Physical Production Shaders with OSL

Using Open Shading Language at Sony Pictures Imageworks

Adam Martinez

Senior Shader Writer

Sony Pictures Imageworks

I am Adam Martinez, a shader writer at Imageworks and I will be filling in the gaps between now and the last time I presented this course in 2010. Specifically, I'm going to discuss the Open Shading Language and its use at Sony Pictures Imageworks.

What is Open Shading Language?

OSL is a shading language specification
and
an open source implementation of that specification

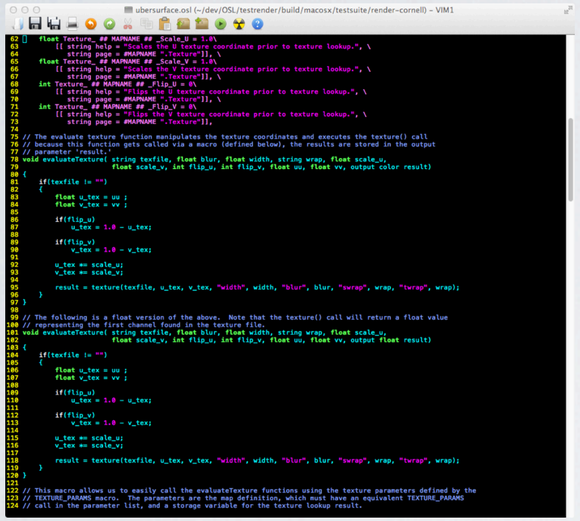
- https://github.com/imageworks/OpenShadingLanguage

SONY PICTURES imageworks
20th Anniversary

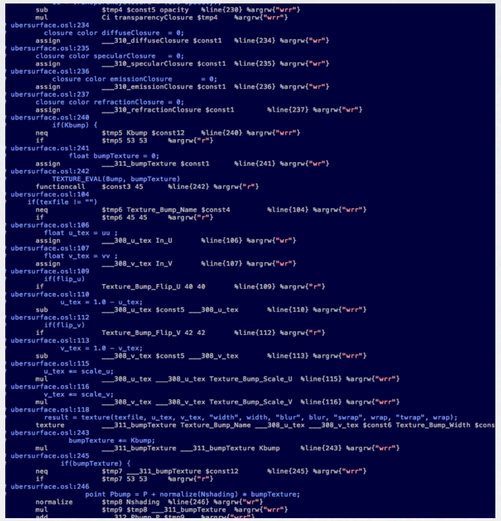
OSL is two things.
It is a shading language specification that describes a simple but robust set of functions and syntax rules for defining shaders.

OSL is also an open source project that is an implementation of this language specification.

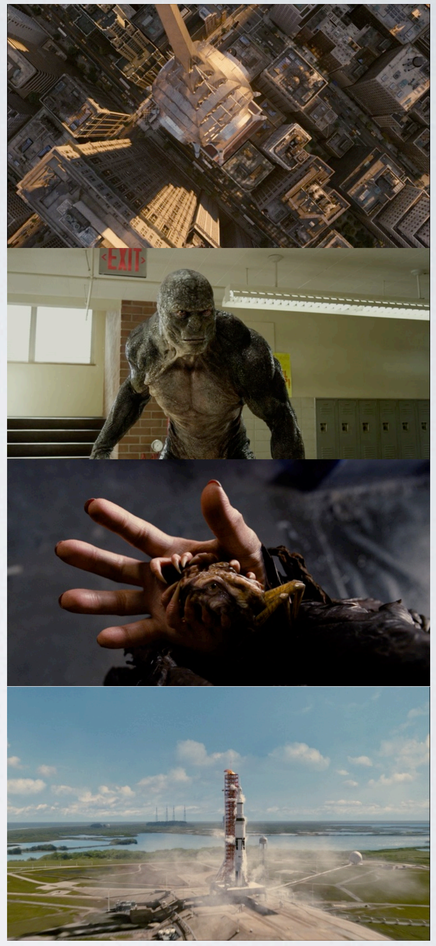
What is Open Shading Language?



oslc



render



- https://github.com/imageworks/OpenShadingLanguage

SONY PICTURES imageworks

20th Anniversary

Practically speaking, OSL works like many other shading languages: OSL Shaders are written as text files and compiled into intermediate byte-code by the oslc executable. These intermediate files are passed to the OSL-enabled renderer for optimization, execution and integration. And out the other end comes the blockbuster movie of your choice.

The project is hosted on GitHub and available at this URL.

What is Open Shading Language?

SONY PICTURES imageworks

20th Anniversary

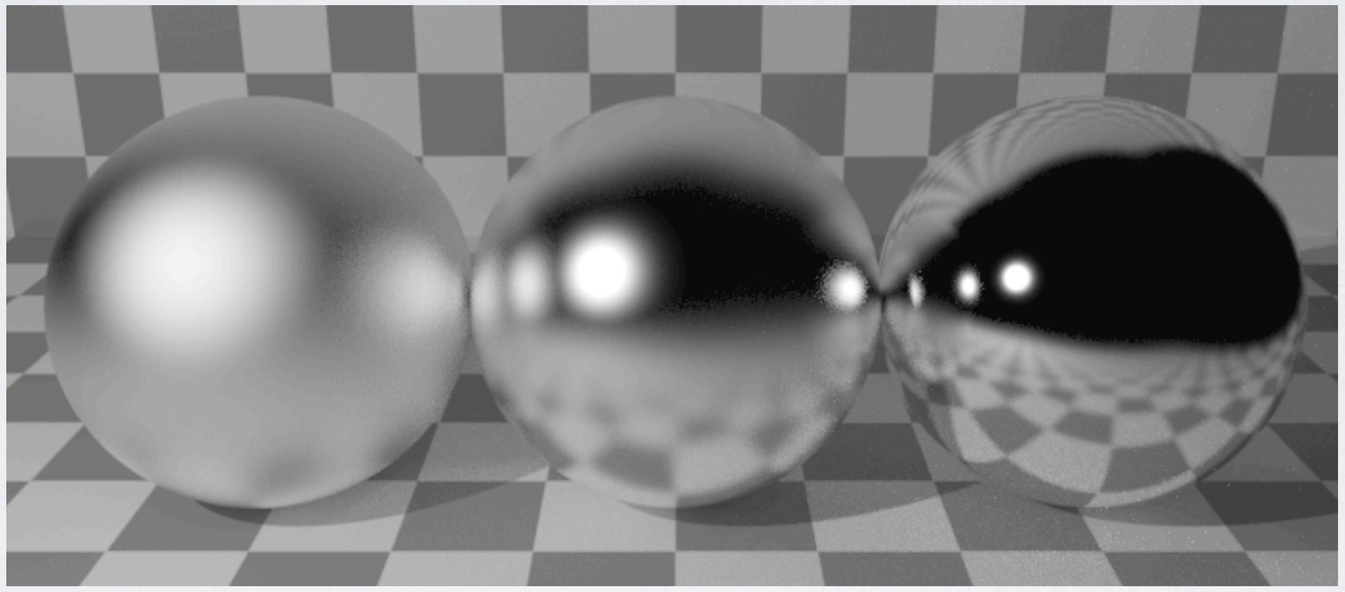
At Sony Pictures Imageworks we use OSL exclusively.

