# The JSON File

SteamVR™ Tracking

## Introduction

Each tracked object in the SteamVR™ Tracking system contains a file that describes its sensor geometry and other important data about the device. This file is written using the JSON file format. Although many files use the JSON format, the file stored in the tracked object is so integral to the development and performance of the object that it receives the title "*The* JSON File." The JSON file begins its life with a minimum of sensor positions and orientations, but is augmented throughout the design and integration process to include IMU data, lens distortion data, and other metadata used by SteamVR™. Finally, calibration routines refine the original sensor geometry values to the exact sensor locations on a specific object and rewrite the JSON file, resulting in a unique JSON file for every tracked object.

This document describes all of the variables that may be stored in the JSON file, what they mean, and how to specify them. This document is referenced from other documents that describe various steps in the design process. To understand when, why, and how each variable is added to the JSON file, follow the process outlined in the **Object Design and Integration Overview**.

## JSON Format

JSON stands for JavaScript Object Notation, and refers to a lightweight data-interchange format. Data-interchange formats are means of formatting data to share between different computing platforms and, sometimes, human readers. A complete description of the JSON format is available at www.json.org. Because the JSON file represents a JSON object, it opens with a brace { and closes with a brace }. The different members stored in the JSON file are stored as pairs, which are delimited using commas. Each pair is identified by a string that is separated from its value by a colon, "name" : value. Valid members of the tracked object JSON file are described below, and a complete JSON file is presented at the end of the document as an example.

> **Tip:** Free source code editors like Notepad++ understand the JSON format and provide convenience features like syntax highlighting and code folding.

## JSON Members

### "manufacturer"

A string value representing the manufacturer's company name.

Example:

```
"manufacturer" : "Valve"
```

### "model_number"

A string value representing the model number of the object, assigned by the manufacturer.

Example:

```
"model_number" : "REF-HMD"
```

## “device_class”

A string value that may be set to “hmd” or “controller”

Setting the device_class to “controller” tells SteamVR™ to render the object in VR. Setting the value to “hmd” tells SteamVR™ to associate the tracked object with the binocular display and use the pose of the object as the point of view into virtual reality.

Example:

```
"device_class" : "hmd"
```

## “device_vid”

The USB vendor identification number. Each manufacturer should request a VID from USB.org. For prototyping purposes, use the Valve VID shown below.

Example:

```
"device_vid" : 10462
```

## “device_pid”

The USB product identification number. Each manufacturer should create a PID per model. For prototyping purposes, use the PID shown below.

Example:

```
"device_pid" : 8960
```

## “device_serial_number”

A string representing the unique serial number of the device.

This value is only used as a reference to match JSON files to actual objects. Each object creates its own serial number from the serial number in the object’s processor. Writing that serial number into the JSON file allows developers to associate JSON files with actual objects. Maintaining that association is particularly important after optical and IMU calibration. The serial number generated by the object is displayed using lighthouse_console.

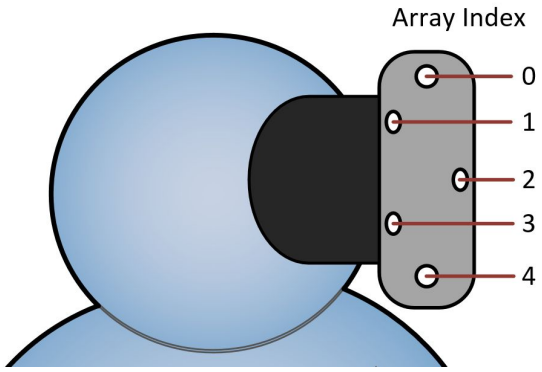
Example:

```
"device_serial_number" : "LHR-F8DE9EBE"
```

## “lighthouse_config”

An object containing three member arrays. Each array index corresponds to a sensor on the tracked object. Specifying the values in modelPoints, modelNormals, and channelMap define the physical placement of the sensor, its orientation, and its electrical connection (port number). When SteamVR™ receives tracking data from a sensor on a particular port, it can use the data in lighthouse_config to associate that signal with a specific physical sensor located on the tracked object. SteamVR™ also reads the sensor placement described in lighthouse_config to establish the exact sensor geometry for the device. SteamVR™ solves incoming tracking data against the known sensor geometry to resolve the current pose of the object.

