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

Doug Smythe - smythe@ilm.com
Jonathan Stone - jstone@lucasfilm.com
July 20, 2016

Statement of Problem

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 to various inputs of shaders through a tree of image processing and blending operators.

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

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

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

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

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

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

Requirements

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

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

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

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

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:

- **<opgraph>** for defining data-processing network operator graphs
- **<geominfo>** for declaring and defining uniform geometric attributes that may be referenced from operator graphs
- **<shader>** for declarations of shader programs, including the sets of inputs and outputs that they support
- **<material>** for defining material instances, which assign specific values and opgraph output connections to shader inputs
- **<look>** for defining object looks, which assign materials and other properties to geometries

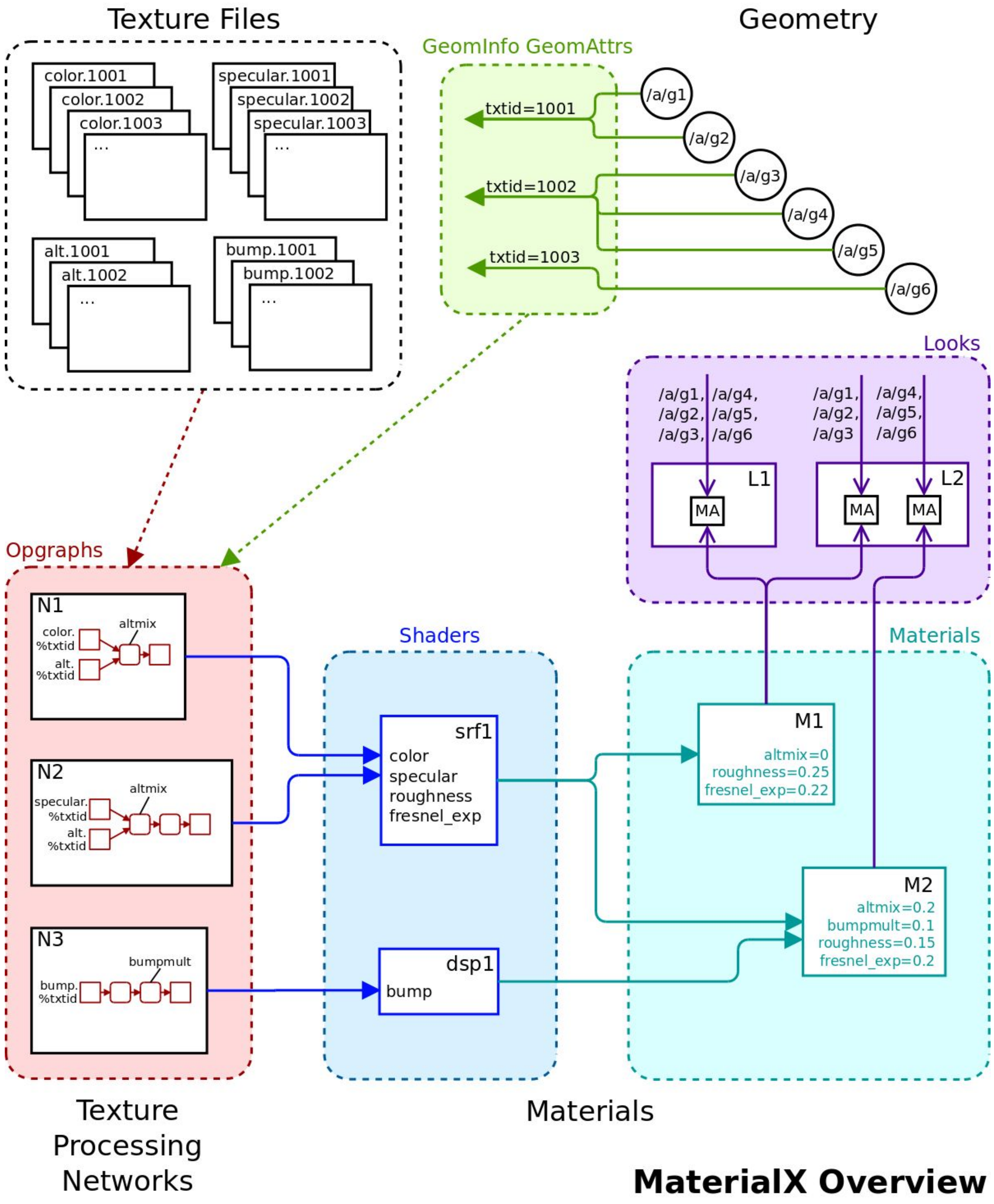
An MTLX file is a standard XML file that represents a MaterialX document, with XML elements and attributes used to represent the corresponding MaterialX elements and attributes. MTLX files may be fully self-contained, or split across several files to encourage sharing and reuse of components.

MaterialX Overview Diagram

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

Flow of information generally proceeds counterclockwise through the diagram. The green "GeomInfo GeomAttrs" box shows how <geominfo> elements can define named string attributes associated with geometries. The red "Opgraphs" box defines a number of texture processing networks, which generally determine which input texture images to read by substituting geominfo strings for each geometry into the specified portion of the image file name. The blue "Shaders" and cyan "Materials" boxes show that shaders connect to opgraph outputs, and materials reference shaders as well as define values for certain public shader or opgraph parameters. The violet "Looks" box shows how looks connect geometry to materials via MaterialAssigns ("MA" in the diagram).

The example diagram defines two looks: L1 and L2. L1 uses materials M1 (assigned to geometry /a/g1 through /a/g6), while L2 uses materials M1 (to /a/g1, /a/g2 and /a/g3) and M2 (to /a/g4, /a/g5 and /a/g6). Both materials reference shader "srf1", but M2 also references shader "dsp1", and each of the materials read from opgraphs N1, N2 and N3, but set different overriding values for the public opgraph parameters "altmix" and (for M2) "bumpmult" as well as different value bindings for srf1's "roughness" and "fresnel_exp" parameters).


Texture Files
color.1001
color.1002
color.1003
...
specular.1001
specular.1002
specular.1003
...
alt.1001
alt.1002
...
bump.1001
bump.1002
...
Geometry
GeomInfo GeomAttrs
txtid=1001
txtid=1002
txtid=1003
/a/g1
/a/g2
/a/g3
/a/g4
/a/g5
/a/g6
Looks
/a/g1, /a/g2, /a/g3, /a/g4, /a/g5, /a/g6
L1
MA
/a/g1, /a/g2, /a/g3
/a/g4, /a/g5, /a/g6
L2
MA
MA
Opgraphs
N1
color. %txtid
alt. %txtid
altmix
N2
specular. %txtid
alt. %txtid
altmix
N3
bump. %txtid
bumpmult
Shaders
srf1
color
specular
roughness
fresnel_exp
dsp1
bump
Materials
M1
altmix=0
roughness=0.25
fresnel_exp=0.22
M2
altmix=0.2
bumpmult=0.1
roughness=0.15
fresnel_exp=0.2
Texture Processing Networks
Materials
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. This concept corresponds closely with the notion of an element in XML.

An **Attribute** is a named property of a MaterialX element. This concept corresponds closely with the notion of an attribute in XML.

An **Opgraph** is a directed acyclic graph of data-processing **Nodes**, each of which possesses zero or more inputs and one or more outputs. This specification defines a set of standardized nodes with precise definitions, and also supports custom nodes for application-specific uses. A **Custom Node** is a user-authored node, whose input and output types are defined with a **NodeDef**.

A **Shader** is an externally-defined shading program, written in a common shading language such as OSL, RSL, or GLSL. The various inputs of a Shader may be declared as **Parameters**, which can hold only constant values; **Inputs**, which are spatially-varying and connectable to Opgraph outputs; or **Coshaders**, which are spatially-varying and connectable to Shader outputs.

An **Application** refers to a Digital Content Creation tool or a renderer; Custom Nodes and Shaders are often authored for specific Applications.

A **Material** is a container for Shader references, with capabilities for binding constant and spatially-varying data to the Shader Parameters and Inputs, and for overriding the values of Public Parameters defined by the Shaders or connected Opgraph Nodes.

An **AOV** (short for Arbitrary Output Variable) is a particular output of a shader, which is used to pass additional color, float, vector or other typed data from one shader to another. AOVs are typically output as additional layers in a multichannel image file, and can describe things like post-illumination color values for one lighting component, or a quantity calculated in the shader that might be useful in compositing.

A **Stream** refers to a flow of image or value data from one Node to another. A Stream generally consists of 1, 2, 3 or 4 channels of floating-point data, but could be data of any single defined 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 naming of layers or multiple layers within a file should be treated as if that layer was named "rgba".

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

A **Public Parameter** is a Parameter of an Opgraph Node or a Shader tagged with a "publicname" attribute, allowing a Material to override it with a new value.

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

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

MaterialX Names

All elements in MaterialX (opgraphs, 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 at the same scope (i.e. elements with the same parent) may share a name. Some element types (e.g. <shaderref>) serve the role of referencing an element at a different scope, and in this situation the referencing element will share a `name` attribute with the element it references.

For maximum compatibility, it is recommended that element names consist of only upper- and lower-case letters, numbers and underscores ("_"). Element names **cannot** include spaces or any of the following symbols:

```
    ! @ # $ % ^ & * ? ( ) [ ] { } < > \ / . , : ' " |
```

MaterialX Attribute Types

By default, the following attribute/parameter types are allowed in MaterialX:

Base Types:

```
    integer, boolean, float, color2, color3, color4, vector2, vector3, vector4,
    matrix, string, filename, opgraphnode, opgraphname, shadernode
```

Array Types:

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

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)

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

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

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

Coordinate values should be separated by commas (with or without whitespace), within quotes:
```
    vector2value = "0.234,0.885"
    vector3value = "-0.13,12.883,91.7"
    vector4value = "-0.13,12.883,91.7,1.0"
```

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

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

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

Opgraphnode, **Opgraphname** and **Shadernode**: these types are just strings within double-quotes, but are used to refer to opgraph or shader elements for the purpose of connecting them with other elements. An **opgraphnode** refers to the value of an opgraph node's "name" attribute and is used to specify nodes for input connections, while an **opgraphname** refers to the name of a top-level opgraph element (so "opgraphnode" is the name of a node, and "opgraphname" is the name of an opgraph). Similarly, a

shadernode refers to the "name" attribute of a shader element.

Matrix: sixteen float values separated by commas (with or without whitespace), within double quotes. Matrices in MaterialX are intended to be 3D transformation matrices and are thus always exactly 4x4 in size, specified in row-major order:

```
    matrixvalue = "1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1"
```

Integerarray, **floatarray**, **color2array**, **color3array**, **color4array**, **vector2array**, **vector3array**, **vector4array**, **stringarray**: any number of values of the same base type, separated by commas (with or without whitespace), within quotes; arrays of color2's, color3's, color4's, vector2's, vector3's or vector4's are simply a 1D list of channel values in order, e.g. "r0 g0 b0 r1 g1 b1 r2 g2 b2". To include a comma within one value of a stringarray, precede it with a '\'. MaterialX does not support 2D or nested arrays.

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

MaterialX does not have any way to concatenate arrays or do any vector, array or matrix operations. These types exist only as a way to set parameter values of those types for material shaders.

Additional types can be defined using <typedef> elements; see the Custom Types section for details.

MTLX File Format Definition

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

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

That is, a standard XML declaration line followed by a <materialx> root element, which contains any number of MaterialX elements and sub-elements.

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

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

Attributes for the root <materialx> element:

- `version` (string, required): a string containing the version number of the MaterialX specification that this document conforms to, specified as a major and minor number separated by a dot. The MaterialX library automatically upgrades older-versioned documents to the current MaterialX version at load time.

- `require` (string, optional): a comma-separated list of implementation-dependent capabilities required by the setup contained within the <materialx> element. See the section **Implementation Compatibility Checking** below for a list of capabilities and further details.
- `cms` (string, optional): the name of the active Color Management System (CMS) : it is the responsibility of the implementation to route any color conversion through the correct CMS. Default is no color management. See the section **Color Spaces and Color Management Systems** below for further details.
- `cmsconfig` (filename, optional): the URI of a configuration file for the active CMS. This file is expected to provide the names of color spaces that may be referenced from the document, along with the transforms between these color spaces. If the active color management system is OCIO, then this attribute defaults to the provided OCIO configuration "mx_config.ocio".
- `colorspace` (string, optional): the name of the "working color space" for this <materialx> element: this is the default color space for all image inputs and all color values, is the space into which all images and values will be converted to if it is declared in a different color space, and the color space in which all color computations will be performed. Note that if no cms and/or working color space is defined, then there will effectively be no color space conversion performed at any point. The default is whatever space the active cmsconfig defines as the "scene_linear" role.

Implementation Compatibility Checking

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

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

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

customtype	defines and uses custom types
customnode	defines and uses custom opgraph nodes
geomops	uses opgraph nodes requiring access to local geometric features such as surface position, normal, and tangent
globalops	uses opgraph nodes requiring access to non-local geometric features
matopgraph	uses materials with an opgraph connected to a shader input
matshadergraph	uses materials with a shader output connected to another

	shader's coshader input
matvarvalue	uses materialvar substitution in material parameter/input values
multiassign	allows multiple materials with non-overlapping types assigned to same geometry.
multilayer	reads individual layers from a multilayer image file
multioutput	defines or uses custom opgraph nodes with multiple outputs
override	uses overrides on publicly-exposed parameters and/or inputs
shadergraph	uses shaders whose <implementation> is defined by an opgraph.

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

Color Spaces and Color Management Systems

MaterialX supports the use of color management systems to associate the RGB components of colors with specific color spaces. This allows MaterialX setups to transform colors in images and parameters from their native color space into a specified working color space upon ingest, and back to a specified output color space upon image output. All computations within opgraphs and shaders are performed in the defined working color space. It is generally presumed that the working color space will be linear (as opposed to log, sRGB, or other non-linear encoding), but 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, there will be no automatic color transformations performed: all values are 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 includes a standard OpenColorIO configuration file "mx_config.ocio", which supports a superset of the color spaces defined in version 1.0 of the Academy Color Encoding System (http://www.oscars.org/science-technology/sci-tech-projects/aces). This cmsconfig is assigned by default to documents that enable OpenColorIO color management, but MaterialX documents can specify a custom configuration using the `cmsconfig` attribute of the <materialx> element:

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

The working color space of a MaterialX document is defined by the `colorspace` attribute of its root <materialx> element,.. For the OpenColorIO CMS, `colorspace` defaults to whatever space is assigned to the "scene_linear" role; other CMS's may provide a similar default. For maximum portability and clarity, it is recommended that all <materialx> elements define a specific `colorspace` if they wish to use a color-managed workflow.

```
  <materialx cms="ocio" colorspace="lin_rec709">
```

The color space of individual color images and values may be defined by setting a different `colorspace` at any enclosing scope. Color images and values in spaces other than the working space are expected to be transformed by the implementation into the working space before computations are performed. In the example below, an image file has been defined in the "srgb_texture" color space, while its default value has been defined in "lin_p3dci"; both should be transformed to the document's working color space before being applied to any computations.

