

MaterialX: An Open Standard for Network-Based CG Object Looks

Doug Smythe - smythe@ilm.com
Jonathan Stone - jstone@lucasfilm.com
June 28, 2017

Introduction

Many Computer Graphics production studios use workflows involving multiple software tools for different parts of the production pipeline. There is also a significant amount of sharing and outsourcing of work across multiple facilities, requiring companies to hand off fully look-developed models to other divisions or studios which may use different software packages and rendering systems. In addition, studio rendering pipelines that previously used monolithic shaders built by expert programmers or technical directors with fixed, predetermined texture-to-shader connections and hard-coded texture color-correction options are moving toward more flexible node graph-based shader networks built up by connecting input texture images and procedural texture generators to various inputs of shaders through a tree of image processing and blending operators.

There are at least four distinct interrelated data relationships needed to specify the complete "look" of a CG object:

1. Define the *texture processing networks* of image sources, image processing operators, connections and parameters used to combine and process one or more sources (e.g. textures) to produce the texture images that will eventually be connected to various shader inputs (e.g. "diffuse_albedo" or "bumpmap").
2. Define *geometry-specific information* such as associated texture filenames or IDs for various map types.
3. Define the parameter values and connections to texture processing networks for the inputs of one or more rendering or post-render blending shaders, resulting in a number of *materials*.
4. Define the associations between geometries in a model and materials to create number of *looks* for the model.

At the moment, there is no common, open standard for transferring all of the above data relationships. Various applications have their own file formats to store this information, but these are either closed, proprietary, inadequately documented or implemented in such a way that using them involves opening or replicating a full application.

Thus, there is a need for an open, platform-independent, well-defined standard for specifying the "look" of computer graphics objects built using shader networks so that these looks or sub-components of a look can be passed from one software package to another or between different facilities.

The purpose of this proposal is to define a schema for Computer Graphics material looks with exact operator and connection behavior for all data components, and a standalone file format for reading and writing material content using this schema. The proposal will not attempt to impose any particular shading models or any interpretation of images or data.

This proposal is not intended to represent any particular workflow or dictate data representations that a tool must support, but rather to define a flexible interchange standard compatible with many different possible ways of working. A particular tool might not support multi-layer images or even shader networks, but it could still write out looks and materials in the proposed format for interchange and read them back in.

Requirements

The following are requirements that the proposed standard *must* satisfy:

- The material schema and file format must be open and well-defined.
- Texture processing operations and their behavior must be well-defined.
- Data flow and connections must be robust and unambiguous.
- The specification must be extensible, and robustly define the processing behavior when an operator type, input or parameter is encountered that is not understood by an implementation.

The following are desirable features that would make a proposed file format easier to use, implement, and integrate into existing workflows:

- The material schema should be both expressible as a standalone file format and embeddable within file formats that support embedded material data, such as OpenEXR and Alembic.
- The file format should consist of human-readable, editable text files based upon open international standards.
- It should be possible to store parameter values and input/output filenames within material content, but it should also be possible to expose certain parameters as externally-accessible "knobs" for users to edit and/or allow external applications to read the file and supply their own values for files or parameters so that the file can be used as a template or in different contexts.
- Color computations should support modern color management systems, and it should be possible to specify the precise color space for any input image or color value, as well as the working color space for computations.
- Data types of all inputs and outputs should be explicitly specified rather than inferred in order to enable type checking upon file read rather than requiring additional information from external files or the host application to resolve ambiguous or undefined data types.

Table of Contents

Introduction	1
Proposal	5
MaterialX Overview Diagram	5
Definitions	7
MaterialX Names	8
MaterialX Data Types	8
Custom Data Types	10
MTLX File Format Definition	12
Implementation Compatibility Checking	13
Color Spaces and Color Management Systems	14
MaterialX Namespaces	15
Geometry Representation	16
Geometry and File Prefixes	16
Public Names	18
Image Filename Substitutions	18
Geometry Name Wildcards	19
Parameter Expressions and Function Curves	20
Custom Attributes, Parameters and Inputs	20
Nodes and Node Graphs	22
NodeGraph Definition	22
Inputs and Parameters	23
Output Elements	24
Standard Source Nodes	25
Texture Nodes	25
Procedural Nodes	27
Global Nodes	29
Geometric Nodes	30
Application Nodes	31
Standard Operator Nodes	32
Math Nodes	32
Adjustment Nodes	35
Compositing Nodes	36
Conditional Nodes	38
Channel Nodes	39
Convolution Nodes	40
Organization Nodes	40

Standard Node Parameters	41
Standard UI Attributes	41
NodeGraph Examples	43
Custom Nodes	48
Custom Node Declaration	48
Custom Node Definition	49
Custom Node Use	52
Shader Nodes	53
Standard Shader-Semantic Operator Nodes	55
Materials	56
Material Elements	56
MaterialInherit Elements	56
MaterialVar Elements	56
ShaderRef Elements	57
BindParam Elements	58
BindInput Elements	58
Override Elements	59
Material Examples	60
Lights	64
Collections	65
Collection Definition	65
CollectionAdd Elements	65
CollectionRemove Elements	66
Geometry Info Elements	67
GeomInfo Definition	67
GeomAttr Elements	67
GeomAttrDefault Elements	69
Reserved GeomAttr Names	69
Look and Property Elements	70
Property Definition	70
Look Definition	71
Look Assignment Elements	71
MaterialAssign Elements	71
Visibility Elements	72
PropertyAssign Elements	73
Look Examples	75

Proposal

We propose a new material content schema, **MaterialX**, along with a corresponding XML-based file format to read and write MaterialX content. The MaterialX schema defines several primary element types plus a number of supplemental and sub-element types. The primary element types are:

- **<nodegraph>** for defining graphs of data-processing nodes
- **<material>** for defining shader instances with bindings to specific uniform parameter values and spatially-varying input data streams
- **<geominfo>** for defining uniform geometric attributes that may be referenced from node graphs
- **<look>** for defining object looks, which bind materials and properties to specific geometries

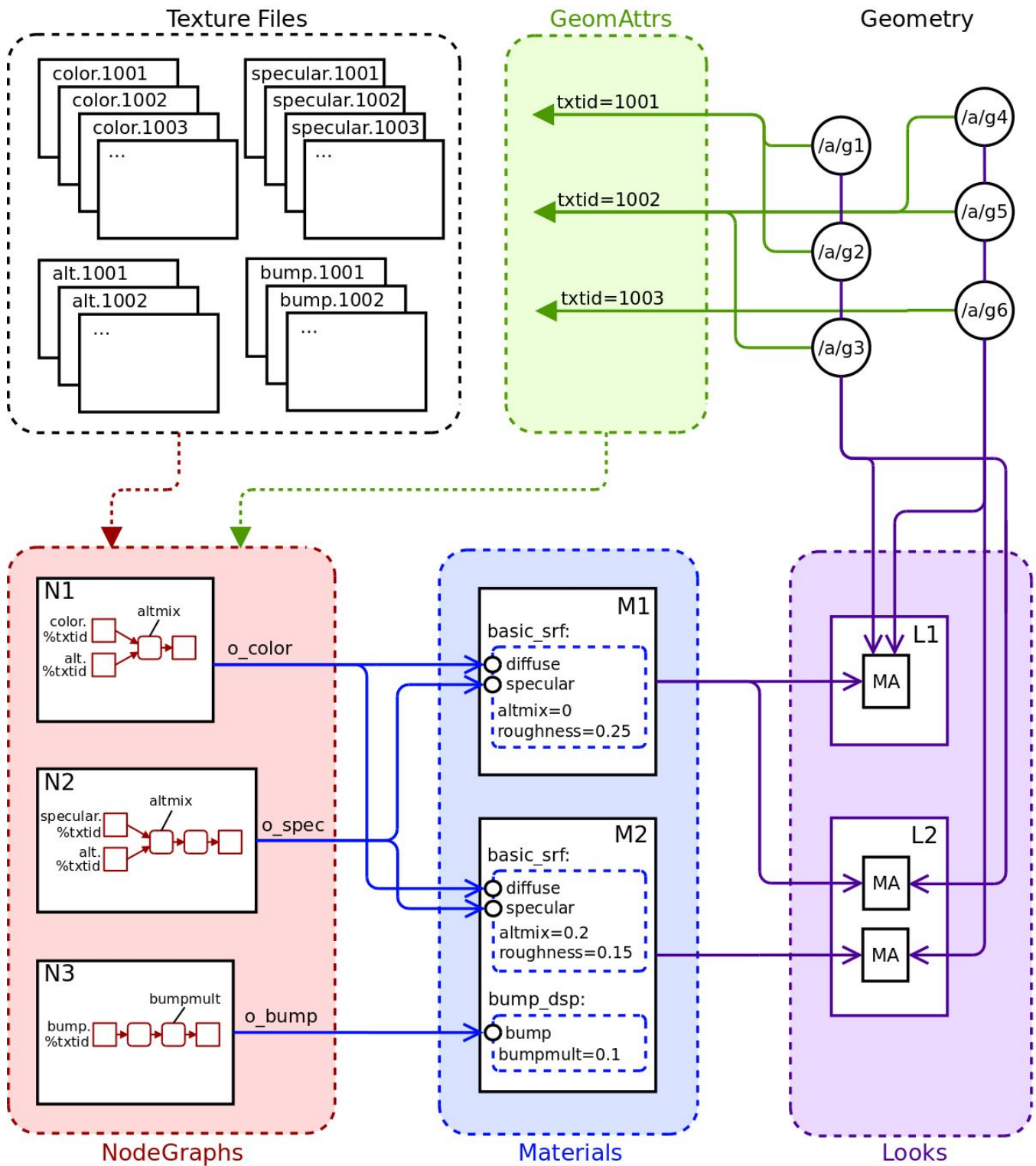
An MTLX file is a standard XML file that represents a MaterialX document, with XML elements and attributes used to represent the corresponding MaterialX elements and attributes. MTLX files may be fully self-contained, or split across several files to encourage sharing and reuse of components.

MaterialX Overview Diagram

The diagram on the following page gives a high-level overview of what each element defines and how the elements connect together to form a complete set of look definitions. Details of the **<nodegraph>**, **<geominfo>**, **<material>**, **<look>** and other elements are described in the sections that follow.

Flow of information generally proceeds counterclockwise through the diagram. The green "GeomAttrs" box shows how named attributes can be associated with geometries. The red "NodeGraphs" box defines a number of texture processing networks, which generally determine which input texture images to read by substituting GeomAttr strings defined for each geometry into a specified portion of the image file name. Rendering materials referencing one or more shaders and assigning values and input bindings to them are illustrated in the blue "Materials" box. These materials are then assigned to specified geometries via MaterialAssigns ("MA" in the diagram) as shown in the violet "Looks" box.

The example diagram defines two looks: L1 and L2. L1 uses material M1 (assigned to geometry /a/g1 through /a/g6), while L2 uses materials M1 (assigned to /a/g1, /a/g2 and /a/g3) and M2 (assigned to /a/g4, /a/g5 and /a/g6). Both materials reference the "basic_srf" shader, but M2 also references the "bump_dsp" shader. Each of the materials bind shader input connections to named outputs from nodegraphs N1, N2 and N3, but set different overriding values for the public node parameters "altmix" and (for M2) "bumpmult" as well as different value bindings for the basic_srf "roughness" parameter.



MaterialX Overview

Definitions

Because the same word can be used to mean slightly different things in different contexts, and because each studio and package has its own vocabulary, it's important to define exactly what we mean by any particular term in this proposal and use each term consistently.

An **Element** is a named object within a MaterialX document, which may possess any number of child elements and attributes. An **Attribute** is a named property of a MaterialX element.

A **Node** is a computer program that generates or processes spatially-varying data. This specification provides a set of standard nodes with precise definitions, and also supports the creation of custom nodes for application-specific uses. The interface for a node's incoming data is declared through **Parameters**, which can hold only uniform values, and **Inputs**, which may be spatially-varying.

A **Pattern** is a node that generates or processes simple scalar, vector, and color data, and has access to local properties of any geometry that has been bound. A **Shader** is a node that can generate or process arbitrary lighting or BxDF data, and has access to global properties of the scene in which it is evaluated.

A **Node Graph** is a directed acyclic graph of nodes, which may be used to define arbitrarily complex generation or processing networks. Common uses of Node Graphs are to describe a network of pattern nodes flowing to a shader input, or to define a complex or layered node in terms of simpler nodes.

A **Material** is a container for shader references, with capabilities for binding constant and spatially-varying data to the shader parameters and inputs, and for overriding the values of public parameters defined by the shaders or connected nodes.

A **Public Parameter** or **Public Input** is a parameter or input of a node tagged with a "publicname" attribute, allowing a material to override it with a new value.

A **Stream** refers to a flow of spatially-varying data from one node to another. A Stream most commonly consists of color, vector, or scalar data, but can transport data of any standard or custom type.

A **Layer** is a named 1-, 2-, 3- or 4-channel color "plane" within an image file. Image file formats that do not support multiple or named layers within a file should be treated as if the (single) layer was named "rgba".

A **Channel** is a single float value within a color or vector value, e.g. each layer of an image might have a red Channel, a green Channel, a blue Channel and an alpha Channel.

A **Geometry** is any renderable object, while a **Partition** refers to a specific named renderable subset of a piece of geometry, such as a face set.

A **Collection** is a recipe for building a list of geometries, which can be used as a shorthand for assigning a Material to a number of geometries in a Look.

A **Target** is a software environment that interprets MaterialX content to generate images, with common examples being digital content creation tools and 3D renderers.

MaterialX Names

All elements in MaterialX (nodegraphs, nodes, materials, shaders, etc.) are required to have a `name` attribute of type "string". The `name` attribute of a MaterialX element is its unique identifier, and no two elements within the same scope (i.e. elements with the same parent) may share a name. Some element types (e.g. `<bindparam>` or `<bindinput>`) serve the role of referencing an element at a different scope, and in this situation the referencing element will share a `name` with the element it references.

Element names are restricted to upper- and lower-case letters, numbers, and underscores (“_”) from the ASCII character set; all other characters and symbols are disallowed.

MaterialX Data Types

All values, input and output ports, and streams in MaterialX are strongly typed, and are explicitly associated with a specific data type. The following standard data types are defined by MaterialX:

Base Types:

`integer`, `boolean`, `float`, `color2`, `color3`, `color4`, `vector2`, `vector3`, `vector4`,
`matrix33`, `matrix44`, `string`, `filename`, `geomname`

Array Types:

`integerarray`, `floatarray`, `color2array`, `color3array`, `color4array`,
`vector2array`, `vector3array`, `vector4array`, `stringarray`, `geomnamearray`

The following examples show the appropriate syntax for MaterialX attributes in MTLX files:

Integer, Float: just a value inside quotes

```
integervalue = "1"  
floatvalue = "1.0"
```

Boolean: the lower-case word "true" or "false" inside quotes

```
booleanvalue = "true"
```

Color types: MaterialX supports three different color types:

- `color2` (red, alpha)
- `color3` (red, green, blue)
- `color4` (red, green, blue, alpha)

Color channel values should be separated by commas (with or without whitespace), within quotes:

```
color2value = "0.1,1.0"  
color3value = "0.1,0.2,0.3"  
color4value = "0.1,0.2,0.3,1.0"
```

Note: all `color3` values and the RGB components of a `color4` value are presumed to be specified in the "working color space" defined in the `<materialx>` element, and any place in a MaterialX file that a value of type `color3` or `color4` is allowed, a `colorspace` attribute can also be specified to define the color space that the value is specified in; implementations are expected to translate those color values into the working color space before performing computations with those values.

Vector types: similar to colors, MaterialX supports three different vector types:

- vector2 (x, y)
- vector3 (x, y, z)
- vector4 (x, y, z, w)

Coordinate values should be separated by commas (with or without whitespace), within quotes:

```
vector2value = "0.234,0.885"  
vector3value = "-0.13,12.883,91.7"  
vector4value = "-0.13,12.883,91.7,1.0"
```

While `colorN` and `vectorN` types both describe vectors of floating-point values, they differ in a number of significant ways. First, the final channel of a `color2` or `color4` value is interpreted as an alpha channel by compositing operators, and is only meaningful within the [0, 1] range, while the fourth channel of a `vector4` value *could be* (but is not necessarily) interpreted as the "w" value of a homogeneous 3D vector. Additionally, channel operators may apply different rules to colors than to vectors, e.g. a conversion from a `color2` to a `color4` replicates the red channel to each component of the RGB triple, but leaves the alpha channel alone. Finally, values of type `color3` and `color4` are always associated with a particular color space and are affected by color transformations, while values of type `vector3` and `vector4` are not. More detailed rules for `colorN` and `vectorN` operations may be found in the **Standard Operators** section of the specification.

Matrix types: MaterialX supports two matrix types that may be used to represent geometric and color transforms. The `matrix33` and `matrix44` types, respectively, represent 3x3 and 4x4 matrices and are written as nine or sixteen float values separated by commas, in row-major order:

```
matrix33value = "1,0,0, 0,1,0, 0,0,1"  
matrix44value = "1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1"
```

String: text within double-quotes; a single backslash (\) can be used as an escape character to allow inserting a double-quote or a backslash within the string (or to escape a comma within a string in a `stringarray`):

```
stringvalue = "some text"
```

Filename: attributes of type "filename" are just strings within double-quotes, but specifically mean a Uniform Resource Identifier (https://en.wikipedia.org/wiki/Uniform_Resource_Identifier) that represents a reference to an external asset, such as a file on disk or a query into a content management system, with image filename string substitution being performed on the string before the URI reference is resolved. For maximum portability between applications, regular filenames relative to a current working directory are generally preferred, especially for <image> filenames.

```
filevalue = "diffuse/color01.tif"  
filevalue = "/s/myshow/assets/myasset/v102.1/wetdrips/drips.$frame.tif"  
filevalue = "https://github.com/organization/project/tree/master/src/node.osl"  
filevalue = "cmsscheme:myassetdiffuse.%UDIM.tif?ver=current"
```

GeomName and **GeomNameArray**: attributes of type "geomname" are just strings within double-quotes, but specifically mean the name of a single geometry using the conventions described in the **Geometry Representation** and **Geometry Name Wildcards** sections. A geomname is allowed to use wildcards as long as it resolves to a single geometry. Attributes of type "geomnamearray" are strings within double-quotes containing a comma-separated list of one or more geomname values with or without wildcards, and may resolve to any number of geometries.

IntegerArray, FloatArray, Color2Array, Color3Array, Color4Array, Vector2Array, Vector3Array, Vector4Array, StringArray: any number of values of the same base type, separated by commas (with or without whitespace), within quotes; arrays of color2's, color3's, color4's, vector2's, vector3's or vector4's are simply a 1D list of channel values in order, e.g. "r0 g0 b0 r1 g1 b1 r2 g2 b2". To include a comma within one value of a stringarray, precede it with a '\'. MaterialX does not support 2D or nested arrays.

```
integerarrayvalue = "1,2,3,4,5"  
floatarrayvalue = "1.0, 2.2, 3.3, 4.4, 5.5"  
color2arrayvalue = "0.1,1.0, 0.2,1.0, 0.3,0.9"  
color3arrayvalue = ".1,.2,.3, .2,.3,.4, .3,.4,.5"  
color4arrayvalue = ".1,.2,.3,1, .2,.3,.4,.98, .3,.4,.5,.9"  
vector2arrayvalue = "0,.1, .4,.5, .9,1.0"  
vector3arrayvalue = "-0.2,0.11,0.74, 5.1,-0.31,4.62"  
vector4arrayvalue = "-0.2,0.11,0.74,1, 5.1,-0.31,4.62,1"  
stringarrayvalue = "hello, there, world"
```

There is also a **None** type, which is the output node type for purely organizational nodes such as <backdrop> that do not have an output.

Custom Data Types

In addition to the standard data types, MaterialX supports the specification of custom data types for the inputs and outputs of shaders and custom nodes [REQ="customtype"]. This allows documents to describe data streams of any complex type an application may require; examples might include BxDF profiles or spectral color samples. The structure of a custom type's contents may be described using a number of <member> elements, though it is also permissible to only declare the custom type's name and treat the type as "blind data".

Types can be declared to have a specific semantic, which can be used to determine how values of that type should be interpreted, and how nodes outputting that type can be connected. Currently, MaterialX defines two semantics:

- "color": the type is interpreted to represent or contain a color, and thus should be color-managed as described in the **Color Spaces and Color Management Systems** section.
- "shader": the type is interpreted as a shader output type; nodegraphs which output a type with a "shader" semantic can be used to define a shader-type node, which can be referenced by a material via a <shaderref>.