The channelMap, modelNormals, and modelPoints arrays have an element for each sensor on the object. If we placed only five sensors on the HMD object pictured below, we would have five entries in each array. For our example, the indices of the array are assigned as shown.



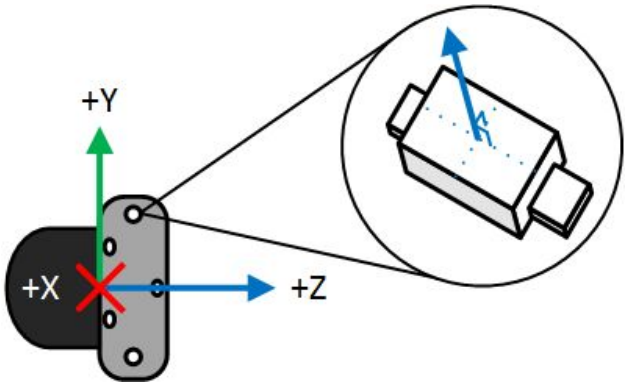

Example: (Replace ellipsis with arrays described in each member below.)

```
"lighthouse_config" : {
    "channelMap" : [...],
    "modelNormals" : [...],
    "modelPoints" : [...]
 }
```

### “modelNormals”

An array of [x, y, z] unit vectors that specify the direction the optical sensor faces in the object’s coordinate system.

Each photodiode is placed on the surface of the object facing outward. Imagine the sensor pictured below faces outward at a 45° angle between the -X and +Y axis.




The unit vector normal to the face of the photodiode points in the direction 135° from the +X axis and 45° from the +Y axis. As a result the unit vector describing that orientation would be [ -0.7071, 0.7071, 0.0 ]. If we have placed the other sensors on 45° facets, we would expect the following normals.

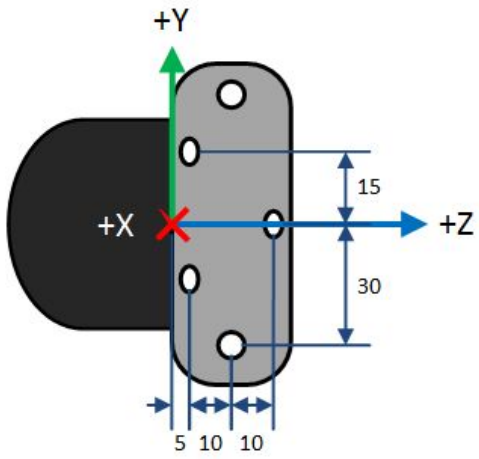
Example:

```
"modelNormals" : [
         [ -0.7071, 0.7071, 0.0 ],
         [ -0.7071, 0.0, -0.7071 ],
         [ -0.7071, 0.0, 0.7071 ],
         [ -0.7071, 0.0, -0.7071 ],
         [ -0.7071, -0.7071, 0.0 ]
      ]
```

### “modelPoints”

An array of [x, y, z] coordinates that specify the location of the center of the optical sensor’s photosensitive area in the object’s coordinate system. Coordinate values are expressed in meters.

If we assigned the following dimensions to the sensor locations on the side of the HMD, we would need to represent those dimensions in the modelPoints array as shown below.



