

glTF - what the ? 概念

GL Transmission Format(GLTF格式)概述



glTF由Khronos组织设计和定义，实现在网络上高效的传输3D内容。

glTF的核心是一个JSON文件，它描述了包含3D模型场景的结构和组成。此文件的顶级元素是：

scenes, nodes

场景的基本结构

cameras

场景视口的配置

meshes

3D几何对象

buffers, bufferViews, accessors

数据引用和数据布局的描述

materials

定义对象如何被渲染（材质）

textures, images, samplers


对象表面观感（纹理）

skins

顶点的蒙皮信息

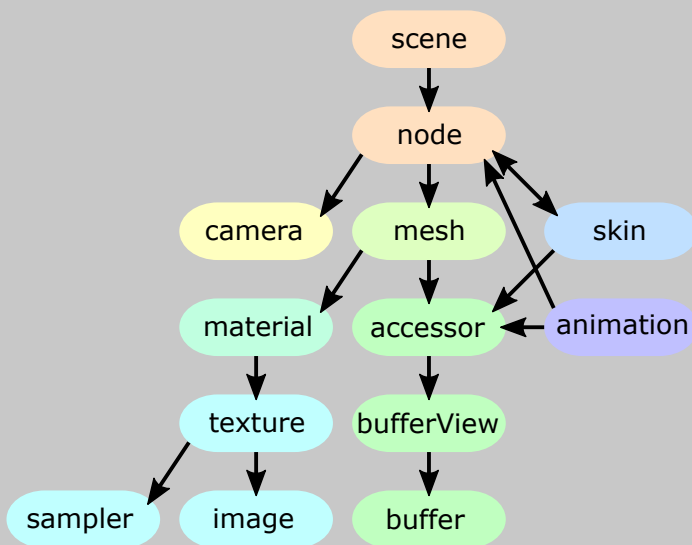
animations

动画:随时间变化的属性

这些元素被包含在数组当中。对象之间的引用通过数组的索引  进行关联。

也可以将整个资产存储在单个二进制glTF文件中。在这种情况下，JSON数据存储为字符串，后跟缓冲区或图像的二进制数据。

glTF资产顶层元素之间的概念关系如下图：



二进制数据引用

glTF资产的buffers和images可能引用包含渲染3D内容所需数据的外部文件：

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

Buffers 指向包含几何体或者动画数据的二进制文件

Images 指向包含模型纹理信息的图片文件

数据通过url来引用，但是也可以直接在json文件中包含数据urls。数据urls定义了MIME类型和一个base64编码的数据字符串。

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>



glTF and the glTF logo are trademarks of the Khronos Group Inc.

Version 2.0d
glTF version 2.0

This overview is non-normative!

Feedback:
gltf@marco-hutter.de



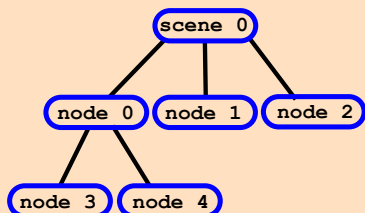
©2016-2022 Marco Hutter
www.marco-hutter.de

scenes, nodes

GITF 的 JSON 文件可以包含 **scenes** (和一个可选的默认 **scene**)。每一个scene可以包含nodes索引数组。

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

每个 **nodes** 可以包含子节点的索引数组。这就允许通过模型来表达一个场景的层次关系。



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

一个node可以包含本地变换。可以以一个 **matrix** 矩阵数组形式给出，也可以单独的以 **translation**, **rotation** 和 **scale** 属性表示，其中rotation是以一个四元数的形式给出。然后本地变换矩阵计算公式如下：

$$M = T * R * S$$

其中 **T**, **R** and **s** 是通过translation, rotation和scale创建的矩阵数组。

一个node的全部变换等于从root到相应的node的所有本地变换的乘积。

每个node也可能会引用到 **mesh** 或 **camera**，使用指向网格和相机数组的索引。然后将这些元素附加到这些节点上。在渲染过程中，这些元素的实例将被创建，并使用节点的全局变换进行变换。