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

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` or `regex` 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` or `regex` attribute, it is presumed that the host application can resolve the location of the geometry definition.

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

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

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

E.g. an initial "/" followed by one or more hierarchy level strings separated by "/"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 used to define regex 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 regex matching.

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

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

and for the child partitions, the syntax would be:

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

Geometry and File Prefixes

As a shorthand convenience, MaterialX allows the specification of a `geomprefix` XML attribute that will be prepended to the value of each `geom` attribute (e.g. in `<geominfo>`, `<collectionadd>`/`<collectionremove>`, `<light>`, `<materialassign>`, or `<lightillum>`/`<lightshadow>` elements) specified within the scope of the element defining the `geomprefix`. If the value of a `geom` attribute contains several comma-separated values, the `geomprefix` is prepended to each of the separate geometry names. Geomprefixes are not prepended to `regex` values. Since the values of the prefix and the geometry are string-concatenated, the value of a `geomprefix` should generally end with a "/". Geomprefix is commonly used to split off leading portions of geometry paths common to all geometry names, perhaps including package-specific conventions or asset name.

So the following MTLX file snippets are all equivalent:

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

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

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

MaterialX also allows the specification of a `fileprefix` attribute whose value will be prepended to the value of any parameter of type "filename" (e.g. in `<image>` or `<output>` opgraph nodes, or any shader parameter of type "filename") specified within the scope of the element defining the `fileprefix`. Since the values of the prefix and the filename are string-concatenated, the value of a `fileprefix` should generally end with a "/". Fileprefixes are frequently used to split off common path components for asset directories.

So the following snippets are also equivalent:

```
<materialx>
  <opgraph name="opgraph1">
    <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>
  </opgraph
</materialx>

<materialx>
  <opgraph name="opgraph1" 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>
  </opgraph
</materialx>
```

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

Public Names

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

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

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

Image Filename Substitutions

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

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

	replaced by a 4-digit zero-padded frame number such as "0010".
$CONTAINER	A special string that will be replaced by the name of the image file in which this MaterialX content is contained. This construct can be used in MaterialX content embedded within the metadata or header of an image.

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

Regex Expressions

Certain elements in MaterialX files support geometry specification via regex expressions. The syntax for MaterialX regex largely follows that of UNIX "csh" filename regex.

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 regex expressions within the braces

A "/" will match only exactly a single "/" in a geometry name, e.g. as a boundary for a hierarchy level.
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 regex ends with "//*", the final "*" will only match leaf geometries in the hierarchy. A regex expression of "//*" by itself will match all leaf geometries in an entire scene, while a regex expression of "//*//" will match all geometries at any level, including nested geometries, and a regex expression of "/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 regex expressions using "//*". Regex expressions will only match scene locations representing renderable geometry, not arbitrary intermediate "parent" locations.

Regex expressions can also be further restricted to match only a subset of geometry types by following the regex string with `(type=`*`geomtype`*`)`, where *geomtype* is a comma-separated list of types from the list: "poly", "mesh", "partition", "curve", "volume", and "procedural". For example, regex="/a/b//*foo*(type=mesh)" will match all geometries of type subdivision surface mesh whose name begins with "/a/b/" and contains the string "foo".

Parameter Expressions and Function Curves

It is noted that many packages allow material parameters to have values set by an expression or a function curve. However, since the syntaxes and capabilities of various packages in use vary widely, the MaterialX specification does not currently support parameter expressions or function curve values.

Operator Graph Elements

Operator graph ("opgraph") elements describe an arbitrary acyclic data processing graph applied to one or more source nodes in order to produce input values to a shader for rendering. A MaterialX document can contain multiple opgraph elements, each defining one or more output names. Operator graphs can be shared by several different shaders or even different inputs to the same shader because each material element specifies which opgraph element and output to connect to various shader inputs, and perhaps specifying different values for public opgraph parameters.

Opgraph Definition

An **<opgraph>** element consists of at least two node elements (minimally, an image or source node and an output node) contained within an <opgraph> element:

```
<opgraph name="graphname">
  ...optional parameter declarations...
  ...node elements...
</opgraph>
```

Optionally, an opgraph may also declare a set of parameters, allowing it to become the definition of a custom opgraph node. Please see the Custom Opgraph Nodes section for details.

Node elements have the form:

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

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

Node elements contain zero or more <parameter> elements defining the name, type, value and perhaps publicname of each node parameter. Some of these parameters describe the input connections to the nodes, while others provide parameter values for the source or operator. Note that parameters of type "opgraphnode" cannot be given publicnames, as opgraph connections are not overridable.

Standard MaterialX opgraph nodes have zero or more inputs and exactly one output. The names of the inputs are defined by the node type, while the output is unnamed: a node's output is referred to by the `name` of the node itself. Custom opgraph nodes can have multiple outputs; please see the Custom Opgraph Nodes section for details.

MaterialX defines a number of standard node types and standard operators which all implementations should support as described (with the possible exception of implementation-dependent capabilities noted with "require" attributes). Specific implementations can also define their own opgraph node types by writing application-specific code for them (e.g. in OSL), and can even define custom types to pass arbitrary types of data from one custom opgraph node to another. Please see the **Custom Opgraph Nodes** section for notes and implementation details.

Standard Node Types

The node types listed below are used to define and select inputs and outputs for the network. Each must declare the MaterialX type of its output, which is most commonly a float, colorN, or vectorN.

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

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

Standard Texture node types:

- **`image`**: samples data from a single image, or from a layer within a multi-layer image. When used in the context of rendering a geometry, the image is mapped onto the geometry based on geometry UV coordinates. Parameters:
  - `file` (filename, required): the URI of an image file. The filename can include one or more substitutions to change the file name (including frame number) that is accessed, as described in the **Image Filename Substitutions** section above. [REQ="multilayer" if multi-layer input file]
  - `layer` (string, optional): the name of the layer to extract from a multi-layer input file. If no value for `layer` is provided and the input file has multiple layers, then the "default" layer will be used, or "rgba" if there is no "default" layer. Note: the number of channels defined by the `type` of the `<image>` must match the number of channels in the named layer.
  - `default` (float or colorN or vectorN, optional): a default value to use if the `file` reference can not be resolved (e.g. if a %geomattr or @materialvar is included in the filename but no substitution value or default is defined, or if the resolved file URI cannot be read), or if the specified `layer` does not exist in the file. The `default` value must be the same type as the `<image>` element itself. If `default` is not defined, the default color value will be 0.0 in all channels.
  - `uaddressmode` (string, optional): determines how U coordinates outside the 0-1 range are processed before sampling the image; see below.
  - `vaddressmode` (string, optional): determines how V coordinates outside the 0-1 range are processed before sampling the image; see below.
  - `uvset` (integer, optional): the index of the UV set to use in image sampling. Default is 0.
  - `uvscale` (vector2, optional): a scaling factor about the (0,0) corner (e.g. a straight multiply) applied to texture coordinates along the U and V axes before image sampling. The default value is (1.0, 1.0); to have an incoming texture tiled or repeated twice, set uvscale to "0.5,

0.5".
    - `uvrotate` (float, optional): a rotation in degrees about the center (0.5,0.5) applied to texture coordinates before image sampling. The default value is 0.0.
    - `uvoffset` (vector2, optional): an offset applied to texture coordinates along the U and V axes before image sampling. The default value is (0.0, 0.0).
- **`triplanarprojection`**: samples data from three images (or layers within multi-layer images), and projects a tiled representation of the images along each of the three respective coordinate axes, computing a weighted blend of the three samples using the geometric normal. [REQ="geomops"]. Parameters:
    - `filex` (filename, required): the URI of an image file to be projected along the x-axis.
    - `filey` (filename, required): the URI of an image file to be projected along the y-axis.
    - `filez` (filename, required): the URI of an image file to be projected along the z-axis.
    - `layerx` (string, optional): the name of the layer to extract from a multi-layer input file for the x-axis projection. If no value for `layerx` is provided and the input file has multiple layers, then the "default" layer will be used, or "rgba" if there is no "default" layer. Note: the number of channels defined by the `type` of the `<image>` must match the number of channels in the named layer.
    - `layery` (string, optional): the name of the layer to extract from a multi-layer input file for the y-axis projection.
    - `layerz` (string, optional): the name of the layer to extract from a multi-layer input file for the z-axis projection.
    - `default` (float or colorN or vectorN, optional): a default value to use if any `fileX` reference can not be resolved (e.g. if a %geomattr or @materialvar is included in the filename but no substitution value or default is defined, or if the resolved file URI cannot be read) The `default` value must be the same type as the `<triplanarprojection>` element itself. If `default` is not defined, the default color value will be 0.0 in all channels.
    - `position` (opgraphnode, optional): the name of a vector3-type node specifying the 3D position at which the projection is evaluated. Default is to use the current 3D world-space coordinate.
    - `scale` (vector3, optional): a scaling factor applied to the position coordinate before image sampling. Default is (1.0, 1.0, 1.0).
    - `rotation` (vector3, optional): a rotation in degrees applied independently to coordinates in each projection axis before image sampling. Default is (0.0, 0.0, 0.0).
    - `offset` (vector3, optional): an offset applied to the position coordinate before image sampling. Default is (0.0, 0.0, 0.0).

The following values are supported by `uaddressmode` and `vaddressmode` parameters:
- “black”: Texture coordinates outside the 0-1 range return a value with 0.0 in all channels. This setting is the default value for address mode parameters.
- “clamp”: Texture coordinates are clamped to the 0-1 range before sampling the image.
- “wrap”: Texture coordinates outside the 0-1 range are wrapped, effectively being processed by a modulo 1 operation before sampling the image.
- “mirror”: Texture coordinates outside the 0-1 range are wrapped, with every other instance of the wrapped image being mirrored along the given axis

Texture coordinate transforms are applied in the following order: `uvrotate`, `uvoffset`, `uvscale`. Position coordinate transforms are applied in the following order: `rotation`, `offset`, `scale`.

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

- `framerange` (string, optional): A string "*minframe-maxframe*", e.g. "10-99", to specify the range of frames that the image file is allowed to have, usually the range of image files on disk. Default is unbounded.
- `frameoffset` (integer, optional): A number that is added to the current frame number to get the image file frame number. E.g. if `frameoffset` is 25, then processing frame 100 will result in reading frame 125 from the imagefile sequence. Default is no frame offset.
- `frameendaction` (string, optional): What to do when the resolved image frame number is outside the `framerange` range:
    - "black": Return a value of 0 in all channels (default action)
    - "hold": Hold the minframe image for all frames before *minframe* and hold the maxframe image for all frames after *maxframe*
    - "wrap": Wrap the frame number, so after the maxframe it will start again at minframe (and similar before minframe)
    - "bounce": Go back-and-forth through the frame range, so if framerange is "1-10", then processing frame 11 will read image frame 9, processing frame 12 will read image frame 8, etc. (and similar before minframe).

Arbitrary frame number expressions and speed changes are not supported.

Source nodes are used to generate color data programmatically, with their inputs typically being limited to uniform parameters and position coordinate data. Implementations are expected to support the standard sources below, and are free to define any number of custom source nodes as well. If an implementation does not understand a custom source type used in a file, it is expected to output black (0.0) in all channels.

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

Standard Source node types:

- **`constant`**: a constant value. When exposed as a public parameter, this is a way to create a value that can be accessed in multiple places in the opgraph. Parameters:
    - `value` (any MaterialX type, required): the value to output
- **`ramplr`**: a left-to-right linear value ramp. Parameters:
    - `valuel` (float or colorN or vectorN, required): the value at the left (u=0) edge
    - `valuer` (float or colorN or vectorN, required): the value at the right (u=1) edge
- **`ramptb`**: a top-to-bottom linear value ramp. Parameters:
    - `valuet` (float or colorN or vectorN, required): the value at the top (v=1) edge
    - `valueb` (float or colorN or vectorN, required): the value at the bottom (v=0) edge

- **`ramp4`**: a 4-corner bilinear value ramp. Parameters:
  - `valuetl` (float or colorN or vectorN, required): the value at the top-left (u=0,v=1) corner
  - `valuetr` (float or colorN or vectorN, required): the value at the top-right (u=1,v=1) corner
  - `valuebl` (float or colorN or vectorN, required): the value at the bottom-left (u=0,v=0) corner
  - `valuebr` (float or colorN or vectorN, required): the value at the bottom-right (u=1,v=0) corner
- **`splitlr`**: a left-right split matte, split at a specified u value. Parameters:
  - `valuel` (float or colorN or vectorN, required): the value at the left (u=0) edge
  - `valuer` (float or colorN or vectorN, required): the value at the right (u=1) edge
  - `center` (float, 0-1, required): the u-coordinate of the split; all pixels to the left of "center" will be `valuel`, all pixels to the right of "center" will be `valuer` (antialiased, of course).
- **`splittb`**: a top-bottom split matte, split at a specified v value. Parameters:
  - `valuet` (float or colorN or vectorN, required): the value at the top (v=1) edge
  - `valueb` (float or colorN or vectorN, required): the value at the bottom (v=0) edge
  - `center` (float, 0-1, required): the v-coordinate of the split; all pixels above "center" will be `valuet`, all pixels below "center" will be `valueb` (antialiased, of course).
- **`noise2d`**: 2D Perlin noise in 1, 2, 3 or 4 channels. Parameters:
  - `scale` (float, required): scaling factor for the noise function coordinate. The coordinate used for the Perlin noise function is the input `position` coordinate divided by `scale`.
  - `amplitude` (float or vectorN, optional): the center-to-peak amplitude of the noise (peak-to-peak amplitude is 2x this value). Default is 1.0.
  - `pivot` (float, optional): the center value of the output noise; effectively, this value is added to the result after the Perlin noise is multiplied by `amplitude`. Default is 0.0.
  - `position` (opgraphnode, optional): the name of a vector2-type node specifying the 2D position at which the noise is evaluated. Default is to use the current u,v coordinate.
- **`noise3d`**: 3D Perlin noise in 1, 2, 3 or 4 channels [REQ="geomops"]. Parameters:
  - `amplitude` (float or vectorN, optional): the center-to-peak amplitude of the noise (peak-to-peak amplitude is 2x this value). Default is 1.0.
  - `pivot` (float, optional): the center value of the output noise; effectively, this value is added to the result after the Perlin noise is multiplied by `amplitude`. Default is 0.0.
  - `position` (opgraphnode, optional): the name of a vector3-type node specifying the 3D position at which the noise is evaluated. Default is to use the current 3D object-space coordinate.
- **`cellnoise2d`**: 2D cellular noise, 1 channel (type float). Parameters:
  - `scale` (float, required): scaling factor for the noise function coordinate. The coordinate used for the cell noise function is the input `position` coordinate divided by `size.`
  - `position` (opgraphnode, optional): the name of a vector2-type node specifying the 2D position at which the noise is evaluated. Default is to use the current u,v coordinate.
- **`cellnoise3d`**: 3D cellular noise, 1 channel (type float) [REQ="geomops"]. Parameters:
  - `position` (opgraphnode, optional): the name of a vector3-type node specifying the 3D position at which the noise is evaluated. Default is to use the current 3D object-space coordinate.

