Physical Production Shaders with OSL

Using Open Shading Language At Sony Pictures Imageworks

SIGGRAPH 2012 Course Notes

Presented By

Adam Martinez
Sony Pictures Imageworks

Sony Pictures Imageworks, Culver City, CA

What is Open Shading Language?

Open Shading Language (OSL) is an open source project started by Larry Gritz at Sony Pictures Imageworks. OSL is a way of describing how surfaces interact with light using a simple but robust language that should be familiar to seasoned shader writers, and easy to learn for new users. OSL is built on a foundation of physically based rendering principles and designed to abstract the material description from the rendering method.

OSL is an open source project that implements the language specification. The project is a C++ library with a programming interface that can be incorporated into a renderer. This allows a renderer to employ any number of techniques to generate imagery, without necessitating intricate and complex shader maintenance. Many BSDF's and shading functions are written into the library, giving renderer developers a jump start with production proven shading models and pattern generation tools.

OSL is hosted on GitHub at https://github.com/imageworks/OpenShadingLanguage

Who is OSL for?

OSL is for Artists...

OSL shaders are easy to read, simple and elegant. Physically plausible materials can be described without much effort. Complex patterns and materials can be generated from a suite of tools including noise, splines and a variety of functions built into the language. The reduction in overall code devoted to shaders can be quite significant. For example, our root illumination shader at Imageworks went from thousands of lines of C code down to a few hundred lines, while retaining the same feature set and parametrization.

OSL is also safe to use. The ability to bring down a renderer or cause NaNs in an image is almost non-existent in OSL. When those events do occur, usually, the problem lands squarely on the shoulders of the renderer developers. This makes OSL extremely robust in production environments that work under very aggressive conditions.

Adding production utility functionality is equally simplified in OSL. AOV's, shadow manipulation, access to geometry variables are all built into the OSL specification. There is no loss to production versatility in using OSL.

OSL shaders can steer advanced rendering techniques without advanced level knowledge, and in many cases without re-development of the material description. This makes deployment and maintenance of shader libraries exponentially easier, and increases the cost-effectiveness of shaders over time. These shaders will suffer less from “version-itis,” or obsolescence; shaders are guaranteed to be portable to virtually all OSL supporting renderers. Because even compiled OSL shaders are stored in a text file, not tied to any hardware or OS platform, or specific renderer, they are completely portable across systems and renderer versions. The machine code is generated on the fly at runtime. So there is no possibility, as there would be with compiled DSO shaders, that a new renderer release could suddenly be incompatible with an old shader. We expect an OSL shader (both as source and compiled to .oso) to work with the renderer it was intended for, for years, possibly forever.

...And Renderer Developers

Again, OSL is safe to use. Your renderer will generally not be at the mercy of shaders attempting to do things that they were never intended to do. The number of bug and crash reports due to issues inside a shader will be limited to the functionality the renderer provides, not open to the abuse of external developers.

To a large degree, OSL shaders (and .oso files) should be compatible between renderers, but there are several ways in which this is not perfectly true: different renderers may supply different sets of nonstandard built-in closures, or may support different ray types, etc. This will only affect a few operations, the vast majority of shader nodes in the shader library, for example, could reasonably be expected to work identically in almost any conceivable renderer implementation.

Since OSL shaders are described independently of the rendering method, developers are free to explore new implementations without worry of breaking existing client investment in a shading library. Shader library compatibility becomes a virtual non-issue, and developers can change a renderer dramatically behind the shield of OSL. And because OSL abstracts the shaders from a low level API, exposed API's can be reduced substantially.

"Men In Black 3" images courtesy of Columbia Pictures. ©2012 Columbia Pictures Industries, Inc. All rights reserved.
Much of Boris the Animal's hand and the "Weasel" are CG elements rendered with SPI's OSL shader library.

How is OSL different from other shading languages?

Like other shading languages, GLSL, Cg, and RSL, OSL provides a rigidly defined framework for describing interactions between light and objects. But there are a few major differences in OSL to these other shading languages that will stand out to veteran shader writers. The first, and perhaps the scariest, is the lack of a light loop, that is, the inability to directly access light sources in the scene that are affecting the material. The second difference is that an OSL shader does not return color or alpha values to the renderer. Rather, `Ci`, a shaders main output, is a closure variable. It is important to understand that an OSL shader describes how a surface material scatters light to the renderer, it does not report pixel values or execute sample compositing. To that end much of the code in a shader will be dedicated to computing interesting and meaningful values as parameters to, and weights of the output closures.

Networking of OSL shaders is a first class concept in the library. Nodes can be dynamically arranged and are evaluated lazily. Because of the way OSL materials are handled during the render process, optimized

networks of nodes are nearly indistinguishable from monolithic shaders to the renderer. OSL is also built on a foundation of physically-based shading and advanced rendering principles. OSL presumes correct physical units throughout the implementation providing solid ground for physically based shading and lighting. Closures, when evaluated, give radiance values:

$$W/m^2/sr$$

Units of radiance, watts per square meter per steradian

This is true for surfaces, emissive geometry, and light sources using OSL shaders. This consistency is what easily allows users of OSL to freely interchange CG light sources, captured HDRI environments, and emissive geometry. We had a terrible time getting these lighting methods to match in the old system, and OSL opened the door to development of some highly accurate lighting techniques that previously relied on guesswork. Within Katana, the Imageworks lighting tool, lighters 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. The raw set data can also be projected onto reconstructed geometry, allowing rendered objects to 'live' in the acquired set.

A captured HDRI environment. This image can be used as a light source directly or...

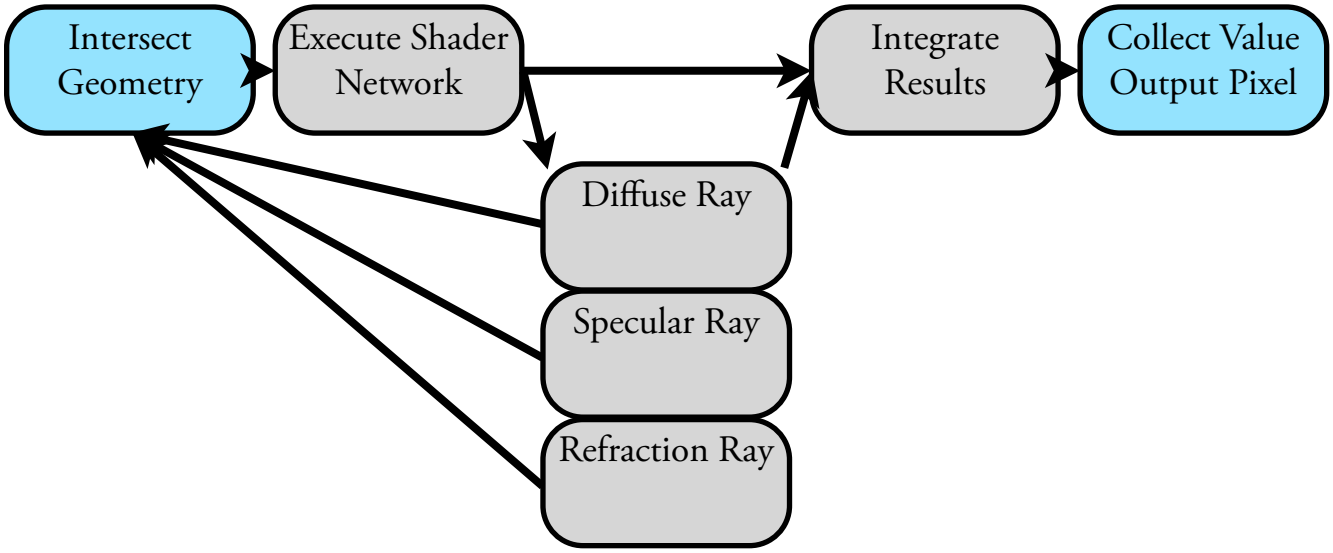
... portions of it can be identified and extracted as separate geometry or area light sources.