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

一个node的translation（平移），rotation（旋转）和scale（缩放）属性也有可能成为一个动画的目标：动画会描述这些属性如何随着时间变化。因此被附加的对象将会允许对移动物体或者相机飞行进行建模。

Nodes也可以使用在顶点蒙皮中：节点的层次结构可以用来定义动画角色的骨架。然后node将会指向一个网格体或者蒙皮。蒙皮将会包含更多的信息关于如何将网格体基于当前的骨架姿势进行改变。

meshes

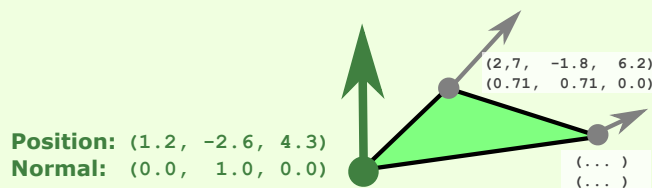
meshes 包含多个网格体。这些网格体指向需要渲染的几何体数据，

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

每一个mesh primitive有一个渲染 **mode**，他是一个常量，用来确定被渲染的是 POINTS, LINES, 或者 TRIANGLES.primitive会使用accessors的索引来指向 **indices**和顶点的**attributes**，渲染时应使用的**material**也通过材质的索引给出。

每个属性都是通过将属性名称映射到包含属性数据的访问器的索引来定义的。在渲染网格时，这些数据将被用作顶点属性。例如，属性可以定义顶点的POSITION和NORMAL。

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	...



网格可以定义多个变形目标。这样的变形目标描述了原始网格的变形。

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

要定义带有变形目标的网格，每个网格基本体可以包含一个 **目标**数组。这些是字典，将属性的名称映射到包含目标几何体位移的访问器的索引。

网格还可以包含一个**权重**数组，该数组定义了每个变形目标对网格最终渲染状态的影响。

结合多个具有不同权重的变形目标可以用于模拟角色的不同面部表情：可以通过动画修改权重，以在不同几何状态之间进行插值。

buffers, bufferViews, accessors

buffers 包含用于三维模型、动画、蒙皮的数据。
bufferViews 增加了这些数据的结构信息。
accessors 定义了数据了类型和布局。

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

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

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

每个**buffers** 都引用一个二进制数据文件，使用**URI**。它是一个具有给定**字节长度**的原始数据块的源。

每个**bufferView**都指向一个buffer。它通过**byteOffset**和**byteLength**，定义buffer中属于bufferView的部分，以及一个可选的OpenGL缓冲**目标**。

accessors定义了bufferView的的数据说明。它可以定义一个额外的**byteOffset**，指向前缀bufferView的开始，并包含有关bufferView数据的类型和布局的信息：

例如，当**类型**为"VEC2"且**componentType**为GL_FLOAT（5126）时，数据可以被定义为 float 精度二维向量。所有数据的范围存储在**min**和**max**属性中。

多个accessors的数据可以在bufferView内部交错。在这种情况下，bufferView定义了一个**byteStride**属性，该属性定义了从accessor元素的开始到下一个accessor元素的开始之间有多少字节。