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 method appropriate to the specific application.

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

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

Geometric nodes are used to reference local geometric properties from within an opgraph:

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

Standard Geometric node types:

- **`position`**: the coordinates associated with the currently-processed data, as defined in a specific coordinate space [REQ="geomops"]. This node must be of type vector3, and takes the following parameter:
    - `space` (string, optional): the name of the coordinate space in which the position is defined. See the section below for details.
- **`normal`**: the geometric normal associated with the currently-processed data, as defined in a specific coordinate space [REQ="geomops"]. This node must be of type vector3, and it takes the following parameter:
    - `space` (string, optional): the name of the coordinate space in which the normal vector is defined. See the section below for details.
- **`tangent`**: the geometric tangent vector associated with the currently-processed data, as defined in a specific coordinate space [REQ="geomops"]. This node must be of type vector3, and it takes the following parameter:
    - `space` (string, optional): the name of the coordinate space in which the tangent vector is defined. See the section below for details.
- **`bitangent`**: the geometric bitangent vector associated with the currently-processed data, as defined in a specific coordinate space [REQ="geomops"]. This node must be of type vector3, and it takes the following parameter:
    - `space` (string, optional): the name of the coordinate space in which the bitangent vector is defined. See the section below for details.
- **`texcoord`**: the 2D or 3D texture coordinates associated with the currently-processed data. This node must be of type vector2 or vector3, and it takes the following parameters:

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

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

Application nodes are used to reference application-defined properties within an opgraph:

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

Standard Application node types:
- **`frame`**: the current frame number as defined by the host environment. This node must be of type float, and it takes no parameters.
- **`time`**: the current time in seconds, as defined by the host environment. This node must be of type float, and it takes no parameters.

Output nodes are used to specify an output of a graph, e.g. to connect to a shader <input>; there can be multiple outputs in a graph. Output nodes may also be used to write pixel data to a file.

```
<output name="albedo" type="color3">
  <parameter name="in" type="opgraphnode" value="n9"/>
  <parameter name="file" type="filename" value="colorout.%txtid.tif"/>
</output>
```

Parameters for Output nodes:
- `in` (opgraphnode, required): the name of the node connected to the Output node's "in" input.
- `width` (int, optional): The width in pixels of the output image.
- `height` (int, optional): The height in pixels of the output image.
- `datatype` (string, optional): the datatype of each channel in the output image, one of "uint8", "int16", "uint16", "int32", "uint32", "float16", "float32", or "float64", where "uint" means unsigned integer, "int" means signed integer", and 8/16/32/64 refer to the number of bits. When outputting int and uint color channels, the floating-point value of 1.0 is mapped to the maximum

representable value for that integer type, e.g. 255 for uint8, 32767 for int16, and 65535 for uint16.

- `file` (filename, optional): the URI of an external image file that can be used to store or cache the resulting image from the opgraph output. Note that `file` file names can include string substitutions, just like input file names; see the **Image Filename Substitutions** section above for details. The file parameter or any enclosing scope can optionally define a `colorspace` attribute value to set the color space into which the output cache file data should be written. The extension of the file name determines the file format of the image, e.g. ".exr", ".tif", or ".jpg". If `file` is specified, then `width` and `height` must also be specified. If `file` is not specified, this output will not be written to disk, but can be connected to a subgraph in other opgraphs, or to shaders.
- `cache` (boolean, optional): If true, then the contents of the output `file` will be used as a cache of the operations feeding into this <output> node; see below. Defaults to `false`, so output image data would only be written and never read back in.

Specific implementations can define values for additional attributes, such as `compression`.

Note that <output> nodes can be connected to the inputs of other nodes: they do not need to be the last nodes in a chain of opgraph nodes. This is useful if you want to "tap" into a partial opgraph computation. If the `file` parameter is used, <output> nodes can be used to store the computation up to that point in the opgraph, and if the <output> node is connected to the input of another node and the `cache` parameter is set to `true`, then the contents of that file will be read and used if the file is found and it passes a checksum and file timestamp test. If a `colorspace` is specified, note that the color space transform will only be applied to the output file data and will not be passed on to any connected nodes. Conversely, if image data is read back in from an output `file` file, the appropriate inverse color space transform will be applied to bring the image data back to the working color space.

Standard Operators

Operators are opgraph nodes that process one or more input streams to form an output. Like other opgraph nodes, each operator must define its output type, determining the number of color channels the operator works on and the types of all connected input nodes.

```
<over name="n11" type="color4">
  <parameter name="fg" type="opgraphnode" value="infg"/>
  <parameter name="bg" type="opgraphnode" value="inbg"/>
</over>
<hueshift name="n12" type="color3">
  <parameter name="in" type="opgraphnode" value="n4"/>
  <parameter name="amount" type="float" value="0.1"/>
</hueshift>
<contrast name="n13" type="float">
  <parameter name="in" type="opgraphnode" value="n7"/>
  <parameter name="amount" type="float" value="0.2"/>
  <parameter name="pivot" type="float" value="0.46"/>
</contrast>
```

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

The rest of this section details the list of all required operator types, their parameters, and implementation notes.

Math Operators

Math operators have one input named "in", and apply the same specified amount to every pixel; to combine two input color streams, use a compositing operator such as "plus", "difference", "mult" or "div" instead. The math operation is done channel-by-channel on the incoming color, and the number of channels in "amount" must either exactly match the number of incoming channels, or be a (single channel) float value: in the latter case, the float "amount" value is used for all channels.

- **`add`**: add a constant "amount" to the incoming float/color/vector. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `amount` (float or colorN or vectorN, required): the value to add
  - `channels` (string, optional): the channels to operate on; see the Standard Node Attributes section for details. Default is "all channels".
- **`subtract`**: subtract a constant "amount" from the incoming float/color/vector, outputting "in-amount". Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `amount` (float or colorN or vectorN, required): the value to subtract from the input value
  - `channels` (string, optional): the channels to operate on. Default is "all channels".
- **`multiply`**: multiply the incoming float/color/vector by the constant "amount". Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `amount` (float or colorN or vectorN, required): the value to multiply by

    - `channels` (string, optional): the channels to operate on. Default is "all channels".
- **`divide`**: divide the incoming float/color/vector by the constant "amount"; dividing a channel value by 0 results in floating-point "NaN". Parameters:
    - `in` (opgraphnode, required): the name of the node to connect to the input
    - `amount` (float or colorN or vectorN, required): the value to divide by
    - `channels` (string, optional): the channels to operate on. Default is "all channels".
- **`invert`**: subtract the incoming float/color/vector from "amount" in all channels, outputting "amount-in". Parameters:
    - `in` (opgraphnode, required): the name of the node to connect to the input
    - `amount` (float or colorN or vectorN, optional): the value to subtract the input value from; default is 1.0 in all channels
    - `channels` (string, optional): the channels to operate on. Default is "all channels".
- **`absval`**: the absolute value of the incoming float/color/vector. Parameters:
    - `in` (opgraphnode, required): the name of the node to connect to the input
    - `channels` (string, optional): the channels to operate on. Default is "all channels".
- **`exponent`** - raise incoming float/color values to the specified exponent, commonly used for "gamma" adjustment. Negative input values will be left unchanged. Parameters:
    - `in` (opgraphnode, required): the name of the node to connect to the input
    - `amount` (float or colorN or vectorN, required): exponent value; output = pow(input, amount)
    - `channels` (string, optional): the channels to operate on. Default is "all channels".
- **`contrast`** - increase or decrease contrast of incoming float/color values using a linear slope multiplier. Parameters:
    - `in` (opgraphnode, required): the name of the node to connect to the input
    - `amount` (float or colorN, required): slope multiplier for contrast adjustment, 0.0 to infinity range. Values greater than 1.0 increase contrast, values between 0.0 and 1.0 reduce contrast
    - `pivot` (float or colorN, optional): center pivot value of contrast adjustment; this is the value that will not change as contrast is adjusted (default value=0.5)
    - `channels` (string, optional): the channels to operate on. Default is "all channels".
- **`clamp`** - clamp incoming values to a specified range of color values. Parameters:
    - `in` (opgraphnode, required): the name of the node to connect to the input
    - `low` (float or colorN or vectorN, optional): clamp low value; any value lower than this will be set to "low" (default value=0)
    - `high` (float or colorN or vectorN, optional): clamp high value; any value higher than this will be set to "high" (default value=1)
    - `channels` (string, optional): the channels to operate on. Default is "all channels".
- **`remap`** - remap incoming values from one range of float/color/vector values to another, optionally applying a gamma correction "in the middle". Input values below `inlow` or above `outhigh` are extrapolated unless `doclamp` is true Parameters:
    - `in` (opgraphnode, required): the name of the node to connect to the input
    - `inlow` (float or colorN or vectorN, optional): low value for input range (default value=0)
    - `inhigh` (float or colorN or vectorN, optional): high value for input range (default value=1)
    - `gamma` (float or colorN or vectorN, optional): exponent applied to input value after first reranging from `inlow`..`inhigh` to 0..1 (default value=1)
    - `outlow` (float or colorN or vectorN, optional): low value for output range (default value=0)
    - `outhigh` (float or colorN or vectorN, optional): high value for output range (default value=1)
    - `doclamp` (boolean, optional): If true, the output is clamped to the range `outlow`..`outhigh`

    (default is to not clamp)
  - `channels` (string, optional): the channels to operate on; see the Standard Node Attributes section for details. Default is "all channels".
- **`normalize`**: outputs the normalized vectorN from the incoming vectorN stream; cannot be used on float or colorN streams. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
- **`magnitude`**: outputs the float magnitude (vector length) of the incoming vectorN stream; cannot be used on float or colorN streams. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input

<u>Color-Correction Operators</u>

Color-correction operators have one input named "in", and apply a specified color manipulation function to color3 or color4 values in the incoming stream. Color-correction operators cannot be applied to values of type float, color2, or vectorN.

- **`hueshift`** - (color3 or color4 only) shift the hue of a color; the alpha channel will be unchanged if present. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `amount` (float, required): amount of hue shift; the `hueshift` operator transforms the incoming color to the CIE XYZ color space, rotates the value about the spectral locus Y axis in this space, and then transforms it back to the working color space. A positive "amount" rotates hue in the "red to green to blue" direction, with amount of 1.0 being the equivalent to a 360 degree (e.g. no-op) rotation.
- **`saturate`** - (color3 or color4 only) adjust the saturation of a color; the alpha channel will be unchanged if present. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `amount` (float, required): multiplier for saturation; the saturate operator transforms the incoming color to the CIE XYZ color space, scales the XZ values in this space relative to the spectral locus Y axis by "amount", and then transforms it back to the working color space. Note that setting amount to 0 will result in an R=G=B gray value equal to the value that luminance (below) returns.
- **`luminance`** - (color3 or color4 only) output a grayscale image containing the luminance of the incoming RGB color in all color channels, computed using the luma coefficients of the active CMS configuration; the alpha channel is left unchanged if present. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
- **`colorspacetransform`** - (color3 or color4 only) transform RGB (alpha unchanged if present) from one CMS color space to another. MaterialX converts color space values of input and output files to/from the working color space automatically, so this operator will not generally be needed, however it is provided in case a setup requires non-standard color space conversion. It is up to the user and/or host application to make sure the "from" and "to" color spaces make sense for actual input and output color values: MaterialX streams do not keep track of what color space values are "currently in". Parameters:
  - `fromcolorspace` (string, optional): the color space to transform from (default=current working color space).
  - `tocolorspace` (string, optional): the color space to transform into (default=current working color space).

Premult/Unpremult Operators

Premult and Unpremult operators have one input named "in" (and for 3-channel variants, a separate "alpha" input), and either apply or unapply the alpha to the float or RGB color.

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

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

For 3-channel (color3) inputs/outputs:

- **`premult`**: For a color3 `in` stream and a float `alpha` stream, multiplies `in` by `alpha`; output is type color3. To do a 1-channel "premult", use `<multiply type="float">`. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `alpha` (opgraphnode, required): the name of the float-type alpha input node
- **`unpremult`**: For a color3 `in` stream and a float `alpha` stream, divides `in` by `alpha`; output is type color3. To do a 1-channel "unpremult", use `<divide type="float">`. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `alpha` (opgraphnode, required): the name of the float-type alpha input node

Compositing Operators

Compositing operators have two inputs named "fg" and "bg", and apply a function to combine them. Compositing operators are split into four groups: Blend operators, Merge operators, Masking operators, and the Mix operator.

Blend operators 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 Operator	Each Channel Output	Supported Types
`average`	(F+B)/2	float, colorN, vectorN
`burn`	1-(1-B)/F	float, colorN
`dodge`	B/(1-F)	float, colorN
`difference`	abs(F-B)	float, colorN, vectorN
`div`	F/B if F>=0 or B>0; 0 if B=0 or F<0 and B<0	float, colorN, vectorN
`max`	max(F,B)	float, colorN, vectorN

`min`	min(F,B)	float, colorN, vectorN
`minus`	F-B	float, colorN, vectorN
`mult`	FB	float, colorN, vectorN
`plus`	F+B	float, colorN, vectorN
`screen`	1-(1-F)(1-B)	float, colorN
`overlay`	2FB if F<0.5; 1-(1-F)(1-B) if F>=0.5	float, colorN
`dotproduct`	F . B	vectorN (output=float)
`crossproduct`	F x B	vector3

Merge operators take two 2-channel (color2) or two 4-channel (color4) inputs and use the built-in alpha channel 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 operators are not available for 1-channel or 3-channel inputs, and cannot be used on vectorN streams**.**

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 operators take one 1-4 channel input "in" plus a separate 1-channel "mask" input and apply the same operator to all channels (if present, the math for alpha is the same as for R or RGB). In the Masking Operator table, "F" refers to any individual channel of the "in" input.

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

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

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

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

The "mix" operator has the following additional parameters:

- `amount` (float, optional): a fixed-percentage blending amount used if a mask image is not provided; `amount` is required if there is no mask input, and if both a mask input and an `amount` are specified the value of `amount` will be ignored.
- `premult` (boolean, optional): defaults to **true**, which will multiply the fg input by the mask. If false, the fg will not be multiplied by the mask, making this useful when the fg image is "matted to black" by the mask image, i.e. already premultiplied by the mask. If no mask is given (and a fixed `amount` is used instead), `premult` has no effect.

