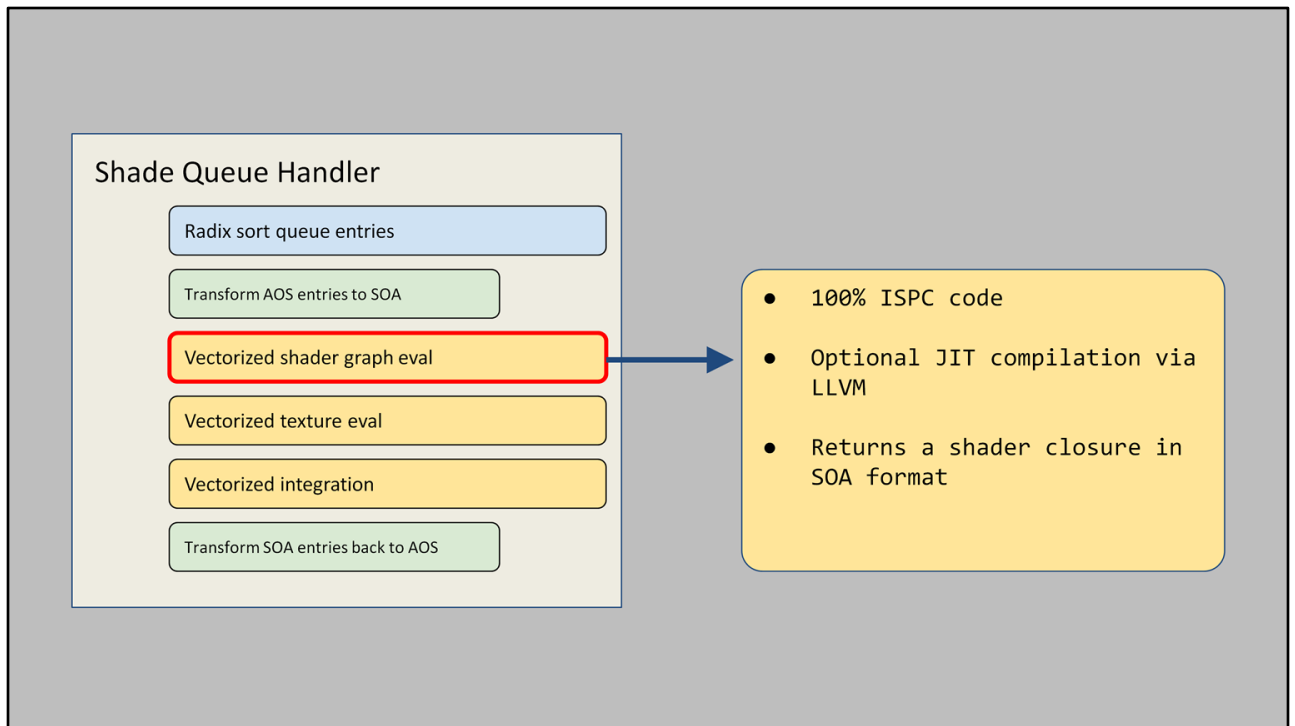
The inputs to the handler are a set of points which need to be shaded by a single material instance.

Before shading we first sort the entries to make texture lookups more coherent.