Types not defined with a specific semantic are assumed to have semantic="default".

Custom types are defined using the <typedef> element:

```
<typedef name="blindtype1"/>  
<typedef name="manifold">  
  <member name="P" type="vector3"/>  
  <member name="N" type="vector3"/>  
  <member name="du" type="vector3"/>  
  <member name="dv" type="vector3"/>  
</typedef>
```

Attributes for `<typedef>` elements:

- `name` (string, required): the name of this type. Cannot be the same as a built-in MaterialX type.
- `semantic` (string, optional): the semantic for this type (see above); the default semantic is "default".
- `context` (string, optional): a semantic-specific context in which this type should be applied. For "shader" semantic types, `context` defines the rendering context in which the shader output is interpreted; please see the **Shader Nodes** section for details.

Attributes for `typedef <member>` elements:

- `name` (string, required): the name of the member variable.
- `type` (string, required): the type of the member variable; can be any built-in MaterialX type; using custom types for `<member>` types is not supported.

If a number of `<member>` elements are provided, then a MaterialX file can specify a value for that type any place it is used, as a semicolon-separated list of numbers and strings, with the expectation that the numbers and strings between semicolons exactly line up with the expected `<member>` types in order. For example, if the following `<typedef>` was declared:

```
<typedef name="exampletype">
  <member name="id" type="integer"/>
  <member name="compclr" type="color3"/>
  <member name="objects" type="stringarray"/>
  <member name="minvec" type="vector2"/>
  <member name="maxvec" type="vector2"/>
</typedef>
```

Then a permissible parameter declaration in a custom node using that type could be:

```
<parameter name="param2" type="exampletype" value="3; 0.18,0.2,0.11; foo,bar;
0.0,1.0; 3.4,5.1"/>
```

If `<member>` child elements are not provided, e.g. if the contents of the custom type cannot be represented as a list of MaterialX types, then a value cannot be provided, and this type can only be used to pass blind data from one custom node's output to another custom node or shader input.

Once a custom type is defined by a `<typedef>`, then that type can be used in any MaterialX element that allows "any MaterialX type"; the list of MaterialX types is effectively expanded to include the new custom type.

The standard MaterialX distribution includes definitions for four "shader"-semantic data types: **surfaceshader**, **displacementshader**, **volumeshader**, and **lightshader**. These types do not define any `<member>` types, and are discussed in more detail in the **Shader Nodes** section below.

MTLX File Format Definition

An MTLX file (with file extension ".mtlx") has the following general form:

```
<?xml version="1.0" encoding="UTF-8"?>
<materialx version="major.minor">
  <!-- various combinations of MaterialX elements and sub-elements -->
</materialx>
```

That is, a standard XML declaration line followed by a <materialx> root element, which contains any number of MaterialX elements and sub-elements. The default character encoding for MTLX files is UTF-8, and this encoding is expected for the in-memory representation of string values in MaterialX implementations.

Standard XML XIncludes are supported (<http://en.wikipedia.org/wiki/XInclude>), as well as standard XML comments:

```
<xi:include href="includedfile.mtlx"/>
<!-- this is a comment -->
```

Attributes for the root <materialx> element:

- `version` (string, required): a string containing the version number of the MaterialX specification that this document conforms to, specified as a major and minor number separated by a dot. The MaterialX library automatically upgrades older-versioned documents to the current MaterialX version at load time.
- `require` (stringarray, optional): a list of implementation-dependent capabilities required by the setup contained within the <materialx> element. See the section **Implementation Compatibility Checking** below for a list of capabilities and further details.
- `cms` (string, optional): the name of the active Color Management System (CMS) : it is the responsibility of the implementation to route any color conversion through the correct CMS. Default is no color management. See the section **Color Spaces and Color Management Systems** below for further details.
- `cmsconfig` (filename, optional): the URI of a configuration file for the active CMS. This file is expected to provide the names of color spaces that may be referenced from the document, along with the transforms between these color spaces.
- `colorspace` (string, optional): the name of the "working color space" for this element and all of its descendants. This is the default color space for all image inputs and color values, and the color space in which all color computations will be performed. The default is "none", for no color management.
- `vdirection` (string, optional): tells whether the V-coordinate in uv space should be interpreted as increasing in the "up" (so uv (0,0) is the lower-left corner) or "down" (so uv-(0,0) is the upper-left corner) direction. The default is "up".
- `namespace` (string, optional): defines the namespace for all elements defined within this <materialx> scope. Please see the **MaterialX Namespaces** section below for details.

Implementation Compatibility Checking

Since different applications have different features and capabilities, it is important that any feature that may not exist or be relevant in all implementations be flagged in the MaterialX data files, so that an application reading a MaterialX file written by another application can know if that file makes use of any features it doesn't support. MaterialX does this through the presence of "require" attributes listing the specific capabilities that are needed placed immediately within the top-level <materialx> element:

```
<materialx require="multilayer,matnodegraph">  
  ...  
</materialx>
```

Here is a list of implementation-dependent capabilities currently defined for MaterialX:

convolveops	uses convolution nodes
customtype	defines and uses custom types
customnode	defines and uses custom nodes
geomops	uses nodes requiring access to local geometric features such as surface position, normal, and tangent
globalops	uses nodes requiring access to non-local geometric features
matnodegraph	uses materials with a nodegraph connected to a shader input
matvar	uses materialvar substitution in material parameter/input values
multiassign	allows multiple materials with non-overlapping types to be assigned to the same geometry.
multilayer	reads individual layers from a multilayer image file
multioutput	defines or uses custom nodes with multiple outputs
override	uses overrides on publicly-exposed parameters and/or inputs
shadergraphdef	uses shaders whose implementation is defined by a nodegraph
shadernode	uses shaders as nodes within nodegraphs

Throughout the rest of this document, [REQ="something"] is used to show that in order to use that particular feature, the named capability must be noted in a "require" attribute in its root <materialx> element to declare the need.

Color Spaces and Color Management Systems

MaterialX supports the use of color management systems to associate the RGB components of colors with specific color spaces. MaterialX documents typically specify the working color space of the application that created them, and any image file or color value described in the document can specify the name of the color space it was created in if different from the working color space. This allows applications using MaterialX to transform color values within images and parameters from their original color space into a desired working color space upon ingest (which may or may not be the same as the document's color space), and back to a specified output color space upon image output. MaterialX does not specify *how* or *when* color values may be transformed: that is up to the host application, and could involve maintaining a parallel set of pre-converted image textures, or converting color values as images are loaded into memory, or any approach appropriate for the application. It is generally presumed that the working color space of a MaterialX document will be linear (as opposed to log, sRGB or some other display-referred space, or other non-linear encoding), although this is not a firm requirement.

If a color management system (CMS) is specified using a `cms` attribute in the top-level `<materialx>` element, the implementation will use that CMS to handle all color transformations. If no CMS is specified, then all values are presumed to be used as-is. One color management system specifically supported by MaterialX is OpenColorIO (<http://opencolorio.org/>):

```
<materialx cms="ocio">
```

MaterialX implementations rely on an external CMS configuration file to define the names and interpretations of all color spaces to be referenced; MaterialX itself does not know or care what a particular color space name actually means. The standard MaterialX distribution links to the OpenColorIO configuration file for version 1.0.3 of the Academy Color Encoding System (<http://www.oscars.org/science-technology/sci-tech-projects/aces>). MaterialX documents can name this or any specific custom configuration using the `cmsconfig` attribute of the `<materialx>` element:

```
<materialx cms="ocio" cmsconfig="studio_config.ocio" colorspace="lin_rec709">
```

The working color space of a MaterialX document is defined by the `colorspace` attribute of its root `<materialx>` element, and it is strongly recommended that all `<materialx>` elements define a specific `colorspace` if they wish to use a color-managed workflow rather than relying on a default colorspace setting from an external configuration file.

The color space of individual color image files and values may be defined via a `colorspace` attribute in a parameter which defines a filename or value. Color images and values in spaces other than the working space are expected to be transformed by the application into the working space before computations are performed. In the example below, an image file has been defined in the “`srgb_texture`” color space, while its default value has been defined in “`lin_p3dci`”; both should be transformed to the application’s working color space before being applied to any computations.

```
<image name="in1" type="color3">
  <parameter name="file" type="filename" value="input1.tif"
    colorspace="srgb_texture"/>
  <parameter name="default" type="color3" value="0.5,0.5,0.5"
    colorspace="lin_p3dci"/>
</image>
```

MaterialX reserves the color space name "none" to mean no color space conversion should be applied to the images and color values within their scope.

MaterialX Namespaces

MaterialX supports the specifying of “namespaces”, which qualify the MaterialX names of all elements within their scope. Namespaces are specified via a `namespace` attribute in the root `<materialx>` element, and other MaterialX files which `<xi:include>` this .mtlx file can refer to its content without worrying about element or object naming conflicts, similar to the way namespaces are used in various programming languages. MaterialX namespaces are most commonly used to define families of custom nodes (nodedefs), material libraries, or commonly-used network shaders or nodegraphs.

Elements defined within namespaces are externally referenced using "*namespace:elementname*", where *namespace* is the value of the `namespace` attribute in the included .mtlx file's `<materialx>` element, and *elementname* is the name of the element in that file to reference.

Example:

Mtllib.mtlx contains the following (assuming that "... " contains necessary `<shaderref>` and other element definitions):

```
<?xml version="1.0" encoding="UTF-8"?>
<materialx version="major.minor" namespace="stdmaterials">
  ...
  <material name="wood">
    ...
  </material>
  <material name="plastic">
    ...
  </material>
</materialx>
```

Then another MaterialX file could reference these materials like this:

```
<xi:include href="mtllib.mtlx"/>
...
<look name="hero">
  <materialassign name="m1" material="stdmaterials:wood" collection="C_wood">
    <materialassign name="m2" material="stdmaterials:plastic"
collection="C_plastic">
  </look>
```

Similarly, if a .mtlx file defining the "site_ops" namespace defined a custom color3-typed node "mynoise" with a single float parameter "f", it could be used in an nodegraph like this:

```
<site_ops:mynoise name="mn1" type="color3">
  <parameter name="f" type="float" value="0.3"/>
</site_ops:mynoise>
```

Geometry Representation

Geometry is referenced by but not specifically defined within MaterialX content. The file in which geometry is defined can optionally be declared using `geomfile` attributes within any element; that `geomfile` declaration will then apply to any geometry name referenced within the scope of that element, e.g. any `geom` attributes, including those defining the contents of collections (but not when referencing the contents of a collection via a `collection` attribute). If a `geomfile` is not defined for the scope of any particular `geom` attribute, it is presumed that the host application can resolve the location of the geometry definition.

The geometry naming conventions used in the MaterialX specification are designed to be compatible with those used in Alembic (<http://www.alembic.io/>). "Geometry" can be any particular geometric object that a host application may support, including but not limited to polygons, meshes, subdivision surfaces, NURBS patches or meshes, implicit surfaces, particle sets, volumes, procedurally-defined objects, etc. The only requirements for MaterialX are that geometries are named using the convention specified below, can be assigned to a material and can be rendered.

The naming of geometry should follow a syntax similar to UNIX full paths:

```
/string1/string2/string3/...
```

E.g. an initial "/" followed by one or more hierarchy level strings separated by "/", ending with a final string and no "/". The strings making up the path component for a level of hierarchy cannot contain spaces or "/"s or any of the characters reserved for geometry name wildcards (see below). Individual implementations may have further restrictions on what characters may be used for hierarchy level names, so for ultimate compatibility it is recommended to use names comprised only of upper- or lower-case letters, digits 0-9, and underscores ("_").

Geometry names (e.g. the full path name) must be unique within the entire set of geometries referenced in a setup. Note that *there is no implied transformation hierarchy in the specified geometry paths*: the paths are simply the names of the geometry. However, the path-like nature of geometry names can be used to benefit in wildcard pattern matching and assignments.

Note: if a geometry mesh is divided into partitions, the syntax for the parent mesh would be:

```
/path/to/geom/meshname
```

and for the child partitions, the syntax would be:

```
/path/to/geom/meshname/partitionname
```

Geometry and File Prefixes

As a shorthand convenience, MaterialX allows the specification of a `geomprefix` attribute that will be prepended to data values of type "geomname" or "geomnamearray" (e.g. `geom` attributes in `<geominfo>`, `<collectionadd>`, `<collectionremove>`, `<materialassign>`, and `<visibility>` elements) specified within the scope of the element defining the `geomprefix`. For data values of type "geomnamearray", the `geomprefix` is prepended to each individual

comma-separated geometry name. Since the values of the prefix and the geometry are string-concatenated, the value of a `geomprefix` should generally end with a `/`. `Geomprefix` is commonly used to split off leading portions of geometry paths common to all geometry names, perhaps including package-specific conventions or an asset name.

So the following MTLX file snippets are all equivalent:

```
<materialx>
  <collection name="c_plastic">
    <collectionadd geom="/a/b/g1,/a/b/g2,/a/b/g5,/a/b/c/d/g6"/>
  </collection>
</materialx>
```

```
<materialx>
  <collection name="c_plastic" geomprefix="/a/b/">
    <collectionadd geom="g1,g2,g5,c/d/g6"/>
  </collection>
</materialx>
```

```
<materialx geomprefix="/a/b/">
  <collection name="c_plastic">
    <collectionadd geom="g1,g2,g5"/>
    <collectionadd geom="c/d/g6"/>
  </collection>
</materialx>
```

MaterialX also allows the specification of a `fileprefix` attribute which will be prepended to parameter values of type `"filename"` (e.g. `file` parameters in `<image>` nodes, or any shader parameter of type `"filename"`) specified within the scope of the element defining the `fileprefix`. Note that `fileprefix` values are only prepended to parameters with a `type` attribute that explicitly states its data type as `"filename"`, and not to attributes such as `cmsconfig` which have an implicit filename data type. Since the values of the prefix and the filename are string-concatenated, the value of a `fileprefix` should generally end with a `/`. Fileprefixes are frequently used to split off common path components for asset directories.

So the following snippets are also equivalent:

```
<nodegraph name="nodegraph1">
  <image name="in1" type="color3">
    <parameter name="file" type="filename" value="textures/color/color1.tif"/>
  </image>
  <image name="in2" type="color3">
    <parameter name="file" type="filename" value="textures/color2/color2.tif"/>
  </image>
</nodegraph>
```

```
<nodegraph name="nodegraph1" fileprefix="textures/color/">
  <image name="in1" type="color3">
    <parameter name="file" type="filename" value="color1.tif"/>
  </image>
  <image name="in2" type="color3">
    <parameter name="file" type="filename" fileprefix="textures/"
      value="color2/color2.tif"/>
  </image>
</nodegraph>
```

```
</image>
</nodegraph>
```

Note in the second example that `<image>` "in2" redefined `fileprefix` for itself, and that any other nodes in the same nodegraph would use the `fileprefix` value ("textures/color/") defined in the parent/enclosing scope.

Note: Application implementations have access to both the raw parameters and attributes (e.g. the "file" name and the current "fileprefix") and to fully-resolved filenames at the scope of any given element.

Public Names

Except where noted, any parameter or input declared in MaterialX may also be assigned a `publicname` attribute. This is a separate name for a parameter that has been exposed for modification outside of the nodegraph or shader: the nodes and shaders expose `publicnames` for parameters, and materials can define `<override>`s to set values for those parameters locally to that material. It is possible for materials to directly bind values to shader parameters using the native shader parameter names, but setting values for node parameters from materials can only be done using `<override>`s.

The main benefit of using `publicnames` is that it allows the creation of user-friendly naming and grouping of parameter values into cohesive functional blocks regardless of which element the parameter was originally defined in. For example, one could define a nodegraph with several `<image>` nodes and some texture processing operators feeding into a shader input for a specular component: using `publicnames`, all the relevant texture processing parameters such as a color multiplier, contrast amount or blend factor with some other amount could be grouped together under a "Specular" folder and be given names like "contrastAmount" instead of just the native "amount" parameter name. `Publicnames` support arbitrarily-deep folder paths using a "/" character as a separator between path components.

```
<parameter name="amount" type="color3" value="1.0,1.0,1.0"
           publicname="Specular/specColorMult"/>
...
<parameter name="amount" type="float" value="0.8"
           publicname="Specular/Texture_Mods/specContrast"/>
```

Note that if an input is given a `publicname`, and that input is connected to something, the `publicname` is no longer visible, and any `override` for that `publicname` will not affect this input.

Image Filename Substitutions

The filename for an input `image` file can include one or more special strings, which will be replaced as described in the following table. Substitution strings beginning with "%" or "@" come from the MaterialX state, while substitution strings beginning with "\$" come from the host application environment.

<i>%geomattr</i>	The value of a specific geometry attribute, defined in a <geominfo> element for the current geometry: e.g. %txtid would be replaced by the value of the "txtid" attribute on the current geometry. The geomattr value will be cast to a string before being inserted into the image filename, so integer and string geom attributes are recommended when using the <i>%geomattr</i> mechanism. Please see the Geometry Info Elements section for details.
%UDIM	A special string that will be replaced with the computed four digit Mari-style "udim" value at render or evaluation time based on the current point's uv value, using the formula $UDIM = 1001 + U + V * 10$, where U is the integer portion of the u coordinate, and V is the integer portion of the v coordinate.
%UVTILE	A special string that will be replaced with the computed Mudbox-style "_mU_nV" string, where m is 1+ the integer portion of the u coordinate, and n is 1+ the integer portion of the v coordinate.
@ <i>materialvar</i>	The value of a materialvar variable, defined in a <materialvar> element within the current <material> or <look>. Materialvar variables referenced within filenames will be cast to a string, so integer and string materialvar types are recommended.
<i>\$hostattr</i>	The host application can define other variables which can be resolved within image filenames.
\$frame	A special string that will be replaced by the current frame number, as defined by the host environment.
\$0Nframe	A special string that will be replaced by the current frame number padded with zeroes to be N digits total (replace N with a number): e.g. \$04frame will be replaced by a 4-digit zero-padded frame number such as "0010".
\$CONTAINER	A special string that will be replaced by the name of the image file in which this MaterialX content is contained. This construct can be used in MaterialX content embedded within the metadata or header of an image.

Note: Implementations are expected to properly "round trip" image file names which contain substitution strings rather than "baking them out" into specific filenames.

Geometry Name Wildcards

Certain elements in MaterialX files support geometry specification via wildcard expressions. The syntax for geometry name wildcards in MaterialX largely follows that of "glob" patterns for filenames in Unix environments, with a few extensions for the specific needs of geometry references.

Within a single hierarchy level (e.g. between "/"s):

- * matches 0 or more characters

- ? matches exactly one character
- [] are used to match any individual character within the brackets, with "-" meaning match anything between the character preceding and the character following the "-"
- {} are used to match any of the comma-separated strings or wildcard expressions within the braces

Additionally, a "/" will match only exactly a single "/" in a geometry name, e.g. as a boundary for a hierarchy level, while a "/" will match a single "/", or two "/"s any number of hierarchy levels apart; "/" can be used to specify a match at any hierarchy depth. If a geometry name ends with "/*", the final "*" will only match leaf geometries in the hierarchy. A geometry name of "/*" by itself will match all leaf geometries in an entire scene, while the name "/*/" will match all geometries at any level, including nested geometries, and the name "/a/b/c/*/" will match all geometries at any level below "/a/b/c". It should be noted that for a mesh with partitions, it is the partitions and not the mesh which are treated as leaf geometry by MaterialX geometry names using "/*". Wildcard expressions will only match scene locations representing renderable geometry, not arbitrary intermediate "parent" locations.

Parameter Expressions and Function Curves

It is noted that many packages allow material parameters to have values set by an expression or a function curve. Since the syntaxes and capabilities of various packages in use vary widely, the MaterialX specification does not currently support direct representations of parameter expressions or function curves. However, MaterialX does support baked function curve values for uniform parameters expressed as one-dimensional arrays of values per frame within a defined range by specifying `valuerange` and `valuecurve` attributes instead of `value` for any `<parameter>`, `<input>`, `<bindparameter>` or `<bindinput>` element. The `valuerange` attribute is an integer array of length 2, specifying the first and last frame number for the values in the `valuecurve`, while `valuecurve` is an array of exactly (last-first+1) values of the type specified by the `<parameter>`. A `valuecurve` value is always accessed at the "current frame" as defined by the host environment, clamped to the range of frames defined by `valuerange`. It is up to the host application to determine how to interpolate in-between-frame values if required.

```
<parameter name="amount" type="float", valuerange="16,25",
valuecurve="0, 0.5, 0.7, 0.8, 0.9, 1, 0.85, 0.75, 0.5, 0"/>
```

Custom Attributes, Parameters and Inputs

While the MaterialX specification describes the attributes and elements that are meaningful to MaterialX-compliant applications, it is permissible to add custom attributes, parameters and inputs to standard MaterialX elements. These custom attributes and child elements are ignored by applications that do not understand them, although applications should preserve and re-output them with their values and connections even if they do not understand their meaning. However, no application should depend on custom data being preserved.