Sparse accessors

当一个**accessor**的元素只有少数与默认值不同（这在变形目标中是常见情况），那么可以使用**稀疏数据**这种非常紧凑的形式描述此类数据：

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

  "sparse": {
    "count": 4,

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

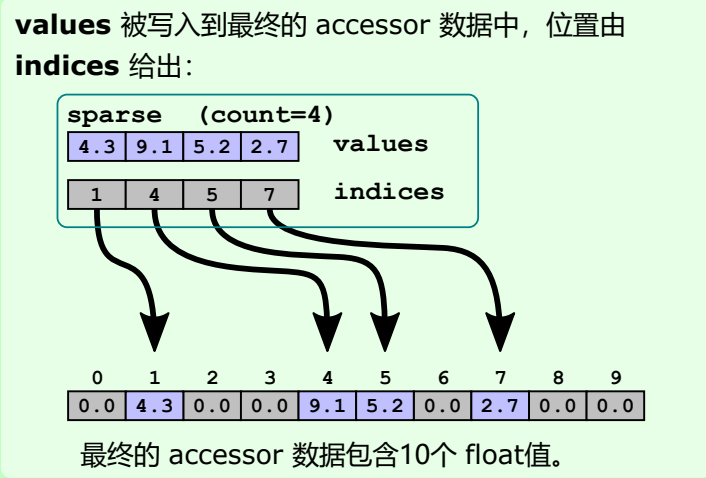
    "indices": {
      "bufferView": 1,
      "componentType": 5123
    }
  }
},
]
```

accessor 定义了数据的类型（这里为标量浮点值），以及总元素**计数**。

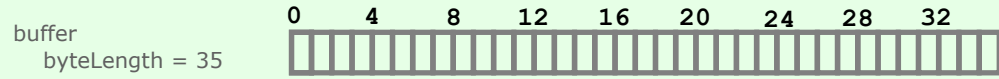
稀疏数据块包含稀疏数据元素的**计数**。

values 指向了包含稀疏数据值的 bufferView

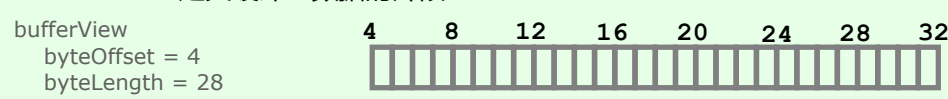
稀疏数据值的目标 **indices** 通过对bufferView和**componentType**的引用来定义。



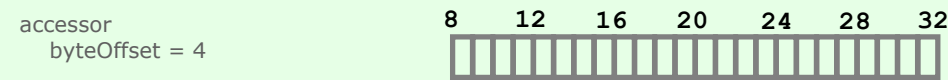
buffer 数据是从文件中读取的:



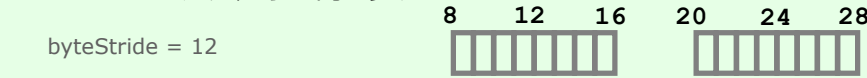
bufferView 定义缓冲区数据的片段



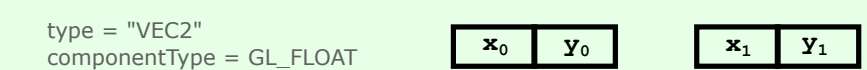
accessor 定义额外的偏移量:



bufferView 定义元素之间的步长



accessor 定义元素为2D float 向量:

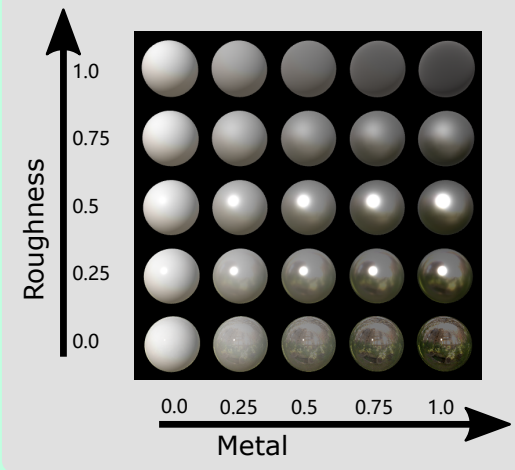


这些数据可以用于，例如，由网格基本体访问2D纹理坐标。首先，我们需要将**bufferView** 的数据绑定为一个OpenGL buffer，这可以通过使用glBindBuffer来实现。然后，我们可以通过将**accessor** 的属性传递给glVertexAttribPointer来定义该缓冲区为顶点属性数据。

materials

每个网格基本体可以引用glTF资产中包含的某个材质 **material**。材质描述了如何基于物理材质属性渲染对象。材质允许应用 **Physically Based Rendering (PBR)** 技术，以确保渲染对象的外观在所有渲染器中保持一致。

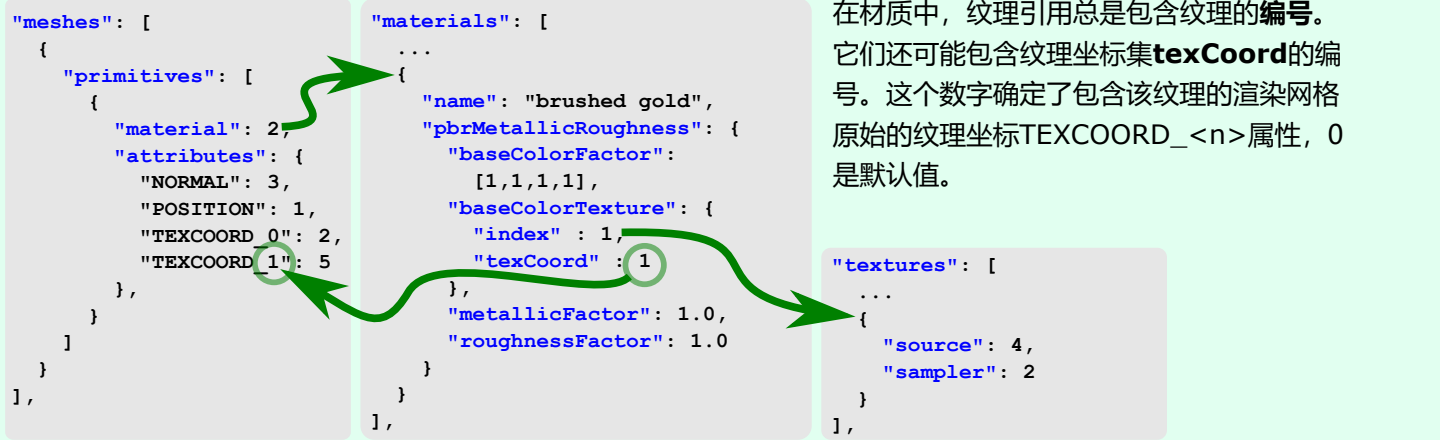
默认的材质模型是**金属度-粗糙度模型**。用0.0到1.0之间的值来描述材质特性与金属的相似程度，以及物体的表面粗糙程度。这些属性可以作为适用于整个物体的单独数值给出，也可以从纹理中读取。