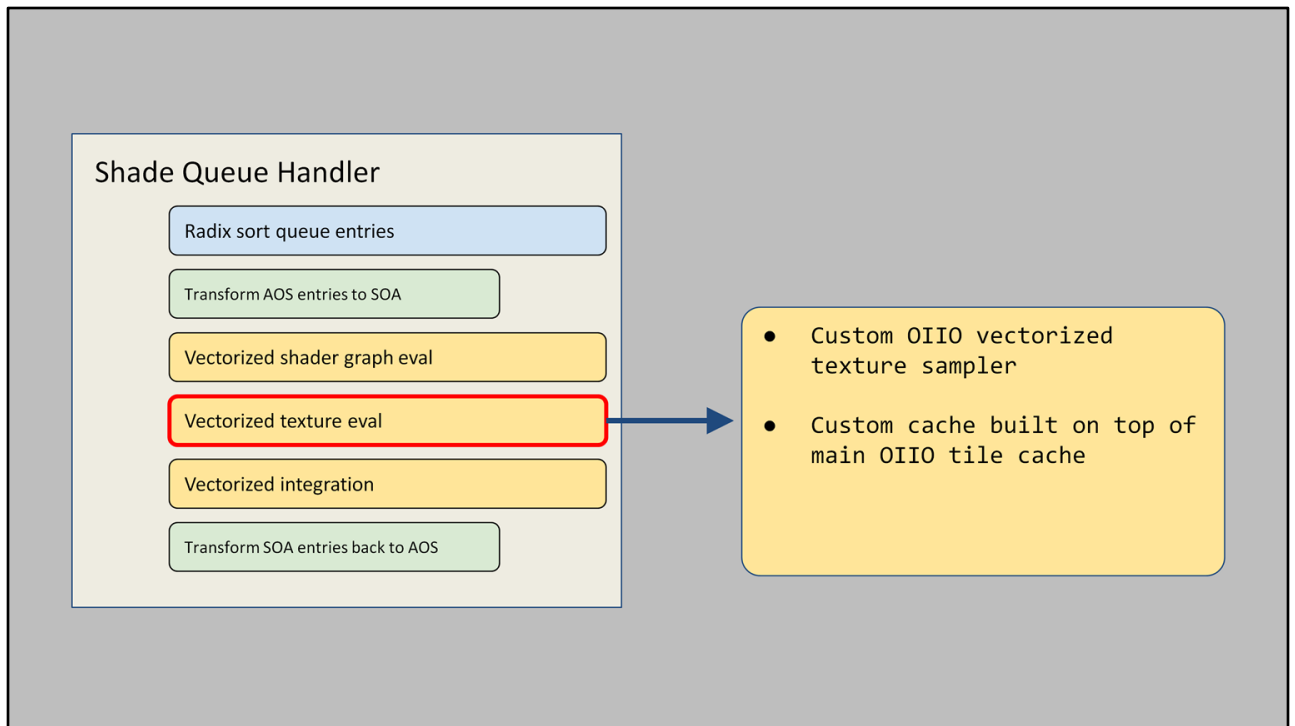
The box on the right shows a simplified set of the criteria we sort by.

Basically we want to get as much reuse out of each texture tile once loaded in, which is why we sort by udim tile first, and uv coordinates last.

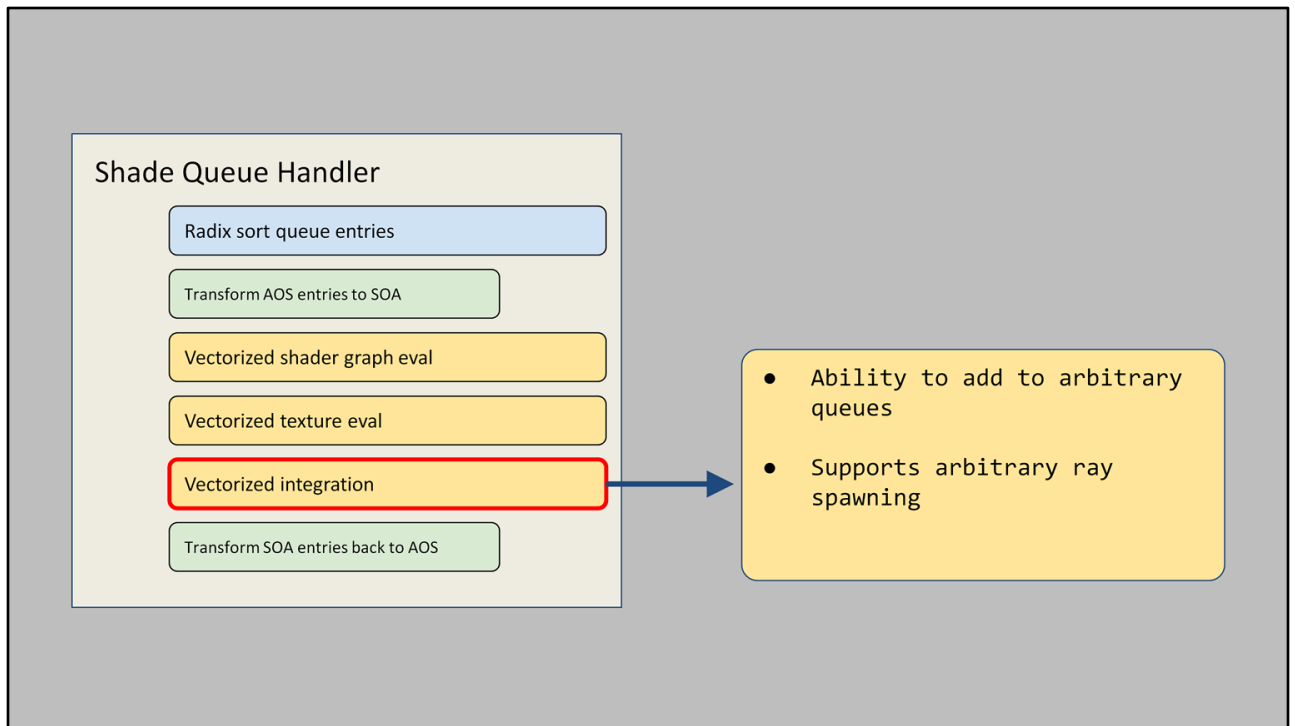
Once our entries are sorted, we're ready to transform them into SOA format for consumption by the ISPC kernels.



