glTF - what the ?

For glTF 2.0!

An overview of the basics of the GL Transmission Format

glTF was designed and specified by the Khronos Group, for the efficient transfer of 3D content over networks.

The core of glTF is a **JSON** file that describes the structure and composition of a scene containing 3D models. The top-level elements of this file are:

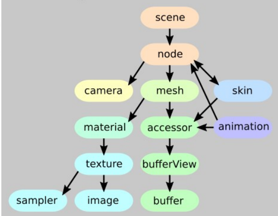
- **scenes, nodes** — Basic structure of the scene
- **cameras** — View configurations for the scene
- **meshes** — Geometry of 3D objects
- **buffers, bufferViews, accessors** — Data references and data layout descriptions
- **materials** — Definitions of how objects should be rendered
- **textures, images, samplers** — Surface appearance of objects
- **skins** — Information for vertex skinning
- **animations** — Changes of properties over time

These elements are contained in arrays. References between the objects are established by using their indices to look up the objects in the arrays.

It is also possible to store the whole asset in a single binary glTF file. In this case, the JSON data is stored as a string, followed by the binary data of buffers or images.

Concepts

The conceptual relationships between the top-level elements of a glTF asset are shown here:



Binary data references

The images and buffers of a glTF asset may refer to external files that contain the data that are required for rendering the 3D content:


```
"buffers": [
  {
    "uri": "buffer01.bin"
    "byteLength": 102040,
  }
],
"images": [
  {
    "uri": "image01.png"
  }
],
```


The **buffers** refer to binary files (.BIN) that contain geometry- or animation data.

The **images** refer to image files (PNG, JPG...) that contain texture data for the models.

The data is referred to via URIs, but can also be included directly in the JSON using data URIs. The data URI defines the MIME type, and contains the data as a base64 encoded string:

Buffer data:

"data:application/gltf-buffer;base64,AAABAAIAAgA..."

Image data (PNG):

"data:image/png;base64,iVBORw0K..."

Further resources

The Khronos glTF landing page: https://www.khronos.org/gltf

The Khronos glTF GitHub repository: https://github.com/KhronosGroup/glTF

Version 2.0b
glTF version 2.0

This overview is non-normative!

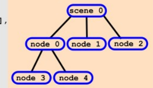
Feedback: gltf@marco-hutter.de

scenes, nodes

The glTF JSON may contain **scenes** (with an optional default **scene**). Each scene can contain an array of indices of nodes.


```
"scene": 0,
"scenes": [
  {
    "nodes": [ 0, 1, 2 ]
  }
],
"nodes": [
  {
    "children": [ 3, 4 ],
    ...
  },
  { ... },
  { ... },
  { ... },
  { ... },
  ...
],
```


Each of the **nodes** can contain an array of indices of its **children**. This allows modeling a simple scene hierarchy:




```
"nodes": [
  {
    "matrix": [
      1,0,0,0,
      0,1,0,0,
      0,0,1,0,
      5,6,7,1
    ],
    ...
  },
  {
    "translation":
      [ 0,0,0 ],
    "rotation":
      [ 0,0,0,1 ],
    "scale":
      [ 1,1,1 ]
    ...
  },
]
```


A node may contain a local transform. This can be given as a column-major **matrix** array, or with separate **translation**, **rotation** and **scale** properties, where the rotation is given as a quaternion. The local transform matrix is then computed as

M = T * R * S

where **T**, **R** and **S** are the matrices that are created from the translation, rotation and scale. The global transform of a node is given by the product of all local transforms on the path from the root to the respective node.

Each node may refer to a **mesh** or a **camera**, using indices that point into the meshes and cameras arrays. These elements are then attached to these nodes. During rendering, instances of these elements are created and transformed with the global transform of the node.


```
"nodes": [
  {
    "mesh": 4,
    ...
  },
  {
    "camera": 2,
    ...
  },
]
```


The translation, rotation and scale properties of a node may also be the target of an animation: The animation then describes how one property changes over time. The attached objects will move accordingly, allowing to model moving objects or camera flights.

