glTF - 是什么  ？

一份关于glTF格式(GL传输格式)基础的概述

glTF 由Khronos Group 设计和规范, 旨在提高3D内容在网络中的传输效率.

glTF的核心是一个 **JSON** 文件， 该文件描述了包含3D模型的场景(scene)的结构和组成. 文件顶层元素有:

scenes, nodes
场景(scene)的基本结构

cameras
场景(scene)的视角配置

meshes
3D物体的几何体

buffers, bufferViews, accessors
数据引用和数据布局的描述

materials
定义物体应该怎样被渲染

textures, images, samplers
物体的外观

skins
顶点蒙皮的信息

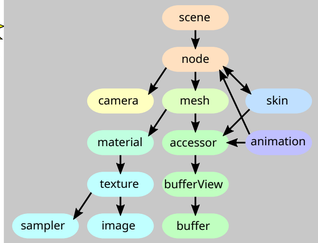
animations
属性随时间的改变

这些元素包含在数组中， 使用索引可以查询的对象，并以此建立起对象间的引用关系。 → 数组中

也可以将整个资源文件存储在单一的二进制 glTF 文件中，此时json数据以字符串的形式存储，随后是缓冲(buffers)或图片(images)的二进制数据.

概念

glTF资产顶层元素之间的概念关系如下图所示:



二进制数据引用

glTF资产的图片(images)跟缓冲(buffers)可以引用外部文件，这些外部文件包含渲染3D内容的数据:

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

buffers 引用了包含几何或动画数据的二进制文件(.BIN)

images 引用了包含模型纹理数据的图片文件(PNG, JPG...)

数据虽然是通过URI引用，但JSON也可以通过使用数据URI直接包含数据.数据URI定义了MIME类型，以base64编码字符串的形式包含数据:

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

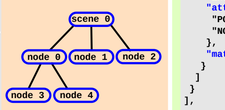
Image data (PNG):
`"..."`

scenes, nodes

glTF JSON 文件中可能包含 **scenes** (带有一个默认可选的 **scene**). 每个scene包含一个由nodes索引组成的数组.

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

每个 **nodes** 可以包含一个数组, 数组则是由它的**children**的索引组成.这便可以搭建起一个简单的场景体系：




```
"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 ]
  ...
},
...
]
```

node可包含一个局部变换，这个变换可以用一个列主序**矩阵**表示, 或者用平移，旋转和缩放属性分别表示，其中旋转使用四元数形式.局部变换矩阵可通过以下计算：

M = T * R * S

其中 T, R 和 S 分别为平移，旋转和缩放矩阵.从根node到相对应node路径上的所有局部变换相乘，便得到这个节点的全局变换

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

node可以使用指向meshes和cameras数组的索引来引用一个mesh或camera.引用后则这些元素会附加到这些节点上。在渲染过程中，这些元素的实例会被创建出来并使用节点的全局变换进行转换。

node的平移，旋转和缩放特性也可以用于动画，因为动画本就是描述某个特性如何随时间变化的。这样附属的对象也会相应地移动，从而允许对移动的物体或移动的摄像机进行建模。

node也可用于顶点蒙皮: node的层次结构可以定义动画角色的骨架，node引用mesh和skin，skin中包含了mesh如何基于当前的骨架姿态进行变形的进一步的信息。

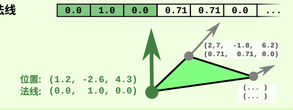
meshes

meshes中可以包含多个 mesh **图元**(primitives).它们引用的几何图形数据正是渲染mesh所需要的.

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

每个mesh 图元都有一个常数表示的渲染模式**mode**，这个模式用来指明应该按照点，线，还是三角形来渲染.使用这个数据访问器的索引，图元也可以引用**indices**和顶点属性**attributes**.通过材质的索引,也可以给出渲染需要的材质**material**。

每个属性通过将属性名称映射到包含该属性数据的访问器（accessor）的索引来定义，这些数据将在渲染网格时用作顶点属性.例如，这些属性可能会定义顶点的位置（POSITION）和法线（NORMAL）