Shader graph evaluation proceeds as described previously



Shaders may call texture evaluation. This makes use of our customized OIIO vectorized sampler and cache.



Integration proceeds on the returned closure, fully vectorized, adding to queues as needed.

Shade Queue Handler

Radix sort queue entries

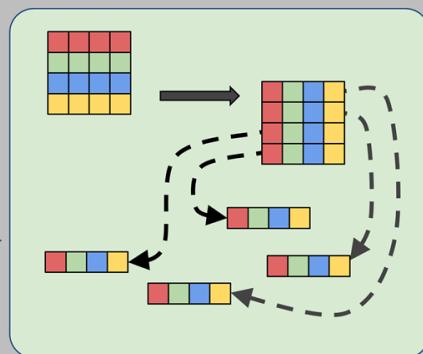
Transform AOS entries to SOA

Vectorized shader graph eval

Vectorized texture eval

Vectorized integration

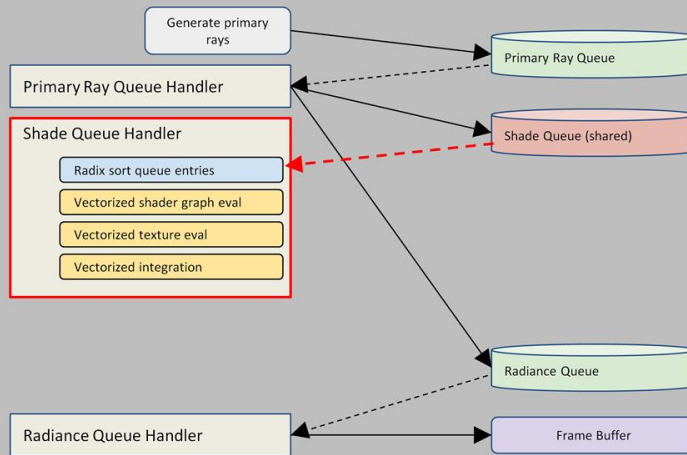
Transform SOA entries back to AOS



The entries which are returned from the integrator are still in SOA format, so we must convert them back to AOS format for subsequent queuing.

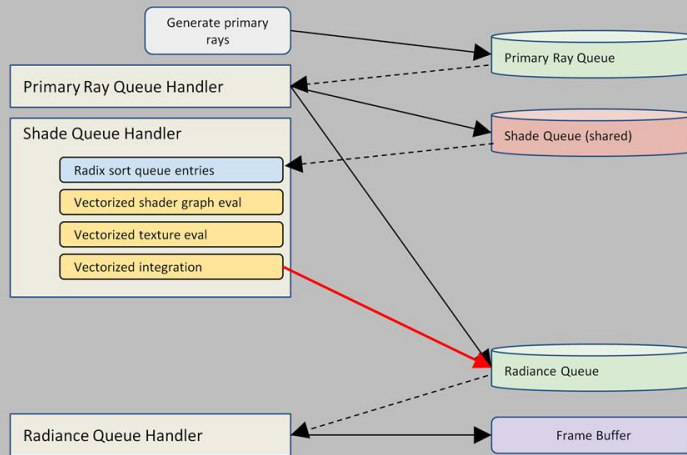
You may ask why the need to convert these back to AOS? We do this so that the entries can be freely resorted during the next wavefront.