Nodes are also used in vertex skinning: A node hierarchy can define the skeleton of an animated character. The node then refers to a mesh and to a skin. The skin contains further information about how the mesh is deformed based on the current skeleton pose.

meshes

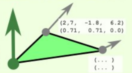
The **meshes** may contain multiple mesh **primitives**. These refer to the geometry data that is required for rendering the mesh.


```
"meshes": [
  {
    "primitives": [
      {
        "mode": 4,
        "indices": 0,
        "attributes": {
          "POSITION": 1,
          "NORMAL": 2
        },
        "material": 2
      }
    ]
  }
],
```


Each mesh primitive has a rendering **mode**, which is a constant indicating whether it should be rendered as **POINTS**, **LINES**, or **TRIANGLES**. The primitive also refers to **indices** and the **attributes** of the vertices, using the indices of the accessors for this data. The **material** that should be used for rendering is also given, by the index of the material.

Each attribute is defined by mapping the attribute name to the index of the accessor that contains the attribute data. This data will be used as the vertex attributes when rendering the mesh. The attributes may, for example, define the **POSITION** and the **NORMAL** of the vertices:

POSITION	1.2	-2.6	4.3	2.7	-1.8	6.2	...
NORMAL	0.0	1.0	0.0	0.71	0.71	0.0	...



A mesh may define multiple morph targets. Such a morph target describes a deformation of the original mesh.


```
{
  "primitives": [
    {
      ...
      "targets": [
        {
          "POSITION": 11,
          "NORMAL": 13
        },
        {
          "POSITION": 21,
          "NORMAL": 23
        }
      ]
    }
  ],
  "weights": [0, 0.5]
}
```


To define a mesh with morph targets, each mesh primitive can contain an array of **targets**. These are dictionaries that map names of attributes to the indices of accessors that contain the displacements of the geometry for the target.

The mesh may also contain an array of **weights** that define the contribution of each morph target to the final, rendered state of the mesh.

Combining multiple morph targets with different weights allows, for example, modeling different facial expressions of a character: The weights can be modified with an animation, to interpolate between different states of the geometry.

buffers, bufferViews, accessors

The **buffers** contain the data that is used for the geometry of 3D models, animations, and skinning. The **bufferViews** add structural information to this data. The **accessors** define the exact type and layout of the data.


```
"buffers": [
  {
    "byteLength": 35,
    "uri": "buffer01.bin"
  }
],
```


Each of the **buffers** refers to a binary data file, using a **URI**. It is the source of one block of raw data with the given **byteLength**.


```
"bufferViews": [
  {
    "buffer": 0,
    "byteOffset": 4,
    "byteLength": 28,
    "byteStride": 12,
    "target": 34963
  }
],
```


Each of the **bufferViews** refers to one buffer. It has a **byteOffset** and a **byteLength**, defining the part of the buffer that belongs to the bufferView, and an optional OpenGL buffer **target**.

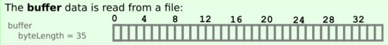

```
"accessors": [
  {
    "bufferView": 0,
    "byteOffset": 4,
    "type": "VEC2",
    "componentType": 5126,
    "count": 2,
    "min": [0.1, 0.2]
    "max": [0.9, 0.8]
  }
]
```


The **accessors** define how the data of a bufferView is interpreted. They may define an additional **byteOffset** referring to the start of the bufferView, and contain information about the type and layout of the bufferView data:

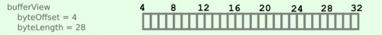
The data may, for example, be defined as 2D vectors of floating point values when the **type** is **"VEC2"** and the **componentType** is **GL_FLOAT** (5126). The range of all values is stored in the **min** and **max** property.

The data of multiple accessors may be interleaved inside a bufferView. In this case, the bufferView will have a **byteStride** property that says how many bytes are between the start of one element of an accessor, and the start of the next.

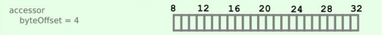
The **buffer** data is read from a file:



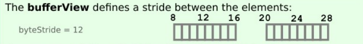
The **bufferView** defines a segment of the buffer data:



The **accessor** defines an additional offset:



The **bufferView** defines a stride between the elements:



The **accessor** defines that the elements are 2D float vectors:

This data may, for example, be used by a mesh primitive, to access 2D texture coordinates: The data of the **bufferView** may be bound as an OpenGL buffer, using **glBindBuffer**. Then, the properties of the **accessor** may be used to define this buffer as vertex attribute data, by passing them to **glVertexAttribPointer** when the bufferView buffer is bound.

