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

Doug Smythe - smythe@ilm.com
Jonathan Stone - jstone@lucasfilm.com
July 16, 2019

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. *Image processing networks* of sources, operators, connections and parameters, outputting a number of spatially-varying data streams.
2. *Geometry-specific information* such as associated texture filenames or IDs for various map types.
3. Associations between spatially-varying data streams and/or uniform values and the inputs of BxDF shaders, defining a number of *materials*.
4. Associations between materials and specific geometries to create a number of *looks*.

To our knowledge, there is no other common, open standard for describing 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.

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:

- A set of **standard nodes** for defining data-processing graphs
- **<nodedef>** for extending the standard node set with BxDF shaders and custom processing operators
- **<material>** for defining shader instances with bindings to spatially-varying data streams and uniform values
- **<geominfo>** for defining geometric attributes that may be referenced from node graphs
- **<look>** for defining specific combinations of material and property bindings to 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.

This document describes the core MaterialX specification. A companion document, **MaterialX Supplemental Notes**, describes additional node types and other information about the library.

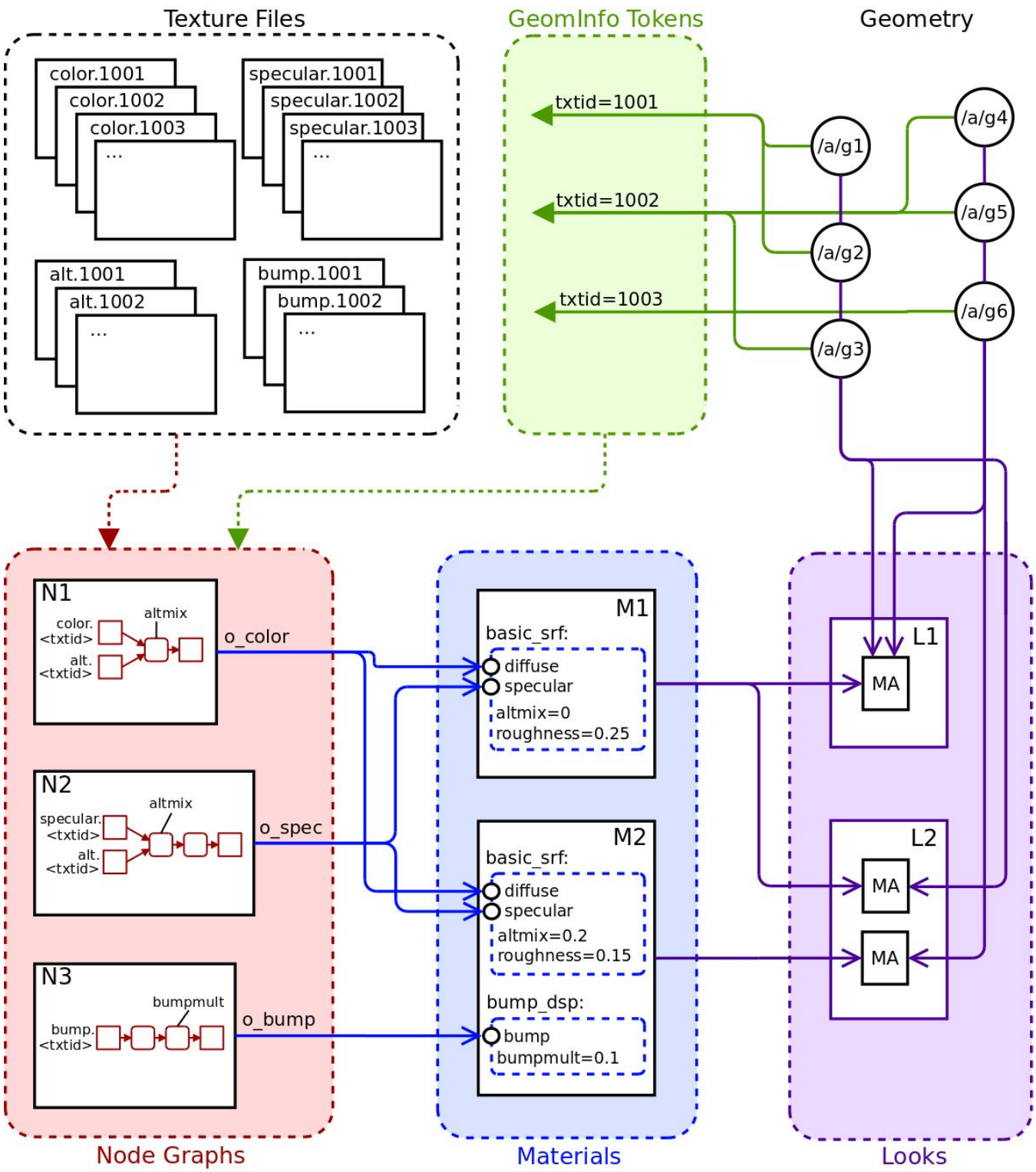
Table of Contents

MaterialX Overview

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 nodes, <geominfo>, <material>, <look> and other elements are described in the sections that follow.

Flow of information generally proceeds counterclockwise through the diagram. The green "GeomInfo Tokens" box shows how named attributes and string tokens can be associated with geometries. The red "Node Graphs" box defines a number of texture processing networks, which generally determine which input texture images to read by substituting GeomInfo Token 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 node graphs N1, N2 and N3, but set different values for the interface 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, **Inputs**, which may be spatially-varying, and **Tokens**, which are string values that can be substituted into image filenames.

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.

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 e.g. 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 (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 names are case-sensitive and are not allowed to begin with a digit.

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 enclosing <materialx> element, although any element within a document may provide a `colorspace` attribute that explicitly states the space in which color values within its scope should be interpreted; implementations are expected to translate those color values into the working color space before performing computations with those values. See the **Color Spaces and Color Management Systems** section below.

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 color*N* and vector*N* 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, 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 color*N* and vector*N* 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: literal text within quotes. See the **MTLX File Format Definition** section for details on representing special characters within string data.

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

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

IntegerArray, **FloatArray**, **Color2Array**, **Color3Array**, **Color4Array**, **Vector2Array**, **Vector3Array**, **Vector4Array**, **StringArray**: any number of values (including zero) 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". Individual string values within stringarrays may not contain commas or semicolons, and any leading and trailing whitespace characters in them is ignored. MaterialX does not support multi-dimensional 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. 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>, it can then 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" [root-level attributes]>
  <!-- various combinations of MaterialX elements and sub-elements -->
</materialx>
```

That is, a standard XML declaration line followed by a root <materialx> 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 and the XML character entities `&quot;`, `&`, `&apos;`, `<` and `>`:

```
<xi:include href="includedfile.mtlx"/>
<!-- this is a comment -->
<parameter name="example" type="string" value="&quot;text in quotes&quot;"/>
```

Each XIncluded document must itself be a valid MTLX file, containing an XML header and its own root `<materialx>` element, the children of which are added to the root element of the including document. Hierarchical root-level attributes such as `colorspace` and `namespace` are distributed to the included children to maintain correct semantics within the including MaterialX document. Global root-level attributes such as `cms` and `cmsconfig` must agree between including and included documents, and it is not considered valid to include a document with conflicting global settings.

Attributes for a <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.
- `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 **Color Spaces and Color Management Systems** section 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.
- `namespace` (string, optional): defines the namespace for all elements defined within this <materialx> scope. Please see the **MaterialX Namespaces** section below for details.

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, a display-referred space such as sRGB, or some 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 color 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_rec709"; 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_rec709"/>
</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 specification of "namespaces", which qualify the MaterialX names of all elements within their scope. Namespaces are specified via a `namespace` attribute in a <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. It is permissible for multiple <materialx> elements to specify the same namespace; the elements from each will simply be merged into the same namespace. <materialx> elements which do not specify a namespace will define elements into the (unnamed) global namespace. MaterialX namespaces are most commonly used to define families of custom nodes (nodedefs), material libraries, or commonly-used network shaders or nodegraphs.

References to elements in a different namespace are qualified using the syntax "*namespace*:*elementname*", where *namespace* is the namespace at the scope of the referenced element and *elementname* is the name of the referenced element. References to elements in the same namespace, or to elements in the global namespace, should not be qualified.

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 a node graph 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/) and USD (http://graphics.pixar.com/usd). "Geometry" can be any particular geometric object that a host application may support, including but not limited to polygons, meshes, subdivision surfaces, NURBS, implicit surfaces, particle sets, volumes, lights, 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 "/"s, 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 expressions (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 geometry name expression 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
```

Assignments to non-leaf locations apply hierarchically to all geometries below the specified location, unless they are the target of another assignment. By extension, an assignment to "/" applies to *all* geometries within the MaterialX setup, unless they are the target of another assignment.

Lights

Computer Graphics assets often include lights as part of the asset, such as the headlights of a car. MaterialX does not define "light" objects per se, but instead allows referencing externally-defined light objects in the same manner as geometry, via a UNIX-like path. MaterialX does not describe the position, view or shape of a light object: MaterialX presumes that these properties are stored within the external representation.

Light object geometries 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.

Geometry Name Expressions

Certain elements in MaterialX files support geometry specification via expressions. The syntax for geometry name expressions 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 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 "//*".

Collections

Collections are recipes for building a list of geometries (which can be any path within the geometry hierarchy), which can be used as a shorthand for assignments to a (potentially large) number of geometries at once. Collections can be built up from lists of specific geometries, geometries matching defined geometry name expressions, other collections, or any combination of those.

A **<collection>** element contains lists of geometry expressions and/or collections to be included, and an optional list of geometry expressions to be excluded:

```
<collection name="collectionname" [includegeom="geomexpr1[,geomexpr2]..."]
    [includecollection="collectionname1[,collectionname2]..."]
    [excludegeom="geomexpr3[,geomexpr4]..."]/>
```

Either `includegeom` and/or `includecollection` must be specified. The `includegeom` and `includecollection` lists are applied first, followed by the `excludegeom` list. This can be used to build up the contents of a collection in pieces, or to add expression-matched geometry then remove specific unwanted matched geometries. The contents of a collection can itself be used to define a portion of another collection. The contents of each `includecollection` collection are effectively evaluated in whole before being added to the collection being built.

If the containing file is capable of defining MaterialX-compliant collections (e.g. an Alembic or USD file), its collections can be referred to in any situation where a `collection="name"` reference is allowed.

Geometric Properties

Geometric Properties, or "geomprops", are intrinsic or user-defined surface coordinate properties of geometries referenced in a specific space and/or index, and are functionally equivalent to USD's concept of "primvars". A number of geometric properties are predefined in MaterialX: `position`, `normal`, `tangent`, `bitangent`, `texcoord` and `geomcolor`, the values of which can be accessed in nodegraphs using elements of those same names; see the **Geometric Nodes** section below for details. The value of a geometric property can also be used as the default value for a node input using a `defaultgeomprop` attribute.

One may also define custom geometric properties using a <geompropdef> element:

```
<geompropdef name="geompropname" type="geomproptype"
      [geomprop="geomproperty"] [space="geomspace"] [index="indexnumber"]/>
```

e.g.

```
<geompropdef name="Pworld" type="vector3" geomprop="position" space="world"/>
<geompropdef name="uv1" type="vector2" geomprop="texcoord" index="1"/>
```

The "geomprop", "space" and "index" attributes are optional; if "geomprop" is specified, it must be one of the standard geometric properties noted above, and if it is not specified, the new geomprop is a blind geometric property, e.g. one that can be referenced but which MaterialX knows no details about. The "space" and "index" attributes may only be specified if a "geomprop" attribute is specified and the standard geomproperty supports it.