The first two feature film productions using this system were “Men in Black 3” and “The Amazing Spider-Man.” The visual effects for both of these films were rendered entirely using Imageworks’ OSL-enabled version of the Arnold renderer.

“Hotel Transylvania” opening next month will be the first animated feature rendered entirely with OSL.

What is Open Shading Language?

```
Ci = microfacet_beckmann(N, Roughness, eta);
```



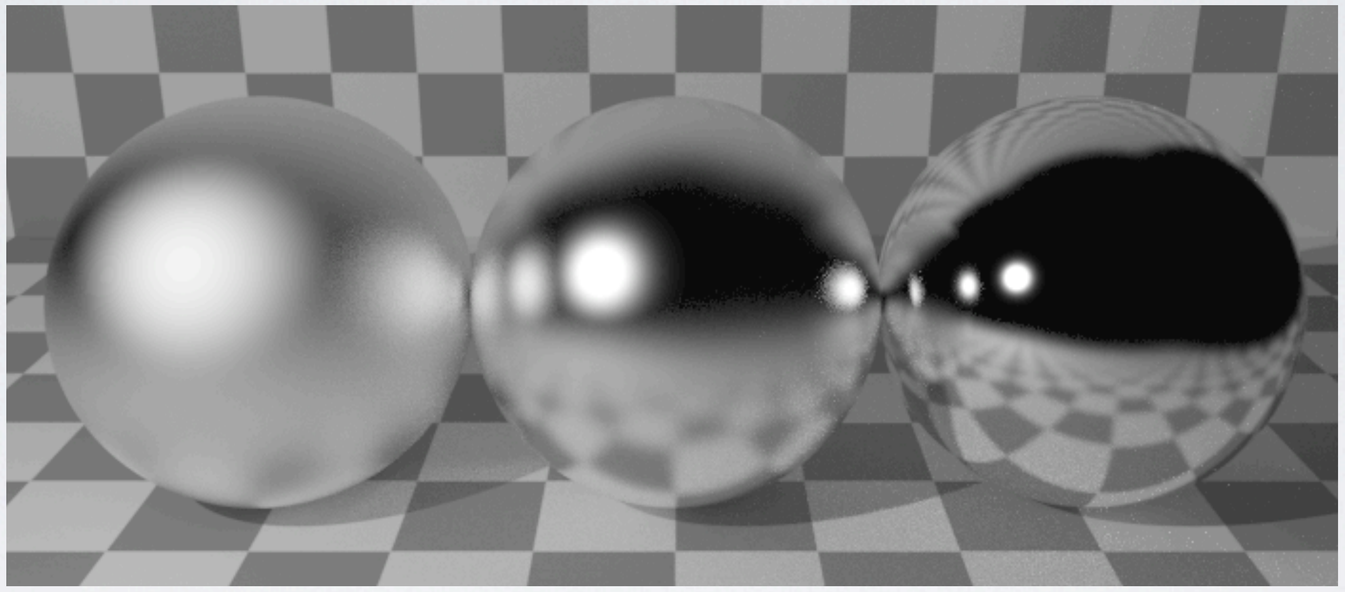
SONY PICTURES imageworks

20th Anniversary

This is as simple as it gets. This one-line example of actual OSL code calls a built in Cook Torrance microfacet specular model using the Beckmann distribution. The image is the result of three calls to this with varying roughness values. This image was rendered with the testrender demonstration application, available in the OSL project. With this one line of code, we see a number of things happening. Both direct illumination from light sources and indirect light from surrounding surfaces are being rendered with this single statement. There are no explicit calls to light shaders, or trace requests. The entire model is packaged up into a single function that is a native part of the OSL implementation.

How OSL Works

But how do we get from this:



SONY PICTURES imageworks

20th Anniversary

So that’s nice, but how does that translate into feature film visual effects production? How do we get from these simple spheres...

How OSL Works

To this:

SONY PICTURES imageworks

20th Anniversary

...to a rampaging half-man-half-monitor lizard rendered using high dynamic range set measured luminance data, composited into a live action plate with correct interactions? Well, let us first look at how OSL works.

How OSL Works

- Built on physically based shading principles
  - Physically correct units of radiance, $W/m^2/sr$
- Materials are arranged in networks of shaders
  - Networks are aggressively optimized at run-time
- Abstracts the shading from the integration
  - The renderer is free to choose an integration method

SONY PICTURES imageworks

20th Anniversary

OSL starts by making a lot of often-used production paradigms first class concepts in the system. It incorporates physically based shading principles both in the conceptual design of the language, and in the practical execution of the shaders. OSL computes radiance values, as watts per square meter per steradian. Networking shader nodes are a native aspect to the system, and supported as efficiently as monolithic shaders. OSL is designed to abstract the material description from the rendering algorithm itself. The renderer should be able to reason about and make smart decisions about how to integrate results based on what is being asked for by the shader. And key to this idea...

How OSL Works

- OSL shaders return “closures”
  - The closure is an object that describes the procedure(s) for evaluating the BSDF
    - closure implementation is up to the renderer
  - Shaders do not directly evaluate lights or sample the scene
    - View independent, mostly
    - No trace, mostly
    - No light loops

```
Ci = microfacet_beckmann(N, Roughness, eta);
```

SONY PICTURES imageworks

20th Anniversary

...is that shaders do not return final pixel values. Shaders do not sample the lights, cast rays into the scene or any of that. Rather they return closures, a symbolic representation of, among other things, how a surface scatters light. In our previous example microfacet_beckmann is a closure, it is not a function that is being evaluated at that point, but rather a set of functions that help the renderer solve for the desired look. Closures can do other interesting things too such as BSSRDF evaluation and utility shadow matte generation. Transparency is also a closure.

No Light Loops!?

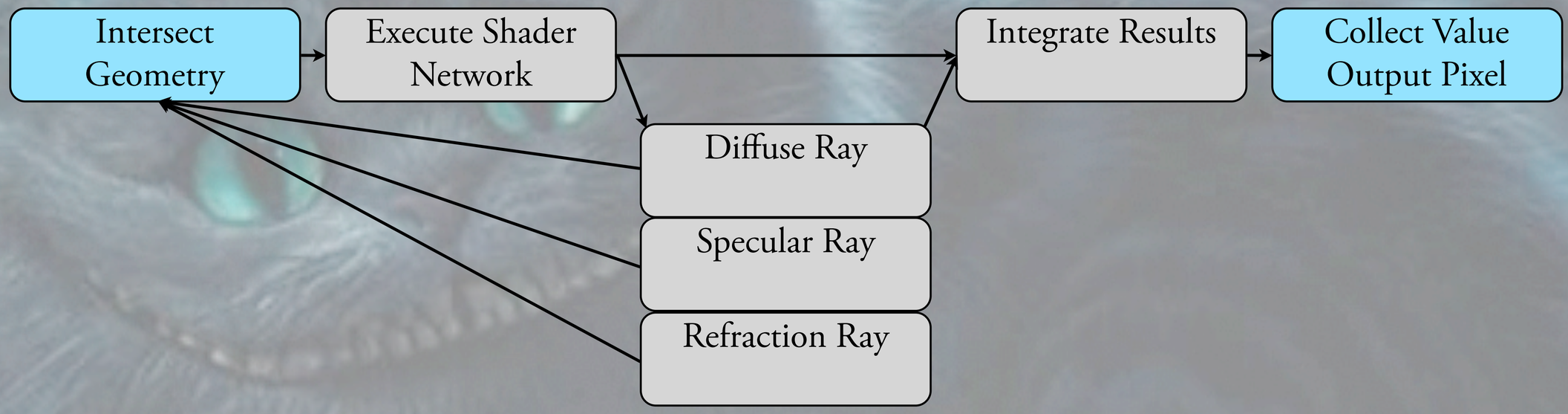
SONY PICTURES imageworks

20th Anniversary

This is what aggravated shader writers look like. No light loops, no trace calls? That's usually an enormous part of what a shader DOES isn't it? What does a shader do if it does not integrate the lighting in a meaningful way? Let's look at why OSL works the way it does, and why it proposes to remove these and other capabilities from within the shader.