Channel Operators

Channel Operators have one input named "in", and add, remove, exchange or reorder individual channels.

- **`reorder`** - reorder the input channels into the defined order in the output without changing the type or number of channels in the stream. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `type` (string, required): the MaterialX type of the output (and input)
  - `channels` (string, required): a string of one, two, three or four characters (one per channel in the output), each of which can be r, g, b, a (for colorN input streams), x, y, z, w, s, t, u, v (for vectorN input streams), 0 or 1. E.g. "rgb1" would output a four-channel (color4) stream, passing through rgb unchanged and setting alpha to solid-white, and "bgr" would output a three-channel stream with red and blue channels swapped. The number and names of characters in `channels` must be the same as the number of channels for the `reorder` node type, e.g. exactly 2 characters within the list r, g, 0, 1 for a `reorder` of type "color2", or 4 characters for a `reorder` of type "color4". There is a `reorder` node of type "float" (taking a single-character channels value of r/x/u/s, 0 or 1), though the usefulness of this variant is questionable.
- **`convert`** - convert a stream from one type to another (e.g. float to color3), adding, copying or removing channels as described in the table below. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `intype` (string, required): the MaterialX type of the input
  - `type` (string, required): the MaterialX type of the output

Table of channel operations for **`convert`**:

`intype` (input)	`type` (output)	Action

`float`	`color2/vector2`, out = "rr"	Red copied to Alpha
`float`	`color3/vector3`, out = "rrr"	Red copied to Green and Blue
`float`	`color4`, out = "rrrr"	Red copied to Green, Blue and Alpha
`float`	`vector4`, out = "rrr1"	Red copied to X,Y,Z; W set to 1
`color2`	`float`, out = "r"	Alpha dropped
`color2`	`vector2`, out = "rg"	Straight channel copy
`color2`	`color3`, out = "rrr"	Red copied to Green and Blue, Alpha dropped
`color2`	`vector3`, out = "rg0"	Red/Green copied, z=0
`color2`	`color4`, out = "rrra"	Red copied to Green and Blue
`color2`	`vector4`, out = "rg01"	Red/Green copied, z=0,w=1
`color3`	`float`, out = "r"	Green and Blue dropped
`color3`	`color2`, out = "r1"	Green and Blue dropped, Alpha=1.0
`color3`	`vector2`, out = "rg"	Red/Green copied, blue dropped
`color3`	`vector3`, out = "rgb"	Straight channel copy
`color3`	`color4/vector4`, out = "rgb1"	Adds Alpha/w=1.0 channel
`color4`	`float`, out = "r"	Green, Blue and Alpha dropped
`color4`	`color2`, out="ra"	Green and Blue dropped
`color4`	`vector2`, out = "rg"	Blue and Alpha dropped
`color4`	`color3/vector3`, out = "rgb"	Alpha dropped
`color4`	`vector4`, out = "rgba"	Straight channel copy
`vector2`	`float`, out = "x"	X copied, Y dropped
`vector2`	`color2`, out = "xy"	Straight channel copy
`vector2`	`color3/vector3`, out = "xy0"	X,Y copied, blue/Z=0
`vector2`	`color4/vector4`, out = "xy01"	X,Y copied, blue/Z=0, alpha/w=1

`vector3`	`float`, out = "x"	Y, Z dropped; for vector magnitude, use the <magnitude> operator
`vector3`	`color2/vector2`, out = "xy"	X,Y copied, Z dropped
`vector3`	`color3`, out = "xyz"	Straight channel copy
`vector3`	`color4/vector4`, out = "xyz1"	X,Y, Z copied, alpha/w=1
`vector4`	`float`, out = "x"	Y, Z, W dropped
`vector4`	`color2/vector2`, out = "xy"	X,Y copied, Z,W dropped
`vector4`	`color3/vector3`, out = "xyz"	W dropped
`vector4`	`color4`, out = "xyzw"	Straight channel copy

Convolution Operators

Convolution operators have one input named "in", and apply a defined convolution function on the input stream.

- **`blur`** - a gaussian-falloff blur. Either `size` or `pixels` must be specified but not both. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `size` (float, optional): the size of the gaussian blur kernel, relative to 0-1 UV space
  - `pixels` (float, optional): the size of the gaussian blur kernel, in absolute pixels.
  - `channels` (string, optional): the channels to operate on; see the Standard Node Attributes section for details. Default is "all channels".
- **`heighttonormal`** - convert a scalar height map to a normal map of type vector3. Parameters:
  - `in` (opgraphnode, required): the name of the node to connect to the input
  - `scale` (float, optional): the scale of normal map deflections relative to the gradient of the height map. Default is 1.0.

Miscellaneous Opgraph Nodes

The following nodes do not have any image processing functionality, but are included as commonly-used "niceties" in interactive 3D packages with UI's.

- **`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. Parameters:
  - `in` (string, required): the name of the node to be connected to the Dot node's "in" input.
  - `note` (string, optional): a text note associated with the dot node; default is no text.
- **`backdrop`** - a no-op with no input; simply a node graph layout UI element used to contain, group, and document a number of nodes. Parameters:
  - `note` (string, optional): a text note associated with the backdrop node; default is no text.

    - `contains` (string, optional): a comma-separated list of node names that the backdrop "contains"; default is to contain no nodes.
    - `width` (float, required): width of the backdrop when drawn in a UI. [See xpos/ypos under Standard Node Attributes for a discussion of UI units]
    - `height` (float, required): height of the backdrop when drawn in a UI.
    - `nodecolor` (color3, optional): the color of the backdrop in the UI; default is to not specify a particular color so the application's default backdrop color would be used.

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 Opgraph Node Parameters

All opgraph node types (both Sources and Operators) support the following UI-related attributes:

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

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.

All opgraph node types (both Sources and Operators) also support the following parameters:

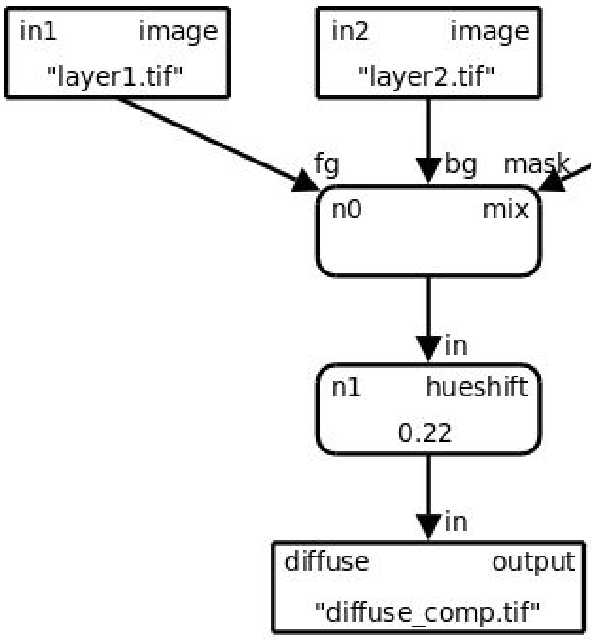
- `disable` (boolean, optional): if set to true, the node will pass its "bg", "in" or "in1" input value to its output, effectively disabling the node; default is false.

Some opgraph Operators also support the following parameters; those that do list these parameters in their definitions above:

- `channels` (string, optional): a string of one, two, three or four characters which describes the list of channels to restrict processing to. Channel names for colorN values are "r", "g", "b" and "a": to restrict an operation to process just the red and blue channel, use `channels="rb"`. For vectorN values, channel names are "x", "y", "z" and "w" (even if the data contained is more appropriately called something else). The `channels` option does not create any new channels in a stream: if any channels are listed that do not exist within the stream, that channel is ignored. The default is to operate on all incoming channels.

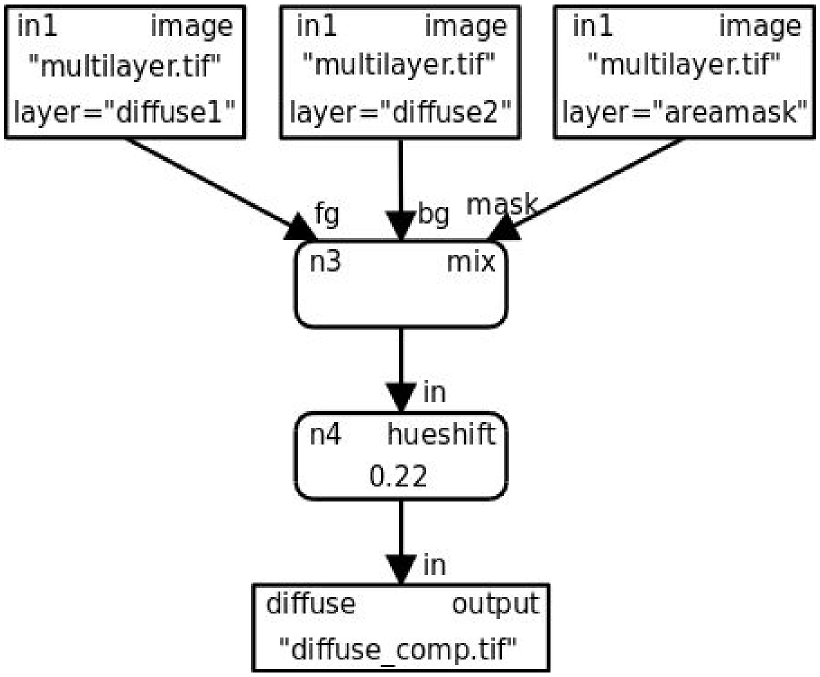
Opgraph Examples

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>
  <opgraph name="opgraph1">
    <image name="in1" type="color3">
      <parameter name="file" type="filename" value="layer1.tif"/>
    </image>
    <image name="in2" type="color3">
      <parameter name="file" type="filename" value="layer2.tif"/>
    </image>
    <image name="in3" type="float">
      <parameter name="file" type="filename" value="mask1.tif"/>
    </image>
    <mix name="n0" type="color3">
      <parameter name="fg" type="opgraphnode" value="in1"/>
      <parameter name="bg" type="opgraphnode" value="in2"/>
      <parameter name="mask" type="opgraphnode" value="in3"/>
    </mix>
    <hueshift name="n1" type="color3">
      <parameter name="in" type="opgraphnode" value="n0"/>
      <parameter name="amount" type="float" value="0.22"/>
    </hueshift>
    <output name="diffuse" type="color3">
      <parameter name="in" type="opgraphnode" value="n1"/>
      <parameter name="file" type="filename" value="diffuse_comp.tif"/>
    </output>
  </opgraph>
</materialx>
```

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



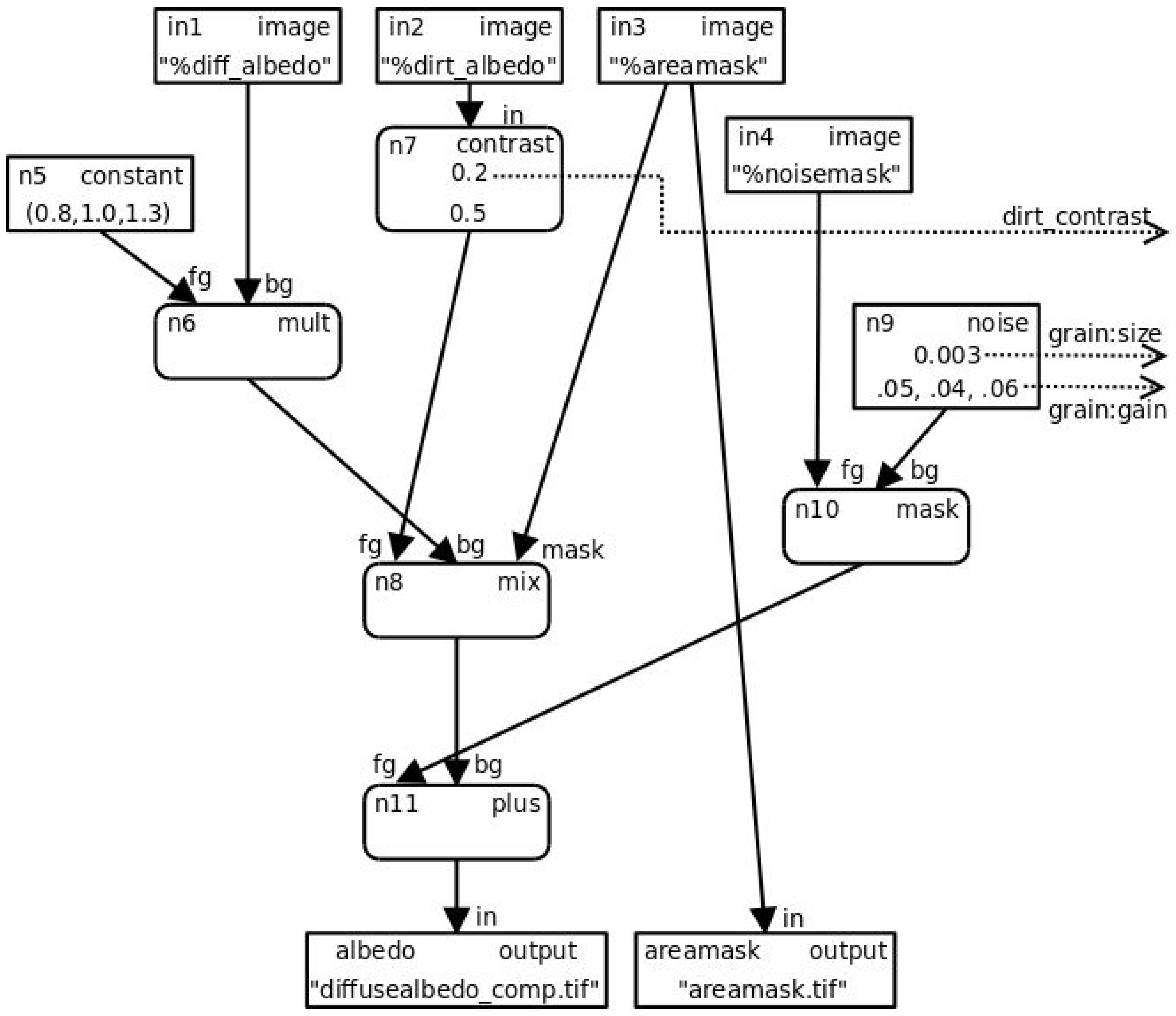
```
<?xml version="1.0" encoding="UTF-8"?>
<materialx require="multilayer">
  <opgraph name="opgraph2">
    <image name="in1" type="color3">
      <parameter name="file" type="filename" value="multilayer.tif"/>
      <parameter name="layer" type="string" value="diffuse1"/>
    </image>
    <image name="in2" type="color3">
      <parameter name="file" type="filename" value="multilayer.tif"/>
      <parameter name="layer" type="string" value="diffuse2"/>
    </image>
    <image name="in3" type="float">
      <parameter name="file" type="filename" value="multilayer.tif"/>
      <parameter name="layer" type="string" value="areamask"/>
    </image>
    <mix name="n3" type="color3">
      <parameter name="fg" type="opgraphnode" value="in1"/>
      <parameter name="bg" type="opgraphnode" value="in2"/>
      <parameter name="mask" type="opgraphnode" value="in3"/>
    </mix>
    <hueshift name="n4" type="color3">
      <parameter name="in" type="opgraphnode" value="n3"/>
      <parameter name="amount" type="float" value="0.22"/>
    </hueshift>
    <output name="diffuse" type="color3">
      <parameter name="in" type="opgraphnode" value="n4"/>
      <parameter name="file" type="filename" value="diffuse_comp.tif"/>
    </output>
  </opgraph>
</materialx>
```

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

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