Once defined, a custom geomprop name may be used any place that a standard geomprop can:

```
<nodedef name="ND1" ... internalgeomprops="position, Pworld, normal, uv1">
```

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>`, `<collection>`, `<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, e.g. to define the "asset root" path.

So the following MTLX file snippets are equivalent:

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

  <materialx geomprefix="/a/b/">
    <collection name="c_plastic" includegeom="g1, g2, g5, c/d/g6"/>
  </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, e.g. to define an asset's "texture root" directory.

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>
```

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.

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 within <>'s come from the current geometry, strings within []'s come from the MaterialX state, and strings within {}'s come from the host application environment.

<*geometry token*>	The value of a specified token declared in a <geominfo> element for the current geometry.
<UDIM>	A special geometry token 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 geometry token that will be replaced with the computed Mudbox-style "u*U*_v*V*" string, where *U* is 1+ the integer portion of the u coordinate, and *V* is 1+ the integer portion of the v coordinate.
[*interface token*]	The value of a specified token declared in the nodegraph's <nodedef> interface; the value for the token may be set in the <shaderref> of a material or within a <variant>; it is an error if the same token is defined in more than one of those places for the current geometry.
{*hostattr*}	The host application may 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.
{0*N*frame}	A special string that will be replaced by the current frame number padded with zeros 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 retain substitution strings within image file names upon export rather than "baking them out" into fully-evaluated filenames.

Nodes

Nodes are individual data generation or processing "blocks". Node functionality can range from simple operations such as returning a constant color value or adding two input values, to more complex image processing operations, 3D spatial data operations, or even complete shader BxDFs. Nodes are connected together into a network or "node graph", and pass typed data streams between them.

Individual node elements have the form:

```
<nodecategory name="nodename" type="outputdatatype" [version="version"]>
  <input name="paramname" type="type" [nodename="nodename"] [value="value"]/>
  <parameter name="paramname" type="type" value="value"/>
  ...additional input or parameter elements...
</nodecategory>
```

where *nodecategory* is the general "category" of the node (e.g. "image", "add" or "mix"), `name` (string, required) defines the name of this instance of the node, which must be unique within the scope it appears in, and `type` (string, required) specifies the MaterialX type (typically float, color*N*, or vector*N*) of the output of that node. If the application uses a different name for this instance of the node in the user interface, a `uiname` attribute may be added to the <*nodecategory*> element to indicate the name of the node as it appears to the user.

Node elements may optionally specify a `version` string attribute in "*major*[.*minor*]" format, requesting that a specific version of that node's definition be used instead of the default version. Please refer to the **Custom Node Declaration** section below for further details.

MaterialX defines a number of Standard Nodes which all implementations should support as described to the degree their architecture and capabilities allow. One can 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 by providing a `nodename` attribute, although inputs can be assigned a uniform constant value instead. An optional `output` attribute may also be provided for <input> elements, allowing the input to connect to a specific, named output of the referenced upstream node. If the referenced node has multiple outputs, `output` is required; if it has only one output, the `output` attribute of the <input> is ignored. Parameter elements exclusively provide uniform values for the source or operator. Parameter and input elements may be connected to an external parameter interface in the node definition, allowing them to be assigned values from materials.

Unless specified otherwise, all parameters and inputs default to a value of 0 in all channels for integer, float, color and vector types, "" for string, filename and geomname types, "false" for boolean types, the identity matrix for matrix types, and an empty array for array types.

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="rrrr"/>
  <input name="in2" type="color4" nodename="v2" channels="xyxy"/>
</add>
<multiply name="m2" type="vector2">
  <input name="in1" type="vector2" nodename="add4" channels="rg"/>
  <input name="in2" type="vector2" nodename="cnode4" member="minvec"
        channels="xy"/>
</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 combined to extract certain channels of an individual member of a custom type.

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

Parameter Expressions and Function Curves

Many applications allow parameters to have values defined by an expression or a function curve. While MaterialX does not currently support direct representations of parameter expressions or function curves, it does support evaluation of "baked" function curves, expressed as a one-dimensional array of per-frame values within a defined frame range. These `valuerange` and `valuecurve` attributes may be used in place of `value` for any <parameter>, <input>, <bindparam> or <bindinput> element. The

`valuerange` attribute is an integerarray 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`. Values at non-integer frame times are interpolated using a centripetal Catmull-Rom cubic spline curve fit through the specified frame values.

```
<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"/>
```

Node Graph Elements

A graph containing any number of nodes and output declarations forms a Node Graph, which may be enclosed within a <nodegraph> element to group them together into a single functional unit. Please see the **Node Graph Implementations** section below for details on how nodegraphs can be used to describe the functionality of new nodes, and the **BindInput Elements** section below for details on how to bind the output of a nodegraph to the input of a shader in a material.

```
<nodegraph name="graphname">
  ...node element(s)...
  ...output element(s)...
</nodegraph>
```

Output Elements

Output data streams are defined using **<output>** elements, and may be used to declare which output streams are connectable to other MaterialX elements. Within a node graph, an <output> element declares an output stream that may be connected to a shader input or to the input of a referencing node in another graph when the nodegraph is the implementation of a custom node. See the **Custom Nodes** section for details on the use of node graphs as node implementations.

```
<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, optional): the name of a node at the same scope within the document, whose result value will be output. This attribute is required for <output> elements within a node graph, but is not allowed in <output> elements within a <nodedef>.
- `output` (attribute, string, optional): if the node specified by `nodename` has multiple outputs, the name of the specific output to connect this <output> to.
- `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 node graphs 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 node graph.

```
<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>
```

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, with the lower-left corner of an image mapping to the (0,0) UV coordinate (or to the fractional (0,0) UV coordinate for tiled images). Parameters and inputs:
  - `file` (parameter, filename): 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.
  - `layer` (parameter, string): 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*): a default value to use if the `file` reference can not be resolved (e.g. if a <*geometry token*>, [*interface token*] or {*hostattr*} 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): 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): determines how U coordinates outside the 0-1 range are processed before sampling the image; see below. Default is "periodic".
  - `vaddressmode` (parameter, string): determines how V coordinates outside the 0-1 range are processed before sampling the image; see below. Default is "periodic".

- `filtertype` (parameter, string): the type of texture filtering to use; standard values include "closest" (nearest-neighbor single-sample), "linear", and "cubic". If not specified, an application may use its own default texture filtering method.

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

- "constant": Texture coordinates outside the 0-1 range return the value of the node's `default` parameter.
- "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.
- "mirror": Texture coordinates outside the 0-1 range will be mirrored back into the 0-1 range, e.g. u=-0.01 will return the u=0.01 texture coordinate value, and u=1.01 will return the u=0.99 texture coordinate value.

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

- `framerange` (parameter, string): 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): 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): what to do when the resolved image frame number is outside the `framerange` range:
  - "constant": Return the value of the node's `default` parameter (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*)
  - "mirror": Frame numbers "mirror" or "ping-pong" at the endpoints of framerange, so a read of the frame after *maxframe* will return the image from frame *maxframe*-1, and a read of the frame before *minframe* will return the image from frame *minframe*+1.

Arbitrary frame number expressions and speed changes are not supported.

Additional texture nodes, including <**`tiledimage`**> and <**`triplanarprojection`**>, may be found in the **MaterialX Supplemental Notes** document.

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="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 non-shader-semantic type): the value to output

- **`ramplr`**: a left-to-right linear value ramp. Parameters and inputs:
  - `valuel` (parameter, float or color*N* or vector*N*): the value at the left (U=0) edge
  - `valuer` (parameter, float or color*N* or vector*N*): the value at the right (U=1) edge
  - `texcoord` (input, vector2): 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.

- **`ramptb`**: a top-to-bottom linear value ramp. Parameters and inputs:
  - `valuet` (parameter, float or color*N* or vector*N*): the value at the top (V=1) edge
  - `valueb` (parameter, float or color*N* or vector*N*): the value at the bottom (V=0) edge
  - `texcoord` (input, vector2): 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*): the value at the left (U=0) edge
  - `valuer` (parameter, float or color*N* or vector*N*): the value at the right (U=1) edge
  - `center` (parameter, float): 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`. Default is 0.5.
  - `texcoord` (input, vector2): 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*): the value at the top (V=1) edge
  - `valueb` (parameter, float or color*N* or vector*N*): the value at the bottom (V=0) edge
  - `center` (parameter, float): a value representing the V-coordinate of the split; all pixels above "center" will be `valuet`, all pixels below "center" will be `valueb`. Default is 0.5.
  - `texcoord` (input, vector2): 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*): the center-to-peak amplitude of the noise (peak-to-peak amplitude is 2x this value). Default is 1.0.
  - `pivot` (parameter, float): 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): 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. Parameters and inputs:
  - `amplitude` (parameter, float or vector*N*): the center-to-peak amplitude of the noise (peak-to-peak amplitude is 2x this value). Default is 1.0.
  - `pivot` (parameter, float): 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): 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. Parameters and inputs:
  - `amplitude` (parameter, float or vector*N*): the center-to-peak amplitude of the noise (peak-to-peak amplitude is 2x this value). Default is 1.0.
  - `octaves` (parameter, integer): the number of octaves of noise to be summed. Default is 3.
  - `lacunarity` (parameter, float): the exponential scale between successive octaves of noise. Default is 2.0.
  - `diminish` (parameter, float): the rate at which noise amplitude is diminished for each octave. Should be between 0.0 and 1.0; default is 0.5.
  - `position` (input, vector3): 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): 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). Inputs:
  - `position` (input, vector3): 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.

- **`worleynoise2d`**: 2D Worley noise, outputting float (distance to closest feature), vector2 (distances to closest 2 features) or vector3 (distances to closest 3 features). Inputs:
  - `texcoord` (input, vector2): 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.
  - `jitter` (parameter, float): amount to jitter the cell center position, with smaller values creating a more regular pattern. Default is 1.0.

- **`worleynoise3d`**: 3D Worley noise, outputting float (distance to closest feature), vector2 (distances to closest 2 features) or vector3 (distances to closest 3 features). Inputs:
  - `position` (input, vector3): 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.
  - `jitter` (parameter, float): amount to jitter the cell center position, with smaller values creating a more regular pattern. Default is 1.0.

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 `ramp`*X*, `split`*X*, `noise2d` or `cellnoise2d` input coordinates, use a <texcoord> or similar Geometric node processed by vector2 <multiply>, <rotate> and/or <add> nodes, and connect to the node's `texcoord` input.

Additional source nodes, including <**`ramp4`**>, may be found in the **MaterialX Supplemental Notes** document.

