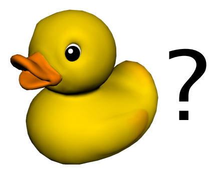
glTF - what the  ?

An overview of the basics of the GL Transmission Format

glTF was designed and specified by the Khronos Group, for the efficient transfer of 3D content over networks.

The core of glTF is a **JSON** file that describes the structure and composition of a scene containing 3D models. The top-level elements of this file are:

scenes, nodes, cameras, animations:
These describe the basic structure of the scene

meshes, textures, images, samplers, skins:
Describing the 3D objects that appear in the scene

buffers, bufferViews, accessors:
The layout of the data that the 3D objects consist of

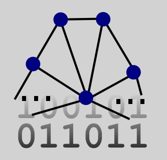
materials, techniques, programs, shaders:
Information about how the objects should be rendered

These elements are given as dictionaries. References between the objects are established by using IDs to look up $\longrightarrow$ the objects in the dictionaries.

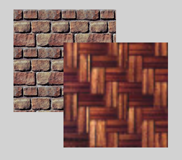
The JSON file refers to external files that are required for rendering the 3D content:

```
"buffers": {
    "buffer01": {
        "byteLength": 102040,
        "type": "arraybuffer",
        "uri": "buffer01.bin"
    }
},

"images": {
    "image01": {
        "name": "image01",
        "uri": "image01.png"
    }
},

"shaders": {
    "shader02_fragment": {
        "type": 35632,
        "uri": "shader02.glsl"
    }
}
```

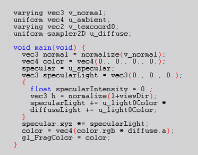
Geometry- or animation data, contained in binary files (.BIN)



Textures for the models, contained in image files (PNG, JPG...)



Shader programs for rendering the models, given as **GLSL** files

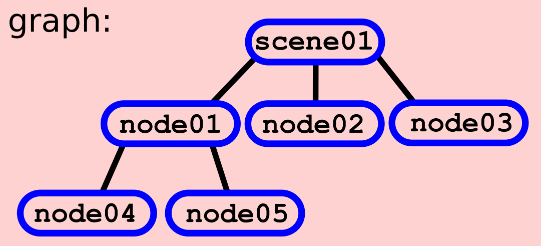


```
varying vec3 v_normal;
uniform vec4 u_ambient;
varying vec2 v_texcoord0;
uniform sampler2D u_diffuse;

void main(void) {
  vec3 normal = normalize(v_normal);
  vec4 color = vec4(0., 0., 0., 0.);
  specular = u_specular;
  vec3 specularLight = vec3(0., 0., 0.);
  {
    float specularIntensity = 0.;
    vec3 h = normalize(l+viewDir);
    specularLight += u_light0Color *
    diffuseLight += u_light0Color;
  }
  specular.xyz *= specularLight;
  color = vec4(color.rgb * diffuse.a);
  gl_FragColor = color;
}
```

The data is referred to via URIs, but can also be included directly in the JSON using data URIs

scenes, nodes, cameras, animations

The glTF JSON may contain **scenes** (with an optional default **scene**). Each scene can contain an array of IDs of nodes. Each of the **nodes** can contain an array of IDs of its children. This allows modeling a scene graph:



```
"scene": "scene01",
"scenes": {
  "scene01": {
    "nodes": [ "node01", "node02", "node03" ].
  }
},
"nodes": {
  "node01": {
    "children": [ "node04", "node05" ],
    ...
  },
  ...
  "node04": {   ...  },
  "node05": {   ...  },
  ...
},
```

Each node may contain a local transformation, given as a column-major **matrix** array, or a **translation**, **rotation** and **scale**, where the rotation is given as a quaternion. The global transform of a node is the product of all local transformations on the path from the scene root to the node

```
"node01": {
    "matrix": [
        1,0,0,0,
        0,1,0,0,
        0,0,1,0,
        5,6,7,1
    ],
    ...
},
"node02": {
    "translation": [ 0, 0, 0 ]
    "rotation": [ 0, 0, 0, 1 ],
    "scale": [ 1, 1, 1 ]
    "camera" : "camera01"
    "meshes" : [
      "mesh01", "mesh02" ],
    ...
},
```

```
"cameras": {
  "camera01": {
    "name": "camera01",
    "type": "perspective",
    "perspective": {
      "aspectRatio": 1.5,
      "yfov": 0.660593,
      "zfar": 100,
      "znear": 0.01
      ...
```

