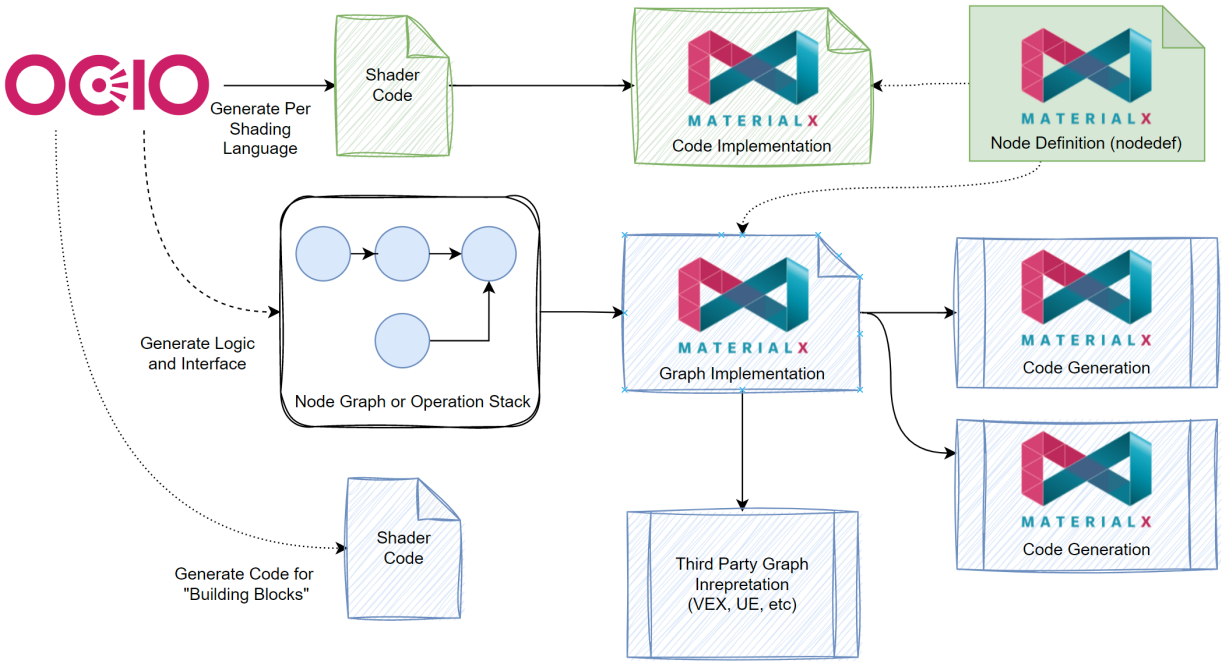
OCIO and MaterialX

This notebook will cover one workflow of using OCIO to generate code for MaterialX. We will be using the API based on OCIO 2.2 and above (released in October 2022).

The aim of this notebook is to go over how OCIO can be used to generate "implementations". The longer term MaterialX aim is to generate functional node graphs. This book will cover the setup required for generation, but can only show how source code implementation generation can be performed at the curren time.

The workflow covered is the "green" parts in this overall workflow diagram:



If / when graph generation is possible source code implementations can be swapped out for graph implementations.

The breakdown is as follows:

1. OCIO setup
2. Setting up OCIO configurations and getting available color spaces.
3. Setting up OCIO "processors" for generating color transforms
4. Generating source code implementations
5. Creating MaterialX implementations and definition wrappers for `color3` and `color4` variants.
6. Add definitions and implementations to the "standard" transform

library (`cmlib`)

For further OCI there is a fair bit of documentation available with a useful starting place here

Setup

OpenColorIO is available as a pre-built Python package on PyPi here

`pip` can be used to install the package which is called `PyOpenColorIO`

User can also build OpenColorIO by cloning the GitHub repository.

For the purposes of generating color transform implementation, most of the build options can be disabled. For For building and installing the Python package, make sure that the appropriate build option is set (current `OCIO_BUILD_PYTHON`, `OCIO_INSTALL_EXT_PACKAGES` respectively).

An example set of options is given here:

```
-DOPENIMAGEIO_INCLUDE_DIR="" -DOPENIMAGEIO_LIBRARY="" -DOCIO_BUILD_DOCS=OFF -
DBUILD_SHARED_LIBS=OFF -DOCIO_BUILD_TESTS=ON -DOCIO_BUILD_GPU_TESTS=OFF -
DOCIO_BUILD_PYTHON=1 -DOCIO_BUILD_JAVA=0 -DOCIO_INSTALL_EXT_PACKAGES=ALL -
DOCIO_BUILD_APPS=0 -DOCIO_BUILD_NUKE=0
```

A third alternative is to use a pre-built Python package which comes with another installation such as a DCC. A check should be made to not inadvertently use the incorrect version of the package.

