glTF-Tutorials

Previous: Buffers, BufferViews, and Accessors | Table of Contents | Next: Animations

A Simple Animation

As shown in the Scenes and Nodes section, each node can have a local transform. This transform can be given either by the `matrix` property of the node or by using the `translation`, `rotation`, and `scale` (TRS) properties.

When the transform is given by the TRS properties, an `animation` can be used to describe how the `translation`, `rotation`, or `scale` of a node changes over time.

The following is the minimal glTF file that was shown previously, but extended with an animation. This section will explain the changes and extensions that have been made to add this animation.

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

  "nodes" : [
    {
      "mesh" : 0,
      "rotation" : [ 0.0, 0.0, 0.0, 1.0 ]
    }
  ],

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

  "animations": [
    {
      "samplers" : [
```

```
        {
          "input" : 2,
          "interpolation" : "LINEAR",
          "output" : 3
        }
      ],
      "channels" : [ {
        "sampler" : 0,
        "target" : {
          "node" : 0,
          "path" : "rotation"
        }
      } ]
    }
  ],

  "buffers" : [
    {
      "uri" : "data:application/octet-stream;base64,AAABAAIAAAAAAAAAAAAAAAAAAAAAIA/AAAAA
      "byteLength" : 44
    },
    {
      "uri" : "data:application/octet-stream;base64,AAAAAAAAgD4AAAA/AABAPwAAgD8AAAAAAAAAA
      "byteLength" : 100
    }
  ],
  "bufferViews" : [
    {
      "buffer" : 0,
      "byteOffset" : 0,
      "byteLength" : 6,
      "target" : 34963
    },
    {
      "buffer" : 0,
      "byteOffset" : 8,
      "byteLength" : 36,
      "target" : 34962
    },
    {
      "buffer" : 1,
      "byteOffset" : 0,
      "byteLength" : 100
    }
  ],
  "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 ]
    },
    {
      "bufferView" : 2,
      "byteOffset" : 0,
      "componentType" : 5126,
      "count" : 5,
      "type" : "SCALAR",
      "max" : [ 1.0 ],
      "min" : [ 0.0 ]
    },
    {
      "bufferView" : 2,
      "byteOffset" : 20,
      "componentType" : 5126,
      "count" : 5,
      "type" : "VEC4",
      "max" : [ 0.0, 0.0, 1.0, 1.0 ],
      "min" : [ 0.0, 0.0, 0.0, -0.707 ]
    }
  ],

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

}
```

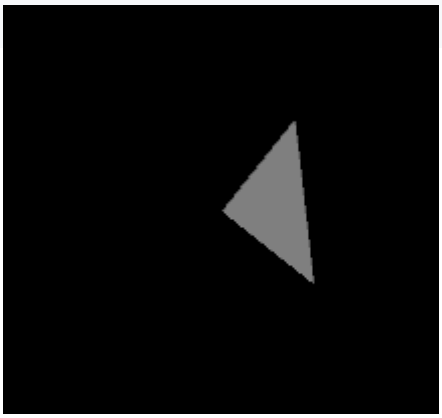

Image 6a: A single, animated triangle.

The `rotation` property of the `node`

The only node in the example now has a `rotation` property. This is an array containing the four floating point values of the quaternion that describes the rotation:

```
"nodes" : [
  {
    "mesh" : 0,
    "rotation" : [ 0.0, 0.0, 0.0, 1.0 ]
  }
],
```

The given value is the quaternion describing a "rotation about 0 degrees," so the triangle will be shown in its initial orientation.

The animation data

Three elements have been added to the top-level arrays of the glTF JSON to encode the animation data:

- A new `buffer` containing the raw animation data;
- A new `bufferView` that refers to the buffer;
- Two new `accessor` objects that add structural information to the animation data.

The `buffer` and the `bufferView` for the raw animation data

A new `buffer` has been added, which contains the raw animation data. This buffer also uses a data URI to encode the 100 bytes that the animation data consists of:

```
"buffers" : [
  ...
  {
    "uri" : "data:application/octet-stream;base64,AAAAAAAAgD4AAAA/AABAPwAAgD8AAAAAAAAAA
    "byteLength" : 100
  }
],