```
"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]
}
],
```

- 在金属粗糙度模型中，属性设置都在定义在 **pbrMetallicRoughness** 的对象中：
- baseColorTexture**就是会被应用到物体上的主要纹理。而**baseColorFactor**则包含了颜色的红、绿、蓝和透明度分量的比例因子。如果没有使用纹理，这些值将决定整个物体的颜色。
 - metallicRoughnessTexture**中的“蓝色”通道包含了金属度值，“绿色”通道包含了粗糙度值。而**metallicFactor**和**roughnessFactor**将与这些值相乘。如果没有给定纹理，那么这些因子将决定整个物体的反射特性。
- 除了通过金属粗糙度模型定义的属性之外，材质还可能包含其他影响物体外观的属性：
- 法线纹理**normalTexture**指的是包含切线空间法线信息的纹理，以及将应用于这些法线的**scale**因子。。
 - 遮挡纹理 **occlusionTexture**指的是定义表面被遮挡而变暗的区域的纹理。这些信息包含在纹理的“红色”通道中。遮挡强度**strength**是应用于这些数值的比例因子。
 - 自发光纹理**emissiveTexture**是一种可用于光源物体表面部分的纹理：它定义了从光源表面发出的光的颜色。**emissiveFactor**包含此纹理的红色、绿色和蓝色分量的比例因子。

纹理中的Material属性



cameras

每个节点都可以引用在glTF资产中定义的一个 **camera**。