Vectorized Path Tracing



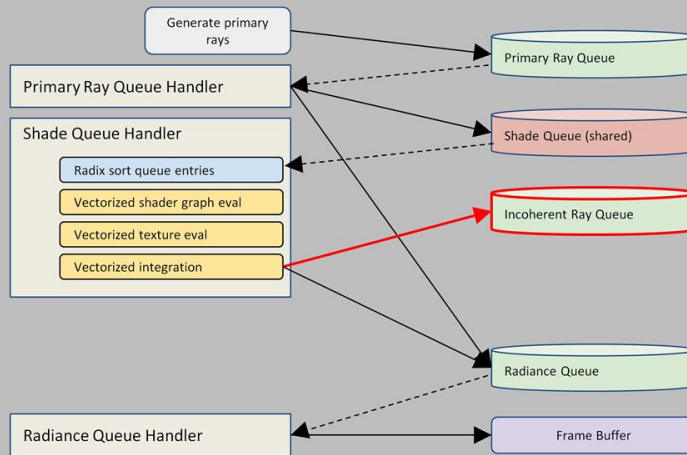
So let's zoom back out to the macro scale again.

Vectorized Path Tracing



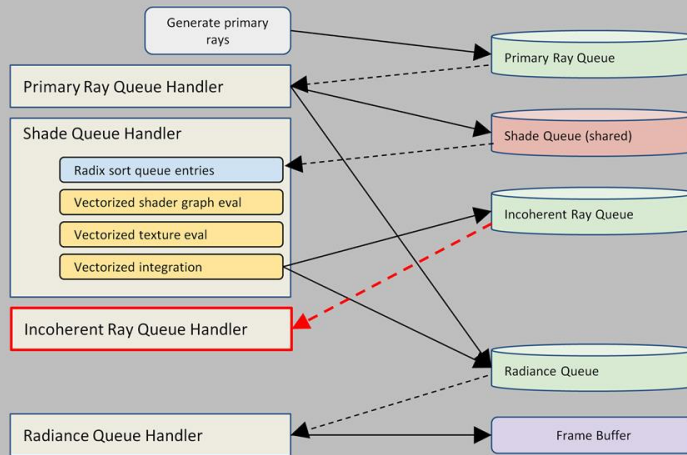
The first type of output which the integrator can generate are radiance values, one example of this is light from emissive surfaces.

Vectorized Path Tracing



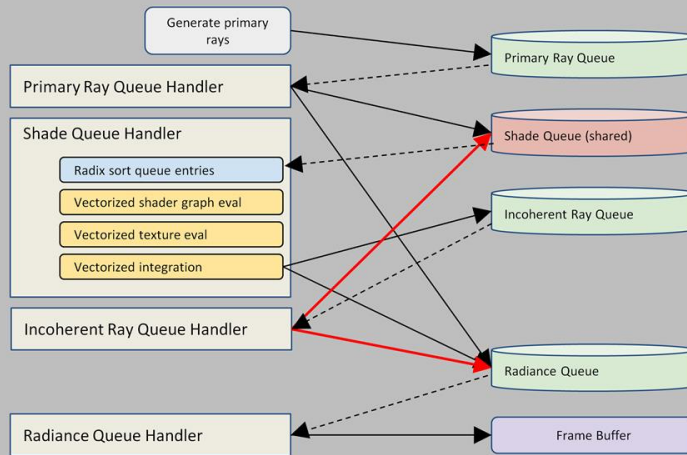
The integrator can arbitrarily spawn rays, which are typically more incoherent than primary rays, so we put them into a separate incoherent ray queue. Using a separate incoherent ray queue gives us the option of sorting these rays. The paper contains more details on our experiments with ray sorting.