```
    <image name="in1" type="color3">
      <parameter name="file" type="filename" value="$CONTAINER"/>
      <parameter name="layer" type="string" value="diffuse1"/>
    </image>
    <image name="in2" type="color3">
      <parameter name="file" type="filename" value="$CONTAINER"/>
      <parameter name="layer" type="string" value="diffuse2"/>
    </image>
    <image name="in3" type="float">
      <parameter name="file" type="filename" value="$CONTAINER"/>
      <parameter name="layer" type="string" value="areamask"/>
    </image>
```

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



```
<?xml version="1.0" encoding="UTF-8"?>
<materialx require="override">
  <!-- Note: in a real file, there would need to be geominfos here to define
%diff_albedo etc. for each geometry-->
  <opgraph name="opgraph3">
    <image name="in1" type="color3">
      <parameter name="file" type="filename" value="%diff_albedo"/>
    </image>
    <image name="in2" type="color3">
      <parameter name="file" type="filename" value="%dirt_albedo"/>
    </image>
    <image name="in3" type="float">
      <parameter name="file" type="filename" value="%areamask"/>
    </image>
    <image name="in4" 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>
    <mult name="n6" type="color3">
      <parameter name="fg" type="opgraphnode" value="n5"/>
      <parameter name="bg" type="opgraphnode" value="in1"/>
    </mult>
    <contrast name="n7" type="color3">
      <parameter name="in" type="opgraphnode" value="in2"/>
      <parameter name="amount" type="color3" value="0.2,0.2,0.2"
          publicname="dirt_contrast"/>
      <parameter name="pivot" type="color3" value="0.5,0.5,0.5"/>
    </contrast>
    <mix name="n8" type="color3">
      <parameter name="fg" type="opgraphnode" value="n7"/>
      <parameter name="bg" type="opgraphnode" value="n6"/>
      <parameter name="mask" type="opgraphnode" value="in3"/>
    </mix>
    <noise2d name="n9" type="color3">
      <parameter name="size" type="float" value="0.003" publicname="grain:size"/>
      <parameter name="amplitude" type="color3" value="0.05,0.04,0.06"
          publicname="grain:gain"/>
    </noise2d>
    <mask name="n10" type="color3">
      <parameter name="fg" type="opgraphnode" value="in4"/>
      <parameter name="bg" type="opgraphnode" value="n9"/>
    </mask>
    <plus name="n11" type="color3">
      <parameter name="fg" type="opgraphnode" value="n10"/>
      <parameter name="bg" type="opgraphnode" value="n8"/>
    </plus>
    <output name="albedo" type="color3">
      <parameter name="in" type="opgraphnode" value="n11"/>
      <parameter name="file" type="filename" value="diffusealbedo_comp.tif"/>
    </output>
    <output name="areamask" type="float">
      <parameter name="in" type="opgraphnode" value="in3"/>
      <parameter name="file" type="filename" value="areamask.tif"/>
    </output>
  </opgraph>
</materialx>
```

Collection Elements

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

Collection Definition

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

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

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

If an external file is capable of defining collections (e.g. in the geometry Alembic file), those collections can be referred to by Look assignments. [TBD: what that mechanism is. It is intended that the geometry lists that MaterialX collections and Alembic collections resolve to are interchangeable.]

CollectionAdd elements add geometries to a collection. There are three ways to specify geometry names to add: listing them individually or separated by commas in a "geom" attribute, via a regex expression using a "regex" attribute, or via the name of another collection:

```
<collectionadd name="name" geom="geom1[,geom2][,geom3...]"
        [includechildren=true|false]/>
<collectionadd name="name" regex="regexexpression"
        [includechildren=true|false]\>
<collectionadd name="name" collection="collectionname"/>
```

The `name` (string, required) attribute is not actually referenced by anything in MaterialX; it is required simply for the benefit of tracking the <collectionadd> declaration within an implementation; if there are multiple <collectionadd>s in a <collection> definition, those names must be unique. The syntax for regex expressions can be found in the Introduction section of this specification document.

The `includechildren` attribute can be used to determine behavior when a specified geometry path or regex matches something that is not a leaf node in a hierarchy. The default is "false", meaning that only that actual path is matched. Setting `includechildren` to true means that all child geometries in the hierarchy will also be included in the <collectionadd>, as if they were all individually specified.

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

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

```
<collectionremove name="name" geom="geom1[,geom2][,geom3...]
         [includechildren=true|false]"/>
<collectionremove name="name" regex="regexexpression"
         [includechildren=true|false]\>
```

The `includechildren` attribute behaves the same way here as in <collectionadd> declarations: false (the default) will match and remove only the exact listed or regex-matched geometries, true will also remove all child geometries. As with <collectionadd>, the `name` attribute is required but not actually referenced by any MaterialX element type.

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

Collections resolve to an exact list of geometries for look assignment purposes: host applications must do whatever is needed to ensure that their internal mechanisms apply the assignments to only and exactly the geometries resolved within the collections and not explicitly apply the assignment to any child geometries.

Geometry Info Elements

Geometry Info ("geominfo") elements declare geometric attribute data associated with specific external geometries. GeomInfo elements can apply multiple attribute values to multiple pieces of geometry or to collections of geometry. It is acceptable for several geominfo elements to reference the same geometry or the same attribute, as long as no two geominfo elements try to assign values of the same geometry attribute to the same geometry.

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

GeomInfo Definition

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

```
<geominfo name="name" [geom="geom1,geom2,geom3"][regex="expr"][collection="coll"]>
  ...geometry attribute declarations...
</geominfo>
```

Attributes for geominfo elements:

- `name` (string, required): the "name" of the geominfo; this name is somewhat arbitrary because no MaterialX element refers to the name of a geominfo, only the geomattrs within it.
- `geom` (string, optional): a comma-separated list of geometries that the geominfo is to apply to
- `regex` (string, optional): a regular expression for matching geometry names
- `collection` (string, optional): the name of a defined collection of geometries (see the Look and Collection Elements section)

Exactly one of a `geom`, a `regex`, or a `collection` must be specified, but not a combination.

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

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

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

GeomAttr elements have the following attributes:

- `name` (string, required): the name of the geometry attribute to define.
- `type` (string, required): the geometry attribute's type.
- `value` (any MaterialX type, optional): the value to assign to that attribute for this geometry; if `extern`=true, this is the default value of the attribute to use if the geometry file does not actually define the attribute value.
- `extern` (boolean, optional): if true, declares the existence of an externally-defined geometry attribute with a particular name and type; the implementation is expected to retrieve the value of this attribute for the current geometry during evaluation. `extern` is most commonly used with geom/regex/collection set to a large amount of geometry, perhaps all. Defaults to false.

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

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

and then reference that geomattr string in an image filename:

```
<opgraph name="op1">
  <image name="cc1" type="color3">
    <parameter name="file" type="filename"
        value="txt/color/asset.color.%txtid.tif"/>
  </image>
  ...
</opgraph>
```

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

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

```
<geominfo name="gi2" geom="/a/g2,/a/g3,/a/g4">
  <geomattr name="imagename" type="string" value="images/color.rustleftside.tif"/>
</geominfo>
<opgraph name="op2">
  <image name="cc1" type="color3">
    <parameter name="file" type="filename" value="%imagename"/>
  </image>
  ...
</opgraph>
```

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

```
<geominfo name="gi3" geom="/a/g5,/a/g7">
  <geomattr name="txtid" type="integer" value="1009"/>
  <geomattr name="clrname" type="string" value="color"/>
  <geomattr name="specname" type="string" value="specular"/>
</geominfo>
<geominfo name="gi4" geom="/a/g6,/a/g8,/a/g9">
  <geomattr name="txtid" type="integer" value="1010"/>
  <geomattr name="clrname" type="string" value="color3"/>
```

```
    <geomattr name="specname" type="string" value="specalt"/>
  </geominfo>
  <opgraph name="op2">
    <image name="cc2" type="color3">
      <parameter name="file" type="filename"
          value="txt/%clrname/asset.%clrname.%txtid.tif"/>
    </image>
    <output name="o_color2" type="color3">
      <parameter name="in" type="opgraphnode" value="cc2"/>
    </output>
    <image name="sc2" type="color3">
      <parameter name="file" type="filename"
          value="txt/%specname/asset.%specname.%txtid.tif"/>
    </image>
    <output name="o_spec2" type="color3">
      <parameter name="in" type="opgraphnode" value="sc2"/>
    </output>
  </opgraph>
```

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

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

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

Shader and Material Elements

Shader and material elements collectively define the input operator graph connections, parent/child coshader connections, parameter values, and optionally the output AOVs for each of one or more shaders referenced by a rendering material, as well as defining override values for public parameters in connected operator graphs and shaders. A MaterialX document can contain multiple material elements and multiple shader elements.

Operator graphs can be shared by multiple shaders, or even by multiple inputs of the same shader, however the parameter values within the opgraph will be the same for all uses within a material regardless of how many shader inputs an opgraph is connected to.

Shader Elements

Shader elements are used to declare external shading programs, providing descriptions of the various inputs they support, and defining the default values and connections of those inputs. Shader elements can contain any number of <input>, <coshader>, <parameter> and output <aov> declarations. Communication from opgraphs to shaders is through individual float and 2- through 4-channel color values connected to a shader <input>, while communication between shaders is accomplished through declared arbitrary output variables (AOVs) and connections to a shader <coshader>; the AOVs that are output by a shader only need to be explicitly declared if a <coshader> of another shader connects to it.

```
...optional <aovset> elements...
<shader name="name" shadertype="shadertype" shaderprogram="shaderprogram">
  ...parameter, input, coshader and aov declarations...
</shader>
```

The attributes for **<shader>** elements are:

- `name` (string, required): a user-chosen name for this shader: if multiple shaders are layered, this `name` is used to specify specific shaders for connections.
- `shadertype` (string, required): the type of shader, specifying the context in which the shader's output values should be interpreted. Common values are "surface", "displacement, "volume", and "light", but other shader types may be used as required by the renderer.
- `shaderprogram` (string, required): the name of the executable shader code, e.g. "blinn_phong", "disney_principled_2012", "bump2d", "volumecloud_vol", etc. It is presumed that a host application can find the actual executable shader file on disk given this `shaderprogram` string.
- `aovset` (string, optional): The name of a pre-declared aovset, defining the set of AOVs output by this shader. AOVs can also be declared with <aov> elements inside the <shader> element, which are added to the list declared by `aovset`.
- `xpos`, `ypos` (float, optional): the x,y position of the material in the UI representation of the shader graph; see the Standard Node Parameters section for details.

There are two additional <shader> attributes used *exclusively* by the standard MaterialX shaders (described below):

- `aovs` (string, optional): a comma-separated list of input AOVs that the shader should operate on.
- `passaovs` (string, optional): a comma-separated list of input AOVs to simply pass through to

the output, without being affected by the operation. For standard MaterialX shaders that take multiple inputs, the AOV value from the "bg" input is the one passed through.

For these MaterialX shaders, the lists of input AOVs specified in `aovs` and `passaovs` should be completely disjoint: any AOV specified should appear in exactly one of those lists. The MaterialX shader will output exactly the same aovs as declared as input aovs/passaovs.

Regular shaders (those calling defined `shaderprogram` code) do not themselves define input aovs. Rather, each <coshader> input declares the specific list of aovs the (parent) shader expects to read from that particular connected (child) shader.

MaterialX defines one special no-op shader, the **dot** shader, which can be used in between the connection of a coshader to its parent shader when the UI representation of the shader connection graph has user-defined "bends". The dot shader does not define or accept any aov definitions: it merely passes through the aov definitions by the actual shaders at either end of the chain of dot shader connections. Dot shaders do not have a shaderprogram, and must define exactly one input coshader for the shader output it connects to (which could be another "dot" shader).

