TK_MATERIAL_GRAPH

A glTF Extension Proposal/Discussion
May 1, 2019

NOTE: Shared with the Khronos Group glTF working group.

Please feel free to edit this document!

Contributors

- Ben Houston (bhouston@threekit.com)
- Jack Caron (jcaron@threekit.com)
- *Jules Urbach (info@otoy.com)*

Change Log

- 2019-05-16
    - Removed ambientocclusion, viewdirection, screensize per Jan Jordan and Tobias Haeussler.
    - Removed complex types (matrices, color specific types and arrays) and the nodes that would have used them per Jan Jordan.
    - Added example use cases to serve as motivation for features by Ben Houston.

Introduction

Motivation

The motivation for this is that glTF is looking to adopt the Dassault Enterprise PBR model. This model specifies a simple BDFR/EDF model but mapping this to parameters such as textures can be complex and not straight forward. If each input is limited to just a single texture input without additional transforms it will be suboptimal in terms of speed and flexibility - for example a normal map is very large compared to a bump map or a procedural map. The natural solution to this is to use the Dassault Enterprise PBR model with a material node-graph.

Support for Ray Tracing

While gltf might not want to target “raytracers” yet, I think it would be nice to *not* promote techniques can hurt such renderers.

Preliminary Sketch

This is a very preliminary high level sketch of a node-graph definition method for glTF. It is a sketch which if it is in a high level are approved, one can fill in the details.

High Level Design

The design would follow MaterialX relatively closely as well as the aborted OMG idea here: https://github.com/Exocortex/OpenMaterialGraph It is also partially based on the Three.JS node-based graph system.

There would be Materials which contain a root BSDF/EDF node whose parameters would be specified by nodes in a DAG. The nodes expose optional inputs and outputs (except for the root node.) These inputs and outputs are named and typed.

It is important to note that MDL and MaterialX are intended for very advanced workflows. glTF is not yet at that level. It is also an imperative that glTF make it easy for tools to implement the node graph rather than being absolutely complete. A good design of glTF would be as a subset of MaterialX, MDL as well as OTOY’s graph systems.

Types

We could likely use the types from MaterialX. These are the types of the

Primitives

Minimum requirements:

- Integer (Integer2,3,4?)
- Boolean (Boolean2,3,4?)
- Float
- Float2
- Float3
- Float4
- String
- Filename (for specifying texture files, UV channels, geometry parts)

These are optional and for more advanced workflows:

- Geomname?

Nodes

Nodes represent both the operations that occur to create the parameters for the BRDF/EDFs as well as the root BRDF/EDFs themselves.

We should use MaterialX node definitions and their names unless we want to deviate.

Definition Registry

There should be a repository of officially support nodes as JSON. These node JSON definitions can be used for validation. Their existence also reduces the amount of data needed to transfer glTF material graphs as one can infer data from these official specifications.

There should be a base set of node types and then one could have various extensions that specific additional node types. The nodes could be named similarly to existing Khronos standard extensions with vendor prefixes.

- Node
    - Canonical Name: string (maybe could have names similar to Khronos extensions?)
    - Descriptive Name: string
    - Type: string
    - Version: semver
    - Inputs: Map (optional)
        - Name: string
            - Type: string
            - Default: <value> (optional)
    - Outputs: Map (optional)
        - Name

- Type: string

Root

These are the BRDF/EDF node types. In Three.JS this would be Basic, BlinnPhong, Standard and Physical. It could be the Dassault Enterprise PBR model. Etc. These should be standardized of course.

Operators

These are the nodes that manipulate the parameters that are fed into the root nodes. These are copied from MaterialX.