"The Smurfs" images courtesy of Columbia Pictures and Sony Pictures Animation. ©2012 Columbia Pictures Industries, Inc. and Sony Pictures Animation Inc, All rights reserved.

CG elements lit using the acquired environment and extracted light sources, and shaded using OSL closures.

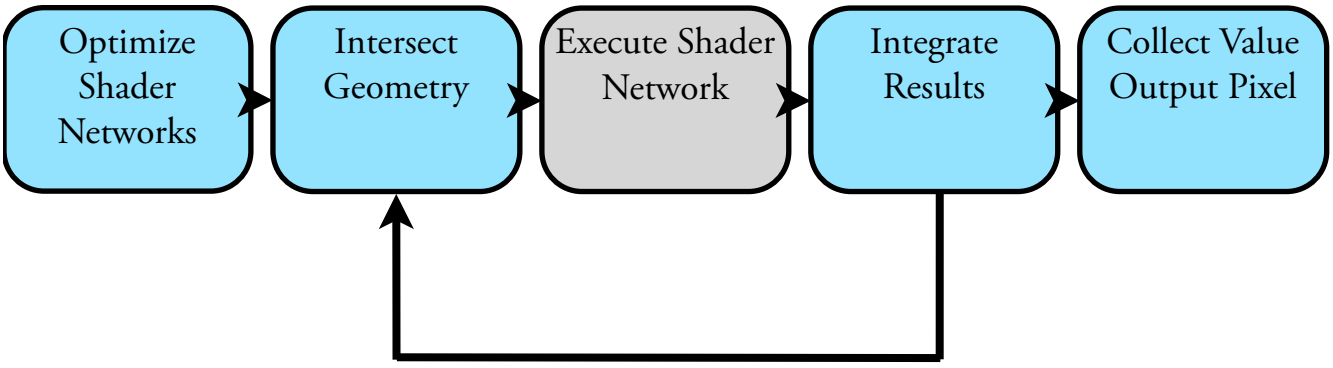
Unit consistency and energy conserving properties are also what allows OSL to unify the description of reflection, refraction, scattering, emission, and transparency into a single return value of surface shaders,

not a bunch of separate features and variables to set. OSL ships with a number of production proven physically plausible BSDFs: Cook Torrance Micro Facet model with Beckmann and GGX distributions, Ward Anisotropic.



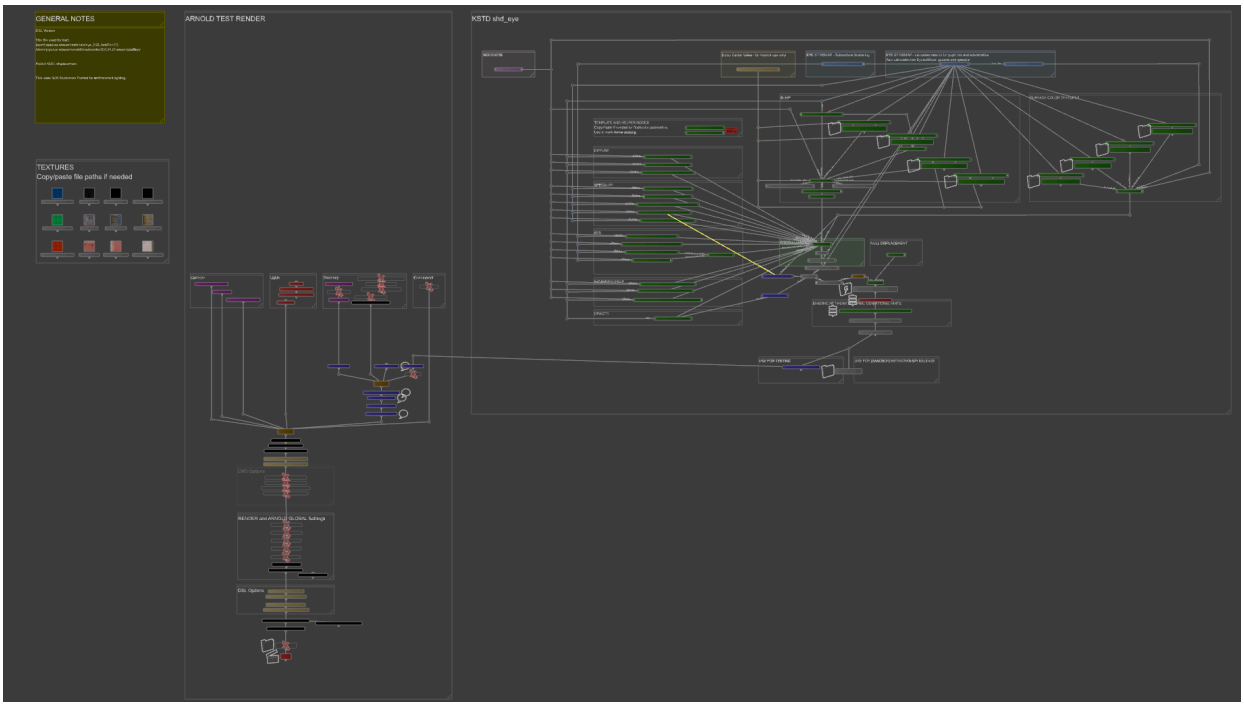
This is a naive diagram of how shading used to work in our old system. The blue boxes represent aspects of the process that the renderer has some direct influence over. The grey boxes all take place inside the shader, where the renderer has practically no capacity to optimize the computation. Renderer developers will recognize a major Achilles' Heel with this execution graph in that shader evaluation is recursive. The renderer is essentially locked out of the rendering process once the shader begins execution. If that shader requires information from the rest of the scene, for example, collecting incoming radiance, the shader hands off execution to another shader, and so on until the recursion loop is terminated. API's can help in limiting the amount of 'unknown' work that a shader does, but ultimately, and especially in open C-based systems, this can never be guaranteed.