"bufferViews" : [
  ...
  {
    "buffer" : 1,
    "byteOffset" : 0,
    "byteLength" : 100
  }
],
```

There is also a new `bufferView`, which here simply refers to the new `buffer` with index 1, which contains the whole animation buffer data. Further structural information is added with the `accessor` objects described below.

Note that one could also have appended the animation data to the existing buffer that already contained the geometry data of the triangle. In this case, the new buffer view would have referred to the `buffer` with index 0, and used an appropriate `byteOffset` to refer to the part of the buffer that then contained the animation data.

In the example that is shown here, the animation data is added as a new buffer to keep the geometry data and the animation data separated.

The `accessor` objects for the animation data

Two new `accessor` objects have been added, which describe how to interpret the animation data. The first accessor describes the *times* of the animation key frames. There are five elements (as indicated by the `count` of 5), and each one is a scalar `float` value (which is 20 bytes in total). The second accessor says that after the first 20 bytes, there are five elements, each being a 4D vector with `float` components. These are the *rotations* that correspond to the five key frames of the animation, given as quaternions.

```
"accessors" : [
  ...
  {
    "bufferView" : 2,
    "byteOffset" : 0,
    "componentType" : 5126,
    "count" : 5,
    "type" : "SCALAR",
    "max" : [ 1.0 ],
    "min" : [ 0.0 ]
  },
  {
    "bufferView" : 2,
    "byteOffset" : 20,
    "componentType" : 5126,
    "count" : 5,
    "type" : "VEC4",
    "max" : [ 0.0, 0.0, 1.0, 1.0 ],
    "min" : [ 0.0, 0.0, 0.0, -0.707 ]
  }
],
```

The actual data that is provided by the *times* accessor and the *rotations* accessor, using the data from the buffer in the example, is shown in this table:

times accessor	***rotations*** accessor	Meaning
0.0	(0.0, 0.0, 0.0, 1.0)	At 0.0 seconds, the triangle has a rotation of 0 degrees
0.25	(0.0, 0.0, 0.707, 0.707)	At 0.25 seconds, it has a rotation of 90 degrees around the z-axis
0.5	(0.0, 0.0, 1.0, 0.0)	At 0.5 seconds, it has a rotation of 180 degrees around the z-axis
0.75	(0.0, 0.0, 0.707, -0.707)	At 0.75 seconds, it has a rotation of 270 (= -90) degrees around the z-axis
1.0	(0.0, 0.0, 0.0, 1.0)	At 1.0 seconds, it has a rotation of 360 (= 0) degrees around the z-axis

So this animation describes a rotation of 360 degrees around the z-axis that lasts 1 second.

The `animation`

Finally, this is the part where the actual animation is added. The top-level `animations` array contains a single `animation` object. It consists of two elements:

- The `samplers`, which describe the sources of animation data;
- The `channels`, which can be imagined as connecting a "source" of the animation data to a "target."

In the given example, there is one sampler. Each sampler defines an `input` and an `output` property. They both refer to accessor objects. Here, these are the *times* accessor (with index 2) and the *rotations* accessor (with index 3) that have been described above. Additionally, the sampler defines an `interpolation` type, which is `"LINEAR"` in this example.

There is also one `channel` in the example. This channel refers to the only sampler (with index 0) as the source of the animation data. The target of the animation is encoded in the `channel.target` object: it contains an `id` that refers to the node whose property should be animated. The actual node property is named in the `path`. So the channel target in the given example says that the `"rotation"` property of the node with index 0 should be animated.

```
"animations": [
  {
    "samplers" : [
      {
        "input" : 2,
        "interpolation" : "LINEAR",
        "output" : 3
```

```
        }
      ],
      "channels" : [ {
        "sampler" : 0,
        "target" : {
          "node" : 0,
          "path" : "rotation"
        }
      } ]
    }
  ],
```

Combining all this information, the given animation object says the following:

> During the animation, the animated values are obtained from the *rotations* accessor. They are interpolated linearly, based on the current simulation time and the key frame times that are provided by the *times* accessor. The interpolated values are then written into the `"rotation"` property of the node with index 0.

A more detailed description and actual examples for the interpolation and the computations that are involved here can be found in the Animations section.

Previous: Buffers, BufferViews, and Accessors	Table of Contents	Next: Animations

---