```
  <shader name="dotname" shadertype="dot" xpos="xpos" ypos="ypos">
    <coshader name="inputname" shader="inputshadername"/>
  </shader>
```

An <implementation> element may also be specified to provide the functional definition of a <shader> for a particular application, either as a URI for a source code file, or as an opgraph. Attributes for shader <implementation> elements:

- `name` (string, required): a unique name for this <implementation>
- `shader` (shadernode, required): the name of the <shader> for which this <implementation> applies
- `sourcefile` (filename, optional): the URI of an external file or directory containing the source code for this shader. Ideally, source code for shaders should be written in a portable language such as OSL, GLSL, or MDL, but any language supported by the specified `application` is acceptable.
- `opgraph` (opgraphname, optional): the name of the opgraph element representing the source for this shader. [REQ="shadergraph"]
- `application` (string, required): the application or comma-separated list of applications for which this `<implementation>` applies.
- `language` (string, optional): the language in which the `sourcefile` code is written; defaults to "osl".

An <implementation> can define a `sourcefile` or an `opgraph`, but not both. If the <implementation> references an opgraph as the source, then the public parameters of that opgraph become the shader's parameters, the parameters of the referenced <opgraph> itself become shader inputs, and the outputs of the opgraph become the shader's AOVs.

Parameter elements define the values for (unconnectable) parameters of shaders: these are values of any MaterialX type (except "opgraphnode", "opgraphname" or "shadernode") that the shader source says cannot vary spatially or be connected to another shader or to an opgraph output. Parameter declarations have the following form:

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

Attributes for Parameter elements:

- name (string, required): the name of the shader parameter
- type (string, required): the MaterialX type of the shader parameter
- value (specified MaterialX type, required): the value for that parameter
- default (specified MaterialX type, optional): the default value of this parameter as defined by the shader: this is the value that would be used if no value was defined. The default attribute is provided mainly for documentation purposes so that the reader of a MaterialX file can see what a parameter's default value is when no value is assigned.
- publicname (string, optional): a publically-accessible name for this parameter, which is used in modifying the parameter from <override> elements. By default, a parameter has no publicname, and its value can not be externally overridden.

The value of any parameter must have the same type as defined for that parameter in the underlying shader code. It is not necessary to explicitly define a value for every shader parameter; those not explicitly defined will use the default value for that parameter defined by the shader itself (which can be declared in MaterialX using a default attribute for the <parameter>).

Input elements define the opgraph connections or values for shader inputs. Shader inputs can be defined to connect to the outputs of opgraphs [REQ="matopgraph"] passing in spatially-varying data, generally 1, 2, 3 or 4 channels of floating-point information.

Input declarations can have one of the following two forms:

```
    <input name="inputname" type="inputtype" value="value"
           [publicname="publicname"]/>
    <input name="inputname" type="inputtype" opgraph="opgraphname"
           graphoutput="outputname"/>
```

Attributes for Input elements:

- name (string, required): the name of the shader input parameter
- type (string, required): the MaterialX type of the shader parameter
- value (specified MaterialX type, optional): a static value to use for the input if an opgraph output is not connected.
- default (specified MaterialX type, optional): the default value of this input as defined by the shader: this is the value that would be used if no value or opgraph connection was defined. The default attribute is provided mainly for documentation purposes so that the reader of a MaterialX file can see what an input's default value is when unconnected and no value is assigned.
- opgraph (opgraphname, optional): the name of the opgraph element to connect to [REQ="matopgraph"]
- graphoutput (opgraphnode, optional): the value of the "name" attribute in an "output" node in the opgraph opgraph to connect to.
- publicname (string, optional): a publically-accessible name for this shader input, which is used

in modifying the input from <override> elements. By default, a shader input has no publicname, and its value can not be externally overridden. Note that only constant-valued shader inputs may be overridden, and the publicname attribute is ignored for shader inputs connected to opgraph outputs.

An input declaration can define either a value, or both an opgraph and a graphoutput. If an opgraph output is connected to a shader input, the type of the opgraph output must match the type of the shader input. It is not necessary to explicitly define a value or connection for every shader input; shader inputs that are defined without a value or connection will use the fixed default value for that input as defined by the shader itself (which can be declared in MaterialX using a `default` attribute for the <input>). Such an input could define a `publicname` for eventual material <override> referencing.

Coshader elements are similar to Input declarations, except they define the connections from shader parameters of type "coshader" to other (child) shaders [REQ="matshadergraph"]. Coshader declarations are different from Input declarations because the under-the-hood mechanism of what kind of data is passed and what form it takes is very different: Inputs take opgraph-output-compatible attribute types such as floats and color3s, while the actual data type for a coshader is implementation-specific and may pass an arbitrary number of channels of data under-the-hood via shader AOVs.

Coshader declarations have the form:

```
    <coshader name="inputname" shader="name" [aovset="aovsetname"]
             [aovs="aovlist"]/>
```

Attributes for Coshader elements:

- `name` (string, required): the name of the shader-type input
- `shader` (shadernode, required): the name of the shader element to connect to.
- `aovset` (string, optional): the name of an aovset defining a list of aovs that this (parent) shader expects to read from the (child) coshader.
- `aovs` (string, optional): a comma-separated list of aovs that this (parent) shader expects to read from the (child) coshader. This list can be a subset of all AOVs output by the connected shader: it is not necessary for the AOVs to match exactly.

For standard MaterialX shaders, neither aovset nor aovs can be defined for a <coshader>; the aovs/passaovs of its enclosing <shader> are used instead. For regular shaders, at least one of `aovset` or `aovs` must be defined; if both are defined, the union of the two lists will be used as the list of input aovs.

Coshaders cannot have a publicname, nor can their shader connection be set by a material <override>.

If a shader is connected to a child coshader through one or more dot shaders, then the expected input aovs are defined in the <coshader> element of the actual parent shader, while output aovs of the child shader are defined by that child <shader> element.

An **AOV** element declares the name and type of one output of a shader; shaders can output multiple AOVs. Most commonly, AOVs are used to describe post-illumination color values for a single lighting component, or an intermediate quantity calculated in the shader that may be useful in compositing, such as world-space position or depth. AOVs only need to be declared for shaders whose outputs will be used by other shaders; it is not necessary to declare output AOVs for the "final" shader in a shader network (e.g. one named in a <shaderref> within a <material> element), although it is not an error to do so. MaterialX allows AOVs of any type to be declared and passed between shaders, including custom types that have been declared through a <typedef> [REQ="customtype"]; see the Custom Types section. [REQ="matshadergraph"]

```
    <aov name="diffuse" type="color3"/>
```

Attributes for AOV elements:

- `name` (string, required): the name of the AOV to declare.
- `type` (string, required): the type of the AOV.

AOVSet elements: in place of declaring individual AOVs for every shader, one can declare an **aovset** (outside of a <shader> or <material>) and reference it in the <shader> declaration. AOV declarations inside a <shader> element are merged with the list of AOVs declared in the aovset.

The syntax for <aovset> elements is straightforward:

```
  <aovset name="standard">
    <aov name="Color" type="color3"/>
    <aov name="Opacity" type="float"/>
    <aov name="diffuse" type="color3"/>
    <aov name="specular" type="color3"/>
    <aov name="Pworld" type="vector3"/>
    <aov name="mydata" type="mystructtype"/>
  </aovset>
```

where `name` (string, required) is the name of the aovset or aov being declared, and `type` is the MaterialX type for that output AOV.

Material Elements

A **<material>** element contains one or more **<shaderref>** elements and/or a **<materialinherit>** element, which define what shaders a material references directly and what material (if any) it inherits from, respectively; there must be at least one shaderref and/or materialinherit defined. Material elements can also dynamically bind values and opgraph outputs to shader parameters and inputs, and declare override values for public parameters in referenced shaders or public parameters of opgraph nodes connected to inputs of a referenced shader.

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

Attributes for <material> elements:

- `name` (string, required): the name of the material.
- `xpos`, `ypos` (float, optional): the x,y position of the material in the UI representation of the shader graph; see the Standard Node Parameters section for details.

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

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

Attributes for <materialinherit> elements:

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

MaterialVar elements are used within a <material> (or a <look>) to define the value of a typed variable that can be substituted into the filename of <image> nodes, and/or the values of <override>s, <bindparam>s, and unconnected <bindinput>s within any opgraph node or shader referenced by the material. Materialvars work like look-specific overrides and are useful for creating variations, e.g. by defining alternate colors for shaders or different texture tokens for image file reads.

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

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

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

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

ShaderRef elements: A **<shaderref>** element declares the name of a shader that the enclosing material references; if the material uses a number of different shaders of the same type layered or composited together (e.g. they are connected via AOVs and coshaders), the <shaderref> should reference the final "top-level" output shader for any given type. There can only be one <shaderref> referencing a <shader> of any particular `shadertype` with a matching application value within a material, but it is perfectly fine for multiple <shaderref>s to be specified as long as the `shadertype/application` values of the referenced <shader>s are all unique.

Attributes for <shaderref> elements:

- `name` (string, required): the `name` of the <shader> element to reference in the material.

BindParam elements are used within <shaderref> or <material> elements to dynamically bind values to shader parameters, replacing any default assignments in the original <parameter> elements. These bindings persist only within the scope of the enclosing <shaderref> or <material> element.

```
<material name="steel">
  <shaderref name="simplesrf">
    <bindparam name="emissionColor" type="color3" value="0.005, 0.005, 0.005" />
    <bindparam name="rfrIndex" type="float" value="1.33" />
    <bindparam name="diffColor" type="color3" value="@steelcolor" />
  </shaderref>
</material>
```

Attributes for Bindparam elements:

- `name` (string, required): the `name` of the shader <parameter> which will be bound to a new value
- `shader` (string, optional): the name of the shader element that this binding would apply to, defaulting to the enclosing <shaderref> shader if the <bindparam> is defined inside a <shaderref>; if not, `shader` is required. This can be used to bind values to parameters of any coshaders connected at some point below the top-level shader referred to by the enclosing <shaderref>.
- `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. This can either be a literal value, or a reference to a materialvar [REQ="matvarvalue"].

BindInput elements are used within <shaderref> or <material> elements to dynamically bind values or opgraph outputs to shader inputs, replacing any default assignments in the original <input> elements. These bindings persist only within the scope of the enclosing <shaderref> or <material> element.

```
<material name="steel">
  <shaderref name="simplesrf">
    <bindinput name="diffColor" type="color3" opgraph="DiffNoiseNetwork"
           graphoutput="o_diffColor" />
    <bindinput name="specColor" type="color3" opgraph="SpecMapNetwork"
           graphoutput="o_specColor" />
    <bindinput name="roughness" type="float" value="0.02" />
  </shaderref>
</material>
```

Attributes for BindInput elements:

- `name` (string, required): the `name` of the shader <input> which will be bound to a new value or opgraph output
- `shader` (string, optional): the name of the shader element that this binding would apply to, defaulting to the enclosing <shaderref> shader if the <bindinput> is defined inside a <shaderref>; if not, `shader` is required. This can be used to bind values to unconnected inputs of any coshaders connected at some point below the top-level shader referred to by the enclosing <shaderref>.
- `type` (string, required): the MaterialX type of the shader <input>
- `value` (specified MaterialX type, optional): a value to bind to the shader input within the scope of this material. This can either be a literal value, or a reference to a materialvar [REQ="matvarvalue"].
- `opgraph` (opgraphname, optional): the name of the opgraph element whose output will be bound [REQ="matopgraph"]
- `graphoutput` (opgraphnode, optional): the name of the opgraph output node that will be bound

Either `value`, or both `opgraph` and `graphoutput` must be declared, but not both.

Override elements are used within <material> elements to change the value of a public parameter in an operator graph or shader. These overrides persist only within the scope of the enclosing material element. [REQ="override"]

```
<override name="grain:gain" type="float" value="0.075"/>
<override name="dirtmix" type="float" value="@dirtmixval"/>
```

Attributes for Override elements:

- `name` (string, required): the **`publicname`** of the public parameter
- `type` (string, required): the MaterialX type of the public parameter that is being overridden
- `value` (specified MaterialX type, required): a value to assign to the public parameter within the scope of this material. This can either be a literal value, or a reference to a materialvar [REQ="matvarvalue"].

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

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

Standard MaterialX Shaders

The following shaders are included as standard for MaterialX implementations that support "matshadergraph" capability, with a specific version for each supported renderer. These shaders are essentially wrappers for certain standard Opgraph operations, iterating over all incoming AOVs and applying the particular operator function to each AOV in turn. Since AOVs sometimes contain non-color data, the shaders allow an explicit declaration of what AOVs to operate on (aovs), and what AOVs to simply pass through unchanged from their input (passaovs). These lists are used *instead of* declaring output aovs or aovsets: the aovs output will be the union of the set specified in aovs/passaovs. If no aovs/passaovs are specified, then the shaders will operate on all float and color3 AOVs present in the "in" (or "bg") shader's output.

Note: the standard MaterialX shaders can only operate on float and color3 AOVs (aovs of any other type will be passed through unchanged). The <input>s of standard MaterialX shaders are all color3 types, e.g. they have a color3 value or connect to a color3 opgraph output; for any aovs that are of type "float", the first (red) channel value of the input will be used. There are standard MaterialX shaders for many opgraph node types listed in the "Standard Operators" section, e.g. most Math, Color-Correction, and the Blend Compositing nodes, but not Premult/Unpremult, Channel, Convolution or Merge, Masking or Mix Compositing nodes.

Math and Color-Correction Shaders

Takes one input coshader "in" and one or more shader <input>s applies the specified operator to all requested AOVs. Note that the `magnitude` operator is not supported as a standard MaterialX shader.

Shaderprogram	Operator	<Input> Names	Output
mx_add	add	amount	in + amount
mx_subtract	subtract	amount	in - amount
mx_multiply	multiply	amount	in * amount
mx_divide	divide	amount	in / amount
mx_invert	invert	amount	amount - in
mx_absval	absval	[none]	abs(in)
mx_exponent	exponent	amount	pow(in, amount)
mx_contrast	contrast	amount, pivot	contrast(in,amount,pivot)
mx_clamp	clamp	low, high	clamp(in,low,high)
mx_remap	remap	inlow,inhigh,gamma,outlow,outhigh,doclamp	remap(in, ...)
mx_normalize	normalize	[none]	normalize(in)
mx_hueshift	hueshift	amount	hueshift(in, amount)

mx_saturate	saturate	amount	saturate(in,amount)
mx_luminance	luminance	[none]	luminance(in)
mx_colorspacetransform	colorspacetransform	fromcolorspace, tocolorspace	colorspacetransform(in, ...)

Compositing Shaders

Takes two input coshaders "bg" and "fg" and performs the specified compositing operator across the two inputs on matching pairs of all requested AOVs; the "mx_mix" shader also has a shader <input> "amount". Note that the `burn`, `dodge`, `div`, `minus`, `overlay`, `dotproduct` and `crossproduct` operators are not supported as a standard MaterialX shader. For all passaovs, the output value will the the corresponding AOV from the "bg" input shader.

Shaderprogram	Operator	Inputs	Output
mx_average	average	[none]	(fg + bg) / 2
mx_difference	difference	[none]	abs(fg - bg)
mx_max	max	[none]	max(fg, bg)
mx_min	min	[none]	min(fg, bg)
mx_mult	mult	[none]	fg * bg
mx_plus	plus	[none]	fg + bg
mx_screen	screen	[none]	1 - (1-fg)*(1-bg)
mx_mix	mix	amount	(fg * amount) + bg * (1-amount)

Examples

Example 1: Define two shaders and two materials with different values assigned to the shader(s). One input ("diff_albedo") is connected to the output of an externally-defined operator graph, and the materials bind values to parameters and inputs for shaders, as well as define override values for a public parameter exposed in the external opgraph. The first material references both a surface and a displacement shader, while the second references only a surface shader.