Geometric Nodes

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

```
<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. This node must be of type vector3.
  - `space` (parameter, string): the name of the coordinate space in which the position is defined. Default is "object", see below for details.

- **`normal`**: the geometric normal associated with the currently-processed data, as defined in a specific coordinate space. This node must be of type vector3.
  - `space` (parameter, string): the name of the coordinate space in which the normal vector is defined. Default is "object", see below for details.

- **`tangent`**: the geometric tangent vector associated with the currently-processed data, as defined in a specific coordinate space. This node must be of type vector3.
  - `space` (parameter, string): the name of the coordinate space in which the tangent vector is defined. Default is "object", see below for details.
  - `index` (parameter, integer): the index of the texture coordinates against which the tangent is computed. The default index is 0.

- **`bitangent`**: the geometric bitangent vector associated with the currently-processed data, as defined in a specific coordinate space. This node must be of type vector3.
  - `space` (parameter, string): the name of the coordinate space in which the bitangent vector is defined. Default is "object", see below for details.
  - `index` (parameter, integer): the index of the texture coordinates against which the tangent is computed. The default index is 0.

- **`texcoord`**: the 2D or 3D texture coordinates associated with the currently-processed data. This node must be of type vector2 or vector3.
  - `index` (parameter, integer): 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. Can be of type float, color2, color3 or color4, and must match the type of the "color" bound to the geometry.
  - `index` (parameter, integer): the index of the color to be referenced, default is 0.

- **`geompropvalue`**: the value of the specified geometric property (defined using <geompropdef>) of the currently-bound geometry. This node's type must match that of the referenced geomprop.
  - `geomprop` (parameter, string): the geometric property to be referenced.
  - `default` (parameter, same type as the geomprop's value): a value to return if the specified `geomprop` is not defined on the current geometry.

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.
- "world": The global coordinate space of the geometry, after local deformations and global transforms have been applied.

Applications may also reference other renderer-specific named spaces, at the expense of portability.

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. 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): 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): the maximum distance from the surface point at which geometric surface features are considered as potential occluders. Defaults to 1e38, e.g. "unlimited".

Application Nodes

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

