glTF なぁにそれ ?

GL 転送フォーマットの
基本的な概要を紹介しよう！

glTF2.0 対応版!

glTF と呼ばれるフォーマットは、ネットワークを介した 3Dコンテンツの効率的な転送のために Khronos Group により設計・制定されるものだ。

glTF の基本部分は、3D モデルを含むシーンの構造や構成を記述する **JSON** ファイルである。
以下に示すトップレベルの要素を見てみよう。

scenes, nodes
scene の基本構造

cameras
scene の視点設定

meshes
3Dオブジェクトのジオメトリ情報

buffers, bufferViews, accessors
情報参照と情報レイアウト情報

materials
オブジェクトのレンダリング方法の定義

textures, images, samplers
オブジェクト表面の外観

skins
頂点スキニングに関する情報

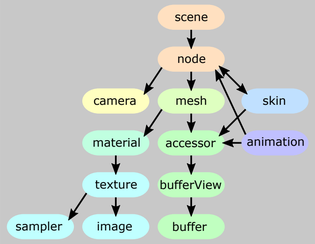
animations
時系列変化に関わる情報

各要素は配列の形式で格納される。オブジェクトに対する参照は、配列の中にあるオブジェクトを指し示す番号、つまりインデックス番号で実装される。

また、単一のバイナリ glTF ファイルには、
アセットと呼ばれる情報全体の格納が可能だ。
その際には、string 型で記述された JSON の後方に、buffer または image のバイナリ情報を記述しよう。

Concepts

glTF アセットにおけるトップレベルの要素間が、
どのように関係するか表した概念図を見てみよう。



Binary data references

glTF アセット概念図の下側にある image と buffer に注目しよう。image と buffer は 3D コンテンツのレンダリングに必要な外部ファイルを参照できる。

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

buffers で参照できる情報は、ジオメトリ情報または animation 情報を含むバイナリファイル（.BIN）である。
images で参照できる情報は、モデルの texture 情報を含む画像ファイル（PNG、JPG ...）である。

通常は情報を URI を用いて参照するが、情報 URI を使用し JSON に直接埋め込むこともできる。
情報 URI は下図のように MIME タイプを定義し、base64 でエンコードされた文字列として格納される。

Buffer data:

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

Image data (PNG):

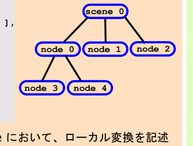
```
"..."
```

scenes, nodes

gltF JSON には **scenes**（既定の **scene** を含む）を記述できる。各 scene には、node のインデックスの配列が記載される。

