

glTF-Tutorials

Previous: [Introduction](#)

[Table of Contents](#)

Next: [A Minimal glTF File](#)

The Basic Structure of glTF

The core of glTF is a JSON file. This file describes the whole contents of the 3D scene. It consists of a description of the scene structure itself, which is given by a hierarchy of nodes that define a scene graph. The 3D objects that appear in the scene are defined using meshes that are attached to the nodes. Materials define the appearance of the objects. Animations describe how the 3D objects are transformed (e.g., rotated or translated) over time, and skins define how the geometry of the objects is deformed based on a skeleton pose. Cameras describe the view configuration for the renderer.

The JSON structure

The scene objects are stored in arrays in the JSON file. They can be accessed using the index of the respective object in the array:

```
"meshes" :  
[  
  { ... }  
  { ... }  
  ...  
,
```

These indices are also used to define the *relationships* between the objects. The example above defines multiple meshes, and a node may refer to one of these meshes, using the mesh index, to indicate that the mesh should be attached to this node:

```
"nodes":  
[  
  { "mesh": 0, ... },  
  { "mesh": 5, ... },  
  ...  
]
```

The following image (adapted from the [glTF concepts section](#)) gives an overview of the top-level elements of the JSON part of a glTF asset:

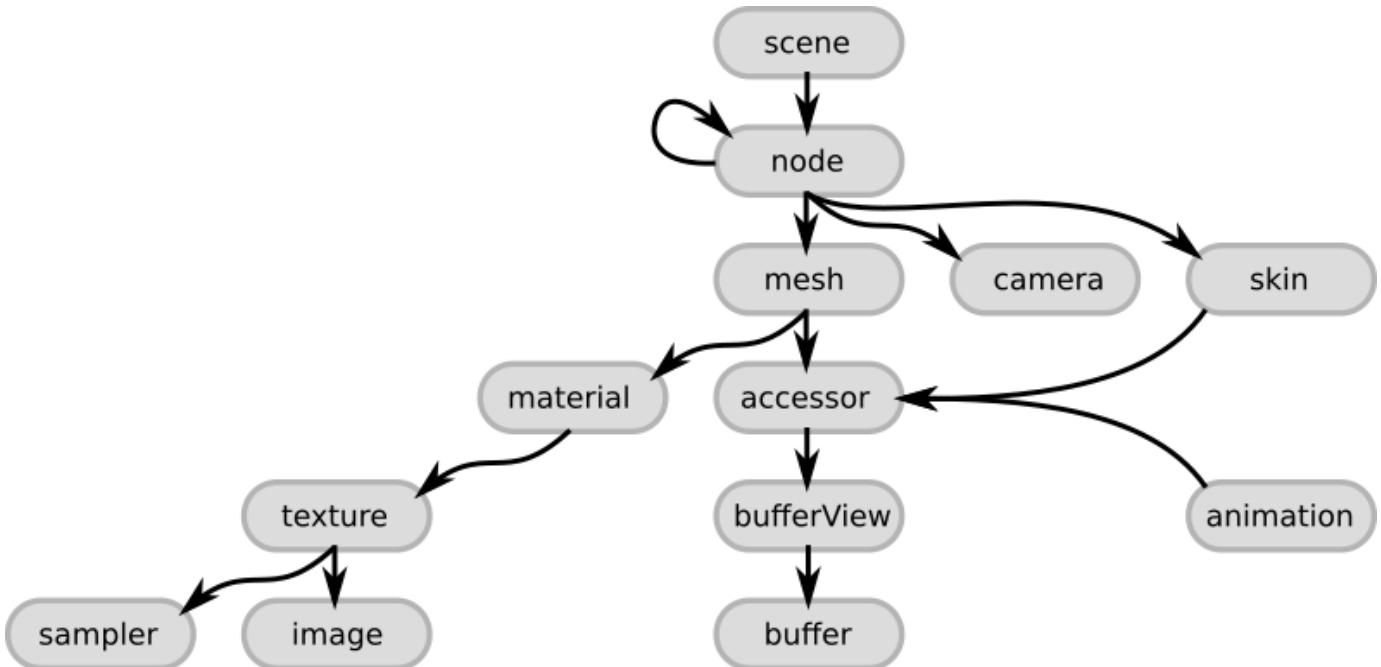


Image 2a: The glTF JSON structure

These elements are summarized here quickly, to give an overview, with links to the respective sections of the glTF specification. More detailed explanations of the relationships between these elements will be given in the following sections.