```
<frame name="f1" type="float"/>
<viewdirection name="viewdir1" type="vector3"/>
```

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): 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.

- **`viewdirection`**: the current scene view direction, as defined by the shading environment. This node must be of type vector3.
  - `space` (parameter, string): the space in which to return the view direction, defaults to "world".

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="n8"/>
  <input name="bg" type="color4" nodename="inbg"/>
</over>
<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 "mix" 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/matrix. See also the **Shader Nodes** section below for additional `add` variants supporting shader-semantic types.
  - `in1` (input, float or color*N* or vector*N* or matrix*NN*): the value or nodename for the primary input
  - `in2` (input, same type as `in1` or float): the value or nodename to add; for matrix types, the default is the zero matrix.

- **`subtract`**: subtract a value from the incoming float/color/vector/matrix, outputting "in1-in2".
  - `in1` (input, float or color*N* or vector*N* or matrix*NN*): the value or nodename for the primary input
  - `in2` (input, same type as `in1` or float): the value or nodename to subtract; for matrix types, the default is the zero matrix

- **`multiply`**: multiply an incoming float/color/vector/matrix by a value. Multiplication of two vectors is interpreted as a component-wise vector multiplication, while multiplication of two matrices is interpreted as a standard matrix product. To multiply a vector and a matrix, use one of the `transform*` nodes. See also the **Shader Nodes** section below for additional `multiply` variants supporting shader-semantic types.
  - `in1` (input, float or color*N* or vector*N* or matrix*NN*): the value or nodename for the primary input
  - `in2` (input, same type as `in1` or float): the value or nodename to multiply by; default is 1.0 in all channels for float/color/vector types, or the identity matrix for matrix types.

- **`divide`**: divide an incoming float/color/vector/matrix by a value; dividing a channel value by 0 results in floating-point "NaN". Division of two vectors is interpreted as a component-wise division of the first vector by the second, while division of two matrices is interpreted as a standard matrix product of the `in1` matrix and the inverse of the `in2` matrix.
  - `in1` (input, float or color*N* or vector*N* or matrix*NN*): the value or nodename for the primary input
  - `in2` (input, same type as `in1` or float): the value or nodename to divide by; default is 1.0 in all channels for float/color/vector types, or the identity matrix for matrix types.

- **`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*): the value or nodename for the primary input
  - `in2` (input, same type as `in1` or float): the modulo value or nodename to divide by, cannot be 0 in any channel; default is 1.0 in all channels

- **`invert`**: subtract the incoming float/color/vector from "amount" in all channels, outputting "amount-in". There is also an **`invert`** operator for matrix33 and matrix44 types, which returns the inverse of the matrix and has no `amount` parameter; if the input matrix is not invertible, the output matrix will consist of all floating-point "NaN" values.
  - `in` (input, float or color*N* or vector*N* or matrix*NN*): the input value or nodename
  - `amount` (parameter, same type as `in` or float): for float/color*N*/vector*N* types, the value to subtract the input value from; default is 1.0 in all channels

- **`absval`**: the per-channel absolute value of the incoming float/color/vector.
  - `in` (input, float or color*N* or vector*N*): the input value or nodename

- **`sign`**: the per-channel sign of the incoming float/color/vector value: -1 for negative, +1 for positive, or 0 for zero.
  - `in` (input, float or color*N* or vector*N*): the input value or nodename

- **`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*): the input value or nodename

- **`ceil`**: the per-channel nearest integer value greater 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*): the input value or nodename

- **`power`**: raise incoming float/color values to the specified exponent, commonly used for "gamma" adjustment.
  - `in1` (input, float or color*N* or vector*N*): the value or nodename for the primary input
  - `in2` (input, same type as `in` or float): exponent value or nodename; output = pow(in1, in2); default is 1.0 in all channels

- **`sin`**: the sine of the incoming value, which is expected to be expressed in radians.
  - `in` (input, float or vector*N*): the input value or nodename

- **`cos`**: the cosine of the incoming value, which is expected to be expressed in radians.
  - `in` (input, float or vector*N*): the input value or nodename

- **`tan`**: the tangent of the incoming value, which is expected to be expressed in radians.
  - `in` (input, float or vector*N*): the input value or nodename

- **`asin`**: the arcsine of the incoming value; the output will be expressed in radians.
  - `in` (input, float or vector*N*): the input value or nodename

- **`acos`**: the arccosine of the incoming value; the output will be expressed in radians.
  - `in` (input, float or vector*N*): the input value or nodename

- **`atan2`**: the arctangent of the incoming values (in2/in1); the output will be expressed in radians. If both `in1` and `in2` are provided, they must be the same type.
  - `in1` (input, float or vector*N*): the value or nodename for the "x" input; default is 1.0.
  - `in2` (input, float or vector*N*): the value or nodename for the "y" input; default is 0.0.

- **`sqrt`**: the square root of the incoming value.
  - `in` (input, float or vector*N*): the input value or nodename

- **`ln`**: the natural log of the incoming value.
  - `in` (input, float or vector*N*): the input value or nodename; default is 1.0.

- **`exp`**: "e" to the power of the incoming value.
  - `in` (input, float or vector*N*): the input value or nodename

- **`clamp`**: clamp incoming values per-channel to a specified range of float/color/vector values.
  - `in` (input, float or color*N* or vector*N*): the input value or nodename
  - `low` (parameter, same type as `in` or float): clamp low value; any value lower than this will be set to "low". Default is 0 in all channels.
  - `high` (parameter, same type as `in` or float): clamp high value; any value higher than this will be set to "high". Default is 1 in all channels.

- **`min`**: select the minimum of the two incoming values
  - `in1` (input, float or color*N* or vector*N*): the first value or nodename
  - `in2` (input, same type as `in1` or float): the second value or nodename

- **`max`**: select the maximum of the two incoming values
  - `in1` (input, float or color*N* or vector*N*): the first value or nodename
  - `in2` (input, same type as `in1` or float): the second value or nodename

- **`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*): the input value or nodename

- **`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*): the input value or nodename

- **`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*): the input value or nodename for the primary input.
  - `in2` (input, same type as `in1`): the secondary value or nodename

- **`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): the input value or nodename for the primary input.
  - `in2` (input, vector3): the secondary value or nodename

- **`transformpoint`**: transforms the incoming vector3 coordinate from one specified space to another, or a vector2/3 coordinate by the specified matrix; cannot be used on any other stream type. A vector2 input may be transformed using a matrix33, and a vector3 by a matrix33 or a matrix44; if necessary, the incoming vector value will be treated as if an additional "1.0" coordinate channel was appended to match the dimension of the transformation matrix. Either `tospace` (and optionally `fromspace`) or `mat` must be specified but not both.
  - `in` (input, vector2 or vector3): the input coordinate vector.
  - `fromspace` (parameter, string): the name of a vector space understood by the rendering target to transform the `in` point from; may be empty to specify the renderer's working or "common" space.
  - `tospace` (parameter, string): the name of a vector space understood by the rendering target for the space to transform the `in` point to.
  - `mat` (input, matrix3/4): the matrix used to transform the vector; default is the identity matrix.

- **`transformvector`**: transforms the incoming vector3 vector from one specified space to another, or a vector2/3/4 coordinate by the specified matrix; cannot be used on any other stream type. A vector2 input may be transformed using a matrix33, a vector3 by a matrix33 or matrix44, and a vector4 by a matrix44; if necessary, the incoming vector value will be treated as if an additional "0.0" coordinate channel was appended to match the dimension of the transformation matrix. Either `tospace` (and optionally `fromspace`) or `mat` must be specified but not both.
  - `in` (input, vector2/3/4): the input vector.
  - `fromspace` (parameter, string): the name of a vector space understood by the rendering

target to transform the `in` point from; may be empty to specify the renderer's working or "common" space.
    - `tospace` (parameter, string): the name of a vector space understood by the rendering target for the space to transform the `in` point to.
    - `mat` (input, matrix3/4): the matrix used to transform the vector; default is the identity matrix.

- **`transformnormal`**: transforms the incoming vector3 normal from one specified space to another, or by the specified matrix; cannot be used on any other stream type. A vector3 input may be transformed by a matrix44; the incoming vector value will be treated as if an additional "0.0" coordinate channel was appended. Either `tospace` (and optionally `fromspace`) or `mat` must be specified but not both.
    - `in` (input, vector3): the input normal vector; default is (0,0,1).
    - `fromspace` (parameter, string): the name of a vector space understood by the rendering target to transform the `in` point from; may be empty to specify the renderer's working or "common" space.
    - `tospace` (parameter, string): the name of a vector space understood by the rendering target for the space to transform the `in` point to.
    - `mat` (input, matrix3/4): the matrix used to transform the vector; default is the identity matrix.

- **`normalmap`**: transform a normal vector from object or tangent space into "world" space.
    - `in` (input, vector3): the input vector; default is (0.5, 0.5, 1.0).
    - `space` (parameter, string): the space to transform the normal from ("tangent" or "object"); defaults to "tangent".
    - `scale` (input, float): a scalar multiplier for the (x,y) components of the incoming vector; defaults to 1.0
    - `normal` (input, vector3): surface normal; defaults to the current world-space normal.
    - `tangent` (input, vector3): surface tangent vector, defaults to the current world-space tangent vector.

- **`transpose`**: outputs the transpose of the incoming matrix.
    - `in` (input, matrix*NN*): the input value or nodename

- **`determinant`**: outputs the float determinant of the incoming matrix*NN* stream.
    - `in` (input, matrix*NN*): the input value or nodename

- **`rotate`**: rotates a vector value about the origin.
    - `in` (input, vector2 or vector3): the input value or nodename
    - `amount` (input, float): the amount to rotate, specified in degrees; default is 0.
    - `axis` (parameter, vector3): For vector3 inputs only, the unit axis vector about which to rotate; default is (0,1,0).

- **`arrayappend`**: creates a two-element array [in1, in2] from two base-type values, or appends the in2 value to the end of the in1 array of the same type.
    - `in1` (input, integer/float/color*N*/vector*N*/string or integerarray/floatarray/color*N*array/ vector*N*array/stringarray): the input value or nodename for the first value or the array to append to. If `in1` is an array type with a value of "" (an empty array), then the output will be a single element array [`in2`].

- `in2` (input, integer/float/color*N*/vector*N*/string): the input value or nodename for the value to append; must be the same base type as `in1`.

Adjustment Nodes

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

- **`remap`**: linearly remap incoming values from one range of float/color/vector values to another. Parameters and inputs:
  - `in` (input, float or color*N* or vector*N*): the input value or nodename
  - `inlow` (input, same type as `in` or float): low value for input range; default is 0.0 in all channels
  - `inhigh` (input, same type as `in` or float): high value for input range; default is 1.0 in all channels
  - `outlow` (input, same type as `in` or float): low value for output range; default is 0.0 in all channels
  - `outhigh` (input, same type as `in` or float): high value for output range; default is 1.0 in all channels

- **`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*): the input value or nodename
  - `low` (input, same type as `in` or float): input low value; an input value of this or lower will result in an output value of 0; default is 0.0 in all channels
  - `high` (input, same type as `in` or float): input high value; an input value of this or higher will result in an output value of 1; default is 1.0 in all channels

- **`curveadjust`**: outputs a smooth remapping of input values using the centripetal Catmull-Rom cubic spline curve defined by specified knot values, using an inverse spline lookup on input knot values and a forward spline through output knot values. All channels of the input will be remapped using the same curve.
  - `in` (input, float or color*N* or vector*N*): the input value or nodename
  - `knots` (parameter, vector2array): the list of (input, output) value pairs defining the curve for the remapping. At least 2 vector2 values must be provided.

- **`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): the input value or nodename
  - `lumacoeffs` (parameter, color3): 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.

- **`rgbtohsv`**: (color3 or color4 only) convert an incoming color from RGB to HSV space (with H and S ranging from 0 to 1); the alpha channel is left unchanged if present. This conversion is not affected by the current color space or CMS configuration.
  - `in` (input, color3/color4): the input value or nodename

- **`hsvtorgb`**: (color3 or color4 only) convert an incoming color from HSV to RGB space; the alpha channel is left unchanged if present. This conversion is not affected by the current color space or CMS configuration.
  - `in` (input, color3/color4): the input value or nodename

Additional adjustment nodes, including <**`contrast`**>, <**`range`**>, <**`saturate`**> and <**`hsvadjust`**> may be found in the **MaterialX Supplemental Notes** document.

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 operate on 2-channel (color2) or 4-channel (color4) inputs/outputs, have one input named `in`, and either apply or unapply the alpha to the float or RGB color.

- **`premult`**: Multiply the R or RGB channels of the input by the Alpha channel of the input.
  - `in` (input, color2 or color4): the input value or nodename; default is (0,1) for color2 and (0,0,0,1) for color4.

- **`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): the input value or nodename; default is (0,1) for color2 and (0,0,0,1) for color4.

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 support an optional float input `mix`, which can be used to mix the original `bg` value (`mix`=0) with the result of the blend operation (`mix`=1, the default).

Blend Operator	Each Channel Output	Supported Types
`plus`	B+F	float, color*N*
`minus`	B-F	float, color*N*
`difference`	abs(B-F)	float, color*N*
`burn`	1-(1-B)/F	float, color*N*
`dodge`	B/(1-F)	float, color*N*
`screen`	1-(1-F)(1-B)	float, color*N*

overlay	2FB if F<0.5; 1-(1-F)(1-B) if F>=0.5	float, color*N*

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 vector*N* streams. Merge nodes support an optional float input `mix`, which can be used to mix the original `bg` value (`mix`=0) with the result of the blend operation (`mix`=1, the default).

Merge Operator	RGB output	Alpha Output
disjointover	F+B if f+b<=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). The default value for the `mask` input is 1.0 for the `inside` operator, and 0.0 for the `outside` operator 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 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 1-channel `mix` input and mixes the `fg` and `bg` according to the mix value. 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 `mix` input value (which has a default value of 0):

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 or equal to 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): the input value or nodename which is compared to `cutoff`.
  - `cutoff` (parameter, float): 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*): the input value or nodename to use if the output value of intest < cutoff.
  - `in2` (input, float or color*N* or vector*N*): the input value or nodename 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*): the values or nodenames to select from based on the value of the `which` parameter. The types of the various `in`*N* inputs must match the type of the `switch` node itself. If `which` is a boolean, then only `in1` and `in2` inputs are allowed. The default value of all `in` inputs is 0.0 in all channels.
  - `which` (parameter, boolean or integer or float): 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 (or "false" for boolean) will pass on the value from in1, 1<=`which`<2 (or "true" for boolean) 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. The default value of `which` is 0 (or "false").

Channel Nodes

Channel nodes are used to perform channel manipulations and data type conversions on float, color*N*, and vector*N* streams, allowing the order and number of channels to be modified, and the data types of streams to be altered.

- **`convert`**: convert a stream from one data type to another. Only certain unambiguous and commonly-needed conversions are supported; see list below.
  - `in` (input, boolean or integer or float or color*N* or vector*N*): the input value or nodename

- **`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 color*N* or vector*N*): the input value or nodename
    - `channels` (parameter, string): a string of one, two, three or four characters (one per channel in the output), each of which may be "r", "g", "b" or "a" for color*N* inputs, or "x", "y", "z" or "w" for vector*N* inputs. E.g. "bgra" would output a four-channel stream with the red and blue channels swapped, 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. Default is a string of characters that replicates the first channel from the input into all channels of the output.

- **`combine`**: combine the channels from two, three or four streams into the same number of channels of a single output 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): the input value or nodename which will be sent to the N channels of the output; default is 0.0 in all channels
    - `in2` (input, float/color2/vector2): the input value or nodename which will be sent to the next N channels of the output; default is 0.0 in all channels
    - `in3` (input, float): for 3- or 4-channel output types, the input value or nodename which will be sent to the next channel of the output after `in2`; default is 0.0
    - `in4` (input, float): for 4-channel output types, the input value or nodename which will be sent to the last channel of the output; default is 0.0

The following input/output data type conversions are supported by **`convert`**:

- float to color*N*/vector*N*: copy the input value to all channels of the output
- color*N* to vector*N* / vector*N* to color*N*, where *N* is the same for in and out: straight copy of channel values
- color3 to color4: copy RGB, set output alpha to 1.0
- color4 to color3: drop alpha channel
- boolean or integer to float: output is 0.0 or 1.0
- vector2 to vector3, or vector3 to vector4: copy incoming channels and append an additional channel with value 1.0 (e.g. convert from non-homogeneous to homogeneous vector)
- vector3 to vector2, or vector4 to vector3: drop the last channel; if a homogeneous vector conversion is desired, use a **`divide`** node with `in1 channels="xyz"` and `in2 channels="w"` instead.

Table of allowable input/output types for **`combine`**:

type (output)	in1	in2	in3	in4	Output Value
color2	float "r"	float "a"	n/a	n/a	"ra"
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"

Additional channel nodes, including <**extract**> and <**separate**>, may be found in the **MaterialX Supplemental Notes** document.

Convolution Nodes

Convolution nodes have one input named "in", and apply a defined convolution function on the input stream. Some of these nodes may not be implementable in ray tracing applications; they are provided for the benefit of purely 2D image processing applications.

- **blur**: a convolution blur.
  - in (input, float or color*N* or vector*N*): the input value or nodename
  - size (parameter, float): the size of the blur kernel, relative to 0-1 UV space; default is 0.
  - filtertype (parameter, string): the spatial filter used in the blur, either "box" for a linear box filter, or "gaussian" for a gaussian filter. Default is "box".

- **heighttonormal**: convert a scalar height map to a normal map of type vector3.
  - in (input, float): the input value or nodename
  - scale (parameter, float): 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 applications.

- **`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): the nodename to be connected to the Dot node's "in" input.
  - `note` (parameter, string): 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): a text note associated with the backdrop node; default is no text.
  - `contains` (parameter, stringarray): a comma-separated list of node names that the backdrop "contains"; default is to contain no nodes.
  - `width` (parameter, float): width of the backdrop when drawn in a UI, default is 1.0. [See xpos/ypos under Standard UI Attributes for a discussion of UI units]
  - `height` (parameter, float): height of the backdrop when drawn in a UI, default is 1.0.

For **`dot`** and **`backdrop`** nodes, the `note` text can contain standard HTML formatting strings, such as <b>, <ul>, <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): 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 elements support the following additional UI-related attributes:

- `doc` (attribute, string): a description of the function or purpose of this element; may include standard HTML formatting strings as described in the **Organizational Nodes** section above. May be used for functional documentation, or for UI pop-up "tool tip" strings.

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

- `xpos` (attribute, float): X-position of the node when drawn in a UI.
- `ypos` (attribute, float): Y-position of the node when drawn in a UI.
- `uicolor` (attribute, vector3): 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").

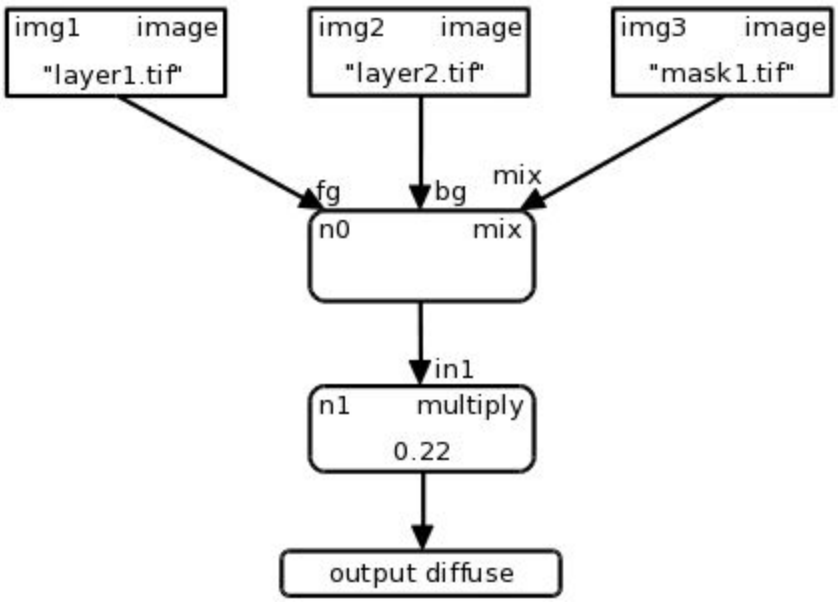
MaterialX defines xpos values to be increasing left to right, ypos values to be increasing top to bottom, and the general flow is generally downward. E.g. node inputs are on the top and outputs on the bottom, and a node at (10, 10) could connect naturally to a node at (10, 11). Content creation applications 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.

The <parameter>, <input> and <token> elements within <nodedef>s and node instantiations (but not within <implementation>s or <nodegraph> parameter interfaces) support the following UI-related attributes:

- `uivisible` (attribute, boolean): whether or not the parameter or input is visible in the UI. If `uivisible` is specified on a parameter/input/token in a <nodedef> that defines the default visibility of that parameter/input/token, while a `uivisible` specified on a parameter/input/token in a node instantiation affects just the visibility of the parameter/input/token within that particular instantiation. Default is "true".
- `uiadvanced` (attribute, boolean): whether or not the parameter or input is considered to be an "advanced" parameter which an application may choose to hide in a more "basic" mode. Should normally be declared only within a <nodedef>. Default is "false", meaning the parameter/input should be displayed if `uivisible` is true, while "true" means the parameter/input would be displayed if `uivisible` is true and the application UI is set to show "advanced" parameters.

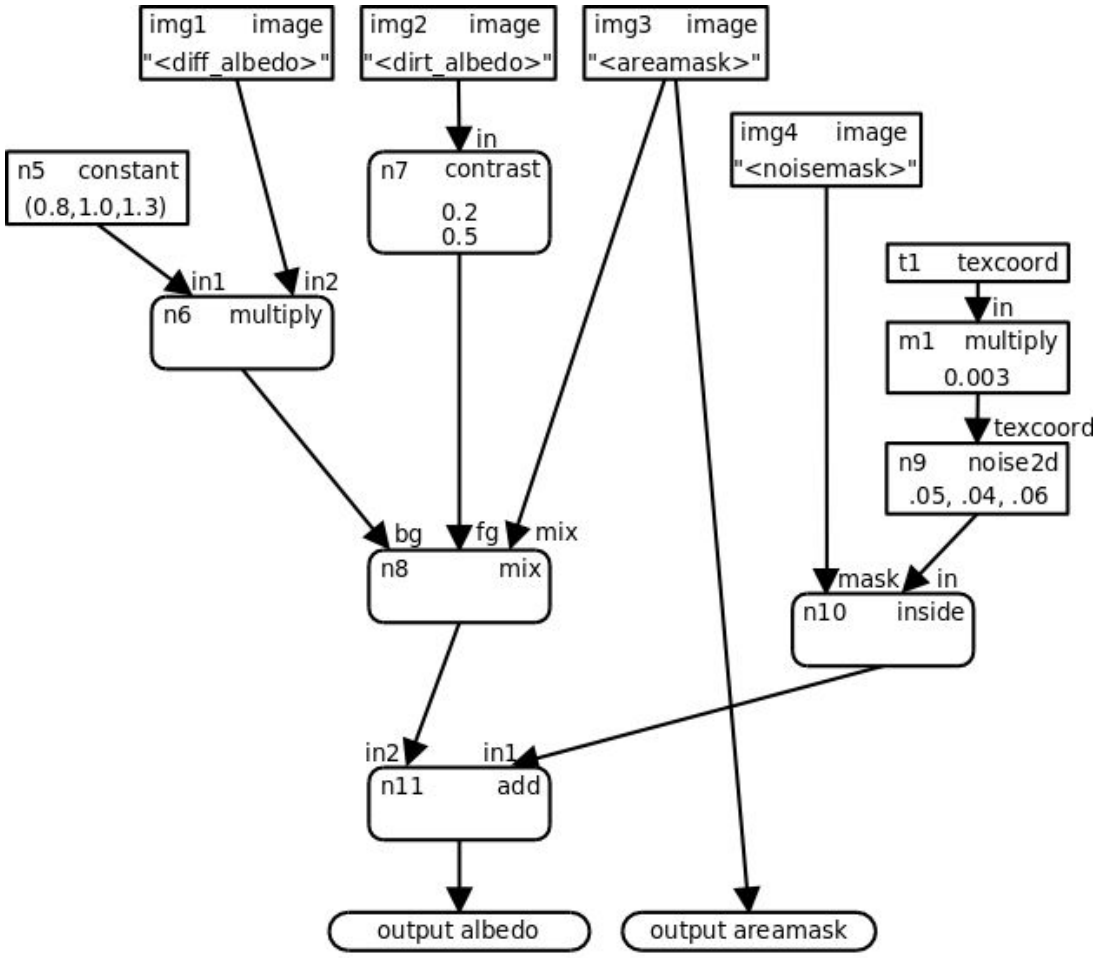
Node Graph Examples

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>
  <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="mix" type="float" nodename="img3"/>
  </mix>
  <multiply name="n1" type="color3">
    <input name="in1" type="color3" nodename="n0"/>
    <input name="in2" type="float" value="0.22"/>
  </multiply>
  <output name="diffuse" type="color3" nodename="n1"/>
</materialx>
```