Sparse accessors

When only few elements of an **accessor** differ from a default value (which is often the case for morph targets), then the data can be given in a very compact form using a **sparse** data description:


```
"accessors": [
  {
    "type": "SCALAR",
    "componentType": 5126,
    "count": 10,

    "sparse": {
      "count": 4,

      "values": {
        "bufferView": 2,
      },

      "indices": {
        "bufferView": 1,
        "componentType": 5123
      }
    }
  }
]
```

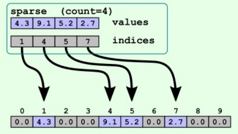

The accessor defines the type of the data (here, scalar float values), and the total element **count**.

The **sparse** data block contains the **count** of sparse data elements.

The **values** refer to the bufferView that contains the sparse data values.

The target **indices** for the sparse data values are defined with a reference to a bufferView and the **componentType**.

The **values** are written into the final accessor data, at the positions that are given by the **indices**:

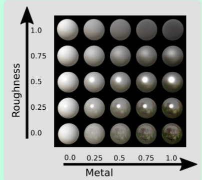


Final **accessor** data with 10 float values

materials

Each mesh primitive may refer to one of the **materials** that are contained in a glTF asset. The materials describe how an object should be rendered, based on physical material properties. This allows to apply **Physically Based Rendering (PBR)** techniques, to make sure that the appearance of the rendered object is consistent among all renderers.

The default material model is the **Metallic-Roughness-Model**. Values between 0.0 and 1.0 are used to describe how much the material characteristics resemble that of a metal, and how rough the surface of the object is. These properties may either be given as individual values that apply to the whole object, or be read from textures.




```
"materials": [
  {
    "pbrMetallicRoughness": {
      "baseColorTexture": {
        "index": 1,
        "texCoord": 1
      },
      "baseColorFactor":
        [ 1.0, 0.75, 0.35, 1.0 ],
      "metallicRoughnessTexture": {
        "index": 5,
        "texCoord": 1
      },
      "metallicFactor": 1.0,
      "roughnessFactor": 0.0,
    }
    "normalTexture": {
      "scale": 0.8,
      "index": 2,
      "texCoord": 1
    },
    "occlusionTexture": {
      "strength": 0.9,
      "index": 4,
      "texCoord": 1
    },
    "emissiveTexture": {
      "index": 3,
      "texCoord": 1
    },
    "emissiveFactor":
      [0.4, 0.8, 0.6]
  }
],
```


The properties that define a material in the Metallic-Roughness-Model are summarized in the **pbrMetallicRoughness** object:

The **baseColorTexture** is the main texture that will be applied to the object. The **baseColorFactor** contains scaling factors for the red, green, blue and alpha component of the color. If no texture is used, these values will define the color of the whole object.

The **metallicRoughnessTexture** contains the metalness value in the "blue" color channel, and the roughness value in the "green" color channel. The **metallicFactor** and **roughnessFactor** are multiplied with these values. If no texture is given, then these factors define the reflection characteristics of the whole object.

In addition to the properties that are defined via the Metallic-Roughness-Model, the material may contain other properties that affect the object appearance:

- The **normalTexture** refers to a texture that contains tangent-space normal information, and a **scale** factor that will be applied to these normals.
- The **occlusionTexture** refers to a texture that defines areas of the surface that are occluded from light, and thus rendered darker. This information is contained in the "red" channel of the texture. The occlusion **strength** is a scaling factor to be applied to these values.
- The **emissiveTexture** refers to a texture that may be used to illuminate parts of the object surface: It defines the color of the light that is emitted from the surface. The **emissiveFactor** contains scaling factors for the red, green and blue components of this texture.