Vectorized Path Tracing



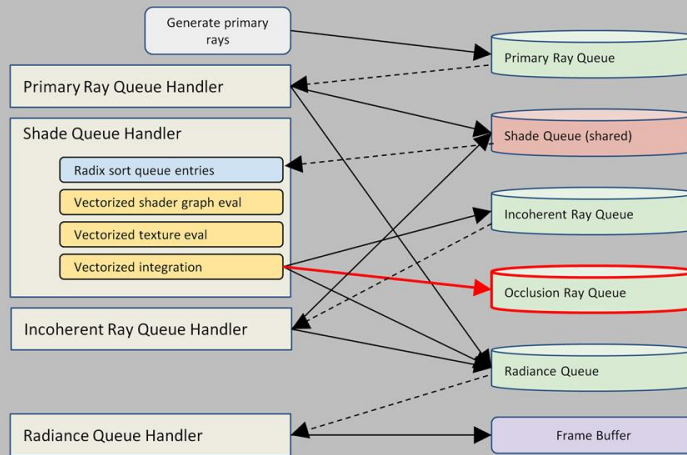
When the incoherent ray queue fills, we invoke the incoherent ray queue handler which sends these rays through embree.

Vectorized Path Tracing



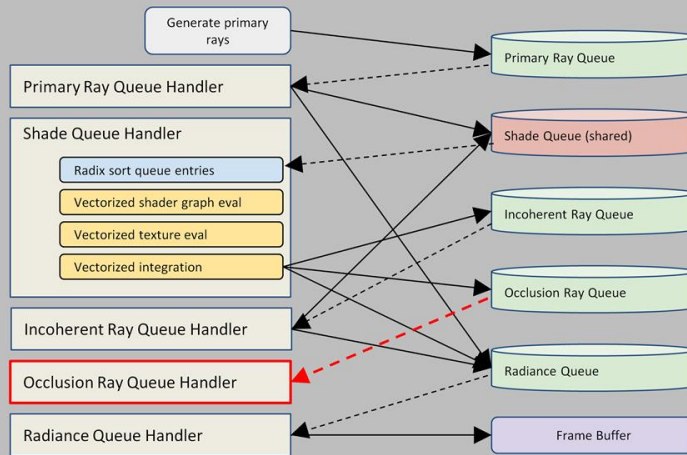
And just like the primary ray queue handler, it generates results which either are sent through to the appropriate shade queue or to the radiance queue. This path is how we mimic depth first recursion in a breadth first context.

Vectorized Path Tracing



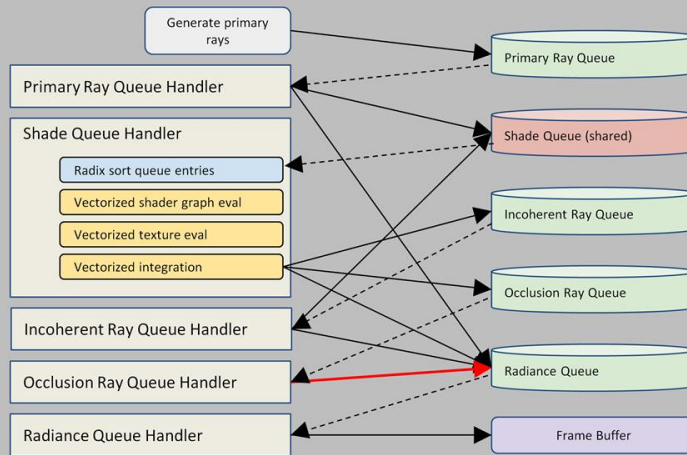
The final type of output generated from the integrator are occlusion rays to lights. There are no dependencies on the rays, they are strictly fire-and-forget. The ray and the amount of radiance to add to the frame buffer in the case where the ray can see the light is stored in the queue.

Vectorized Path Tracing



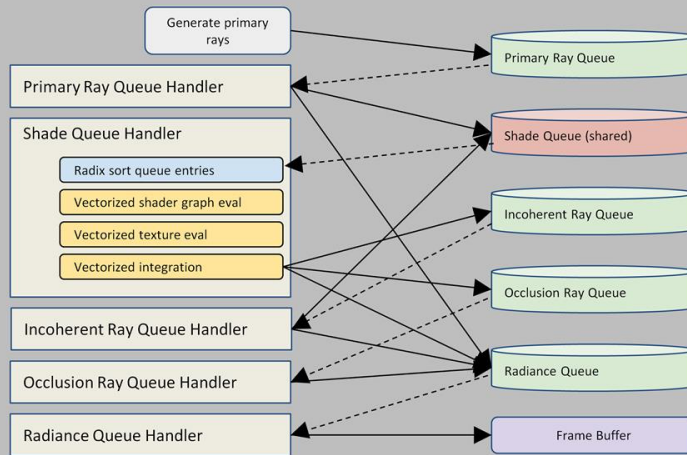
This queue, like all other queues, will get flushed at some future point in time, and all the light contributions will be accounted for.