Conditional Nodes

- Compare (ThreeJS: CondNode)
- Switch

Channel Nodes

- Convert
- Swizzle (ThreeJS: SwitchNode)
- Combine (ThreeJS: JoinNode)

Math Operators

Unary

- Negate (ThreeJS: Math1Node)
- Inverse (ThreeJS: Math1Node)
- Normalize (ThreeJS: Math1Node)
- Determinant
- Absolute (ThreeJS: Math1Node)
- Sign (ThreeJS: Math1Node)
- Floor (ThreeJS: Math1Node)
- Ceiling (ThreeJS: Math1Node)
- Cos (ThreeJS: Math1Node)
- Sin (ThreeJS: Math1Node)
- Tan (ThreeJS: Math1Node)
- Acos (ThreeJS: Math1Node)
- Asin (ThreeJS: Math1Node)
- Atan (ThreeJS: Math1Node)
- Exp (ThreeJS: Math1Node)
- Pow2 (ThreeJS: Math2Node !!!)
- Sqrt (ThreeJS: Math1Node)
- Ln (ThreeJS: Math1Node)

- Log2 (ThreeJS: Math1Node)
- Log10
- Magnitude (ThreeJS: Math1Node)
- Norm // same as normalize?
- SmoothStep (ThreeJS: Math3Node)
- SmoothStep2
- Luminance
- RGBtoHSV // ThreeJS: related: ColorSpacNode, ColorAdjustmentNode
- HSVtoRGB

Binary

- Addition (ThreeJS: OperatorNode)
- Subtraction (ThreeJS: OperatorNode)
- Multiply (ThreeJS: OperatorNode)
- Divide (ThreeJS: OperatorNode)
- Power (ThreeJS: Math2Node)
- Log with Base? // ThreeJS: achievable with Math1Node and division from OperatorNode
- Modulo (ThreeJS: Math2Node)
- Atan2
- Min (ThreeJS: Math2Node)
- Max (ThreeJS: Math2Node)
- DotProduct (ThreeJS: Math2Node)
- CrossProduct (ThreeJS: Math2Node)
- Rotate
- Scale
- Sheer
- Translate

Ternary

- Clamp (ThreeJS: Math3Node)
- Mix (ThreeJS: Math3Node)
- Unmix

Other

- Transform
- CurveAdjust
- Remap
- HeightToNormal (ThreeJS: BumpMapNode takes in TextureNode, outputs normal)
- Blur (ThreeJS: BlurNode)
- TrilinearMapping (MaterialX supplemental node.) (Also implemented in one of the demos)
- ProjectVector

Sources

- Texture Node (‘TextureNode’ in ThreeJS) (looks up texel in a bitmap using the provided texel)
- Value Node (useful for constants) (ThreeJS: FloatNode, ColorNode, Vector[2/3/4]Node, etc)

Procedural Nodes

- Constant (useful for centralizing shared values) (ThreeJS: ConstNode)
- Ramp (based on ramplr from MaterialX)
- Split (based on splitlr from MaterialX)
- Noise2D (ThreeJS: NoiseNode)
- Noise3D
- Fractal3D
- CellNoise2D
- CellNoise3D

Geometry Queries

- Position (ThreeJS: PositionNode)
- Normal (ThreeJS: NormalNode)
- Tangent
- Bitangent
- UV Coordinate (ThreeJS: UVNode, but probably supports 2 channels only) (one can specify the map channel)
- Color (vertex color interpolated) (ThreeJS: ColorsNode)
- Depth? (ThreeJS: CameraNode)
- ScreenPosition (relative)? (ThreeJS: ScreenUVNode)
- ScreenVelocity (relative for motion blur? Not part of MaterialX.) (ThreeJS: VelocityNode)
- Cosine between view and normal (before any normal/bump map effect), called facing ratio in V-ray
- Frame time, if something depends on the time (ThreeJS: TimerNode)

Application

- Frame Index
- Time (ThreeJS: TimerNode)

Graphs

A graph is specified by declaring instances of nodes and then connect their inputs with the outputs of other nodes or specifying their input nodes explicitly. The graph should always contain a root BRDF/EDF node. It should create a DAG, no cycles are allowed of any type.

The graph is specified in JSON.

Specification of Graphs

Something like this:

- Nodes: map // The map of nodes is not ordered
    - NodeName: string (unique name in the graph, used for making connections)
        - NodeType: string (references the official node type definitions.)
        - Inputs: map (optional)
            - InputName: (either value or connection)
                - Value: valuetype
                - Connection: string // nodename/inputname

Tooling

A number of tools should be created to support the material graphs.