If an application requires additional information related to any standard MaterialX element, it may add additional attributes with non-standard names.

```
<material name="sssmarble" maxmtlname="SSS Marble">
  <shaderref name="srl" node="marblesrf"/>
</material>
```

In the above example, a 3DSMax-specific name for a material has been provided in addition to its MaterialX-compliant name, in order to preserve the original package-specific name. It is assumed that `maxmtlname` is the attribute name used by the particular implementation for that purpose.

If an application requires the storage of custom parameters within standard MaterialX elements, it may add parameters with non-standard names. One can specify a `target` attribute to explicitly declare what target the parameter is intended for, but this is not required.

```
<image name="image1" type="color4">
  <parameter name="file" type="filename" value="image1.tif"/>
  <parameter name="filterType" type="string" value="Quadratic" target="maya"/>
  <parameter name="preFilter" type="boolean" value="true" target="maya"/>
</image>
```

In the above example, Maya-specific parameters "filterType" and "preFilter" have been added to an image node. The Maya implementation would then be responsible for passing these parameters on to whatever renderer may need them.

Applications may also add custom inputs to any of the standard source and operator nodes described in this specification. Just like standard inputs, these custom inputs can be assigned a value or be connected to another node's output, and can be given a default value to be used if unconnected. If a particular implementation does not understand a custom input, then it is expected to ignore the influence of any node connection that it has been assigned.

```
<max name="max1" type="color3">
  <input name="in1" type="color3" nodename="n2"/>
  <input name="in2" type="color3" value="0.001, 0.001, 0.001"/>
  <input name="in3" type="color3" nodename="n4" target="myshop"/>
</max>
```

In the above example, target "myshop" has an implementation of the "max" operator that accepts a third input, which has been connected to the output of node "n4".

Nodes and Node Graphs

A NodeGraph element defines an arbitrary acyclic data-processing graph applied to one or more source nodes in order to produce spatially-varying data streams for rendering or further processing. A MaterialX document can contain multiple nodegraph elements, each defining one or more output names. NodeGraphs can be shared by several different shaders or even different inputs to the same shader because each material specifies which nodegraph element and output connect to each of various shader inputs, and perhaps specifying different values for public node parameters.

A nodegraph may also be used to define a custom pattern or shader node in terms of simpler node primitives. Details on this usage of nodegraphs can be found in the **Custom Nodes** and **Shader Nodes** sections below.

NodeGraph Definition

A `<nodegraph>` element consists of at least one node element and at least one `<output>` element contained within a `<nodegraph>` element:

```
<nodegraph name="graphname" [nodedef="nodedefname" [target="target"]]>
  ...node element(s)...
  ...output element(s)...
</nodegraph>
```

The `nodedef` and `target` attributes are only applicable when the nodegraph is used as the definition of a custom node. Please see the **Custom Nodes** section for details.

Individual node elements have the form:

```
<nodetype name="nodename" type="datatype">
  <input name="paramname" type="type" [nodename="nodename"] [value="value"]/>
  <parameter name="paramname" type="type" value="value"/>
  ...additional input or parameter elements...
</nodetype>
```

where `name` (string, required) defines the name of the node, which must be unique at least within the scope of the `<nodegraph>` it appears in, and `type` (string, required) specifies the MaterialX type (typically float, color N , or vector N) of the output of that node, and thus the number of channels that the node operates on.

MaterialX defines a number of standard nodes which all implementations should support as described (with the possible exception of implementation-dependent capabilities noted with "require" attributes). One can also define new nodes by declaring their parameter interfaces and providing portable or target-specific implementations. Please see the **Custom Nodes** section for notes and implementation details.

Inputs and Parameters

Node elements contain zero or more `<input>` and `<parameter>` elements defining the name, type, and connecting nodename or value of each node input and parameter. Input elements typically define the input connections to the nodes (although they can be given a uniform constant value instead), while Parameter elements exclusively provide uniform values for the source or operator. Parameter and input elements may additionally be assigned public names, allowing them to be overridden from materials.

A node input must generally be connected to outputs of the same type, but MaterialX allows extraction of individual members of custom types, and/or the extraction or rearrangement of channels within a multichannel type.

Individual member values of custom-type node outputs can also be accessed and connected to pattern or shader node inputs of the member's type by adding a "member" attribute:

```
<custnode name="cnode4" type="exampletype"/>
<multiply name="mult6" type="color3">
  <input name="in1" type="color3" nodename="cnode4" member="compclr"/>
  <input name="in2" type="color3" value="0.6, 0.5, 0.45"/>
</multiply>
```

Inputs may also extract and/or reorder ("swizzle") the channels of multi-channel data types upon input to allow type conversion between float, color N and vector N types by adding a "channels" attribute, a string of characters indicating which channels from the incoming stream to use in each channel of the input, in order, exactly following the conventions and syntax of the **swizzle** Channel node:

```
<constant name="c4" type="color4">
  <parameter name="value" type="color4" value="0.1, 0.2, 0.3, 0.9"/>
</constant>
<constant name="v2" type="vector2">
  <parameter name="value" type="vector2" value="0.4, 0.5"/>
</constant>
<add name="add4" type="color4">
  <input name="in1" type="color4" nodename="c4" channels="rrr1"/>
  <input name="in2" type="color4" nodename="v2" channels="xy00"/>
</add>
<multiply name="m3" type="vector3">
  <input name="in1" type="vector3" nodename="add4" channels="rgb"/>
  <input name="in2" type="vector3" nodename="cnode4" member="minvec"
    channels="xy0"/>
</multiply>
```

The "member" and "channels" attributes are valid in any element that allows a "nodename" attribute. As seen in the final part of the example, the "member" and "channels" attributes may be combined to extract certain channels of an individual member of a custom type. Implementations that do not directly support input connections to sub-attributes of streams are expected to insert a **swizzle** node "behind the scenes" upon import.

Standard MaterialX nodes have exactly one output, while custom nodes may have any number of outputs; please see the **Custom Nodes** section for details.

Output Elements

Output data streams for nodegraphs are defined using **<output>** elements. An output may be used, for example, to connect data from a node graph to a shader input, or to declare an expected output for a custom node or shader implementation.

```
<output name="albedo" type="color3" nodename="n9"/>
<output name="precomp" type="color4" nodename="n13" width="1024" height="512"
    bitdepth="16"/>
```

Attributes for Output elements:

- **name** (attribute, string, required): the name of the output
- **type** (attribute, string, required): the MaterialX type of the output
- **nodename** (attribute, string, required): the name of a node within the same nodegraph, whose result value will be output by this port.
- **member** (attribute, string, optional): if **nodename** specifies a node outputting a custom type containing several members, the name of the specific member to output.
- **colorspace** (attribute, string, optional): the name of the color space for the output image. Applications that support color space management are expected to perform the required transformations of output colors into this space.
- **width** (attribute, integer, optional): the expected width in pixels of the output image.
- **height** (attribute, integer, optional): the expected height in pixels of the output image.
- **bitdepth** (attribute, integer, optional): the expected per-channel bit depth of the output image, which may be used to capture expected color quantization effects. Common values for **bitdepth** are 8, 16, 32, and 64. It is up to the application to determine what the internal representation of any declared bit depth is (e.g. scaling factor, signed or unsigned, etc.).

The **colorspace**, **width**, **height** and **bitdepth** attributes are intended to be used in applications which process nodegraphs in 2D space and save or cache outputs as images for efficiency.

Standard Source Nodes

Source nodes use external data and/or procedural functions to form an output; they do not have any required inputs. Each source node must define its output type.

This section defines the Source Nodes that all MaterialX implementations are expected to support. Standard Source Nodes are grouped into the following classifications: Texture nodes, Procedural nodes, Global nodes, Geometric nodes, and Application nodes.

Texture Nodes

Texture nodes are used to read filtered image data from image or texture map files for processing within a nodegraph.

```
<image name="in1" type="color4">
  <parameter name="file" type="filename" value="layer1.tif"/>
  <parameter name="default" type="color4" value="0.5,0.5,0.5,1"/>
</image>
<image name="in2" type="color3">
  <parameter name="file" type="filename" value="%albedomap"/>
  <parameter name="default" type="color3" value="0.18,0.18,0.18"/>
</image>
<triplanarprojection name="tri4" type="color3">
  <parameter name="filex" type="filename" value="%colorname.X.tif"/>
  <parameter name="filey" type="filename" value="%colorname.Y.tif"/>
  <parameter name="filez" type="filename" value="%colorname.Z.tif"/>
  <parameter name="default" type="color3" value="0.0,0.0,0.0"/>
</triplanarprojection>
```

Standard Texture nodes:

- **image**: samples data from a single image, or from a layer within a multi-layer image. When used in the context of rendering a geometry, the image is mapped onto the geometry based on geometry UV coordinates. Parameters and inputs:
 - **file** (parameter, filename, required): the URI of an image file. The filename can include one or more substitutions to change the file name (including frame number) that is accessed, as described in the **Image Filename Substitutions** section above. [REQ="multilayer" if multi-layer input file]
 - **layer** (parameter, string, optional): the name of the layer to extract from a multi-layer input file. If no value for **layer** is provided and the input file has multiple layers, then the "default" layer will be used, or "rgba" if there is no "default" layer. Note: the number of channels defined by the **type** of the `<image>` must match the number of channels in the named layer.
 - **default** (parameter, float or color N or vector N , optional): a default value to use if the **file** reference can not be resolved (e.g. if a %geomattr or @materialvar is included in the filename but no substitution value or default is defined, or if the resolved file URI cannot be read), or if the specified **layer** does not exist in the file. The **default** value must be the same type as the `<image>` element itself. If **default** is not defined, the default color value will be 0.0 in all channels.

- `texcoord` (input, vector2, optional): the name of a vector2-type node specifying the 2D texture coordinate at which the image data is read. Default is to use the current `u,v` coordinate.
- `uaddressmode` (parameter, string, optional): determines how U coordinates outside the 0-1 range are processed before sampling the image; see below.
- `vaddressmode` (parameter, string, optional): determines how V coordinates outside the 0-1 range are processed before sampling the image; see below.
- **triplanarprojection**: samples data from three images (or layers within multi-layer images), and projects a tiled representation of the images along each of the three respective coordinate axes, computing a weighted blend of the three samples using the geometric normal.
[REQ="geomops"] Parameters and inputs:
 - `filex` (parameter, filename, required): the URI of an image file to be projected along the x-axis.
 - `filey` (parameter, filename, required): the URI of an image file to be projected along the y-axis.
 - `filez` (parameter, filename, required): the URI of an image file to be projected along the z-axis.
 - `layerx` (parameter, string, optional): the name of the layer to extract from a multi-layer input file for the x-axis projection. If no value for `layerx` is provided and the input file has multiple layers, then the "default" layer will be used, or "rgba" if there is no "default" layer. Note: the number of channels defined by the `type` of the `<image>` must match the number of channels in the named layer.
 - `layery` (parameter, string, optional): the name of the layer to extract from a multi-layer input file for the y-axis projection.
 - `layerz` (parameter, string, optional): the name of the layer to extract from a multi-layer input file for the z-axis projection.
 - `default` (parameter, float or color N or vector N , optional): a default value to use if any `fileX` reference can not be resolved (e.g. if a `%geomattr` or `@materialvar` is included in the filename but no substitution value or default is defined, or if the resolved file URI cannot be read) The `default` value must be the same type as the `<triplanarprojection>` element itself. If `default` is not defined, the default color value will be 0.0 in all channels.
 - `position` (input, vector3, optional): a spatially-varying input specifying the 3D position at which the projection is evaluated. Default is to use the current 3D object-space coordinate.
 - `normal` (input, vector3, optional): a spatially-varying input specifying the 3D normal vector used for blending. Default is to use the current object-space surface normal.

The following values are supported by `uaddressmode` and `vaddressmode` parameters:

- “black”: Texture coordinates outside the 0-1 range return a value with 0.0 in all channels.
- “clamp”: Texture coordinates are clamped to the 0-1 range before sampling the image.
- “periodic”: Texture coordinates outside the 0-1 range "wrap around", effectively being processed by a modulo 1 operation before sampling the image. This setting is the default value for address mode parameters.

Texture nodes using `file` or `filex/y/z` parameters also support the following parameters to handle boundary conditions for image file frame ranges for all `file*` inputs:

- `framerange` (parameter, string, optional): A string "`minframe-maxframe`", e.g. "10-99", to

specify the range of frames that the image file is allowed to have, usually the range of image files on disk. Default is unbounded.

- `frameoffset` (parameter, integer, optional): A number that is added to the current frame number to get the image file frame number. E.g. if `frameoffset` is 25, then processing frame 100 will result in reading frame 125 from the imagefile sequence. Default is no frame offset.
- `frameendaction` (parameter, string, optional): What to do when the resolved image frame number is outside the `framerange` range:
 - "black": Return a value of 0 in all channels (default action)
 - "clamp": Hold the `minframe` image for all frames before `minframe` and hold the `maxframe` image for all frames after `maxframe`
 - "periodic": Frame numbers "wrap around", so after the `maxframe` it will start again at `minframe` (and similar before `minframe` wrapping back around to `maxframe`)

Arbitrary frame number expressions and speed changes are not supported.

Procedural Nodes

Procedural nodes are used to generate color data programmatically, with their inputs typically being limited to uniform parameters and position coordinate data.

```
<constant name="n8" type="color3">
  <parameter name="value" type="color3" value="0.8,1.0,1.3"/>
</constant>
<noise2d name="n9" type="float">
  <parameter name="size" type="float" value="0.003"/>
  <parameter name="pivot" type="float" value="0.5"/>
  <parameter name="amplitude" type="float" value="0.05"/>
</noise2d>
```

Standard Procedural nodes:

- **constant**: a constant value. Parameters:
 - `value` (parameter, any type, required): the value to output
- **ramp1r**: a left-to-right linear value ramp. Parameters and inputs:
 - `value1` (parameter, float or color N or vector N , required): the value at the left edge
 - `value2` (parameter, float or color N or vector N , required): the value at the right edge
 - `texcoord` (input, vector2, optional): the name of a vector2-type node specifying the 2D texture coordinate at which the ramp interpolation is evaluated. Default is to use the first set of texture coordinates.
- **ramp2b**: a top-to-bottom linear value ramp. Parameters and inputs:
 - `value1` (parameter, float or color N or vector N , required): the value at the top edge
 - `value2` (parameter, float or color N or vector N , required): the value at the bottom edge
 - `texcoord` (input, vector2, optional): the name of a vector2-type node specifying the 2D texture coordinate at which the ramp interpolation is evaluated. Default is to use the first set of texture coordinates.

- **ramp4**: a 4-corner bilinear value ramp. Parameters and inputs:
 - `valuetl` (parameter, float or color N or vector N , required): the value at the top-left corner
 - `valuetr` (parameter, float or color N or vector N , required): the value at the top-right corner
 - `valuebl` (parameter, float or color N or vector N , required): the value at the bottom-left corner
 - `valuebr` (parameter, float or color N or vector N , required): the value at the bottom-right corner
 - `texcoord` (input, vector2, optional): the name of a vector2-type node specifying the 2D texture coordinate at which the ramp interpolation is evaluated. Default is to use the first set of texture coordinates.

- **splitlr**: a left-right split matte, split at a specified u value. Parameters and inputs:
 - `valueL` (parameter, float or color N or vector N , required): the value at the left edge
 - `valueR` (parameter, float or color N or vector N , required): the value at the right edge
 - `center` (parameter, float, required): a value representing the u-coordinate of the split; all pixels to the left of "center" will be `valueL`, all pixels to the right of "center" will be `valueR` (antialiased, of course).
 - `texcoord` (input, vector2, optional): the name of a vector2-type node specifying the 2D texture coordinate at which the split position is evaluated. Default is to use the first set of texture coordinates.

- **splittb**: a top-bottom split matte, split at a specified v value. Parameters and inputs:
 - `valueT` (parameter, float or color N or vector N , required): the value at the top edge
 - `valueB` (parameter, float or color N or vector N , required): the value at the bottom edge
 - `center` (parameter, float, required): a value representing the v-coordinate of the split; all pixels above "center" will be `valueT`, all pixels below "center" will be `valueB` (antialiased, of course).
 - `texcoord` (input, vector2, optional): the name of a vector2-type node specifying the 2D texture coordinate at which the split position is evaluated. Default is to use the first set of texture coordinates.

- **noise2d**: 2D Perlin noise in 1, 2, 3 or 4 channels. Parameters and inputs:
 - `amplitude` (parameter, float or vector N , optional): the center-to-peak amplitude of the noise (peak-to-peak amplitude is 2x this value). Default is 1.0.
 - `pivot` (parameter, float, optional): the center value of the output noise; effectively, this value is added to the result after the Perlin noise is multiplied by `amplitude`. Default is 0.0.
 - `texcoord` (input, vector2, optional): the name of a vector2-type node specifying the 2D texture coordinate at which the noise is evaluated. Default is to use the first set of texture coordinates.

- **noise3d**: 3D Perlin noise in 1, 2, 3 or 4 channels [REQ="geomops"]. Parameters and inputs:
 - `amplitude` (parameter, float or vector N , optional): the center-to-peak amplitude of the noise (peak-to-peak amplitude is 2x this value). Default is 1.0.
 - `pivot` (parameter, float, optional): the center value of the output noise; effectively, this value is added to the result after the Perlin noise is multiplied by `amplitude`. Default is 0.0.

- `position` (input, vector3, optional): the name of a vector3-type node specifying the 3D position at which the noise is evaluated. Default is to use the current 3D object-space coordinate.
- **fractal3d**: Zero-centered 3D Fractal noise in 1, 2, 3 or 4 channels, created by summing several octaves of 3D Perlin noise, increasing the frequency and decreasing the amplitude at each octave. [REQ="geomops"]. Parameters and inputs:
 - `amplitude` (parameter, float or vectorN, optional): the center-to-peak amplitude of the noise (peak-to-peak amplitude is 2x this value). Default is 1.0.
 - `octaves` (parameter, integer, optional): the number of octaves of noise to be summed. Default is 3.
 - `lacunarity` (parameter, float, optional): the exponential scale between successive octaves of noise. Default is 2.0.
 - `diminish` (parameter, float, optional): the rate at which noise amplitude is diminished for each octave. Default is 0.5.
 - `position` (input, vector3, optional): the name of a vector3-type node specifying the 3D position at which the noise is evaluated. Default is to use the current 3D object-space coordinate.
- **cellnoise2d**: 2D cellular noise, 1 channel (type float). Inputs:
 - `texcoord` (input, vector2, optional): the name of a vector2-type node specifying the 2D position at which the noise is evaluated. Default is to use the first set of texture coordinates.
- **cellnoise3d**: 3D cellular noise, 1 channel (type float) [REQ="geomops"]. Inputs:
 - `position` (input, vector3, optional): the name of a vector3-type node specifying the 3D position at which the noise is evaluated. Default is to use the current 3D object-space coordinate.

The `ramp4`, `ramp4`, `split4`, `noise2d` and `cellnoise2d` sources all respect the setting of the global `<materialx>` element `"vdirection"` attribute to determine if the "t"op is `v=1` (`vdirection="up"`, the default), or `v=0` (`vdirection="down"`). Also note that for the various `ramp`, `split`, and `noise` sources, the integer portions of the specified texture coordinates are effectively discarded (performing a "modulo 1.0") for the purposes of interpolating left/right/top/bottom values or for comparison with the `center` value.

To scale or offset the noise pattern generated by `noise3d`, `fractal3d` or `cellnoise3d`, use a `<position>` or other Geometric node (see below) connected to vector3 `<multiply>` and/or `<add>` nodes, in turn connected to the noise node's `position` input. To scale or offset `rampX`, `splitX`, `noise2d` or `cellnoise2d` input coordinates, use a `<texcoord>` or similar Geometric node processed by vector2 `<multiply>`, `<scale>`, `<rotate2d>` and/or `<add>` nodes, and connect to the node's `texcoord` input.