Vectorized Path Tracing



Rays which pass the occlusion tests are added to the radiance queue.

Vectorized Path Tracing



And that is it, that's the way data flows in and out of queue and handlers for the vectorized code path.

Vectorization Results

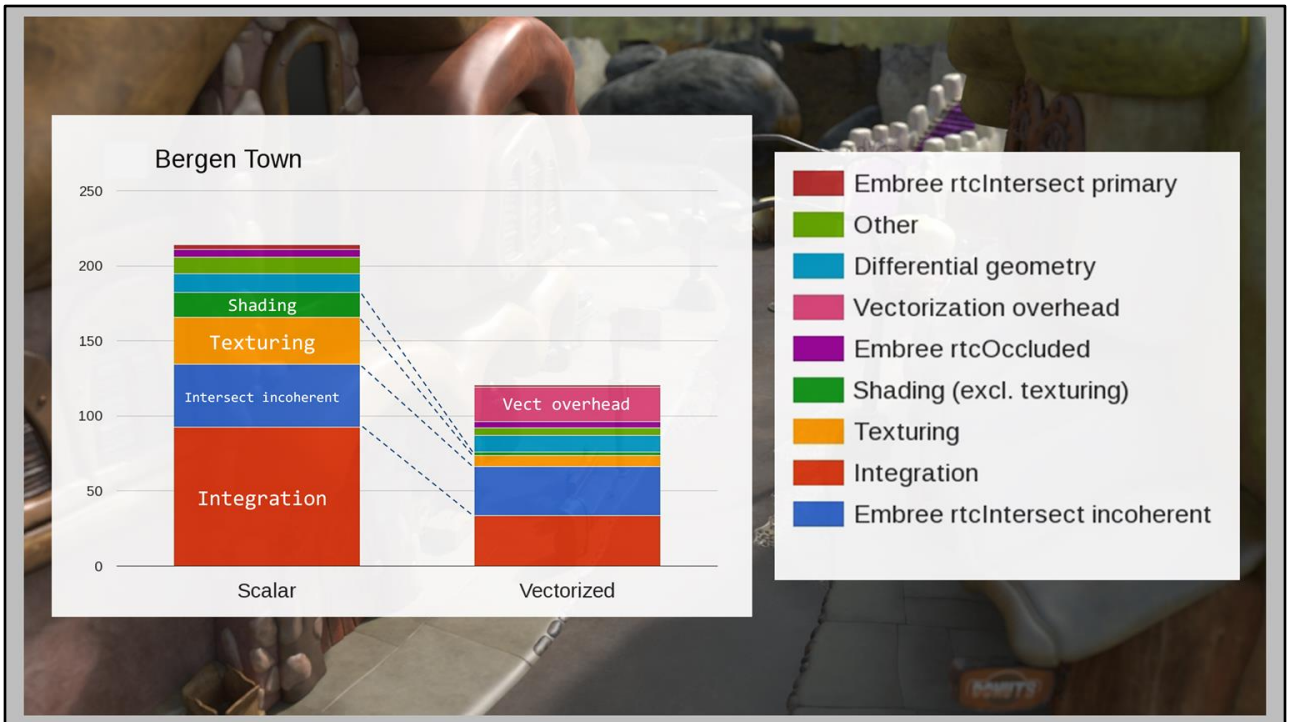
AVX2 (8 lanes wide)

Let's look at some scenes we've profiled.

We targeted the AVX2 instruction set for all of these tests, which is 8 lanes in width, allowing us to process batches of 8 rays or 8 shading samples simultaneously.



This is a render of Bergen Town from the movie Trolls.



The charts for this scene show that integration time dominates here.

Such scenes are a good case for the vectorized code path, since any scenes where we spend a good portion of time doing integration, shading, and/or texturing stand to see the most benefit.

Even though the vectorization overhead, the pink block in the top middle, stands out, we still observe a 77% overall speedup for this scene.

