ShaderX: A shader generation extension to MaterialX

Niklas Harrysson
niklas.harrysson@autodesk.com

Document Draft
2018-09-03

Contents

1 Introduction **3**
- 1.1 Intent . . . 3
- 1.2 What is ShaderX? . . . 3
- 1.3 Design Goals . . . 4
  - 1.3.1 Application Agnostic . . . 4
  - 1.3.2 Data Driven . . . 4
  - 1.3.3 Physically Plausible . . . 4
  - 1.3.4 Pragmatic . . . 4

2 Shader Generation **5**
- 2.1 Scope . . . 5
- 2.2 Languages and Shader Generators . . . 5
- 2.3 Node Implementations . . . 6
  - 2.3.1 Inline Expression . . . 6
  - 2.3.2 Shading Language Function . . . 7
  - 2.3.3 Implementation by Node Graph . . . 8
  - 2.3.4 Dynamic Code Generation . . . 9
- 2.4 Shader Generation Steps . . . 10
- 2.5 Bindings and Shading Context . . . 12
- 2.6 Shader Stages . . . 12
- 2.7 Shader Variables . . . 13
  - 2.7.1 Variable Naming Convention . . . 14

3 Physical Material Model **16**
- 3.1 Scope . . . 16
- 3.2 Physically-Plausible Materials . . . 16
- 3.3 Quantities and Units . . . 16
- 3.4 Color Management . . . 17
- 3.5 Surfaces . . . 17
  - 3.5.1 Layering . . . 18
  - 3.5.2 Bump/Normal Mapping . . . 19
- 3.6 Volumes . . . 19
- 3.7 Lights . . . 19

4 ShaderX PBR Library **20**
- 4.1 Data Types . . . 20
- 4.2 BSDF Nodes . . . 20
- 4.3 EDF Nodes . . . 23
- 4.4 VDF Nodes . . . 24
- 4.5 Shader Nodes . . . 24
- 4.6 Utility Nodes . . . 25

1 Introduction

1.1 Intent

Digital content creation tools in the CGI industry all have requirements for material specification. They tend to generally agree on how geometry and animations are defined and there are open standards for this. But for various reasons materials have been hard to standardize and there are different systems for different applications. This means materials and lights which are configured for one system are not portable to another, making collaboration between applications difficult. At the same time, the needs for material design and specification are converging: film and games want to share materials, game engines are being used to preview architectural designs, and the need for material portability is clear.

MaterialX is an open source initiative by Lucasfilm that aims to solve this by creating a standard for representing the data relationships required to describe and transport materials from one application or rendering platform to another.

For the texturing domain MaterialX has a well defined standard library of nodes, including a reference implementations in OSL to demonstrate what each node is doing. However, for the shading model domain there is no description or standard library. The surface, light or volume shaders being referenced in MaterialX materials are treated as black boxes. Only the name and parameter interface is known. This means they are not easily portable between applications, and a fair amount of reverse engineering could be needed to translate from one renderer to another.

Even with the implementation details known, as in the texturing domain, integrating MaterialX into an application involves a fair amount of work. All nodes need to be implemented in a form suitable for the target application, and various translation layers need to be implemented to turn MaterialX documents into executable code and bindings in a renderer.

The intent with ShaderX is to help to solve these problems; extending MaterialX with building blocks for describing shading models, as well as supplying functionality for generating executable shader code from MaterialX node graphs. The goal is for these features to be included and distributed within MaterialX in the future.

1.2 What is ShaderX?

At a high level ShaderX comprises two different contributions to MaterialX:

1. It acts as an implementation layer for MaterialX, helping applications to transform the agnostic MaterialX data description into executable shader code for a specific renderer.
2. It adds an additional library on top of the MaterialX standard library, with data types and nodes for physically-based shading. As mentioned above MaterialX itself has no notion of a shading model. The ShaderX PBR library provides building blocks to describe this with layered physically-based shading, using an abstraction level that is similar to closures in OSL, but not limited to OSL renderers.

ShaderX aims to be somewhere between über-shaders and device-specific shaders. In the über-shader case the entire shader is a black box which makes it hard to port into a new shading language, since there can be a lot of ambiguity in how parameters such as color and roughness map to actual BSDF lobes. In the device-specific case, features that are irrelevant for the material authoring start spilling into the description. ShaderX aims to help artists express material properties, compose new materials from intuitive building blocks, etc., all while avoiding the need to specify renderer specific details.

It is not possible to execute the MaterialX/ShaderX graphs directly, rather they need to be translated into an executable language. This means that multiple applications can share a single common description and collaborate on it, even if the execution of the graph is very different in the different

applications. MaterialX already supports the definition of multiple implementations for each node; one for each back-end you want to support. This is how ShaderX is able to support multiple target renderers.

1.3 Design Goals

1.3.1 Application Agnostic

- The shader description should be portable between different applications.
- The shader description should be free from domain-specific content.
- The execution of shaders should be done by translating into an application-specific language using code generators.
- It should be possible for an editor to manipulate a shader graph, as well as to validate the correctness of a shader graph without requiring domain specific knowledge.

1.3.2 Data Driven

- Node implementations should be data to all applications.
- It should be possible to introduce new content without having to recompile the applications.
- There should be no dependencies on plugins to handle custom content from third parties.

1.3.3 Physically Plausible

- The materials created should follow the laws of physics to the extent of being physically plausible.
- Paradigms and parameters from physics should be used, with some deviations allowed where it makes sense for the purpose of artistic freedom.
- The system should be easy to fit into color managed pipelines.

1.3.4 Pragmatic

- The shader description should be pragmatic and derived from common denominators in existing shading systems, making it easy to adopt by many renderers.

2 Shader Generation

2.1 Scope

The shader generation features of ShaderX are implemented as an extension to MaterialX. A new library module named `MaterialXShaderGen` contains the core shader generation features as well as derived generators targeting specific languages.

Note that ShaderX has no runtime and the output produced is source code, not binary executable code. The source code produced needs to be compiled by a shading language compiler before being executed by the renderer. See Figure 1 for a high level overview of the system.