Global Nodes

Global nodes generate color data using non-local geometric context, requiring access to geometric features beyond the surface point being processed. This non-local context can be provided by tracing rays into the scene, rasterizing scene geometry, or any other appropriate method.

```
<ambientocclusion name="occl1" type="float">
  <parameter name="maxdistance" type="float" value="10000.0"/>
</ambientocclusion>
```

Standard Global nodes:

- **ambientocclusion**: Compute the ambient occlusion at the current surface point, returning a scalar value between 0 and 1 [REQ="globalops"]. Ambient occlusion represents the accessibility of each surface point to ambient lighting, with larger values representing greater accessibility to light. This node must be of type float.
 - **coneangle** (parameter, float, optional): the half-angle of a cone about the surface normal, within which geometric surface features are considered as potential occluders. The unit for this parameter is degrees, and its default value is 90.0 (full hemisphere).
 - **maxdistance** (parameter, float, optional): the maximum distance from the surface point at which geometric surface features are considered as potential occluders. Defaults to unlimited.

Geometric Nodes

Geometric nodes are used to reference local geometric properties from within a nodegraph:

```
<position name="wp1" type="vector3" space="world"/>
<texcoord name="c1" type="vector2">
  <parameter name="index" type="integer" value="1"/>
</texcoord>
```

Standard Geometric nodes:

- **position**: the coordinates associated with the currently-processed data, as defined in a specific coordinate space [REQ="geomops"]. This node must be of type vector3.
 - **space** (parameter, string, optional): the name of the coordinate space in which the position is defined. See the section below for details.
- **normal**: the geometric normal associated with the currently-processed data, as defined in a specific coordinate space [REQ="geomops"]. This node must be of type vector3.
 - **space** (parameter, string, optional): the name of the coordinate space in which the normal vector is defined. See the section below for details.
- **tangent**: the geometric tangent vector associated with the currently-processed data, as defined in a specific coordinate space [REQ="geomops"]. This node must be of type vector3.
 - **space** (parameter, string, optional): the name of the coordinate space in which the tangent vector is defined. See the section below for details.
- **bitangent**: the geometric bitangent vector associated with the currently-processed data, as defined in a specific coordinate space [REQ="geomops"]. This node must be of type vector3.
 - **space** (parameter, string, optional): the name of the coordinate space in which the bitangent vector is defined. See the section below for details.

- **texcoord**: the 2D or 3D texture coordinates associated with the currently-processed data. This node must be of type vector2 or vector3 [REQ="geomops" for vector3 variant].
 - `index` (parameter, integer, optional): the index of the texture coordinates to be referenced. The default index is 0.
- **geomcolor**: the color associated with the current geometry at the current `position`, generally bound via per-vertex color values [REQ="geomops"]. Can be of type float, color2, color3 or color4, and must match the type of the "color" bound to the geometry.
 - `index` (parameter, integer, optional): the index of the color to be referenced, default is 0.
- **geomattrvalue**: the value assigned to the currently-bound geometry through the specified `<geomattr>` name [REQ="geomops"].
 - `attrname` (parameter, string, required): the name of the `<geomattr>` to be referenced.

The following values are supported by the `space` parameters of Geometric nodes:

- "model": The local coordinate space of the geometry, before any local deformations or global transforms have been applied.
- "object": The local coordinate space of the geometry, after local deformations have been applied, but before any global transforms. This is the default value for `space` parameters.
- "world": The global coordinate space of the geometry, after local deformations and global transforms have been applied.

Application Nodes

Application nodes are used to reference application-defined properties within a nodegraph, and have no inputs:

```
<frame name="f1" type="float"/>
<time name="t1" type="float"/>
```

Standard Application nodes:

- **frame**: the current frame number as defined by the host environment. This node must be of type float. Applications may use whatever method is appropriate to communicate the current frame number to the `<frame>` node's implementation, whether via an internal state variable, a custom parameter, or other method.
- **time**: the current time in seconds, as defined by the host environment. This node must be of type float.
 - `fps` (parameter, float, optional): the number of frames per second for the frame to time conversion. The default value is 24.0. Applications may use whatever method is appropriate to communicate the current time to the `<time>` node's implementation, whether via an internal state variable, a custom parameter, or other method.

Standard Operator Nodes

Operator nodes process one or more required input streams to form an output. Like other nodes, each operator must define its output type, which in most cases also determines the type(s) of the required input streams.

```
<multiply name="n7" type="color3">
  <input name="in1" type="color3" nodename="n5"/>
  <input name="in2" type="float" value="2.0"/>
</multiply>
<over name="n11" type="color4">
  <input name="fg" type="color4" nodename="n7"/>
  <input name="bg" type="color4" nodename="inbg"/>
</over>
<hueshift name="n12" type="color3">
  <input name="in" type="color3" nodename="n4"/>
  <parameter name="amount" type="float" value="0.1"/>
</hueshift>
<add name="n2" type="color3">
  <input name="in1" type="color3" nodename="n12"/>
  <input name="in2" type="color3" nodename="img4"/>
</add>
```

The inputs of compositing operators are called "fg" and "bg" (plus "alpha" for float and color3 variants, and "mask" for all variants of the `mix` operator), while the inputs of other operators are called "in" if there is exactly one input, or "in1", "in2" etc. if there are more than one input. If an implementation does not support a particular operator, it should pass through the "bg", "in" or "in1" input unchanged.

This section defines the Operator Nodes that all MaterialX implementations are expected to support. Standard Operator Nodes are grouped into the following classifications: [Math nodes](#), [Adjustment nodes](#), [Compositing nodes](#), [Conditional nodes](#), [Channel nodes](#), and [Convolution nodes](#).

Math Nodes

Math nodes have one or two spatially-varying inputs and a number of uniform parameters, and are used to perform a math operation on values in one spatially-varying input stream, or to combine two spatially-varying input streams using a specified math operation. The given math operation is performed for each channel of the input stream(s), and the data type of each parameter must either match that of the input stream(s), or be a float value that will be applied to each channel separately.

- **add**: add a value to the incoming float/color/vector. See also the **Shader Nodes** section below for additional `add` variants supporting shader-semantic types.
 - `in1` (input, float or color N or vector N , required): the name of the node to connect to the primary input
 - `in2` (input, same type as `in1` or float, required): the value to add
- **subtract**: subtract a value from the incoming float/color/vector, outputting "in1-in2".
 - `in1` (input, float or color N or vector N , required): the name of the node to connect to the primary input
 - `in2` (input, same type as `in1` or float, required): the value to subtract

- **multiply**: multiply an incoming float/color/vector by a value. See also the **Shader Nodes** section below for additional `multiply` variants supporting shader-semantic types.
 - `in1` (input, float or color N or vector N , required): the name of the node to connect to the primary input
 - `in2` (input, same type as `in1` or float, required): the value to multiply by
- **divide**: divide an incoming float/color/vector by a value; dividing a channel value by 0 results in floating-point "NaN".
 - `in1` (input, float or color N or vector N , required): the name of the node to connect to the primary input
 - `in2` (input, same type as `in1` or float, required): the value to divide by
- **modulo**: the remaining fraction after dividing an incoming float/color/vector by a value and subtracting the integer portion.
 - `in1` (input, float or color N or vector N , required): the name of the node to connect to the primary input
 - `in2` (input, same type as `in1` or float, required): the modulo value to divide by, cannot be 0 in any channel
- **invert**: subtract the incoming float/color/vector from "amount" in all channels, outputting "amount-in".
 - `in` (input, float or color N or vector N , required): the name of the node to connect to the input
 - `amount` (parameter, same type as `in` or float, optional): the value to subtract the input value from; default is 1.0 in all channels
- **absval**: the absolute value of the incoming float/color/vector.
 - `in` (input, float or color N or vector N , required): the name of the node to connect to the input
- **floor**: the per-channel nearest integer value less than or equal to the incoming float/color/vector; the output remains in floating point per-channel, i.e. the same type as the input.
 - `in` (input, float or color N or vector N , required): the name of the node to connect to the input
- **exponent**: raise incoming float/color values to the specified exponent, commonly used for "gamma" adjustment. Negative input values will be left unchanged.
 - `in1` (input, float or color N or vector N , required): the name of the node to connect to the primary input
 - `in2` (input, same type as `in` or float, required): exponent value; output = $\text{pow}(\text{in1}, \text{in2})$
- **clamp**: clamp incoming values to a specified range of color values.
 - `in` (input, float or color N or vector N , required): the name of the node to connect to the input
 - `low` (parameter, same type as `in` or float, optional): clamp low value; any value lower than this will be set to "low" (default value=0)
 - `high` (parameter, same type as `in` or float, optional): clamp high value; any value higher than this will be set to "high" (default value=1)

- **min**: select the minimum among incoming values
 - `in1` (input, float or color N or vector N , required): the name of the node to connect to the primary input
 - `in2` (input, same type as `in1` or float, required): the secondary value or input name
- **max**: select the maximum among incoming values
 - `in1` (input, float or color N or vector N , required): the name of the node to connect to the primary input
 - `in2` (input, same type as `in1` or float, required): the secondary value or input name
- **normalize**: outputs the normalized vector N from the incoming vector N stream; cannot be used on float or color N streams. Note: the fourth channel in vector4 streams is not treated any differently, e.g. not as a homogeneous "w" value.
 - `in` (input, vector N , required): the name of the node to connect to the input
- **magnitude**: outputs the float magnitude (vector length) of the incoming vector N stream; cannot be used on float or color N streams. Note: the fourth channel in vector4 streams is not treated any differently, e.g. not as a homogeneous "w" value.
 - `in` (input, vector N , required): the name of the node to connect to the input.
- **dotproduct**: outputs the (float) dot product of two incoming vector N streams; cannot be used on float or color N streams.
 - `in1` (input, vector N , required): the name of the node to connect to the primary input.
 - `in2` (input, same type as `in1`, required): the secondary value or input name
- **crossproduct**: outputs the (vector3) cross product of two incoming vector3 streams; cannot be used on any other stream type. A disabled `crossproduct` node passes through the value of `in1` unchanged.
 - `in1` (input, vector3, required): the name of the node to connect to the primary input.
 - `in2` (input, vector3, required): the secondary value or input name
- **scale**: scales a vector2 or vector3 value about a center point in 2D or 3D pattern space. The input `u,v` or `x,y,z` coordinates are divided by `amount`, so an `amount` greater than 1.0 will effectively make the image or pattern bigger.
 - `in` (input, vector2 or vector3, required): the name of the node to connect to the input
 - `amount` (parameter, same type as `in` or float, required): the pattern-space scaling factor to apply; specifying a float will scale all channels uniformly.
 - `center` (parameter, same type as `in`, optional): the center coordinate about which to scale. Defaults to (0.5, 0.5) for vector2, or (0.5, 0.5, 0.5) for vector3.
- **rotate2d**: rotates a vector2 value about a center point in 2D pattern space. The input `u,v` texture coordinates are rotated clockwise by `amount`, which effectively rotates the image or pattern counter-clockwise.
 - `in` (input, vector2, required): the name of the node to connect to the input
 - `amount` (parameter, float, required): the amount to rotate, specified in degrees
 - `center` (parameter, vector2, optional): the center coordinate about which to rotate. Defaults to (0.5, 0.5).

Adjustment Nodes

Adjustment nodes have one input named "in", and apply a specified function to values in the incoming stream.

- **contrast**: increase or decrease contrast of incoming float/color values using a linear slope multiplier.
 - `in` (input, float or color N or vector N , required): the name of the node to connect to the input
 - `amount` (parameter, same type as `in` or float, required): slope multiplier for contrast adjustment, 0.0 to infinity range. Values greater than 1.0 increase contrast, values between 0.0 and 1.0 reduce contrast
 - `pivot` (parameter, same type as `in` or float, optional): center pivot value of contrast adjustment; this is the value that will not change as contrast is adjusted (default value=0.5)
- **remap**: remap incoming values from one range of float/color/vector values to another, optionally applying a gamma correction "in the middle". Input values below `inlow` or above `outhigh` are extrapolated unless `doclamp` is true.
 - `in` (input, float or color N or vector N , required): the name of the node to connect to the input
 - `inlow` (parameter, same type as `in` or float, optional): low value for input range (default value=0)
 - `inhigh` (parameter, same type as `in` or float, optional): high value for input range (default value=1)
 - `gamma` (parameter, same type as `in` or float, optional): exponent applied to input value after first transforming from `inlow..inhigh` to 0..1 (default value=1)
 - `outlow` (parameter, same type as `in` or float, optional): low value for output range (default value=0)
 - `outhigh` (parameter, same type as `in` or float, optional): high value for output range (default value=1)
 - `doclamp` (parameter, boolean, optional): If true, the output is clamped to the range `outlow..outhigh` (default is false)
- **smoothstep**: outputs a smooth (hermite-interpolated) remapping of input values from low-high to output 0-1.
 - `in` (input, float or color N or vector N , required): the name of the node to connect to the input
 - `low` (parameter, same type as `in` or float, optional): input low value; an input value of this or lower will result in an output value of 0 (default value=0)
 - `high` (parameter, same type as `in` or float, optional): input high value; an input value of this or higher will result in an output value of 1 (default value=1)
- **hueshift**: (color3 or color4 only) shift the hue of a color; the alpha channel will be unchanged if present.
 - `in` (input, color3/color4, required): the name of the node to connect to the input
 - `amount` (parameter, float, required): amount of hue shift; the `hueshift` operator transforms the incoming color to HSV space, adds `amount` to the Hue, and then transforms it back to RGB space. A positive "amount" rotates hue in the "red to green to blue" direction, with amount of 1.0 being the equivalent to a 360 degree (e.g. no-op) rotation. Negative or greater-than-1.0 values of `amount` are allowed, wrapping at the 0-1 boundaries.

- **saturate**: (color3 or color4 only) adjust the saturation of a color; the alpha channel will be unchanged if present.
 - `in` (input, color3/color4, required): the name of the node to connect to the input
 - `amount` (parameter, float, required): a multiplier for saturation; the saturate operator performs a linear interpolation between the luminance of the incoming color value (copied to all three color channels) and the incoming color value itself. Note that setting amount to 0 will result in an R=G=B gray value equal to the value that the `luminance` node (below) returns.
 - `lumacoeffs` (parameter, color3, optional): the luma coefficients of the current working color space; if no specific color space can be determined, the ACEScg (ap1) luma coefficients [0.272287, 0.6740818, 0.0536895] will be used. Applications which support color management systems may choose to retrieve this value from the CMS to pass to the <saturate> node's implementation directly, rather than exposing it to the user.
- **luminance**: (color3 or color4 only) output a grayscale value containing the luminance of the incoming RGB color in all color channels, computed using the dot product of the incoming color with the luma coefficients of the active CMS configuration; the alpha channel is left unchanged if present.
 - `in` (input, color3/color4, required): the name of the node to connect to the input
 - `lumacoeffs` (parameter, color3, optional): the luma coefficients of the current working color space; if no specific color space can be determined, the ACEScg (ap1) luma coefficients [0.272287, 0.6740818, 0.0536895] will be used. Applications which support color management systems may choose to retrieve this value from the CMS to pass to the <luminance> node's implementation directly, rather than exposing it to the user.

Compositing Nodes

Compositing nodes have two (required) inputs named "fg" and "bg", and apply a function to combine them. Compositing nodes are split into five subclassifications: Premult nodes, Blend nodes, Merge nodes, Masking nodes, and the Mix node.

Premult nodes have one input named "in" (and for 3-channel variants, an additional "alpha" input), and either apply or unapply the alpha to the float or RGB color.

For 2-channel (color2) or 4-channel (color4) inputs/outputs:

- **premult**: Multiply the R or RGB channels of the input by the Alpha channel of the input.
 - `in` (input, color2 or color4, required): the name of the node to connect to the input
- **unpremult**: Divide the R or RGB channels of the input by the Alpha channel of the input. If the input has 1 or 3 channels, or if the Alpha value is zero, it is passed through unchanged.
 - `in` (input, color2 or color4, required): the name of the node to connect to the input

For 3-channel (color3) inputs/outputs:

- **premult**: For a color3 `in` stream and a float `alpha` stream, multiplies `in` by `alpha`; output is

type color3. To do a 1-channel "premult", use `<multiply type="float">`.

- `in` (input, color3, required): the name of the node to connect to the input
- `alpha` (input, float, required): the name of the float-type alpha input node

- **unpremult**: For a color3 `in` stream and a float `alpha` stream, divides `in` by `alpha`; output is type color3. To do a 1-channel "unpremult", use `<divide type="float">`.

- `in` (input, color3, required): the name of the node to connect to the input
- `alpha` (input, float, required): the name of the float-type alpha input node

Blend nodes take two 1-4 channel inputs and apply the same operator to all channels (the math for alpha is the same as for R or RGB). In the Blend Operator table, "F" and "B" refer to any individual channel of the fg and bg inputs respectively. Blend nodes have no parameters.

Blend Operator	Each Channel Output	Supported Types
burn	$1-(1-B)/F$	float, colorN
dodge	$B/(1-F)$	float, colorN
screen	$1-(1-F)(1-B)$	float, colorN
overlay	$2FB$ if $F < 0.5$; $1-(1-F)(1-B)$ if $F \geq 0.5$	float, colorN

Merge nodes take two 2-channel (color2) or two 4-channel (color4) inputs and use the built-in alpha channel(s) to control the compositing of the fg and bg inputs. In the Merge Operator table, "F" and "B" refer to the non-alpha channels of the fg and bg inputs respectively, and "f" and "b" refer to the alpha channels of the fg and bg inputs. Merge nodes are not defined for 1-channel or 3-channel inputs, and cannot be used on vectorN streams. Merge nodes have no parameters.

Merge Operator	RGB output	Alpha Output
disjointover	$F+B$ if $f+b \leq 1$; $F+B(1-f)/b$ if $f+b > 1$	$\min(f+b, 1)$
in	Fb	fb
mask	Bf	bf
matte	$Ff+B(1-f)$	$f+b(1-f)$
out	$F(1-b)$	$f(1-b)$
over	$F+B(1-f)$	$f+b(1-f)$

Masking nodes take one 1-4 channel input "in" plus a separate float "mask" input and apply the same operator to all channels (if present, the math for alpha is the same as for R or RGB). In the Masking Operator table, "F" refers to any individual channel of the "in" input. Masking nodes have no parameters.

Masking Operator	Each Channel Output
inside	Fm
outside	$F(1-m)$

Note: for color3 types, *inside* is equivalent to the `premult` node, and for all types, *inside* is equivalent to the `multiply` node: both operators exist to provide companion functions for other data types or their respective inverse or complementary operations.

The Mix node takes two 1-4 channel inputs "fg" and "bg" plus a separate optional 1-channel "mask" input and mixes the fg and bg according to the mask. The equation for "mix" is as follows, with "F" and "B" referring to any channel of the fg and bg inputs respectively (which can be float, color N or vector N but must match), and "m" referring to the float mask input value:

Mix Operator	Each Channel Output
mix	$Fm+B(1-m)$

The "mix" operator has no parameters. See also the **Shader Nodes** section below for additional `mix` operator variants supporting shader-semantic types.

Conditional Nodes

Conditional nodes are used to compare values of two streams, or select the value from one of several streams.

- **compare**: test the value of an incoming float selector stream against a specified cutoff value, then pass the value of one of two other incoming streams depending on whether the selector stream value is greater than the fixed cutoff value. Compare nodes can be of output type float, color N or vector N , and have three spatially-varying inputs, *intest*, *in1* and *in2*, and one uniform parameter *cutoff*; *cutoff* and the output of the *intest* node must be float type, while the output type of the *in1* and *in2* nodes must match the compare node's output type.
 - *intest* (input, float, required): the name of the node whose output (which must be of type float) is compared to *cutoff*.
 - *cutoff* (parameter, float, required): a fixed value to compare against the value of the *intest* node. Default is 0.0.
 - *in1* (input, float or color N or vector N , required): the name of the node specifying the value to use if the output value of *intest* \leq *cutoff*.
 - *in2* (input, float or color N or vector N , required): the name of the node specifying the value to use if the output value of *intest* $>$ *cutoff*.