For this book, we install from PyPi.

In [ ]:
```
# Package install
#pip install OpenColorIO
```

Here we import PyOpenColorIO and MaterialX.

In [ ]:
```
# Import OCIO package
import PyOpenColorIO as OCIO
import MaterialX as mx

print('OCIO version:', OCIO.GetVersion())
print('MaterialX version:', mx.getVersionString())
```

```
OCIO version: 2.3.0
MaterialX version: 1.38.8
```

Configurations

As of version 2.2, `ACES Cg Config` and `ACES Studio Config` are packaged with `OCIO`, meaning that they are available to use without having to download them separately. The `getBuildInConfigs()` API is explained here

In [ ]:
```
# Get the OCIO built in configs
registry = OCIO.BuiltinConfigRegistry().getBuiltinConfigs()
```

This items return canned be scanned and the appropriate configuration instantiated using `CreateFromBuiltInConfig()` In the following example we built a dictionary of configs along with the available color spaces.

In [ ]:
```
# Create a dictionary of configs
configs = {}
for item in registry:
    # The short_name is the URI-style name.
    # The ui_name is the name to use in a user interface.
```

```
    short_name, ui_name, isRecommended, isDefault = item

    # Don't present built-in configs to users if they are no longer recommended.
    if isRecommended:
        # Create a config using the Cg config
        config = OCIO.Config.CreateFromBuiltinConfig(short_name)
        colorSpaces = None
        if config:
            colorSpaces = config.getColorSpaces()

        if colorSpaces:
            configs[short_name] = [config, colorSpaces]

# Print the configs
for config in configs:
    print('Built-in config:', config)
    csnames = configs[config][0].getColorSpaceNames()
    print('- Number of color spaces: %d' % len(csnames))
    #for csname in csnames:
    #    print('  -', csname)
```

```
Built-in config: cg-config-v2.1.0_aces-v1.3_ocio-v2.3
- Number of color spaces: 15
Built-in config: studio-config-v2.1.0_aces-v1.3_ocio-v2.3
- Number of color spaces: 41
```

A more direct way to get the desired config is to call `CreateFomFile` with the appropriate built in path. In this case we get the `ACES Cg Config`.`

In [ ]:
```
acesCgConfigPath = 'ocio://cg-config-v1.0.0_aces-v1.3_ocio-v2.1'
builtinCfgC = OCIO.Config.CreateFromFile(acesCgConfigPath)
print('Built-in config:', builtinCfgC.getName())
csnames = builtinCfgC.getColorSpaceNames()
print('- Number of color spaces: %d' % len(csnames))
```

```
Built-in config: cg-config-v1.0.0_aces-v1.3_ocio-v2.1
- Number of color spaces: 14
```

Color Spaces

To check what color space identifiers can be used we print out each color space name along with any aliases by calling `getAliases()` on each color space.

In [ ]:
```
from IPython.display import display_markdown

title = '| Configuration | Color Space | Aliases |\n'
title = title + '| --- | --- | --- |\n'

rows = ''
for c in configs:
    config = configs[c][0]
    colorSpaces = configs[c][1]
    for colorSpace in colorSpaces:
        aliases = colorSpace.getAliases()
        rows = rows + '| ' + c + ' | ' + colorSpace.getName() + ' | ' + ', '.join(aliases) + ' |

md = '<details><summary>Color Spaces</summary>\n\n' + title + rows + '</details>'
display_markdown(md, raw=True)
```

▶ Color Spaces

Supported Color Spaces in MaterialX

MaterialX currently uses color space names for :

1. Color space (`colorspace` attribute) tagging for input images (`filename` attributes) and colors (`color3` and `color4` types).
2. Node category identifiers for node definitions of the form `<from color space>_<to color space name>` to specify color space conversion nodes. (Node definitions are support as of MaterialX 1.38.7)

Note that any valid color space name can be used for input tagging.

At time of writing only specific color space conversions are supported via node definitions and hence can perform code injection during shader code generation. Any color space information can still be passed as meta-data to the code generated (e.g. as is possible with the `OSL` generator).