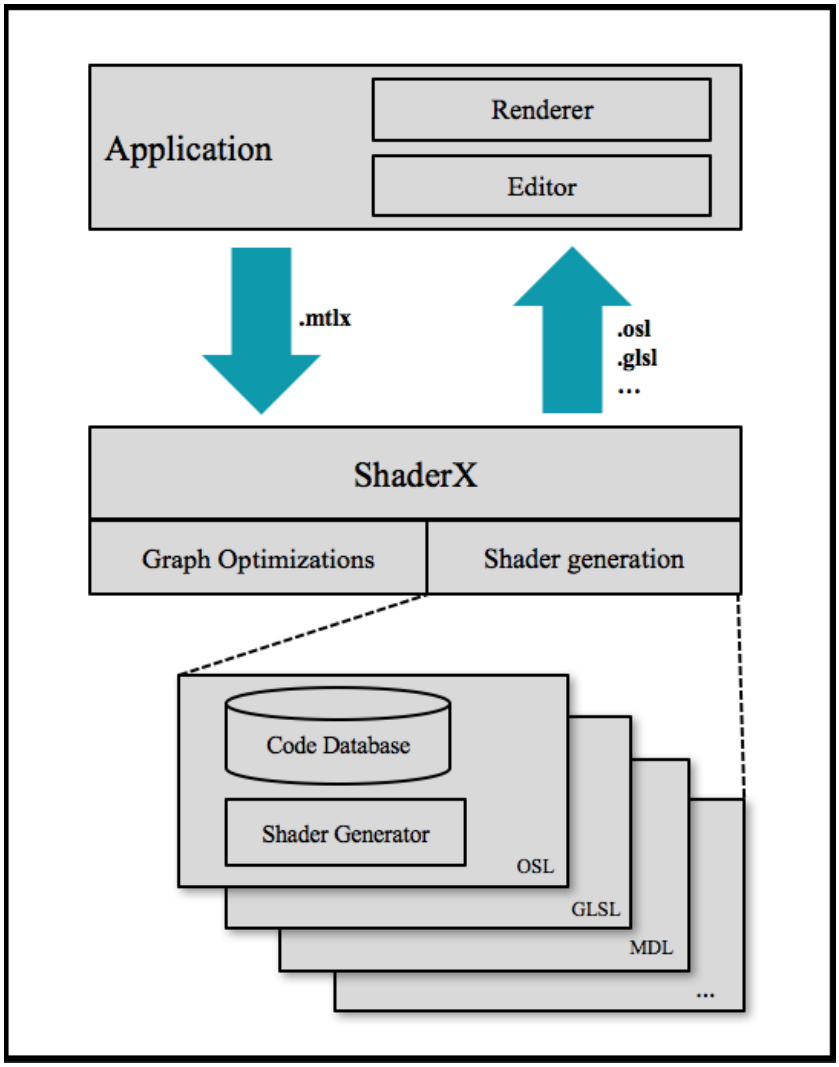

Figure 1: ShaderX with multiple shader generators.

2.2 Languages and Shader Generators

The ShaderX description is free from device specific details and all implementation details needs to be taken care of by the shader generators. There is one shader generator for each supported shading language. However for each language there can also be variations needed for different renderers.

For example; OpenGL renderers supporting GLSL can use forward rendering or deferred rendering, each with very different requirements for how the shaders are constructed. Another example is different renderers supporting OSL but with different sets of closures or closure parameters. Hence a separate shader generator can be defined for each language/target combination.

To add a new shader generator for a target you add a new C++ class derived from the base class `ShaderGenerator`, or one of the existing derived shader generator classes (`HwShaderGenerator`, `GlslShaderGenerator`, `OslShaderGenerator`, etc.), and override the methods you need to customize. You might also need to derive a new `Syntax` class, which is used to handle syntactical differences between different shading languages. Then you need to make sure there are implementations defined for all the nodes you want to support, standard library nodes and nodes from other libraries, by either reusing existing implementations where applicable or adding in new ones. See 2.3 on how that is done.

Note that a shader generator doesn't need to be defined at the time when node definitions are added. New shader generators can be added later, and node implementation for new targets can be added for existing nodes.

2.3 Node Implementations

There are four different methods to define the implementation of a node in ShaderX:

- Using an inline expression.
- Using a function written in the target language.
- Using a nodegraph that defines the operation performed by the node.
- Using a C++ class that emits code dynamically during shader generation.

For all methods the implementation is tied to a specific nodedef with a well defined interface of typed inputs and outputs. In the following sub-sections each of these methods are explained.

2.3.1 Inline Expression

ShaderX's code generators support a very simple expression language for inlining code. This is useful for nodes where the operation can be expressed as a single line of code. Inlining will reduce the number of function calls and produce more compact code. The syntax to use is the same as the target shading language, with the addition of using the node's input ports as variables wrapped in double curly brackets: {{`input`}}. The code generator will replace these variables with values assigned or connected to the respective inputs. Figure 2 gives an example.

Connecting the expression to the nodedef is done using an `<implementation>` element as seen in Figure 2. The file extension is used to differentiate inline expressions from source code functions, using "filename.inline".

```
// Nodedef elements for node <add>
<nodedef name="ND_add_float" node="add" type="float" defaultinput="in1">
  <input name="in1" type="float"/>
  <input name="in2" type="float"/>
</nodedef>
<nodedef name="ND_add_color3" node="add" type="color3" defaultinput="in1">
  <input name="in1" type="color3"/>
  <input name="in2" type="color3"/>
</nodedef>
<... more types ...>

// Implementation elements for node <add>
<implementation name="IM_add_float" nodedef="ND_add_float" file="mx_add.inline"/>
<implementation name="IM_add_color3" nodedef="ND_add_color3" file="mx_add.inline"/>
<... more types ...>

// Nodedef elements for node <mix>
<nodedef name="ND_mix_float" node="mix" type="float" defaultinput="bg">
  <input name="fg" type="float"/>
  <input name="bg" type="float"/>
  <input name="mix" type="float"/>
</nodedef>
<nodedef name="ND_mix_color3" node="mix" type="color3" defaultinput="bg">
  <input name="fg" type="color3"/>
  <input name="bg" type="color3"/>
  <input name="mix" type="float"/>
</nodedef>
<... more types ...>

// Implementation elements for node <mix>
<implementation name="IM_mix_float" nodedef="ND_mix_float" file="mx_mix.inline"/>
<implementation name="IM_mix_color3" nodedef="ND_mix_color3" file="mx_mix.inline"/>
<... more types ...>
```

```
// File 'mx_add.inline' contains:
{{in1}} + {{in2}}

// File 'mx_mix.inline' contains:
mix({{bg}}, {{fg}}, {{mix}})
```

Figure 2: Inline expressions for implementing nodes `<add>` and `<mix>`.

Inline expressions can often be reused for polymorphic implementation, as the example above. But also reused for several different shading languages, if the syntax is identical. This makes it suitable to use for implementing math nodes, arithmetic operations or calls to common built in functions like abs(), clamp(), mix(), etc.

2.3.2 Shading Language Function

For nodes that can't be implemented by inline expressions a function definition can be used instead. The function signature should match the nodedefs interface with inputs and outputs. See Figure 3 for an example. Connecting the source code to the nodedef is done using an `<implementation>` element, see [1] for more information.

```
// Nodedef element
<nodedef name="ND_image_color3" node="image" type="color3" default="0.0, 0.0, 0.0">
  <parameter name="file" type="filename"/>
  <parameter name="layer" type="string" value=""/>
  <parameter name="default" type="color3" value="0.0, 0.0, 0.0"/>
  <input name="texcoord" type="vector2" defaultgeomprop="texcoord"/>
  <parameter name="filtertype" type="string" value="linear"/>
  <parameter name="uaddressmode" type="string" value="periodic"/>
  <parameter name="vaddressmode" type="string" value="periodic"/>
  <parameter name="framerange" type="string" value=""/>
  <parameter name="frameoffset" type="integer" value="0"/>
  <parameter name="frameendaction" type="string" value="black"/>
</nodedef>

// Implementation element
<implementation name="IM_image_color3_osl" nodedef="ND_image_color3"
    file="mx_image_color3.osl" language="osl"/>
```

```
// File 'mx_image_color3.osl' contains:
void mx_image_color3(string file, string layer, color defaultvalue,
                     vector2 texcoord, string filtertype,
                     string uaddressmode, string vaddressmode,
                     string framerange, int frameoffset,
                     string frameendaction, output color out)
{
  // Sample the texture
  out = texture(file, texcoord.x, texcoord.y,
                "interp", filtertype,
                "subimage", layer,
                "missingcolor", defaultvalue,
                "wrap", uaddressmode);
}
```

Figure 3: Shading language functions implementation for node `<image>` in OSL.

2.3.3 Implementation by Node Graph

As an alternative to define source code, there is also an option to reference a nodegraph as the implementation of a nodedef. The only requirement is that the nodegraph and nodedef have matching inputs and outputs.