Below is an equally naive diagram of the shading pipeline with OSL, using Imageworks' same path tracing renderer (Arnold) :

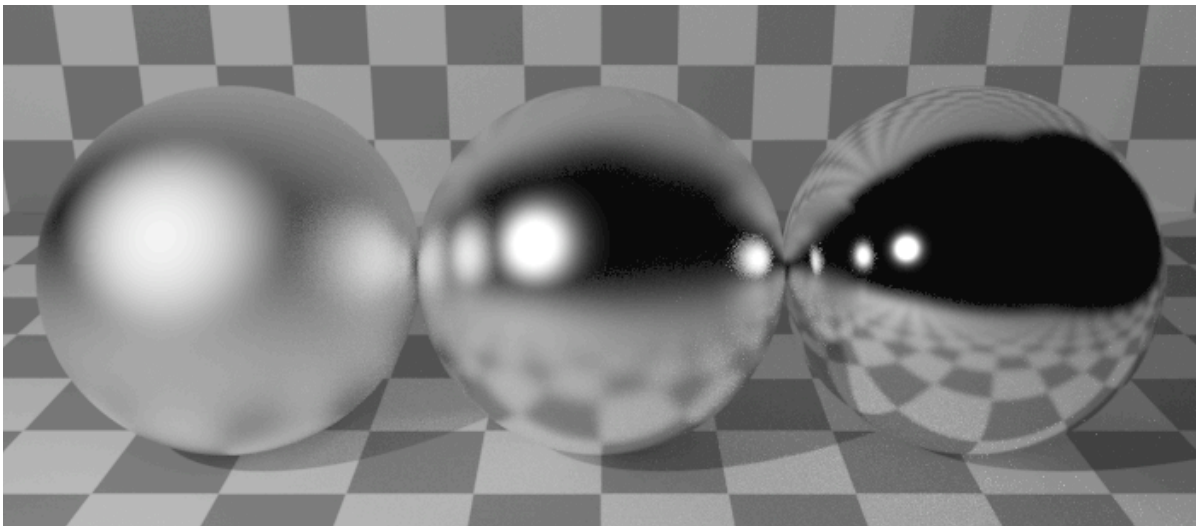


There are quite a few differences in this diagram from the previous, and the introduction of a few ambiguities. Notice the lack of recursion within the shader. This is very important because it means is that execution order and the integration of those results can be left entirely up to the renderer. Once a shader network is executed, either individually or as part of a collection, the renderer is handed a list of closures

and weights describing the material. At this point the renderer can take over and make interesting decisions about how, and when, to integrate the material descriptions to arrive at a final output value. Closures can be reasoned about and analyzed by the renderer, evaluated (closure + lights + view direction + samples = final color), sampled accurately and efficiently, and also can be stored for later evaluation. This separation of the shading and integration stages is key to effective multiple importance sampling. Other example uses include: interpolation, ray sorting/reordering, or rapid relighting by re-evaluating the closures with new lights rather than recomputing all shading.



The general purpose shading network at Imageworks has stayed topologically familiar through multiple generations of the underlying shader library. This is a screen shot of the Katana node graph of an OSL shader.



The testrender command line utility is very useful for testing new shaders

How OSL Works: Foundations and Definitions

OSL is a simple but powerful language that reduces complex shading principles to manageable functions. Ultimately, a shader is responsible to delivering a material description in the form of a weighted sum of closures.

In computer science and language design, a closure is a packaged up bit of code -- say, a function to call and the arguments to give it -- that can be passed around as an object, entirely separate from the evaluation (execution) of that function. That's really what's going on with OSL material closures: instead of choosing samples and evaluating the BSDF for a particular direction and returning only the resulting color from a shader (i.e., the status quo way), OSL is returning an object that describes the procedure for evaluating the BSDF, with everything baked down to numeric parameters and weights except for the sampling itself. Once you have this (ideally view-independent) package, the renderer can sample it intelligently, evaluate it for any (or multiple) directions, stash it away somewhere, save it for later, whatever.

A closure in OSL is a symbolic representation of the way a surface (or volume) scatters and emits light. The representation is opaque to the shader writer, and the renderer is free to implement them as it chooses. OSL closure functions are a single, common interface to the BRDF, sampling, and PDF functions of shading models, a number of which ship with the project, such as the widely used Cook-Torrance micro-facet model. At SPI the most popular distribution is Beckmann. Represented in OSL, this closure function looks like:

```
microfacet_beckmann(N, roughness, ior...) * weight
```

Material transparency and refractions are closures treated like any other BSDF in the shader, and I will demonstrate a practical example of this later on.

It is important to note that while I focus here on BSDF closures, OSL supports closures for other shading effects as well. For example, closures for sub-surface scattering BSSRDF's, shader and renderer debugging and shadow mattes are also possible.

A Brief History and The State of The Art

In my Physically Based Shading presentation for Siggraph 2010 [1], I described a shading system developed at SPI that evolved from a combination of dissatisfaction with a pass-based approach, and a desire to utilize the path tracing renderer Arnold. At that time, the Arnold renderer was a being co-developed with Solid Angle. Arnold had seen limited use between the production of the animated features "Monster House," in 2006, and "Cloudy With A Chance of Meatballs" in 2009. For visual effects work, it was determined that a new set of shaders would have to be developed to make greater use of physically based shading principles.

This shading system was built as a large set of C plug-ins that were chained together into feature-rich networks. It went through a number of generations as it was used over the course of "Alice in Wonderland", "2012", "Green Lantern", "The Smurfs" and "Arthur Christmas". This shading system packed a lot of technology into it: All of our basic illumination models and their sampling routines were built into the shaders directly. Secondary outputs, including some geometric data, were handled internally by a complex method of message passing between materials. Illumination effects that required sampling of the scene were all hand-coded to do so, as in the case of traced subsurface scattering. Material layering, while a powerful tool for look developers, became burdensome as it saw more widespread use and abuse.

At the same time issues with the shader library maintenance was becoming problematic due to varying show demands. Feature creep and inefficiency in the shading system alone was reaching critical levels. The

renderer developers were somewhat boxed in by concerns over breaking a delicate production shading system. Essentially, the shaders hijacked the renderer, which was by design, but we were realizing the liability in that decision.

From a renderer developer's stand point, the greatest fault with this shading system was not merely the majority of computation time spent in the shaders. The shaders were an inscrutable black box to the renderer, and so even the parts that the renderer was responsible for were often done by brute force because the renderer couldn't make intelligent choices based on knowledge of what the shaders were doing. Various renderer internals were so over-exposed to the shaders themselves that the rendering team was severely constrained in its ability to change internals or refactor the renderer code in any way, which in many cases was standing in the way of either performance improvements or increased ability to easily maintain the code. Between the fact that half the algorithm was either encoded in the shaders themselves, or the shaders were dependent on certain ways of doing things, we really had no ability to try different rendering algorithms (experimenting with something like BDPT was a non-starter). So the rate of algorithmic improvement of the renderer was highly constrained.

Before “Green Lantern” production started in earnest, the shading department had started working with early versions of the OSL library that had been shoe-horned into our production build of the Arnold renderer. Initial results were very promising. We were able to rapidly meet feature requirements for the “Green Lantern” look development teams, and the quality of the results were a marked improvement over it's predecessors. Performance, however, became an unavoidable issue, as optimizations were not in place and would not be available in time to meet the needs of the production. A novel compromise was reached that allowed us to stage OSL’s release to the facility, and also to divide and conquer aspects of the system that needed attention. The core OSL-side BSDF's were made available to the C API essentially allowing us to utilize the robust importance sampling routines from within our existing C shading framework. In addition, integration of both the light loop and the sampling loop were all handled by the renderer, at the shader's request. This hybrid OSL/C solution delivered the familiar interface of the older shading system with the look and feel of the new sampling and illumination methods, and satisfactorily met the rendering needs of the show. We were able to rigorously production test and refine the core shading functionality of OSL while the OSL and Arnold teams continued to develop the infrastructure, feature set, and optimizations that would allow us to switch completely over to OSL. All of these developments are reflected in the current public OSL trunk.