Example 2: 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 graph outputs the area mask layer separately from the composited diffuse albedo color.




```
<?xml version="1.0" encoding="UTF-8"?>
<materialx>
  <!-- Note: in a real file, there would need to be geominfos here to define
<diff_albedo> etc. token values for each geometry-->
  <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>
  <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"/>
    <input name="amount" type="float" value="0.2"/>
    <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="mix" type="float" nodename="img3"/>
  </mix>
  <texcoord name="t1" type="vector2"/>
  <multiply name="m1" type="vector2">
    <input name="in1" type="vector2" nodename="t1"/>
    <input name="in2" type="float" value="0.003"/>
  </multiply>
  <noise2d name="n9" type="color3">
    <input name="texcoord" type="vector2" nodename="m1"/>
    <parameter name="amplitude" type="vector3" value="0.05,0.04,0.06"/>
  </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"/>
</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, 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, with child <parameter>, <input>, <token> and <output> elements 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
- `inherit` (string, optional): the `name` of a <nodedef> to inherit node definitions from; the output types of this nodedef and the inherited one must match, and the parameter/input/output definitions of this nodedef will be applied on top of those in the inherited-from one.
- `nodegroup` (string, optional): an optional group to which this node declaration belongs. Standard MaterialX nodes have `nodegroup` 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".
- `version` (string, optional): a version string for this nodedef, allowing usage of a node to reference a specific version of a node. Version strings should be of the format "*major*[.*minor*]", i.e. one or two integer numbers separated by a dot (the minor version is assumed to be "0" if not provided). If there are multiple nodedefs for the same `node` and `target` with the same combination of input and output types, they must each specify a `version`.
- `isdefaultversion` (boolean, optional): If true, then this nodedef should be used for node instances which do not request a specific version. Specifying `isdefaultversion` "true" is only required if there are multiple nodedefs for a node declaring a `version`, and it is not permissible for multiple nodedefs for the same `node` and `target` with the same combination of input and output types to set `isdefaultversion` "true". Defaults to "false".
- `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.
- `internalgeomprops` (stringarray, optional): a list of MaterialX geometric properties (e.g. "position", "normal", "texcoord", etc. or any name defined by a <geompropdef> element) that the node expects to be able to access internally. This metadata hint allows code generators to ensure this data is available and can be used for error checking. `Internalgeomprops` is most useful for nodes whose implementation is defined by external code; it is not necessary for nodegraph-defined nodes, as the list of geometric properties accessed can be determined by examining the nodegraph.

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 for the same target and version may be provided for a single `node` name.

The `inherit` attribute may be provided to allow one <nodedef> to inherit from another: this is most useful for defining additional parameters in a target- or version-specific <nodedef>, inheriting from a generic, canonical definition of a node or shader. NodeDefs which inherit from another nodedef may not re-declare <output>s from the parent nodedef, only add additional new <output>s.

NodeDefs must define one or more child <output> elements within the <nodedef> to state the name and type of each output; for nodes defined using a nodegraph, the names and types of the outputs must agree with the <output> elements in the nodegraph. The output name for a single-output <nodedef> is less important, as any connection made to the output of a single-output node will succeed regardless of the actual `name` referenced, although by convention, the name "out" is preferred for single-output nodes.

The parameter interface of a custom node is specified via a set of child <parameter>, <input> and <token> 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"]/>
```

Attributes for NodeDef 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 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 this parameter would be in error.
- `uiname` (string, optional): an alternative name for this parameter as it appears in the UI. If `uiname` is not provided, then `name` is the presumed UI name for the parameter.
- `uifolder` (string, optional): the pathed name of the folder in which this parameter appears in the UI, using a "/" character as a separator for nested UI folders.
- `enum` (attribute, stringarray, optional): a comma-separated list of string value descriptors that the parameter is allowed to take: for string- and stringarray-type parameters, these are the actual parameter values (or values per array index for stringarrays); for other types, these are the "enum" labels e.g. as shown in the application user interface for each of the actual underlying values specified by `enumvalues`. MaterialX itself does not enforce that a specified parameter value is actually in this list.
- `enumvalues` (attribute, *type*array, optional): for non-string/stringarray types, a comma-separated list of values of the same base type as the <parameter> or <input>, representing the values that would be used if the corresponding `enum` string was chosen in the UI. MaterialX itself does not enforce that a specified parameter value is actually in this list. Note that implementations are allowed to redefine `enumvalues` (but not `enum`) for specific targets: see the **Custom Node Definition** section below.
- `uimin` (attribute, integer or float or color*N* or vector*N*, optional): for parameters of type integer, float, color*N* or vector*N*, 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): for parameters of type integer, float, color*N* or vector*N*, the maximum value that the UI allows for this particular value. MaterialX itself does not enforce this as an actual maximum for value.

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

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

Attributes for NodeDef 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" or vector3-type custom geometric property for vector3 inputs, or "texcoord" or vector2-type custom geometric property for vector2 inputs. For standard geometric properties, this is effectively the same as declaring a default connection of the input to a Geometric Node with default parameters.
- `uiname` (attribute, string, optional): an alternative name for this input as it appears in the UI. If `uiname` is not provided, then `name` is the presumed UI name for the input.
- `uifolder` (attribute, string, optional): the pathed name of the folder in which this input appears in the UI, using a "/" character as a separator for nested UI folders.
- `uimin` (attribute, integer or float or color*N* or vector*N*, optional): for inputs of type integer, float, color*N* or vector*N*, 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): for inputs of type integer, float, color*N* or vector*N*, the maximum value that the UI allows for this particular value. MaterialX itself does not enforce this as an actual maximum for value.

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.

Token elements are used within a <nodedef> to declare uniform string-substitution values to be referenced and substituted within image filenames used in a node's nodegraph implementation:

```
<token name="tokenname" type="tokentype" [value="value"]/>
```

Attributes for NodeDef Token elements:

- `name` (string, required): the name of the token
- `type` (string, required): the MaterialX type of the token; when the token's value is substituted into a filename, the token value will be cast to a string, so string or integer types are recommended for tokens, although any MaterialX type is permitted.
- `value` (same type as `type`, optional): a default value for this token, to be used if the node is invoked without a value defined for this token. If a default value is not defined, then the token becomes required, so any invocation of the custom node without a value assigned to that token

would be in error.

- `uiname` (attribute, string, optional): an alternative name for this token as it appears in the UI. If `uiname` is not provided, then `name` is the presumed UI name for the token.
- `uifolder` (attribute, string, optional): the pathed name of the folder in which this token appears in the UI, using a "/" character as a separator for nested UI folders.

Please see Example 3 in the **Material Examples** section below for an example of how Tokens are used.