```
"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
    }
  }
]
```

有两种类型的相机：透视 **perspective** 和正交 **orthographic** 相机，它们定义了不同的投影矩阵。

透视相机的远截面距离 **zfar** 的值是可选的。当它被省略时，相机用一个特殊矩阵实现无穷远投影。

当某个节点引用了一个 camera 后，就会创建一个该 camera 的实例。该实例的相机矩阵由该节点的全局变换矩阵给出。

textures, images, samplers

Textures 包含了可以应用到渲染对象的纹理信息：通过材质 materials 引用纹理 textures，可以定义对象的基本颜色以及影响对象外观的物理属性。

```
"textures": [
  {
    "source": 4,
    "sampler": 2
  },
  ...
],
"images": [
  ...
  {
    "uri": "file01.png"
  },
  {
    "bufferView": 3,
    "mimeType": "image/jpeg"
  }
],
"samplers": [
  ...
  {
    "magFilter": 9729,
    "minFilter": 9987,
    "wrapS": 10497,
    "wrapT": 10497
  },
  ...
]
```

纹理 texture 由纹理源 **source** 的引用和一个采样器 **sampler** 引用组成，纹理源是资产中的一张图片 **image**。

Images 定义了用于纹理的图像数据。这些数据可以通过图像文件的位置 **URI** 或对 **bufferView** 的引用以及定义存储在 **bufferView** 中的图像数据类型的 **MIME 类型** 来提供。

采样器 **samplers** 描述了纹理的环绕和缩放方式。（这些常量对应于 OpenGL 常量，可以直接传递给 `glTexParameter`）。

skins

一个 glTF 文件可以包含执行顶点蒙皮所需的信息。顶点蒙皮可以让网格的顶点受到骨骼的影响，根据骨骼当前的姿势而变化。

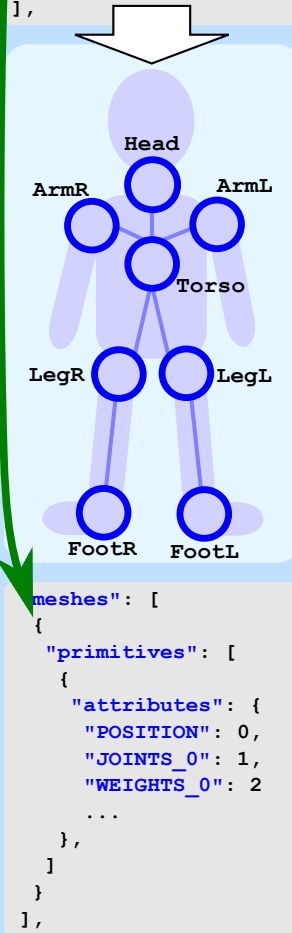
```
"nodes": [
  {
    "name": "Skinned mesh node",
    "mesh": 0,
    "skin": 0,
    ...
  },
  {
    "name": "Torso",
    "children": [ 2, 3, 4, 5, 6 ],
    "rotation": [...],
    "scale": [...],
    "translation": [...]
  },
  ...
  {
    "name": "LegL",
    "children": [ 7 ],
    ...
  },
  ...
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

引用网格 **mesh** 的节点也可能引用蒙皮 **skin**。

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

蒙皮 **skins** 包含关节数组 **joints**，这些关节是定义骨骼层次结构的节点索引，还包括 **inverseBindMatrices** (反向绑定矩阵)，这是一个引用到访问器的数据，该访问器为每个关节存储一个矩阵。

骨架层次结构与场景结构一样，都是通过节点来建模的：每个关节节点可以有本地变换和子节点的数组，而骨架的“骨骼”则通过关节之间的连接来隐式给出。



蒙皮网格的基本体包含了指向顶点位置访问器的 **POSITION** 属性，以及两个进行蒙皮所需的特殊属性：一个 **JOINTS_0** 属性和一个 **WEIGHTS_0** 属性，分别指向一个访问器。