“Men In Black 3”, “The Amazing Spider-Man”, and the animated “Hotel Transylvania”, would be the first shows at SPI to truly push OSL on a real production level. While “Green Lantern” was wrapping up, artists for these films were starting to get their first taste of a purely OSL shading system. The reactions were extremely positive from the start, and remained so during the run of all the productions. OSL's optimizations had improved to the point where it was actually significantly outperforming the equivalent C shaders by somewhere in the neighborhood of 20% or so as measured by overall rendering time. These three films were rendered simultaneously, solely with the OSL shading system.

For all shows we were able to quickly and effectively deliver novel solutions to unique problems in look development and shot rendering. Fully path traced subsurface single scattering was introduced as well as new shading models for hair. It required very minor changes to the shading system to deploy, and pushed the looks of our characters into new areas of realism; we could now adequately represent subcutaneous occlusion in translucent materials, such as bones and cartilage.

"Hotel Transylvania" images courtesy of Sony Pictures Animation. ©2012 Sony Pictures Animation Inc. All rights reserved.
The character of "Goopta" utilizes ray traced single scattering translucency almost exclusively.

At SPI, we were afforded the opportunity to step back from our shading system, critically scrutinize it, and make dramatic changes. By this time, OSL was highly informed by how we were working in the older system. In a lot of ways we could perfectly replicate the familiar shading system, and in many cases that is just what we did. However, a lot of the features that would be normally coded directly in the shader, became a managed part of the renderer and exposed via OSL functions or closures. OSL also allowed us to rapidly prototype, test, refine, and release ideas without a complex and lengthy build and qualification procedure. In the summer of 2011, the publicly exposed C API for writing shaders in the Imageworks renderer was entirely removed.

The OSL Experience -or- How I Learned to Stop Worrying and Love the Closure

The experienced shader writer at first might see OSL as a threat to the flexibility and power. At SPI, we had many reservations about using the new system, most of which was driven by fears of an unknown. Things we got used to supporting in C shaders were being implemented directly inside the renderer. This included core features of the shaders such as illumination and sampling loops, but also extended into specialty and utility features, such as subsurface scattering and secondary outputs. Shaders could no longer arbitrarily trace rays at any point in the rendering process, no matter how benign the stated intention. The perception, from the point of view of the shader writers, was that we were undergoing a huge loss of capability and flexibility.

The transition was difficult for the renderer developers as well. Support requests for shader and renderer issues in the young system largely fell on the renderer development team to address. This meant that productions, who would normally go through the shading department, were now applying pressure to a development team that was normally sheltered from the panic that can ensue during a project. On top of refining existing functionality, optimizing the system to meet performance goals, and fielding the needs of the shading department, the Arnold team had a lot to handle.

At a certain point features and behaviors inside OSL stabilized and the delineation of responsibilities between the teams shifted but settled. The shading department took on a more active role in developing

new cutting edge features directly in the renderer, as well as becoming more engaged in production. The rendering team also became more engaged in production and helped to define best practices. We realized quite quickly that new feature implementation costs decreased dramatically. New rendering methods for shading hair, thin film interference, subsurface scattering were all developed, tested and deployed to productions in a matter of weeks. As shader writers, we were able to meet the needs of artists much faster, and in ways that would have previously required much more extensive development schedules.

Our OSL materials are doing much more per sample than their older C counterparts, and they are measurably faster. Where previously we would take a hit to render speed as material complexity increased, we could now count on OSL's built in optimizations to cull unimportant code from the execution path. In addition, network complexity significantly reduced; because many features now lived renderer side, they no longer needed exclusive representation in shader networks.

Things we were predicting would be problematic, turned out not to be. The inability to trace rays from inside a shader arbitrarily was met with a compromise with the `trace()` function. This function allows us to immediately trace a ray and collect information about the scene in a rigidly managed way, it does not shade or integrate. While it has been used for experimental shaders and non-photorealistic tests, this particular function has been utilized in only one specialized production shader.

OSL opened the door to unprecedented rapid prototyping and experimentation in the shading system. New interface ideas can be rapidly mocked up, tested and critiqued by a large number of potential users. Different illumination models and patterns can be easily networked together, tested, and released into the wild without much concern for adverse consequences.

At Imageworks we are also realizing the rapid development of new rendering techniques. Imageworks' volumetric rendering features in Arnold (based on the work of Chris Kulla and Marcos Fajardo [2]) fully leverages the OSL shading strategy. The utilization of these features in production shaders is near-instant. For example, the dipole subsurface scattering implementation used on films up until “Hotel Transylvania,” relied on a traditional method of caching radiance values in a point cloud. A recently developed purely ray-traced scattering routine with the same interface, and exactly the same look, required no shader modification to deploy. The benefits of this development were immediately leveraged in production with great success, and is now the default method for all of our subsurface scattering needs.

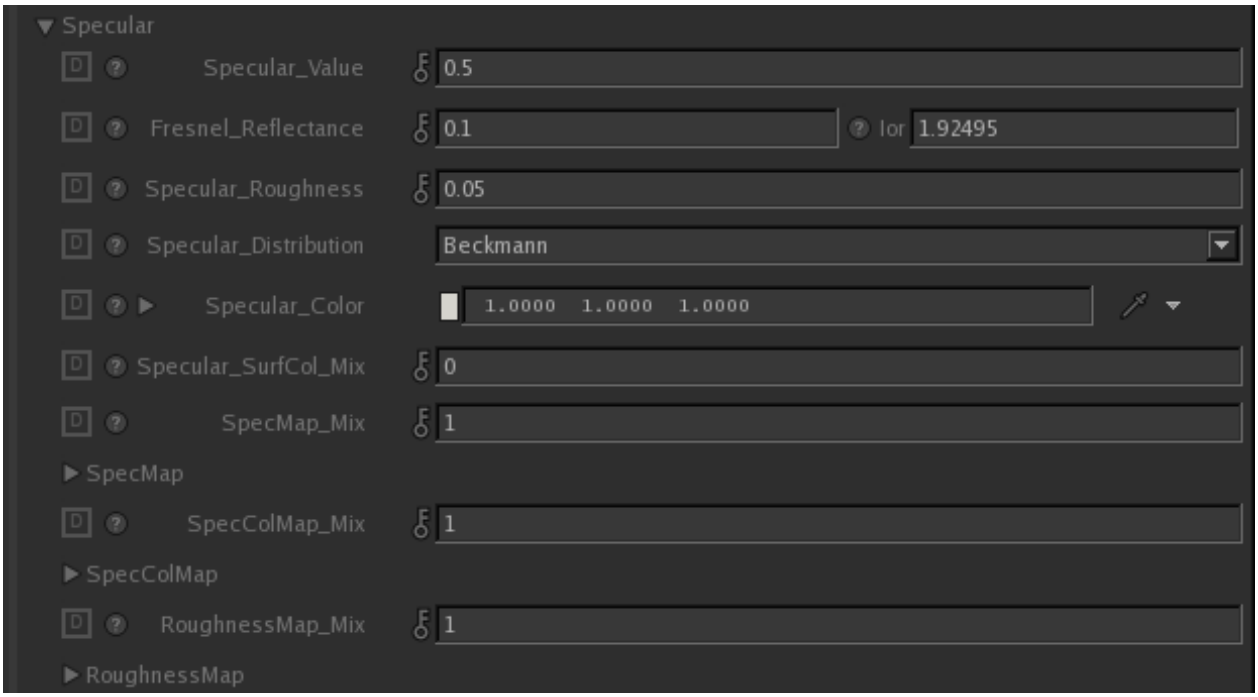
Of course, as a shader writer, having access to the renderer developers is a huge benefit. Being able to bounce ideas off of the OSL and Arnold developers offers a level of feedback that can be hard to acquire with commercial software. Given that, the public OSL project is exactly the same library we use in the SPI renderer, and to that end the public project benefits from the constant feedback loop that the developers have with production shader writers. OSL is deployed to production straight out of the public GitHub master development trunk and any specialized shading functionality specific to SPI exists renderer side. The code for the OSL libraries themselves is easily understood, and extensible, certainly more so than most rendering engines.