- **switch**: pass on the value of one of five input streams, according to the value of a selector parameter *which*. Switch nodes can be of output type float, color N or vector N , and have five inputs, *in1* through *in5* (not all of which must be connected), which must match the output type.
 - *in1*, *in2*, *in3*, *in4*, *in5* (input, float or color N or vector N , optional): the names of the nodes to select values from based on the value of the *which* parameter. The types of the

- various `inN` inputs must match the type of the `switch` node itself.
- `which` (parameter, float, required): a selector to choose which input to take values from; the output comes from input "`floor(which)+1`", clamped to the 1-5 range. So `which<1` will pass on the value from `in1`, `1<=which<2` will pass the value from `in2`, `2<=which<3` will pass the value from `in3`, `3<=which<4` will pass the value from `in4`, and `4<=which` will pass the value from `in5`. If the input that `which` selects is not connected, the `<switch>` node will output 0.0 in all channels.

Channel Nodes

Channel nodes are used to perform channel manipulations on `float`, `colorN`, and `vectorN` streams, allowing the order and number of channels to be modified, and the data types of streams to be altered.

- **swizzle**: perform an arbitrary permutation of the channels of the input stream, returning a new stream of the specified type. Individual channels may be replicated or omitted, and the output stream may have a different number of channels than the input.
 - `in` (input, float or `colorN` or `vectorN`, required): the name of the node to connect to the input
 - `channels` (parameter, string, required): a string of one, two, three or four characters (one per channel in the output), each of which may be "r", "g", "b", "a", "0" or "1" for `colorN` inputs, or "x", "y", "z", "w", "0" or "1" for `vectorN` inputs. E.g. "bgra" would output a four-channel stream with the red and blue channels swapped, "rg01" would output a four-channel stream with red and green passed through and blue set to 0 and alpha set to 1, and "zzz" would output a three-channel stream with the Z component replicated to all channels. The number of characters in `channels` must be the same as the number of channels for the `swizzle` node's output type, e.g. exactly 2 characters for a `swizzle` of type "vector2", or 4 characters for a `swizzle` of type "color4". If the input's type is "float", then either "r" or "x" may be used interchangeably to represent the one incoming data channel.
- **pack**: pack the channels from two, three or four streams into the same number of channels of a single stream of a specified compatible type; please see the table below for a list of all supported combinations of input and output types. For color output types, no colorspace conversion will take place; the channels are simply copied as-is.
 - `in1` (input, float/`color2`/`color3`/`vector2`/`vector3`, required): the name of the float-type node whose output will be sent to the first channel of the output
 - `in2` (input, float/`color2`/`vector2`, required): the name of the float-type node whose output will be sent to the second channel of the output
 - `in3` (input, float, optional): for 3- or 4-channel output types, the name of the float-type node whose output will be sent to the third channel of the output
 - `in4` (input, float, optional): for 4-channel output types, the name of the float-type node whose output will be sent to the fourth channel of the output

Table of allowable input/output types for **pack**:

type (output)	in1	in2	in3	in4	Output Value
color2	float "r"	float "g"	n/a	n/a	"rg"
vector2	float "x"	float "y"	n/a	n/a	"xy"
color3	float "r"	float "g"	float "b"	n/a	"rgb"
vector3	float "x"	float "y"	float "z"	n/a	"xyz"
color4	float "r"	float "g"	float "b"	float "a"	"rgba"
vector4	float "x"	float "y"	float "z"	float "w"	"xyzw"
color4	color3 "rgb"	float "a"	n/a	n/a	"rgba"
vector4	vector3 "xyz"	float "w"	n/a	n/a	"xyzw"
color4	color2 "rg"	color2 "ba"	n/a	n/a	"rgba"
vector4	vector2 "xy"	vector2 "zw"	n/a	n/a	"xyzw"

Convolution Nodes

Convolution nodes have one input named "in", and apply a defined convolution function on the input stream. Note that these nodes are not generally implementable in raytracing applications; they are provided for the benefit of purely 2D image processing applications. [REQ="convolveops"]

- **blur**: a gaussian-falloff blur.
 - in (input, float or color N or vector N , required): the name of the node to connect to the input
 - size (parameter, float, optional): the size of the gaussian blur kernel, relative to 0-1 UV space

- **heighttonormal**: convert a scalar height map to a normal map of type vector3.
 - in (input, float, required): the name of the node to connect to the input
 - scale (parameter, float, optional): the scale of normal map deflections relative to the gradient of the height map. Default is 1.0.

Organization Nodes

The following nodes provide no data-processing functionality of their own, and are included to support organization and documentation of node graphs in 3D content packages.

- **dot**: a no-op, passes its input through to its output unchanged. Users can use dot nodes to shape edge connection paths or provide documentation checkpoints in node graph layout UI's.

- `in` (input, any type, required): the name of the node to be connected to the Dot node's "in" input.
- `note` (parameter, string, optional): a text note associated with the dot node; default is no text.
- **backdrop**: a layout element used to contain, group, and document other nodes.
 - `note` (parameter, string, optional): a text note associated with the backdrop node; default is no text.
 - `contains` (parameter, stringarray, optional): a comma-separated list of node names that the backdrop "contains"; default is to contain no nodes.
 - `width` (parameter, float, optional): width of the backdrop when drawn in a UI. [See `xpos/ypos` under Standard UI Attributes for a discussion of UI units]
 - `height` (parameter, float, optional): height of the backdrop when drawn in a UI.

For **dot** and **backdrop** nodes, the `note` text can contain standard HTML formatting strings, such as ``, ``, `<p>`, etc. but no complex formatting such as CSS or external references (e.g. no hyperlinks or images).

Standard Node Parameters

All standard nodes which define a `defaultinput` or `default` value support the following parameter:

- `disable` (parameter, boolean, optional): if set to true, the node will pass its default input or value to its output, effectively disabling the node; default is false. Applications may choose to implement the `disable` parameter by skipping over the disabled node during traversal and instead passing through a connection to the `defaultinput` node or outputting the node's default value, rather than using an actual `disable` parameter in the node implementation.

Standard UI Attributes

All `<parameter>` and `<input>` elements of any type also allow specification of the following additional UI-related attributes:

- `helptext` (attribute, string, optional): a description of the function or purpose of this parameter or input; may include standard HTML formatting strings as described in the **Organizational Nodes** section above.

All `<parameter>` and `<input>` elements of type integer, float, `color N` or `vector N` also allow specification of the following additional UI-related attributes:

- `uimin` (attribute, integer or float or `color N` or `vector N` , optional): the minimum value that the UI allows for this particular value. MaterialX itself does not enforce this as an actual minimum for value.
- `uimax` (attribute, integer or float or `color N` or `vector N` , optional): the maximum value that the UI allows for this particular value. MaterialX itself does not enforce this as an actual maximum for value.

All `<parameter>` and `<input>` elements of type string also allow specification of the following additional UI-related attributes:

- `uienum` (attribute, stringarray, optional): a comma-separated list of string values that the parameter value is allowed to take. MaterialX itself does not enforce that a specified parameter value is actually in this list.

All node types (both Sources and Operators) as well as `<shaderref>`, `<material>` and `<look>` elements support the following UI-related attributes:

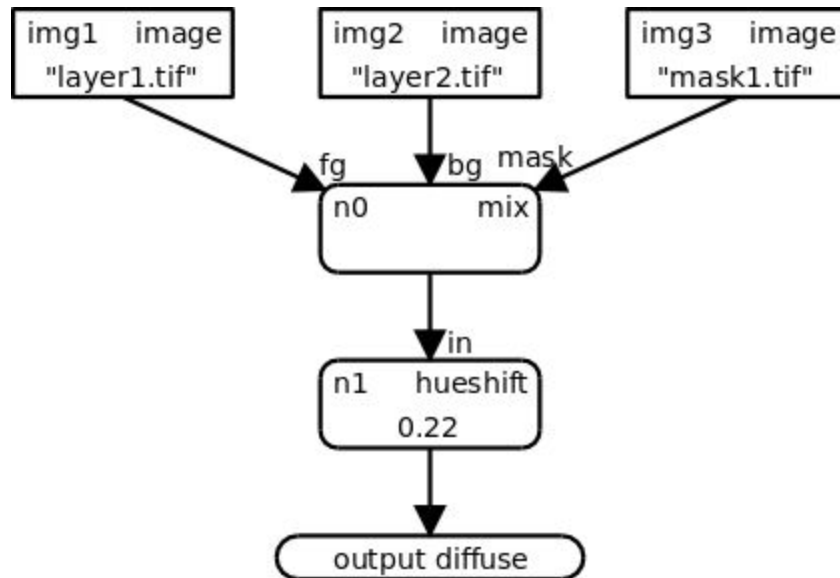
- `xpos` (attribute, float, optional): X-position of the node when drawn in a UI.
- `ypos` (attribute, float, optional): Y-position of the node when drawn in a UI.
- `uicolor` (attribute, vector3, optional): the display-referred color of the node as drawn in the UI, normalized to 0.0-1.0 range; default is to not specify a particular color so the application's default node color would be used. `uicolor` values are expressed as vector3 values rather than color3, and thus are not affected by the current `colorspace`.

The scale of `xpos` (and `ypos`) is such that when drawn in a UI, a node drawn at position (x, y) will "look good" next to nodes drawn at position (x+1, y) and at position (x, y+1): unit scale on this "grid" is sufficient to hold a typical sized node plus any connection edges and arrows. It is not necessary that nodes be placed exactly on integer grid boundaries; this merely states the scale of nodes. It is also not assumed that the pixel scaling factors for X and Y are the same: the actual UI unit "grid" does not have to be square. If `xpos` and `ypos` are not both specified, placement of the node when drawn in a UI is undefined, and it is up to the application to figure out placement (which could mean "all piled up in the center in a tangled mess").

MaterialX defines `xpos` values to be increasing left to right, `ypos` values to be increasing top to bottom, and the general flow is generally downward. E.g. node inputs are on the top and outputs on the bottom, and a node at (10, 10) could connect naturally to a node at (10, 11). Content creation applications using left-to-right flow can simply exchange X and Y coordinates in their internal representations when reading or writing MaterialX data, and applications that internally use Y coordinates increasing upward rather than downward can invert the Y coordinates between MTLX files and their internal representations.

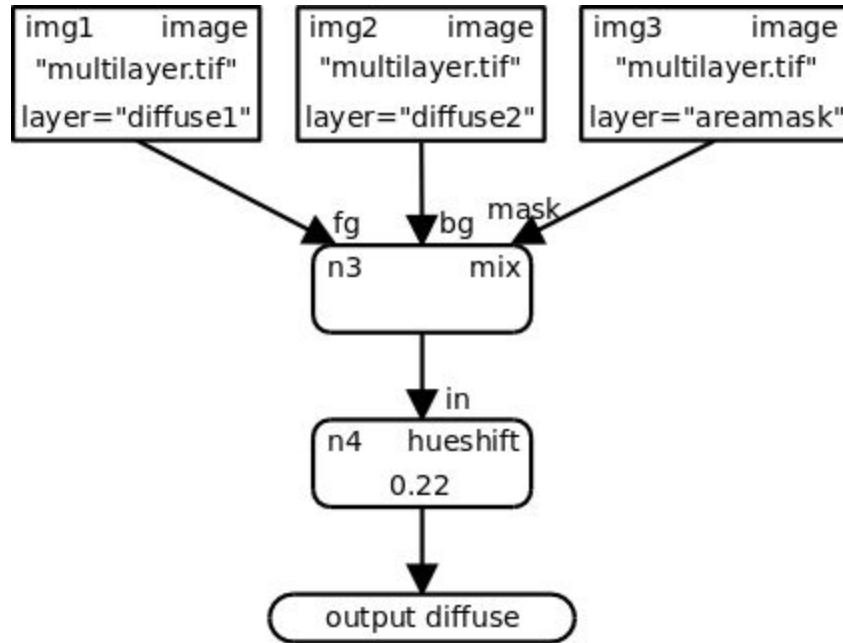
NodeGraph Examples

Example 1: Simple merge of two single-layer images with a separate mask image, followed by a simple color operation.



```
<?xml version="1.0" encoding="UTF-8"?>
<materialx>
  <nodegraph name="nodegraph1">
    <image name="img1" type="color3">
      <parameter name="file" type="filename" value="layer1.tif"/>
    </image>
    <image name="img2" type="color3">
      <parameter name="file" type="filename" value="layer2.tif"/>
    </image>
    <image name="img3" type="float">
      <parameter name="file" type="filename" value="mask1.tif"/>
    </image>
    <mix name="n0" type="color3">
      <input name="fg" type="color3" nodename="img1"/>
      <input name="bg" type="color3" nodename="img2"/>
      <input name="mask" type="float" nodename="img3"/>
    </mix>
    <hueshift name="n1" type="color3">
      <input name="in" type="color3" nodename="n0"/>
      <parameter name="amount" type="float" value="0.22"/>
    </hueshift>
    <output name="diffuse" type="color3" nodename="n1"/>
  </nodegraph>
</materialx>
```

Example 2: Same as above, but replacing the three single-channel input files with a single multi-channel input file.



```

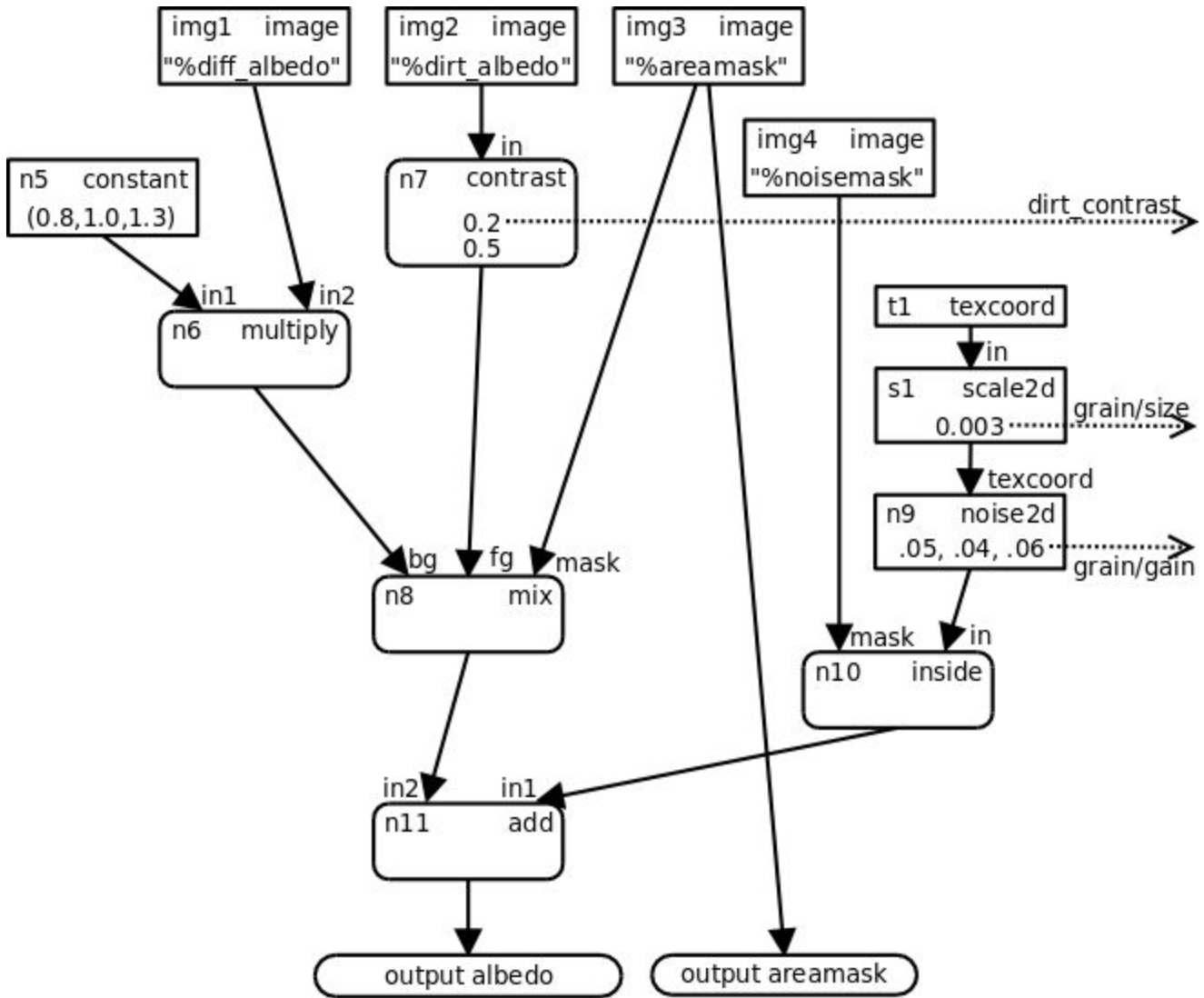
<?xml version="1.0" encoding="UTF-8"?>
<materialx require="multilayer">
  <nodegraph name="nodegraph2">
    <image name="img1" type="color3">
      <parameter name="file" type="filename" value="multilayer.tif"/>
      <parameter name="layer" type="string" value="diffuse1"/>
    </image>
    <image name="img2" type="color3">
      <parameter name="file" type="filename" value="multilayer.tif"/>
      <parameter name="layer" type="string" value="diffuse2"/>
    </image>
    <image name="img3" type="float">
      <parameter name="file" type="filename" value="multilayer.tif"/>
      <parameter name="layer" type="string" value="areamask"/>
    </image>
    <mix name="n3" type="color3">
      <input name="fg" type="color3" nodename="img1"/>
      <input name="bg" type="color3" nodename="img2"/>
      <input name="mask" type="float" nodename="img3"/>
    </mix>
    <hueshift name="n4" type="color3">
      <input name="in" type="color3" nodename="n3"/>
      <parameter name="amount" type="float" value="0.22"/>
    </hueshift>
    <output name="diffuse" type="color3" nodename="n4"/>
  </nodegraph>
</materialx>
  
```

Note: Because the preceding MaterialX file makes use of the non-required capability "multilayer" to read data from multilayer input images, a "require" element was added to the opening <materialx> element to declare that need.

The above file could be embedded within "multilayer.exr"s metadata field by setting the "file" parameter of the input nodes to the special token "\$CONTAINER":

```
<image name="img1" type="color3">
  <parameter name="file" type="filename" value="$CONTAINER"/>
  <parameter name="layer" type="string" value="diffuse1"/>
</image>
<image name="img2" type="color3">
  <parameter name="file" type="filename" value="$CONTAINER"/>
  <parameter name="layer" type="string" value="diffuse2"/>
</image>
<image name="img3" type="float">
  <parameter name="file" type="filename" value="$CONTAINER"/>
  <parameter name="layer" type="string" value="areamask"/>
</image>
```

Example 3: A more complex example, using geometry attributes to define two diffuse albedo colors and two masks, then color-correcting one albedo less red and more blue and increasing the contrast of the other, blending the two through an area mask, and adding a small amount of scaled 2D Perlin noise within a second mask. The contrast amount for the second color map and the size and amplitude for the overall noise have been given publicnames to expose them as externally-overridable public parameters, and the graph outputs the area mask layer separately from the composited diffuse albedo color.



```

<?xml version="1.0" encoding="UTF-8"?>
<materialx require="override">
  <!-- Note: in a real file, there would need to be geominfos here to define
  %diff_albedo etc. for each geometry-->
  <nodegraph name="nodegraph3">
    <image name="img1" type="color3">
      <parameter name="file" type="filename" value="%diff_albedo"/>
    </image>
    <image name="img2" type="color3">
      <parameter name="file" type="filename" value="%dirt_albedo"/>
    </image>
    <image name="img3" type="float">
      <parameter name="file" type="filename" value="%areamask"/>
    </image>
    <image name="img4" type="float">
      <parameter name="file" type="filename" value="%noisemask"/>
    </image>
    <constant name="n5" type="color3">
      <parameter name="value" type="color3" value="0.8,1.0,1.3"/>
    </constant>
  </nodegraph>

```

```

<multiply name="n6" type="color3">
  <input name="in1" type="color3" nodename="n5"/>
  <input name="in2" type="color3" nodename="img1"/>
</multiply>
<contrast name="n7" type="color3">
  <input name="in" type="color3" nodename="img2"/>
  <parameter name="amount" type="float" value="0.2"
    publicname="dirt_contrast"/>
  <parameter name="pivot" type="float" value="0.5"/>
</contrast>
<mix name="n8" type="color3">
  <input name="fg" type="color3" nodename="n7"/>
  <input name="bg" type="color3" nodename="n6"/>
  <input name="mask" type="float" nodename="img3"/>
</mix>
<texcoord name="t1" type="vector2"/>
<scale name="s1" type="vector2">
  <input name="in" type="vector2" nodename="t1"/>
  <parameter name="amount" type="float" value="0.003" publicname="grain/size"/>
</scale>
<noise2d name="n9" type="color3">
  <input name="texcoord" type="vector2" nodename="s1"/>
  <parameter name="amplitude" type="color3" value="0.05,0.04,0.06"
    publicname="grain/gain"/>
</noise2d>
<inside name="n10" type="color3">
  <input name="mask" type="float" nodename="img4"/>
  <input name="in" type="color3" nodename="n9"/>
</inside>
<add name="n11" type="color3">
  <input name="in1" type="color3" nodename="n10"/>
  <input name="in2" type="color3" nodename="n8"/>
</add>
<output name="albedo" type="color3" nodename="n11"/>
<output name="areamask" type="float" nodename="img3"/>
</nodegraph>
</materialx>

```

Custom Nodes

Specific applications will commonly support sources and operators that do not map directly to standard MaterialX nodes. Individual implementations may provide their own custom nodes [REQ="customnode"], with `<nodedef>` elements to declare their parameter interfaces, and `<implementation>` and/or `<nodegraph>` elements to define their behaviors.