JOINTS_0 属性数据包含了应该影响顶点的关节索引。

WEIGHTS_0 属性数据定义了权重，表示关节对顶点影响的强度。

根据这些信息，可以计算出蒙皮矩阵。

这在 **"Computing the skinning matrix"** 计算蒙皮矩阵中有详细说明。

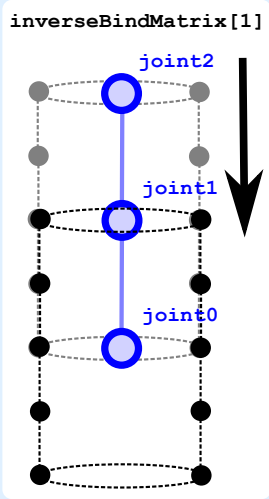
Computing the skinning matrix 计算蒙皮矩阵

蒙皮矩阵描述了网格顶点根据骨骼当前姿势的变换方式。它是关节矩阵的加权组合。

计算关节矩阵

蒙皮引用了反向绑定矩阵 **inverseBindMatrices**。这是一个包含每个关节的反向绑定矩阵的访问器。这些矩阵将网格转换到关节的局部空间。

对于蒙皮中**关节索引**出现的每个节点，可以计算出一个全局变换矩阵。它会根据关节的当前全局变换，将网格从关节的局部空间变换到全局空间，称为 **globalJointTransform**。

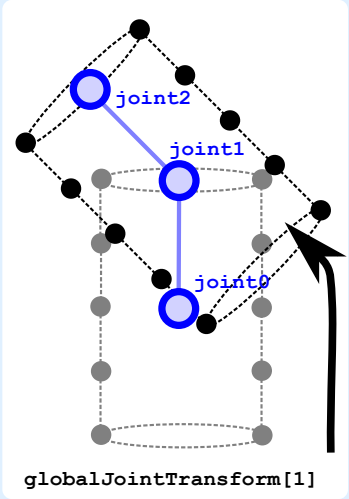


从这些矩阵中，可以为每个关节计算出一个关节矩阵 **jointMatrix**：

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

包含网格和蒙皮的节点上的任何全局变换都会被通过将关节矩阵与该变换的逆矩阵进行预乘而抵消掉

对于基于OpenGL或WebGL的实现，jointMatrix数组将作为uniform传递到顶点着色器中。



组合关节矩阵以创建蒙皮矩阵

蒙皮网格的基本元素包含 POSITION、JOINT 和 WEIGHT 属性，这些属性引用访问器。这些访问器对于每个顶点包含一个元素：

	POSITION	JOINTS_0	WEIGHTS_0
vertex 0:	P _x P _y P _z P _w	j ₀ j ₁ j ₂ j ₃	w ₀ w ₁ w ₂ w ₃
⋮	⋮	⋮	⋮
vertex n:	P _x P _y P _z P _w	j ₀ j ₁ j ₂ j ₃	w ₀ w ₁ w ₂ w ₃

这些访问器中的数据将作为属性与 jointMatrix 数组一起传递给顶点着色器。

在顶点着色器中，计算出**蒙皮矩阵**。它是关节矩阵的线性组合，其索引包含在JOINTS_0属性中，并使用WEIGHTS_0值进行加权：

```
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;
}
```