"The Amazing Spider-Man" images courtesy of Columbia Pictures. ©2012 Columbia Pictures Industries, Inc. All rights reserved.
The Lizard and Spider-Man were among the very first creatures rendered with OSL shaders, the Lizard being one of the most complex creatures ever built at Imageworks.

Writing OSL Shaders, A Practical Example

Writing OSL shaders is remarkably simple, and familiar if you've ever written a shader in RSL, C or GLSL. In this section we will construct a few simple yet practical shaders that adhere to physically plausible shading principles. I will provide a very brief overview of the material that is covered to a much greater extent in the osl-languagespec.pdf document available with the distribution.

OSL mostly resembles C in its syntax. All of the familiar math operators, logical constructs and data types are supported. OSL supports include files, pre-processor definitions and macros, as well as specialized support for meta-data. Meta-data tags are useful for specifying user interface hints to host applications. Widget styles, value ranges and help text, are common tags attached with parameter meta-data.



OSL metadata tags are used to partition parameters and create widgets in Katana.

Standard data types supported are: `int, float, color, vector, point, matrix`, and `string`. All of these types are supported as parameter types as well.

As mentioned above, the specialized `closure` type is used to store references to closure functions and is the primary delivery method of a material description to the renderer. At declaration closure variables do require a subtype, however at this time only color is supported. Closure variables are subject to a number of special rules owing to the unique data they represent. Closures can be weighted by float and color data types, and closure variables may only be added together.

Static arrays and variable structures can be constructed from any of the supported data types. OSL does not yet support dynamic arrays.

The definition of shader parameters for OSL shaders is straightforward. All parameters require a default value so as to ensure un-ambiguous results. Closure parameters can use a default of 0. Parameter declarations can include meta-data to provide user interface hints in host applications:

```
color   Diffuse_Color = 1
   [[   string help = "Diffuse closure color",
        float min = 0, float max = 1 ]],
```

Parameters that are tagged as output parameters are made available to connect to any other parameter on any other node in the shading network. The reserved global variable `Ci` is of type `output closure color` and is used to report the final material closures to the renderer.

A Simple Shader

Here is a complete listing for a shader that renders a texture mapped diffuse appearance:

```
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);
}
```

For two lines of code in the body, this shader purports to do quite a lot: a user parameter texture name is read in using the u and v texture coordinates from the geometry. That value provides the weight of a closure function for diffuse reflectance of both direct and indirect illumination. The core component of this shader is the `diffuse(N)` closure which represents a Lambertian reflectance function. The internals of the closure are left to the implementation in the renderer, however the testrender program in the OSL project provides a few closure examples for reference.

Let's extend this shader by providing a simple Phong specular model. We need to add the specular exponent parameter which is required by the function. Then all that is left is adding a closure to the `Ci` variable:

```
surface
example_shader_2
        [[string description = “Simple texture mapped diffuse material”]]
(
        string texture_name = “”
              [[string help = “A texture file name”]]
        float specular_exponent = 7
              [[string help = “The exponent of the specular reflection”]]
)
{
        color paint = texture(texture_name, u, v);
        Ci  = paint * diffuse(N) + phong(N, specular_exponent);
}
```

This shader is problematic now in that the amount of energy returned from the surface is potentially more than what is incident at the surface. We can rectify this by using an energy conserving microfacet closure, and by balancing the weighting of the closures individually. The following section will demonstrate these solutions in more complex shader examples.

A Physically Plausible Glass Shader

In physically based shading contexts one of the most challenging problems is accurately representing glass and other highly transmissive materials. A good glass shader should provide energy conservation between its reflective and refractive components, support total internal reflection, and properly represent real world index of refraction behaviors. In advanced rendering schemes, such as bi-directional path tracing or metropolis light transport, physically accurate glass materials should also exhibit caustics. Accomplishing

all of these goals, on top of providing production flexibility is exceptionally difficult. Consider the following OSL shader:

```
surface
example_shader_3
      [[ string help = “Simple dielectric material” ]]
(
      color Cs = 1
            [[ string help = “Base Color”,
                float min = 0, float max = 1 ]],
      float eta = 1.5
            [[ string help = “Index of refraction” ]],
)
{
      if( backfacing() )
      {
            Ci = refraction(N, 1.0 / eta) + reflection(N, 1.0 / eta);
      } else {
            Ci = refraction(N, eta) + reflection(N, eta);
      }
      Ci *= Cs;
}
```

This shader returns a refraction and reflection closure weighted by a user-specified color. Both closures receive the same parameter values, ensuring conservation across reflected and transmitted energy. In the case that the surface normal is facing away from the viewing direction, the index of refraction is inverted. This condition is detected by the OSL function `backfacing()`. Total internal reflection is accomplished by adding the reflection closure for the back facing case.

While this shader does satisfy many of the requirements for a glass appearance, it is far from production friendly. We can flesh out the capabilities of this shader by supporting roughness, bump mapping and user options to control contextual behavior:

```
surface
example_shader_4
   [[ 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_caustics = 0
      [[ string help = “Enables indirect lighting”]],
   int enable_tir = 1
      [[ string help = “Enables total internal reflection”]],
   float bump_value = N
      [[ string help = “Input normal” ]]
)
{
   if( enable_caustics || (!raytype(“glossy”) && !raytype(“diffuse”)) )
   {
      vector bump_normal = normalize(calculatenormal(P + N*bump_value));

      if( backfacing() )
      {
         Ci = microfacet_beckmann_refraction(bump_normal, roughness, 1.0 / eta );
```

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

This shader demonstrates a number of capabilities of the OSL shading system. We have changed the primary closure type from the mirror-like `reflection()` and `refraction()`, to the more physically interesting `microfacet_beckmann()` and `microfacet_beckmann_refraction()`. These closures take an additional roughness parameter to control how 'blurry' the specular qualities are owing to microfacet variation on the surface. In the same way that reflection and refraction will balance singular ray evaluations given the same parameter values, the microfacet closures do the same with like roughness values.

We have also introduced a trap for indirect illumination to curtail evaluation of the shader in both glossy and diffuse contexts. The OSL function `raytype()` is used to detect the renderer state at the shader's evaluation. By passing the strings `"glossy"` and `"diffuse"` to `raytype()`, we are preventing a potential explosion in traced rays for secondary light paths, since those ray types are likely to represent multiple rather than singular rays. We could make our shader more efficient by specifying entire branches of the ray tree:

```
(!raytype("path:glossy") && !raytype("path:diffuse"))
```

By prefixing '`path:`' to the ray type, we are asking the renderer whether any ray in the current evaluations history matches the specified type. In this case, if any ray in the tree leading to this evaluation was of type glossy or diffuse, the shader will not evaluate. It is entirely possible to do away with this check if the renderer provides its own detection of potential issues. `raytype` is an example of an OSL function that exposes renderer internals and therefore behavior of this function is not guaranteed to be portable. Shaders should be written such that when the call fails (unknown ray type) - the material takes no shortcuts and returns the full material. In this way the shader can be future-proofed to new rendering algorithms that classify rays differently (or don't use rays at all!).