Bergen Town Vectorization Speedups

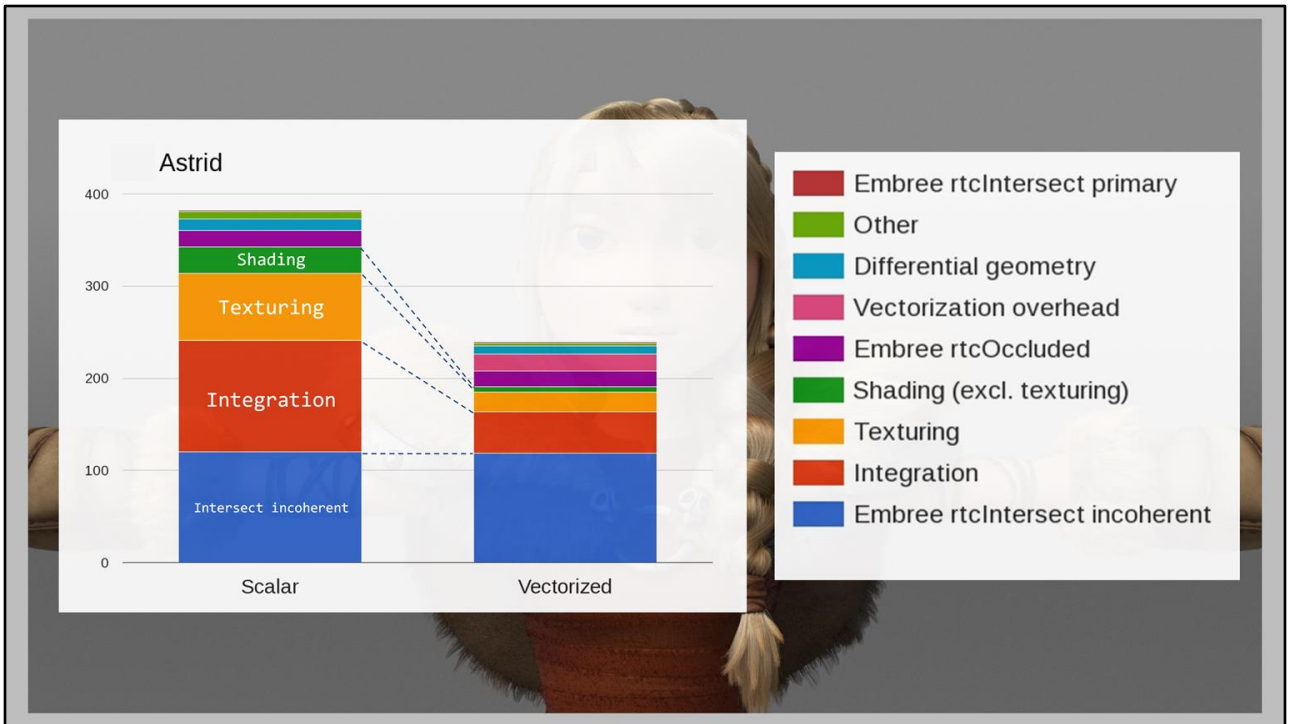
Ray intersection subsystem speedup	1.20x
Shading subsystem speedup	6.19x
Texturing subsystem speedup	4.24x
Integration subsystem speedup	2.75x
Overall speedup	1.77x

The shading speedup in particular stands out here.

Although the shaders themselves are relatively simple, it's nice to see over a 6x speedup for this scene.



Next is the character Astrid from the film How to Train your Dragon II.



There is a lot of fur and hair in this scene and the biggest block of time, at the bottom in blue, is spent inside of embree.

Since embree is already well vectorized in both our Scalar and Vectorized code paths, we don't expect an improvement in this area.

However, we observe significant gains in shading, texturing, and integration, resulting in a 60% gain overall.