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

各 **nodes** には、
children のインデックスの配列を記述する。これにより、簡易的な scene の階層構造を記述できる。




```
"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 において、ローカル変換を記述できる。これは列優先**matrix** の配列、もしくは **translation**（平行移動）**rotation**（回転）**scale**（拡大縮小）の特性どちらか一方を用いる。ただしrotation は quaternion で表される。
後者の場合、ローカル変換行列は次のように定義される。

M = T * R * S

ここで、T、RおよびSは、translation、rotation、scale から作られる matrix である。
node のグローバル変換は、root から各 node へのパス上のすべてのローカル変換の積で求まる。

各 node は、**mesh** 配列または **camera** 配列を指すインデックスにより mesh と camera を参照する。
各要素は、各 node に関連付けられる。レンダリングにおいて、各要素のインスタンスが作成され、node のグローバル変換行列により変換される。

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

node の平行移動、回転、スケールの特性は animation の目標になることもある。
接続されたオブジェクトはそれに応じて移動し、移動するオブジェクトや camera の飛行などをモデル化できる。

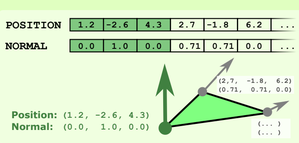
また、node は頂点 skinning の際にも使用される。node 階層は、animation 化されたキャラクターの骨格を定義できる。
次に、node は mesh と skin を参照する。
skin には、現在の姿勢に基づいて mesh がどのように変形されるかに関する詳しい情報が記述されている。

meshes

meshs には複数の **primitives** を記述できる。primitive には、mesh のレンダリングに必要なジオメトリ情報を参照するインデックスを記述する。

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

各 primitive には
レンダリングの **mode** が設定でき、POINTS、LINES、TRIANGLES として
レンダリングする必要があるかどうかを示す定数を入れる。
また、primitive は、この情報の accessors の indices を使用し、頂点の **indices** と **attributes** を参照する。レンダリングに使用される **material** はまた、material のインデックスにより与えられる。
各 attribute では属性名を、属性情報を含む accessor のインデックスとの対応付けを行うことで定義される。
この情報は、mesh のレンダリング時に頂点属性として使用される。属性の例としては、頂点の POSITION や NORMAL などが挙げられる。



また、mesh はもとの mesh の変形を記述する複数のモーフの target を定義できる。

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

モーフの target を mesh に定義するには、各 primitive に **targets** の配列を記述しよう。これらは、属性の名前を、target のジオメトリの変位を含む accessor のインデックスにマップする Dictionary だ。

mesh には、各モーフの target の最終レンダリング状態へ活用される **weights** 配列も記述できる。

異なる重みを有する複数のモーフ target を組み合わせることで、例えばキャラクターの様々な表情をモデル化できる。weight は，ジオメトリの異なる状態間を補間する必要のある animation で活用される。

buffers, bufferViews, accessors

buffers には、model、animation、および skinning のジオメトリに使用される情報が記載される。
bufferViews は、buffers がどのように構成されるかを記述される。また、**accessors** には情報の具体的な型と構成が記述される。

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

各 **buffers** は、**URI** を使用しバイナリ情報ファイルを参照する。これは指定された **byteLength** を持つ生情報の1ブロックのソースである。

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

各 **bufferViews** は、
一つの buffer を参照する。
その時 bufferView に関する buffer の一部と任意の OpenGL buffer **target** を定義する **byteOffset** と **byteLength** がある。

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

accessors は、
bufferView の情報が
どのように解析されるか
定義する。bufferView の
開始位置を示す **byteOffset** を定義し、bufferView 情報の type とレイアウトに関する情報を記述する：

例えば情報の **type** が「VEC2」で **componentType** が GL_FLOAT（5126）である場合、浮動小数点値の 2D ベクトルとして解釈される。
すべての値の範囲は、**min**、**max** 特性で設定する。
また、複数の accessor の情報は bufferView 内でインターリーブされてもよい。
この場合 bufferView には、accessor の1つの要素の開始点と次の要素の開始点との間のバイト数を示す **byteStride** 特性を設定しよう。

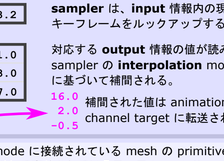
Sparse accessors

accessor のいくつかの要素だけが既定値（モーフ目標の場合が多い）と異なる場合、sparse 記述を使用して非常にコンパクトな形式で記述できる。

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

    "sparse": {
      "count": 4,

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

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

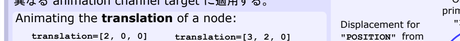
accessor は、情報の型（ここではスカラ浮動小数点値）、および総エレメントの **count** を定義する。
sparse ブロックは、sparse 情報の **count** を持つ。
values は、
sparse 情報値を含む bufferView を参照する。
sparse 情報値の target **indices** は、
bufferView と **componentType** への参照で定義される。

"values"は、インデックスによって指定された位置で最終 accessor 情報に書き込まれる。

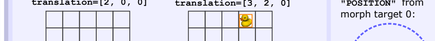


Final accessor data with 10 float values

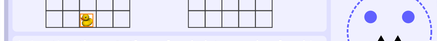
The **buffer** data is read from a file:



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

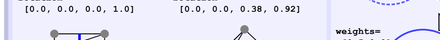


The **accessor** defines an additional offset:



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

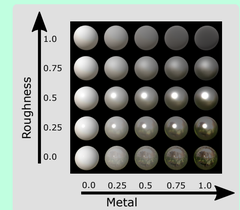
The **accessor** defines that the elements are 2D float vectors:



本情報は、多くの場合
mesh primitive によって
2D texture 座標にアクセスするために使用する。
bufferView の情報は、
glBindBuffer を使用して
OpenGL bufferとして
バインドする。
次に、**accessor** 特性を
使用して、bufferView
buffer がバインドされているときに glVertexAttribPointer に渡すことで、この buffer を頂点属性情報として定義できる。

materials

各 mesh primitive は、glTF アセットに含まれる **materials** のうち一つを参照している。material の特性は、物理的な特性に基づく値を用いて、オブジェクトのレンダリング方法を記述している。
これにより **Physically Based Rendering (PBR)** 技術を採用できるため、オブジェクトのレンダリング結果が、すべてのレンダラー間で一貫していることが保証される。
既定の material のモデルは **Metallic-Roughness-Model** だ。0.0と1.0の間の値は、material 特性が金属とどれほど類似するか、およびオブジェクトの表面がどれほど粗いかを表すために使用される。各特性は、オブジェクト全体に適用される個々の値として決定してもよいし、texture から読み込まれてもよい。