It's important to note that we've added a number of conditionals into this shader now. The shader writer should be aware that there is no run-time cost to conditionals; based on parameter values, almost all of them will be folded away by OSL's run-time optimizer which will know the final values of all the shader parameters by the time it is generating the actual executable code for the shader network.

`backfacing()` and `raytype()` are two functions that provide the shader some information about the state of the renderer and the scene at the time of evaluation. Two other functions that retrieve render state information are `surfacearea()`, which is specifically for area lights, and `getmessage()`, which is capable of retrieving information about ray intersections when used in conjunction with `traceprobe()`.

The above shader also provides a new input parameter, `bump_value`. This value is used to offset `P`, and

passed as a parameter to the `calculatenormal()` function[1], which returns a new vector representing the normal at this new point. This new parameter allows us to connect another shader with an `output float` type parameter, such as a basic 3d Perlin noise pattern:

```
shader
simple_noise
(
      float frequency = 2,
      float amplitude = 1,
      output float out_noise = 0
)
{
      out_noise = noise(P * frequency) * amplitude;
}
```

This shader is a simple pattern generator. Since it does not set `Ci`, it cannot be used to render directly but must be incorporated into a network that will use the value of `out_noise` as a weight or parameter to a closure.

A More Complex Example

The goal here is to create a fairly generic shader that supports diffuse, specular, bump, refraction as well as texture handling. Such a shader should be a suitable production platform, provided it meets the artistic needs of the user base. To this end, we should be explicit about the kinds of controls we want to provide:

- We will need a base color control for the diffuse aspects of the material
- Specular qualities such as intensity, roughness and index of refraction, as well as independent color control.
- Bump magnitude, presuming bump patterns are provided by an external source, we want a way to control the apparent height of the bump locally available.
- Transparency mapping should be supported. Refraction support should be physically accurate and follow the specular settings, but allow for some artistic manipulation.

In addition to this list, every aspect of this shader should be texture mappable to the extent that it can. There are two approaches we can take to satisfy this requirement. The first, and probably the most efficient method is to use an external set of texturing nodes and use OSL's shading graph to connect nodes together. This will ensure a degree of consistent behavior and potentially circumvent a lot of repetitive coding. An alternative way would be to make use of preprocessor macros and definitions to handle the generation of texturing parameters and functionality. While the former is an attractive production proposition, the latter affords the capacity to demonstrate more features of OSL:

Below is a partial definition of a preprocessor macro that generates texture parameters:

```
#define TEXTURE_PARAMS(MAPNAME) \
    string Texture_ ## MAPNAME ## _Name = ""\
        [[ string widget = "filename", string help = "A texture file name", \
```

[1] It is worth mentioning that the OSL implementation of `calculatenormal()` is different from traditional implementations of RSL in that calls which require derivatives do not require the shading system to run the shader on a grid of points internally. Analytical derivatives are tracked by the runtime automatically from a single shading point, including through loops and conditionals.

```
        string page = #MAPNAME ".Texture"]],\
float   Texture_ ## MAPNAME ## _Blur    = 0.0\
    [[  string help = "The blur amount, as a portion of the image size.", \
        string page = #MAPNAME ".Texture"]],  \
         ...
```

This macro uses the tag `MAPNAME` to populate a set of parameters common to any texture.
Note that the metadata tags used here are specific to Katana. The '`page`' metadata tag specifies a named group of parameters that are collectively displayed in the user interface.

We populate the texture parameters by referencing this macro in the surface shader parameter declaration list:

```
surface
example_shader_5
(

    color   Diffuse_Color = 1
       [[   string help = "Diffuse closure color",
            float UImin = 0, float UImax = 1]],

    TEXTURE_PARAMS(Diffuse)
)
...
```

This will result in a list of parameters that looks like:

```
color   Diffuse_Color
string  Texture_Diffuse_Name
float   Texture_Diffuse_Blur
...
```

Texture lookups will be handled by a locally defined function that calls `texture()`. We will want to wrap this local function in another macro, so that we can make use of the `MAPNAME` tag in the body of the shader:

```
#define TEXTURE_EVAL(MAPNAME, RESULT)\
    evaluateTexture(\
    Texture_ ## MAPNAME ## _Name,\
    Texture_ ## MAPNAME ## _Blur,\
    Texture_ ## MAPNAME ## _Width,\
    Texture_ ## MAPNAME ## _Wrap_Mode,\
    Texture_ ## MAPNAME ## _Scale_U,\
    Texture_ ## MAPNAME ## _Scale_V,\
    Texture_ ## MAPNAME ## _Flip_U,\
    Texture_ ## MAPNAME ## _Flip_V,\
    In_U,\
    In_V,\
    RESULT);
```

This macro takes the `MAPNAME` tag and hashes it into the same parameter names as we defined in the

TEXTURE_PARAMS macro and passes them as arguments to a generic evaluateTexture function. The RESULT is storage for whatever the texture lookup returns, usually this will be a color or a float, but other types are possible. OSL follows an extensive but well defined set of conversion rules.

At this point, we've replicated the functionality of our first example shader, but extended the controls quite a bit. Our texture macros allow user level control over the texture lookup including blur and tiling. We can combine and extend this to include the work we did in the glass shader example. By incorporating the bump mapping and specular parameters our shader interface has grown quite a bit:

```
surface
example_shader_5
(

   color   Diffuse_Color = 1
      [[   string help = "Diffuse closure color",
           float min = 0, float max = 1]],

   TEXTURE_PARAMS(Diffuse)

   float   IOR = 1.33
      [[   string help = "The index of refraction for specular effects.",
           string page = "Specular"]],

   float   Roughness = 0.2
      [[   string help = "The surface roughness for specular effects.",
            string page = "Specular"]],

   color   Specular_Color = 1
      [[   string help = "Color tint of specular reflections.",
           string page = "Specular"]],

   TEXTURE_PARAMS(Specular)

   float   Kt = 0
      [[   string help = "Transmissive, or refraction, closure weight.",
           string page = "Refraction"]],

   color   Refraction_Color = 0
      [[   string help = "Refraction closure color",
           string page = "Refraction"]],

   TEXTURE_PARAMS(Refraction)

   float   Opacity = 1
      [[   string help = "The overall opacity of the surface",
           string page = "Opacity"]],

   TEXTURE_PARAMS(Opacity)

   float   Kbump = 0
      [[   string help = "The amount to bump the surface by",
           string page = "Bump"]],

   TEXTURE_PARAMS(Bump)
```

```
)
```

Let's begin the body of the shader by evaluating the bump parameters and storing the new shading normal. We will use the `TEXTURE_EVAL` macro we defined earlier to read a bump texture if available:

```
float  totalOpacity = 1;
normal Ns = N;

// Opacity
TEXTURE_EVAL(Opacity, totalOpacity)
totalOpacity *= Opacity;
totalOpacity = clamp(totalOpacity, 0, 1);

// Bump
if(Kbump) {
    float bumpTexture = 0;
    TEXTURE_EVAL(Bump, bumpTexture)
    bumpTexture *= Kbump;

    if(bumpTexture) {
       point Pbump = P + normalize(Nshading) * bumpTexture;
       Ns = normalize(calculatenormal(Pbump));
    }
}
```

The variable `Ns` now stores the normal we will want to use for the remainder of the shader computations. The variable `totalOpacity` contains the combined weighting of the user parameter and the texture map, clamped to a valid range.

