

glTF-Tutorials

[Previous: Basic glTF Structure](#)[Table of Contents](#)[Next: Scenes and Nodes](#)

A Minimal glTF File

The following is a minimal but complete glTF asset, containing a single, indexed triangle. You can copy and paste it into a `gltf` file, and every glTF-based application should be able to load and render it. This section will explain the basic concepts of glTF based on this example.

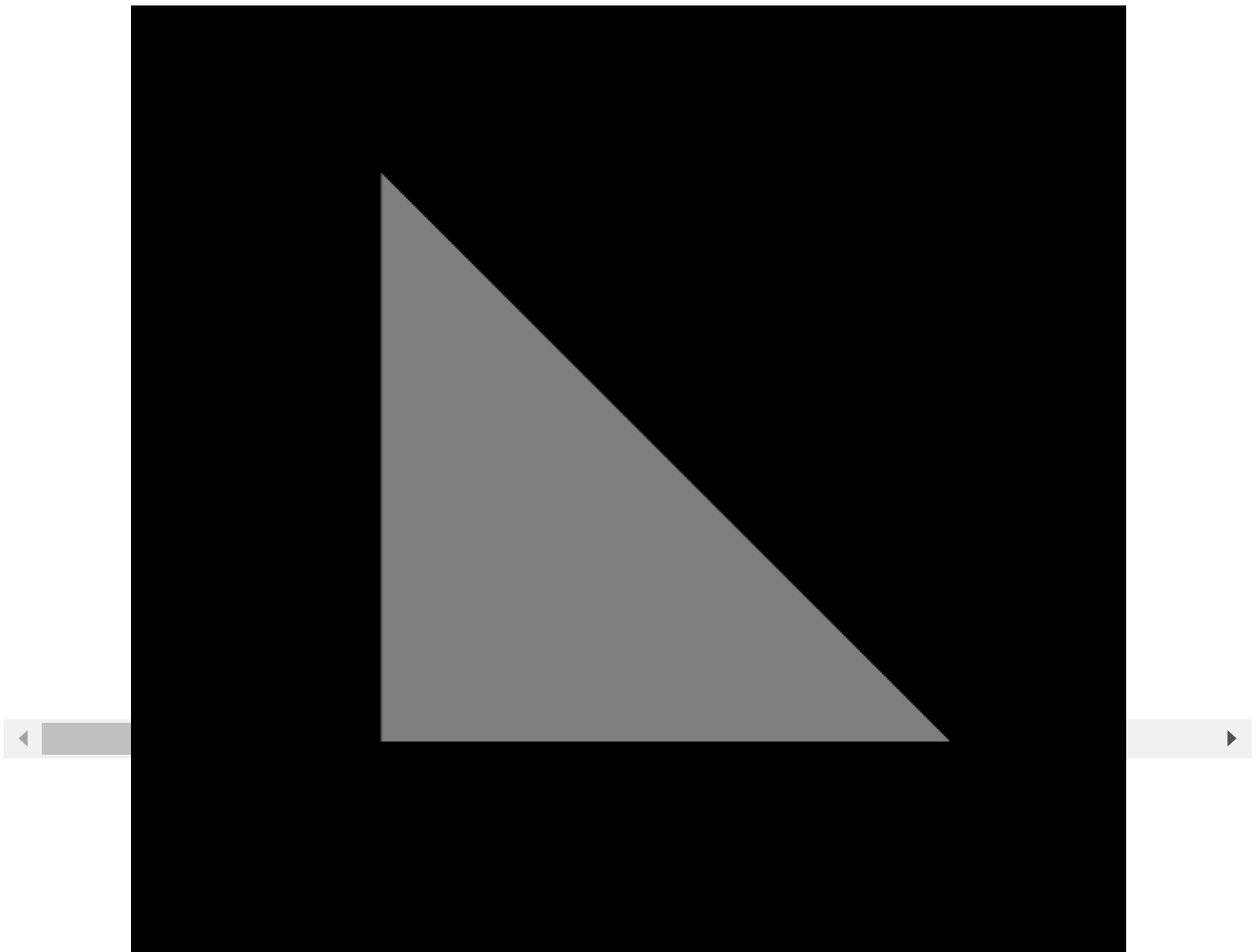
```
{
  "scene": 0,
  "scenes" : [
    {
      "nodes" : [ 0 ]
    }
  ],

  "nodes" : [
    {
      "mesh" : 0
    }
  ],

  "meshes" : [
    {
      "primitives" : [ {
        "attributes" : {
          "POSITION" : 1
        },
        "indices" : 0
      } ]
    }
  ],

  "buffers" : [
    {
      "uri" : "data:application/octet-stream;base64,AAABAAIAAAAAAAAAAAAAAAAAAAAAIA/AAAAA/"
      "byteLength" : 44
    }
  ],
  "bufferViews" : [
    {
      "buffer" : 0,
      "byteOffset" : 0,
      "byteLength" : 6,
    }
  ]
}
```

```
    "target" : 34963
  },
  {
    "buffer" : 0,
    "byteOffset" : 8,
    "byteLength" : 36,
    "target" : 34962
  }
],
"accessors" : [
  {
    "bufferView" : 0,
    "byteOffset" : 0,
    "componentType" : 5123,
    "count" : 3,
    "type" : "SCALAR",
    "max" : [ 2 ],
    "min" : [ 0 ]
  },
  {
    "bufferView" : 1,
    "byteOffset" : 0,
    "componentType" : 5126,
    "count" : 3,
    "type" : "VEC3",
    "max" : [ 1.0, 1.0, 0.0 ],
    "min" : [ 0.0, 0.0, 0.0 ]
  }
],
"asset" : {
  "version" : "2.0"
}
}
```



The scene and nodes structure

The example here consists of a single scene. The `scene` property indicates that this scene is the default scene that should be displayed when the asset is loaded. The scene refers to the only node in this example, which is the node with the index 0. This node, in turn, refers to the only mesh, which has the index 0:

https://github.com/khronos.org/glTF-Tutorials/glTFTutorial/glTFTutorial_003_MinimalGlTFFile.html

```
{
  "mesh" : 0
},
```

More details about scenes and nodes and their properties will be given in the [Scenes and Nodes](#) section.