```
<?xml version="1.0" encoding="UTF-8"?>
<materialx require="matopgraph,override">
  <-- Assume "opgraph3.mtlx" defines "opgraph3", which contain nodes defining --/>
  <-- publicnames "diffcolor", "dirt_contrast", "grain:gain" and "grain:size". --/>
  <xi:include href="opgraph3.mtlx"/>
  <shader name="srf1" shadertype="surface" shaderprogram="basic_surface">
    <input name="diff_albedo" type="color3" opgraph="opgraph3"
```

```
        graphoutput="albedo"/>
    <input name="spec_color" type="color3"/>
    <input name="roughness" type="float"/>
    <parameter name="fresnel_exp" type="float"/>
  </shader>

  <shader name="dsp1" shadertype="displacement" shaderprogram="noise_bump">
    <parameter name="bump_scale" type="float" default="0.02"/>
    <parameter name="bump_ampl" type="float" default="0.015"/>
  </shader>

  <material name="material1">
    <shaderref name="srf1">
      <bindinput name="spec_color" type="color3" value="1.0, 0.99, 0.95"/>
      <bindinput name="roughness" type="float" value="0.15"/>
      <bindparam name="fresnel_exp" type="float" value="0.2"/>
    </shaderref>
    <shaderref name="dsp1">
      <bindparam name="bump_ampl" type="float" value="0.0125"/>
    </shaderref>
    <override name="diffcolor" type="color3" value="0.25, 0.24, 0.16"/>
    <override name="dirt_contrast" type="float" value="0.33"/>
    <override name="grain:gain" type="float" value="0.1"/>
    <override name="grain:size" type="float" value="0.008"/>
  </material>

  <material name="material2">
    <shaderref name="srf1">
      <bindinput name="spec_color" type="color3" value="1,1,1"/>
      <bindinput name="roughness" type="float" value="0.1"/>
      <bindparam name="fresnel_exp" type="float" value="0.3"/>
    </shaderref>
    <override name="dirt_contrast" type="float" value="0.31"/>
    <override name="grain:gain" type="float" value="0.075"/>
    <override name="grain:size" type="float" value="0.008"/>
  </material>
</materialx>
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<materialx require="matopgraph">
  <opgraph name="shaderparams">
    <constant name="steel_diffc" type="color3">
      <parameter name="value" type="color3" value="0.0318, 0.0318, 0.0318"/>
    </constant>
    <constant name="steel_specc" type="color3">
      <parameter name="value" type="color3" value="0.476, 0.476, 0.476"/>
    </constant>
    <constant name="steel_roughf" type="float">
      <parameter name="value" type="float" value="0.05"/>
    </constant>
    <image name="rust_diffc" type="color3">
      <parameter name="file" type="filename" value="rust_diffc.tif"/>
    </image>
    <constant name="rust_specc" type="color3">
      <parameter name="value" type="color3" value="0.043, 0.043, 0.043"/>
    </constant>
    <constant name="rust_roughf" type="float">
      <parameter name="value" type="float" value="0.5"/>
    </constant>
    <constant name="paint_diffc" type="color3">
      <parameter name="value" type="color3" value="0.447, 0.447, 0.447"/>
    </constant>
    <constant name="paint_specc" type="color3">
      <parameter name="value" type="color3" value="0.144, 0.144, 0.144"/>
    </constant>
    <constant name="paint_roughf" type="float">
      <parameter name="value" type="float" value="0.137"/>
    </constant>
    <image name="mask_rust" type="float">
      <parameter name="file" type="filename" value="mask_rust.tif"/>
    </image>
    <image name="mask_paint" type="float">
      <parameter name="file" type="filename" value="mask_paint.tif"/>
    </image>
    <mix name="mix_diff1" type="color3">
      <parameter name="fg" type="opgraphnode" value="rust_diffc"/>
      <parameter name="bg" type="opgraphnode" value="steel_diffc"/>
      <parameter name="mask" type="opgraphnode" value="mask_rust"/>
    </mix>
    <mix name="mix_diff2" type="color3">
      <parameter name="fg" type="opgraphnode" value="paint_diffc"/>
      <parameter name="bg" type="opgraphnode" value="mix_diff1"/>
      <parameter name="mask" type="opgraphnode" value="mask_paint"/>
    </mix>
    <output name="o_diffcolor" type="color3">
      <parameter name="in" type="opgraphnode" value="mix_diff2"/>
    </output>
    <mix name="mix_spec1" type="color3">
      <parameter name="fg" type="opgraphnode" value="rust_specc"/>
      <parameter name="bg" type="opgraphnode" value="steel_specc"/>
      <parameter name="mask" type="opgraphnode" value="mask_rust"/>
    </mix>
    <mix name="mix_spec2" type="color3">
      <parameter name="fg" type="opgraphnode" value="paint_specc"/>
      <parameter name="bg" type="opgraphnode" value="mix_spec1"/>
      <parameter name="mask" type="opgraphnode" value="mask_paint"/>
    </mix>
    <output name="o_speccolor" type="color3">
      <parameter name="in" type="opgraphnode" value="mix_spec2"/>
    </output>
    <mix name="mix_rough1" type="float">
      <parameter name="fg" type="opgraphnode" value="rust_roughf"/>
      <parameter name="bg" type="opgraphnode" value="steel_roughf"/>
      <parameter name="mask" type="opgraphnode" value="mask_rust"/>
    </mix>
    <mix name="mix_rough2" type="float">
      <parameter name="fg" type="opgraphnode" value="paint_roughf"/>
      <parameter name="bg" type="opgraphnode" value="mix_rough1"/>
      <parameter name="mask" type="opgraphnode" value="mask_paint"/>
    </mix>
    <output name="o_roughness" type="float">
      <parameter name="in" type="opgraphnode" value="mix_rough2"/>
    </output>
  </opgraph>

  <shader name="rman_srf3" shadertype="surface" application="rman"
          shaderprogram="basic_srf">
    <input name="diff_albedo" type="color3" opgraph="shaderparams"
          graphoutput="o_diffcolor"/>
    <input name="spec_color" type="color3" opgraph="shaderparams"
          graphoutput="o_speccolor"/>
    <input name="roughness" type="float" opgraph="shaderparams"
          graphoutput="o_roughness"/>
  </shader>
  <shader name="vray_srf3" shadertype="surface" application="vray"
          shaderprogram="basic_surface">
    <input name="albedo" type="color3" opgraph="shaderparams"
          graphoutput="o_diffcolor"/>
    <input name="speccolor" type="color3" opgraph="shaderparams"
          graphoutput="o_speccolor"/>
    <input name="roughness" type="float" opgraph="shaderparams"
          graphoutput="o_roughness"/>
    <parameter name="fresnel" type="float" value="0.25"/>
  </shader>

  <material name="blendedmtl">
    <shaderref name="rman_srf3"/>
    <shaderref name="vray_srf3"/>
  </material>
</materialx>
```

Example 3: A material using post-shader compositing. This example defines three surface shaders: "steel", "plastic", and "glass", and blends their output AOVs using standard MaterialX shaders. Because the standard MaterialX shaders have necessarily been written generically, they must be told which AOVs to process and which to pass through. This example uses an aovset to declare the AOVs output by the "steel", "plastic", and "glass" shaders.

```
<?xml version="1.0" encoding="UTF-8"?>
<materialx require="matopgraph,matshadergraph">
  <opgraph name="shadermaps">
    <image name="plastic_diffc" type="color3">
      <parameter name="file" type="filename" value="plastic_diffc.tif"/>
    </image>
    <output name="o_plastic_diff" type="color3">
      <parameter name="in" type="opgraphnode" value="plastic_diffc"/>
    </output>
    <image name="mask_plastic" type="float">
      <parameter name="file" type="filename" value="mask_plastic.tif"/>
    </image>
    <output name="o_mask_plastic" type="float">
      <parameter name="in" type="opgraphnode" value="mask_plastic"/>
    </output>
    <image name="mask_glass" type="float">
      <parameter name="file" type="filename" value="mask_glass.tif"/>
    </image>
    <output name="o_mask_glass" type="float">
      <parameter name="in" type="opgraphnode" value="mask_glass"/>
    </output>
  </opgraph>

  <aovset name="standard">
    <aov name="Color" type="color3"/>
    <aov name="Opacity" type="float"/>
    <aov name="diffuse" type="color3"/>
    <aov name="specular" type="color3"/>
    <aov name="Pworld" type="vector3"/>
  </aovset>

  <shader name="steel" shadertype="surface" shaderprogram="basic_surface"
          aovset=”standard”>
    <input name="diff_albedo" type="color3" value="0.0318, 0.0318, 0.0318"/>
    <input name="spec_color" type="color3" value="0.476, 0.476, 0.476"/>
    <input name="roughness" type="float" value="0.05"/>
    <parameter name=”fresnel_expon” type="float" value="0.25"/>
  </shader>

  <shader name="plastic" shadertype="surface" shaderprogram="basic_surface"
          aovset=”standard”>
    <input name="diff_albedo" type="color3" opgraph="shadermaps"
          graphoutput="o_plastic_diff"/>
    <input name="spec_color" type="color3" value="0.35, 0.35, 0.35"/>
    <input name="roughness" type="float" value="0.38"/>
    <parameter name=”fresnel_expon” type="float" value="0.3"/>
  </shader>

  <shader name="glass" shadertype="surface" shaderprogram="basic_surface"
          aovset=”standard”>
    <input name="diff_albedo" type="color3" value="0.001, 0.001, 0.001"/>
    <input name="spec_color" type="color3" value="0.6, 0.6, 0.6"/>
    <input name="roughness" type="float" value="0.01"/>
    <input name="opacity" type="float" value="0.002"/>
    <parameter name=”fresnel_expon” type="float" value="0.2"/>
  </shader>

  <shader name="blendsrf1" shadertype="surface" shaderprogram="mx_mix"
          aovs="Color,Opacity,diffuse,specular" passaovs=”Pworld”>
    <coshader name="bg" shader="steel"/>
    <coshader name="fg" shader="plastic"/>
    <input name="amount" type="float" opgraph="shadermaps"
          graphoutput="o_mask_plastic"/>
  </shader>
```

```
  <shader name="blendsrf2" shadertype="surface" shaderprogram="mx_mix"
          aovs="Color,Opacity,diffuse,specular" passaovs=”Pworld”>
    <coshader name="bg" shader="blendsrf1"/>
    <coshader name="fg" shader="glass"/>
    <input name="amount" type="float" opgraph="shadermaps"
          graphoutput="o_mask_glass"/>
  </shader>

  <material name="layeredmat">
    <shaderref name="blendsrf2"/>
  </material>
</materialx>
```

The final pair of shaders and material in the previous example could also be expressed as follows, presuming the existence of a user-provided "A over B over C" layering shader called "threelayer_surface" that is AOV-compatible with the "basic_surface" used by the steel, plastic and glass shaders. The following material declaration would take the place of the "blend1" and "blend2" shaders and the "layeredmat" material in the MaterialX file above; note the required addition of aovset attributes for the coshader elements:

```
  <shader name="blendsrf3" shadertype="surface" shaderprogram="threelayer_surface">
    <coshader name="layer1" shader="steel" aovset="standard"/>
    <coshader name="layer2" shader="plastic" aovset="standard"/>
    <coshader name="layer3" shader="glass" aovset="standard"/>
    <input name="blend1_2" type="float" opgraph="shadermaps"
          graphoutput="o_mask_plastic"/>
    <input name="blend12_3" type="float" opgraph="shadermaps"
          graphoutput="o_mask_glass"/>
  </shader>
  <material name="userblendmat">
    <shaderref name="blendsrf3"/>
  </material>
```

Lights

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

Since MaterialX treats lights like any other geometry other than the type of shader assigned to it, lights can be turned off (muted) in looks by making the light geometry "invisible".

A **<light>** element defines two attributes in addition to its name: a `geom` defining the name of the light object in the external scene, and a `material` defining the light's shader(s) and parameters.

```
  <light name="lightname" geom="lightgeom" material="materialname"/>
```

Attributes for <light> elements:

- `name` (string, required): the name of the light: this name is what is referenced by <lightillum> and <lightshadow> elements within looks, and does not have to match the name of the light object in the external scene.
- `geom` (string, required): the full pathname for the object in the scene representing the light (see the Geometry Representation section); `geomfile` and `geomprefix` are respected if defined in the current scope.
- `material` (string, required): the name of the <material> element which defines the shader(s) and parameters used by the light.

Look and Property Elements

Look elements define the assignments of materials and other properties to geometries and geometry collections. In MaterialX, a number of geometries are associated with each stated material 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 several standard MaterialX property types that can be applied regardless of what renderer is used, as well as a mechanism to define renderer-specific properties for geometries or collections.

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

PropertySet Definition

A **<propertyset>** element consists of one or more <property> elements, each of which define the name, type and value of a non-material property of geometry; the connection between propertysets and specific geometries or collections is done in a <look> element, so that these propertysets can be reused across different geometries, and be enabled in some looks but not others. Property names for specific applications are allowed to define arbitrarily-deep categorization paths using a ":" character as a separator between path components.

```
<propertyset name="set1">
  <property name="twosided" type="boolean" value="true"/>
  <property name="vistocamera" type="boolean" value="false"/>
  <property name="trace:bias" application="rman" type="float" value="0.002"/>
</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
`invisible`	boolean	false
`vistocamera`	boolean	true
`vistoshadow`	boolean	true
`vistosecondary`	boolean	true

When the "invisible" property is set to "true", it takes precedence over any "visto..." properties, and geometry assigned to the "invisible" propertyset will not be visible to camera, shadows or secondary rays. The "invisible" property can be used to turn off geometry not needed in certain variations of an asset, e.g. different costume pieces or damage shapes. The "invisible" property can also be assigned to a light "geometry" to mute/disable the light for that look.

In the example above, the "trace:bias" property is renderer-specific, and has been restricted to the context of Renderman by setting its `application` attribute to “rman”. For more details on application and renderer-specific properties, see the **Custom MaterialX Elements** section of the specification.

Look Definition

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

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

Looks can inherit the assignments from another look by including a **<lookinherit>** element. The look can then specify additional assignments that will apply on top of/in place of whatever came from the source look. This is useful for defining a base look and then one or more "offshoot" or "variation" looks. Implementations that do not natively support look inheritance should insert the representation of the parent lookinherit look into the host application's "look" mechanism right before the representation of the other assigns in the look, effectively concatenating the assignments into a single list. It is permissible for an inherited-from look to itself inherit from another look, but a look can inherit from only one parent look.

```
<look name="lookname">
  <lookinherit name="looktoinheritfrom">
  ...materialassign, lightillum/lightshadow, propertysetassign declarations...
  ...optional <materialvar> elements...
</look>
```

MaterialAssign, **LightIllum**, **LightShadow** and **PropertySetAssign elements** are used within a <look> to connect a specified material, asset light or propertyset to one or more geometries, collections or regexes.

```
  <materialassign name="materialname" [geom="geom1[,geom2...]"]
       [collection="collectionname"] [regex="regexpr"] [exclusive=true|false]/>
  <lightillum name="lightname" [geom="geom1[,geom2...]"]
       [collection="collectionname"] [regex="regexpr"] [global=true|false]/>
  <lightshadow name="lightname" [geom="geom1[,geom2...]"]
       [collection="collectionname"] [regex="regexpr"]/>
  <propertysetassign name="propertysetname" [geom="geom1[,geom2...]"]
       [collection="collectionname"] [regex="regexpr"]/>
```

Each of the above assignment declaration element types can contain either a `geom` attribute, a `collection` attribute, or a `regex` attribute. Multiple assignment declarations can be used with the same material/light/propertysetname to add more and more geometries or collections assigned to the same material, light or propertyset; the assignments are additive. Note that <propertysetassign> must refer to the name of a <propertyset>; individual <property> names cannot be assigned.