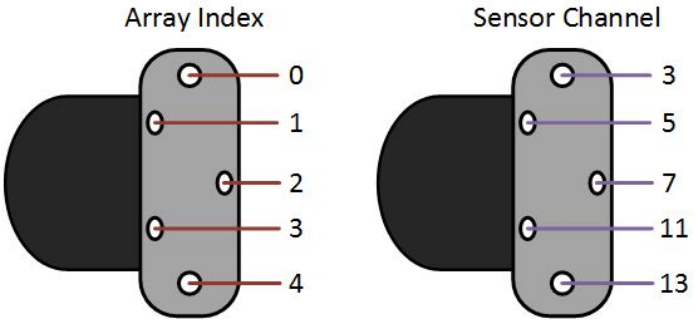

Example:

```
"modelPoints" : [
        [ -0.055, 0.030, 0.015 ],
         [ -0.050, 0.015, 0.005 ],
         [ -0.050, 0.0, 0.025 ],
         [ -0.050, -0.015, 0.005 ],
         [ -0.055, -0.030, 0.015 ]
      ]
```

### “channelMap”

An array of port numbers. There is an element in the array for each sensor on the object. The value in the array element corresponds to the electrical channel connected to the sensor. When SteamVR™ receives tracking data from a channel, it can use this array to map that data to a sensor location specified in modelPoints.

If the five sensors shown on the side of the HMD in this example were connected to the electrical channels as indicated below, we would need the channelMap array [3, 5, 7, 11, 13]. The sensor channel number is determined by the sensor’s electrical connection to the object’s FPGA.




**Note:** The channel numbers are zero based, and cover the range 0 - 31. Altium's multichannel schematic feature forces numbering to start at 1. Be careful when copying channel numbers from the net names in Altium. Net SENSOR_X1 is most likely connected to FPGA channel number 0. Subtract 1 from the reference designator to get the correct channel number.

Example:

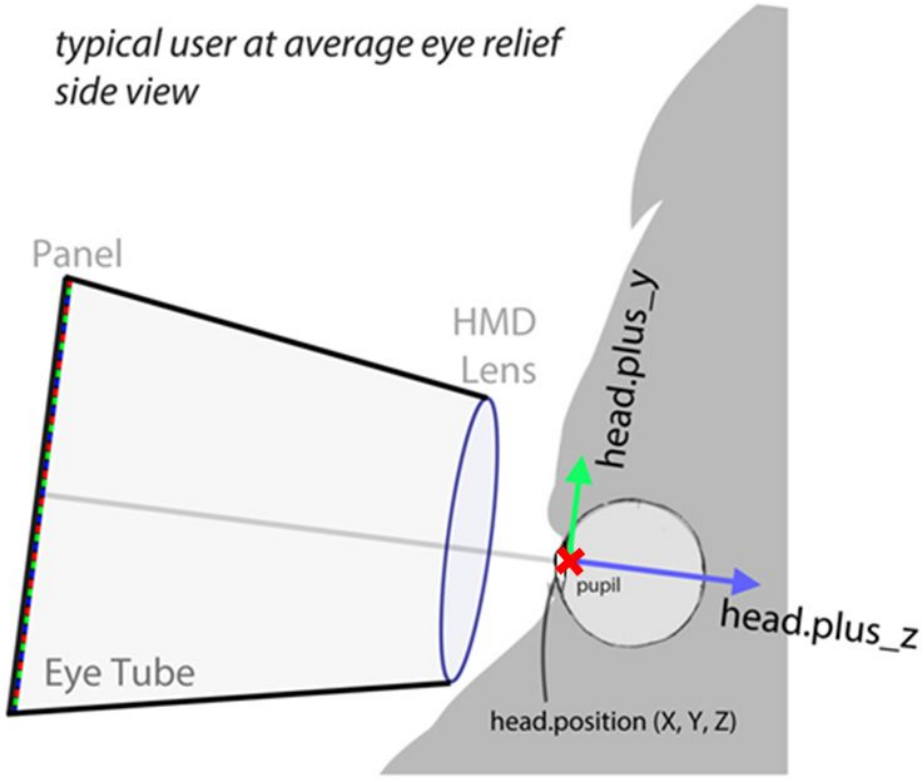
```
"channelMap" : [3, 5, 7, 11, 13]
```