```
"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]
  }
],
```

Metallic-Roughness-Model の material を定義する特性は、**pbrMetallicRoughness** オブジェクトにまとめられている。

baseColorTexture は、オブジェクトに適用される主な texture だ。baseColorFactor には、色の赤、緑、青、およびアルファ成分のスケーリング係数を記述する。 texture を使用しない場合には、これらの値を用いてオブジェクト全体の色を定義することになる。
metallicRoughnessTexture には、「青」チャンネルに metallicFactor を、「緑」チャンネルに roughnessFactor を定義する。
metallicFactor と **roughnessFactor** は material における各値に乗算される。texture が与えられていない場合、各要素はオブジェクト全体の特性を定義することになる。
また、Metallic-Roughness-Modelを介して定義された特性の他に、オブジェクトにはその外観に影響を与える他の特性が記載される場合がある。

- **normalTexture** には、
  接線空間の通常の情報と、これらの法線に適用される **scale** を含む texture を参照する。
- **occlusionTexture** には、
  光から遮蔽されてより暗くレンダリングされる表面の領域を定義する texture を参照する。この情報は texture の「赤」チャンネルに含まれる。オクルージョンの **strength** は、各値に適用されるスケーリング係数だ。
- **emissiveTexture** は、
  オブジェクト表面の一部を照らすために使用できる texture を参照する。つまり、表面から放出されるライトの色を定義できる。
  emissiveFactor には、この texture の赤、緑、青の成分のスケーリング要素が記述される。

Material properties in textures

material 内の texture 参照には常に texture の **index** が用いられる。
texCoord セットインデックスも同様だ。
この texture の texture 座標を含むレンダリングされた mesh primitive の TEXCOORD_ <n> 属性を決定する数値だ。既定値は0となる。

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

Further resources

Khronos glTF ホームページ:
https://www.khronos.org/gltf

Khronos glTF GitHub リポジトリ:
https://github.com/KhronosGroup/glTF

Version 2.0a
glTF version 2.0
本概要は参考でお願いします。

Feedback:
gltf@marco-hutter.de

Translated to Japanese by
Takuto Takahashi
randall2835@gmail.com

KHRONOS GROUP
CONNECTING SOFTWARE TO SILICON

glTF と glTFロゴは
Khronos Group Inc. の商標です。

cameras

各 nodes は、glTF アセットに定義される **cameras** のうち一つを参照する。

```
"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
    }
  }
]
```

cameras には、
perspective または
orthographic どちらか2種類を選択でき、
それにより投影行列が定義される。

ここでの透視投影における **zfar** の far クリッピング平面の距離の値の記述は任意である。
これを省略すると、camera は無限投影を行う特殊な投影行列を使用する。

"node"が"camera"を参照すると、この camera のインスタンスが生成される。 このインスタンスの camera 行列は、node のグローバル変換行列によって決定される。

textures, images, samplers

textures には、レンダリングされたオブジェクトに適用される texture に関する情報が含まれる。
texture は、オブジェクトの基本的な色やオブジェクトの外観に影響を与える物理的な特性を定義するために material によって参照される。

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

texture は、
texture の **source** と
アセットのうちの一つの
images への参照と
sampler への参照から
構成される。

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

images は、
texture に使用される
画像情報を定義する。
この情報は、画像ファイルの場所である **URI** か、
bufferView に格納されている画像情報の型を示す **MIME Type** への参照により定義される。

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

samplers は、texture の折り返しとスケーリングを記述する。（この定数値は、glTexParameter に直接渡すことが可能な OpenGL 定数と対応している）

skins

glTF アセットには、頂点 skinning に必要な情報を記述できる。頂点 skinning を使用すると、
mesh の頂点が、現在の姿勢に基づいて影響を受ける。

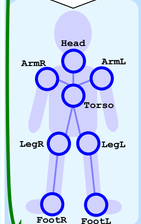
```
"nodes": [
  {
    "name" :
      "Skinned mesh node",
    "mesh": 0
    "skin": 0,
  },
  ...
  {
    "name": "Torso",
    "children":
      [ 2, 3, 4, 5, 6 ],
    "rotation": [...],
    "scale": [ ...],
    "translation": [...]
  },
  ...
  {
    "name": "LegL",
    "children": [ 7 ],
    ...
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

mesh を参照する node は、**skin** を参照する。

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

skins は、
骨格階層を定義する node の indices である **joints** 配列と、各 joint の行列が含まれる accessor への参照である **inverseBindMatrices** を記述する。