This is useful for creating a compound for a set of nodes performing some common operation. It can then be referenced as a node inside other nodegraphs. It is also useful for creating compatibility graphs for unknown nodes. If a node is created by some third party, and its implementation is unknown or proprietary, a compatibility graph can be created using known nodes and be referenced as a stand-in implementation. Linking a nodegraph to a nodedef is done by simply setting a `nodedef` attribute on the nodegraph definition. See Figure 4 for an example.

```
<!-- Define a node that implements a checker board pattern -->
<!-- using a graph for the implementation.                 -->
<nodedef name="ND_checker_float" node="checker" type="float">
  <input name="scale" type="vector2" value="8.0, 8.0"/>
</nodedef>
<nodegraph name="IM_checker_float" nodedef="ND_checker_float">
  <texcoord name="texcoord1" type="vector2">
    <parameter name="index" type="integer" value="0"/>
  </texcoord>
  <multiply name="mult1" type="vector2">
    <input name="in1" type="vector2" nodename="texcoord1"/>
    <input name="in2" type="vector2" interfacename="scale"/>
  </multiply>
  <swizzle name="swizz_x" type="float">
    <input name="in" type="vector2" nodename="mult1"/>
    <parameter name="channels" type="string" value="x"/>
  </swizzle>
  <swizzle name="swizz_y" type="float">
    <input name="in" type="vector2" nodename="mult1"/>
    <parameter name="channels" type="string" value="y"/>
  </swizzle>
  <floor name="floor1" type="float">
    <input name="in" type="float" nodename="swizz_x"/>
  </floor>
  <floor name="floor2" type="float">
    <input name="in" type="float" nodename="swizz_y"/>
  </floor>
  <add name="add1" type="float">
    <input name="in1" type="float" nodename="floor1"/>
    <input name="in2" type="float" nodename="floor2"/>
  </add>
  <modulo name="mod1" type="float">
    <input name="in1" type="float" nodename="add1"/>
    <input name="in2" type="float" value="2.0"/>
  </modulo>
  <output name="out" type="float" nodename="mod1"/>
</nodegraph>
```

Figure 4: Implementation using a nodegraph.

2.3.4 Dynamic Code Generation

In some situations static source code is not enough to implement a node. The code might need to be customized depending on parameters set on the node. Or for a HW render target vertex streams or uniform inputs might need to be created in order to supply the data needed for the node implementation.

In this case a C++ class can be added to ShaderX to handle the implementation of the node. The class should be derived from the base class `SgImplementation`. It should specify what language and target it is for by overriding `getLanguage()` and `getTarget()`. It can also be specified to support all languages or all targets by setting the identifier to an empty string, as done for the target identifier in the example below. It then needs to be registered for a ShaderGenerator by calling `ShaderGenerator::registerImplementation()`. See Figure 5 for an example.

When an `SgImplementation` class is used for a nodedef the corresponding `<implementation>` element don't need a `file` attribute, since no static source code is used. The `<implementation>` element will then act only as a declaration that there exists an implementation for the nodedef for a particular language and target.

```
/// Implementation of 'foo' node for OSL
class FooOsl : public SgImplementation
{
public:
  static SgImplementationPtr create() { return std::make_shared<FooOsl>(); }

  const string& getLanguage() const override { return LANGUAGE_OSL; }
  const string& getTarget() const override { return EMPTY_STRING; }

  void emitFunctionDefinition(const SgNode& node, ShaderGenerator& sg,
                              Shader& shader) override
  {
    // Emit function definition if needed for the node
  }

  void emitFunctionCall(const SgNode& node, const SgNodeContext& context,
                        ShaderGenerator& sg, Shader& shader) override
  {
    // Emit function call, or inline shader code, for the node
  }
};
```

```
OslShaderGenerator::OslShaderGenerator()
    : ShaderGenerator(std::make_shared<OslSyntax>())
{
  ...
  // Register foo implementation for nodedefs it should be used for
  registerImplementation("IM_foo_color2_osl", FooOsl::create);
  registerImplementation("IM_foo_color3_osl", FooOsl::create);
  registerImplementation("IM_foo_color4_osl", FooOsl::create);
  ...
}
```

Figure 5: C++ class for dynamic code generation.

Note that the use of `SgImplementation` classes for dynamic code generation goes against our design goal of being data driven. An application needs to be recompiled after adding a new node implementation class. However this usage is only needed in special cases. And in these cases the code produced can be made much more efficient by allowing this dynamic generation. As a result we choose to support this method of code generation, but it should only be used when inline expressions or static source code functions are not enough to handle the implementation of a node.

2.4 Shader Generation Steps

This section outlines the steps taken in general to produce a shader from the MaterialX description. The `ShaderGenerator` base class and its support classes will handle this for you, but it's good to know the steps involved in case more custom changes are needed to support a new target.

ShaderX supports generating a shader starting from either a graph output port, an arbitrary node inside a graph, or a shaderref in a material. A shader is generated by calling your shader generator class with an element of either of these types as input. The given element and all dependencies upstream will be translated into a single monolithic shader in the target shading langage.

```
// Generate a shader starting from the given element, translating
// the element and all dependencies upstream into shader code.
ShaderPtr ShaderGenerator::generate(const string& shaderName,
                                    ElementPtr element,
                                    const SgOptions& options)
```

The shader generation process can be divided into initialization and code generation. The initialization is handled by the `Shader` class, and consists of a number of steps:

1. Create an optimized version of the graph as a tree with the given element as root, and with only the used dependencies connected upstream. This involves removing unused paths in the graph, converting constant nodes to constant values, and adding in any default nodes for ports that are unconnected but has default connections specified. Removal of unused paths typically involves constant folding and pruning of conditional branches that never will be taken. Since the resulting shader in the end will be compiled by a shading language compiler, and receive a lot of additional optimizations, we don't need to do too much work in this optimization step. However a few graph level optimizations can make the resulting shader a lot smaller and save time and memory during shader compilation. It will also produce more readable source code which is good for debugging purposes.

   This step is also a good place to do other custom optimizations needed by a particular target. For example simplification of the graph, which could involve substituting expensive nodes with approximate nodes, identification of common subgraphs that can be merged, etc.

2. Track which graph interface ports are being used. It's done by finding which graph interface ports have connections to nodes inside the graph. This is useful in later steps to be able to ignore interface ports that are not used.

3. The nodes are sorted in topological order. Since a node can be referenced by many other nodes in the graph we need an ordering of the nodes so that nodes that has a dependency on other nodes comes after all nodes it depends on. This step also makes sure there are no cyclic dependencies in the graph.

4. The shader signature; its interface of uniforms and varyings are established. This consists of the graph interface ports that are in use, as well as internal ports that has been published to the interface (an example of the latter is for a HW shader generator where image texture filenames gets converted to texture samplers which needs to be published in order to be bound by the target application). This information is stored on the `Shader` class, and can be retrieved from it, together with the emitted source code when generation is completed.

5. Information about scope is tracked for each node. This information is needed to handle branching by conditional nodes. For example, if a node is used only by a particular branch on a varying conditional we want to calculate this node only inside that scope, when that corresponding branch is taken. A node can be used in global scope, in a single conditional scope or by multiple conditional scopes.

The output from the initialization step is a new graph representation constructed using the classes `SgNode`, `SgInput`, `SgOutput`, `SgNodeGraph`, etc. This is a graph representation optimized for shader generation with quick access and traversal of nodes and ports, as well as caching of extra information needed by shader generation.