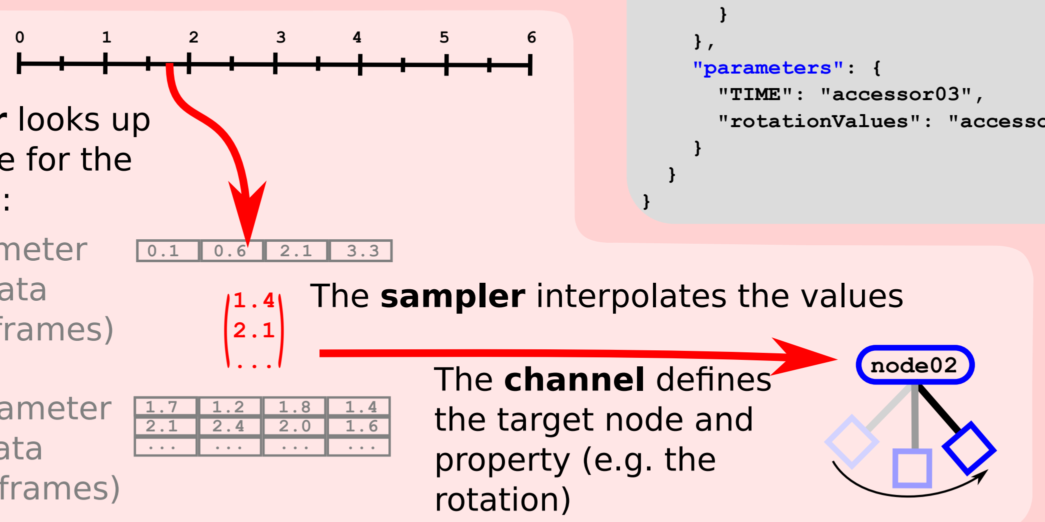
Each node may contain a camera ID, referring to one of the **cameras**. The global transform of the node is used as the camera matrix.

Nodes may also refer to **meshes**. These meshes are transformed with the global node transform.

The **animations** are defined by multiple elements: The **channels** describe the node ID and the "path", which is the animated property (e.g. the rotation). A channel refers to one of the **samplers**. The sampler defines how to interpolate two of the **parameters**. Each parameter refers to the animation data using one of the data **accessors**.

```
"animations": {
  "animation01": {
    "channels": [
      {
        "target": {
          "id": "node02",
          "path": "rotation"
        },
        "sampler": "sampler01"
      }
    ],
    "samplers": {
      "sampler01": {
        "input": "TIME",
        "interpolation": "LINEAR",
        "output": "rotationValues"
      }
    },
    "parameters": {
      "TIME": "accessor03",
      "rotationValues": "accessor04"
    }
  }
}
```

Global time:



The **sampler** looks up the key frame for the "global" time:

Input parameter accessor data (TIME key frames)

The **sampler** interpolates the values

Output parameter accessor data (value key frames)

The **channel** defines the target node and property (e.g. the rotation)

meshes, textures, images, samplers, skins (skins are not part of this overview)

```
"meshes": {
  "mesh01": {
    "name": "mesh01",
    "primitives": [
      {
        "mode": 4,
        "indices": "accessor01",
        "attributes": {
          "NORMAL": "accessor03",
          "POSITION": "accessor02",
          "TEXCOORD_0": "accessor04"
        },
        "material": "material01"
      }
    ]
  }
},
```

The **meshes** may contain multiple mesh **primitives**. These describe the data that is required for rendering the mesh with OpenGL: They contain the rendering **mode** (points, lines, triangles, ...), a reference to the **indices**, and to the **attributes** of the primitive. These references are given as IDs for the **accessors** for the data. The meshPrimitive also refers to the **material** that should be used for rendering, using the ID of one of the **materials**.

```
"textures": {
  "texture01": {
    "target": 3553,
    "type": 5121,
    "format": 6408,
    "internalFormat": 6408,
    "source": "image01",
    "sampler": "sampler01"
  }
},
```

```
"images": {
  "image01": {
    "name": "image01",
    "uri": "file01.png"
  }
},
```

The **images** refer to an image file using a **URI**

```
"samplers": {
  "sampler01": {
    "magFilter": 9729,
    "minFilter": 9987,
    "wrapS": 10497,
    "wrapT": 10497
  }
},
```

The **samplers** describe the wrapping and scaling of textures, using OpenGL constants that can be passed to **`glTexParameter`**.