Material properties in textures


```
"meshes": [
  {
    "primitives": [
      {
        "material": 2,
        "attributes": {
          "NORMAL": 3,
          "POSITION": 1,
          "TEXCOORD_0": 2,
          "TEXCOORD_1": 5
        },
      }
    ]
  }
],
```



```
"materials": [
  ...
  {
    "name": "brushed gold",
    "pbrMetallicRoughness": {
      "baseColorFactor":
        [1,1,1,1],
      "baseColorTexture": {
        "index" : 1,
        "texCoord" : 1
      },
      "metallicFactor": 1.0,
      "roughnessFactor": 1.0
    }
  }
],
```



```
"textures": [
  ...
  {
    "source": 4,
    "sampler": 2
  }
  ...
],
```


The texture references in a material always contain the **index** of the texture. They may also contain the **texCoord** set index. This is the number that determines the **TEXCOORD_<n>** attribute of the rendered mesh primitive that contains the texture coordinates for this texture, with 0 being the default.

cameras

Each of the nodes may refer to one of the **cameras** that are defined in the glTF asset.


```
"cameras": [
  {
    "type": "perspective",
    "perspective": {
      "aspectRatio": 1.5,
      "yfov": 0.65,
      "zfar": 100,
      "znear": 0.01
    }
  },
  {
    "type": "orthographic",
    "orthographic": {
      "xmag": 1.0,
      "ymag": 1.0,
      "zfar": 100,
      "znear": 0.01
    }
  }
]
```


There are two types of cameras: **perspective** and **orthographic** ones, and they define the projection matrix.

The value for the far clipping plane distance of a perspective camera, **zfar**, is optional. When it is omitted, the camera uses a special projection matrix for infinite projections.

When one of the nodes refers to a camera, then an instance of this camera is created. The camera matrix of this instance is given by the global transform matrix of the node.

textures, images, samplers

The **textures** contain information about textures that may be applied to rendered objects: Textures are referred to by materials to define the basic color of the objects, as well as physical properties that affect the object appearance.


```
"textures": [
  {
    "source": 4,
    "sampler": 2
  }
  ...
],

"images": [
  ...
  {
    "uri": "file01.png"
  }
  {
    "bufferView": 3,
    "mimeType" :
      "image/jpeg"
  }
],

"samplers": [
  ...
  {
    "magFilter": 9729,
    "minFilter": 9987,
    "wrapS": 10497,
    "wrapT": 10497
  }
],
```


The texture consists of a reference to the **source** of the texture, which is one of the **images** of the asset, and a reference to a **sampler**.

The **images** define the image data used for the texture. This data can be given via a **URI** that is the location of an image file, or by a reference to a **bufferView** and a **MIME type** that defines the type of the image data that is stored in the buffer view.

The **samplers** describe the wrapping and scaling of textures. (The constant values correspond to OpenGL constants that can directly be passed to **glTexParameter**).

skins

A glTF asset may contain the information that is necessary to perform vertex skinning. With vertex skinning, it is possible to let the vertices of a mesh be influenced by the bones of a skeleton, based on its current pose.

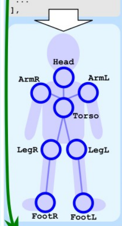

```
"nodes": [
  {
    "name" :
      "Skinned mesh node",
    "mesh": 0
    "skin": 0,
  },
  ...
  {
    "name": "Torso",
    "children":
      [ 2, 3, 4, 5, ...],
    "rotation": [...],
    "scale": [ ...],
    "translation": [...]
  },
  ...
  {
    "name": "LegL",
    "children": [ 7 ],
    ...
  },
  ...
  {
    "name": "FootL",
    ...
  },
  ...
],
```


A node that refers to a **mesh** may also refer to a **skin**.


```
"skins": [
  {
    "inverseBindMatrices": 12,
    "joints": [ 1, 2, 3 ... ]
  }
],
```


The **skins** contain an array of **joints**, which are the indices of nodes that define the skeleton hierarchy, and the **inverseBindMatrices**, which is a reference to an accessor that contains one matrix for each joint.

The skeleton hierarchy is modeled with nodes, just like the scene structure: Each joint node may have a local transform and an array of children, and the "bones" of the skeleton are given implicitly, as the connections between the joints.




```
"meshes": [
  {
    "primitives": [
      {
        "attributes": {
          "POSITION": 0,
          "JOINTS_0": 1,
          "WEIGHTS_0": 2
          ...
        },
      }
    ]
  }
],
```


The mesh primitives of a skinned mesh contain the **POSITION** attribute that refers to the accessor for the vertex positions, and two special attributes that are required for skinning: A **JOINTS_0** and a **WEIGHTS_0** attribute, each referring to an accessor.

The **JOINTS_0** attribute data contains the indices of the joints that should affect the vertex.

The **WEIGHTS_0** attribute data defines the weights indicating how strongly the joint should influence the vertex.