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 assigns, the last assign wins. However, some applications allow multiple materials to be assigned to the same geometry as long as the shadertypes don't overlap. If the `exclusive` attribute is set to false (default is true), then earlier material assigns will still take effect for all shadertypes not defined in the materials of later assigns: for each shadertype, the shader within the latest material assigned wins. If a particular application does not support multiple material assignments to the same geometry, the value of `exclusive` is ignored and only the last full material and its shaders are assigned to the geometry, and the parser should issue a warning. [REQ="multiassign" in order to set `exclusive=false`]

Lighting assignments indicate either what geometry is illuminated by a specified light (<lightillum>), or what geometry will occlude illumination (e.g. cast shadows) from the specified light (<lightshadow>). Multiple <lightillum> and/or <lightshadow> declarations can be specified for any light, resulting in assignment to the union of the geometries referenced. Illumination and shadow assignments are made referencing the MaterialX name given to the light, not the name of the light "geometry" in the external scene. Lights must explicitly be assigned to some geometry in order to "turn the light on"- they are not considered on by default. Assigning an `invisible` property to light geometry will effectively mute the light so it will not illuminate anything; any <lightillum> for that light will be ignored. This can be useful in inherited look situations, where a master look defines all the asset lights, and individual looks inheriting it can turn certain lights off when needed. Light assignments can be declared `global=true` (the default) to have the asset light also illuminate other assets, or `global=false` to only illuminate within the asset.

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

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<materialx>
  <!-- <shader> and <material> elements to define Mplastic1,2 and Mmetal1,2 here -->
  <collection name="c_plastic">
    <collectionadd name="ca1" geom="/a/g1,/a/g2,/a/g5"/>
  </collection>
  <collection name="c_metal">
    <collectionadd name="ca2" geom="/a/g3,/a/g4"/>
  </collection>
  <collection name="c_headlampAssembly">
    <collectionadd name="ca3" regex="/a/lamp1/housing*Mesh"/>
  </collection>
  <collection name="c_setgeom">
```

```
    <collectionadd name="ca4" regex="/b//*"/>
  </collection>
  <shader name="s_headlight1" shadertype="light" shaderprogram="disk_lgt">
    <parameter name="emissionmap" type="filename" value="txt/fresnellamp.1.tif"/>
    <parameter name="gain" type="float" value="2000.0"/>
  </shader>
  <material name="mheadlight">
    <shaderref name="s_headlight1"/>
  </material>
  <light name="headlgt1" geom="/a/b/headlight" material="mheadlight"/>
  <propertyset name="default">
    <property name="displacementbound:sphere" application="rman" type="float"
           value="0.05"/>
    <property name="trace:bias" application="rman" type="float" value="0.002"/>
  </propertyset>
  <propertyset name="shadowblocker">
    <property name="vistocamera" type="boolean" value="false"/>
    <property name="vistoshadow" type="boolean" value="true"/>
  </propertyset>
  <look name="lookA">
    <materialassign name="Mplastic1" collection="c_plastic"
    <materialassign name="Mmetal1" collection="c_metal"
    <lightillum name="headlgt1" regex="//*"/>
    <lightshadow name="headlgt1" collection="c_headlampAssembly"/>
    <propertysetassign name="default" regex="//*"/>
  </look>
  <look name="lookB">
    <materialassign name="Mplastic2" collection="c_plastic"
    <materialassign name="Mmetal2" collection="c_metal"
    <propertysetassign name="default" regex="//*"/>
    <propertysetassign name="shadowblocker" collection="c_setgeom"/>
  </look>
</materialx>
```

Custom MaterialX Elements

The MaterialX Specification describes the elements that are meaningful to MaterialX-compliant applications and renderers, but it is permissible to add custom attributes to defined elements, as well as completely new opgraph operator elements. If applications are capable of preserving and re-outputting these custom attributes and elements (complete with connections and parameter values) they should, however no application should depend on custom elements or attributes being preserved.

Custom Attributes

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

```
<material name="sssmarble" maxmtlname="SSS Marble">
  <shaderref name="marblesrf2"/>
</material>
```

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

Custom attributes are ignored by applications that do not understand them.

Custom Parameters and Properties

If an application or renderer requires the storage of custom parameters or properties within standard MaterialX elements, it may add parameters or properties with non-standard names, with the `application` attribute set to the name of the application or renderer in which these elements should be considered valid; these parameters and properties are ignored by all other applications or renderers.

```
<image name="image1" type="color4">
  <parameter name="file" type="filename" value="image1.tif"/>
  <parameter name="uicolor" application="maya" type="color3" value="0.0,0.5,0.5"/>
</image>

<propertyset name="set1">
  <property name="twosided" type="boolean" value="true"/>
  <property name="trace:bias" application="rman" type="float" value="0.002"/>
</propertyset>
```

In the above examples, a Maya-specific parameter named “uicolor” has been added to an image node, and a Renderman-specific property named “trace:bias” has been added to a property set.

Custom Types

MaterialX allows the specification of arbitrary named data types for the inputs and outputs of custom opgraph nodes and shaders [REQ="customtype"]. This allows data to be passed to and from custom opgraph nodes and shaders of any complex type an application may require; examples might include BXDF profiles or spectral color samples. It is not necessary to define the contents or structure of a custom type, only to declare its name.

Custom types are defined using the <typedef> element:

```
<typedef name="customtype"/>
```

Attributes for <typedef> elements:

- `name` (string, required): the name of this type. Can not be the same as a built-in MaterialX type.
- `valueformat` (string, optional): a space-separated string of MaterialX basic or array type names, e.g. "integer float color3 stringarray[4]". Array types must specify their length in square brackets; variable-length arrays are not supported.

If a `valueformat` attribute is provided, then a .mtlx file can specify the value for that type any place it is used, as a space-separated list of numbers and strings, with the expectation that the numbers and strings exactly line up with the expected types in the `valueformat`. For example, if the following <typedef> was declared:

```
<typedef name="exampletype" valueformat="integer color3 stringarray[2] vector2
     vector2"/>
```

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

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

Custom Opgraph Nodes

Specific applications will likely have operators and sources that do not map directly to individual or combinations of standard MaterialX operators and sources. Individual implementations may define and reference their own custom opgraph nodes [REQ="customnode"], subject to the following guidelines.

- Nodes that process input image data (referred to as **custom operators**) have one or more "opgraphnode"-type input parameter, typically named `in` for single-input functions, `in1` and

in2 or fg and bg for two-input functions (note however, there is no intrinsic limit to the number of inputs or a required convention for naming of inputs). Custom operators cannot be inputless. Implementations that do not understand a custom operator should pass the "in", "in1" or "bg" input value unchanged to the output (or 0.0 in all channels if no such named input exists), and applying the type conversion rules of the `<convert>` operator if the input and output types do not match.
- Nodes that generate image data as opposed to processing incoming values (referred to as **custom sources**) can not have any "opgraphnode"-type inputs. Implementations that do not understand a custom source should treat it as outputting 0.0 in all output channels.
- Any values for the custom node should be specified as parameter elements within the element for the custom node: these parameter elements should have "name", "type" and "value" attributes.

Custom Node Definition

Each custom opgraph node must be explicitly declared with a <nodedef> element, specifying the expected names and types of the node's inputs and outputs. The <nodedef> element can contain one or more parameters with `name`, `type` and (for externally-defined nodes) `default` values; input parameters of type "opgraphnode" must also define `intype`, specifying the MaterialX or custom type that the input expects. It is permissible to define multiple <nodedef>s with the same `node` name but different input, output and/or parameter types (e.g. one version with float inputs and another with color3 inputs). A custom node parameter without a default value becomes a required parameter, so any invocation of that custom opgraph node without a value for that parameter would be in error.

Attributes for <nodedef> elements:
- `name` (string, required): a unique name for this <nodedef>
- `node` (string, required): the name of the custom node being defined
- `type` (string, required): the type of the output of this custom node, which can be a standard MaterialX type or a custom type declared through a <typedef>. <Nodedef>s for custom nodes with multiple outputs should declare `type` as "multioutput" [REQ="multioutput"].

Note: it is permissible for `node` to be the name of a standard MaterialX opgraph node name, such as "`add`", as long as the combination of node parameter types and the <nodedef> (output) type include at least one <typedef>'ed custom type so the <nodedef> does not conflict with a standard definition. This allows overloading of standard MaterialX node types to support custom data types.

The implementation of a custom opgraph node may be defined either through external source code or via an opgraph. The following attributes are supported by <implementation> elements for custom opgraph nodes:
- `name` (string, required): a unique name for this <implementation>
- `nodedef` (string, required): the name of the <nodedef> to which this <implementation> applies
- `file` (filename, optional): the URI of an external file containing the source code for 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 specified `application` is acceptable.
- `application` (string, required): when `file` is specified, the application or comma-separated

list of applications for which this `<implementation>` applies.

- `language` (string, optional): when `file` is specified, the language in which the `file` code is written; defaults to "osl".
- `opgraph` (opgraphname, optional): the name of an opgraph describing the functionality of the <nodedef>'ed node.

Either `file` or `opgraph` can be specified, but not both.

Here is an example of some custom nodes defined with external file implementations:

```
<nodedef name="mblendcolor3" node="mariBlend" type="color3">
  <parameter name="in1" type="opgraphnode" intype="color3"/>
  <parameter name="in2" type="opgraphnode" intype="color3"/>
  <parameter name="ColorA" type="color3" default="0.0, 0.0, 0.0"/>
  <parameter name="ColorB" type="color3" default="0.0, 0.0, 0.0"/>
</nodedef>
<nodedef name="mblendfloat" node="mariBlend" type="float">
  <parameter name="in1" type="opgraphnode" intype="float"/>
  <parameter name="in2" type="opgraphnode" intype="float"/>
  <parameter name="ColorA" type="float" default="0.0"/>
  <parameter name="ColorB" type="float" default="0.0"/>
</nodedef>
<nodedef name="mnoisecolor3" node="mariCustomNoise" type="color3">
  <parameter name="ColorA" type="color3" default="0.5, 0.5, 0.5"/>
  <parameter name="Size" type="float" value="1.0"/>
</mariCustomNoise>
<implementation name="mari_mblendc3" nodedef="mblendcolor3" application="mari"
       file="lib/mtlx_mari_funcs.glsl" language="glsl"/>
<implementation name="mari_mblendf" nodedef="mblendfloat" application="mari"
       file="lib/mtlx_mari_funcs.glsl" language="glsl/>
<implementation name="mari_mnoisec3" nodedef="mnoisecolor3" application="mari"
       file="lib/mtlx_mari_funcs2.glsl" language="glsl/>
<implementation name="maya_mblendc3" nodedef="mblendcolor3" application="maya"
       file="lib/mtlx_maya_funcs.osl" language="osl"/>
<implementation name="maya_mblendf" nodedef="mblendfloat" application="maya"
       file="lib/mtlx_maya_funcs.osl" language="osl"/>
<implementation name="maya_mnoisec3" nodedef="mnoisecolor3" application="maya"
       file="lib/mtlx_maya_funcs2.osl" language="osl"/>
...
```

The above example defines two templates for a custom operator called "mariBlend" (one operating on color3 values, and one operating on floats), and one template for a custom source called "mariNoise". Implementations of these functions have been defined for Mari and for Maya.

<Nodedef>s with multiple outputs must define at least two <output> elements within the <nodedef> to define the name and types of each output [REQ="multioutput"]. Note: the name the output of single-output opgraph nodes is always "out".

```
<nodedef name="dblclr3" node="doublecolor" type="multioutput">
  <parameter name="seed" type="float" default="1.0"/>
  <output name="c1" type="color3"/>
  <output name="c2" type="color3"/>
</nodedef>
<implementation name="osl_dc3" nodedef="dblclr3" application="maya"
```

```
        file="lib/mtlx_maya_funcs.osl"/>
```

Here is an example of a custom node defined using an opgraph. Note that the parameters of the opgraph are the "inputs" of the <nodedef>'ed custom node, and that the nodes within the opgraph reference those parameters in the "value" attributes of appropriate internal nodes by prefixing the parameter name with a "$". Note also that because the opgraph's "amount" parameter defines a default value, it is not necessary for the <nodedef> to define a default, and that uses of this custom node in another opgraph do not need to provide an "amount" parameter; the default value will be used.

```
  <opgraph name="BlendAdd">
    <parameter name="fg" type="opgraphnode"/>
    <parameter name="bg" type="opgraphnode"/>
    <parameter name="amount" type="float" value="1.0"/>
    <multiply name="n1" type="color4">
      <parameter name="in" type="opgraphnode" value="$fg"/>
      <parameter name="amount" type="float" value="$amount"/>
    </multiply>
    <plus name="n2" type="color4">
      <parameter name="fg" type="opgraphnode" value="n1"/>
      <parameter name="bg" type="opgraphnode" value="$bg"/>
    </plus>
    <output name="o_result" type="color4">
      <parameter name="in" type="opgraphnode" value="n2"/>
    </output>
  </opgraph>
  <nodedef name="bladdc4" node="BlendAdd" type="color4">
    <parameter name="in1" type="opgraphnode" intype="color4"/>
    <parameter name="in2" type="opgraphnode" intype="color4"/>
    <parameter name="amount" type="float"/>
  </nodedef>
```

The `intype`s of the opgraphnode-type input parameters for the custom node are defined in the <nodedef>; they do not need to be defined in the opgraph parameter declarations.

Custom Node Use

Once defined with a <nodedef>, invoking a custom opgraph node within an opgraph looks very much the same as using any other standard opgraph node: the name of the element is the name of the custom operator, and the MaterialX type of the node's output is required; the custom operator element contains parameters defining connection of inputs to other opgraph node outputs as well as any parameter values for the custom operator.

```
  <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">
    <parameter name="in1" type="opgraphnode" value="custnoise1"/>
    <parameter name="in2" type="opgraphnode" value="n7"/>
    <parameter name="ColorA" type="color3" value="1.0, 1.0, 1.0"/>
    <parameter name="ColorB" type="color3" value="0.0, 0.0, 0.0"/>
  </mariBlend>
```

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

```
  <doublecolor name="dc1" type="multioutput">
    <parameter name="seed" type="float" value="0.442367"/>
  </doublecolor>
  <contrast name="n2" type="color3">
    <parameter name="in" type="opgraphnode" value="dc1" outputname="c1"/>
    <parameter name="amount" type="float" value="0.14"/>
  </contrast>
  <plus name="n4" type="color3">
    <parameter name="bg" type="opgraphnode" value="dc1" outputname="c2"/>
    <parameter name="fg" type="opgraphnode" value="n1"/>
  </plus>
```