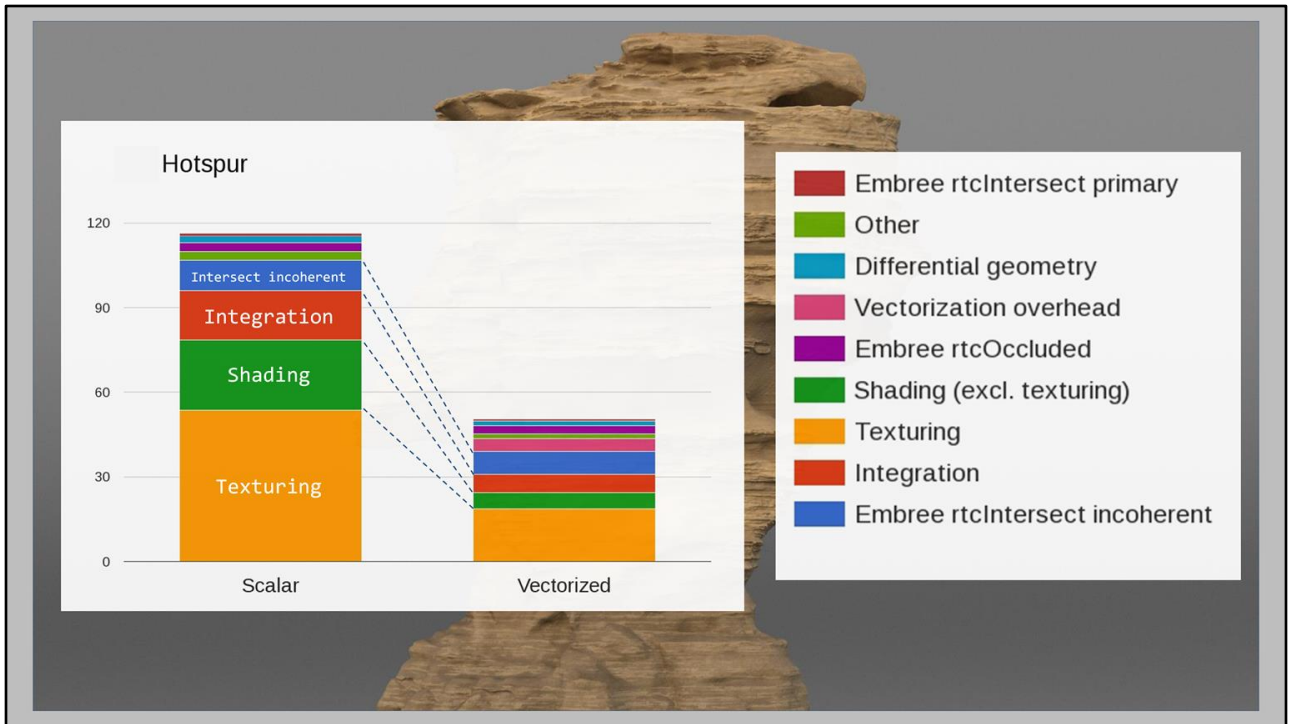
Astrid Vectorization Speedups

Ray intersection subsystem speedup	1.00x
Shading subsystem speedup	4.54x
Texturing subsystem speedup	3.43x
Integration subsystem speedup	2.68x
Overall speedup	1.60x

And here is the summary for Astrid.



Unfortunately we didn't get permission to show any of the more complex scenes at the studio since these are being generated for unreleased movies. I did want to show this particular asset however since it's an example of where texturing and shading dominates.



This is a best case scenario for us. In this case Texturing and Integration speed up by nearly 3x, and shading by over 4x. Given us over a 2.3x overall performance improvement.

Hotspur Vectorization Speedups

Ray intersection subsystem speedup	1.14x
Shading subsystem speedup	4.20x
Texturing subsystem speedup	2.90x
Integration subsystem speedup	2.72x
Overall speedup	2.31x

And here are the results.



Final Thoughts

MoonRay is just beginning its journey

Development focus and personality has been performance

Production specific features (such as AOVs) growing

Vectorization extensions for volume rendering and bi-directional path tracing (see you next year!)

Good Luck MoonRay and MoonShine teams!

I'd now like to share some final thoughts. In development for four years, MoonRay has just embarked, within the last 6 months, on its life as the primary production renderer at DreamWorks. Our primary focus to date has been performance, but production features are catching up. The challenge is to maintain a high level of performance, while growing this feature set. There are still challenges on the vectorization front, including extensions for volume rendering and bi-directional path tracing. Would love to chat with any of you about ideas on this.

Finally, I'd like to thank the Moonray and Moonshine teams at DreamWorks for being such awesome folks to work with. Good Luck. And of course a BIG Thank-you to all of you for your time and attention today. It has been an honor.



Thank-You!

SIGGRAPH 2017

Thank-you for your time and attention.