The **textures** contain parameters describing the format (OpenGL constants that can be passed to **`glTexture2D`**). They refer to one of the **images**, which is the **source** of the texture. They have a **sampler** ID that refers to one of the **samplers**, which is used for mapping the texture.

materials, techniques, programs, shaders

```
""materials": {
  "material01": {
      "name": "material01",
      "technique": "technique01",
      "values": {
          "ambient": [ 0,0,0,1 ],
          "diffuse": "texture01"
      }
  }
},
```

Each of the **materials** refers to a **technique**. They can define the **values** that overwrite the default values of the technique parameters.

```
"techniques": {
  "technique01": {
    "program": "program01",
    "attributes": {
      "a_position": "position",
      "a_normal": "normal",
      "a_texcoord0": "texcoord0"
    },
    "uniforms": {
      "u_ambient": "ambient",
      "u_diffuse": "diffuse",
      "u_modelViewMatrix":
        "modelViewMatrix", ...
    },
    "parameters": {
      "position": {
        "type": 35665,
        "semantic": "POSITION"
      },
      "modelViewMatrix": {
        "type": 35676,
        "semantic": "MODELVIEW"
      },
      ...
      "ambient": {
          "type": 35666
      },
      "diffuse": {
        "type": 35678
      }
    }
  }
},
```

Each of the **techniques** refers to its program. The keys of the technique **attributes** dictionary are the attributes of the vertex shader. They are mapped to the technique **parameters**. There, their **type** is defined with a constant like **`GL_UNSIGNED_BYTE`**, or **`GL_FLOAT_MAT3`**. They may also receive a certain **semantic**. Attribute semantics may be **`POSITION`**, **`NORMAL`** and others. The semantic is used for looking up the mesh primitive attribute whose accessor provides the attribute data.

```
"programs": {
  "program01": {
    "vertexShader": "shader01_v",
    "fragmentShader": "shader02_f",
    "attributes": [
      "a_normal",
      "a_position",
      "a_texcoord0"
    ],
  }
},
```

The **programs** refer to their **vertexShader** and their **fragmentShader**, and list the **attributes** that appear in the vertex shader.

```
"shaders": {
  "shader01_v": {
    "type": 35633,
    "uri": "shader01_vertex.glsl"
  }
  "shader02_f": {
    "type": 35632,
    "uri": "shader02_fragment.glsl"
  },
},
```

Vertex Shader

```
attribute vec3 a_position;
attribute vec3 a_normal;
attribute vec2 a_texcoord0;
uniform mat4 u_modelViewMatrix;
...
varying vec3 v_position;
...
void main(void) {
 ...
    v_position = ...;
    gl_Position = ...;
}
```

Fragment Shader

```
uniform vec4 u_ambient;
uniform sampler2D u_diffuse;
...
varying vec3 v_position;
...
void main(void) {
   ...
  gl_FragColor = ...;
}
```

The **shaders** use **URIs** to refer to GLSL files that contain the shader code.

The keys of the technique **uniforms** dictionary are the uniforms of the vertex- and fragment shader. They are mapped to the technique **parameters** where they receive a type, like **`GL_FLOAT_MAT3`** or **`GL_SAMPLER_2D`**, and an optional semantic like **`MODELVIEW`** or **`PROJECTION`**. This semantic is usually interpeted in the context of the node that contains the mesh with the mesh primitive that should be rendered: The global transform of the node defines the current **`MODEL`** matrix. The **`VIEW`** matrix is determined by the transform of the currently active camera.

buffers, bufferViews, accessors

Each of the **buffers** refers to a binary data file, using a **URI**.