Output elements are used within a <nodedef> to declare an output for node definitions, including the output's name, type, and default value or "defaultinput" connection:

```
<output name="outputname" type="outputtype" [value="value"]/>
```

Attributes for NodeDef Output elements:

- `name` (string, required): the name of the output. For nodes with a single output, the name "out" is preferred.
- `type` (string, required): the MaterialX type of the output.
- `defaultinput` (string, optional): the name of an <input> element within the <nodedef>, which must be the same type as `type`, that will be passed through unmodified by applications that don't have an implementation for this node.
- `default` (same type as `type`, optional): 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.

The <output> elements for NodeDefs are similar to those for NodeGraph outputs, except that they may define default output values for the node but may not define a connection to another node (except for the `defaultinput` pass-through connection declaration) or any output file-related attributes such as width, height, colorspace or bitdepth.

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 external 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 is 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
- `implname` (string, optional): an alternative name for this node for the specified target; this allows one to say that for this particular target, the node/shader is called something else but is functionally equivalent to the node described by the nodedef. Note that node graphs in MaterialX documents should always use the node names defined in the nodedefs, never implementation-specific names.
- `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 external file that contains the implementation of this node.
- `language` (string, optional): when `file` is specified, the language in which the `file` code is written; defaults to "osl". Recommended values for `language` include "osl", "glsl", "hlsl", "mdl", and "cpp" (for C++).
- `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 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 `language` and/or `target` is intended to help applications differentiate between different versions of nodes and imply compatibility for specific situations, but does not necessarily guarantee compatibility: they 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 the reserved words in various shading languages, or may simply be different for specific targets, MaterialX allows <implementation> elements to contain a number of <input> and/or <parameter> elements to remap the `name`s of <input>s and <parameter>s as specified in the <nodedef> to different `implname`s 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 `implname`s 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_color3" nodedef is actually called "default_value" in the "mx_image_color" function:

```
<implementation name="IM_image_color3_osl" nodedef="ND_image_color3"
    file="mx_image_color.osl" function="mx_image_color" language="osl">
  <parameter name="default" type="color3" implname="default_value"/>
</implementation>
```

For parameters whose nodedef description includes an enum list of allowable values, individual implementations may associate different target-specific resolved values for them potentially of a different type; these may be described by providing an `enumvalues` attribute on the parameter within an <implementation> and if appropriate, an `impltype` to declare the target-specific type of these enumvalues. Note that if the type of the enum parameter in the nodedef is an array type, then the `impltype` (if specified) must also be an array type, while `enumvalues` is a list of values of the base

(non-array) type. The following <implementation> states that for the "mystudio" target, the uaddressmode and vaddressmode parameters of the "image" node are actually called "extrapolate_u" and "extrapolate_v", are integers rather than strings, and take different values (e.g. "clamp" is 2):

```
    <!-- In ND_image_color3, u/vaddressmode have enum="constant,clamp,periodic,mirror"
-->
    <implementation name="IM_image_color3_mystudio"
        nodedef="ND_image_color3" target="mystudio">
      <parameter name="uaddressmode” type="string"
        implname="extrapolate_u" impltype="integer" enumvalues="0, 2, 1, 3"/>
      <parameter name="vaddressmode” type="string"
        implname="extrapolate_v" impltype="integer" enumvalues="0, 2, 1, 3"/>
    </implementation>
```

Example of custom nodes defined with external file implementations:

```
  <nodedef name="ND_mariblend_color3" node="mariBlend">
    <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"/>
    <output name="out" type="color3" defaultinput="in1"/>
  </nodedef>
  <nodedef name="ND_mariblend_float" node="mariBlend">
    <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"/>
    <output name="out" type="float" defaultinput="in1"/>
  </nodedef>
  <nodedef name="ND_marinoise_color3" node="mariCustomNoise">
    <parameter name="ColorA" type="color3" value="0.5, 0.5, 0.5"/>
    <parameter name="Size" type="float" value="1.0"/>
    <output name="out" type="color3" default="0.5,0.5,0.5"/>
  </nodedef>
  <implementation name="IM_mariblend_color3_glsl" nodedef="ND_mariblend_color3"
      file="lib/mtlx_funcs.glsl" language="glsl"/>
  <implementation name="IM_mariblend_float_glsl" nodedef="ND_mariblend_float"
      file="lib/mtlx_funcs.glsl" language="glsl/>
  <implementation name="IM_marinoise_color3_glsl" nodedef="ND_marinoise_color3"
      file="lib/mtlx_funcs.glsl" language="glsl/>
  <implementation name="IM_mariblend_color3_osl" nodedef="ND_mariblend_color3"
      file="lib/mtlx_funcs.osl" language="osl"/>
  <implementation name="IM_mariblend_float_osl" nodedef="ND_mariblend_float"
      file="lib/mtlx_funcs.osl" language="osl"/>
  <implementation name="IM_marinoise_color3_osl" nodedef="ND_marinoise_color3"
      file="lib/mtlx_funcs.osl" language="osl"/>
  <implementation name="IM_marinoise_color3_osl_vray" nodedef="ND_marinoise_color3"
      file="lib/mtlx_vray_funcs.osl" language="osl" target="vray"/>
```

This 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 VRay, as if the author had determined that the generic OSL version was not appropriate for that renderer.

Here is an example of a two-output node definition and external implementation declaration.

```
<nodedef name="ND_doublecolor_c3c3" node="doublecolor">
  <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="IM_doublecolor_c3c3_osl" nodedef="ND_doublecolor_c3c3"
    file="lib/mtlx_funcs.osl" language="osl"/>
```

Node Graph Implementations

Alternatively, a custom node's implementation may be described using a Node Graph. A <nodegraph> element wraps a graph of standard or custom nodes, taking the inputs and parameters and producing the output(s) described in the specified <nodedef>.

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"]>
  [...parameter/input element(s)...]
  ...node element(s)...
  ...output element(s)...
</nodegraph>
```

The <nodegraph> element specifies a `nodedef` attribute (and optionally a `target` attribute as well) to indicate 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(s) of the <output>(s) of the <nodedef>) and the type(s) of the nodegraph <output>(s) must agree, and if there are multiple outputs, then the `name`s of the <output>s in the <nodegraph> and <nodedef> must also agree. The inputs and parameters of the <nodedef> can 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="in2" type="color3" interfacename=”i2”/>
```

Note that it is acceptable for the `interfacename` in an <input> of a node within the nodegraph to reference a parameter in the nodedef, but an `interfacename` in a <parameter> of a node within the nodegraph may not reference an input in the nodedef.

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, though in general if there was both a <nodegraph> and an <implementation> for the same nodedef target/version, the <implementation> should prevail in order to allow optimized native-code node implementations.

The <nodegraph> itself may contain target-specific top-level <parameter> and/or <input> elements, which may be used to modify the names of parameters and/or inputs or the impltypes and enumvalues of enum parameters described in the referenced <nodedef>, or to declare additional target-specific parameters and/or inputs. These declarations work in exactly the same manner as described above for Implementation elements.

Example of a custom node defined using a nodegraph:

```
<nodedef name="ND_blendadd_color4" node="blend_add">
  <input name="fg" type="color4" value="0,0,0,0"/>
  <input name="bg" type="color4" value="0,0,0,0"/>
  <parameter name="amount" type="float" value="1.0"/>
  <output name="out" type="color4" defaultinput="bg"/>
</nodedef>
<nodegraph name="NG_blendadd_color4" nodedef="ND_blendadd_color4">
  <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="out" type="color4" nodename="n2"/>
</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" and "bg" inputs provide default values which are used if an 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".

Custom Node Use

Once defined with a <nodedef>, invoking a custom node within a node graph 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>
```

When invoking nodes with multiple outputs, the `type` of the node should be declared as "multioutput", and other node inputs connecting to an output of the node must include an `output` attribute to specify which output of the node to connect to:

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

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" node="shaderfunctionname">
  ...parameter and input declarations...
  <output name="out" type="shadertype"/>
</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.
- `node` (string, required): the name of the shader node being defined, which typically matches the name of an associated shader function such as “blinn_phong”, “Disney_BRDF_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.

The child <output> element within the <nodedef> defines 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.

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="ND_unifiedsrf_studio" node="unified_srf">
  <input name="diffc" type="color3" value="0.18,0.18,0.18"/>
  <parameter name="spec1roughness" type="float" value="0.3"/>
  ...
  <output name="out" type="studio_aovs"/>
</nodedef>
```

It should be noted that the primary benefit of declaring and using specific types for shader nodes would be to differentiate which shader node outputs can be connected into other node 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 so use of the standard "surfaceshader" etc. types should be sufficient in most situations.

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.

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 or bound to new (uniform) values, or left at their declared default (uniform) values.

An input with a shader-semantic type may be given a value of "" to indicate no shader node is connected to this input; this is typically the default for shader-semantic inputs of operator nodes. It is up to applications to decide what to do with unconnected shader-semantic inputs.

Standard Shader-Semantic Operator Nodes

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

- **`add`**: add two surface/displacement/volumeshader closures.
  - `in1` (input, surface/displacement/volumeshader, required): the name of the first shader-semantic node
  - `in2` (input, surface/displacement/volumeshader, required): the name of the first shader-semantic node

- **`multiply`**: multiply a surface/displacement/volumeshader closure by a float or color3/vector3 value: surfaceshaders and volumeshaders may be multiplied by a float or color3, while displacementshaders may be multiplied by a float or vector3.
  - `in1` (input, surface/displacement/volumeshader, required): the name of the input shader-semantic node
  - `in2` (input, float or color3 or vector3, required): the value to multiply the closure by

- **`mix`**: linear blend between two surface/displacement/volumeshader closures.
  - `bg` (input, surface/displacement/volumeshader, required): the name of the background shader-semantic node
  - `fg` (input, surface/displacement/volumeshader, required): the name of the foreground shader-semantic node
  - `mix` (input, float, required): the blending factor used to mix the two input closures

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.

If an application requires additional information related to any MaterialX element, it may define and utilize additional attributes with non-standard names. Custom attributes are defined using <attributedef> elements:

```
<attributedef name="name" attrname="attrname" type="type" value="defaultvalue"
```



```
        [target="targets"] [elements="elements"]/>
```

where *name* is a unique name for the attributedef, *attrname* is the name of the custom attribute to define, *type* is the type of the attribute (typically string, stringarray, integer or boolean, although any MaterialX type is allowed), *defaultvalue* is the default value for the attribute, *target* is an optional list of targets to which this attribute applies, and *elements* is an optional list of element names or elementname/parametername in which the attribute may be used. Examples:

```
<attributedef name="AD_maxmtlname" attrname="maxmtlname" type="string" value=""
       target="3dsmax" elements="material"/>
<attributedef name="AD_img_vflip" attrname="vflip" type="boolean" value="false"
       target="mystudio" elements="image/file"/>
```

The first example above defines a 3ds Max-specific name attribute for materials, which may be given a value in addition to its MaterialX-compliant name in order to preserve the original package-specific name; it is assumed here that `maxmtlname` is the attribute name used by that particular implementation for this purpose. The second example defines a "mystudio"-specific boolean attribute "vflip", which could be used in the "file" parameter of <image> nodes.

Once defined, custom attributes may be used in exactly the same manner as standard attributes:

```
<material name="sssmarble" maxmtlname="SSS Marble">
  <shaderref name="sr1" node="marblesrf"/>
</material>
<image name="im1" type="color3">
  <parameter name="file" type="filename" value="X.tif" vflip="true"/>
  ...
</image>
```

If an application requires additional custom parameters or inputs within a standard MaterialX node, it may define a target application-specific <nodedef> for that node inheriting the base parameter/input definitions from the standard node's <nodedef>, then add parameters and inputs specific to that target application.

