UsdPreviewSurface Proposal

 Revision 2.2

Goal

The goal of this proposal is to have a preview material with a basic set of nodes that can be used to interchange assets from one platform to another in film and game pipelines, and which are supported natively by the production renderers that ship with USD/Hydra. The actual preview surface is intended to provide a solution for most situations and to move assets across multiple environments, the surface can work with metallic and specular workflows.

You will notice we have left out concepts like subsurface, anisotropic specular highlights, cloth shaders, just to mention a few. For these cases, we recommend creating a more specific surface. Our assessment is that the industry is approaching a more flexible, non-ubersurface, style of describing surfaces, as hinted by NVIDIA's MDL, but that industry-wide adoption, especially for realtime game engines, is still a ways off. We are excited for USD/UsdShade to support such efforts (and UsdShade should already be fully capable of representing such shading networks). But our goal with this proposal is to promote interchange in the industry starting in 2018, thus this first version of a preview surface is a more constrained ubersurface that reflects what is interchangeable around 2018.

Our starting requirements:

- Rather than design a single node like the Alembic Preview Material, we wanted to design a preview surface that is friendly to network shading in UsdShade, so that for clients that can support it, any signal input to the surface could be driven by pattern networks. This guided us to design a "minimally complete" suite of four nodes to allow primvar and texture-driven inputs to the surface.
- We wanted to ensure our design would promote reliable interchange between offline/real-time renderers and content creation tools. We examined a number of other specifications, and we believe the preview surface can match up reasonably well.
- Although much of the game industry relies on the *metallic workflow*, many packages also support the more expressive *specular workflow* ; given that several of the above-named applications support both in one form or another, and since we want to use this preview surface internally as well and use the specular workflow for our preview shading, we designed a surface that supports both workflows.

Core Nodes

Preview Surface

The UsdPreviewSurface is meant to model a "modern" physically based surface that strikes a balance between expressiveness and reliable interchange between current day DCC's and game engines and other real-time rendering clients. We expect it to eventually evolve in a versioned way, as the state of the industry evolves.

This preview surface supports both "workflows", *specular* and *metalness*.

Node Id

- **UsdPreviewSurface**

Inputs (name - type - fallback)

- **diffuseColor - color3f - (0.18, 0.18, 0.18)**
  When using metallic workflow this is interpreted as albedo.
- **emissiveColor - color3f - (0.0, 0.0, 0.0)**
  Emissive component.

- **useSpecularWorkflow - int - 0**
  This node can fundamentally operate in two modes : *Specular workflow* where you provide a texture/value to the "specularColor" input. Or, *Metallic workflow* where you provide a texture/value to the "metallic" input. Depending on the 0 or 1 value of this parameter, the following parameters are conditionally enabled:
  - *useSpecularWorkflow = 1: (Specular workflow)*
    - **specularColor - color3f - (0.0, 0.0, 0.0)**
      Specular color to be used. This is the color at 0 incidence. Edge color is assumed white. Transition between the two colors according to Schlick fresnel approximation.
  - *useSpecularWorkflow = 0: (Metalness workflow)*
    - **metallic - float - 0.0**
      Use 1 for metallic surfaces and 0 for non-metallic.
      - If metallic is 1.0, then F0 (reflectivity at 0 degree incidence) will be derived from ior ((1-*ior*)/(1+*ior*))^2, then multiplied by *Albedo*; while edge F90 reflectivity will simply be the *Albedo*.
      (As an option, you can set *ior* to 0 such that F0 becomes equal to F90 and thus the *Albedo*).
      - If metallic is 0.0, then *Albedo* is ignored; F0 is derived from *ior* and F90 is white. In between, we interpolate.
- **roughness - float - 0.5**
  Roughness for the specular lobe. The value ranges from 0 to 1, which goes from a perfectly specular surface at 0.0 to maximum roughness of the specular lobe. This value is usually squared before use with a GGX or Beckmann lobe.
- **clearcoat - float - 0.0**
  Second specular lobe amount. The color is white.
- **clearcoatRoughness - float - 0.01**
  Roughness for the second specular lobe.
- **opacity - float - 1.0**
  When *opacity* is 1.0 then the prim is fully opaque, if it is smaller than 1.0 then the prim is translucent, when it is 0 the prim is transparent. Note that even a fully transparent object still receives lighting as, for example, perfectly clear glass still has a specular response.
- **opacityThreshold - float - 0.0**
  The opacityThreshold input is useful for creating geometric cut-outs based on the opacity input. A value of 0.0 indicates that no masking is applied to the opacity input, while a value greater than 0.0 indicates that rendering of the surface is limited to the areas where the opacity is greater than that value. A classic use of opacityThreshold is to create a leaf from an opacity input texture, in that case the threshold determines the parts of the opacity texture that will be fully transparent and not receive lighting. Note that when *opacityThreshol* is greater than zero, then *opacity* modulates the **presence** of the surface, rather than its transparency - pathtracers might implement this as allowing ((1 - *opacity*) * 100) % of the rays that do intersect the object to instead pass through it unhindered, and rasterizers may interpret *opacity* as pixel coverage. Thus, *opacityThreshold* serves as a switch for how the *opacity* input is interpreted; this "translucent **o r** masked" behavior is common in engines and renderers, and makes the *UsdPreviewSurface* easier to interchange. It does imply, however, that it is not possible to faithfully recreate a glassy/translucent material that also provides an opacity-based mask... so no single-polygon glass leaves.
- **ior - float - 1.5**
  Index of Refraction to be used for translucent objects.
- **normal - normal3f - (0.0, 0.0, 1.0)**
  Expects normal in tangent space [(-1,-1,-1), (1,1,1)] This means your texture reader implementation should provide data to this node that is properly scaled and ready to be consumed as a tangent space normal.
- **displacement - float - 0.0**
  Displacement in the direction of the normal.
- **occlusion - float - 1.0**
  Extra information about the occlusion of different parts of the mesh that this material is applied to. Occlusion only makes sense as a surface-varying signal, and pathtracers will likely choose to ignore it. An occlusion value of 0.0 means the surface point is fully occluded by other parts of the surface, and a value of 1.0 means the surface point is completely unoccluded by other parts of the surface.