The other textures we read will act as weights on each illumination component. So for each illumination component, we want to create a closure variable:

```
closure color diffuseClosure = 0;
```

As stated, there isn't much we can do with a closure variable except to add other closures into it, and weight it by color or float quantities. This particular closure variable will store the diffuse illumination component. Let's use a simple Lambert diffuse and weight it by the `Diffuse` texture map as well as the `Diffuse_Color` parameter:

```
color diffuseTexture = color(1);
TEXTURE_EVAL(Diffuse, diffuseTexture);
diffuseClosure = diffuse(Ns) * Diffuse_Color * diffuseTexture;
```

For the specular model of our shader, we will use the same Cook-Torrance specular model we used in the glass example, `microfacet_beckmann()`, and the corresponding refraction closure. Since our shader is potentially transparent, and it is possible we will be rendering back facing geometry, we will also add some logic to invert the index of refraction when necessary:

```
float eta = IOR;
if(backfacing()) eta = 1.0/IOR;

color specularTexture = color(1);
```

```
TEXTURE_EVAL(Specular, specularTexture)
closure color specularClosure = microfacet_beckmann(Ns, Roughness, eta) *
Specular_Color * specularTexture;

color refractionTexture = color(1);
TEXTURE_EVAL(Refraction, refractionTexture)
closure color refractionClosure = microfacet_beckmann_refraction(Ns,
Roughness, eta) * Refraction_Color * refractionTexture *Kt;
```

Transparency is simply another closure. We call it with a weight that represents how transparent the material is:

```
closure color transparentClosure = transparent()*(1.0-totalOpacity);
```

It now remains to collect the closures and add them into `Ci`, attenuating the results for opacity:

```
Ci = (diffuseClosure + specularClosure + refractionClosure) * totalOpacity +
transparentClosure;
```

In addition to adding the ray type checks for glossy and diffuse, such as we did in example shader 2, we can test for shadow contexts and make sure we do the minimal amount of computation possible:

```
if(raytype("shadow")) {
    Ci = transparentClosure;

} else {
  …
```

There is much more we can do with this shader to make it more efficient, and more physically plausible. For example, we can add float coefficients as a convenience for balancing the weighting of the specular and diffuse components, or go a step further and balance the weights internally. We can add functionality to allow users to switch specular models. We can (and probably should) invert the refraction coefficient `Kt` and attenuate the diffuse component, since it makes physical sense to do so. This shader can be extended in many more ways to increase its capabilities. Adding translucence, emission, subsurface scattering and other illumination models is as easy as adding the appropriately weighted closures to `Ci`.

In some production environments it may be appropriate to use a node-based approach to providing generalized shader functionality. While the texture evaluation macro defined in the above shader is a convenient way to add texture mapping to a shader it would be far more flexible to defer these operations to an external node. Having all of the texture lookup work take place separately from the main illumination logic would easily allow for more extensive texture mapping options, and simplify the code of this particular shader. As stated above, there is no penalty for providing functionality as an external node, the run-time optimizer will collect and condense all of the operations in the shading network as a whole.

Debugging OSL Shaders

OSL shaders are easy to debug owing to the readability of the code, and the legibility of the syntax checking of oslc. Render time debugging of OSL shaders in operation is not quite as straightforward, since renderers may implement some features differently. OSL provides a `printf()` function that allows a shader to print values during evaluation, including closure statements. For example, the following statement:

```
printf("Ci: %s", Ci);
```

could yield the following information about a sample at render time[2]:

"The Amazing Spider-Man" images courtesy of Columbia Pictures. ©2012 Columbia Pictures Industries, Inc. All rights reserved.

In some cases it was necessary to print out the resulting closure list from a material to figure out exactly what was going on

```
[osl-core] Ci: (0.02, 0.033, 0.02) * microfacet_beckmann ((-0.02, 0.47, 0.88),
0.05, 1.33)
      + (0.08, 0.07, 0.05) * debug ("Surface_Color")
      + (0.05, 0.06, 0.05) * microfacet_beckmann ((-0.02, 0.47, 0.88), 0.11, 1.33)
      + (0.08, 0.07, 0.05) * debug ("Surface_Color")
      + (0.04, 0.03, 0.02) * diffuse ((-0.02, 0.47, 0.88))
      + (1, 1, 1) * microfacet_beckmann_refraction ((-0.02, 0.47, 0.88), 0.6, 1.27,
"absorption", (2.93, 4.44, 6.22), "scattering", (0.41, 0.60, 0.63), "eccentricity",
0)
      + (0.02, 0.03, 0.02) * microfacet_beckmann ((-0.02, 0.47, 0.88), 0.22, 1.33)
      + (0.08, 0.07, 0.05) * debug ("Surface_Color")
```

Shaders can also provide a number of debug capabilities such as setting extra output values, or messages (using the `setmessage()` and `getmessage()` functions) that can be re-directed to `Ci`. At Imageworks we employ this capability to provide shader writers, look development artists and lighters a way to inspect certain intermediate results and other things that help them diagnose problems.

Render time statistics provide useful information about the status of OSL during execution. Optimization statistics are printed that let a users know how many symbols and operations were optimized out of the shading network:

```
INFO: Optimized shader group: New syms 5/11 (-54.5%), ops 2/6 (-66.7%)
INFO:     (0.12s = 0.00 spc, 0.00 lllock, 0.03 llset, 0.00 ir, 0.08 opt, 0.01 jit)
```

---

[2] Decimal precision is truncated for readability in this example.

This is a very simple shader that generated 11 symbols, reduced to 5, and 6 operations, reduced to 2, after the user parameters were populated. Let's look at a more extreme production example:

```
[osl-core] Optimized shader group /root/world/geo/lizard_hi/root/lizard_hi_cn_body_vis/
lizard_hi_cn_body/lizard_hi_cn_bodyShape:arnoldSurfaceShader:
[osl-core]     New syms 7607/413411 (-98.2%), ops 21816/732837 (-97.0%)
[osl-core]     (4.78s = 1.35 spc, 0.00 lllock, 0.01 llset, 0.22 ir, 2.39 opt, 0.81 jit;
local mem 42KB)
```