- The `scene` is the entry point for the description of the scene that is stored in the glTF. It refers to the `node`s that define the scene graph.
- The `node` is one node in the scene graph hierarchy. It can contain a transformation (e.g., rotation or translation), and it may refer to further (child) nodes. Additionally, it may refer to `mesh` or `camera` instances that are “attached” to the node, or to a `skin` that describes a mesh deformation.
- The `camera` defines the view configuration for rendering the scene.
- A `mesh` describes a geometric object that appears in the scene. It refers to `accessor` objects that are used for accessing the actual geometry data, and to `material`s that define the appearance of the object when it is rendered.
- The `skin` defines parameters that are required for vertex skinning, which allows the deformation of a mesh based on the pose of a virtual character. The values of these parameters are obtained from an `accessor`.
- An `animation` describes how transformations of certain nodes (e.g., rotation or translation) change over time.
- The `accessor` is used as an abstract source of arbitrary data. It is used by the `mesh`, `skin`, and `animation`, and provides the geometry data, the skinning parameters and the time-dependent animation values. It refers to a `bufferView`, which is a part of a `buffer` that contains the actual raw binary data.
- The `material` contains the parameters that define the appearance of an object. It usually refers to `texture` objects that will be applied to the rendered geometry.

- The `texture` is defined by a `sampler` and an `image`. The `sampler` defines how the texture `image` should be placed on the object.

References to external data

The binary data, like geometry and textures of the 3D objects, are usually not contained in the JSON file. Instead, they are stored in dedicated files, and the JSON part only contains links to these files. This allows the binary data to be stored in a form that is very compact and can efficiently be transferred over the web. Additionally, the data can be stored in a format that can be used directly in the renderer, without having to parse, decode, or preprocess the data.