一个网格（mesh）可以定义多个形变目标（morph targets）.形变目标描述的是原始网格的变形。

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

为了定义具有形变目标的网格，每个网格图元（mesh primitive）可以包含一个形变目标数组.这些数组是字典形式，将属性的名称映射到访问器（accessors）的索引，这些访问器包含目标形变对应的几何位移数据。

mesh也可以包含一个**权重数组**,其中权重表示每个形变目标对网格最终渲染状态的贡献。

通过对不同形变目标应用不同的权重组合，可以实现多种效果，例如建模一个角色的不同面部表情：这些权重可以通过动画进行修改，以在几何体的不同状态之间进行插值。

buffers, bufferViews, accessors

缓冲区（buffers）包含用于3D模型几何体.动画和蒙皮的数据。(bufferViews）为这些数据添加了结构信息，而访问器（accessors）则定义了数据的确切类型和布局。

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

每个缓冲区通过一个**URI**引用一个二进制数据文件.该文件是一个具有特定字节长度（**byteLength**）的原始数据块。

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

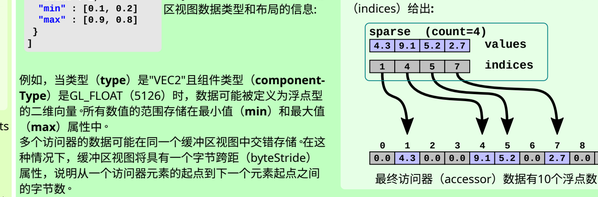
每个缓冲区视图（**bufferViews**）引用一个缓冲区（**buffer**），它具有字节偏移量（**byteOffset**）和字节长度（**byteLength**），用来定义属于该缓冲区视图的数据部分，并且可以选择性地包含一个OpenGL缓冲区目标（**target**）。

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

访问器（**accessors**）定义了如何解释缓冲区视图的数据.它们可能会定义缓冲区视图的额外偏移量（byteOffset），并包含关于缓冲区视图数据类型和布局的信息:

例如，当类型（**type**）是"VEC2"且组件类型（**component-Type**）是GL_FLOAT（5126）时，数据可能被定义为浮点型的二维向量.所有数值的范围存储在最小值（**min**）和最大值（**max**）属性中。

多个访问器的数据可能在同一个缓冲区视图中交错存储.在这种情况下，缓冲区视图将具有一个字节跨距（byteStride）属性，说明从一个访问器元素的起点到下一个元素起点之间的字节数。



Sparse accessors

当一个访问器（accesor）只有少数元素不同于默认值时（这在形变目标中经常出现），可以通过使用稀疏（sparse）数据描述的方式，以非常紧凑的形式给出这些数据：

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

  "sparse": {
   "count": 4,

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

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

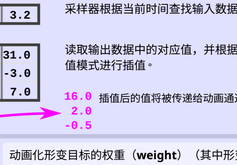
accessor 定义了数据类型（这里是标量浮点数)和元素总数(**count**).

访问器定义了数的类型（**type**），在这里是标量浮点值（**SCALAR**）以及元素总量（**co-unt**）。

values引用了包含这些稀疏数据值的缓冲区视图（**buffer-View**）。

目标索引（**indices**）通过引用一个缓冲区视图和组件类型（**componentType**）来定义。

values被写入最终的访问器数据中，数据的位置由索引（indices）给出:

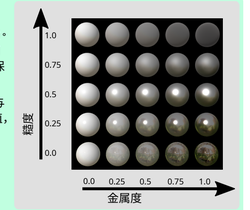


最终访问器（accessor）数据有10个浮点数：

materials

每个网格图元（mesh primitive）都可以关联到 glTF 资产中包含的一种材质。材质定义了该物体在基于物理的属性下应该如何进行渲染.通过这种方式，可以实现基于物理的渲染（PBR）技术，确保在不同渲染器中渲染对象的外观保持一致。

默认的材质模型是金属度-粗糙度模型.使用 0.0 到 1.0 之间的值来描述材质与金属的相似程度，以及物体表面的粗糙程度.这些属性既可以是个单独的数值，从而应用于整个物体，也可以从纹理中读取，从而应用于物体的不同部分。




```
"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]
 }
],
```