Custom Node Declaration

Each custom node must be explicitly declared with a `<nodedef>` element, specifying the expected names and types of the node's inputs and output(s).

Attributes for `<nodedef>` elements:

- `name` (string, required): a unique name for this `<nodedef>`
- `node` (string, required): the name of the custom node being defined
- `type` (string, required): the type of the output of this custom node, which can be a standard MaterialX type or a custom type declared through a `<typedef>`. `<Nodedef>`s for custom nodes with multiple outputs should declare `type` as "multioutput" [REQ="multioutput"].
- `nodecategory` (string, optional): an optional category name for this node. Standard MaterialX nodes have `nodecategory` values matching the titles of the section headings in which they are described, e.g. "texture", "procedural", "global", "geometric", "application", "math", "adjustment", "compositing", "conditional", "channel", "convolution", or "organizational".
- `defaultinput` (string, optional): for nodes with a single output, the name of an `<input>` element within the `<nodedef>`, which must be the same type as `type`, and will be passed through unmodified by applications that don't have an implementation for this node. "multioutput"-type `nodedefs` may not specify a `defaultinput`.
- `default` (same type as `type`, optional): for nodes with a single output, a constant value which will be output by applications that don't have an implementation for this node, or if a `defaultinput` input is specified but that input is not connected. "multioutput"-type `nodedefs` may not specify a `default`.
- `target` (stringarray, optional): the set of targets to which this `nodedef` is restricted. By default, a `nodedef` is considered universal, not restricted to any specific targets, but it is possible that certain targets may have different parameter names or usage for the same node.
- `require` (string, optional): If a custom node's functionality requires access to non-local or geometric information, it should declare this compatibility requirement as discussed in the **Implementation Compatibility Checking** section above by providing a `require` attribute value: "convolveops" for nodes requiring access to nearby "texture" information; "globalops" for nodes requiring access to non-local geometric features, and "geomops" for nodes requiring access to local geometric features such as surface position, normal, and tangent

Custom nodes are allowed to overload a single `node` name by providing multiple `<nodedef>` elements with different combinations of input and output types. This overloading is permitted both for custom `node` names and for the standard MaterialX node set. Within the scope of a single MaterialX document and its included content, no two `<nodedef>` elements with an identical combination of input and output types may be provided for a single `node` name.

NodeDefs with multiple outputs must additionally define the appropriate number of child `<output>`

elements within the `<nodedef>` to define the name and types of each output [REQ="multioutput"]; for nodes defined using a `nodegraph`, the names and types of the outputs must agree with the `<output>` elements in the `nodegraph`. Single-output `<nodedef>`s cannot contain an `<output>` element, as the (nameless) output's type and default is set in the `<nodedef>` itself.

The parameter interface of a custom node is specified as a set of child `<parameter>` and `<input>` elements of the `<nodedef>`.

Parameter elements are used within a `<nodedef>` to declare the uniform parameters of a node:

```
<parameter name="parametername" type="parametertype" [value="value"]
  [publicname="publicname"]/>
```

Attributes for Parameter elements:

- `name` (string, required): the name of the parameter
- `type` (string, required): the MaterialX type of the parameter
- `value` (same type as `type`, optional): a default value for this parameter, to be used if the node is invoked without a value or connection defined for this parameter. If a default value is not defined, then the parameter becomes required, so any invocation of the custom node without a value assigned to that parameter would be in error.

Input elements are used within a `<nodedef>` to declare the spatially-varying inputs for a node:

```
<input name="inputname" type="inputtype" [value="value"]
  [publicname="publicname"]/>
```

Attributes for Input elements:

- `name` (string, required): the name of the shader input
- `type` (string, required): the MaterialX type of the shader input
- `value` (same type as `type`, optional): a default value for this input, to be used if the input remains unconnected and is not otherwise assigned a value
- `defaultgeomprop` (string, optional): for vector2 or vector3 inputs, the name of an intrinsic geometric property that provides the default value for this input, must be one of "position", "normal", "tangent", "bitangent" or "texcoord" for vector3 inputs, or "texcoord" for vector2 inputs. This is effectively the same as declaring a default connection of the input to a Geometric Node with default parameters.

It is permissible to define a `value` or a `defaultgeomprop` for an input but not both. If neither `value` or `defaultgeomprop` are defined, then the input becomes required, and any invocation of the custom node without providing a value or connection for this input would be in error.

Custom Node Definition

Once the parameter interface of a custom node has been declared through a `<nodedef>`, MaterialX provides two methods for precisely defining its functionality: via an `<implementation>` element that references source code, or via a `<nodegraph>` element that composes the required functionality from existing nodes. Providing a definition for a custom node is optional in MaterialX, but it's recommended for maximum clarity and portability.

Implementation elements are used to associate external function source code with a specific `nodedef`. Implementation elements support the following attributes:

- `name` (string, required): a unique name for this `<implementation>`
- `nodedef` (string, required): the name of the `<nodedef>` for which this `<implementation>` applies
- `file` (filename, optional): the URI of an external file containing the source code for the entry point of this particular node template. This file may contain source code for other templates of the same custom node, and/or for other custom nodes. Ideally, source code for nodes should be written in a portable language such as OSL, MDL or HLSL, but any language supported by the target system (if specified) is acceptable.
- `function` (string, optional): the name of a function within the given external file that contains the implementation of this node. If a `file` is specified, then `function` is required.
- `language` (string, optional): when `file` is specified, the language in which the `file` code is written; defaults to "osl".
- `target` (stringarray, optional): the set of targets to which this implementation is restricted. By default, an implementation is considered applicable to all targets that the referenced `nodedef` applies to. If the referenced `<nodedef>` also specifies a target, then this `target` must be a proper subset of the `nodedef`'s target list.

If an `<implementation>` element specifies a `language` and/or `target` with no `file`, then it is interpreted purely as documentation that a private definition exists for the given target. Because the definition in an `<implementation>` may be restricted to specific targets, a `<nodedef>` that is defined with such restrictions may not be available in all applications; for this reason, a `<nodedef>` that is defined through an `<implementation>` is expected to provide a value for `default` and/or `defaultinput` when possible, specifying the expected behavior when no definition for the given node can be found. It should be noted that specifying a `language` and/or a `target` does not necessarily guarantee compatibility: these are intended to be hints about the particular implementation, and it is up to the host application to determine which `<implementation>`, if any, is appropriate for any particular use.

Because the names used for node inputs or parameters (such as "normal" or "default") may conflict with reserved words in various shading languages, MaterialX allows `<implementation>` elements to contain a number of `<input>` and/or `<parameter>` elements to remap the names of `<input>`s and `<parameter>`s as specified in the `<nodedef>` to different `implnames` to indicate what the input or parameter name is actually called in the implementation's code. Only the inputs and parameters that need to be remapped to new `implnames` need to be listed; for each, it is recommended that the `type` of that input or parameter be listed for clarity, but if specified, it must match the type specified in the `<nodedef>`: `<implementation>`s are not allowed to change the type or any other attribute defined in the `<nodedef>`. In this example, the `<implementation>` declares that the "default" parameter defined in the "ND_image_color" `nodedef` is actually called "default_value" in the "mx_image_color" function:

```
<implementation name="IM_image_color3_osl" nodedef="ND_image_color"
    file="mx_image_color.osl" function="mx_image_color" language="osl">
  <parameter name="default" type="color3" implname="default_value"/>
</implementation>
```

Alternatively, a `<nodegraph>` element may specify a `nodedef` attribute (and optionally a `target` attribute as well): this indicates both that the `nodegraph` is a functional definition for that `<nodedef>`, and

that the <nodedef> declares the set of inputs and parameters that the nodegraph accepts. The type of the <nodedef> (or the types of its <output>s for "multioutput" nodedefs) and the type(s) of the nodegraph <output>(s) must agree. The inputs and parameters of the <nodedef> can then be referenced within <input> and <parameter> elements of nodes within the nodegraph implementation using interfacename attributes in place of value or nodename attributes, e.g. a nodedef parameter named "p1" and a nodedef input "i2" can be referenced as follows:

```
<parameter name="amount" type="float" interfacename="p1"/>
<input name="in1" type="color3" interfacename="i1"/>
<input name="in2" type="color3" interfacename="i2"/>
```

It is permissible to define multiple nodegraph- and/or file-based implementations for a custom node for the same combination of input and output types. It is recommended that the specified language/target combinations be unique, e.g. one implementation in "osl" and another in "glsl", although this is not required: in the case of multiple applicable implementations for a target, it would be up to the host application to determine which implementation to actually use.

Example 1: custom nodes defined with external file implementations:

```
<nodedef name="mblendcolor3" node="mariBlend" type="color3" defaultinput="in1">
  <input name="in1" type="color3" value="0.0, 0.0, 0.0"/>
  <input name="in2" type="color3" value="1.0, 1.0, 1.0"/>
  <parameter name="ColorA" type="color3" value="0.0, 0.0, 0.0"/>
  <parameter name="ColorB" type="color3" value="0.0, 0.0, 0.0"/>
</nodedef>
<nodedef name="mblendfloat" node="mariBlend" type="float" defaultinput="in1">
  <input name="in1" type="float" value="0.0"/>
  <input name="in2" type="float" value="1.0"/>
  <parameter name="ColorA" type="float" value="0.0"/>
  <parameter name="ColorB" type="float" value="0.0"/>
</nodedef>
<nodedef name="mnoisecolor3" node="mariCustomNoise" type="color3"
  default="0.5,0.5,0.5">
  <parameter name="ColorA" type="color3" value="0.5, 0.5, 0.5"/>
  <parameter name="Size" type="float" value="1.0"/>
</nodedef>
<implementation name="glsl_mblendc3" nodedef="mblendcolor3"
  file="lib/mtlx_funcs.glsl" language="glsl"/>
<implementation name="glsl_mblendf" nodedef="mblendfloat"
  file="lib/mtlx_funcs.glsl" language="glsl"/>
<implementation name="glsl_mnoisec3" nodedef="mnoisecolor3"
  file="lib/mtlx_funcs.glsl" language="glsl"/>
<implementation name="osl_mblendc3" nodedef="mblendcolor3"
  file="lib/mtlx_funcs.osl" language="osl"/>
<implementation name="osl_mblendf" nodedef="mblendfloat"
  file="lib/mtlx_funcs.osl" language="osl"/>
<implementation name="osl_mnoisec3" nodedef="mnoisecolor3"
  file="lib/mtlx_funcs.osl" language="osl"/>
<implementation name="oslvray_mnoisec3" nodedef="mnoisecolor3"
  file="lib/mtlx_vray_funcs.osl" language="osl" target="vray"/>
```

The above example defines two templates for a custom operator node called "mariBlend" (one operating on color3 values, and one operating on floats), and one template for a custom source node called

"mariCustomNoise". Implementations of these functions have been defined in both OSL and GLSL. There is also in this example an alternate implementation of the "mariCustomNoise" function specifically for V-Ray, as if the author had determined that the generic OSL version was not appropriate for that renderer.

Here is an example of a multioutput node definition and external implementation declaration. Note the use of <output> elements within the <nodedef> to describe the names, types, and defaults for each output.

```
<nodedef name="dblclr3" node="doublecolor" type="multioutput">
  <input name="in1" type="color3" value="0.0, 0.0, 0.0"/>
  <parameter name="seed" type="float" value="1.0"/>
  <output name="c1" type="color3" default="1.0, 1.0, 1.0"/>
  <output name="c2" type="color3" defaultinput="in1"/>
</nodedef>
<implementation name="osl_dc3" nodedef="dblclr3" file="lib/mtlx_funcs.osl"
  language="osl"/>
```

Example 2: a custom node defined using a nodegraph. The parameters of the nodegraph are declared by the <nodedef>, and the nodes within the nodegraph reference those parameters using `interfacename` attributes. The "fg" input parameter provides a default value which is used if the "fg" input is left unconnected when the custom node is used, and the "amount" parameter defines a default value which will be used if invocations of the node do not explicitly provide a value for "amount". The "bg" input does not provide a default, so it would be an error to invoke this node without connecting "bg".

```
<nodedef name="bladdc4" node="blend_add" type="color4" defaultinput="bg">
  <input name="fg" type="color4" value="0,0,0,0"/>
  <input name="bg" type="color4"/>
  <parameter name="amount" type="float" value="1.0"/>
</nodedef>
<nodegraph name="blend_add_ng" nodedef="bladdc4">
  <multiply name="n1" type="color4">
    <input name="in1" type="color4" interfacename="fg"/>
    <input name="in2" type="float" interfacename="amount"/>
  </multiply>
  <add name="n2" type="color4">
    <input name="in1" type="color4" nodename="n1"/>
    <input name="in2" type="color4" interfacename="bg"/>
  </add>
  <output name="o_result" type="color4" nodename="n2"/>
</nodegraph>
```

Custom Node Use

Once defined with a <nodedef>, invoking a custom node within a nodegraph looks very much the same as using any other standard node: the name of the element is the name of the custom node, and the MaterialX type of the node's output is required; the custom node's child elements define connections of inputs to other node outputs as well as any parameter values for the custom node.

```
<mariCustomNoise name="custnoise1" type="color3">
  <parameter name="ColorA" type="color3" value="1.0, 1.0, 1.0"/>
```

```

    <parameter name="Size" type="float" value="0.5"/>
</mariCustomNoise>
<mariBlend name="customblend1" type="color3">
    <input name="in1" type="color3" nodename="custnoise1"/>
    <input name="in2" type="color3" value="0.3, 0.4, 0.66"/>
    <parameter name="ColorA" type="color3" value="1.0, 1.0, 0.9"/>
    <parameter name="ColorB" type="color3" value="0.2, 0.4, 0.6"/>
</mariBlend>

```

In this example, some inputs of nodes n2 and n4 have been connected to the two named outputs of the custom doublecolor operator "dc1": an `output` attribute is used to specify which output of "dc1" to connect to in each case.

```

<doublecolor name="dc1" type="multioutput">
    <input name="in1" type="color3" nodename="n0"/>
    <parameter name="seed" type="float" value="0.442367"/>
</doublecolor>
<contrast name="n2" type="color3">
    <input name="in" type="color3" nodename="dc1" output="c1"/>
    <parameter name="amount" type="float" value="0.14"/>
</contrast>
<add name="n4" type="color3">
    <input name="in1" type="color3" nodename="dc1" output="c2"/>
    <input name="in2" type="color3" nodename="n1"/>
</add>

```

If a custom node's implementation uses a nodegraph, and that nodegraph contains parameters and/or inputs that define publicnames, then the invocation of that node may define a publicnameprefix, which will be prepended to all publicnames. This allows a custom node to be used more like a macro than a function. For example, if a custom node called "stdadjustments" internally defines publicnames "Contrast" and "RGBmult", then this invocation of that node:

```

<stdadjustments name="adj1" type="color3" publicnameprefix="Specular/spec">

```

will result in those internal publicnames resolving to "Specular/specContrast" and "Specular/specRGBmult".

Shader Nodes

Custom nodes that output data types with a "shader" semantic are referred to in MaterialX as "Shader Nodes". Shaders, along with their inputs and parameters, are declared using the same `<nodedef>`, `<implementation>` and `<nodegraph>` elements described above:

```

<nodedef name="name" type="shadertype" node="shaderprogramname">
    ...parameter and input declarations...
</nodedef>

```

The attributes for `<nodedef>` elements as they pertain to the declaration of shaders are:

- `name` (string, required): a user-chosen name for this shader node definition element.
- `type` (string, optional): the "data type" of the output for this shader, which must have been

defined with a "shader" semantic; see the **Custom Data Types** section above and discussion below for details.

- `node` (string, required): the name of the shader node being defined, which typically matches the name of an associated shader program such as "blinn_phong", "disney_principled_2012", "volumecloud_vol". Just as for custom nodes, this shading program may be defined precisely through an `<implementation>` or `<nodegraph>`, or left to the application to locate by name using any shader definition method that it chooses.

NodeDef elements defining shader nodes do not typically include `default` or `defaultinput` attributes, though they are permitted using the syntax described in the **Custom Data Types** section if the output type of the shader node is not a blind data type.

As mentioned in the **Custom Data Types** section earlier, the standard MaterialX distribution includes the following standard data types for shaders:

```
<typedef name="surfaceshader" semantic="shader" context="surface"/>
<typedef name="volumeshader" semantic="shader" context="volume"/>
<typedef name="displacementshader" semantic="shader" context="displacement"/>
<typedef name="lightshader" semantic="shader" context="light"/>
```

These types all declare that they have "shader" semantic, but define different contexts in which a rendering target should interpret the output of the shader node. For a shading language based on deferred lighting computations (e.g. OSL), a shader-semantic data type is equivalent to a radiance closure. For a shading language based on in-line lighting computations (e.g. GLSL), a shader-semantic data type is equivalent to the final output values of the shader.

It is allowable for applications to define additional types for shader nodes; in particular, one could define a custom type with explicitly-defined members to represent the output AOVs for a class of shader nodes:

```
<typedef name="studio_aovs" semantic="shader" context="surface">
  <member name="rgba" type="color3"/>
  <member name="diffuse" type="color3"/>
  <member name="specular" type="color3"/>
  <member name="indirect" type="color3"/>
  <member name="opacity" type="float"/>
  <member name="Pndc" type="vector3"/>
</typedef>
```

and then use this type when declaring surface shader nodes:

```
<nodedef name="ilm_unifieddef" type="studio_aovs" node="unified_srf">
  <input name="diffc" type="color3" value="0.18,0.18,0.18"/>
  <parameter name="specrroughness" type="float" value="0.3"/>
  ...
</nodedef>
```

It should be noted that the primary benefit of declaring and using specific types for shader nodes would be to differentiate which shader nodes' outputs can be connected into other nodes' inputs (e.g. the types match) for applications such as post-shading layering and blending operations. It should also be noted that using non-blind data types for shaders with specific members greatly limits portability of graphs to

other systems, so their use should be restricted to situations which require them; MaterialX materials and looks do not require knowledge of the exact contents of shader output and use of the standard "surfaceshader" etc. types should be sufficient and is encouraged.

Declarations of shader node source implementations are also accomplished using `<implementation>` elements for external source file declarations and `nodedef` attributes within `<nodegraph>` elements for nodegraph-based definitions [REQ="shadergraphdef" for nodegraph-based shader node implementations].

As with non-shader custom nodes, **Parameter** elements are used within a `<nodedef>` to declare the uniform parameters of a shader node, and **Input** elements are used within a `<nodedef>` to declare the spatially-varying input ports for a shader node. When a shader node is instantiated in a `<material>`, its parameters may be bound to new uniform values or left at their declared default values, and its input ports may be connected to the spatially-varying output ports of nodegraphs [REQ="matnodegraph"] or bound to new (uniform) values, or left at their declared default (uniform) values.

Standard Shader-Semantic Operator Nodes

The Standard MaterialX Library defines the following node variants operating on "shader"-semantic types.

- **add**: add two surfaceshader closures.
 - `in1` (input, surfaceshader, required): the name of the first shader-semantic node
 - `in2` (input, surfaceshader, required): the name of the first shader-semantic node
- **multiply**: multiply a surfaceshader closure by a float or color3 value.
 - `in1` (input, surfaceshader, required): the name of the input shader-semantic node
 - `in2` (input, float or color3, required): the value to multiply the closure by
- **mix**: add a value to the incoming float/color/vector. See also the **Shader Nodes** section below for additional `add` variants supporting shader-semantic types.
 - `bg` (input, surfaceshader, required): the name of the background shader-semantic node
 - `fg` (input, surfaceshader, required): the name of the foreground shader-semantic node
 - `amount` (input, float, required): the blending factor used to mix the two input closures

Materials

Material Elements

Material elements are used to define instantiations of one or more of shader nodes of different types, and associate specific parameter values, connection bindings from nodegraph outputs to shader node inputs, and public parameter override values with them.