Outputs (name - type)

In UsdShade, by convention and limitation of Usd/SdfLayer's native representable types, we assign the SdfValueTypeName type '*token*' to all inputs and outputs of "rich types" (e.g. structs), while allowing use of *renderType* metadata (a string) on UsdShadeInput and UsdShadeOutput to carry typeName information that may be useful to a renderer or shading system.

- **surface - token**
- **displacement - token**

```
def Shader "UsdPreviewSurface" (
    doc = "Preview surface specification"
    sdrMetadata = {
       token role = "surface"
    }
)
{
    uniform token info:id = "UsdPreviewSurface"

    # Outputs
    token outputs:surface
    token outputs:displacement
```

```
    # Inputs
    color3f inputs:diffuseColor = (0.18, 0.18, 0.18) (
        doc = """Parameter used as diffuseColor when using the specular
                  workflow, when using metallic workflow this is interpreted
                  as albedo."""
    )

    color3f inputs:emissiveColor = (0.0, 0.0, 0.0) (
        doc = """Emissive component."""
    )

    int inputs:useSpecularWorkflow = 0 (
        connectability = "interfaceOnly"
        doc = """This node can fundamentally operate in two modes :
            Specular workflow where you provide a texture/value to the
            "specularColor" input. Or, Metallic workflow where you
            provide a texture/value to the "metallic" input."""
    )

    color3f inputs:specularColor = (0.0, 0.0, 0.0) (
        doc = """Used only in the specular workflow.
             Specular color to be used.
             This is the color at 0 incidence. Edge color is assumed white.
             Transition between the two colors according to Schlick fresnel
             approximation."""
    )

    float inputs:metallic = 0.0 (
        doc = """Used only in the metalness workflow.
            1 for metallic surfaces and 0 for non-metallic.
            - If metallic is 1, then F0 (reflectivity at 0 degree incidence)
            will be derived from ior ( (1-ior)/(1+ior) )^2, then multiplied by albedo;
            while edge F90 reflectivity will simply be the albedo.
            (As an option, you can put ior to 0 such that F0 comes equal to F90 and thus the
albedo).
            - If metallic is 0, then albedo is ignored; F0 is derived from ior and F90 is white.
            In between, we interpolate."""
    )

    float inputs:roughness = 0.5 (
        doc = """Roughness for the specular lobe. The value ranges from 0 to 1,
        which goes from a perfectly specular surface at 0.0 to maximum roughness
        of the specular lobe. This value is usually squared before use with a
        GGX or Beckmann lobe."""
    )

    float inputs:clearcoat = 0.0 (
        doc = """Second specular lobe amount. The color is white."""
    )

    float inputs:clearcoatRoughness = 0.01 (
        doc = """Roughness for the second specular lobe."""
    )

    float inputs:opacity = 1.0 (
        doc = """Opacity of the material."""
    )


    float inputs:opacityThreshold = 0.0 (
        connectability = "interfaceOnly"
        doc = """Threshold used to determine opacity values that will be considered fully
transparent."""
    )

    float inputs:ior = 1.5 (
        doc = """Index of Refraction to be used for translucent objects."""
    )

    normal3f inputs:normal = (0.0, 0.0, 1.0) (
```

```
        doc = """Expects normal in tangent space [(-1,-1,-1), (1,1,1)]
            This means your texture reader implementation should provide
            data to this node that is properly scaled and ready
            to be consumed as a tangent space normal."""
    )

    float inputs:displacement = 0.0 (
        doc = """Displacement in the direction of the normal. """
    )

    float inputs:occlusion = 1.0 (
        doc = """Occlusion signal. This provides extra information about the
        occlusion of different parts of the mesh that this material is applied
        to.  Occlusion only makes sense as a surface-varying signal, and
        pathtracers will likely choose to ignore it.  An occlusion value of 0.0
        means the surface point is fully occluded by other parts of the surface,
        and a value of 1.0 means the surface point is completely unoccluded by
        other parts of the surface. """
    )
}
```