Alice, Arthur and Smurfs

A typical shader execution model



SONY PICTURES imageworks

20th Anniversary

During production of *Alice in Wonderland, Arthur Christmas, The Smurfs* and *Green Lantern*, we used a shader execution model in our renderer that looked pretty much like this.

The blue boxes are parts of the pipeline handled by the renderer, and the grey boxes are handled in the shader.

The render would start, trace a ray from the camera, hit a surface and run the shader on that surface. At that point, a plug-in external to the renderer, the shader, took over almost entirely. The shader would execute its code in sequence, casting diffuse and specular rays, hitting other geometry and recursively solving for a final pixel value. At a certain point the renderer becomes little more than a memory structure for a shader to bounce around in.

Obviously this is a simplistic view of what is actually going on, since much of a shader will use renderer APIs to varying degrees. But it illustrates how much impact the shader has on the rendering pipeline in this model.

How OSL Works

Problems with this model:

–The shader is a black box

–The renderer's evolution is impacted

–Shader maintenance is difficult

SONY PICTURES imageworks

20th Anniversary

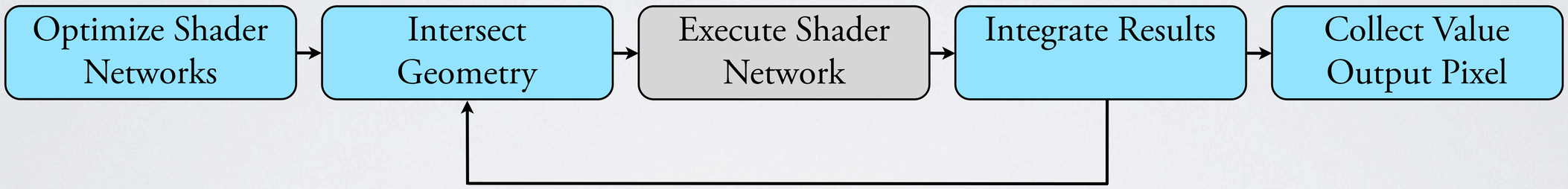
We realized that even in a networked shading environment this model presents many problems. First and foremost, the renderer cannot make intelligent decisions about the sampling strategy, given that it is locked out of shader execution. The shader is a black box to the renderer.

Renderer developers cannot explore alternative integration strategies such as bi-directional path tracing or metropolis light transport, without fear of breaking downstream shader library investments. In large part, shaders would have to be rewritten entirely if the renderer-side integration methods changed, and certain shader features may not even be portable to those methods.

Various API calls expose internals of the renderer which in turn expose the renderer to instability and artifacts such as NaNs. Shaders may be doing some very interesting things, but as they evolve separately from the renderer, maintenance becomes a problem. API changes can cause shader incompatibility.

How OSL Works

The OSL shader execution model



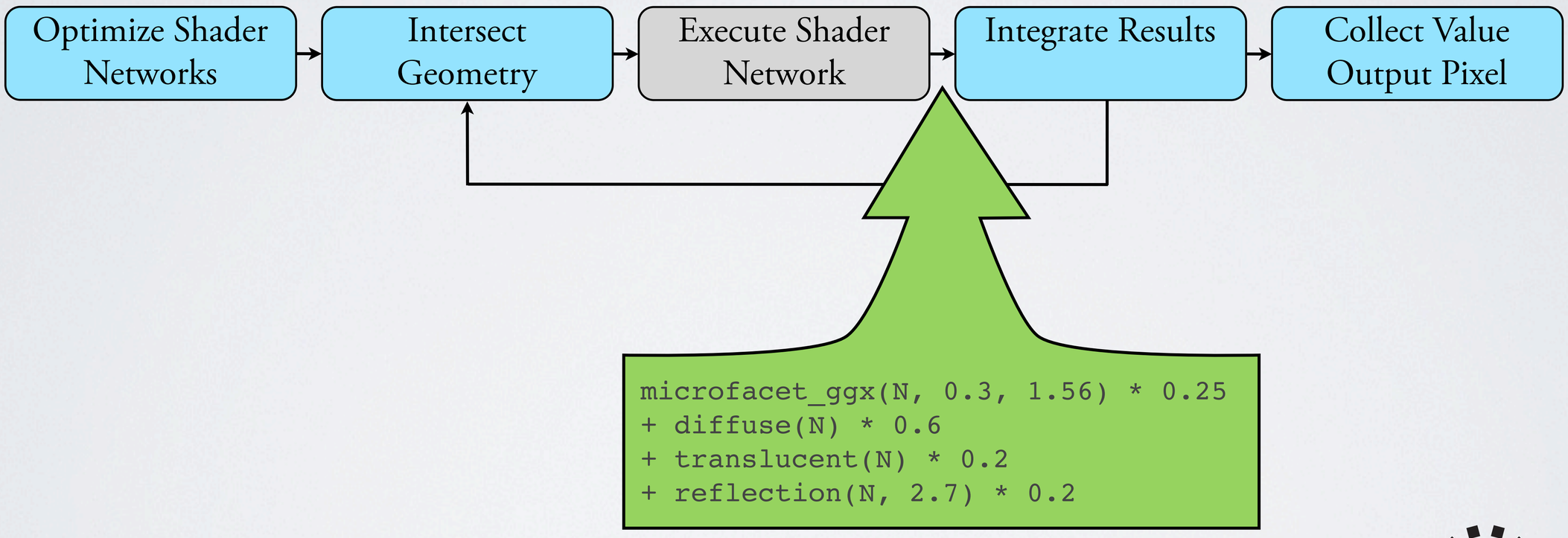
The OSL execution graph looks, albeit very naively, like this.

There is now an optimization step when a render begins, I'll discuss that in further detail shortly.

The shader execution stage is now one box, and the recursion step happens under the control of the render. The integration step executes the light loops and sampling loops, which recurse back to the intersection step. So what is this shader doing? When the shader is executed it returns...

How OSL Works

The OSL shader execution model



SONY PICTURES imageworks

20th Anniversary

a list of closures and weights which are passed to the renderer, where the OSL execution library takes them and calls the appropriate renderer API functions to solve the closures and integrate the results. The renderer can now analyze, evaluate and sample the closures or store them for later evaluation. Having the recipe for the description of the surface now means that the renderer can determine the best strategy for integration, and can do so very differently, if necessary, depending on context.

This separation of shading and integration is key to making multiple importance sampling work, but can also be used for interpolation, ray sorting, or rapid relighting.