After initialization the code generation steps are handled by the `ShaderGenerator` class and derived classes. This part is specific for the particular generator being used, but in general it consists of the following steps:

1. Typedefs are emitted as specified by the `Syntax` class.

2. Function definitions are emitted for all the atomic nodes that has shading language functions for their implementations. For nodes using dynamic code generation their `SgImplementation`

instances are called to generate the functions. For nodes that are implemented by graphs a function definition representing the graph computation is emitted.

3. The shader signature is emitted with all uniforms set to default values. The shader uniforms can later be accessed on the returned `Shader` instance in order for applications to be able to bind values to them.
4. The function calls for all nodes are emitted, in the right dependency order, propagating output results from upstream nodes as inputs to downstream nodes. Inline expressions are emitted instead of functions calls for nodes that use this.
5. The final shader output is produced and assigned to the shader output variable.

Note that if a single monolithic shader for the whole graph is not appropriate for your system the generator can be called on elements at any point in your graph, and generate code for sub-parts. It is then up to the application to decide where to split the graph, and to assemble the shader code for sub-parts after all have been generated.

2.5 Bindings and Shading Context

There are a number of ways to bind values to input ports on nodes and graphs. A port can be assigned a constant default value or be connected to other nodes. If a port is connected it will read the value from the upstream node. Ports can also be set to have a default node connected if the user has not made a specific connection to it. This is for example used for input ports that expect to receive geometric data from the current shading context, like normal or texture coordinates. If no such connection is made explicitly a default geometric node is connected to supply the data.

Geometric data from the current shading context is supplied using MaterialX's geometric nodes. There are a number of predefined geometric nodes in the MaterialX standard library to supply shading context data like position, normal, tangents, texture coordinates, vertex colors, etc. The vectors can be returned in different coordinate spaces: model, object or world space. If the data available from the standard geometric nodes are not enough the general purpose primvar node `geomattr` can be used to access any named data on geometry using a string identifier. It is up to the shader generator and node implementation of these geometric nodes to make sure the data is supplied, and where applicable transformed to the requested coordinate space.

2.6 Shader Stages

The `Shader` base class supports multiple shader stages. This is needed in order to generate separate code for multiple stages on HW render targets. By default the base class has only a single stage, called the *pixel stage*. But there is another sub-class `HwShader` that adds an additional stage, the *vertex stage*. If more stages are needed you can sub-class and extend on this.

When creating shader input variables you can specify which stage the variable should be used in, see 2.7 for more information on shader variable creation.

Node implementations using static source code (function or inline expressions) are always emitted to the *pixel stage*. Controlling the *vertex stage*, or other stages, is not supported using static source code. In order to do that you must use dynamic code generation with a custom `SgImplementation` sub-class for your node. You are then able to control how it affects all stages separately. Inside `emitFunctionDefinition` and `emitFunctionCall` you can add separate sections for each stage using begin/end shader stage macros. Figure 6 shows how the `texcoord` node for GLSL is emitting different code into the vertex and pixel stages.

```
/// Implementation of 'texcoord' node for GLSL
class TexCoordGlsl : public SgImplementation
{
public:
  static SgImplementationPtr create() { return std::make_shared<TexCoordGlsl>(); }

  void createVariables(const SgNode& node, ShaderGenerator& sg, Shader& s) override
  {
      HwShader& shader = static_cast<HwShader&>(s);

      const SgOutput* output = node.getOutput();
      const SgInput* indexInput = node.getInput(INDEX);
      const string index = indexInput ? indexInput->value->getValueString() : "0";

      shader.createAppData(output->type, "i_texcoord_" + index);
      shader.createVertexData(output->type, "texcoord_" + index);
  }

  void emitFunctionCall(const SgNode& node, ShaderGenerator& sg, Shader& s) override
  {
      HwShader& shader = static_cast<HwShader&>(s);

      const string& blockInstance = shader.getVertexDataBlock().instance;
      const string blockPrefix = blockInstance + ".";

      const SgInput* indexInput = node.getInput(INDEX);
      const string index = indexInput ? indexInput->value->getValueString() : "0";
      const string variable = "texcoord_" + index;

      // For the vertex stage set texcoords from the requested uv-set
      BEGIN_SHADER_STAGE(shader, HwShader::VERTEX_STAGE)
          if (!shader.isCalculated(variable))
          {
              shader.addLine(blockPrefix + variable + " = i_" + variable);
              shader.setCalculated(variable);
          }
      END_SHADER_STAGE(shader, HwShader::VERTEX_STAGE)

      // For the pixel stage return the texcoords set in the vertex stage
      BEGIN_SHADER_STAGE(shader, HwShader::PIXEL_STAGE)
          shader.beginLine();
          sg.emitOutput(node.getOutput(), true, shader);
          shader.addStr(" = " + blockPrefix + variable);
          shader.endLine();
      END_SHADER_STAGE(shader, HwShader::PIXEL_STAGE)
  }
};
```

Figure 6: Implementation of node `texcoord` in GLSL. Using an `SgImplementation` sub-class in order to control shader variable creation and code generation into separate shader stages.

2.7 Shader Variables

When generating a shader from a node graph or shaderref the inputs and parameters on those elements will be published as shader uniforms on the resulting shader. A listing of the created uniforms can be red from the produced `Shader` instance. The shader uniforms can then be presented to the user and have their values set by the application.

Adding new uniforms to a shader is done by first creating a uniform block and then adding uniforms into the block. There are two predefined uniform blocks that can be used directly, one named `PublicUniforms` and another named `PrivateUniforms`. Public is used for uniforms to be published to the user, as described above, and private is used for uniforms needed by node implementations but set by the application and not published. All uniform blocks can be queried and accessed by the application from the `Shader` instance after generation.

For inputs representing geometric data streams a separate variable block is used. In ShaderX these variables are named application data inputs (in a HW shading language they are often named varying inputs or attributes). The `Shader` class has a method for creating such variables, in a block named `AppData`. This is also accessible from the `Shader` instance after generation.

In order to pass data between shader stages in a hardware shader additional variable blocks are needed. For this purpose the `HwShader` class (derived from `Shader`) contains a variable block named `VertexData`. It is used for transporting data from the vertex stage to the pixel stage. The `HwShader` class has methods for creating such vertex data variables.

Creating shader variables and binding values to them needs to be done in agreement with the shader generator side and application side. The application must know what a variable is for in order to bind meaningful data to it. One way of handling this is by using semantics. All shader variables created in ShaderX can be assigned a semantic if that is used by the target application. ShaderX does not impose a specific set of semantics to use, so for languages and applications that use this any semantics can be used. For languages that does not use semantics a variable naming convention needs to be used instead.

Figure 6 shows how creation of shader inputs and vertex data variables are done for a node implementation that needs this.

2.7.1 Variable Naming Convention

ShaderX's built-in shader generators and accompanying node implementations are using a naming convention for its shader variables. A custom shader generator that derives from and takes advantage of built-in features should preferably use the same convention.