Texture Reader

Node that can be used to read UV textures, including tiled textures such as Mari UDIM's.

> **UDIM Tiling Constraint**
>
> To keep interchange simple(r) and to aid in efficient processing, **we stipulate a maximum of ten tiles in the U direction** for UDIM.

Node Id

- **UsdUVTexture**

Inputs (name - type - fallback)

- **file - asset - <EMPTY STRING>**
  Path to the texture. Following the 1.36 MaterialX spec, Mari UDIM substitution in *file* values uses the "<UDIM>" token, so for example in USD, we might see a value @textures/occlusion.<UDIM>.tex@
- **st - float2 - (0.0, 0.0)**
  Texture coordinate to use to fetch this texture. This node defines a mathematical/cartesian mapping from st to uv to image space: the (0, 0) *st* coordinate maps to a (0, 0) *uv* coordinate that samples the **lower-left-hand corner** of the texture image, as viewed on a monitor, while the (1, 1) *st* coordinate maps to a (1, 1) *uv* coordinate that samples the **upper-right-hand corner** of the texture image, as viewed on a monitor. See Texture Coordinate Orientation in USD for more details.
- **wrapS - token - useMetadata**
  Wrap mode when reading this texture.
  Possible Values:
  - *black* : Reader returns black outside unit square
  - *clamp* : extend edge values outside unit square
  - *repeat* : repeat texture outside unit square
  - *mirror* : flip and repeat texture outside unit square
  - *useMetadata* : look for **wrapS** and **wrapT** metadata in the texture file itself, that are expected to be string-valued fields whose value is one of *black, clamp, repeat,* or *mirror*. If the texture contains no such metadata, then fall back to ***black*.** If a texture format (such as Pixar *tex* files) already have their own conventions for storing this data, it is the responsibility of the texture loader implementation to translate to the expected values enumerated here.
- **wrapT - token - useMetadata**
  Wrap mode when reading this texture. Same options and caveats as wrapS.
- **fallback - float4 - (0.0, 0.0, 0.0, 1.0)**
  fallback value used when texture can not be read.
- **scale - float4 - (1.0, 1.0, 1.0, 1.0)**
  Scale to be applied to all components of the texture.
  output = textureValue * *scale* + *bias*
- **bias - float4 - (0.0, 0.0, 0.0, 0.0)**
  Bias to be applied to all components of the texture.
  output = textureValue * *scale* + *bias*