Further note that only certain **aliases** which are valid MaterialX identifiers are recognized in this context. For example `g18_rec709` is used for color space `Gamma 1.8 Rec.709 - Texture` and `lin_rec709` is used for color space `Linear Rec.709 (sRGB)` .

OCIO Shader Code Generation

It is possible to generate code color space transforms for certain code generation targets.

This is done by:

1. Calling `getProcessor()` on the config with desired "source" and "destination" color spaces for the transform.
2. Creating a CPU or GPU processor
3. Set the appropriate target language
4. Getting the shader code using `getShaderText()`

In [ ]:
```
def generateShaderCode(config, sourceColorSpace, destColorSpace, language):
    cshaderCodee = ''
    if not config:
        return shaderCode

    # Create a processor for a pair of colorspaces (namely to go to linear)
    processor = None
    try:
        processor = config.getProcessor(sourceColorSpace, destColorSpace)
    except:
        return shaderCode

    gpuProcessor = None
    if processor:
        gpuProcessor = processor.getDefaultGPUProcessor()
    if gpuProcessor:
        shaderDesc = OCIO.GpuShaderDesc.CreateShaderDesc()
        if shaderDesc:
            shaderDesc.setLanguage(language)
            gpuProcessor.extractGpuShaderInfo(shaderDesc)
            shaderCode = shaderDesc.getShaderText()

    return shaderCode
```

```
# Use GLSL as the shader language to produce, and linear as the target color space
language = OCIO.GpuLanguage.GPU_LANGUAGE_GLSL_4_0
targetColorSpace = 'lin_rec709'

# Go through all the config and create code for each transform

title = '| Source | Target | Code |\n'
title = title + '| --- | --- | --- |\n'

rows = ''
testedSources = set()
for c in configs:
    config = OCIO.Config.CreateFromBuiltinConfig(c)
    colorSpaces = config.getColorSpaces()
    for colorSpace in colorSpaces:
        colorSpaceName = colorSpace.getName()
        # Skip if the colorspace is already tested
        if colorSpaceName in testedSources:
            continue
        testedSources.add(colorSpaceName)

        code = generateShaderCode(config, colorSpace.getName(), targetColorSpace, language)
        code = code.replace('\n', '<br>')
        code = '<code>' + code + '</code>'
        rows = rows + '| ' + colorSpace.getName() + ' | ' + targetColorSpace + ' | ' + code + '|

md = '<details><summary>Transform Code for GLSL</summary>\n\n' + title + rows + '</details>'
display_markdown(md, raw=True)
```

▶ Transform Code for GLSL

Integrating OCIO with MaterialX

We will pick an example transform to go over details on mapping from OCIO to MaterialX.

The first thing of note is OCIO function signatures

- Currently all signatures transform 4 channel color inputs while MaterialX supports both 3 and 4 channel variants. This can be easily handled by adding pre and post conversion nodes, or by creating variant function signatures. The former is more robust and more in line with the proposed direction to have all OCIO transforms to be represented as graphs.
- The signature name is not unique. This can be handled as OCIO provides mechanism to override the function names using `setFunctionName` and `setResourcePrefix`.

Following the current MaterialX convention we use the signature notation:
`mx_<sourceName>_to_<targetname>_<type>` where `type` is either `color3` or `color4` for 3 or 4 channel variants.

We add in two new utilities:

1. `createTransformName` which will generate the unique function name
2. `setShaderDescriptionParameters` which overrides the function name but also adds a prefix to uniquely identify dependent resources.

These are then used in a new code generation variation called `generateShaderCode2()` which has additionally been modified to return the number of dependent texture resources, which can be queried from the shader descriptor via the `GpuShaderDesc.getTextures()` iterator.

In [ ]:
```
def createTransformName(sourceSpace, targetSpace, typeName):
    transformFunctionName = "mx_" + mx.createValidName(sourceSpace) + "_to_" + targetSpace + "_"
    return transformFunctionName

def setShaderDescriptionParameters(shaderDesc, sourceSpace, targetSpace, typeName):
    transformFunctionName = createTransformName(sourceSpace, targetSpace, typeName)
    shaderDesc.setFunctionName(transformFunctionName)
    shaderDesc.setResourcePrefix(transformFunctionName)

def generateShaderCode2(config, sourceColorSpace, destColorSpace, language):
    shaderCode = ''
    textureCount = 0
    if not config:
        return shaderCode, textureCount

    # Create a processor for a pair of colorspaces (namely to go to linear)
    processor = None
    try:
        processor = config.getProcessor(sourceColorSpace, destColorSpace)
    except:
        print('Failed to generated code for transform: %s -> %s' % (sourceColorSpace, destColorS
        return shaderCode, textureCount

    if processor:
        gpuProcessor = processor.getDefaultGPUProcessor()
        if gpuProcessor:
            shaderDesc = OCIO.GpuShaderDesc.CreateShaderDesc()
            if shaderDesc:
                try:
                    shaderDesc.setLanguage(language)
                    if shaderDesc.getLanguage() == language:
                        setShaderDescriptionParameters(shaderDesc, sourceColorSpace, destColorSp
                        gpuProcessor.extractGpuShaderInfo(shaderDesc)
                        shaderCode = shaderDesc.getShaderText()
                        for t in shaderDesc.getTextures():
                            textureCount += 1
                except OCIO.Exception as err:
                    print(err)

    return shaderCode, textureCount
```