How OSL Works

Run-Time Optimization:

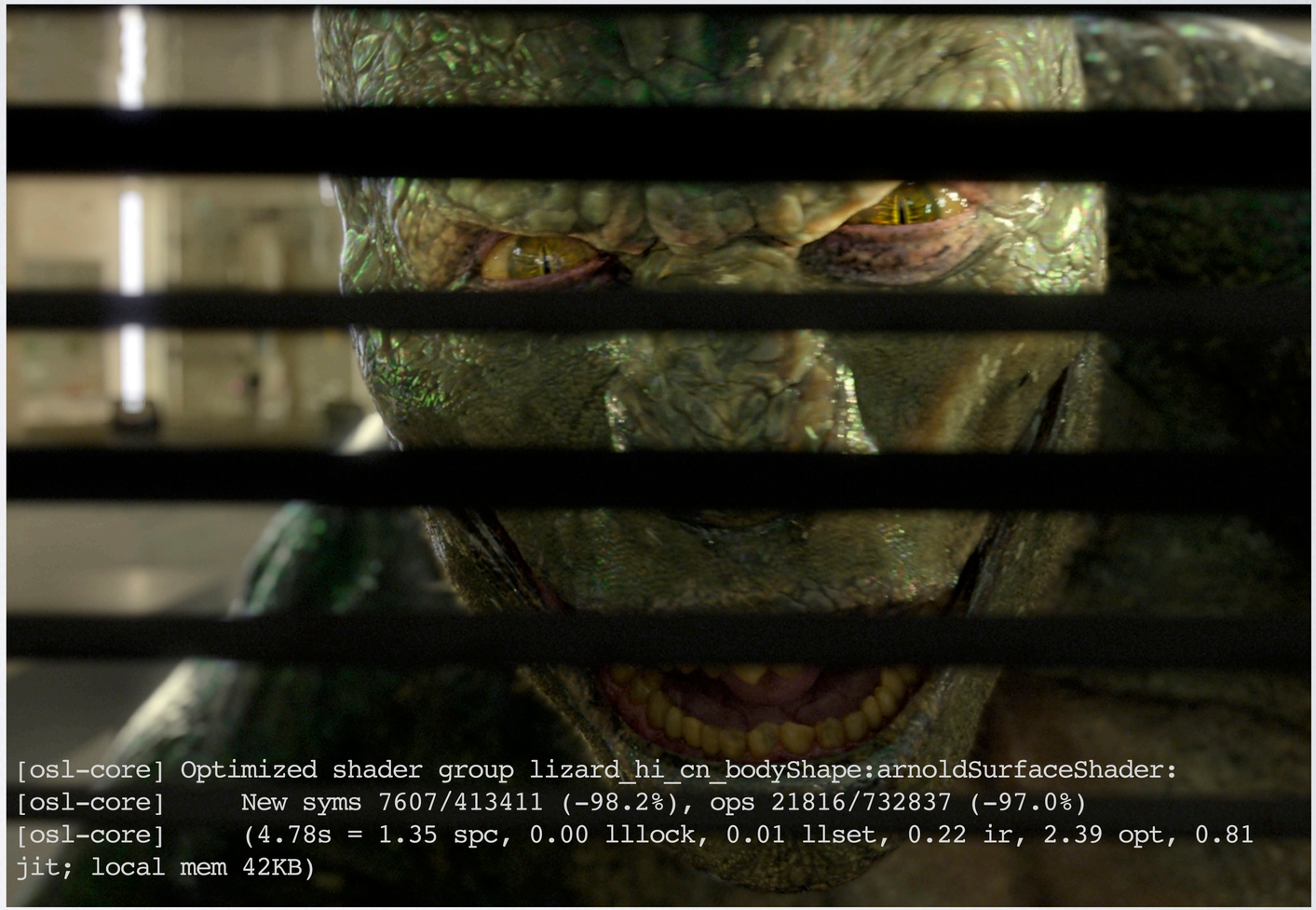
–Merges user parameters

–Constant folding

–Dead-code elimination

–Strength reduction

–Works across networks



SONY PICTURES imageworks

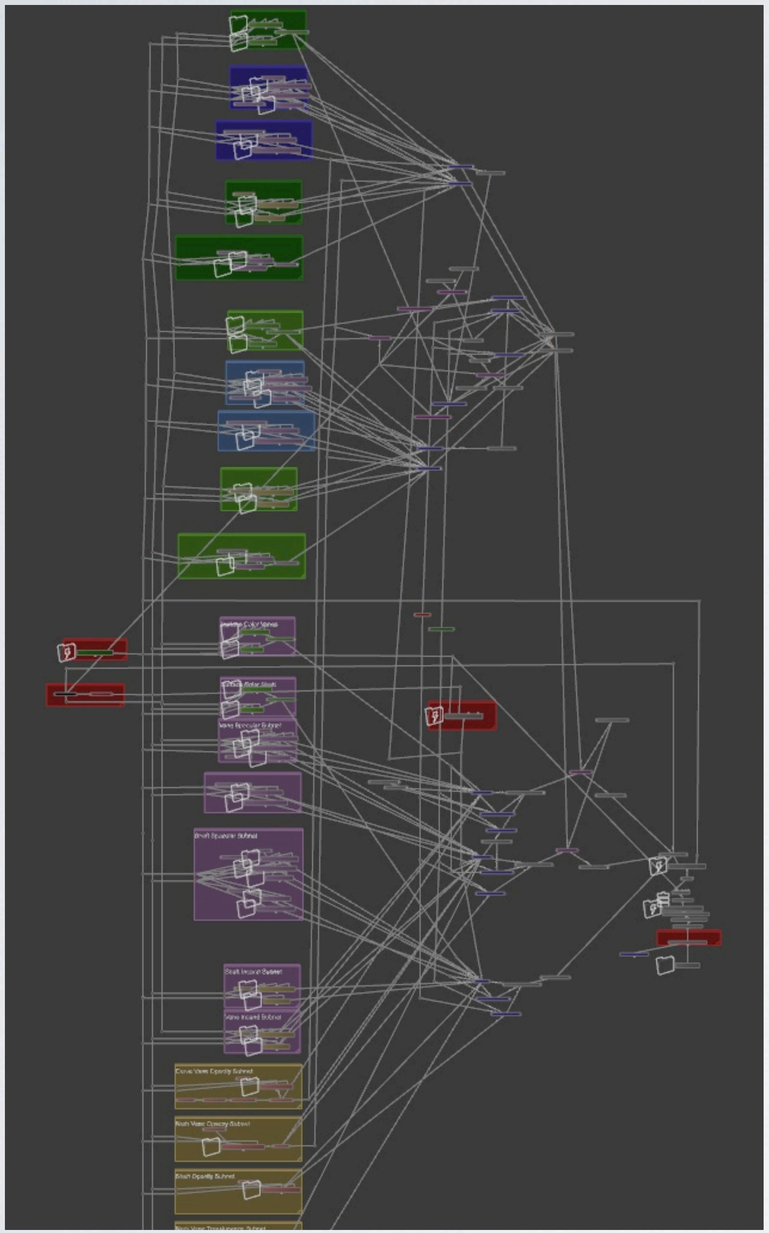
20th Anniversary

The run-time optimization step is a large part of OSL's efficiency during execution. At the bottom of the image in this slide is the optimization output of OSL during a production render of the Lizard from "The Amazing Spider-Man." This particular shader network is applied to the Lizard's body. This tells us that the original shader network, made up of over 700,000 operations, was reduced at run time to a little over 21,000. This is a 97 percent reduction in potential operations.