Outputs

- **r - float, g - float, b - float, a - float, rgb - float3, rgba - float4**
  Outputs one or more values. If the texture is 8 bit per component [0, 255] values will first be converted to floating point [0, 1] and apply any transformations (bias, scale) indicated. Otherwise it will just apply any transformation (bias, scale). If a single-channel texture is fed into a UsdUVTexture, the **r, g**, and **b** components of the **rgb** and **rgba** outputs will repeat the channel's value, while both the single **a** output and the **a** component of the **rgba** outputs will be set to 1.0. If a two-channel texture is fed into a UsdUVTexture, the **r, g**, and **b** co mponents of the **rgb** and **rgba** outputs will repeat the first channel's value, while both the single **a** output and the **a** component of the **rgba** outputs will be set to the second channel's value.
  The UsdUVTexture node outputs data that is ready to be consumed by the UsdPreviewSurface. Image loaders can do any required pre-multiplication when required by the pipeline.

```
def Shader "UsdUVTexture" (
    doc = """Texture Node Specification represents a node that can be used to
    read UV textures, including tiled textures such as Mari UDIM's.

    Reads from a texture file and outputs one or more values. If the texture has
    8 bits per component, [0, 255] values will first be converted to floating
    point in the range [0, 1] and then any transformations (bias, scale)
    indicated  are applied. Otherwise any indicated transformation (bias,
    scale) is just applied. If a single-channel texture is fed into a
    UsdUVTexture, the r, g, and b  components of the rgb and rgba outputs will
    repeat the channel's value, while both the single 'a' output and the 'a'
    component of the rgba outputs will be set to 1.0.  If a two-channel texture
    is fed into a UsdUVTexture, the r, g, and b components of the rgb and rgba
    outputs will repeat the first channel's value, while both the single "a"
    output and the "a" component of the rgba outputs will be set to the second
    channel's value.
"""
    sdrMetadata = {
        token role = "texture"
    }
)
{
    uniform token info:id = "UsdUVTexture"

    asset inputs:file = @@ (
        connectability = "interfaceOnly"
        doc = """Path to the texture this node uses."""
    )

    float2 inputs:st (
        doc = """This input provides the texture coordinates. It is usually
                connected to a (primvar) node that will provide the texture
                coords."""
    )

    token inputs:wrapS = "useMetadata" (
        connectability = "interfaceOnly"
        doc = """<options> black, clamp, repeat, mirror, useMetadata."""
    )

    token inputs:wrapT = "useMetadata" (
        connectability = "interfaceOnly"
        doc = """<options> black, clamp, repeat, mirror, useMetadata."""
    )

    float4 inputs:fallback = (0.0, 0.0, 0.0, 1.0) (
        doc = """Fallback value to be used when no texture is connected."""
        sdrMetadata = {
            token defaultInput = "1"
        }
    )

    float4 inputs:scale = (1.0, 1.0, 1.0, 1.0) (
        connectability = "interfaceOnly"
        doc = """Scale to be applied to all components of the texture.
```

```
                value * scale + bias"""
    )

    float4 inputs:bias = (0.0, 0.0, 0.0, 0.0) (
        connectability = "interfaceOnly"
        doc = """Bias to be applied to all components of the texture.
                value * scale + bias)"""
    )

    float outputs:r (
        doc = "Outputs the red channel."
    )

    float outputs:g (
        doc = "Outputs the green channel."
    )

    float outputs:b (
        doc = "Outputs the blue channel."
    )

    float outputs:a (
        doc = "Outputs the alpha channel."
    )
    float3 outputs:rgb (
        doc = "Outputs the red, green and blue channels."
    )

    float4 outputs:rgba (
        doc = "Outputs all 4 channels (red, green, blue and alpha)."
    )
}
```

Primvar Reader

The Primvar Reader node provides the ability for shading networks to consume (potentially) surface-varying data defined on geometry (UsdGeom Primvars), including texture coordinates. In contrast to the UsdUVTexture node, which has a fixed set of "common" outputs, more than one of which may be consumed in a shading network, we feel the Primvar reader node is more clearly represented as a "variably typed" node, where its type is determined by the type of the primvar data it consumes from the geometry. By convention, for nodes with variable typed inputs/outputs, we include that information in the "info:id" name to make sure we have a unique identifier for each implementation. We present the **float2** instantiation, with the other allowable instantiations being **float, float3, float4, int, string, normal, point, vector, matrix**. The underlying datatype for **normal**, **point**, and **vector** is *float3*, and the underlying type for **matrix** is *matrix4d*. Note that *color* is not one of the types; we elide it for two reasons:

1. No special processing is required by the node or renderer based on the knowledge of a primvar having the *color* role.
2. Some shading systems and renderers assume that *color* implies *color3f*. We would like to make it as easy as possible to serve and connect 4-channel colors as well as 3-channel. Any primvar whose SdfValueType in USD is *color3f or float3* will successfully bind to a Primvar Reader of the *float3* type, and any primvar whose SdfValueType in USD is *color4f or float4* will successfully bind to a Primvar Reader of the *float4* type.