OCIO Resource Dependencies

Resource dependencies is a second major issue to examine.

In the example below we convert two different source color spaces.

One is "self-contained" in that there are no support functions being produced (`ACES2065-1`), while the second adds additional function and resources. Note that we maintain uniqueness of these additions by using `setFunctionName` and `setResourcePrefix` respectively.

Example 1: Self-contained

In [ ]:
```
sourceColorSpace = 'ACES2065-1' # "acescg"
textureCount = 0
code = ''
code, textureCount = generateShaderCode2(builtinCfgC, sourceColorSpace, targetColorSpace, langua
if code:
    code = code.replace("// Declaration of the OCIO shader function\n",
                        "// " + sourceColorSpace + " to " + targetColorSpace + " function. Textu
    code = '```c++\n' + code + '\n```'
    display_markdown(code, raw=True)
```

```
// ACES2065-1 to lin_rec709 function. Texture count: 0

vec4 mx_ACES2065_1_to_lin_rec709_color4(vec4 inPixel)
{
 vec4 outColor = inPixel;

 // Add Matrix processing

 {
  vec4 res = vec4(outColor.rgb.r, outColor.rgb.g, outColor.rgb.b, outColor.a);
  vec4 tmp = res;
  res = mat4(2.5216861867438798, -0.27647991422992202, -0.015378064966034201, 0., -1.1341309882397
199, 1.37271908766826, -0.152975335867399, 0., -0.38755519850416398, -0.096239173438334005, 1.168353
40083343, 0., 0., 0., 0., 1.) * tmp;
  outColor.rgb = vec3(res.x, res.y, res.z);
  outColor.a = res.w;
 }

 return outColor;
}
```

Example 2: Secondary Dependencies

In [ ]:
```
sourceColorSpace = 'ACEScc' # "acescg"
code, textureCount = generateShaderCode2(builtinCfgC, sourceColorSpace, targetColorSpace, langua
if code:
    code = code.replace("// Declaration of the OCIO shader function\n",
                        "// " + sourceColorSpace + " to " + targetColorSpace + " function. Textu
    code = '```c++\n' + code + '\n```\n'

    md = '<details><summary>Secondary Dependency Sample Code</summary>\n\n' + code + '</details>
    display_markdown(md, raw=True)
```

▶ Secondary Dependency Sample Code

Issues With Texture Resources

From an integration point of view any introduction of texture lookups requires resource declarations in the code. (such as the `uniform sampler2D mx_ACEScc_to_lin_rec709_color4_ocio_lut1d_0Sampler;` sampler declaration).

1. The only way to handle these is to have additional logic added for code insertion of color transforms, such that the shader function declarations and resources can be inserted into the code independently. The current MaterialX code generation logic does not otherwise support this using the "default color system".

    > Note : An experiment was attempted previously but does not align with the current proposal to have stand-alone node definitions. It was thus abandoned. (For those interested the full code with code changes can be found here). Here 1D lookups (LUTS) were specified as input arrays and code generation created 1D textures dynamically based on the array inputs. 3D lookups were not handled.

2. **From the point of view of creating node graphs, any implementation resource dependencies means it cannot be cleanly wrapped up into a self-contained node definition and implementation.**

For now these can be "skipped" until such time as they are require, or the implementation changes to avoid using these.

Finding Transforms Using Texture Resources

We can re-iterate through all of the transforms of interest, and find these transforms using the following code.

> Note that the code generation is not necessary but is written this way to

reuse the existing utility `generateShaderCode2`).