The meshes

A `mesh` represents an actual geometric object that appears in the scene. The mesh itself usually does not have any properties, but only contains an array of `mesh.primitive` objects, which serve as building blocks for larger models. Each mesh primitive contains a description of the geometry data that the mesh consists of.

The example consists of a single mesh, and has a single `mesh.primitive` object. The mesh primitive has an array of `attributes`. These are the attributes of the vertices of the mesh geometry, and in this case, this is only the `POSITION` attribute, describing the positions of the vertices. The mesh primitive describes an *indexed* geometry, which is indicated by the `indices` property. By default, it is assumed to describe a set of triangles, so that three consecutive indices are the indices of the vertices of one triangle.

The actual geometry data of the mesh primitive is given by the `attributes` and the `indices`. These both refer to `accessor` objects, which will be explained below.

```
"meshes" : [
  {
    "primitives" : [ {
      "attributes" : {
        "POSITION" : 1
      },
      "indices" : 0
    } ]
  }
],
```

A more detailed description of meshes and mesh primitives can be found in the [meshes](#) section.

The buffer, bufferView, and accessor concepts

The `buffer`, `bufferView`, and `accessor` objects provide information about the geometry data that the mesh primitives consist of. They are introduced here quickly, based on the specific example. A more detailed description of these concepts will be given in the [Buffers, BufferViews, and Accessors](#) section.

Buffers

A `buffer` defines a block of raw, unstructured data with no inherent meaning. It contains an `uri`, which can either point to an external file that contains the data, or it can be a `data URI` that encodes the binary data directly in the JSON file.

In the example file, the second approach is used: there is a single buffer, containing 44 bytes, and the data of this buffer is encoded as a data URI:

```
"buffers" : [
  {
    "uri" : "data:application/octet-stream;base64,AAABAAIAAAAAAAAAAAAAAAAAAAAAIA/AAAAA/
    "byteLength" : 44
  }
],
```

This data contains the indices of the triangle, and the vertex positions of the triangle. But in order to actually use this data as the geometry data of a mesh primitive, additional information about the *structure* of this data is required. This information about the structure is encoded in the `bufferView` and `accessor` objects.

Buffer views

A `bufferView` describes a “chunk” or a “slice” of the whole, raw buffer data. In the given example, there are two buffer views. They both refer to the same buffer. The first buffer view refers to the part of the buffer that contains the data of the indices: it has a `byteOffset` of 0 referring to the whole buffer data, and a `byteLength` of 6. The second buffer view refers to the part of the buffer that contains the vertex positions. It starts at a `byteOffset` of 8, and has a `byteLength` of 36; that is, it extends to the end of the whole buffer.

```
"bufferViews" : [
  {
    "buffer" : 0,
    "byteOffset" : 0,
    "byteLength" : 6,
    "target" : 34963
  },
  {
    "buffer" : 0,
    "byteOffset" : 8,
    "byteLength" : 36,
    "target" : 34962
  }
],
```

Accessors

The second step of structuring the data is accomplished with `accessor` objects. They define how the data of a `bufferView` has to be interpreted by providing information about the data types and the layout.

In the example, there are two accessor objects.

The first accessor describes the indices of the geometry data. It refers to the `bufferView` with index 0, which is the part of the `buffer` that contains the raw data for the indices. Additionally, it specifies the `count` and `type` of the elements and their `componentType`. In this case, there are 3 scalar elements, and their component type is given by a constant that stands for the `unsigned short` type.

The second accessor describes the vertex positions. It contains a reference to the relevant part of the buffer data, via the `bufferView` with index 1, and its `count`, `type`, and `componentType` properties say that there are three elements of 3D vectors, each having `float` components.

```
"accessors" : [
  {
    "bufferView" : 0,
    "byteOffset" : 0,
    "componentType" : 5123,
    "count" : 3,
    "type" : "SCALAR",
    "max" : [ 2 ],
    "min" : [ 0 ]
  },
  {
    "bufferView" : 1,
    "byteOffset" : 0,
    "componentType" : 5126,
    "count" : 3,
    "type" : "VEC3",
    "max" : [ 1.0, 1.0, 0.0 ],
    "min" : [ 0.0, 0.0, 0.0 ]
  }
],
```

As described above, a `mesh.primitive` may now refer to these accessors, using their indices:

```
"meshes" : [
  {
    "primitives" : [ {
      "attributes" : {
        "POSITION" : 1
      },
      "indices" : 0
    }
  ]
},
```

```
    } ]  
  }  
],
```

When this `mesh.primitive` has to be rendered, the renderer can resolve the underlying buffer views and buffers and will send the required parts of the buffer to the renderer, together with the information about the data types and layout. A more detailed description of how the accessor data is obtained and processed by the renderer is given in the [Buffers, BufferViews, and Accessors](#) section.

The `asset` description

In glTF 1.0, this property is still optional, but in subsequent glTF versions, the JSON file is required to contain an `asset` property that contains the `version` number. The example here says that the asset complies to glTF version 2.0:

```
"asset" : {  
  "version" : "2.0"  
}
```

The `asset` property may contain additional metadata that is described in the [asset specification](#).

[Previous: Basic glTF Structure](#)[Table of Contents](#)[Next: Scenes and Nodes](#)

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