可以在金属度-粗糙度模型（Metallic-Roughness-Model）中定义材质，他们的属性汇总在 **pbrMetallicRoughness** 中:

baseColorTexture 是将会被应用到物体上的主要纹理，**baseColorFactor** 包含了颜色中红 -绿 -蓝分量以及透明度分量的缩放因子。如果不使用纹理，这些数值就会定义物体整体的颜色。
metallicRoughnessTexture 包含金属度与粗糙度，其中金属度数值使用颜色中的蓝色通道表示, 粗糙度数值使用颜色中的绿色通道表示。**metallicFactor** 跟 **roughnessFactor** 再与这些数值相乘.如果没有纹理，这两个因子将定义整个物体的反射特性。

除了通过金属度-粗糙度模型定义的属性外，材质还可能包含影响物体外观的其他属性：

- normalTexture 指向包含切线空间法线信息的纹理，以及一个应用于这些法线的缩放因子。
- occlusionTexture 指向用于描绘光线被遮蔽的表面区域的纹理，因此会渲染地更暗.这些信息的数值包含在纹理的"红色"通道中.**strength**参数是应用于这些数值的缩放因子。
- emissiveTexture 指向用于照亮物体表面某些部分的纹理：它定义了从表面发射的光的颜色。emissiveFactor 包含该纹理红 -绿 -蓝分量的缩放因子。

纹理中的材质属性

```
"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
  }
 }
],
```

材质中的纹理引用始终包含纹理的索引，也可能包含texCoord 集的索引.这些索引决定了渲染出的网格图元的TEXCOORD_<n> 中的数字n，其中 0 为默认值，网格图元中又包含着这个纹理的纹理坐标。

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

cameras

glTF资产中定义了摄像机（**cameras**），每个节点可以引用其中一个。

```
"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
   }
 }
]
```

摄像机有两种类型: 透视型**(perspective)** 以及正交型**(orthographic)**，他们定义了投影矩阵。

透视摄像机的远裁剪平面距离值**zfar**，是可选的. 当省略该值时，摄像机将使用一个用于无限远投影的特殊投影矩阵。

当某个节点引用摄像机时，将创建该摄像机的一个实例。此实例的摄像机矩阵由该节点的全局变换矩阵决定。

textures, images, samplers

纹理（**textures**）包含了可以用来渲染物体的纹理信息：材质通过引用纹理来定义物体的基本颜色，以及影响物体外观的物理属性。

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

纹理由两个部分组成：一个是对纹理源（**source**）的引用，即资产中的某个图像（**images**），另一个是对采样器（**sampler**）的引用。

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

图像（**images**）定义了纹理所使用的图像数据.这些数据可以通过 **URI**（即图像文件的位置）提供，或者通过对**bufferView** 以及一个mimeType来提供，其中mimeType定义了存储在bufferview中图像数据的类型。

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

采样器（**samplers**）描述了纹理的包裹模式和缩放方式。(这些常量值对应于 OpenGL 的常量，能够直接传递给 **glTexParameter**）。

skins

glTF 资产可以包含进行顶点蒙皮所需的信息.通过顶点蒙皮，可以基于当前姿态，根据骨架的骨骼来影响网格的顶点。

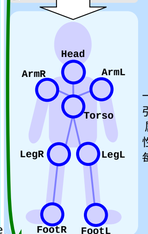
```
"nodes": [
 {
  "name" :
    "Skinned mesh node",
  "mesh": 0
  "skin": 0,
 },
 ...
 {
  "name": "Torso",
  "children":
    [ 2, 3, 4, 5, 6 ],
  "rotation": [...],
  "scale": [ ...],
  "translation": [...]
 },
 ...
 {
  "name": "LegL",
  "children": [ 7 ],
  ...
 },
 {
  "name": "FootL",
  ...
 },
 ...
],
```

一个节点可以引用网格（**mesh**），也可以引用蒙皮（**skin**）

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

蒙皮包含一个关节数组**joints**以及一个 **inverseBindMatrices**，其中joints数组内是定义骨架层次结构的节点的索引，inverseBindMatr-ices 则引用了包含每个关节对应矩阵的访问器（accessor)。

和场景结构类似，骨架层次结构也使用节点建模：每个关节节点可以有一个局部变换和一个子节点数组，骨架的骨骼通过关节之间的连接隐式地给出。



一个蒙皮网格的网格图元包含一个引用顶点位置访问器的 POSITION 属性，以及用于蒙皮的两个特殊属性：JOINTS_0 和 WEIGHTS_0，每个属性都引用一个访问器。