姿勢の階層構造はちょうど scene 構造と同じように、node でモデル化される。
各関節 node は、
ローカル変換と children の配列を有していてもよく、および姿勢の「bones」は関節間の接続として、自動的に定義される。



skinnin された mesh の mesh primitive には、頂点位置の accessor を参照するPOSITION 属性と、skinning に必要な2つの特別な属性が含まれる。

JOINTS_0およびWEIGHTS_0 属性。各属性は accessor を参照する。 JOINTS_0属性情報には、頂点に影響を与える関節のインデックスが含まれる。

WEIGHTS_0属性情報は、関節が頂点にどれだけ強く影響するかを示す weights を定義する。
この情報から
skinning matrix を計算する。

これについては、
「**Computing the skinning matrix**」で詳しく説明する。

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

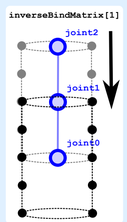
Computing the skinning matrix

skinning 行列は、姿勢の現在の姿勢に基づいてメッシュの頂点がどのように変換されるかを記述する。
このskinning 行列は、関節行列の重み付けされた組み合わせで表現される。

Computing the joint matrices

skin は **inverseBindMatrices** を参照する。
これは、各関節に対して1つの逆バインド行列を含む accessor だ。各行列のそれぞれは、mesh を関節のローカル空間に変換する。

skin の **joints** に現れるインデックスを持つ
各 node について、グローバル変換行列が計算される。
関節の現在のグローバル変換に基づいて、
関節のローカル空間から mesh を変換する。
これは globalJointTransform と呼ばれる。

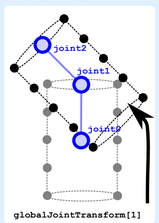


各行列から、各 joints に jointMatrix を計算する：

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

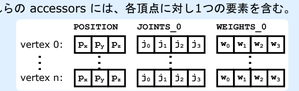
mesh と skin を含む node のグローバル変換は、この変換の逆行列を関節行列の事前の乗算によって相殺される。

OpenGL か WebGL に基づく実装の場合、jointMatrix 配列は、uniform 配列として頂点シェーダに渡される。



Combining the joint matrices to create the skinning matrix

skin mesh の primitive は accessor への参照を通じて、POSITION、JOINT、およびWEIGHT の attribute を持つ。これらの accessors には、各頂点に対し1つの要素を含む。

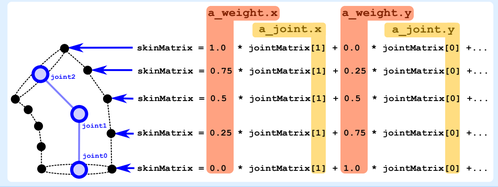


各 accessors の情報は、jointMatrix 配列と共に頂点シェーダに属性として渡される。
頂点シェーダでは、skinMatrix が計算される。
JOINTS_0 属性にインデックスが含まれ、WEIGHTS_0 値で重み付けされた関節行列の線形結合だ。

Vertex Shader

```
uniform mat4 u_jointMatrix[12];
attribute vec3 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;
}
```

skinMatrix は、
モデルビュー
透視行列で
変形される前に、
姿勢ポーズに基づき
頂点を変換する。



COLLADAにおけるスキニングのWikiはこちら: https://www.khronos.org/collada/wiki/Skinning
Section 4-7 in the COLLADA specification: https://www.khronos.org/files/collada_spec_1_5.pdf

(COLLADAにおける頂点スキニングはglTFのそれと結構似ているのだ)

animations

glTF アセットには **animations** の格納が可能だ。animation には、node のローカル変換を定義する node の特性、またはモーフ target の weights に適用できる。

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

各 animation は、**channels** の配列と **samplers** の配列の
2つの要素で構成される
各 channel は、animation の **target** を定義する。この target は通常、この node のインデックスを使用する **node** と、animation 化された特性の名前である **path** を参照する。path は node により参照される mesh のモーフ target の重みを animation 化するため、node のローカル変換に影響を及ぼす平行移動、回転または拡大縮小を記述できる。また channel は animation 情報を要約する **sampler** も参照する。

sampler は、情報を提供する accessor のインデックスを使用して、**input** と **output** 情報を参照する。input は、animation のキーフレームの時間であるスカラ浮動小数点値を参照して accessor を参照する。
output は、それぞれのキーフレームで animation 化された特性の値を含む accessor を参照する。sampler はまた、animation のための
LINEAR、 STEP、 CATMULLROMSPLINE 、CUBICSPLINE のような animation の **interpolation** mode を定義する。