```
<nodedef name="ND_image_color4_maya" node="image" target="maya"
    inherit="ND_image_color4">
  <parameter name="preFilter" type="boolean" value="true"/>
</nodedef>
```

In the above example, a Maya-specific version of the color4-type <image> node has been declared, inheriting from the standard declaration then adding a maya-specific "preFilter" parameter.

When using a node, the definition appropriate for the current target will automatically be used, and other targets will ignore any inputs or parameters that are not part of the nodedef for that target. However, one may specify a documentational `target` attribute on parameters or inputs to hint what target the parameter is intended for if desired. In this example, the "preFilter" parameter has indicated that it is specific to the "maya" target.

```
<image name="image1" type="color4">
  <parameter name="file" type="filename" value="image1.tif"/>
```



```
  <parameter name="preFilter" type="boolean" value="true" target="maya"/>
</image>
```

Please see the **Custom Node Definition** section above for further details.

Materials

Material Elements

Material elements are used to define instantiations of shader nodes, and to bind their parameters and inputs to uniform data values and spatially-varying data streams.

A **<material>** element contains one or more **<shaderref>** elements, which define what shaders a material references.

```
<material name="materialname" [inherit="materialtoinheritfrom"]>
  ...<shaderref> elements (optional if inherit provided)...
</material>
```

Attributes for <material> elements:

- `name` (string, required): the name of the material.
- `inherit` (string, optional): the name of another material to inherit the shader references and bindings from.

If an `inherit` attribute is provided, then the shader references and bindings of the inherited material will be applied first, and the ones defined within this <material> applied on top of the inherited ones. For maximum compatibility, it is recommended that materials that inherit from other materials only include value and input bindings and not add or change shaders.

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

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.
- `version` (string, optional): the version string of the node to use; if not specified, the default version of the referenced node will be used.
- `nodedef` (string, optional): the name of a <nodedef> defining a specific shader-semantic node. Since this refers to a specific <nodedef> element, a `version` string is neither needed nor allowed with `nodedef`. `nodedef` takes precedence over `node` if both are specified in a <shaderref>.
- `type` (string, optional): the type of the referenced shader-semantic node; if provided, it must match that of the `node`/`nodedef`'s output; this is provided mainly for documentational and

type-checking purposes, since the actual node definition could likely be in a different (e.g. XIncluded) file. If the referenced shader-semantic node has multiple outputs, e.g. both a "surfaceshader" and a "displacementshader", then `type` should be "multioutput".

- `target` (string, optional): the target to which this shaderref should apply; if specified, it must match that of the referenced <nodedef>.

Either `node` or `nodedef` must be specified. 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="ND_DisneyBRDF2012_surface"
    node="Disney_BRDF_2012">
  ...
  <output name="out" type="surfaceshader"/>
</nodedef>
```

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

```
<shaderref name="sref1" node="Disney_BRDF_2012">
```

but could alternatively be referenced like this:

```
<shaderref name="sref2" nodedef="ND_DisneyBRDF2012_surface">
```

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 node graph.

```
<material name="steel">
  <shaderref name="sref3" node="simplesrf">
    <bindparam name="emissionColor" type="color3" value="0.005, 0.005, 0.005" />
    <bindparam name="rfrIndex" type="float" value="1.33" />
  </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, required): a value to bind to the shader parameter within the

scope of this material.

BindInput Elements

BindInput elements are used within <shaderref> elements to dynamically bind values or node graph 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 node graph.

```
<material name="steel">
  <shaderref name="sref4" node="simplesrf">
    <bindinput name="diffColor" type="color3" nodegraph="DiffNoiseNetwork"
        output="o_diffColor"/>
    <bindinput name="specColor" type="color3" output="o_specColor"/>
    <bindinput name="roughness" type="float" value="0.02"/>
  </shaderref>
</material>
```

In the above example, output "o_specColor" is presumed to be defined in a free-standing network of nodes, while output "o_diffColor" is defined within the "DiffNoiseNetwork" nodegraph.

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.
- `nodegraph` (string, optional): the name of the nodegraph element whose output will be bound
- `output` (string, optional): the name of the nodegraph output that will be bound to this input

Either `value` or `output` must be declared, but not both.

BindToken Elements

BindToken elements are used within <shaderref> elements to dynamically bind values to shader node interface tokens, replacing any default assignments in the original <token> elements. These bindings persist only within the scope of the enclosing <shaderref> element. BindTokens can only be applied to shader nodes referenced by a <shaderref> of this material; they cannot apply to a shader node contained within a node graph.

```
<material name="steel_wet">
  <shaderref name="sref5" node="simplesrf">
    <bindtoken name="diffmap" type="string" value="diff_wet2"/>
    <bindtoken name="specmap" type="string" value="spec_wet2"/>
  </shaderref>
</material>
```

Attributes for Bindtoken elements:

- `name` (string, required): the `name` of the shader <token> which will be bound to a new value
- `type` (string, required): the MaterialX type of the shader <token>
- `value` (specified MaterialX type, required): a value to bind to the shader token within the scope of this material.

Material Variants

A Variant is a container for any number of uniform values for material parameters, inputs, and interface tokens. One or more mutually-exclusive variants are defined as part of a <variantset>; variants may not be defined outside of a <variantset>.

```
<variantset name="wetvars">
  <variant name="wet1">
    <token name="diffmap" type="string" value="diff_wet1"/>
    <token name="specmap" type="string" value="spec_wet1"/>
    <parameter name="roughness" type="float" value="0.001"/>
  </variant>
  ...additional <variant> declarations for this variantset...
</variantset>
```

<Input> elements within a <variant> may only define a `value`, not a connection to a node or <output>.

Example uses for variants include defining a number of allowable colors and texture tokens for different costume variations, and defining values for progressively increasing levels of damage to a model.

Variants and variantsets are not intrinsically associated with any particular material; they merely state a number of values for a number of named parameters/inputs/tokens. However, variantsets may state that they are associated with specific shader-semantic nodes and/or <nodedef> declarations by providing stringarray-type `node` and/or `nodedef` attributes:

```
<variantset name="damagevars" node="Disney_BRDF_2012,Disney_BRDF_2015">
  ...
<variantset name="costumevars" nodedef="ND_unifiedsrf_studio">
  ...
```

Variants are applied to materials within <look>s; please see the **Look Assignment Elements** section below for information on using variants.

Material Examples

Example 1: Define two shaders and two materials with different bindings assigned to the shader(s). The first material references both a surface and a displacement shader, while the second references only a surface shader. In each case, some inputs do not have explicit value bindings, so their default values are used.

```
<?xml version="1.0" encoding="UTF-8"?>
<materialx>
```



```
  <nodedef name="ND_simplesrf_surface" node="simplesrf">
    <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"/>
    <output name="out" type="surfaceshader"/>
  </nodedef>

  <nodedef name="ND_noisebump_displacement"
           node="noisebump">
    <parameter name="bump_scale" type="float" value="0.02"/>
    <parameter name="bump_ampl" type="float" value="0.015"/>
    <output name="out" type="displacementshader"/>
  </nodedef>

  <material name="material1">
    <shaderref name="sr1" node="simplesrf">
      <bindinput name="diff_albedo" type="color3" value="0.31, 0.14, 0.09"/>
      <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="noisebump">
      <bindparam name="bump_ampl" type="float" value="0.0125"/>
    </shaderref>
  </material>

  <material name="material2">
    <shaderref name="sr3" node="simplesrf">
      <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 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>
  <!-- Define a basic surface shaders with two implementations; first might be
       applicable to several renderers, second is specific to rmanris and has
       slightly different parameter names.
  -->
  <nodedef name="ND_basic_surface_surface"
           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="0.25"/>
  <output name="out" type="surfaceshader"/>
</nodedef>

<implementation name="IM_basicsurface_surface_osl"
      nodedef="ND_basic_surface_surface" file="basic_surface.osl"/>
<implementation name="IM_basicsurface_surface_rmanris"
      nodedef="ND_basic_surface_surface" implname="basic_srf"
      target="rmanris" file="basic_srf.C" language="cpp">
  <input name="diff_albedo" type="color3" implname="diffColor"/>
  <input name="spec_color" type="color3" implname="specColor"/>
  <input name="roughness" type="float" implname="specRoughness"/>
</implementation>

<!-- Define interface and shading network for a simple blended material driven
     by mask image files.
-->
<nodedef name="ND_triblendsrf_surface" node="triblendsrf">
  <parameter name="paintmaskfile" type="filename"/>
  <parameter name="rustmaskfile" type="filename"/>
  <output name="out" type="surfaceshader"/>
</nodedef>

<nodegraph name="NG_triblendsrf_surface" nodedef="ND_triblendsrf_surface">
  <!-- Material constants for base layer, "steel" -->
  <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>

  <!-- Material constants for middle layer, "paint" -->
  <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>

  <!-- Material constants for top layer, "rust" -->
  <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>
```



```
  <!-- Blending masks -->
  <image name="mask_paint" type="float">
    <parameter name="file" type="filename" interfacename="paintmaskfile"/>
  </image>
  <image name="mask_rust" type="float">
    <parameter name="file" type="filename" interfacename="rustmaskfile"/>
  </image>

  <!-- Define blended values for diffcolor, speccolor, roughness -->
  <mix name="mix_diff1" type="color3">
    <input name="bg" type="color3" nodename="steel_diffc"/>
    <input name="fg" type="color3" nodename="paint_diffc"/>
    <input name="mix" type="float" nodename="mask_paint"/>
  </mix>
  <mix name="mix_diff2" type="color3">
    <input name="bg" type="color3" nodename="mix_diff1"/>
    <input name="fg" type="color3" nodename="rust_diffc"/>
    <input name="mix" type="float" nodename="mask_rust"/>
  </mix>

  <mix name="mix_spec1" type="color3">
    <input name="bg" type="color3" nodename="steel_specc"/>
    <input name="fg" type="color3" nodename="paint_specc"/>
    <input name="mix" type="float" nodename="mask_paint"/>
  </mix>
  <mix name="mix_spec2" type="color3">
    <input name="bg" type="color3" nodename="mix_spec1"/>
    <input name="fg" type="color3" nodename="rust_specc"/>
    <input name="mix" type="float" nodename="mask_rust"/>
  </mix>

  <mix name="mix_rough1" type="float">
    <input name="bg" type="float" nodename="steel_roughf"/>
    <input name="fg" type="float" nodename="paint_roughf"/>
    <input name="mix" type="float" nodename="mask_paint"/>
  </mix>
  <mix name="mix_rough2" type="float">
    <input name="bg" type="float" nodename="mix_rough1"/>
    <input name="fg" type="float" nodename="rust_roughf"/>
    <input name="mix" type="float" nodename="mask_rust"/>
  </mix>

  <!-- Basic_Surface shader, connected to above blended values -->
  <basic_surface name="bsrf1" type="surfaceshader">
    <input name="diff_albedo" type="color3" nodename="mix_diff2"/>
    <input name="spec_color" type="color3" nodename="mix_spec2"/>
    <input name="roughness" type="float" nodename="mix_rough2"/>
  </basic_surface>

  <!-- Define the output of the shading network nodegraph -->
  <output name="out" type="surfaceshader" nodename="bsrf1"/>
</nodegraph>

<!-- Sample blended material using the above network shader -->
<material name="blendedmtl">
```



```
    <shaderref name="sr4" node="triblendsrf">
      <bindparam name="paintmaskfile" type="filename" value="paint_mask.tif"/>
      <bindparam name="rustmaskfile" type="filename" value="rust_mask.tif"/>
    </shaderref>
  </material>
</materialx>
```

Example 3: A material using post-shader compositing to blend the outputs of two surface shaders. A nodegraph containing image reading and processing nodes, two shader-semantic nodes and a blending operation is defined, then turned into a single shader which is then referenced by a material; the material uses tokens to define texture filenames, and a standalone texture image read bound to the mix amount shader input.

