glTF-Tutorials

Previous: Simple Cameras | Table of Contents | Next: Simple Morph Target

Cameras

The example in the Simple Cameras section showed how to define perspective and orthographic cameras, and how they can be integrated into a scene by attaching them to nodes. This section will explain the differences between both types of cameras, and the handling of cameras in general.

Perspective and orthographic cameras

There are two kinds of cameras: *Perspective* cameras, where the viewing volume is a truncated pyramid (often referred to as “viewing frustum”), and *orthographic* cameras, where the viewing volume is a rectangular box. The main difference is that rendering with a *perspective* camera causes a proper perspective distortion, whereas rendering with an *orthographic* camera causes a preservation of lengths and angles.

The example in the Simple Cameras section contains one camera of each type, a perspective camera at index 0 and an orthographic camera at index 1:

```
"cameras" : [
  {
    "type": "perspective",
    "perspective": {
      "aspectRatio": 1.0,
      "yfov": 0.7,
      "zfar": 100,
      "znear": 0.01
    }
  },
  {
    "type": "orthographic",
    "orthographic": {
      "xmag": 1.0,
      "ymag": 1.0,
      "zfar": 100,
      "znear": 0.01
    }
  }
],
```

The `type` of the camera is given as a string, which can be `"perspective"` or `"orthographic"`. Depending on this type, the `camera` object contains a `camera.perspective` object or a `camera.orthographic` object. These objects contain additional parameters that define the actual viewing volume.

The `camera.perspective` object contains an `aspectRatio` property that defines the aspect ratio of the viewport. Additionally, it contains a property called `yfov`, which stands for *Field Of View in Y-direction*. It defines the “opening angle” of the camera and is given in radians.

The `camera.orthographic` object contains `xmag` and `ymag` properties. These define the magnification of the camera in x- and y-direction, and basically describe the width and height of the viewing volume.

Both camera types additionally contain `znear` and `zfar` properties, which are the coordinates of the near and far clipping plane. For perspective cameras, the `zfar` value is optional. When it is missing, a special “infinite projection matrix” will be used.

Explaining the details of cameras, viewing, and projections is beyond the scope of this tutorial. The important point is that most graphics APIs offer methods for defining the viewing configuration that are directly based on these parameters. In general, these parameters can be used to compute a *camera matrix*. The camera matrix can be inverted to obtain the *view matrix*, which will later be post-multiplied with the *model matrix* to obtain the *model-view matrix*, which is required by the renderer.

Camera orientation

A `camera` can be transformed to have a certain orientation and viewing direction in the scene. This is accomplished by attaching the camera to a `node`. Each `node` may contain the index of a `camera` that is attached to it. In the simple camera example, there are two nodes for the cameras. The first node refers to the perspective camera with index 0, and the second one refers to the orthographic camera with index 1:

```
"nodes" : {
  ...
  {
    "translation" : [ 0.5, 0.5, 3.0 ],
    "camera" : 0
  },
  {
    "translation" : [ 0.5, 0.5, 3.0 ],
    "camera" : 1
  }
},
```

As shown in the Scenes and Nodes section, these nodes may have properties that define the transform matrix of the node. The global transform of a node then defines the actual orientation of the camera in the scene. With the option to apply arbitrary animations to the nodes, it is even possible to define camera flights.

When the global transform of the camera node is the identity matrix, then the eye point of the camera is at the origin, and the viewing direction is along the negative z-axis. In the given example, the nodes both have a `translation` about `(0.5, 0.5, 3.0)`, which causes the camera to be transformed accordingly: it is translated about 0.5 in the x- and y- direction, to look at the center of the unit square, and about 3.0 along the z-axis, to move it a bit away from the object.

Camera instancing and management

There may be multiple cameras defined in the JSON part of a glTF. Each camera may be referred to by multiple nodes. Therefore, the cameras as they appear in the glTF asset are really “templates” for actual camera *instances*: Whenever a node refers to one camera, a new instance of this camera is created.

There is no “default” camera for a glTF asset. Instead, the client application has to keep track of the currently active camera. The client application may, for example, offer a dropdown-menu that allows one to select the active camera and thus to quickly switch between predefined view configurations. With a bit more implementation effort, the client application can also define its own camera and interaction patterns for the camera control (e.g., zooming with the mouse wheel). However, the logic for the navigation and interaction has to be implemented solely by the client application in this case. Image 15a shows the result of such an implementation, where the user may select either the active camera from the ones that are defined in the glTF asset, or an “external camera” that may be controlled with the mouse.

Previous: Simple Cameras | Table of Contents | Next: Simple Morph Target

This site is open source. Improve this page.