蒙皮矩阵会在顶点被模型视图投影矩阵进行变换之前，根据骨架的姿态变换顶点。



animations

glTF资产可以包含动画**animations**。动画可以应用于定义节点局部变换的节点属性，或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
      }
    ]
  }
]
```

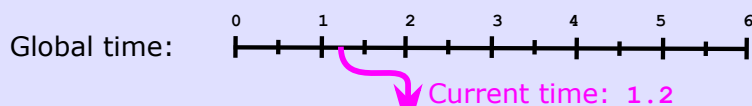
每个animation由两个元素组成：通道 **channels**数组和采样器**samplers**数组。

每个通道都定义了动画的目标**target**。通常，这个目标是一个**节点**，使用节点的索引来表示，以及一个路径，表示动画属性的名称。路径可以是"translation"（平移），"rotation"（旋转）或"scale"（缩放），影响节点的局部变换，或者是"weights"（权重），用于对节点引用的网格的形变目标的权重进行动画。通道还引用一个**采样器**，该采样器包含实际的动画数据。

采样器就是动画的**输入**和**输出**数据，它使用提供数据的访问器的索引。输入的数据是一个带有标量浮点值的访问器，这些值就是动画关键帧的时间。输出的数据是一个包含动画属性在相应关键帧上的值的访问器。采样器还定义了动画的**插值**模式，可以是 "LINEAR"、"STEP" 或 "CUBICSPLINE" 。

Animation samplers 动画采样器

在动画过程中，会不断推进一个“全局”动画时间（以秒为单位）。



动画采样器的输入访问器数据，包含了关键帧的时间。

0.0	0.8	1.6	2.4	3.2
-----	-----	-----	-----	-----

动画采样器的输出访问器数据，包含了动画属性的关键帧值。

10.0	14.0	18.0	24.0	31.0
5.0	3.0	1.0	-1.0	-3.0
-5.0	-2.0	1.0	4.0	7.0

采样器在**输入**数据中查找当前时间的关键帧。

读取**输出**数据中对应的数值，并根据采样器的**插值**模式进行插值。

16.0
2.0
-0.5
插值后的数值被转发到动画通道目标
Animation channel targets.

Animation channel targets 动画通道目标

动画采样器提供的插值数值可以应用于不同的动画通道目标。

节点**平移**动画：

translation=[2, 0, 0]

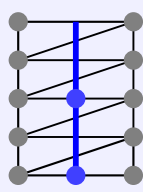


translation=[3, 2, 0]

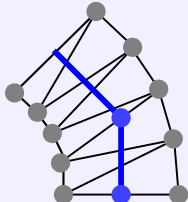


对骨骼节点的**旋转**进行动画处理：

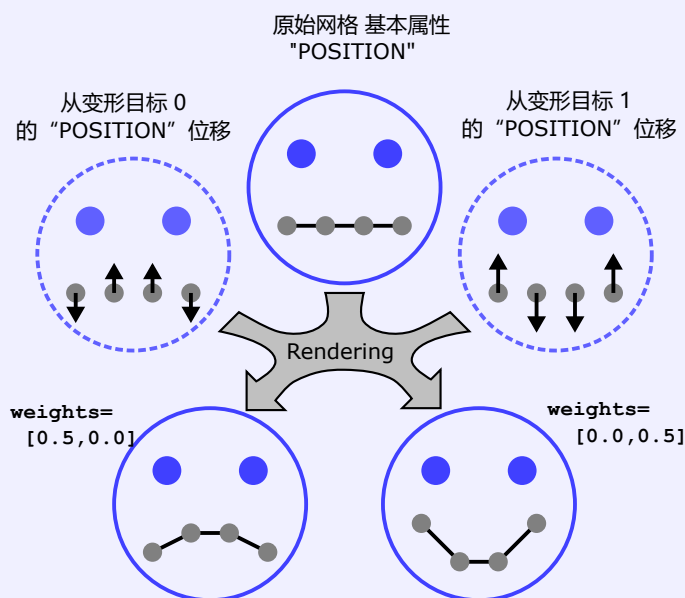
rotation=[0.0, 0.0, 0.0, 1.0]



rotation=[0.0, 0.0, 0.38, 0.92]



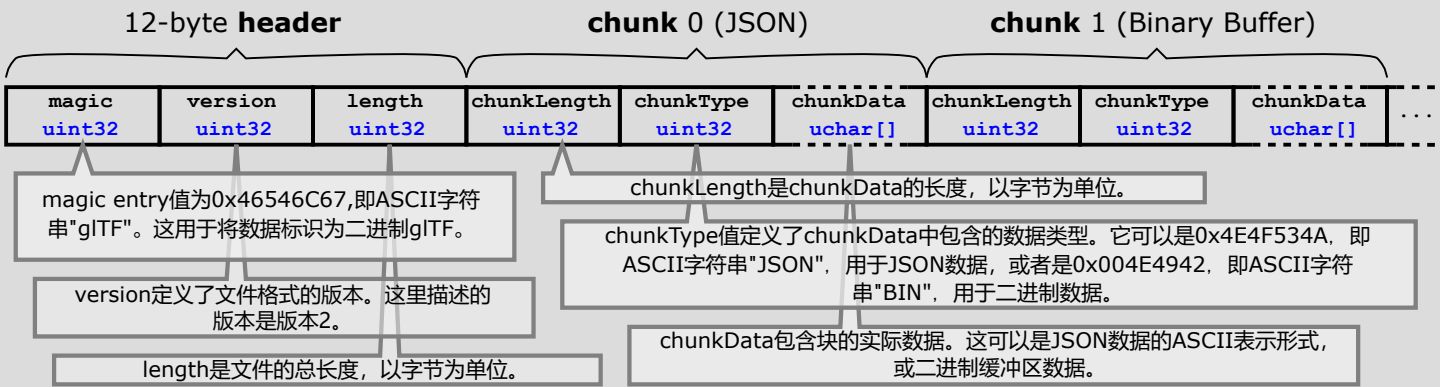
为连接到节点的网格的基本形状定义的变形目标的**权重**进行动画处理：