## "head"

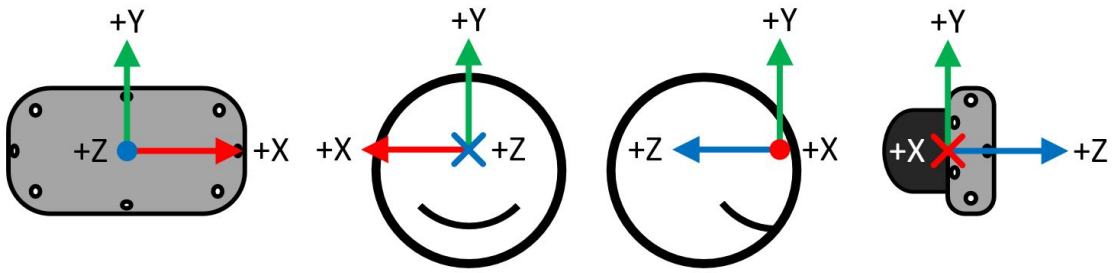
An object with members that orient the tracked object's coordinate system to the real world. The head member has two different meanings, depending on the value of device_class.

### HMD

When the object is acting as an HMD, the head variable orients SteamVR™'s coordinate system to the tracked object. The origin of SteamVR™ is a point between an average user's pupils, oriented with +Y pointing up, +X pointing to the user's right, and -Z pointing out along the line of sight. If the model has a coordinate system that does not match this, then changing the head member is required.



For example, consider the HMD object below, with its coordinate system.




The “plus_x” value answers the question, “The head’s +X axis is pointing in which direction within the HMD’s CAD coordinate system?” The head’s +X axis points in the direction of the HMD’s -X axis. Therefore, the value for “plus_x” is [-1, 0, 0].

Likewise, the +Z axis of the head’s coordinate system points in the direction of the -Z axis of the HMD’s coordinate system. This requires a “plus_z” value of [0, 0, -1] to match the orientations.

The head’s origin is centered between the pupils, but the HMD’s origin is between the HMD’s lenses. When worn, the origin of the head is behind the origin of the HMD by 20 mm along the -Z axis. Matching the origins requires a value of [0.0, 0.0, -0.020] for the “position” coordinates.

Example:

```
"head" : {
   "plus_x" : [ -1, 0, 0 ],
   "plus_z" : [ 0, 0, -1 ],
   "position" : [ 0.0, 0.0, -0.020 ]
}
```

### Controller

When the object is acting as a controller, the head variable orients the render model shown in SteamVR™ to the tracked object. The render model is created in the SteamVR™ coordinate system as described in **The Render Model**. One method for determining the head variable is to export the render model as an STL file, import that STL file into the 3D CAD space of the object. Then, measure the plus_x and plus_z normals and origin of the render model from the origin of the object.

Example:

```
"head" : {
   "plus_x": [1, 0, 0],
   "plus_z": [0, 0.05233595624, 0.99862953475],
   "position": [0, 0.015, -0.040]
}
```

### “plus_x”

HMD: A unit vector [x, y, z] representing the direction of the head’s +X axis in the object’s coordinate system.

Controller: A unit vector [x, y, z] that aligns the render model in SteamVR™ to the object’s coordinate system.

### “plus_z”

HMD: A unit vector [x, y, z] representing the direction  of the head’s +Z axis in the object’s coordinate system.

Controller: A unit vector [x, y, z] that aligns the render model in SteamVR™ to the object’s coordinate system.