I should note here that the optimization is a feature of the open source project implementation of OSL, it is not described in or a requirement of the OSL shading language specification.

The optimizer runs at a point in the rendering pipeline where the renderer knows about all of the non-geometric properties of the material, such as user parameters and shader network topology. Using this information the optimizer can aggressively eliminate dead code paths, collapse operations that result in constant values, and run strength reduction operations.

How OSL Works



Becomes

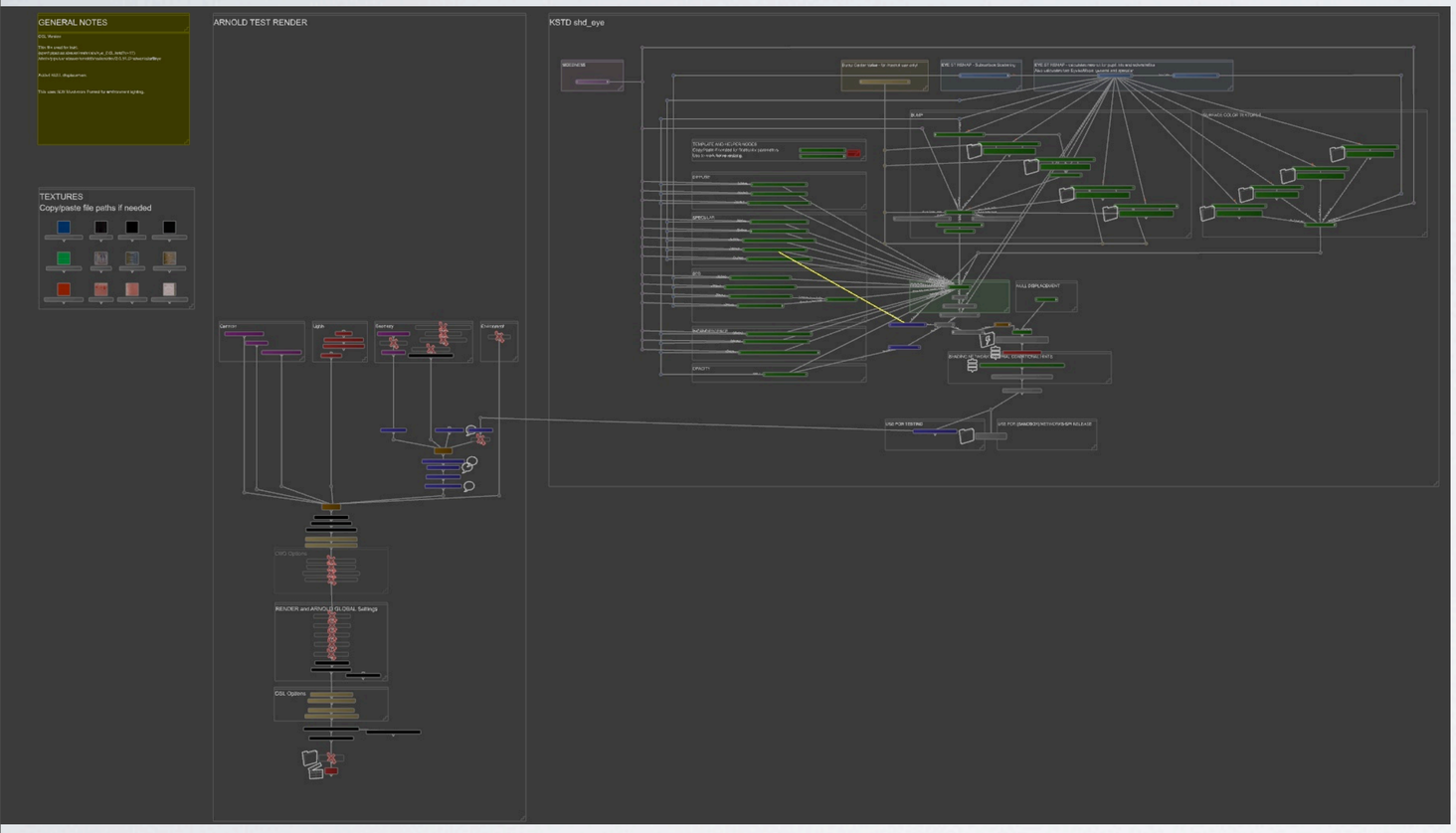
```
Ci = microfacet_beckmann(N, 0.1, 1.3) * 0.5;
```

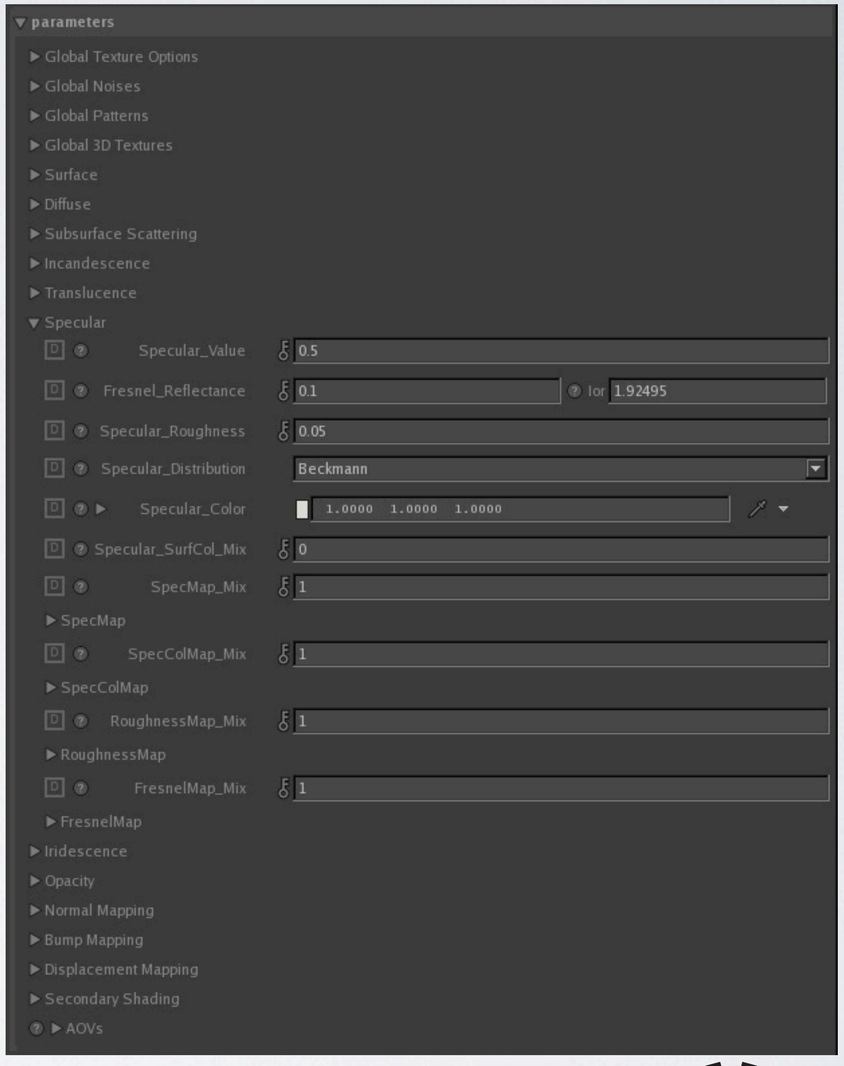
SONY PICTURES imageworks

20th Anniversary

Generally we see that level of optimization at SPI across the board, regardless of network complexity.