In [ ]:
```
# Scan through all the color spaces on the configs to check for texture resource usage.
testedSources = set()
for c in configs:
    config = OCIO.Config.CreateFromBuiltinConfig(c)
    colorSpaces = config.getColorSpaces()
    for colorSpace in colorSpaces:
        colorSpaceName = colorSpace.getName()
        # Skip if the colorspace is already tested
        if colorSpaceName in testedSources:
            continue
        testedSources.add(colorSpaceName)

        # Test for texture resource usage
        code, textureCount = generateShaderCode2(config, colorSpace.getName(), targetColorSpace,
        if textureCount:
            print('- Transform "%s" to "%s" requires %d texture resources' % (colorSpace.getName
```

```
- Transform "ACEScc" to "lin_rec709" requires 1 texture resources
- Transform "ADX10" to "lin_rec709" requires 1 texture resources
- Transform "ADX16" to "lin_rec709" requires 1 texture resources
- Transform "CanonLog2 CinemaGamut D55" to "lin_rec709" requires 1 texture resources
- Transform "CanonLog3 CinemaGamut D55" to "lin_rec709" requires 1 texture resources
```

Target Language Support

At time of writing the target languages supported by OCIO and MaterialX differ. This includes non-core support such as `MDL` and current versions of `OSL` . Also as no logical operators are provided as with MaterialX, targets such as `Vex` which parses and maps MaterialX nodes as operators is not easy to do.

OCIO and MaterialX recently added in `Metal` language support. It is to be checked if there would be any issues with the additional `struct` wrappers required for this language as it is uncommon for MaterialX code generation to call into a `struct` function at the current time.

In [ ]:
```
sourceColorSpace = "acescg"
language = OCIO.GpuLanguage.GPU_LANGUAGE_MSL_2_0
code, textureCount = generateShaderCode2(builtinCfgC, sourceColorSpace, targetColorSpace, langua
if code:
    code = code.replace("// Declaration of the OCIO shader function\n", "// " + sourceColorSpace
    code = '```c++\n' + code + '\n```\n'
    md = '<details><summary>MSL struct usage</summary>\n\n' + code + '</details>'
    display_markdown(md, raw=True)
```

▶ MSL struct usage

Using version 2.3 to access `OSL` there appears to be additional issues with the code generated as additional utility functions may be inserted which are not renamed to avoid collisions.

For example functions called `max()`, `pow()` etc are added which are outside the scope of the main shader declaration for which do not seem to be included in the logic for unique function name. As well as include additional include files which should be part of the OSL distribution. These need to be stripped out to embed this code as part of a larger OSL shader which already includes these functions to avoid name clashes. An issue has been logged for this.

As `OSL` integrations will generally perform color management outside of the shader, it is to be seen if this is important enough to address.

In [ ]:
```
if OCIO.GpuLanguage.LANGUAGE_OSL_1:
    sourceColorSpace = "acescg"
    language = OCIO.GpuLanguage.LANGUAGE_OSL_1
    code, textureCount = generateShaderCode2(builtinCfgC, sourceColorSpace, targetColorSpace, la
    if code:
        # Bit of ugly patching to make the main function name consistent.
        transformName = createTransformName(sourceColorSpace, targetColorSpace, 'color4')
        code = code.replace('OSL_' + transformName, '__temp_name__')
        code = code.replace(transformName, transformName + '_impl')
        code = code.replace('__temp_name__', transformName)
        code = code.replace("// Declaration of the OCIO shader function\n", "// " + sourceColorS
        code = '```c++\n' + code + '\n```\n'
        md = '<details><summary>OSL dependent function / includes code</summary>\n\n' + code + '
        display_markdown(md, raw=True)
```

▶ OSL dependent function / includes code

Creating MaterialX Node Definitions / Implementation

Given source code for now, it is still possible to create implementations and node definitions. If in the future the implementations can become MaterialX node graphs then the definition interface can still be used.

To create a new definition:

1. The source and target color space is used to derive:
   - a transform name: using the previously described `createTransformName()` utility
   - a node name by replace the 'mx_' function name with the "standard" 'ND_' prefix used for definition

- a node category

2. Use `addNodeDef()` API to add a new definition
3. Set node group to be `colortransform` which is the recommended group for new color transforms.
4. Add a single input and output of the appropriate type (`color3` or `color4`).
5. Set a default value on the input. For this code we assume the defaul to be opaque black.

Thie logic is encapsulated in a new `generateMaterialXDefinition()` utility function.