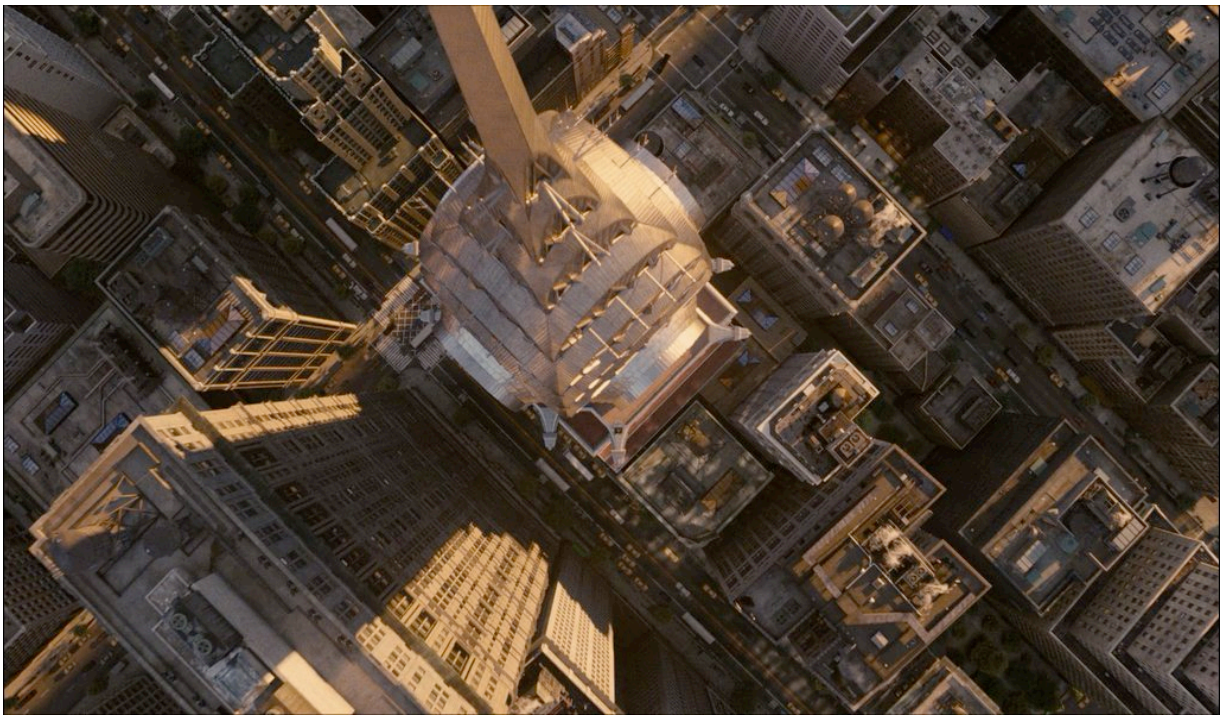
In this example from a "Spider-Man" production render log file, a fairly massive shader network (over 700,000 ops!) is reduced to an instruction set of 21,816, a fraction of the original. This information lets us know how much of the shading network is being used and how long the optimization step is taking (almost 6 seconds, not long at all). In terms of evaluating extensive render times, this can be used to verify the number of instructions present in a given shader network at run-time. Note that these statistics are displaying the reduction of a group of shaders networked together, not individual shaders. So while 700,000 ops may sound huge (it is) those ops are contained within numerous individual shaders within a network, and many of those shaders may never actually be used depending on user options.

End of frame rendering statistics gives vital information about OSL's resource usage. These statistics can help identify problems with material assignment, shader use and overall shader performance. Consider the following end-of-frame statistics from the same render that provided the above run-time optimization:

```
OSL ShadingSystem statistics (0x395ba20)
   Shaders:
     Requested: 6957
     Loaded:    59
     Masters:   59
     Instances: 6957 requested, 6957 peak, 6957 current
   Shading groups:   31
     Total instances in all groups: 6957
     Avg instances per group: 224.4
   Shading contexts: 8 requested, 8 peak, 8 current
   Compiled 31 groups, 6957 instances
   After optimization, 5730 empty instances (82%)
   After optimization, 6 empty groups (19%)
   Optimized 5823785 ops to 109039 (-98.1%)
   Optimized 3305863 symbols to 37625 (-98.9%)
   Runtime optimization cost: 50.13s
     locking:                    26.35s
     runtime specialization:     9.95s
     LLVM setup:                 0.48s
     LLVM IR gen:                0.81s
     LLVM optimize:              8.22s
     LLVM JIT:                   4.26s
   Regex's compiled: 0
   Largest generated function local memory size: 42 KB
   Memory total: 329.2 MB requested, 80.6 MB peak, 14.1 MB current
     Master memory: 4.3 MB requested, 4.3 MB peak, 4.3 MB current
         Master ops:             2.0 MB requested, 2.0 MB peak, 2.0 MB current
         Master args:            331 KB requested, 331 KB peak, 331 KB current
```

```
    Master syms:           2.0 MB requested, 2.0 MB peak, 2.0 MB current
    Master defaults:       20 KB requested, 20 KB peak, 20 KB current
    Master consts:         13 KB requested, 13 KB peak, 13 KB current
Instance memory: 324.9 MB requested, 76.3 MB peak, 9.7 MB current
    Instance syms:         318.6 MB requested, 70.0 MB peak, 3.4 MB current
    Instance param values: 2.1 MB requested, 2.1 MB peak, 2.1 MB current
    Instance connections:  2.7 MB requested, 2.7 MB peak, 2.7 MB current
LLVM JIT memory: 4.1 MB
```

Here we see that the total overhead for optimization to render-time was about 50 seconds. Timings are measured across all threads, so for this 8-thread render the optimization cost would have been around 6 wall-clock seconds. The total amount of memory requested to store the shaders, networks and associated data was 329 MB. The scene consisted of 31 groups, or networks, totaling 6,957 individual instances of 59 distinct shaders. After optimization, the scene only required 25 networks and a fraction of the overall shader instances.



The entirely CG New York City was one of the most challenging initial trials for OSL.

Closure

This document is intended to be in part anecdotal evidence of the success of OSL in production, as well as an introduction to the nuts and bolts of a new, and evolving, shading language specification. The implementation of OSL in the public project is already very rich and versatile, in ways that I have not covered here. For example, there is functionality to read numerous file formats for use as textures or generic data, such as XML. The specification also continues to grow and evolve; there will eventually be the capacity to author closures within the OSL language itself. This alone will greatly increase the capacity for shader writers to define new BSDF's, extend existing ones, and accomplish unforeseen, potentially non-physical effects with the library.

As a language, one may find OSL lacking in certain niceties common to modern programming languages. For example, the lack of a switch statement came up early in the development of the project as a construct that we missed from C. Many of these omissions are planned to be implemented, but not prioritized in light of the fact that they amount to no significant post-optimization benefit. Switch would amount to nothing more than a series of if-else statements. Certain aspects of the OSL paradigm do prevent us from creating a truly energy conserving shading system. Some layered shading models[3] are difficult to express as closures because they would require large chains of nested closures which may be unwieldy for the renderer to traverse efficiently.

The Imageworks OSL shader library currently consists of about 136 shading nodes used to create 43 networks. These networks vary in complexity with the largest ones, such as that containing our general surface material employing 294 nodes. The majority of these nodes are generic texture nodes, and each of these can be procedurally extended in look development or lighting contexts to allow compositing of textures in the shading network. Each shading network describes a single material, and then these can also be stacked as material layers to accomplish more complex looks. All of the shading networks are assembled in Katana, and published as Katana standards with a single hand built user interface exposing only the necessary parameters for user interaction. Every show in the facility uses the same core shader library and networks. It is worth noting that the Foundry's shipping version of Katana supports all of the same OSL functionality that we enjoy at Imageworks. OSL interfaces can be viewed, and networks graphically constructed within the application.

In addition to the SPI version of the Arnold renderer, there are a handful of projects, both CPU and GPU that are implementing the language specification, if not the library directly. The hope is that as more users come to work with advanced path tracing algorithms in production, a language that can commonly speak to these methods will prove essential to their effective use.

References

[1] Martinez, A. 2010. Faster Photorealism in Wonderland. *SIGGRAPH* 5-7.
[2] Kulla, C. Fajardo, M. 2012 Importance Sampling Techniques for Path Tracing in Participating Media. *EGSR*.
[3] Weidlich, A., Wilkie A. 2007 Arbitrarily Layered Micro-Facet Surfaces. *GRAPHITE* 171-178