Templated Definition for UsdPrimvarReader<TYPE>

Node Id

- **UsdPrimvarReader_TYPE**

Inputs

- **varname - token - <EMPTY TOKEN>**
  Name of the primvar to be read from the mesh
- **fallback - TYPE**
  fallback value to be returned if geometry fetch failed.

Outputs

- **result - TYPE**
  Result of the geometry fetch.
  When the UsdPrimvarReader node is used to fetch color data from a mesh, this data is assumed to be ready for consumption by the UsdPreviewSurface node. There is no need to consider pre-multiplication.

Here, for example, is the **float2** variant:

Node Id

- **UsdPrimvarReader_float2**

Inputs (name - type - fallback)

- **varname - token - <EMPTY TOKEN>**
  Name of the primvar to be read from the mesh
- **fallback - float2 - (0.0, 0.0)**
  fallback value to be returned if geometry fetch failed.

Outputs

- **result - float2**
  Result of the geometry fetch

```
class "UsdPrimvarReader" (
    sdrMetadata = {
        token role = "primvar"
    }
)
{
    token inputs:varname  (
        connectability = "interfaceOnly"
        doc = """Name of the primvar to be fetched from the geometry."""
        sdrMetadata = {
            token primvarProperty = "1"
        }
    )
}

def Shader "UsdPrimvarReader_float2" (
    inherits = </UsdPrimvarReader>
)
{
    uniform token info:id = "UsdPrimvarReader_float2"

    float2 inputs:fallback = (0.0, 0.0) (
        doc = """Fallback value to be returned when fetch failed."""
        sdrMetadata = {
            token defaultInput = "1"
        }
    )

    float2 outputs:result
}
```

Transform 2d

Node that takes a 2d input and applies an affine transformation to it. This is especially useful for transforming 2d texture coordinates, which corresponds to:

- the 2D subset of the MDL `rotation_translation_scale` function
- the glTF KHR_texture_transform extension - modulo glTF's use of radians rather than degrees, and the already-noted inverted texture coordinate system in glTF

- A Nodegraph in MaterialX consisting of a chain of **vector2** versions of **multiply, rotate, add** nodes.

The full transformation provided by this node is a standard SRT, i.e.:

```
result = in * scale * rotate * translation
```

> The **UsdTransform2d** node transforms texture coordinates, not the textures themselves, so to get the effect of moving, resizing, or rotating an image being applied to a surface, one must apply the **inverse** transformation of how you expect the image to move.

Node Id

- **UsdTransform2d**

Inputs (name - type - fallback)

- **in - float2 - (0.0, 0.0)**
  Input data to be transformed by this node.
  For instance, you can connect the output of a float2 primvar reader to this input to transform it.
- **rotation - float - (0.0)**
- Counter-clockwise rotation in degrees around the origin.
- **scale - float2 - (1.0, 1.0)**
  Scale around the origin to be applied to all components of the data.
- **translation - float2 - (0.0, 0.0)**
  Translation to be applied to all components of the data.

Outputs

- **result - float2**
  Outputs transformed float2 values.

```
#usda 1.0

def Shader "UsdTransform2d" (
    doc = """Transform 2d represents a node that can be used to
    transform 2d data (for instance, texture coordinates).
    The node applies the following transformation :
    in * scale * rotate + translation"""
    sdrMetadata = {
        token role = "math"
    }
)
{
    uniform token info:id = "UsdTransform2d"

    float2 inputs:in = (0.0, 0.0) (
        doc = """This input provides the data. It is usually
                connected to a UsdPrimvarReader_float2 that
                will provide the data."""
    )
    float inputs:rotation = (0.0) (
        connectability = "interfaceOnly"
        doc = """Counter-clockwise rotation in degrees around the origin to be applied
        to all components of the data."""
    )
    float2 inputs:scale = (1.0, 1.0) (
        connectability = "interfaceOnly"
        doc = """Scale around the origin to be applied to all components of the data."""
    )
    float2 inputs:translation = (0.0, 0.0) (
        connectability = "interfaceOnly"
        doc = """Translation to be applied to all components of the data."""
    )
    float2 outputs:result (
        doc = "Outputs transformed float2 values."
```

```
    )
}
```