Uniform variables are prefixed with `u_` and application data inputs with `i_`. For languages not using semantics Figure 7 shows the naming used for variables (inputs and uniforms) with predefined binding rules:

```
--------------------------------------------------------------------------------
    NAME                                TYPE    BINDING
--------------------------------------------------------------------------------

App data input variables:
    i_position                          vec3    Vertex position in object space.
    i_normal                            vec3    Vertex normal in object space.
    i_tangent                           vec3    Vertex tangent in object space.
    i_bitangent                         vec3    Vertex bitangent in object space.
    i_texcoord_N                        vec2    Vertex texture coord for N:th uv set.
    i_color_N                           vec4    Vertex color for N:th color set.

Uniform variables:
    u_worldMatrix                       mat4    World transform.
    u_worldInverseMatrix                mat4    World transform, inverted.
    u_worldTransposeMatrix              mat4    World transform, transposed.
    u_worldInverseTransposeMatrix       mat4    World transform, inverted, transposed.
    u_viewMatrix                        mat4    View transform.
    u_viewInverseMatrix                 mat4    View transform, inverted.
    u_viewTransposeMatrix               mat4    View transform, transposed.
    u_viewInverseTransposeMatrix        mat4    View transform, inverted, transposed.
    u_projectionMatrix                  mat4    Projection transform.
    u_projectionInverseMatrix           mat4    Projection transform, inverted.
    u_projectionTransposeMatrix         mat4    Projection transform, transposed.
    u_projectionInverseTransposeMatrix  mat4    Projection transform, inverted,
                                                transposed.
    u_worldViewMatrix                   mat4    World-view transform.
    u_viewProjectionMatrix              mat4    View-projection transform.
    u_worldViewProjectionMatrix         mat4    World-view-projection transform.
    u_viewPosition                      vec3    World-space position of the viewer.
    u_viewDirection                     vec3    World-space direction of the viewer.
    u_frame                             float   The current frame number as defined
                                                by the host application.
    u_time                              float   The current time in seconds.
    u_geomattr_<name>                   <type>  A named attribute of given <type>
                                                where <name> is the name of the
                                                variable on the geometry.
    u_numActiveLightSources             int     The number of currently active
                                                light sources. Note that in shader
                                                this is clamped against the maximum
                                                allowed number of light sources.
    u_lightData[]                       struct  Array of struct LightData holding
                                                parameters for active light sources.
                                                The LightData struct is built
                                                dynamically depending on requirements
                                                for bound light shaders.
```

Figure 7: Listing of predefined variables with their binding rules.

3 Physical Material Model

This section describes the material model used in the ShaderX PBR library, and the rules we must follow to be physically plausible.

DISCLAIMER: The material model is work in progress. This design may be subject to change.

3.1 Scope

A material describes the properties of a surface or medium that involves how it reacts to light. To be efficient, a material model is split into different parts, where each part handles a specific type of light interaction: light being scattered at the surface, light being emitted from a surface, light being scattered inside a medium, etc. The goal of our material model definition is to describe light-material interactions typical for physically plausible rendering systems, including those in feature film production, real time preview, and game engines.

The current state of our model has support for surface materials, which includes scattering and emission of light from the surface of objects, and volume materials, which includes scattering and emission of light in a participating medium. For lighting we support local lights and distant light from environments. Geometric modification is supported in the form of bump and normal mapping as well as displacement mapping.

3.2 Physically-Plausible Materials

The initial requirements for a physically-plausible material are that it should be energy conserving and support reciprocity. The first requirement says that the sum of reflected and transmitted light leaving a surface must be less than or equal to the amount of light reaching it. The reciprocity requirement says that if the direction of the traveling light is reversed, the response from the material remains unchanged. That is, the response is identical if the incoming and outgoing directions are swapped. All materials implemented for ShaderX should respect these requirements and only in rare cases deviate from it when it makes sense for the purpose of artistic freedom.

3.3 Quantities and Units

Radiometric quantities are used by the material model for interactions with the renderer. The fundamental radiometric quantity is *radiance* (measured in $Wm^{-2}sr^{-1}$) and gives the intensity of light arriving at, or leaving from, a given point in a given direction. If radiance is integrated over all directions we get *irradiance* (measured in Wm^{-2}), and if we integrate this over surface area we get *power* (measured in W). Input parameters for materials and lights specified in photometric units can be suitably converted to their radiometric counterparts before being submitted to the renderer.

The interpretation of the data types returned by surface and volume shaders are unspecified, and left to the renderer and the shader generator for that renderer to decide. For an OpenGL type renderer it will be tuples of floats containing radiance calculated directly by the shader node, but for an OSL type renderer it could be closure primitives that is used by the renderer in the light transport simulation.

In general a color given as input to the renderer is considered to represent a linear RGB color space. However, there is nothing stopping a renderer from interpreting the color type differently, for instance to hold spectral values. In that case, the shader generator for that renderer needs to handle this in the implementation of the nodes involving the color type.

3.4 Color Management

MaterialX supports the use of color management systems to associate colors with specific color spaces. A MaterialX document typically specify the working color space that is to be used for the doccument as well as in what color space input values and textures are given. If these color spaces are different from the working color space it is the application and shader generators responsibility to transform them.

ShaderX has an interface that can be used to integrate support for different color management systems. A simplified implementation with some popular and commonly used color transformations are supplied and enabled by default. A full integration of OpenColorIO [2] is planned for the future.

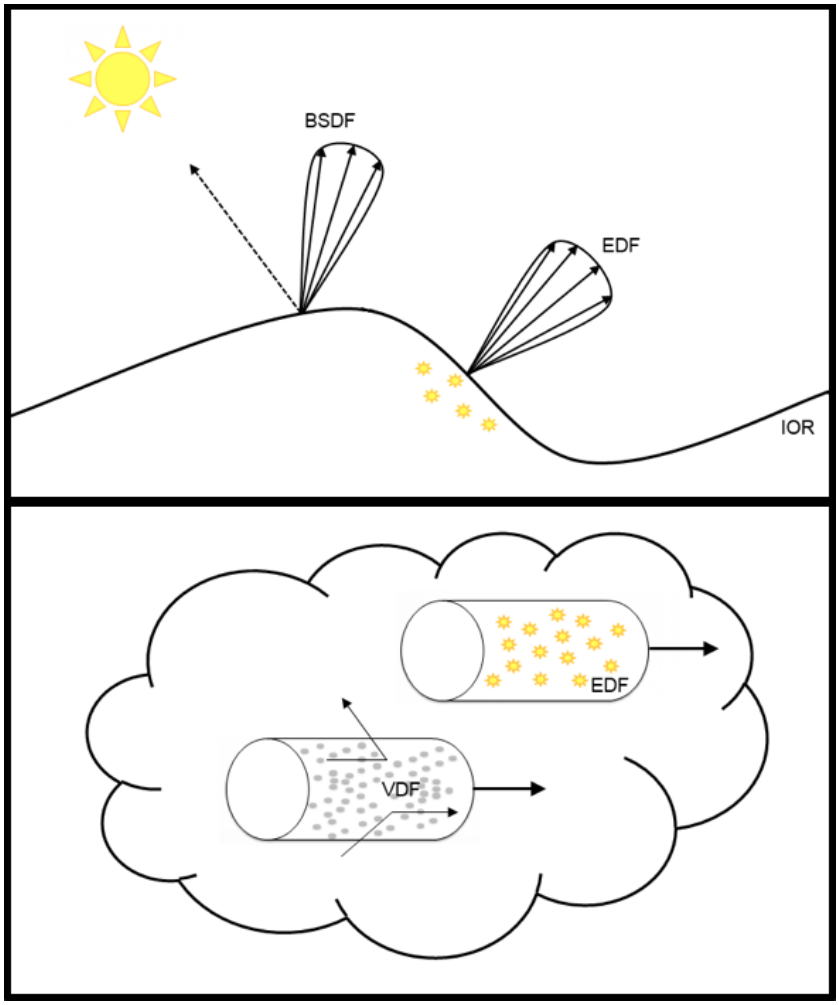

Figure 8: Distribution functions.