In [ ]:
```
def generateMaterialXDefinition(doc, sourceColorSpace, targetColorSpace, inputName, type):

    # Create a definition
    transformName = createTransformName(sourceColorSpace, targetColorSpace, type)
    nodeName = transformName.replace('mx_', 'ND_')

    comment = doc.addChildOfCategory('comment')
    comment.setDocString(' Color space %s to %s transform. Generated via OCIO. ' % (sourceColorS

    definition = doc.addNodeDef(nodeName, 'color4')
    category = sourceColorSpace + '_to_' + targetColorSpace
    definition.setNodeString(category)
    definition.setNodeGroup('colortransform')

    defaultValueString = '0.0 0.0 0.0 1.0'
    defaultValue = mx.createValueFromStrings(defaultValueString, 'color4')
    input = definition.addInput(inputName, type)
    input.setValue(defaultValue)
    output = definition.getOutput('out')
    output.setAttribute('default', 'in')

    return definition
```

Another utility called `writeShaderCode()` is used to write the code to file following the "standard" MaterialX naming convention.

In [ ]:
```
def writeShaderCode(code, transformName, extension, target):

    # Write source code file
    filename = mx.FilePath('./data') / mx.FilePath(transformName + '.' + extension)
    print('Write target[%s] source file %s' % (target,filename.asString()))
    f = open(filename.asString(), 'w')
    f.write(code)
    f.close()
```

The implementation creation logic is encapsulated in a `createMaterialXImplementation()` utility function which appends a new implementation to an existing Document.

The transform name is assumged to be precreated using `createTransformName()` .

This is used to create a unique implementation Element name, and source code filename. We decied to use a file on disk as it is not possible to inline 1 or more function functions created by OCIO.

The implementation is associated with an existing node definition which is passed in and a target is explicit set to indicate which shading language (target) the implementation is for.

In [ ]:
```
def createMaterialXImplementation(doc, definition, transformName, extension, target):
    '''
    Create a new implementation in a document for a given definition.
```

```
    '''
    implName = transformName + '_' + target
    filename = transformName + '.' + extension
    implName = implName.replace('mx_', 'IM_')

    # Check if implementation already exists
    impl = doc.getImplementation(implName)
    if impl:
        print('Implementation already exists: %s' % implName)
        return impl

    comment = doc.addChildOfCategory('comment')
    comment.setDocString(' Color space %s to %s transform. Generated via OCIO for target: %s'
                         % (sourceColorSpace, targetColorSpace, target))
    impl = doc.addImplementation(implName)
    impl.setFile(filename)
    impl.setFunction(transformName)
    impl.setTarget(target)
    impl.setNodeDef(definition)

    return impl
```

Finally a small utility is added to write the document to disk.

In [ ]:
```
def writeDocument(doc, filename):
    print('Write MaterialX file:', filename.asString())
    mx.writeToXmlFile(doc, filename)
```

Using these utilities we:

- Create separate definition and implementation Documents.
- Generate shader code for `GLSL`, `MSL`, and `OSL`` for the same color transform. (OSL` is available in version 2.3).
- Create a new definition for that transform
- Create a new implementation for each shader code result

In [ ]:
```
# Example to create:
# - source code for a given transform for 2 shader targets
# - A definition wrapper for the source
# - An implementations per source code. All implementations are associated with single definitio
definitionDoc = mx.createDocument()
definition = None

implDoc = mx.createDocument()

sourceColorSpace = "acescg"
type = 'color4'
transformName = createTransformName(sourceColorSpace, targetColorSpace, type)

# All code has the same input name
# It is possible to use a different name than the name used in the generated function ('inPixel'
#IN_PIXEL_STRING = 'inPixel'
IN_PIXEL_STRING = 'in'

# Pick a source and target color space
sourceColorSpace = 'acescg'
targetColorSpace = 'lin_rec709'

# List of MaterialX target language, source code extensions, and OCIO GPU languages
```

```
generationList = [['genmsl', 'metal', OCIO.GpuLanguage.GPU_LANGUAGE_MSL_2_0],
                  ['genglsl', 'glsl', OCIO.GpuLanguage.GPU_LANGUAGE_GLSL_4_0] ]

if OCIO.GpuLanguage.LANGUAGE_OSL_1:
    generationList.append(['genosl', 'osl', OCIO.GpuLanguage.LANGUAGE_OSL_1])

for gen in generationList:
    target = gen[0]
    extension = gen[1]
    language = gen[2]

    code, textureCount = generateShaderCode2(builtinCfgC, sourceColorSpace, targetColorSpace, la
    if code:
        # Emit the source code file
        writeShaderCode(code, transformName, extension, target)

        # Create the definition once
        if not definition:
            definition = generateMaterialXDefinition(definitionDoc, sourceColorSpace, targetColo
                                                     IN_PIXEL_STRING, type)

        # Create the implementation
        createMaterialXImplementation(implDoc, definition, transformName, extension, target)
```

```
Write target[genmsl] source file .\data\mx_acescg_to_lin_rec709_color4.metal
Write target[genglsl] source file .\data\mx_acescg_to_lin_rec709_color4.glsl
Write target[genosl] source file .\data\mx_acescg_to_lin_rec709_color4.osl
```