It may be best to restrict the scope of tooling to be as minimal as possible and rely on translation to other graph-based formats as much as possible. This way we can reuse the OTOY, MDL and MaterialX tooling as possible.

Validator

The existing glTF validator should be extended to support the validation of material graphs. This would work in concert with the node definitions and ensure that types match between the node connections, the connections form a correct DAG, and that the node definitions are used properly.

Shading Language Compilers

One could create a compiler for the material node graph to straight a shader implementation language. The result would be glsl.

Although for the Three.JS case, it is likely that we just utilize the existing Three.JS Node system. Thus the shader compiler would be just a converter from glTF material node graph to the Three.JS Node system graph.

While one could create an glTF node-material graph to OSL compiler, it may be best to utilize the existing MDL, MaterialX, or OTOY to OSL compilers as these exist and are robust.

Format Conversion

Converting between high level material formats should be relatively easy of those other formats are graph-based.

The main challenge will arise with regard to the root BRDF/EDF node compatibility - often these are hard coded and hard to modify.

Also converting from higher end material definitions to glTF will likely be lossy to a degree if advanced features are used.

MaterialX

It should be possible to write a converter from the glTF node material format to MaterialX format with little trouble.

A subset of MaterialX can likely be converted to the glTF node material format.

MDL

It should be possible to write a converter from the glTF node material format to the MDL format with little trouble.

A subset of MDL can likely be converted to the glTF node material format.

Renderers

There is a large set of renderers available in the wild, UE4, Unity, Apple ModelIO, Filament, RenderMan, Arnold, V-Ray, Octane, Redshift to name just a few. It would be best if these renderers could adopt MaterialX or MDL and then we use the standard-based conversion workflow to integrate with these renderers rather than building a large set of renderer specific conversion tools.

Baking

It should be possible to write a tool that given a specific geometry and a node-material graph definition could bake the resulting unwrapped maps.

This would be similar to the NVIDIA MDL Distiller which bakes MDL materials to standard textures that can be put into a simple UE4-like Physical material.

Challenge: Unwrapped UVs Required

The main challenge is that an unwrapped UV set will be required, which the tool could create, or it could be a requirement that the model has an already unwrapped UV set available.

Questions

- Where do color spaces come into things? Are color spaces specified in textures already? Specifically sRGB, RGB, LogUV, etc. Is it necessary to provide color space conversion routines in the standard node library?

- Texture encoding/decoding like RGBE, RGBM, RGBD, etc. Should these be added to the standard node library?]

- Why not embed MDL into a glTF?
    - MDL is not web-native JSON, it is a non-standard format that requires a custom parser to handle.
    - MDL has a large feature set of which we would only support a subset of.
    - **Thus our proposed solution should be basically equivalent to embedding MDL except that it uses JSON as the format and it only supports a subset of MDL features but is otherwise compatible with it.**

- Why not embed MaterialX into a glTF?
    - MaterialX is XML, this requires an XML parsing library in the browser where as JSON is JavaScript native.
    - MaterialX has a large feature set (included embedded OSL for custom nodes) of which we would only support a subset of.
    - **Thus our proposed solution should be basically equivalent to embedding MaterialX except that it uses JSON as the format and it only supports a subset of MaterialX features but is otherwise compatible with it.**

- Why not embed OTOY Graph Format into a glTF?
    - OTOY Graph Format is XML and this would require an XML parsing library in the browser were as JSON is JavaScript native.
    - OTOY has a lot of advanced features (included embedded OSL for custom nodes, and a large set of BRDF/EDFs) of which we would only support a subset of.
    - The node subset we have picked should be a proper subset of OTOY's feature set. **Thus our proposed solution should be basically equivalent to embedding OTOY except that it uses JSON as the format and it only supports a subset of OTOY features but is otherwise compatible with it.**

Appendix

Example Use Cases

This section outlines a few simple use cases which the material graph proposal should support.

Basic Bump Map

A bump map is a lot more efficient to transmit than a normal map and less susceptible to compression artifacts.