3.5 Surfaces

In our surface shading model the scattering and emission of light is controlled by distribution functions. Incident light can be reflected off, transmitted through, or absorbed by a surface. This is represented by a Bidirectional Scattering Distribution Function (BSDF). Light can also be emitted from a surface, for instance from a light source or glowing material. This is represented by an Emission Distribution Function (EDF). The PBR library introduce the data types `BSDF` and

`EDF` to represent the distribution functions and there are nodes for constructing, combining and manipulating them.

Another important property is the index of refraction (IOR), which describes how light is propagated through a medium. It controls how much a light ray is bent when crossing the interface between two media of different refractive indices. It also determines the amount of light that is reflected and transmitted when reaching the interface, as described by the Fresnel equations.

In reality IOR varies with the wavelength of light. But to simplify our model we use a single scalar value for dielectric surfaces, and for conductor surfaces we use an artistic parameterization given as reflectance at facing and gracing angles [6]. This parametrization is easier to work with and understand than the physical complex refraction index and extinction coefficients. However, if a physical parameterization is needed this can still be done using a node that convert to the artistic parameterization, see 4.6.

A surface shader is represented with the data type `surfaceshader`. In the PBR library there is a node that constructs a `surfaceshader` from a `BSDF` and an `EDF`. Since there are nodes to combine and modify them you can easily build surface shaders from different combinations of distribution functions. Inputs on the distribution function nodes can be connected, and nodes from the standard library can be combined into complex calculations, giving flexibility for the artist to author material variations over the surfaces.

3.5.1 Layering

In order to simplify authoring of complex materials, our model supports the notion of layering. Typical examples include: adding a layer of clear coat over a car paint material, or putting a layer of dirt or rust over a metal surface. Layering can be done in a couple of different ways:

- **Horisontal Layering:** A simple way of layering is using per-shading-point linear mixing of different BSDF's where a weight is given per BSDF controlling its contribution. Since the weight is calculated per shading point it can be used as a mask to hide contributions on different parts of a surface. The weight can also be calculated dependent on view angle to simulate approximate Fresnel behavior. This type of layering can be done both on a BSDF level and on a surface shader level. The latter is useful for mixing complete shaders which internally contains many BSDF's, e.g. to put dirt over a car paint, grease over a rusty metal or adding decals to a plastic surface. We refer to this type of layering as *horisontal layering* and the various `mix` nodes in the PBR library can be used to achive this, see chapter 4.
- **Vertical Layering:** A more physically correct form of layering is also supported where a top BSDF layer is placed over another base BSDF layer, and the light not reflected by the top layer is assumed to be transmitted to the base layer. For example adding a dielectric coating layer over a substrate. The refraction index and roughness of the coating will then affect the attenuation of light reaching the substrate. The substrate can be a transmissive BSDF to transmit the light further, or a reflective BSDF to reflect the light back up through the coating. The substrate can in turn be a reflective BSDF to simulate multiple specular lobes. We refer to this type of layering as *vertical layering* and it is modelled using a `base` BSDF input on nodes that support this type of layering, where a BSDF representing the base layer can be connected. See `dielectricbrdf`, `sheenbrdf` and `thinfilmbsdf` in chapter 4.
- **Shader Input Blending:** Calculating and blending many BSDF's or separate surface shaders can be expensive. In some situations good results can be achieved by blending the texture/value inputs instead, before any illumination calculations. Typically you would use this with an über-shader that can simulate many different materials, and by masking or blending its inputs over the surface you get the appearance of having multiple layers, but with less expensive texture or value blending.

3.5.2 Bump/Normal Mapping

The normal used for shading calculations is supplied as input to each BSDF that requires it. The normal can be perturbed by bump or normal mapping, before it is given to the BSDF. As a result you can supply different normals for different BSDFs, for the same shading point. When layering BSDF's, each layer can use different bump and normal maps.

3.6 Volumes

In our volume shader model the scattering of light in a participating medium is controlled by a volume distribution function (VDF) and coefficients controlling the rate of absorption and scattering. The VDF represents what physicists call a *phase function*. It describes how the light is distributed from its current direction when it is scattered in the medium. This is analogous to how a BSDF describes scattering at a surface, but with one important difference; a VDF is normalized, summing to 1.0 if all directions are considered. And the amount of absorption and scattering is controlled by coefficients that gives the rate (probability) per distance traveled in world space. The absorption coefficient sets the rate of absorption for light traveling through the medium, and the scattering coefficient sets the rate of which the light is scattered from its current direction. The unit for these are m^{-1}.

Light can also be emitted from a volume. This is represented by an EDF analog to emission from surfaces. But in this context the emission is given as radiance per distance traveled through the medium. The unit for this is $Wm^{-3}sr^{-1}$. The emission distribution is oriented along the current direction.

There is a node in the PBR library that constructs a volume shader from the individual VDF and EDF components. There are also nodes to construct some different VDF's as well as nodes to combine them to build more complex ones.

VDF's can also be used to describe the interior of a surface. A typical example would be to model how light is absorbed or scattered when transmitted through colored glass or turbid water. This is done by connecting a VDF as input to the BSDF node describing the surface transmission, see 4.2

3.7 Lights

Light sources can be divided into environment lights and local lights. Environment lights represent contributions coming from infinitely far away. All other lights are local lights and have a position and extent in space.

Local lights are specified as light shaders assigned to a locator, modeling an explicit light source, or in the form of emissive geometry using an emissive surface shader. There is a node in the PBR library that constructs a light shader from an EDF. There are also nodes to construct some different EDF's as well as nodes to combine them to build more complex ones. Emissive properties of surface shaders are also modelled using EDF's. See chapter 4 for more information.[1]

Light contributions coming from far away are handled by environments. These are typically photographically-captured or procedurally-generated images that surround the whole scene. This also includes light sources like the sun, where the long distance traveled makes the light almost directional and without falloff. For all shading points an environment is seen as being infinitely far away. Environments are work in progress and not yet defined in the PBR library.

[1] In a future version of the specification we could include explicit definitions for some common local light types, like *Point*, *Spot*, *RectangleArea* and *SphericalArea*.

4 ShaderX PBR Library

ShaderX provides a MaterialX library of types and nodes for creating physically-plausible materials and lights as described in chapter 3. This chapter outlines the content of that library.

DISCLAIMER: The library is work in progress. All content may be subject to change.

4.1 Data Types

- `BSDF`: Data type representing a Bidirectional Scattering Distribution Functions.
- `EDF`: Data type representing an Emission Distribution Functions.
- `VDF`: Data type representing a Volume Distribution Functions.
- `surfaceshader`: Data type representing a surface shader.
- `lightshader`: Data type representing a light shader.
- `volumeshader`: Data type representing a volume shader.
- `displacementshader`: Data type representing a displacement shader.

4.2 BSDF Nodes

A naming convention is used to differentiate BSDF nodes handling reflection and transmission. Nodes that handle reflection ends with "brdf" and nodes that handle transmission ends with "btdf". For nodes that are applicable to both reflection and transmission, for example mixing nodes an modifiers, a "bsdf" ending is used.

- `diffusebrdf`: Constructs a diffuse reflection BSDF based on the Oren-Nayar reflectance model. A roughness of 0.0 gives Lambertian reflectance. Output: BSDF
  - `weight` (input, float): Weight for this BSDF's contribution, range [0.0, 1.0]. Defaults to 1.0.
  - `color` (input, color3): Diffuse reflectivity (albedo). Defaults to (0.18,0.18,0.18)).
  - `roughness` (input, float): Surface roughness, range [0.0, 1.0]. Defaults to 0.0.
  - `normal` (input, vector3): Normal vector of the surface. Defaults to world space normal.