How OSL Works





SONY PICTURES imageworks

20th Anniversary

At Imageworks our networks are fairly sizable to begin with. Any single shading network which describes a single material can contain upwards of 150 separate nodes (most of them for texture reading). On top of that, these networks can be layered arbitrarily, and procedurally extended, increasing the node count. On the left is an example of one of the shader networks we use, as it is laid out in the Katana application. On the right is a common user-interface for such a shader network. At Imageworks, the shader networks are presented to the user as a unified interface of parameters laid out in discrete pages or groups. Much of the network is dedicated to texturing and pattern generation.

Even under extreme circumstances, the OSL optimization steps make it possible, even efficient, to hand our artists shading tools that maximize capability and flexibility, that in previous generations had more severe costs associated with them.

Physically Based Shading and Lighting



Physical units allows us to seamlessly change lighting paradigms

SONY PICTURES imageworks

20th Anniversary

I mentioned earlier the physically based units of radiance that OSL uses.

This consistency is what easily allows users of OSL to freely interchange CG light sources, captured HDRI environments, and emissive geometry.

And this is what enables our image-based lighting pipeline at Imageworks. Lighting artists are able to evaluate an acquired lighting environment from the film set and graphically determine areas of important illumination information.

These areas can then be seamlessly and non-destructively extracted and promoted to either emissive geometry, or direct CG light sources. This allows our CG characters to respond to illumination accurately and dynamically as they move through the environment.

A Simple Shader

```
surface
example_shader_1
  [[string help = “Simple texture mapped diffuse material”]]
(
  string texture_name = “”
    [[string help = “A texture file name”]]
)
{
  color paint = texture(texture_name, u, v);
  Ci  = paint * diffuse(N);
}
```

SONY PICTURES imageworks

20th Anniversary

Let’s look at a very simple shader example.

This shader returns a texture mapped diffuse surface.

We can see basic components of the OSL shader: we have the shader type declaration, surface, the parameter block, and the body of the shader. In blue we have the metadata, which can store help text or user interface hints.

In orange, the OSL functions, and in green the OSL keywords.

This shader is amazingly simplistic, and yet when executed and integrated, it will collect all of the incoming radiance from direct and indirect sources and weight that result by a value read in from a texture map. In other shading architectures, this same shader would be considerably longer.

A Physically Plausible Glass Shader

```
surface
example_shader_2
   [[ string help = “A better dielectric material” ]]
(
   color Cs = 1
      [[ string help = “Base Color”,
         float min = 0, float max = 1 ]],
   float roughness = 0.05
      [[ string help = “surface roughness”]],
   float eta = 1.5
      [[ string help = “Index of refraction” ]],
   int enable_tir = 1
      [[ string help = “Enables total internal reflection”]],
)
{
   if( backfacing() )
   {
      Ci = microfacet_beckmann_refraction(N, roughness, 1.0 / eta );

      if( enable_tir)
      {
         Ci += microfacet_beckmann(N, roughness, 1.0 / eta);
      }
   } else {
      Ci = microfacet_beckmann_refraction(N, roughness, eta) +
           microfacet_beckmann(N, roughness, eta);
   }
   Ci *= Cs;
}
```

SONY PICTURES imageworks

20th Anniversary

What about a more complicated example?

Glass is usually an area where a lot of tricks are employed to accomplish particular looks.

Creating a physically plausible glass shader that is also production viable can be especially daunting. In this example shader, we add together a series of closures to create the particular combination of reflection and refraction based on a number of conditions, chiefly the direction of the surface normal relative to the view direction and the user optional total internal reflection.

A Physically Plausible Glass Shader

```
surface
example_shader_2
   [[ string help = "A better dielectric material" ]]
(
   color Cs = 1
      [[ string help = "Base Color",
         float min = 0, float max = 1 ]],
   float roughness = 0.05
      [[ string help = "surface roughness"]],
   float eta = 1.5
      [[ string help = "Index of refraction" ]],
   int enable_tir = 1
      [[ string help = "Enables total internal reflection"]],
)
{
   if( backfacing() )
   {
      Ci = microfacet_beckmann_refraction(N, roughness, 1.0 / eta );

      if( enable_tir)
      {
         Ci += microfacet_beckmann(N, roughness, 1.0 / eta);
      }
   } else {
      Ci = microfacet_beckmann_refraction(N, roughness, eta) +
           microfacet_beckmann(N, roughness, eta);
   }
   Ci *= Cs;
}
```

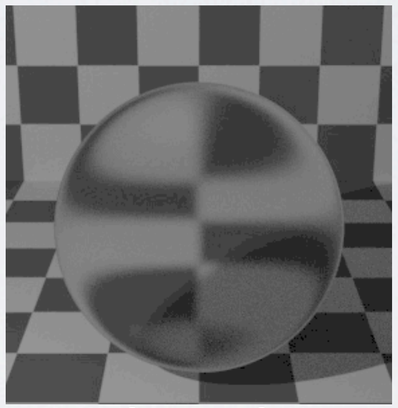
SONY PICTURES imageworks

20th Anniversary

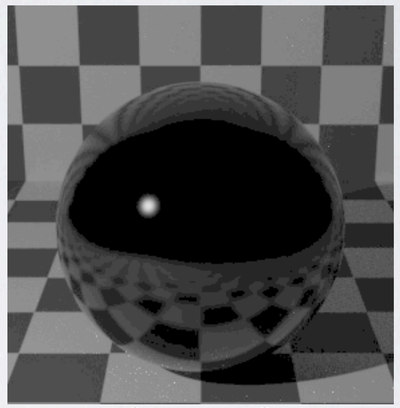
The backfacing function will tell us if the surface currently being shaded is facing away from the view direction or not. Let's start with the backfacing is false case.