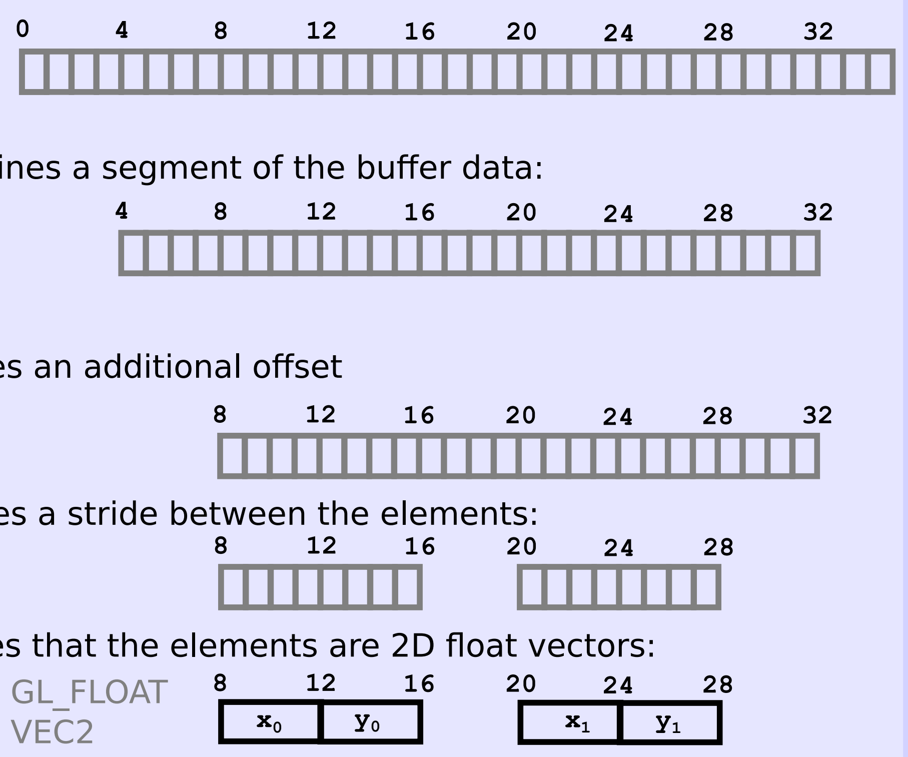
```
"buffers": {
  "buffer01": {
    "byteLength": 102040,
    "type": "arraybuffer",
    "uri": "buffer01.bin"
  }
},
```

Each of the **bufferViews** refers to one buffer, and defines a **byteOffset** and a **byteLength**. These describe the part of the buffer that belongs to the bufferView.

```
"bufferViews": {
  "bufferView01": {
    "buffer": "buffer01",
    "byteOffset": 12213,
    "byteLength": 25272,
    "target": 34963
  },
},
```

The **accessors** define how the data of a bufferView is interpreted. It summarizes the **count** of elements in the bufferView, and the type of the elements. E.g. it may define the elements to be 2D vectors of floating point numbers when the **type** is **`"VEC2"`**, and the **componentType** is **`GL_FLOAT`** (5126). The **byteOffset** and **byteStride** define where the data for the accessor starts inside the bufferView, and how many bytes are between the start of one element and the start of the next.

```
"accessors": {
  "accessor01": {
    "bufferView": "bufferView01",
    "count": 2399,
    "componentType": 5126,
    "type": "VEC2",
    "byteOffset": 0,
    "byteStride": 12
  },
}
```



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

buffer
byteLength = 35

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

bufferView
byteOffset = 4
byteLength = 28

The **accessor** defines an additional offset

accessor
byteOffset = 4

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

byteStride = 12

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

componentType = GL_FLOAT
componentType = VEC2

So this **accessor** may be referred to, for example, by a mesh primitive, to access the data that is used as 2D texture coordinates.

Summary: Rendering a glTF

The scene graph consisting of nodes is traversed, keeping track of the current node and the accumulated (global) transform. When a mesh with a mesh primitive is found, its material is looked up. The technique of the material determines the program and the shaders to be used for rendering.

The attribute inputs of the shader program are set based on the attributes of the technique: E.g. the attribute with the name **`"a_position"`** is looked up in the technique attributes dictionary, to find the technique parameter **`"position"`**, with the type **`GL_FLOAT_VEC3`** and the semantic **`"POSITION"`**. This is looked up in the mesh primitive attributes, to find the ID of an accessor that provides the input data for the attribute.

The uniform inputs of the shaders are set by examining the remaining technique parameters. They may have default values, or values that are overwritten by the corresponding value of the material. If the parameter has a semantic, for example, **`"MODELVIEW"`**, then the uniform parameter value is computed from the current context: The **`MODEL`** matrix is the global transform of the current node. The **`VIEW`** matrix is the inverse of the camera matrix, which is the global transform of the node that contains the currently active camera.

Further resources

This overview is not official and not complete (and maybe not even correct). Its goal is to give an overview of the basic ideas and concepts, and the relationships of the entities that appear in a glTF. The definite and official references for glTF are
- the Khronos glTF landing page: https://www.khronos.org/gltf
- the Khronos glTF GitHub repository: https://github.com/KhronosGroup/glTF

which contain the official specification and sample models.