USD Sample

Here is an example using the previous nodes.

```
#usda 1.0
(
   upAxis = "Z"
)

def Material "mat"
{
    #
    # Outputs available for the material, they are usually connected to the
    # output of your surface node.
    #
    token outputs:surface.connect      = </mat/pbrMat1.outputs:surface>
    token outputs:displacement.connect = </mat/pbrMat1.outputs:displacement>

    #
    # Public Interface Example: This is how you could define a material public interface
    # This is an easy way for tools to quickly detect tweakable parameters
    # inside the material network
    #
    float inputs:ior = 1.9 # See connection from </mat/pbrMat1.inputs:ior> to this attribute

    #
    # Parameters only useful for tangent space normal mapping
    # Note : Currently, we only support one tangent frame
    #
    # Details : Name of the primvar in your geom to use for the tangents.
    # Default : "tangents"
    token inputs:frame:tangentsPrimvarName = "tangents"

    # Details : Name of the primvar in your geom to use for the binormals
    # Default : "binormals"
    token inputs:frame:binormalsPrimvarName = "binormals"

    # Details : Name of the texture coordinate to be use to calculate
    #           the tangent frame. Mikktspace or similar is recommended for
    #           consistency
    # Default : "st"
    token inputs:frame:stPrimvarName = "st"

    #
    # Preview surface shader.
    #
    def Shader "pbrMat1"
    {
        # Indicate the type of the node.
        uniform token info:id = "UsdPreviewSurface"

        # Outputs available in this shader.
        token outputs:surface
        token outputs:displacement

        # Material Inputs
        int inputs:useSpecularWorkflow       = 0
        color3f  inputs:diffuseColor.connect = </mat/baseColorTex.outputs:rgb>
        color3f  inputs:specularColor        = (0, 0, 0)
        color3f  inputs:emissiveColor        = (0, 0, 0)
        float    inputs:displacement         = 0.0
```

```
    float    inputs:opacity              = 1.0
    float    inputs:opacityThreshold     = 0.0
    float    inputs:roughness            = 0.01
    float    inputs:metallic.connect     = </mat/metallicTex.outputs:r>
    float    inputs:clearcoat.connect    = </mat/clearcoatTex.outputs:r>
    float    inputs:clearcoatRoughness.connect = </mat/clearcoatTex.outputs:g>
    float    inputs:occlusion.connect    = </mat/PrimvarOcclusion.outputs:result>
    normal3f inputs:normal.connect       = </mat/normalTex.outputs:rgb>
    float    inputs:ior.connect          = </mat.inputs:ior>
}


#
# Texture nodes bound to the texture coordinate read by the "Primvar" node
#
def Shader "baseColorTex"
{
    uniform token info:id = "UsdUVTexture"
        float4 inputs:fallback = (0, 1, 0, 1)
    asset inputs:file = @mat_baseColor.png@
    float2 inputs:st.connect = </mat/PrimvarSt1.outputs:result>
    token inputs:wrapS = "black"
    token inputs:wrapT = "clamp"
    float3 outputs:rgb
    float outputs:a
}

def Shader "PrimvarSt1"
{
    uniform token info:id = "UsdPrimvarReader_float2"
    token inputs:varname = "st1"
    float2 outputs:result
}

def Shader "metallicTex"
{
    uniform token info:id = "UsdUVTexture"
        float4 inputs:fallback = (0.3, 0, 0, 1)
    asset inputs:file = @mat_metallic.png@
    float2 inputs:st.connect = </mat/PrimvarSt1.outputs:result>
    float outputs:r
}

def Shader "clearcoatTex"
{
    uniform token info:id = "UsdUVTexture"
        float4 inputs:fallback = (.5, .5, .5, .5)
    asset inputs:file = @mat_clearcoat.png@
    float2 inputs:st.connect = </mat/PrimvarSt1.outputs:result>
    float outputs:r
    float outputs:g
}


#
# Example : Texture using a secondary texture coordinate for UV
#
def Shader "normalTex"
{
    uniform token info:id = "UsdUVTexture"
    float2 inputs:st.connect = </mat/PrimvarSt.outputs:result>
    float4 inputs:scale = (2.0, 2.0, 2.0, 2.0)
    float4 inputs:bias  = (-1.0, -1.0, -1.0, -1.0)
    float3f outputs:rgb
}

def Shader "PrimvarSt"
{
    uniform token info:id = "UsdPrimvarReader_float2"
    token inputs:varname.connect = </mat.inputs:frame:stPrimvarName>
    float2 outputs:result
```

```
    }


    #
    # Example : Primvar data in the mesh being used in the material.
    #
    def Shader"PrimvarOcclusion"
    {
        uniform token info:id = "UsdPrimvarReader_float"
        float inputs:fallback = 1.0
        token inputs:varname = "ao"
        float outputs:result
    }
}

def Mesh "plane1"
{
    float3[] extent = [ (-0.5, -0.1, -0.5), (0.5, 0.1, 0.5)]
    int[] faceVertexCounts = [4, 4]
    int[] faceVertexIndices = [0, 1, 4, 3, 1, 2, 5, 4]
    normal3f[] normals = [(0, 1, 0), (0, 1, 0), (0, 1, 0), (0, 1, 0), (0, 1, 0), (0, 1, 0)]
    point3f[]  points = [(-0.5, 0, 0.5), (0, 0, 0.5), (0.5, 0, 0.5), (-0.5, 0, -0.5), (0, 0, -0.5),
(0.5, 0, -0.5)]
    float[] primvars:ao = [0, 0.5, 1, 1, 0.1, 1] (
        interpolation = "vertex"
    )
    int[] primvars:ao:indices = [0, 1, 4, 3, 2, 5]

    texCoord2f[] primvars:st = [(0, 0), (0.5, 0), (0.5, 1), (0, 1), (1, 0), (1, 1)] (
        interpolation = "vertex"
    )
    int[] primvars:st:indices = [0, 1, 4, 3, 2, 5]
    texCoord2f[] primvars:st1 = [(0, 0), (0.5, 0), (0.5, 1), (0, 1), (1, 0), (1, 1)] (
        interpolation = "vertex"
    )
    int[] primvars:st1:indices = [0, 1, 4, 3, 2, 5]
    uniform token subdivisionScheme = "none"

    rel material:binding = </mat>
}
```