A Physically Plausible Glass Shader

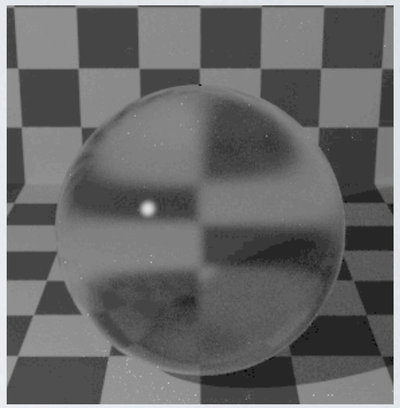
```
surface
example_shader_2
   [[ string help = “A better dielectric material” ]]
(
   color Cs = 1
      [[ string help = “Base Color”,
         float min = 0, float max = 1 ]],
   float roughness = 0.05
      [[ string help = “surface roughness”]],
   float eta = 1.5
      [[ string help = “Index of refraction” ]],
   int enable_tir = 1
      [[ string help = “Enables total internal reflection”]],
)
{
   if( backfacing() )
   {
      Ci = microfacet_beckmann_refraction(N, roughness, 1.0 / eta );

      if( enable_tir)
      {
         Ci += microfacet_beckmann(N, roughness, 1.0 / eta);
      }
   } else {
      Ci = microfacet_beckmann_refraction(N, roughness, eta) +
           microfacet_beckmann(N, roughness, eta);
   }
   Ci *= Cs;
}
```



microfacet_refraction



microfacet_reflection



result

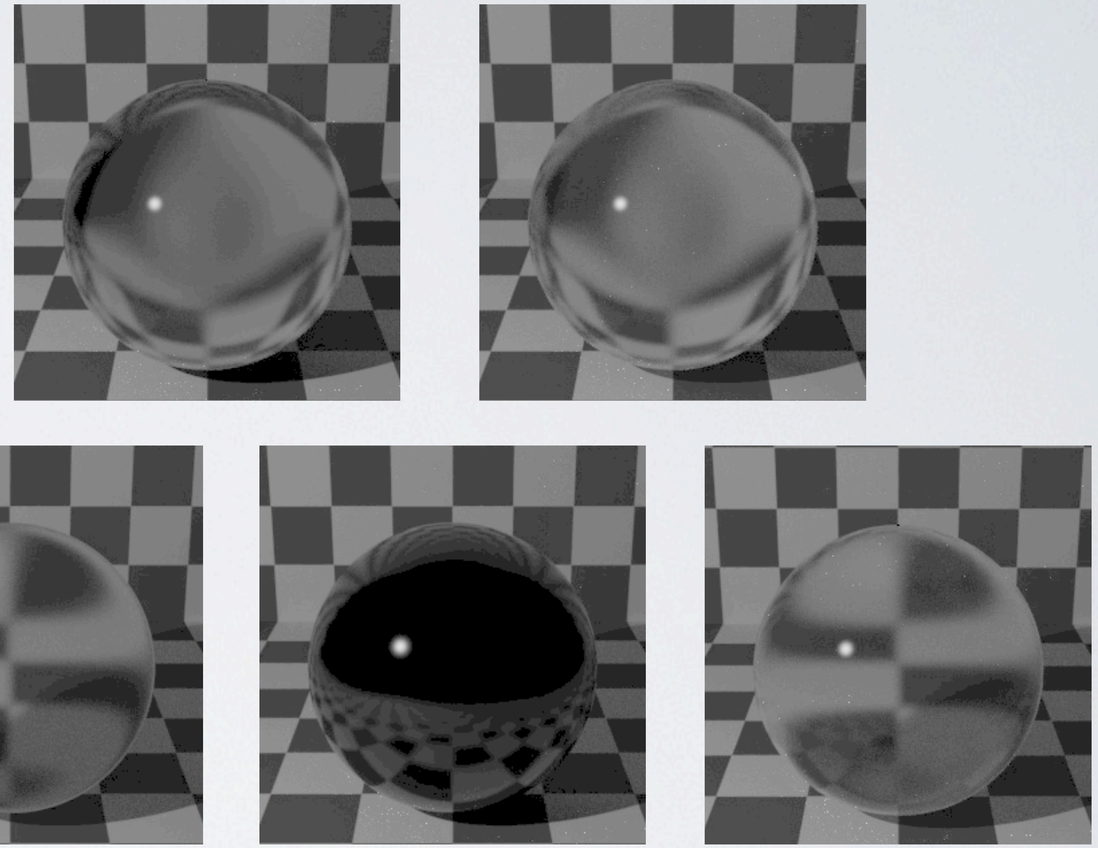
SONY PICTURES imageworks

20th Anniversary

This means that the surface normal is pointing towards the view direction, and we are entering the refractive object. What we want to do is add both a refraction and a reflection closure with the same parameters. These two will combine to form an energy-conserving formulation of a dielectric material. On the far left image, we see a render of just the refraction closure. In the middle, the reflection closure, and on the right the summed result of both of these closures. We are only computing, literally, one side of the solution however.

A Physically Plausible Glass Shader

```
surface
example_shader_2
   [[ string help = “A better dielectric material” ]]
(
   color Cs = 1
      [[ string help = “Base Color”,
         float min = 0, float max = 1 ]],
   float roughness = 0.05
      [[ string help = “surface roughness”]],
   float eta = 1.5
      [[ string help = “Index of refraction” ]],
   int enable_tir = 1
      [[ string help = “Enables total internal reflection”]],
)
{
   if( backfacing() )
   {
      Ci = microfacet_beckmann_refraction(N, roughness, 1.0 / eta );

      if( enable_tir)
      {
         Ci += microfacet_beckmann(N, roughness, 1.0 / eta);
      }
   } else {
      Ci = microfacet_beckmann_refraction(N, roughness, eta) +
           microfacet_beckmann(N, roughness, eta);
   }
   Ci *= Cs;
}
```



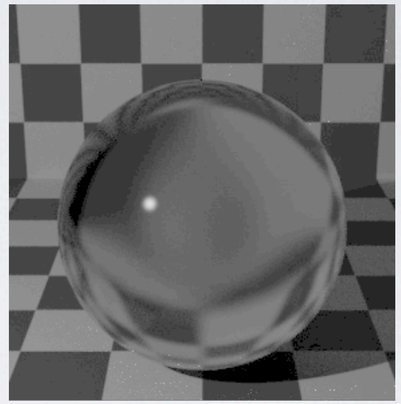
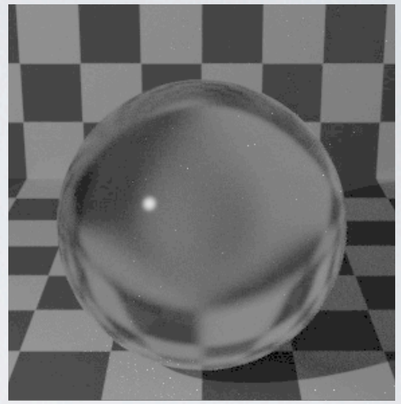
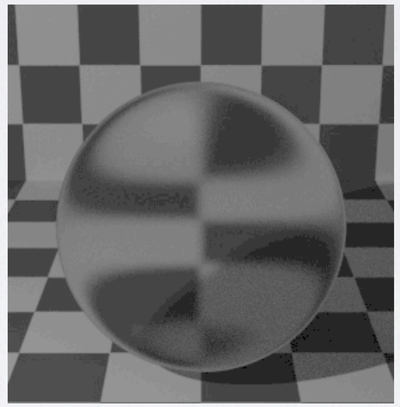
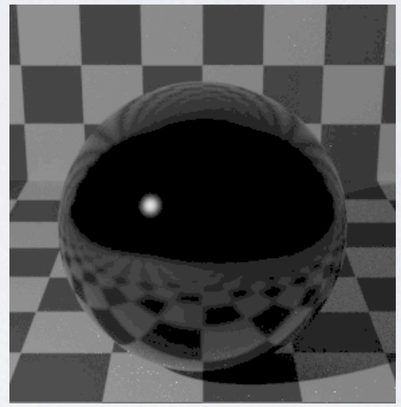
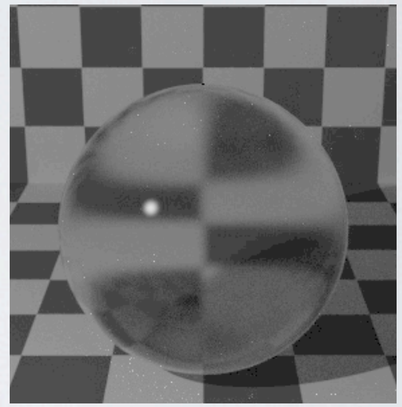
SONY PICTURES imageworks

20th Anniversary

In the case that backfacing is true, the surface normal is pointing away from the view direction, and we are exiting the refractive object. We basically want to do the same thing as the forward-facing case, but with an inverted IOR. We have also added a switch to enable the inclusion of the internal reflection closure, since this effect can be expensive and in production is frequently sacrificed. On the top row of images you can see the total internal reflection missing on the left image, as compared to the right image.