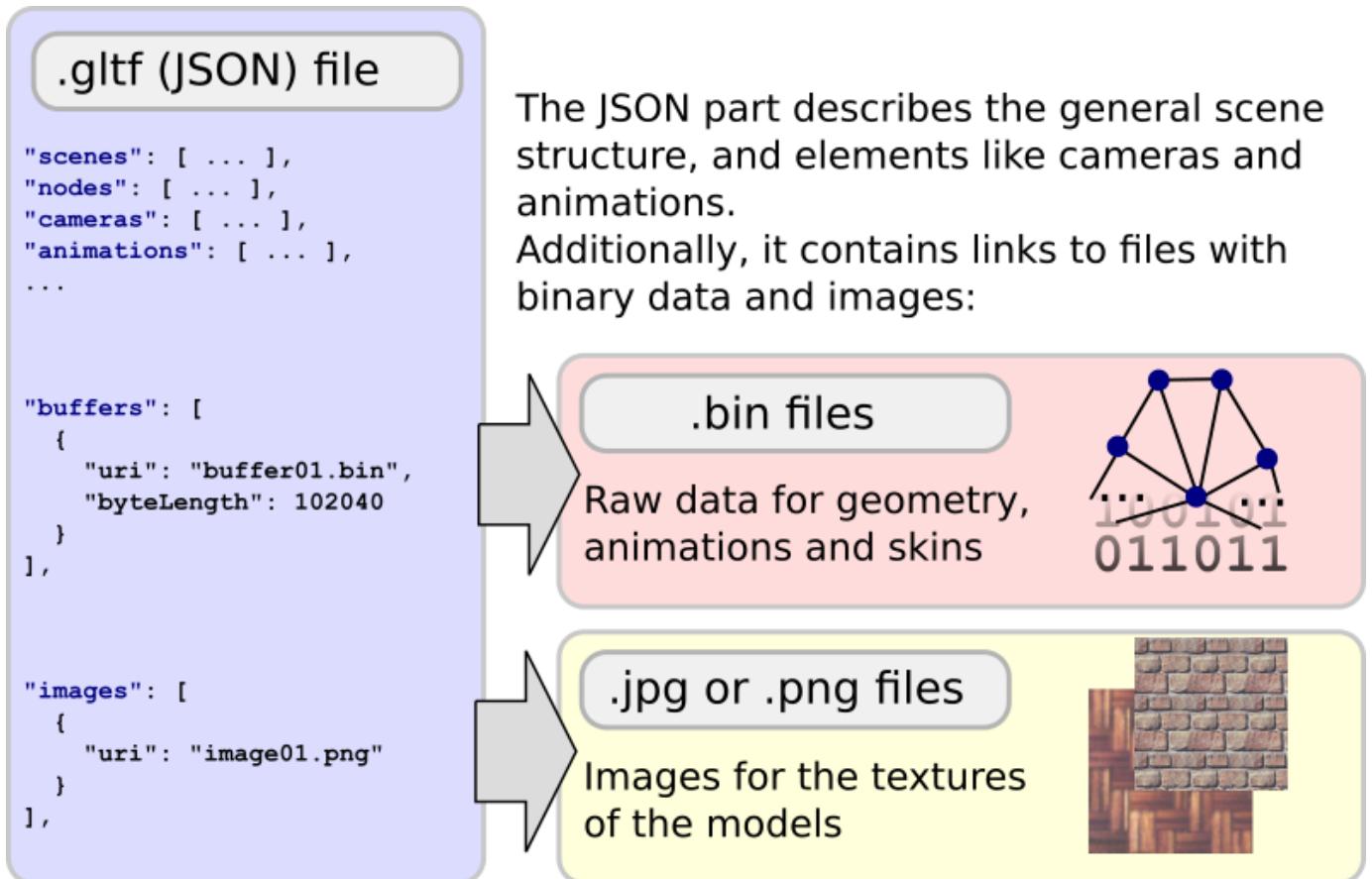


Image 2b: The glTF structure

As shown in the image above, there are two types of objects that may contain such links to external resources, namely `buffers` and `images`. These objects will later be explained in more detail.

Reading and managing external data

Reading and processing a glTF asset starts with parsing the JSON structure. After the structure has been parsed, the `buffer` and `image` objects are available in the top-level `buffers` and `images` arrays, respectively. Each of these objects may refer to blocks of binary data. For further processing, this data is read into memory. Usually, the data will be stored in an array so that they may be looked up using the same index that is used for referring to the `buffer` or `image` object that they belong to.

Binary data in `buffers`

A `buffer` contains a URI that points to a file containing the raw, binary buffer data:

```
"buffer01": {
  "byteLength": 12352,
  "type": "arraybuffer",
  "uri": "buffer01.bin"
}
```

This binary data is just a raw block of memory that is read from the URI of the `buffer`, with no inherent meaning or structure. The [Buffers, BufferViews, and Accessors](#) section will show how this raw data is extended with information about data types and the data layout. With this information, one part of the data may, for example, be interpreted as animation data, and another part may be interpreted as geometry data. Storing the data in a binary form allows it to be transferred over the web much more efficiently than in the JSON format, and the binary data can be passed directly to the renderer without having to decode or pre-process it.

Image data in `images`

An `image` may refer to an external image file that can be used as the texture of a rendered object:

```
"image01": {
  "uri": "image01.png"
}
```

The reference is given as a URI that usually points to a PNG or JPG file. These formats significantly reduce the size of the files so that they may efficiently be transferred over the web. In some cases, the `image` objects may not refer to an external file, but to data that is stored in a `buffer`. The details of this indirection will be explained in the [Textures, Images, and Samplers](#) section.

Binary data in data URIs

Usually, the URIs that are contained in the `buffer` and `image` objects will point to a file that contains the actual data. As an alternative, the data may be *embedded* into the JSON, in binary format, by using a [data URI](#).

Previous: [Introduction](#)

[Table of Contents](#)

Next: [A Minimal glTF File](#)

This site is open source. [Improve this page.](#)