```
<materialx>
  <!-- Define external "alSurface" shader -->
  <nodedef name="ND_alsurface_srf" node="alSurface">
    <input name="diffuseColor" type="color3" value="0.2,0.2,0.2"/>
    <input name="specular1Color" type="color3" value="1,1,1"/>
    <input name="specular1Roughness" type="float" value="0.3"/>
    <output name="out" type="surfaceshader"/>
  </nodedef>

  <nodedef name="ND_twolayersurface_surface"
           node="twoLayerSurface">
    <input name="diffmult1" type="color3" value="1,1,1"/>
    <token name="color1" type="string" value="color"/>
    <token name="spec1" type="string" value="spec"/>
    <parameter name="roughness1" type="float" value="0.5"/>
    <input name="diffmult2" type="color3" value="0.1,0.1,0.1"/>
    <token name="color2" type="string" value="coloralt"/>
    <token name="spec2" type="string" value="specalt"/>
    <parameter name="roughness2" type="float" value="0.5"/>
    <input name="mixamt" type="float" value="0"/>
    <output name="out" type="surfaceshader"/>
  </nodedef>
  <nodegraph name="NG_twolayersurface_srf" nodedef="ND_twolayersurface_srf">
    <image name="i_diff1" type="color3">
      <parameter name="file" type="filename"
                 value="txt/[color1]/[color1].<UDIM>.tif"/>
    </image>
    <multiply name="mult1" type="color3">
      <input name="in1" type="color3" nodename="i_diff1"/>
      <input name="in2" type="color3" interfacename="diffmult1"/>
    </multiply>
    <image name="i_diff2" type="color3">
      <parameter name="file" type="filename"
                 value="txt/[color2]/[color2].<UDIM>.tif"/>
    </image>
    <multiply name="mult2" type="color3">
      <input name="in1" type="color3" nodename="i_diff2"/>
      <input name="in2" type="color3" interfacename="diffmult2"/>
    </multiply>
    <image name="i_spec1" type="color3">
      <parameter name="file" type="filename"
```



```
                value="txt/[spec1]/[spec1].<UDIM>.tif"/>
    </image>
    <image name="i_spec2" type="color3">
      <parameter name="file" type="filename"
                value="txt/[spec2]/[spec2].<UDIM>.tif"/>
    </image>

    <alSurface name="als1" type="surfaceshader">
      <input name="diffuseColor" type="color3" nodename="mult1"/>
      <input name="specular1Color" type="color3" nodename="i_spec1"/>
      <input name="specular1Roughness" type="float" interfacename="roughness1"/>
    </alSurface>
    <alSurface name="als2" type="surfaceshader">
      <input name="diffuseColor" type="color3" nodename="mult2"/>
      <input name="specular1Color" type="color3" nodename="i_spec2"/>
      <input 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="mix" type="float" interfacename="mixamt"/>
    </mix>
    <output name="out" type="surfaceshader" nodename="srfmix"/>
  </nodegraph>

  <image name="i_mixamt" type="float">
    <parameter name="file" type="filename" value="txt/rustmix/rustmix.<UDIM>.tif"/>
  </image>
  <output name="o_mixamt" type="float" nodename="i_mixamt"/>

  <material name="mblended1">
    <shaderref name="sr6" node="twoLayerSurface">
      <bindtoken name="color1" type="string" value="basecolor"/>
      <bindtoken name="spec1" type="string" value="basespec"/>
      <bindparam name="roughness1" type="float" value="0.34"/>
      <bindinput name="diffmult2" type="color3" value="0.8,0.82,0.79"/>
      <bindtoken name="color2" type="string" value="rustcolor"/>
      <bindtoken name="spec2" type="string" value="rustspec"/>
      <bindparam name="roughness2" type="float" value="0.6"/>
      <bindinput name="mixamt" type="float" output="o_mixamt"/>
    </shaderref>
  </material>
</materialx>
```


Geometry Info Elements

Geometry Info ("geominfo") elements are used to define sets of named attributes with constant values, and to associate them with specific external geometries.

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 texture name string would be a separate <token> within the <geominfo>. These 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 and/or token definitions, and associates them and their values with all geometries listed in the `geom` or `collection` parameter of the <geominfo>:

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

Note that no two <geominfo>s may define values for the same geometry attribute or token for the same geometry, whether the geometry is specified directly, matched via a geometry name expression, or contained within a specified collection.

Attributes for GeomInfo elements:

- `name` (string, required): the unique name of the GeomInfo element
- `geom` (geomnamearray, optional): the list of geometries and/or geometry name expressions that the GeomInfo is to apply to
- `collection` (string, optional): the name of a geometric collection

Either a `geom` or a `collection` may be specified, but not both.

GeomProp Elements

GeomProp elements can be used to associate specific uniform values of a specified geometric property with specific geometries; see the **Geometric Properties** section above. This could include application-specific metadata, attributes passed from a lighting package to a renderer, or other geometry-specific data.

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

GeomProp elements have the following attributes:

- `name` (string, required): the name of the geometric property to define

- `type` (string, required): the data type of the given property
- `value` (any MaterialX type, required): the value to assign to the given property.

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

```
<geompropdef name="surfid" type="integer"/>
<geominfo name="gi1" geom="/a/g1">
  <geomprop name="surfid" type="integer" value="15"/>
</geominfo>
```

GeomProp values can be accessed from a nodegraph using a `<geompropvalue>` node:

```
<geompropvalue name="srfidval1" type="integer" geomprop="surfid" default="0">
```

Token Elements

Token elements are used within <geominfo> elements to define constant string values which are associated with specific geometries. These values can be substituted into filenames within image nodes; see the **Image Filename Substitutions** section above for details:

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

The "value" can be any MaterialX type, but since tokens are used in image filename substitutions, string and integer values are recommended.

Token elements have the following attributes:

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

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

```
<geominfo name="gi1" geom="/a/g1">
  <token name="txtid" type="string" value="Lengine"/>
</geominfo>
```

and then reference that token's value in an image filename:

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

The <txtid> in the file name would be replaced by whatever value the txtid token had for each geometry.

TokenDefault Elements

TokenDefault elements define the default value for a specified geometry token name; this default value will be used in a filename string substitution if an explicit token value is not defined for the current geometry. Since TokenDefault does not apply to any geometry in particular, it must be used outside of a

<geominfo> element.

```
<tokendefault name="diffmap" type="string" value="color1"/>
```

Reserved GeomProp Names

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

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

<geominfo name="gi5" geom="/a/g4">
  <geomprop name="uvtileset" type="stringarray" value="u2_v1,u2_v2"/>
</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
`twosided`	boolean	false
`matte`	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, variant, visibility and/or propertyset assignment declarations:

```
  <look name="lookname" [inherit="looktoinheritfrom"]>
    ...materialassign, variantassign, visibilityassign, property/propertysetassign
declarations...
  </look>
```

Looks can inherit the assignments from another look by including an `inherit` attribute. 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 "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.

A number of looks can be grouped together into a **LookGroup**, e.g. to indicate which looks are defined for a particular asset:

```
  <lookgroup name="lookgroupname" looks="look1[,look2[,look3...]]"
[active="lookname"]/>
```

where *`lookgroupname`* is the name of the lookgroup being defined, *`look1`*/*`look2`*/etc. are the names of <look> or <lookgroup> elements to be contained in the lookgroup (a lookgroup name would resolve to the set of looks recursively contained in that lookgroup), and `active` (if specified) specifies the name of one of the looks defined in `looks` which is deemed by the application to be the current active look. A look can be contained in any number of lookgroups.

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

Assignment Elements

Various types of assignment elements are used within looks to assign materials, categorized visibility and properties to specific geometries, or variants to materials.

For elements which make assignments to geometries, 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 effectively 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 (either a `geom` or a `collection` may be specified, but not both).

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

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.

VariantAssign Elements

VariantAssign elements are used within a <materialassign> or a <look> to apply the values defined in one variant of a variantset to one assigned material, or to all applicable materials in a look.

```
<look name="look1">
  <variantassign name="va1" variantset="varset1" variant="var1"/>
  <materialassign name="ma1" material="material1" geom="...">
    <variantassign name="va2" variantset="varset2" variant="var2"/>
  </materialassign>
  <materialassign name="ma2" material="material2" geom="..."/>
  ...
</look>
```

VariantAssign elements have the following attributes:

- `name` (string, required): the unique name of the VariantAssign element
- `variantset` (string, required): the name of the variantset to apply the variant from
- `variant` (string, required): the name of the variant within `variantset` to use

In the above example, the parameter/input/token value bindings defined within variant "var1" will be applied to matching-named parameters/inputs/tokens found in either "material1" or "material2", while bindings defined within variant "var2" will only be applied to matching-named bindings in "material1". VariantAssigns are applied in the order specified within a scope, with those within a <materialassign> taking precedence over those which are direct children of the <look>.

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 geometry name 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 but not both; similarly, one cannot define both a `viewergeom` and a `viewercollection`.

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 `viewergeom`s 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 `vistype`s are predefined by MaterialX; applications are free to define additional `vistype`s:

Vistype	Description
camera	camera or "primary" ray visibility
illumination	geom or collection is illuminated by the viewergeom light(s)
shadow	geom or collection casts shadows from the viewergeom light(s)
secondary	indirect/bounce ray visibility of geom or 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` or `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", while geometry "/c" would not be visible to *any* secondary rays:

```
<visibility name="v2" viewergeom="/a" geom="/b" vistype="secondary"
    visible="false"/>
<visibility name="v3" geom="/c" 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"]/>
```

Either a `geom` or a `collection` may be specified, but not both. Multiple property/propertyset assignments can be made to the same geometry or collection, as long as no conflicting assignment is 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" includegeom="/a/g1,/a/g2,/a/g5"/>
  <collection name="c_metal" includegeom="/a/g3,/a/g4"/>
  <collection name="c_lamphouse" includegeom="/a/lamp1/housing*Mesh"/>
  <collection name="c_setgeom" includegeom="/b"/>
  <nodedef name="ND_disklgt_lgt" node="disk_lgt">
    <parameter name="emissionmap" type="filename" value=""/>
    <parameter name="gain" type="float" value="2000.0"/>
    <output name="out" type="lightshader"/>
  </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>
  <lookgroup name="sampleassetlooks" looks="lookA,lookB"/>
</materialx>
```