Requirements:

- Access bump map using a UV channel
- Convert the height map into a normal
- Connect the result into the normal input of the root BRDF

Arbitrary UV Map per Texture

Right now glTF specifies which uv maps should be used for which texture inputs. This is highly restrictive and often not a good choice. It is best to allow for one to choose which UV map should be used for each texture read.

Requirements:

- Access a map using a specified UV map

Combining Different Textures Read Using Different UVs

This scenario is generally done for skin on a human face. There is a normal map on the unwrapped UVs that does the major face creases while there is a secondary bump map that may use a repeating UV tiling that does the microstructure pore details. The combination of these two would result in the final normal input into the resulting BRDF.

https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/HowTo/Human_Skin
https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/HowTo/DetailTexturing

Requirements:

- Access normal map using one UV map
- Access the height map using a secondary UV map.
- Convert the height map to a normal
- Combine the normal map and the height map normal
- Connect the result into the normal input of the root BRDF

Triplanar Mapping

In many situation it is not possible to easily unwrap UVs on an object. Instead artists will employ a triplanar mapping that uses three orthogonal planes in either the local or world coordinates to project UVs onto the object and blend between the resulting texture values based on where the surface normal is pointing too.

This is a specific case of a real-time UV projection combined with interpolation between those specific texture reads.

http://www.martinpalko.com/triplanar-mapping/

Requirements:

- Specify coordinate space + scale factor + uv map + texture and get a texture result.
- Pass the texture result into the albedo input of the root BRDF.

Unpack Single Bitmap to Multiple Channels

It is common in UE4 and other tools to combine multiple channels (roughness, metalness, ambient occlusion) into a single bitmap.

Requirements:

- Read a texture map using a specific UV map
- Extract the different packed data in the texture map
- Pass different components into separate inputs into the root BRDF

Modulate Maps

Many complex materials, such as cloth, can be simplified to just a simple repeating thread pattern that is combined with some noise and then a slightly varying color pattern. This can be represented primarily by a single small scale bump map, a singular color and a noise texture. This is an efficient and compact representation, the alternative would be large baked maps with lots of detail that would be more than 10x larger.

Requirements:

- Access the bump map using a specific UV map.
- Access the noise texture using a specific UV map.
- Get the specified color.
- Add randomness to bump map via addition of noise map.
- Convert bump map to normal map.
- Create variations in color by modulating color slightly by noise map.
- Set color to albedo input of root BRDF.
- Set normal to normal input of root BRDF.

Masking Albedo

A very common feature that artists use is to use one bitmap as a mask of others. This allows for very complex transition effects between two different materials -- the transition complexity is limited by the resolution of the mask texture, rather than geometry. It also allows for varying transitions if one uses semi-transparency on the mask.

The masked textures are often repeating and to bake them with the mask into single textures would result in very large files.

https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/HowTo/Masking

Requirements:

- Read a mask texture map using a specific UV Map
- Use the mask data to interpolate between two different inputs.
- Pass the result to an input on the root BRDF.

Anisotropic Representation Conversion?

The historic representation of anisotropy was as a strength and a rotation usually read from two separate bitmaps. The more modern representation is as a 2-component vector whose magnitude represents strength and whose direction is the rotation.

Requirements:

- Read anisotropic vector texture using a specific UV Map
- Get magnitude of vector.
- Get angle of vector direction.
- Set anisotropic magnitude and angle to inputs on root BRDF.

Decode Bitmap?

Often texture data is packed into RGBE, RGBM16 and other similar representations. It would be nice to be able to decode texture data to an HDR format or transform its color space. Should this be a texture property or should it be a feature of the node graph?

Requirement:

- Read texture map using a specific UV Map
- Apply a decode function to the color data to transform it
- Pass the resulting data into an input on the root BRDF.

References

1. MaterialX, https://www.materialx.org/assets/MaterialX.v1.36.Spec.pdf
2. Open Material Graph (2015) https://github.com/Exocortex/OpenMaterialGraph
3. MDL from NVIDIA https://developer.download.nvidia.com/designworks/mdl-sdk/secure/MDL_spec_1.3.2_16Sep2016.pdf
4. ThreeJS Node Graph https://github.com/mrdoob/three.js/tree/dev/examples/js/nodes