Binary glTF files

在标准的glTF格式中，包含外部二进制资源（如缓冲数据和纹理）有两种选项：可以通过URI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，每个外部资源都会引发一个新的下载请求。当它们作为数据URI嵌入时，二进制数据的Base64编码会显著增加文件大小。

为了克服这些缺点，有一个选项，即将glTF JSON和二进制数据合并成一个单一的**二进制glTF**文件。这是一个小端文件，扩展名为".glb"。它包含一个**header**，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**chunks**。第一个块始终包含JSON数据，其余的块包含二进制数据。



Extensions

glTF格式允许扩展添加新功能，或简化常用属性的定义。

在glTF资产中使用扩展时，必须在顶层的**extensionsUsed**属性中列出它。

extensionsRequired属性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。此类对象的名称与扩展的名称相同，它可以包含进一步的、特定于扩展的属性。

Existing extensions

很多扩展在Khronos GitHub存储库上进行开发和维护。完整的扩展列表可以在以下位置找到：<https://github.com/KhronosGroup/glTF/tree/main/extensions/2.0>。

以下是一些由Khronos Group批准的官方扩展：

- **KHR_draco_mesh_compression**: glTF几何体可以使用Draco库进行压缩。
- **KHR_lights_punctual**: 增加了对点光源、聚光灯和方向光（平行光）的支持。
- **KHR_materials_clearcoat**: 允许向现有的glTF PBR材质添加清漆涂层。
- **KHR_materials_ior**: 扩展透明材料折射率。
- **KHR_materials_iridescence**: 模型薄膜效应，其中色调取决于视角。
- **KHR_materials_sheen**: 为布料纤维引起的反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**: 更真实的反射、折射和不透明度建模。
- **KHR_materials_unlit**: 允许NPR
- **KHR_materials_variants**: 在相同的几何体上定义多个材料，以便在运行时进行选择。
- **KHR_materials_volume**: 详细模拟体积对象的厚度和衰减。
- **KHR_mesh_quantization**: 使用较小数据类型的顶点属性的更紧凑表示。
- **KHR_mesh_basisu**: 添加对具有Basis Universal超级压缩的KTX v2图像的支持。
- **KHR_texture_transform**: 支持纹理的偏移、旋转和缩放，以创建纹理图集。
- **KHR_xmp_json_ld**: 支持场景、节点、网格和其他glTF对象上的XMP元数据。