The resulting MaterialX wrappers are then written to disk as follows:

In [ ]:

```
# Write the definition document
filename = mx.FilePath('./data') / mx.FilePath(definition.getName() + '.' + 'mtlx')
print('Write MaterialX definition file:', filename.asString())
mx.writeToXmlFile(definitionDoc, filename)

# Write the implementation document
implFileName = mx.FilePath('./data') / mx.FilePath('IM_' + transformName + '.' + 'mtlx')
print('Write MaterialX implementation file:', implFileName.asString())
result = mx.writeToXmlFile(implDoc, implFileName)

# Emit the results for display
result = mx.writeToXmlString(definitionDoc)
display_markdown('#### Generated MaterialX Definition\n' + '```xml\n' + result + '```\n', raw=Tru
result = mx.writeToXmlString(implDoc)
display_markdown('#### Generated MaterialX Implementations\n' + '```xml\n' + result + '```\n', r
```

```
Write MaterialX definition file: .\data\ND_acescg_to_lin_rec709_color4.mtlx
Write MaterialX implementation file: .\data\IM_mx_acescg_to_lin_rec709_color4.mtlx
```

Generated MaterialX Definition

```
<?xml version="1.0"?>
<materialx version="1.38">
  <!-- Color space acescg to lin_rec709 transform. Generated via OCIO. -->
  <nodedef name="ND_acescg_to_lin_rec709_color4" node="acescg_to_lin_rec709" nodegroup="colortransform">
    <output name="out" type="color4" default="in" />
    <input name="in" type="color4" value="0, 0, 0, 1" />
  </nodedef>
</materialx>
```

Generated MaterialX Implementations

```
<?xml version="1.0"?>
<materialx version="1.38">
  <!-- Color space acescg to lin_rec709 transform. Generated via OCIO for target: genmsl-->
  <implementation name="IM_acescg_to_lin_rec709_color4_genmsl" file="mx_acescg_to_lin_rec709_color4.metal" function="mx_acescg_to_lin_rec709_color4" target="genmsl" nodedef="ND_acescg_to_lin_rec709_color4" />
  <!-- Color space acescg to lin_rec709 transform. Generated via OCIO for target: genglsl-->
  <implementation name="IM_acescg_to_lin_rec709_color4_genglsl" file="mx_acescg_to_lin_rec709_color4.glsl" function="mx_acescg_to_lin_rec709_color4" target="genglsl" nodedef="ND_acescg_to_lin_rec709_color4" />
  <!-- Color space acescg to lin_rec709 transform. Generated via OCIO for target: genosl-->
  <implementation name="IM_acescg_to_lin_rec709_color4_genosl" file="mx_acescg_to_lin_rec709_color4.osl" function="mx_acescg_to_lin_rec709_color4" target="genosl" nodedef="ND_acescg_to_lin_rec709_color4" />
</materialx>
```

Variant Creation

Given a node definition for the `color4` variant is is possible to create a functional graph and corresponding definition for a `color3` variant. The graph simple convertes from `color3` to `color4` before connecting to the transform and then converts back to `color3` for output.