A **<material>** element contains one or more **<shaderref>** elements and optionally a **<materialinherit>** element, which define what shaders a material references and what material (if any) it inherits from, respectively. Material elements can also declare override values for public parameters in referenced shader or pattern nodes connected to inputs of a referenced shader.

```
<material name="materialname">
  ...optional <materialinherit> element...
  ...optional <materialvar> elements...
  ...<shaderref> elements...
  ...optional <override> declarations...
</material>
```

Attributes for **<material>** elements:

- name (string, required): the name of the material.

<material> elements also support other attributes such as `xpos`, `ypos` and `uicolor` as described in the Standard UI Attributes section above.

MaterialInherit Elements

Materials can inherit the shader references, bindings and overrides from another material by including a **<materialinherit>** element. The material can then specify additional shaders, bindings and/or overrides that will be applied on top of or in place of whatever came from the source material. For maximum compatibility, it is recommended that materials that inherit from other materials only include bindings and parameter overrides and not add or change shaders.

```
<material name="materialname">
  <materialinherit name="miname" material="materialtoinheritfrom">
  ...
</material>
```

Attributes for **<materialinherit>** elements:

- name (string, required): the unique name of this materialinherit element.
- material (string, required): the name of the material element to inherit from.

MaterialVar Elements

MaterialVar elements are used within a **<material>**, or more commonly a **<look>**, to define the value of a typed variable that can be substituted into the filename of **<image>** nodes (via an `"@materialvarname"` image filename substitution string), and/or the values of **<override>**s, **<bindparam>**s, and unconnected

<bindinput>s within any pattern or shader node referenced by the material (via a `materialvar` attribute) [REQ="matvar"]. Materialvars work like look-specific overrides and are useful for creating variations, e.g. by defining alternate colors for shaders or different texture tokens for image file reads.

```
<materialvar name="glowcolor" type="color3" value="0.4, 0.1, 0.05"/>
<materialvar name="damageeffect" type="string" value="damagelevel2"/>
```

If defined in a <material>, a materialvar only affects nodes and shaders referenced hierarchically by that material; a material that inherits from a material defining a materialvar would inherit the materialvar definition(s), just like it inherits overrides, bindparams, and shaderrefs. If defined in a <look>, the materialvar would affect nodes and shader parameters in all materials referenced by the look. If the same materialvar is defined in both a <material> and a <look> referencing that <material>, the definition in the <look> "wins".

Please see the **BindParam Elements** section below for an example of referencing a materialvar in a material parameter binding.

MaterialVarDefault elements define the default value for a specified materialvar name; this default value will be used in a filename string substitution if an explicit materialvar value is not defined within the referencing scope [REQ="matvar"]. MaterialVarDefault is a top-level element, to be used *outside* of a <material> or <look> element, so that explicit <Materialvar> settings can be used within <material>s or <look>s only as needed.

```
<materialvardefault name="damageeffect" type="string" value="nodamage"/>
```

ShaderRef Elements

ShaderRef elements instantiate previously-declared nodes with "shader" semantic within the context of a material, allowing their parameters and inputs to be bound to values and data streams. Any number of <shaderref> elements may be specified within a <material>, as long as no two refer to shader nodes with the same combination of output type `context` and implementation `target`. For example, one could use two <shaderref>s to instantiate both a "surface"-context and a "displacement"-context shader for a material, or different "surface"-context shaders for different renderers within a single material.

Attributes for <shaderref> elements:

- `name` (string, required): the unique name for this shaderref
- `node` (string, optional): the shader-semantic `node` name to reference in the material.
- `nodedef` (string, optional): the name of a <nodedef> defining a specific shader-semantic node.

Either `node` or `nodedef` must be specified, but not both. It is preferred to reference the `nodedef`'s `node` attribute value, but if multiple <nodedef>s for the same node exist it is acceptable to reference the name of the <nodedef> to disambiguate exactly which definition is to be used. So if a shader node was defined using this <nodedef>:

```
<nodedef name="dprin_shaderdef" type="surfaceshader" node="disney_principled">
```

it would normally be referenced within a <shaderref> like this:

```
<shaderref name="sref1" node="disney_principled">
```

but could alternatively be referenced like this:

```
<shaderref name="sref2" nodedef="dprin_shaderdef">
```

If the `<shaderref>` is within a `<material>` element that inherits from another material, and its `node` or `nodedef` refers to the same `<nodedef>` element as the `<shaderref>` in a parent material, then the `<bindparam>`s and `<bindinput>`s within the `shaderref` will apply to the same shader-semantic node as the parent, overlaying its bindings on top of those specified by the parent; it is not possible to create a separate instantiation of the same shader-semantic node within a child material.

`<shaderref>` elements also support other attributes such as `xpos`, `ypos` and `uicolor` as described in the Standard UI Attributes section above.

BindParam Elements

BindParam elements are used within `<shaderref>` elements to dynamically bind values to shader node parameters, replacing any default assignments in the original `<parameter>` elements. These bindings persist only within the scope of the enclosing `<shaderref>` element. BindParams can only be applied to shader nodes referenced by a `<shaderref>` of this material; they cannot apply to a shader node contained within a nodegraph.

```
<material name="steel">
  <shaderref name="sref3" node="simplerf">
    <bindparam name="emissionColor" type="color3" value="0.005, 0.005, 0.005" />
    <bindparam name="rfrIndex" type="float" value="1.33" />
    <bindparam name="diffColor" type="color3" materialvar="steelcolor" />
  </shaderref>
</material>
```

Attributes for Bindparam elements:

- `name` (string, required): the name of the shader `<parameter>` which will be bound to a new value
- `type` (string, required): the MaterialX type of the shader `<parameter>`
- `value` (specified MaterialX type, optional): a value to bind to the shader parameter within the scope of this material.
- `materialvar` (string, optional): the name of a materialvar with a matching type [REQ="matvar"].

Either a `value` or a `materialvar` must be defined, but not both.

BindInput Elements

BindInput elements are used within `<shaderref>` elements to dynamically bind values or nodegraph outputs to shader node inputs, replacing any default assignments in the original `<input>` elements. These bindings persist only within the scope of the enclosing `<shaderref>` element. BindInputs can only be applied to shader nodes referenced by a `<shaderref>` of this material; they cannot apply to a shader node contained within a nodegraph.

```
<material name="steel">
  <shaderref name="sref4" node="simplerf">
    <bindinput name="diffColor" type="color3" nodegraph="DiffNoiseNetwork">
```

```

        output="o_diffColor" />
    <bindinput name="specColor" type="color3" nodegraph="SpecMapNetwork"
        output="o_specColor" />
    <bindinput name="roughness" type="float" value="0.02" />
</shaderref>
</material>

```

Attributes for BindInput elements:

- **name** (string, required): the name of the shader <input> which will be bound to a new value or nodegraph output
- **type** (string, required): the MaterialX type of the shader <input>
- **value** (specified MaterialX type, optional): a uniform value to bind to the shader input within the scope of this material.
- **materialvar** (string, optional): the name of a materialvar with a matching type [REQ="matvar"].
- **nodegraph** (string, optional): the name of the nodegraph element whose output will be bound [REQ="matnodegraph"]
- **output** (string, optional): the name of the nodegraph output that will be bound to this input
- **publicnameprefix** (string, optional): if specified, this string will be prepended to any publicnames defined within the bound nodegraph. See below for details.

Either **value**, **materialvar**, or both **nodegraph** and **output** must be declared, but not any other combination.

Note: if an application does not support lazy binding of inputs to texture processing networks in a material, it is recommended that the application create one parent <material> containing just the nodegraph/input bindings to express these (fixed) connections, then have the material(s) that set parameter values inherit from this parent material.

If a bindinput connects to a nodegraph, and that nodegraph contains parameters and/or inputs that define publicnames, then the bindinput may define a publicnameprefix, which will be prepended to all publicnames. For example, if a nodegraph internally defines publicnames "diffColor" and "specRoughness", then a bindinput with publicnameprefix="Layer1/" will result in those internal publicnames resolving to "Layer1/diffColor" and "Layer1/specRoughness".

Override Elements

Override elements are used within <material> elements to modify public parameter values within the nodegraphs and shaders they reference. These overrides persist only within the scope of the enclosing material element. [REQ="override"]

```

<override name="ov1" publicname="grain/gain" type="float" value="0.075"/>
<override name="ov2" publicname="dirtmix" type="float"
    materialvar="dirtmixval"/>

```

Attributes for Override elements:

- **name** (string, required): the unique name for this override element
- **publicname** (string, required): the publicname of the public parameter to which this override applies

- `type` (string, required): the MaterialX type of the public parameter that is being overridden
- `value` (specified MaterialX type, required): a value to assign to the public parameter within the scope of this material.
- `materialvar` (string, optional): the name of a materialvar with a matching type [REQ="matvar"].

Either a `value` or a `materialvar` must be defined, but not both.

Note that overrides may modify *any* public parameter referenced within the current MaterialX document, so they denote their target by its `publicname`, rather than by its `name`. This allows a referencing `<override>` to remain agnostic of where the public parameter is implemented, and allows an `<override>` to remain valid as underlying implementation details are modified.

If a single shader parameter is the target of both a `<bindparam>` and an `<override>`, then the `<override>` takes precedence.

Material Examples

Example 1: Define two shaders and two materials with different values assigned to the shader(s). The materials bind values to parameters and inputs for shaders, as well as define override values for public parameters exposed in the external nodegraph. The first material references both a surface and a displacement shader and connects one input ("diff_albedo") to the externally-defined nodegraph, while the second references only a surface shader and leaves the `diff_albedo` input unconnected to use its constant default value.

```
<?xml version="1.0" encoding="UTF-8"?>
<materialx require="matnodegraph,override">
  <!-- Assume "nodegraph3.mtlx" defines "nodegraph3", containing nodes defining -->
  <!-- publicnames "diffcolor", "dirt_contrast", "grain/gain" and "grain/size". -->
  <xi:include href="nodegraph3.mtlx"/>
  <nodedef name="basicsrfdef" type="surfaceshader" node="basic_surface">
    <input name="diff_albedo" type="color3" value="0.18,0.18,0.18"/>
    <input name="spec_color" type="color3" value="1,1,1"/>
    <input name="roughness" type="float" value="0.3"/>
    <parameter name="fresnel_exp" type="float" value="0.2"/>
  </nodedef>

  <nodedef name="noisebumpdef" type="displacementshader" node="noise_bump">
    <parameter name="bump_scale" type="float" value="0.02"/>
    <parameter name="bump_ampl" type="float" value="0.015"/>
  </nodedef>

  <material name="material1">
    <shaderref name="sr1" node="basic_surface">
      <bindinput name="diff_albedo" type="color3" nodegraph="nodegraph3"
        output="albedo"/>
      <bindinput name="spec_color" type="color3" value="1.0, 0.99, 0.95"/>
      <bindinput name="roughness" type="float" value="0.15"/>
    </shaderref>
    <shaderref name="sr2" node="noise_bump">
```

```

    <bindparam name="bump_ampl" type="float" value="0.0125"/>
  </shaderref>
  <override name="ov1" publicname="diffcolor" type="color3" value="0.25, 0.24,
0.16"/>
  <override name="ov2" publicname="dirt_contrast" type="float" value="0.33"/>
  <override name="ov3" publicname="grain/gain" type="float" value="0.1"/>
  <override name="ov4" publicname="grain/size" type="float" value="0.008"/>
</material>

<material name="material2">
  <shaderref name="sr3" node="basic_surface">
    <bindinput name="spec_color" type="color3" value="0.7,0.7,0.7"/>
    <bindinput name="roughness" type="float" value="0.1"/>
    <bindparam name="fresnel_exp" type="float" value="0.3"/>
  </shaderref>
</material>
</materialx>

```

Example 2: A material using pre-shader compositing of colors and textures. The parameter values for three different surface types ("steel", "rust" and "paint") are defined as constant color values (or, in the case of "rust_diffc", a color texture). The parameter values are then blended using mask textures in a nodegraph before being connected into a single surface shader. This example also demonstrates the use of the "target" attribute of a shader implementation element to define multiple renderer-specific shaders of the same type referenced within a single material.

```

<?xml version="1.0" encoding="UTF-8"?>
<materialx require="matnodegraph">
  <nodegraph name="shaderparams">
    <constant name="steel_diffc" type="color3">
      <parameter name="value" type="color3" value="0.0318, 0.0318, 0.0318"/>
    </constant>
    <constant name="steel_specc" type="color3">
      <parameter name="value" type="color3" value="0.476, 0.476, 0.476"/>
    </constant>
    <constant name="steel_roughf" type="float">
      <parameter name="value" type="float" value="0.05"/>
    </constant>
    <image name="rust_diffc" type="color3">
      <parameter name="file" type="filename" value="rust_diffc.tif"/>
    </image>
    <constant name="rust_specc" type="color3">
      <parameter name="value" type="color3" value="0.043, 0.043, 0.043"/>
    </constant>
    <constant name="rust_roughf" type="float">
      <parameter name="value" type="float" value="0.5"/>
    </constant>
    <constant name="paint_diffc" type="color3">
      <parameter name="value" type="color3" value="0.447, 0.447, 0.447"/>
    </constant>
    <constant name="paint_specc" type="color3">
      <parameter name="value" type="color3" value="0.144, 0.144, 0.144"/>
    </constant>
    <constant name="paint_roughf" type="float">

```

```

    <parameter name="value" type="float" value="0.137"/>
</constant>
<image name="mask_rust" type="float">
    <parameter name="file" type="filename" value="mask_rust.tif"/>
</image>
<image name="mask_paint" type="float">
    <parameter name="file" type="filename" value="mask_paint.tif"/>
</image>
<mix name="mix_diff1" type="color3">
    <input name="fg" type="color3" nodename="rust_diffc"/>
    <input name="bg" type="color3" nodename="steel_diffc"/>
    <input name="mask" type="float" nodename="mask_rust"/>
</mix>
<mix name="mix_diff2" type="color3">
    <input name="fg" type="color3" nodename="paint_diffc"/>
    <input name="bg" type="color3" nodename="mix_diff1"/>
    <input name="mask" type="float" nodename="mask_paint"/>
</mix>
<output name="o_diffcolor" type="color3" nodename="mix_diff2"/>
<mix name="mix_spec1" type="color3">
    <input name="fg" type="color3" nodename="rust_specc"/>
    <input name="bg" type="color3" nodename="steel_specc"/>
    <input name="mask" type="float" nodename="mask_rust"/>
</mix>
<mix name="mix_spec2" type="color3">
    <input name="fg" type="color3" nodename="paint_specc"/>
    <input name="bg" type="color3" nodename="mix_spec1"/>
    <input name="mask" type="float" nodename="mask_paint"/>
</mix>
<output name="o_speccolor" type="color3" nodename="mix_spec2"/>
<mix name="mix_rough1" type="float">
    <input name="fg" type="float" nodename="rust_roughf"/>
    <input name="bg" type="float" nodename="steel_roughf"/>
    <input name="mask" type="float" nodename="mask_rust"/>
</mix>
<mix name="mix_rough2" type="float">
    <input name="fg" type="float" nodename="paint_roughf"/>
    <input name="bg" type="float" nodename="mix_rough1"/>
    <input name="mask" type="float" nodename="mask_paint"/>
</mix>
<output name="o_roughness" type="float" nodename="mix_rough2"/>
</nodegraph>

<nodedef name="osl_basicsrfdef" type="surfaceshader" node="basic_surface">
    <input name="albedo" type="color3" value="0.15,0.15,0.15"/>
    <input name="speccolor" type="color3" value="1,1,1"/>
    <input name="roughness" type="float" value="0.3"/>
    <parameter name="fresnel" type="float" value="0.25"/>
</nodedef>
<implementation name="osl_basicsrfimpl" nodedef="osl_basicsrfdef"
file="basic_surface.osl"/>

<nodedef name="rmanris_basicsrfdef" type="surfaceshader" node="basic_srf"
target="rmanris">
    <input name="diff_albedo" type="color3" value="0.18,0.18,0.18"/>

```

```

    <input name="spec_color" type="color3" value="1,1,1"/>
    <input name="roughness" type="float" value="0.3"/>
</nodedef>
<implementation name="rmanris_basicsrffimpl" nodedef="rmanris_basicsrffdef"
    file="basic_srf.sl"/>

<material name="blendedmtl">
  <shaderref name="sr4" node="basic_surface">
    <bindinput name="albedo" type="color3" nodegraph="shaderparams"
      output="o_diffcolor"/>
    <bindinput name="speccolor" type="color3" nodegraph="shaderparams"
      output="o_speccolor"/>
    <bindinput name="roughness" type="float" nodegraph="shaderparams"
      output="o_roughness"/>
  </shaderref>
  <shaderref name="sr5" node="basic_srf">
    <bindinput name="diff_albedo" type="color3" nodegraph="shaderparams"
      output="o_diffcolor"/>
    <bindinput name="spec_color" type="color3" nodegraph="shaderparams"
      output="o_speccolor"/>
    <bindinput name="roughness" type="float" nodegraph="shaderparams"
      output="o_roughness"/>
  </shaderref>
</material>
</materialx>

```

Example 3: A material using post-shader compositing to blend the outputs of two surface shaders. A nodegraph containing two shader-semantic nodes and a blending operation is defined, then turned into a single shader which is then referenced by a material.

```

<materialx require="shadernode">
  <!-- Assume an external file defines nodegraph "shaderparams", with outputs-->
  <!-- "o_diffcolor1", "o_diffcolor2", "o_speccolor1" and "o_speccolor2". -->
  <!-- Define external "alSurface" shader -->
  <nodedef name="alSurfaceDef" type="surfaceshader" node="alSurface">
    <input name="DiffuseColor" type="color3" value="0.2,0.2,0.2"/>
    <input name="Specular1Color" type="color3" value="1,1,1"/>
    <parameter name="Specular1Roughness" type="float" value="0.3"/>
  </nodedef>

  <nodedef name="twolayersrffdef" type="surfaceshader" node="twoLayerSurface">
    <input name="diff1" type="color3" value="0.1,0.1,0.1"/>
    <input name="spec1" type="color3" value="1,1,1"/>
    <parameter name="roughness1" type="float" value="0.5"/>
    <input name="diff2" type="color3" value="0.1,0.1,0.1"/>
    <input name="spec2" type="color3" value="1,1,1"/>
    <parameter name="roughness2" type="float" value="0.5"/>
    <input name="mixamt" type="float" value="0"/>
  </nodedef>

  <nodegraph name="ng_twolayersrff" nodedef="twolayersrffdef">
    <alSurface name="als1" type="surfaceshader">
      <input name="DiffuseColor" type="color3" interfacename="diff1"/>
      <input name="Specular1Color" type="color3" interfacename="spec1"/>
      <parameter name="Specular1Roughness" type="float" interfacename="roughness1"/>
    </alSurface>
  </nodegraph>

```

```

</alSurface>
<alSurface name="als2" type="surfaceshader">
  <input name="DiffuseColor" type="color3" interfacename="diff2"/>
  <input name="Specular1Color" type="color3" interfacename="spec2"/>
  <parameter name="Specular1Roughness" type="float" interfacename="roughness2"/>
</alSurface>
<mix name="srfmix" type="surfaceshader">
  <input name="bg" type="surfaceshader" nodename="als1"/>
  <input name="fg" type="surfaceshader" nodename="als2"/>
  <input name="mask" type="float" interfacename="mixamt"/>
</mix>
<output name="o_out" type="surfaceshader" nodename="srfmix"/>
</nodegraph>

<material name="mblended1">
  <shaderref name="sr6" node="twoLayerSurface">
    <bindinput name="diff1" type="color3" nodegraph="shaderparams"
      output="o_diffcolor1"/>
    <bindinput name="spec1" type="color3" nodegraph="shaderparams"
      output="o_speccolor1"/>
    <bindparameter name="roughness1" type="color3" value="0.34"/>
    <bindinput name="diff2" type="color3" nodegraph="shaderparams"
      output="o_diffcolor2"/>
    <bindinput name="spec2" type="color3" nodegraph="shaderparams"
      output="o_speccolor2"/>
    <bindparameter name="roughness2" type="color3" value="0.6"/>
    <bindinput name="mixamt" type="float" nodegraph="shaderparams"
      output="o_mixamt"/>
  </shaderref>
</material>
</materialx>

```

Lights

It is not uncommon for Computer Graphics assets to include lights as part of the asset, such as the headlights of a car. MaterialX does not define actual "light" objects per se, but it does allow referencing externally-defined light objects in the same manner as geometry, via a UNIX-like path. MaterialX does not describe the view or geometry of a light object: there is no notion of position in space, animation, object/view/aim constraints, etc.: MaterialX presumes that these spatial properties are stored within the external geometry representation.