From this information, the **skinning matrix** can be computed.

This is explained in detail in **"Computing the skinning matrix"**.

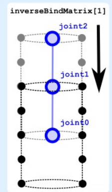
Computing the skinning matrix

The skinning matrix describes how the vertices of a mesh are transformed based on the current pose of a skeleton. The skinning matrix is a weighted combination of joint matrices.

Computing the joint matrices

The skin refers to the **inverseBindMatrices**. This is an accessor which contains one inverse bind matrix for each joint. Each of these matrices transforms the mesh into the local space of the joint.

For each node whose index appears in the **joints** of the skin, a global transform matrix can be computed. It transforms the mesh from the local space of the joint, based on the current global transform of the joint, and is called **globalJointTransform**.

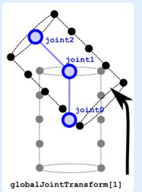


From these matrices, a **jointMatrix** may be computed for each joint:

```
jointMatrix[j] =
    inverse(globalTransform) *
    globalJointTransform[j] *
    inverseBindMatrix[j];
```

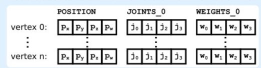
Any global transform of the node that contains the mesh and the skin is cancelled out by pre-multiplying the joint matrix with the inverse of this transform.

For implementations based on OpenGL or WebGL, the **jointMatrix** array will be passed to the vertex shader as a uniform.



Combining the joint matrices to create the skinning matrix

The primitives of a skinned mesh contain the **POSITION**, **JOINT** and **WEIGHT** attributes, referring to accessors. These accessors contain one element for each vertex:



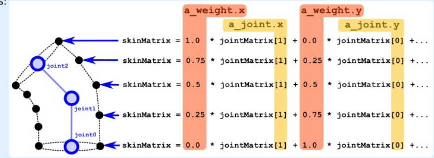
The data of these accessors is passed as attributes to the vertex shader, together with the **jointMatrix** array.

In the vertex shader, the **skinMatrix** is computed. It is a linear combination of the joint matrices whose indices are contained in the **JOINTS_0** attribute, weighted with the **WEIGHTS_0** values:

Vertex Shader

```
uniform mat4 u_jointMatrix[12];
attribute vec4 a_position;
attribute vec4 a_joint;
attribute vec4 a_weight;
...
void main(void) {
    ...
    mat4 skinMatrix =
        a_weight.x * u_jointMatrix[int(a_joint.x)] +
        a_weight.y * u_jointMatrix[int(a_joint.y)] +
        a_weight.z * u_jointMatrix[int(a_joint.z)] +
        a_weight.w * u_jointMatrix[int(a_joint.w)];
    gl_Position =
        modelViewProjection * skinMatrix * position;
}
```

The **skinMatrix** transforms the vertices based on the skeleton pose, before they are transformed with the model-view-perspective matrix.



Wiki page about skinning in COLLADA: https://www.khronos.org/collada/wiki/Skinning
Section 4-7 in the COLLADA specification: https://www.khronos.org/files/collada_spec_1_5.pdf
(The vertex skinning in COLLADA is similar to that in glTF)

animations

A glTF asset can contain **animations**. An animation can be applied to the properties of a node that define the local transform of the node, or to the weights for the morph targets.


```
"animations": [
  {
    "channels": [
      {
        "target": {
          "node": 1,
          "path": "translation"
        },
        "sampler": 0
      }
    ],
    "samplers": [
      {
        "input": 4,
        "interpolation": "LINEAR",
        "output": 5
      }
    ]
  }
]
```

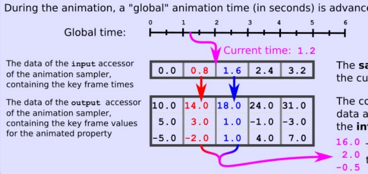

Each animation consists of two elements: An array of **channels** and an array of **samplers**.

Each channel defines the **target** of the animation. This target usually refers to a **node**, using the index of this node, and to a **path**, which is the name of the animated property. The path may be **"translation"**, **"rotation"** or **"scale"**, affecting the local transform of the node, or **"weights"**, in order to animate the weights of the morph targets of the meshes that are referred to by the node. The channel also refers to a **sampler**, which summarizes the actual animation data.