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

JOINTS_0 属性数据包含应该影响顶点的关节的索引

WEIGHTS_0 属性数据定义了关节对顶点影响的权重。

根据这些信息可以计算蒙皮矩阵。

这些内容在"计算蒙皮矩阵"一节中有详细说明。

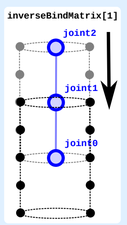
计算蒙皮矩阵

蒙皮矩阵描述了基于当前的骨架姿态网格顶点如何变换。蒙皮矩阵是对关节矩阵的加权结合。

计算关节矩阵

蒙皮引用了 inverseBindMatrices，这是一个访问器，包含每个关节对应的一个逆绑定矩阵.每个矩阵都会把网格转换到关节的局部空间。

对于蒙皮中关节数组所引用的每个节点，可以计算出一个全局变换矩阵.他根据当前关节的全局变换，将网格从关节的局部空间转换到全局空间，即globalJointTransform。

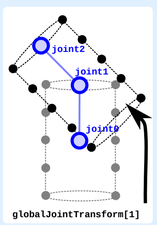


从这些矩阵出发, 每个关节的**jointMatrix** 可以如下计算

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

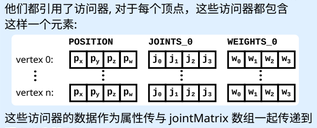
包含网格和蒙皮的节点的任何全局变换都可以通过预乘一个全局变换的逆矩阵来抵消这个全局变换。

对于基于 OpenGL 或 WebGL 的实现，jointMatrix数组将作为 "统一变量(uniform)" 传递给顶点着色器。



结合关节矩阵来创造蒙皮矩阵

蒙皮网格的图元包含 POSITION .joINT 和 WEIGHT 属性,，他们都引用了访问器, 对于每个顶点，这些访问器都包含这样一个元素:

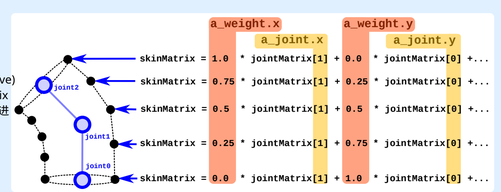


这些访问器的数据作为属性传与 jointMatrix 数组一起传递到顶点着色器中。

蒙皮矩阵(**skinMatrix**)在顶点着色器中计算. 它是那些索引值包含在JOINTS_0 属性内的关节矩阵，通过WEIGHTS_0进行加权之后的线性组合。

```
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;
}
```

在使用模型-视图-投影(model-view-perspective)矩阵变换前，skinMatrix 先根据骨架姿态对顶点进行变换。



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

glTF 资产可以包含动画.动画可以应用于节点的属性，这些属性定义了节点的局部变换，或者应用于形变目标的权重。

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

每个动画由两个元素组成：一个通道数组和一个采样器数组。

每个通道定义了动画的目标.这个目标通常引用一个使用索引的node以及作为动画特性名称的path。path可以是"平移(translation)"，"旋转(rotation)"或"缩放(scale)"，影响节点的局部变换，或者是"weights"，用来动画化被节点引用的网格的形变目标的权重，通道还引用了一个采样器，该采样器总结了实际的动画数据。

采样器引用了输入和输出数据，使用访问器（accessors）的索引来提供数据。
输入引用了一个包含标量浮点值的访问器，这些值是动画关键帧的时间。输出引用了一个包含了各自关键帧动画属性数值的采样器.采样器还定义了动画的插值模式，可以是"线性(LINEAR)插值"，"阶跃(STEP)插值"，或者"三次样条(CUBICSPLINE)插值"。

动画采样器

在动画过程中，"全局" 动画时间（以秒为单位）会不断推进。

采样器根据当前时间查找输入数据中的关键帧。

读取输出数据中的对应值，并根据采样器的插值模式进行插值。

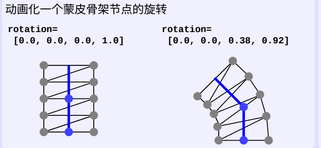
16.0 插值后的值将被传递给动画通道目标
2.0
-0.5

动画通道目标

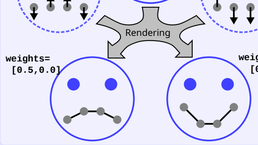
动画采样器提供的插值后的数值，可以用于不同的动画通道目标。

动画化一个节点的平移:

动画化一个蒙皮骨架节点的旋转



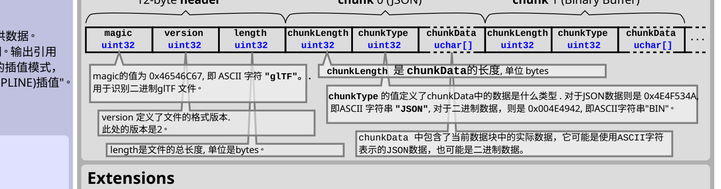
动画化形变目标的权重（**weight**）（其中形变目标是由附加到节点的网格图元定义）：



二进制的glTF 文件

在标准的 glTF 格式中，有两种方式包含外部的二进制资源（如缓冲区数据和纹理）：它们可以通过 URI 进行引用，或者使用数据 URI 嵌入到 glTF 的 JSON 部分中.当通过 URI 引用时，每一个外部资源都会产生一个新的下载请求.当它们嵌入为数据 URI 时，二进制数据的 base64 编码将显著增加文件大小。

为了克服这些缺点，有一种选择是将 glTF 的 JSON 和二进制数据合并成一个单一的二进制 glTF 文件.这是一种小端字节序文件，扩展名为 ".glb".该文件包含一个头部，提供关于数据版本和结构的基本信息，以及一个或多个包含实际数据的块.第一个块总是包含 JSON 数据，剩下的块包含二进制数据。



Extensions

glTF 格式允许通过扩展来添加新功能，或简化常用属性的定义。

当在 glTF 资产中使用扩展时，必须在顶层的 extensionsUsed 属性中列出。extensionsRequired 属性则列出了严格要求以正确加载资产的扩展。

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

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

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

扩展允许在其他对象的 extensions 属性中添加任意对象.这类对象的名称与扩展的名称相同，并且可以包含更多与该扩展相关的特定属性。

存在的扩展

有若干扩展在 Khronos 的 GitHub 仓库中开发和维护.完整的扩展列表可以在 https://github.com/KhronosGroup/glTF/tree/main/extensions/2.0. 找到.以下是 Khronos Group 认证的官方扩展：

- **KHR_draco_mesh_compression**: glTF 几何体可以使用 Draco 库进行压缩。
- **KHR_lights_punctual**: 添加对点光源.聚光灯和方向光的支持。
- **KHR_materials_clearcoat**: 允许为现有的 glTF PBR 材质添加透明涂层。
- **KHR_materials_ior**: 透明材料可以通过折射率进行扩展。
- **KHR_materials_iridescence**: 模拟薄膜效应，其中颜色随观察角度变化。
- **KHR_materials_sheen**: 为由布料纤维引起的背向散射添加颜色参数。
- **KHR_materials_specular**: 允许定义镜面反射的强度和颜色。
- **KHR_materials_transmission**: 更现实地模拟反射 -折射和透明度。
- **KHR_materials_unlit**: 允许定义不属于基于物理渲染的材料。
- **KHR_materials_variants**: 同一几何体上支持多种材质，可在运行时选择。
- **KHR_materials_volume**: 详细建模体积对象的厚度和衰减。
- **KHR_mesh_quantization**: 使用较小的数据类型更紧凑地表示顶点属性。
- **KHR_mesh_basisu**: 支持使用 Basis Universal 超压缩的 KTX v2 图像。
- **KHR_texture_transform**: 支持纹理的偏移 -旋转和缩放，用于创建纹理集。
- **KHR_xmp_json_ld**: 为场景 -节点 -网格和其他 glTF 对象添加 XMP 元数据支持。

更多资源

Khronos glTF 登陆页地址:
https://www.khronos.org/gltf

Khronos glTF GitHub 仓库:
https://github.com/KhronosGroup/glTF

Version 2.0d
gltF version 2.0
本概览并非规范性文档!

反馈:
gltf@marco-hutter.de