In [ ]:
```
color4Name = definition.getName()
color3Name = color4Name.replace('color4', 'color3')
color3Def = definitionDoc.addNodeDef(color3Name)
color3Def.copyContentFrom(definition)
c3input = color3Def.getInput(IN_PIXEL_STRING)
c3input.setType('color3')
c3input.setValue(mx.createValueFromStrings('0.0 0.0 0.0', 'color3'))

ngName = color3Def.getName().replace('ND_', 'NG_')
ng = definitionDoc.addNodeGraph(ngName)
c4instance = ng.addNodeInstance(definition)
c4instance.addInputsFromNodeDef()
c4instanceIn = c4instance.getInput(IN_PIXEL_STRING)
c3to4 = ng.addNode('convert', 'c3to4', 'color4')
c3to4Input = c3to4.addInput('in', 'color3')
c4to3 = ng.addNode('convert', 'c4to3', 'color3')
```

```
c4to3Input = c4to3.addInput('in', 'color4')
ngout = ng.addOutput('out', 'color3')
#ngin = ng.addInput('in', 'color3')
ng.setNodeDef(color3Def)

c4instanceIn.setNodeName(c3to4.getName())
c4to3Input.setNodeName(c4instance.getName())
ngout.setNodeName(c4to3.getName())
c3to4Input.setInterfaceName(IN_PIXEL_STRING)

result = mx.writeToXmlString(definitionDoc)
display_markdown('#### Generated Color3 Variant\n' + '```xml\n' + result + '```\n', raw=True)

valid, log = definitionDoc.validate()
if not valid:
    print('Document created is invalid:', log)

filename = mx.FilePath('./data') / mx.FilePath(definition.getName() + '_2.' + 'mtlx')
print('Write MaterialX definition variant file:', filename.asString())
mx.writeToXmlFile(definitionDoc, filename)
```

Generated Color3 Variant

```
<?xml version="1.0"?>
<materialx version="1.38">
  <!-- Color space acescg to lin_rec709 transform. Generated via OCIO. -->
  <nodedef name="ND_acescg_to_lin_rec709_color4" node="acescg_to_lin_rec709" nodegroup="colortransform">
    <output name="out" type="color4" default="in" />
    <input name="in" type="color4" value="0, 0, 0, 1" />
  </nodedef>
  <nodedef name="ND_acescg_to_lin_rec709_color3" node="acescg_to_lin_rec709" nodegroup="colortransform">
    <output name="out" type="color3" />
    <input name="in" type="color3" value="0, 0, 0" />
  </nodedef>
  <nodegraph name="NG_acescg_to_lin_rec709_color3" nodedef="ND_acescg_to_lin_rec709_color3">
    <acescg_to_lin_rec709 name="node1" type="color4" nodedef="ND_acescg_to_lin_rec709_color4">
      <input name="in" type="color4" value="0, 0, 0, 1" nodename="c3to4" />
    </acescg_to_lin_rec709>
    <convert name="c3to4" type="color4">
      <input name="in" type="color3" interfacename="in" />
    </convert>
    <convert name="c4to3" type="color3">
      <input name="in" type="color4" nodename="node1" />
    </convert>
    <output name="out" type="color3" nodename="c4to3" />
  </nodegraph>
</materialx>
```

```
Write MaterialX definition variant file: .\data\ND_acescg_to_lin_rec709_color4_2.mtlx
```

MaterialX Standard Library Inclusion

It is possible to add a new color space transform to the "standard" MaterialX library locations. This can be done for local testing for in some cases when additional search paths are not supported.

Introduced with version **1.38.7**, the location of definitions and implementations can be found in the `cmlib` folder under the installation `libraries` location. Note that target specific source file directories were removed as all implementations are now target independent node graphs.

To include source file implementations the appropriate `target` folders can be added in. Under that folder an implementation file can be added for each target along with associated source files. The new definition can be added to `cmlib_defs.mtlx.

See the Creating Definitions book for more on how set up new definitions.

The standard library folder structure is partially shown below. The bolded items would be the ones of interest. The `stdlib` folder is also shown to show how each folder follows the same naming and hierarchy conventions.

- cmlib // Color space transform library
  - **cmlib_defs.mtlx** // definitions file
  - cmlib_ng.mtlx // functional node graphs file
  - **genglsl** // GLSL implementation folder
    - **cmlib_genglsl_impl.mtlx** // Implementation declarations
    - **GLSL files reside here**
  - **genmsl** // MSL implementation folder
    - **cmlib_genmsl_impl.mtlx** // Implementation declarations
    - **MSL files reside here**
- stdlib // "Standard" node library
  - genglsl
    - stdlib_genglsl_impl.mtlx
  - genmdl
  - genmsl
    - stdlib_genmsl_impl.mtlx
  - genosl
  - targets

Future Exploration

As mentioned the notion of representing OCIO shader logic as MaterialX nodes is still under discussion. There are thoughts to try and export each lower level functional "block" as a reusable functions with the appripriate argument values, hence allowing for a custom OCIO code generator to emit a new MaterialX node for each function.

At time of writing the ASWF OCIO techincal steering committee (TSC) additionally has on it's roadmap to reduce the size of OCIO and consider how it could be deployed in a Web or mobile environment. Here it is possible to consider both a source code or a node graph based option. The latter could fit in with the interest to transport some subset of MaterialX via formats such as `glTF`.