- `diffusebtdf`: Constructs a diffuse transmission BSDF based on the Lambert reflectance model. Output: BSDF
  - `weight` (input, float): Weight for this BSDF's contribution, range [0.0, 1.0]. Defaults to 1.0.
  - `color` (input, color3): Color transmission. Defaults to (1.0,1.0,1.0)).
  - `normal` (input, vector3): Normal vector of the surface. Defaults to world space normal.

- `conductorbrdf`: Constructs a reflection BSDF based on a microfacet reflectance model. Uses a Fresnel curve with complex refraction index for conductors/metals, but with an artistic parametrization: reflectivity and edgecolor. If a scientific complex refraction index is needed the utility node `complexior` can be connected to handle this. Output: BSDF
  - `weight` (input, float): Weight for this BSDF's contribution, range [0.0, 1.0]. Defaults to 1.0.
  - `reflectivity` (input, color3): Reflectivity per color component at facing angles. Defaults to (0.8,0.8,0.8)).
  - `edgecolor` (input, color3): Reflectivity per color component at gracing angles. Defaults to (1.0,1.0,1.0)).
  - `roughness` (input, vector2): Surface roughness. The x and y components gives roughness in tangent and bitangent directions respectively supporting anisotropic reflection. Range [0.0, 1.0]. Defaults to (0.1, 0.1).
  - `normal` (input, vector3): Normal vector of the surface. Defaults to world space normal.
  - `tangent` (input, vector3): Tangent vector of the surface. Defaults to world space tangent.
  - `distribution` (parameter, string): Microfacet distribution type. Defaults to "ggx".

- `dielectricbrdf`: Constructs a reflection BSDF based on a microfacet reflectance model and a Fresnel curve for dielectrics. A BSDF for the surface beneath can be connected to simulate a layered material with vertical layering. By chaining multiple dielectricbrdf nodes you can describe a surface with multiple specular lobes. Output: BSDF
  - `weight` (input, float): Weight for this BSDF's contribution, range [0.0, 1.0]. Defaults to 1.0.
  - `tint` (input, color3): Color weight to tint the reflected light. Defaults to (1.0,1.0,1.0). Note that changing the tint gives non-physical results and should only be done when needed for artistic purposes.
  - `ior` (input, float): Index of refraction of the surface. Defaults to 1.52. If set to 0.0 the Fresnel curve is disabled and reflectivity is controlled only by `weight` and `tint`.
  - `roughness` (input, vector2): Surface roughness. The x and y components gives roughness in tangent and bitangent directions respectively supporting anisotropic reflection. Range [0.0, 1.0]. Defaults to (0.1, 0.1).
  - `normal` (input, vector3): Normal vector of the surface. Defaults to world space normal.
  - `tangent` (input, vector3): Tangent vector of the surface. Defaults to world space tangent.
  - `distribution` (parameter, string): Microfacet distribution type. Defaults to "ggx".
  - `base` (input, BSDF, optional): BSDF for the base surface below the coating.

- `dielectricbtdf`: constructs a transmission BSDF based on a microfacet reflectance model and a Fresnel curve for dielectrics. A VDF describing the surface interior can be connected to handle absorption and scattering inside the medium, useful for colored glass, turbid water, etc. Output: BSDF
  - `weight` (input, float): Weight for this BSDF's contribution, range [0.0, 1.0]. Defaults to 1.0.

  - `tint` (input, color3): Color weight to tint the transmitted light. Defaults to (1.0,1.0,1.0)). Note that changing the tint gives non-physical results and should only be done when needed for artistic purposes. To simulate colored glass an absorption VDF can be used as interior for better physical results.
  - `ior` (input, float): Index of refraction for the surface interior. Defaults to 1.52.
  - `roughness` (input, vector2): Surface roughness. The x and y components gives roughness in tangent and bitangent directions respectively supporting anisotropic transmission. Range [0.0, 1.0]. Defaults to (0.0, 0.0).
  - `normal` (input, vector3): Normal vector of the surface. Defaults to world space normal.
  - `tangent` (input, vector3): Tangent vector of the surface. Defaults to world space tangent.
  - `distribution` (parameter, string): Microfacet distribution type. Defaults to "ggx".
  - `interior` (input, VDF, optional): VDF for the surface interior.

- `subsurfacebrdf`: Constructs a subsurface scattering BSDF for true subsurface scattering. Output: BSDF
  - `weight` (input, float): Weight for this BSDF's contribution, range [0.0, 1.0]. Defaults to 1.0.
  - `color` (input, color3): Diffuse reflectivity (albedo). Defaults to (0.18,0.18,0.18)).
  - `radius` (input, vector3): Sets the average distance that light might propagate below the surface before scattering back out. This is also known as the mean free path of the material. The radius can be set for each color component separately.
  - `anisotropy` (input, float): Anisotropy factor, controlling the scattering direction, range [-1.0, 1.0]. Negative values give backwards scattering, positive values give forward scattering, and a value of zero gives uniform scattering.
  - `normal` (input, vector3): Normal vector of the surface. Defaults to world space normal.

- `sheenbrdf`: Constructs a microfacet BSDF for the back-scattering properties of cloth-like materials. This can be layered on top of another BSDF by connecting to the base layer input. All energy that is not reflected will be transmitted to the base layer. Output: BSDF
  - `weight` (input, float): Weight for this BSDF's contribution, range [0.0, 1.0]. Defaults to 1.0.
  - `color` (input, color3): Sheen reflectivity. Defaults to (1.0,1.0,1.0)).
  - `roughness` (input, float): Surface roughness, range [0.0, 1.0]. Defaults to 0.2.
  - `normal` (input, vector3): Normal vector of the surface. Defaults to world space normal.
  - `base` (input, BSDF, optional): BSDF for the base surface below the sheen layer.

- `thinfilmbsdf`: Adds an iridescent thin film layer over a microfacet base BSDF. A connection to the base input is required, as the node is a modifier and cannot be used as a standalone BSDF. Output: BSDF
  - `thickness` (input, float): Thickness of the thin film layer.
  - `ior` (input, float): Index of refraction of the thin film layer.
  - `base` (input, BSDF, required): BSDF for the base surface below the thin film.

- `mixbsdf`: Mix two BSDF's according to a weight. Performs horisontal layering by linear interpolation between the two inputs: $in1 * (1 - weight) + in2 * weight$. Output: BSDF
  - `in1` (input, BSDF, required): The first BSDF.
  - `in2` (input, BSDF, required): The second BSDF.
  - `weight` (input, float): The mixing weight, range [0.0, 1.0].

- `scalebsdf`: Adjust the contribution of a BSDF with a weight. The weight is a color which can attenuate the channels separately. To be energy conserving the scaling weight is normalized to hold maximum 1.0 in any channel. Output: BSDF
  - `in` (input, BSDF, required): The BSDF to scale.
  - `weight` (input, color3): The scaling weight.

4.3 EDF Nodes

- `uniformedf`: Constructs an EDF emitting light uniformly in all directions. Output: EDF
  - `color` (input, color3): Radiant emittance of light leaving the surface.