Since MaterialX treats lights like any other kind of geometry, lights can be turned off (muted) in looks by making the light geometry invisible, assignment of "light"-context shader materials can be done using a <materialassign> within a <look>, and illumination and shadowing assignments can be handled using <visibility> declarations for the light geometry. See the **Look Definition** section below for details.

Collections

Collections are recipes for building a list of geometries, which can be used as a shorthand for assigning a Material to a (potentially large) number of geometries in a Look. Collections can be built up from lists of specific geometries, geometries matching defined wildcard expressions, other collections, or any combination of those.

Collection Definition

A **<collection>** element consists of one or more `collectionadd` declarations and zero or more `collectionremove` declarations, contained within a `<collection>` element:

```
<collection name="collectionname">
  ...collectionadd/collectionremove declarations...
</collection>
```

To ensure greater compatibility between packages, a collection's name cannot be the same as a geometry name. The `collectionadd` and `collectionremove` declarations are processed in the order specified, so it is syntactically possible to build up the contents of a collection in pieces, add entire hierarchies of geometry and prune off unwanted "child" geometry, or add wildcard-matched geometry and remove unwanted specific matched geometries. The contents of a collection can itself be used to define a portion of another collection.

If an external file is capable of defining collections (e.g. in a geometry Alembic file), those collections can be referred to by Look assignments or any other place a `collection="name"` reference is allowed.

CollectionAdd Elements

CollectionAdd elements add geometries to a collection. There are two ways to specify geometry names to add: via a comma-separated list of explicit or wildcard names, or via the name of another collection:

```
<collectionadd name="name" geom="geomexpr1[,geomexpr2][,geomexpr3...]" />
<collectionadd name="name" collection="collectionname" />
```

Attributes for CollectionAdd elements:

- `name` (string, required): the unique name of the CollectionAdd element
- `geom` (geomnamearray, optional): the list of geometries and/or wildcard expressions that should be added to the collection
- `collection` (string, optional): the name of an external collection that should be added to this one

Since one can have as many `<collectionadd>` declarations as desired, it is not necessary to build one extremely long line to contain all the geometries in one collection.

CollectionRemove Elements

CollectionRemove elements remove specified geometries from a collection. They have the exact same attributes to remove a comma-separated list of geometry names or wildcard expressions:

```
<collectionremove name="name" geom="geomexpr1[,geomexpr2][,geomexpr3...]"/>
```

Attributes for CollectionRemove elements:

- `name` (string, required): the unique name of the CollectionRemove element
- `geom` (geomnamearray, required): the list of geometries and/or wildcard expressions that should be removed from the collection

Note that `<collectionremove>` does *not* support a `collection="."` attribute to remove the contents of one collection from another: this is due to incompatible differences in the way various contemporary packages process boolean combinations of collections.

For maximum compatibility, it is recommended to only use CollectionAdds possibly followed by CollectionRemoves when building Collections, rather than intermixing adds and removes.

Geometry Info Elements

Geometry Info ("geominfo") elements are used to define sets of named attributes with constant values, and to assign them to specific external geometries. GeomInfo elements can apply multiple attribute values to multiple pieces of geometry or to collections of geometry. It is acceptable for several geominfo elements to reference the same geometry or the same attribute, as long as no two geominfo elements try to assign values of the same geometry attribute to the same geometry.

The most common use for geominfo elements is to define the filenames (or portions of filenames) of texture map images mapped onto the geometry. Typically, there are several types of textures such as color, roughness, bump, opacity, etc. associated with each geometry: each would be a separate <geomattr> within the <geominfo>. Each of those images could contain texture data for multiple geometries, which would either be listed in the `geom` attribute of the <geominfo> element, or be assembled into a collection and the name of that collection would be specified in the `collection` attribute.

GeomInfo Definition

A <geominfo> element contains one or more geometry attribute definitions, and associates those geometry attribute values with all geometries listed in the `geom` or `collection` parameter:

```
<geominfo name="name" [geom="geomexpr1,geomexpr2,geomexpr3"] [collection="coll"]>
  ...geometry attribute definitions...
</geominfo>
```

Attributes for GeomInfo elements:

- `name` (string, required): the unique name of the GeomInfo element
- `geom` (geomnamearray, optional): the list of geometries and/or wildcard expressions that the GeomInfo is to apply to
- `collection` (string, optional): the name of a defined collection of geometries (see the Look and Collection Elements section)

GeomAttr Elements

GeomAttr elements define constant values directly associated with specific geometries. This could be application-specific metadata, attributes passed from a lighting package to a renderer, or values that can be substituted into filenames within image nodes (please see the **Image Filename Substitutions** section above for details on this last use case):

```
<geomattr name="attrname" type="attrtype" value="value"/>
```

The "value" can be any MaterialX type, but if a geomattr is used in an image filename substitution, it will be cast to a string before being substituted into the image filename, so string and integer values are recommended.

GeomAttr elements have the following attributes:

- name (string, required): the name of the geometry attribute to define.
- type (string, required): the geometry attribute's type.
- value (any MaterialX type, optional): the value to assign to that attribute for this geometry.

For example, one could specify a texture ID value associated with a geometry:

```
<geominfo name="gi1" geom="/a/g1">
  <geomattr name="txtid" type="integer" value="1001"/>
</geominfo>
```

and then reference that geomattr string in an image filename:

```
<nodegraph name="ng1">
  <image name="cc1" type="color3">
    <parameter name="file" type="filename"
      value="txt/color/asset.color.%txtid.tif"/>
  </image>
  ...
</nodegraph>
```

The %txtid in the file name would be replaced by whatever value the txtid GeomAttr had for each geometry.

One could also define the entire image name and apply to it several geometries at once, e.g.:

```
<geominfo name="gi2" geom="/a/g2,/a/g3,/a/g4">
  <geomattr name="imagename" type="string" value="images/color.rustleftside.tif"/>
</geominfo>
<nodegraph name="ng2">
  <image name="cc1" type="color3">
    <parameter name="file" type="filename" value="%imagename"/>
  </image>
  ...
</nodegraph>
```

or use multiple GeomAttrs to define different portions of various image names, e.g.:

```
<geominfo name="gi3" geom="/a/g5,/a/g7">
  <geomattr name="txtid" type="integer" value="1009"/>
  <geomattr name="clrname" type="string" value="color"/>
  <geomattr name="specname" type="string" value="specular"/>
</geominfo>
<geominfo name="gi4" geom="/a/g6,/a/g8,/a/g9">
  <geomattr name="txtid" type="integer" value="1010"/>
  <geomattr name="clrname" type="string" value="color3"/>
  <geomattr name="specname" type="string" value="specalt"/>
</geominfo>
<nodegraph name="ng3">
  <image name="cc2" type="color3">
    <parameter name="file" type="filename"
      value="txt/%clrname/asset.%clrname.%txtid.tif"/>
  </image>
  <output name="o_color2" type="color3" value="cc2"/>
  <image name="sc2" type="color3">
```

```

    <parameter name="file" type="filename"
      value="txt/%specname/asset.%specname.%txtid.tif"/>
  </image>
  <output name="o_spec2" type="color3" nodename="sc2"/>
</nodegraph>

```

Note: if there is a `fileprefix` set within the scope of the `<geominfo>`, the final value of the `GeomAttr` will *not* have the fileprefix prepended/appended, even if it looks like a filename: this is because fileprefix only affects values of type `filename`, not `string`. Rather, the fileprefix (if defined in the scope of a `nodegraph`) would be prepended/appended to the value of the "file" parameter in the `<image>` node.

GeomAttrDefault Elements

`GeomAttrDefault` elements define the default value for a specified `GeomAttr` name; this default value will be used in a filename string substitution if an explicit `geomattr` value is not defined for the current geometry. Since `GeomAttrDefault` does not apply to any geometry in particular, it must be used outside of a `<geominfo>` element.

```

<geomattrdefault name="txtid" type="integer" value="1000"/>
<geomattrdefault name="clrname" type="string" value="color"/>

```

Reserved GeomAttr Names

Workflows involving textures with implicitly-computed filename tokens based on `u,v` coordinates (such as `%UDIM` and `%UVTILE`) can be made more efficient by explicitly listing the set of values that these tokens resolve to for any given geometry. The MaterialX specification reserves two `geomattr` names for this purpose, `udimset` and `uvtileset`, each of which is a `stringarray` containing a comma-separated list of `UDIM` or `UVTILE` tokens:

```

<geominfo name="gi4" geom="/a/g1,/a/g2">
  <geomattr name="udimset" type="stringarray" value="1002,1003,1012,1013"/>
</geominfo>

<geominfo name="gi5" geom="/a/g4">
  <geomattr name="uvtileset" type="stringarray" value="_2U_1V,_2U_2V"/>
</geominfo>

```

Look and Property Elements

Look elements define the assignments of materials, visibility and other properties to geometries and geometry collections. In MaterialX, a number of geometries are associated with each stated material, visibility type or property in a look, as opposed to defining the particular material or properties for each geometry.

Property elements define non-material properties that can be assigned to geometries or collections in Looks. There are a number of standard MaterialX property types that can be applied universally for any rendering target, as well as a mechanism to define target-specific properties for geometries or collections.

A MaterialX document can contain multiple property and/or look elements.

Property Definition

A `<property>` element defines the name, type and value of a look-specific non-material property of geometry; `<propertyset>` elements are used to group a number of `<property>`s into a single named object. The connection between properties or propertysets and specific geometries or collections is done in a `<look>` element, so that these properties can be reused across different geometries, and enabled in some looks but not others. `<Property>` elements may only be used within `<propertyset>`s; they may not be used independently, although a dedicated `<propertyassign>` element may be used within a `<look>` to declare a property name, type, value and assignment all at once.

```
<propertyset name="set1">
  <property name="twosided" type="boolean" value="true"/>
  <property name="trace_maxdiffusedepth" target="rmanris" type="float" value="3"/>
</propertyset>
```

The following properties are considered standard in MaterialX, and should be respected on all platforms that support these concepts:

Property	Type	Default Value
<code>twosided</code>	boolean	false
<code>matte</code>	boolean	false

where `twosided` means the geometry should be rendered even if the surface normal faces away from camera, and `matte` means the geometry should hold out, or "matte" out anything behind it (including in the alpha channel).

In the example above, the `"trace_maxdiffusedepth"` property is target-specific, having been restricted to the context of Renderman RIS by setting its `target` attribute to "rmanris".

Look Definition

A **<look>** element contains one or more material, visibility and/or propertyset assignment declarations, optionally preceded by a **<lookinherit>** element:

```
<look name="lookname">
  ...optional <lookinherit> element...
  ...materialassign, visibility, property/propertysetassign declarations...
  ...optional <materialvar> elements...
</look>
```

Looks can inherit the assignments from another look by including a **<lookinherit>** element. The look can then specify additional assignments that will apply on top of/in place of whatever came from the source look. This is useful for defining a base look and then one or more "offshoot" or "variation" looks. It is permissible for an inherited-from look to itself inherit from another look, but a look can inherit from only one parent look.

```
<look name="lookname">
  <lookinherit name="lname" look="looktoinheritfrom">
    ...materialassign, visibility, property/propertysetassign declarations...
    ...optional <materialvar> elements...
  </lookinherit>
</look>
```

<look> elements also support other attributes such as `xpos`, `ypos` and `uicolor` as described in the Standard UI Attributes section above.

Look Assignment Elements

Look assignment elements are used to assign materials, categorized visibility and properties to specific geometries within a look. Each of the following assignment declaration element types can include any combination of `geom` attributes (of type "geomnamearray") and/or `collection` attributes.

The pathed names within `geom` attributes or stored within collections do not need to resolve strictly to "leaf" path locations or actual renderable geometry names: assignments can also be made to intermediate "branch" geometry path locations, which will then apply to any geometry at a deeper level in the path hierarchy which does not have another "closer to the leaf" level assignment. E.g. an assignment to `"/a/b/c"` will apply to `"/a/b/c/d"` and `"/a/b/c/foo/bar"` (and anything else whose full path name begins with `"/a/b/c/"`) if no other assignment is made to `"/a/b/c/d"`, `"/a/b/c/foo"`, or `"/a/b/c/foo/bar"`. If a look inherits from another look, the child look can replace assignments made to any specific path location (e.g. a child assignment to `"/a/b/c"` would take precedence over a parent look's assignment to `"/a/b/c"`), but an assignment by the parent look to a more "leaf"-level path location would take precedence over a child look assignment to a higher "branch"-level location.

MaterialAssign Elements

MaterialAssign elements are used within a **<look>** to connect a specified material to one or more geometries or collections.

```
<materialassign name="maname" material="materialname"
  [geom="geomexpr1[,geomexpr2...]] [collection="collectionname"]
  [exclusive=true|false]/>
```

Material assignments are generally assumed to be mutually-exclusive, that is, any individual geometry is assigned to only one material. Therefore, assign declarations should be processed in the order they appear in the file, and if any geometry appears in multiple <materialassign>s, the last <materialassign> wins. However, some applications allow multiple materials to be assigned to the same geometry as long as the shader node types don't overlap. If the `exclusive` attribute is set to false (default is true), then earlier material assigns will still take effect for all shader node types not defined in the materials of later assigns: for each shader node type, the shader within the last assigned material referencing a matching shader node type wins. If a particular application does not support multiple material assignments to the same geometry, the value of `exclusive` is ignored and only the last full material and its shaders are assigned to the geometry, and the parser should issue a warning. [REQ="multiassign" in order to set `exclusive=false`]

Visibility Elements

Visibility elements are used within a <look> to define various types of generalized visibility between a "viewer" object and other geometries. A "viewer object" is simply a geometry that has the ability to "see" other geometries in some rendering context and thus may need to have the list of geometries that it "sees" in different contexts be specified; the most common examples are light sources and a primary rendering camera.

```
<visibility name="vname" [viewergeom="objectname"]
  [geom="geomexpr1[,geomexpr2...]] [collection="collectionname"]
  [vistype="visibilitytype"] [visible="false"]/>
```

Visibility elements have the following attributes:

- `name` (string, required): the unique name of the Visibility element
- `viewergeom` (geomnamearray, optional): the list of viewer geometry objects that the <visibility> assignment affects
- `viewercollection` (string, optional): the name of a collection containing viewer geometry objects that the <visibility> assignment affects
- `geom` (geomnamearray, optional): the list of geometries and/or wildcard expressions that the `viewergeom` object should (or shouldn't) "see"
- `collection` (string, optional): the name of a defined collection of geometries that the `viewergeom` object should (or shouldn't) "see"
- `vistype` (string, optional): the type of visibility being defined; see table below
- `visible` (boolean, optional): if false, the geom/collection objects will be invisible to this particular type of visibility; defaults to "true".

The `viewergeom` attribute (and/or the contents of a collection referred to by the `viewercollection` attribute) typically refers to the name of a light (or list of lights) or other "geometry viewing" object(s). If `viewergeom/viewercollection` are omitted, the visibility applies to all applicable viewers (camera, light, geometry) within the given render context; `viewergeom/viewercollection` are not typically specified for `vistype` "camera". Either `geom` or `collection` must be defined.

The `vistype` attribute refers to a specific type of visibility. If a particular `vistype` is not assigned within a `<look>`, then all geometry is visible by default to all `viewergeoms` for that `vistype`; this means that to have only a certain subset of geometries be visible (either overall or to a particular `vistype`), it is necessary to first assign `<visibility>` with `visible="false"` to all geometry. Additional `<visibility>` assignments to the same `vistype` within a `<look>` are applied on top of the current visibility state. The following `vistypes` are predefined by MaterialX; applications are free to define additional `vistypes`:

Vistype	Description
camera	camera or "primary" ray visibility
illumination	geom/collection is illuminated by the viewergeom light(s)
shadow	geom/collection casts shadows from the viewergeom light(s)
secondary	indirect/bounce ray visibility of geom/collection to viewergeom geometry

If `vistype` is not specified, then the visibility assignment applies to *all* visibility types, and in fact will take precedence over any specific `vistype` setting on the same geometry: geometry assigned a `<visibility>` with no `vistype` and `visible="false"` will not be visible to camera, shadows, secondary rays, or any other ray or render type. This mechanism can be used to cleanly hide geometry not needed in certain variations of an asset, e.g. different costume pieces or alternate damage shapes.

If the `<visibility>` `geom/collection` refers to light geometry, then assigning `vistype="camera"` determines whether or not the light object itself is visible to the camera/viewer (e.g. "do you see the bulb"), while assigning `visible="false"` with no `vistype` will mute the light so it is neither visible to camera nor emitting any light.

For the "secondary" `vistype`, `viewergeom` should be renderable geometry rather than a light, to declare that certain other geometry is or is not visible to indirect bounce illumination or raytraced reflections in that `viewergeom`. In this example, `"/b"` would not be seen in reflections nor contribute indirect bounce illumination to `"/a"`:

```
<visibility name="v2" viewergeom="/a" geom="/b" vistype="secondary"
  visible="false"/>
```

PropertyAssign Elements

`PropertyAssign` and `PropertySetAssign` elements are used within a `<look>` to connect a specified property value or `propertyset` to one or more geometries or collections.

```
<propertyassign name="paname" property="propertyname" type="type" value="value"
  [target="target"]
  [geom="geomexpr1[,geomexpr2...]] [collection="collectionname"]/>
<propertysetassign name="psaname" propertyset="propertysetname"
  [geom="geomexpr1[,geomexpr2...]] [collection="collectionname"]/>
```

Multiple property/propertyset assignments can be made to the same geometry or collection, as long as there is no conflicting assignment made. If there are any conflicting assignments, it is up to the host application to determine how such conflicts are to be resolved, but host applications should apply property assignments in the order they are listed in the look, so it should generally be safe to assume that if two property/propertyset assignments set different values for the same property to the same geometry, the later assignment will win.

Look Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<materialx>
  <!-- <nodedef> and <material> elements to define Mplastic1,2 and Mmetal1,2 here -->
  <collection name="c_plastic">
    <collectionadd name="ca1" geom="/a/g1,/a/g2,/a/g5"/>
  </collection>
  <collection name="c_metal">
    <collectionadd name="ca2" geom="/a/g3,/a/g4"/>
  </collection>
  <collection name="c_lamphouse">
    <collectionadd name="ca3" geom="/a/lamp1/housing*Mesh"/>
  </collection>
  <collection name="c_setgeom">
    <collectionadd name="ca4" geom="/b/**"/>
  </collection>
  <nodedef name="nd_headlight1" type="lightshader" node="disk_lgt">
    <parameter name="emissionmap" type="filename" value=""/>
    <parameter name="gain" type="float" value="2000.0"/>
  </nodedef>
  <material name="mheadlight">
    <shaderref name="lgtsr1" node="disk_lgt">
      <bindparam name="gain" type="float" value="500.0"/>
    </shaderref>
  </material>
  <propertyset name="standard">
    <property name="displacementbound_sphere" target="rmanris" type="float"
      value="0.05"/>
    <property name="trace_maxdiffusedepth" target="rmanris" type="float" value="3"/>
  </propertyset>
  <look name="lookA">
    <materialassign name="ma1" material="Mplastic1" collection="c_plastic"/>
    <materialassign name="ma2" material="Mmetal1" collection="c_metal"/>
    <materialassign name="ma3" material="mheadlight" geom="/a/b/headlight"/>
    <visibility name="v1" viewergeom="/a/b/headlight" vistype="shadow" geom="/**"
visible="false"/>
    <visibility name="v2" viewergeom="/a/b/headlight" vistype="shadow"
collection="c_lamphouse"/>
    <propertysetassign name="psa1" propertysetname="standard" geom="/**"/>
  </look>
  <look name="lookB">
    <materialassign name="ma4" material="Mplastic2" collection="c_plastic"/>
    <materialassign name="ma5" material="Mmetal2" collection="c_metal"/>
    <propertysetassign name="psa2" propertysetname="standard" geom="/**"/>
    <!-- make the setgeom invisible to camera but still visible to shadows and
reflections -->
    <visibility name="v3" vistype="camera" collection="c_setgeom" visible="false"/>
  </look>
</materialx>
```