A sampler refers to the **input** and **output** data, using the indices of accessors that provide the data. The input refers to an accessor with scalar floating-point values, which are the times of the key frames of the animation. The output refers to an accessor that contains the values for the animated property at the respective key frames. The sampler also defines an **interpolation** mode for the animation, which may be **"LINEAR"**, **"STEP"**, or **"CUBICSPLINE"**.

Animation samplers

During the animation, a "global" animation time (in seconds) is advanced.



The **sampler** looks up the key frames for the current time, in the **input** data.

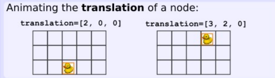
The corresponding values of the **output** data are read, and interpolated based on the **interpolation** mode of the sampler.

The interpolated value is forwarded to the animation channel target.

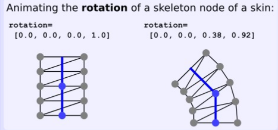
Animation channel targets

The interpolated value that is provided by an animation sampler may be applied to different animation channel targets.

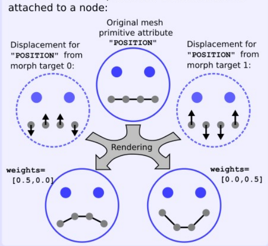
Animating the **translation** of a node:



Animating the **rotation** of a skeleton node of a skin:



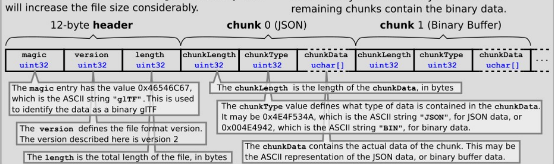
Animating the **weights** for the morph targets that are defined for the primitives of a mesh that is attached to a node:



Binary glTF files

In the standard glTF format, there are two options for including external binary resources like buffer data and textures: They may be referenced via URIs, or embedded in the JSON part of the glTF using data URIs. When they are referenced via URIs, then each external resource implies a new download request. When they are embedded as data URIs, the base 64 encoding of the binary data will increase the file size considerably.

To overcome these drawbacks, there is the option to combine the glTF JSON and the binary data into a single **binary glTF** file. This is a little-endian file, with the extension **".glb"**. It contains a **header**, which gives basic information about the version and structure of the data, and one or more **chunks** that contain the actual data. The first chunk always contains the JSON data. The remaining chunks contain the binary data.



Extensions

The glTF format allows extensions to add new functionality, or to simplify the definition of commonly used properties.

When an extension is used in a glTF asset, it has to be listed in the top-level **extensionsUsed** property. The **extensionsRequired** property lists the extensions that are strictly required to properly load the asset.


```
"extensionsUsed" : [
  "KHR_lights_common",
  "CUSTOM_EXTENSION"
]

"extensionsRequired" : [
  "KHR_lights_common"
]
```



```
"textures" : [
  {
    ...
    "extensions" : {
      "KHR_lights_common" : {
        "lightSource" : true,
      },
      "CUSTOM_EXTENSION" : {
        "customProperty" :
          "customValue"
      }
    }
  }
]
```


Extensions allow adding arbitrary objects in the **extensions** property of other objects.

The name of such an object is the same as the name of the extension, and it may contain further, extension-specific properties.

Existing extensions

The following extensions are developed and maintained on the Khronos GitHub repository:

- **Specular-Glossiness Materials**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/2.0/Khronos/KHR_materials_pbrSpecularGlossiness
  This extension is an alternative to the default Metallic-Roughness material model: It allows to define the material properties based on specular and glossiness values.
- **Unlit Materials**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/2.0/Khronos/KHR_materials_unlit
  This extension allows the definition of materials for which no physically based lighting computations should be performed.
- **Punctual Lights**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/2.0/Khronos/KHR_lights_punctual
  This extension allows adding different types of lights to the scene hierarchy. This refers to point lights, spot lights and directional lights. The lights can be attached to the nodes of the scene hierarchy.
- **WebGL Rendering Techniques**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/2.0/Khronos/KHR_techniques_webgl
  With this extension, it is possible to define GLSL shaders that should be used for rendering the glTF asset in OpenGL or WebGL
- **Texture transforms**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/2.0/Khronos/KHR_texture_transform
  This extension allows defining offset, rotation, and scaling for textures, so that multiple textures can be combined in order to create a texture atlas