- `conicaledf`: Constructs an EDF emitting light inside a cone around the normal direction. Output: EDF
  - `color` (input, color3): Radiant emittance of light leaving the surface.
  - `normal` (input, vector3): Normal vector of the surface. Defaults to world space normal.
  - `inner_angle` (parameter, float): Angle of inner cone where intensity falloff starts (given in degrees). Defaults to 60.
  - `outer_angle` (parameter, float): Angle of outer cone where intensity goes to zero (given in degrees). If set to a smaller value than inner_angle no falloff will occure whithin the cone. Defaults to 0.

- `measurededf`: Constructs an EDF emitting light according to a measured IES light profile. Output: EDF
  - `color` (input, color3): Radiant emittance of light leaving the surface.
  - `normal` (input, vector3): Normal vector of the surface. Defaults to world space normal.
  - `file` (parameter, filename, required): Path to a file containing the IES light profile data.

- `mixedf`: Mix two EDF's according to a weight. Performs linear interpolation between the two inputs: $in1 * (1 - weight) + in2 * weight$. Output: EDF
  - `in1` (input, EDF, required): The first EDF.
  - `in2` (input, EDF, required): The second EDF.
  - `weight` (input, float): The mixing weight, range [0.0, 1.0].

- `scaleedf`: Adjust the intensity of an EDF by scaling its emittance. The scaling weight is a color which can scale the channels of the emittance separately. Output: EDF
  - `in` (input, EDF, required): The EDF to scale.
  - `weight` (input, color3): The scaling weight.

4.4 VDF Nodes

- `absorptionvdf`: Constructs a VDF for pure light absorption. Output: VDF
  - `absorption` (input, color3): Absorption rate for the medium (rate per distance traveled in the medium, given in m^{-1}).

- `anisotropicvdf`: Constructs a VDF scattering light for a participating medium, based on the Henyey-Greenstein phase function. Forward, backward and uniform scattering is supported and controlled by the `anisotropy` parameter. Output: VDF
  - `absorption` (input, color3): Absorption rate for the medium (rate per distance traveled in the medium, given in m^{-1}).
  - `scattering` (input, color3): Scattering rate for the medium (rate per distance traveled in the medium, given in m^{-1}).
  - `anisotropy` (input, float): Anisotropy factor, controlling the scattering direction, range [-1.0, 1.0]. Negative values give backwards scattering, positive values give forward scattering, and a value of zero gives uniform scattering.

- `mixvdf`: Mix two VDF's according to a weight. Performs linear interpolation between the two inputs: $in1 * (1 - weight) + in2 * weight$. Output: VDF
  - `in1` (input, VDF, required): The first VDF.
  - `in2` (input, VDF, required): The second VDF.
  - `weight` (input, float): The mixing weight, range [0.0, 1.0].

- `scalevdf`: Adjust the density of a volume by scaling its absorption and scattering coefficients. The scaling weight is a color which can scale the channels of the coefficients separately. Output: VDF
  - `in` (input, VDF, required): The VDF to scale.
  - `weight` (input, color3): The scaling weight.

4.5 Shader Nodes

- `surface`: Constructs a surface shader describing light scattering and emission at surfaces. Output: surfaceshader
  - `bsdf` (input, BSDF, required): Bidirection scattering distribution function for the surface.
  - `edf` (input, EDF, required): Emission distribution function for the surface.
  - `opacity` (input, float): Cutout opacity of this surface. Default to 1.0.

- `volume`: Constructs a volume shader describing a participating medium. Output: volumeshader
  - `vdf` (input, VDF, required): Volume distribution function for the medium.
  - `edf` (input, EDF, required): Emission distribution function for the medium.

- `light`: Constructs a light shader describing an explicit light source. Output: lightshader
  - `edf` (input, EDF, required): Emission distribution function for the light source.
  - `intensity` (input, color3): Intensity multiplier for the EDF's emittance. Default to (1.0,1.0,1.0).
  - `exposure` (input, float): Exposure control for the EDF's emittance. Default to 0.0.

- `displacement`: Constructs a displacement shader describing geometric modification to surfaces. Output: displacementshader
  - `displacement` (input, vector3): Vector displacement for each position.
  - `scale` (input, float): Scale factor for the displacement vector.

- `mixsurface`: Mix two surface shaders according to a weight. Performs horisontal layering by linear interpolation between the two inputs: $in1 * (1 - weight) + in2 * weight$. Output: surfaceshader
  - `in1` (input, surfaceshader, required): The first surface shader.
  - `in2` (input, surfaceshader, required): The second surface shader.
  - `weight` (input, float): The mixing weight, range [0.0, 1.0].

4.6 Utility Nodes

- `backfacing`: Returns 1.0 if the surface being shaded is seen from the back side (or the inside of a closed object). Returns 0.0 if seen from the front or the outside of a closed object. Output: float

---

- `roughness`: Calculates an anisotropic roughness value from a scalar rougness and anisotropy parameterization. The roughness is also squared to achieve a more linear roughness look over the input parameter range of [0,1]. Output: vector2
  - `roughness` (input, float): Surface roughness, range [0.0, 1.0]. Defaults to 0.0.
  - `anisotropy` (input, float): Amount of anisotropy, range [0.0, 1.0]. Defaults to 0.0.

- `blackbody`: Returns the radiant emittance of a blackbody radiator with the given temperature. Output: color3
  - `temperature` (input, float): Temperature in Kelvin.

- `complexior`: Converts complex IOR values to the artistic parameterization `reflectivity` and `edgecolor`. Can be used with the metalbsdf node to input true complex IOR values instead of artistic reflectivity. Output: multioutput

- `ior` (input, color3): Index of refraction.
- `extinction` (input, color3): Extinction coefficient.
- `reflectivity` (output, color3): Reflectivity per color component at facing angles.
- `edgecolor` (output, color3): Reflectivity per color component at gracing angles.

References

[1] Lucasfilm, *MaterialX specification.*
http://www.materialx.org, 2018.

[2] Sony Pictures Imageworks, *OpenColorIO.*
http://opencolorio.org, 2018.

[3] Brent Burley, *Physically-Based Shading at Disney.*
http://blog.selfshadow.com/publications/s2012-shading-course/, 2012.

[4] Bruce Walter et. al. *Microfacet Models for Refraction through Rough Surfaces.*
http://www.graphics.cornell.edu/ bjw/microfacetbsdf.pdf, 2007

[5] Laurent Belcour et. al. *A Practical Extension to Microfacet Theory for the Modeling of Varying Iridescence.*
https://belcour.github.io/blog/research/2017/05/01/brdf-thin-film.html, 2017

[6] Ole Gulbrandsen, *Artist Friendly Metallic Fresnel.*
http://jcgt.org/published/0003/04/03/paper.pdf, 2014

[7] Mahdi M. Bagher et. al, *Accurate fitting of measured reflectances using a Shifted Gamma micro-facet distribution.*
http://hal.inria.fr/hal-00702304, 2012.

[8] Joakim Löw et. al, *BRDF Models for Accurate and Efficient Rendering of Glossy Surfaces.*
http://vcl.itn.liu.se/publications/2012/LKYU12/, 2012.

[9] Andrea Weidlich et. al, *Arbitrarily Layered Micro-Facet Surfaces.*
http://www.cg.tuwien.ac.at/research/publications/2007/weidlich_2007_almfs/weidlich_2007_almfs-paper.pdf, 2007.

[10] Alejandro Conty Estevez, Christopher Kulla, *Production Friendly Microfacet Sheen BRDF.*
http://blog.selfshadow.com/publications/s2017-shading-course/imageworks/s2017_pbs_imageworks_sheen.pdf