Animation samplers

animation 中、「グローバル」animation 時間（秒）が進行する。

sampler は、**input** 情報内の現在時間のキーフレームをルックアップする。

対応する **output** 情報の値が読み出され、sampler の **interpolation** mode に基づいて補間される。

補間された値は animation channel target に転送される。

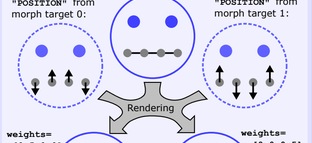
Animation channel targets

animation channel targetは、
animation sampler によって提供される補間値を、異なる animation channel target に適用する。

Animating the **translation** of a node:

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

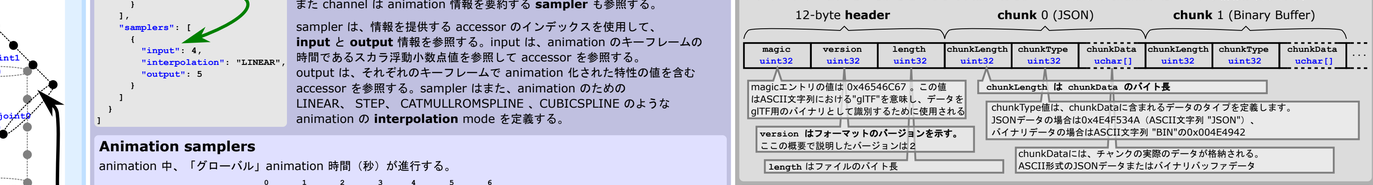
node に接続されている mesh の primitive に対して定義されているモーフ target の **weights** を animation 化する：



Binary glTF files

標準の glTF 形式では，buffer 情報や texture などの外部バイナリリソースを含むため、二種類のオプションが用意されている。URI 経由での参照または、glTF の JSON 部分に情報 URI 形式で埋め込む方法の二種類だ。
URI を介して参照されると、各外部リソースは
新しいダウンロード要求を自動的に行う。
ただし、情報 URI として埋め込また場合、
バイナリ情報の base64 エンコーディングは
ファイルサイズを大幅に増加させてしまう欠点がある。

これらの欠点を克服するために、glTF JSON とバイナリ情報を一つの **binary glTF** ファイルに結合するオプションがある。
これはリトルエンディアンファイルで、
拡張子は .glb だ。情報のバージョンと構造、および実際の情報を含む1つ以上の **chunks** に関する基本情報を示す **header** が含まれる。
最初の chunk には常に JSON 情報が含まれ、
残りの chunk にはバイナリ情報が含まれる。



Extensions

glTF では、拡張機能によって新たな機能を追加したり、特性の定義を簡素化することができる。

拡張機能が glTF アセットに導入される、拡張機能は最上位の **extensionsUsed** 特性にその拡張機能を列挙する必要がある。
extensionsRequired 特性には、アセットを適切に読み込むための必要な拡張機能を列挙する。

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

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

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

拡張機能では、
他のオブジェクトの
extensions 特性に
任意のオブジェクトを
追加できる。
そのようなオブジェクトの名前は拡張の名前と同じであり、さらに
拡張固有の特性を
含むこともできる。

Existing extensions

以下の拡張機能は Khronos GitHub リポジトリによって開発・管理されている:

- **Specular-Glossiness Materials**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/Khronos/KHR_materials_pbrSpecularGlossiness
  この拡張機能はデフォルトの Metallic-Roughness material モデルの代替品となる。
  従来の鏡面反射と光沢の特性に基づく material の設定が可能となる。
- **Common Materials**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/Khronos/KHR_materials_cmnBlinnPhong
  この拡張機能により、非物理的な材料の定義が容易になる。
  ここでは、CAD アプリケーションでよく使用される Blinn-Phong モデルが利用される。
  material の特性として、拡散色、鏡面反射色、発光色、光沢値などがある。
- **Common Lights**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/Khronos/KHR_lights
  この拡張機能はよく使われる種類の光源を scene 階層に追加することができる。具体的には、点光源、スポットライトと指向性光源がある。光源は、scebe 階層の node に記述することができる。
- **WebGL Rendering Techniques**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/Khronos/KHR_technique_webgl
  この拡張機能により、OpenGL または WebGL で glTF アセットをレンダリングする際に使用される独自の GLSL シェーダを定義できる。