A Physically Plausible Glass Shader

```
surface
example_shader_2
   [[ string help = “A better dielectric material” ]]
(
   color Cs = 1
      [[ string help = “Base Color”,
         float min = 0, float max = 1 ]],
   float roughness = 0.05
      [[ string help = “surface roughness”]],
   float eta = 1.5
      [[ string help = “Index of refraction” ]],
   int enable_tir = 1
      [[ string help = “Enables total internal reflection”]],
)
{
   if( backfacing() )
   {
      Ci = microfacet_beckmann_refraction(N, roughness, 1.0 / eta );

      if( enable_tir)
      {
         Ci += microfacet_beckmann(N, roughness, 1.0 / eta);
      }
   } else {
      Ci = microfacet_beckmann_refraction(N, roughness, eta) +
           microfacet_beckmann(N, roughness, eta);
   }
   Ci *= Cs;
}
```

SONY PICTURES imageworks

20th Anniversary

These basic code examples should give you an impression of how rapidly shaders in OSL can be developed.

In the C shading system of the recent past, shader builds were inextricably tied to the major renderer versions they were compiled against. Every time the renderer API was updated, the entire shader library had to be rebuilt, and then tagged for a minimum renderer compatibility requirement.

Now, our shader versioning system is almost entirely independent of the renderer version.

Rapid and virtually maintenance-free development continues to be one of the greatest benefits to using OSL in an aggressive production environment.

Shader Writing Post-Mortem

OSL removes a huge maintenance component...

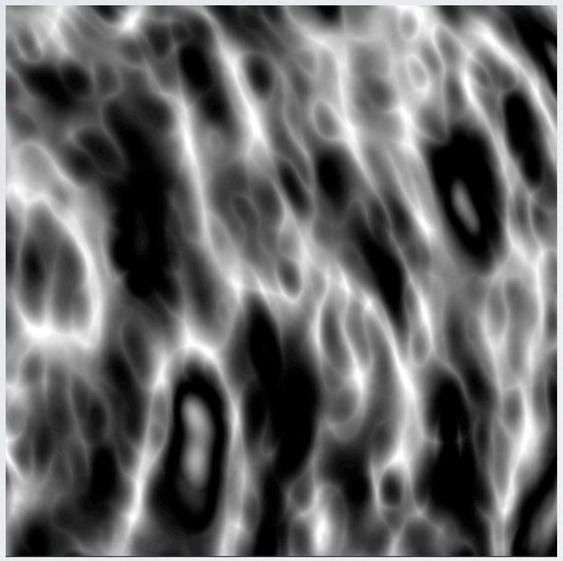
(...but OSL also limits capabilities inside the shaders themselves)

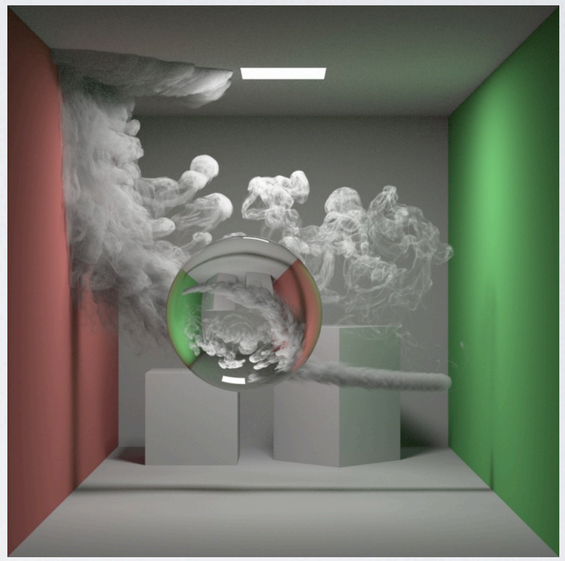
SONY PICTURES imageworks

20th Anniversary

As our first shows using OSL have wrapped, it is a good opportunity to offer a bit of a post-mortem on how the work of a shader writer has changed at Imageworks. OSL has allowed us to realize dramatically faster turnaround times on deployment and maintenance. However, the shaders, in and of themselves have become quite mechanical. They are now, largely, wrappers around functionality contained within the renderer. Honestly this was a tough pill to swallow, after previous generations of the shading system did so much work.

Shader Writing Post-Mortem





Solve harder problems!

SONY PICTURES imageworks

20th Anniversary

But our shaders required so much dedication to maintaining the very foundations of our rendering pipeline. A lot of time was dedicated to managing the production BSDFs with evolving renderer technology.

With OSL, we essentially have that problem ‘licked’ and we can focus on solving more interesting and sometimes more difficult problems, such as BRDF fitting to measured data, biological iridescence, new pattern generation methods, and global illumination in volumes. As a look developer the benefits have been huge.

Shader Writing Post-Mortem

SONY PICTURES imageworks

20th Anniversary

Render quality has never been more stable and consistent. In large part we are able to leverage new developments in the renderer almost immediately. Lizard is the culmination of a lot of work by a lot of very talented people working a lot of hours. From a development standpoint, productions benefit from the rapid turnaround time to feature additions and special-case shading needs that OSL brought in over the older C shading library. In addition, renderer capabilities such as ray traced subsurface scattering methods which were not available at the beginning of production, made their way into the shading system much faster, and much more transparently that previously.

So where do we go from here? The foundation of the OSL project continues to solidify and stabilize as it gets more production use at Imageworks. OSL is still quite young and some may find OSL lacking in certain conveniences common in modern programming languages. Many of these omissions are planned to be implemented, however they are not prioritized in light of the fact that they amount to no significant post-optimization benefit.

As more productions begin to use OSL and demand particular features to accomplish particular looks, the specification and the library will definitely expand. OSL has attracted the attention of a number of developers that have been exploring, experimenting and offering input into the growth of the project. Another exciting possibility on the horizon is a GPU implementation of OSL, which could allow for continuous material representation across preview and final-render contexts.

Thanks

Larry Gritz

Christopher Kulla

John Monos

Rob Bredow

Erik Strauss

the Imageworks Shading Department

the Imageworks Arnold Department

Dave Smith

John Haley

Brian Steiner

Marcos Fajardo and SolidAngle, LLC

my fellow course presenters and the organizers, Stephen Hill and Stephen McAuley

SONY PICTURES imageworks

20th Anniversary

Thank you for your time, and I'd like to throw out a special thanks to the people listed.