Other Notes

Texture Coordinate Orientation in USD

In pursuit of the goal of making USD a reliable and simple (as possible) standard of interchange, we require that all texture coordinates in USD adhere to the same coordinate system, so that there is never any question of whether texture coordinates need to be "flipped" as they are consumed by our Texture Reader node. The coordinate system we stipulate follows the cartesian coordinate system: if we are viewing, on the same monitor, an axis-aligned quadrilateral and a texture image, the lower left-hand corner of the quadrilateral should be the (0, 0) *st* coordinate, which maps to the lower left-hand corner of the image as the (0, 0) *uv* coordinate. More specifically, for the four vertices of a quadrilateral, in order, the *st* coordinates should be [(0, 0), (1, 0), (1, 1), (0, 1)] .

This mapping is shared by MaterialX and MDL, though unfortunately not glTf, so when converting between glTf and USD, texture coordinates must be "t flipped" (i.e. *tFinal* = 1.0 - *t*) before being consumed by the texture reader.

As an implementation detail, because most image/texture reading packages (including OpenImageIO) consider the *upper-left* corner of an image to be uv (0, 0), the image-reader abstraction in USD that serves as an interface to OIIO and other image readers **flips the layout of the image bottom-to-top** before making it available to shading consumers.

Roughness vs Glossiness

There is no widespread agreement on whether artists should author roughness or glossiness. Rouhness is *usually* used in the metalness workflow, however Unity exposes glossiness (as "Smoothness") in both its metallic and specular workflows, and Substance Painter provides

Metal/Roughness and Specular/Glossiness preview surfaces, while we have chosen to expose roughness in UsdPreviewSurface for both metalness and specular. Therefore, when using UsdPreviewSurface as a transport between systems that may not agree, we must be sensitive of textures and defaults generated for glossiness vs roughness.

Happily, the conversion between the two is easy: **roughness = 1 - glossiness** . Even more happily, the *scale* and *bias* inputs on UsdUVTexture allow us to encode this conversion efficiently, without need for either extra nodes, or modifying textures. If you have a texture that was produced as an input for glossiness, then simply set:

- scale = -1.0
- bias = 1.0

on the UsdUVTexture used to feed the texture to a UsdPreviewSurface's *roughness* input (and/or apply the same inversion to any authored default value).

Earlier Versions

Version 2.0 - Initial Public Specification