### “position”

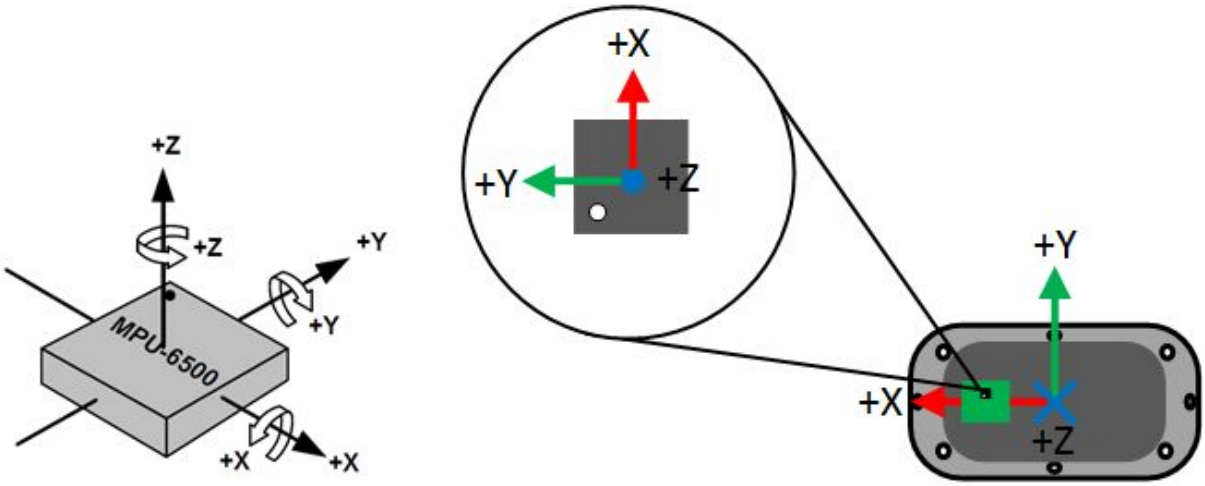
HMD: A coordinate [x, y, z] located at the midpoint between the average user’s physical pupil locations. This is typically determined by extending a vector out from the lens surface along the primary optical axis, until it crosses the plane of the average user’s pupil depth.

Controller: A coordinate [x, y, z]  that positions the render model in SteamVR™ in the object’s coordinate system.

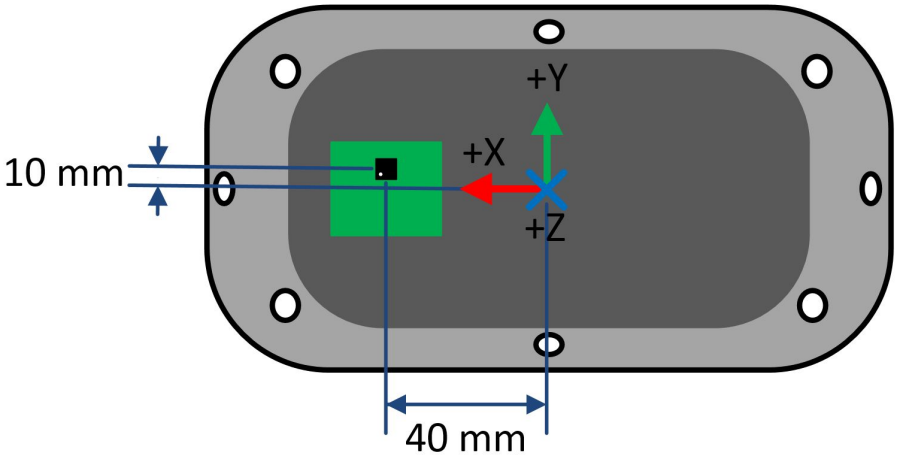
| |
|---|
| **Note:** A “plus_y” member is not required, because a right-handed coordinate system is assumed. |

### “imu”

An object containing calibration and orientation data about the IMU. The plus_x, plus_z, and position members map the IMU coordinate system into the object’s coordinate system. The acc and gyro members hold accelerometer and gyroscope calibration data. For an example, consider the following IMU, placed on a PCB in an HMD object.




The datasheet specifies the coordinate system of the IMU. However, that coordinate system is not aligned with the coordinate system of the HMD. In fact, all three axes axes need adjustment. The IMU’s +X axis points in the HMD’s +Y direction, requiring a plus_x value of [0, 1, 0]. The IMU’s +Z axis points in the HMD’s -Z direction, requiring a plus_z value of [0, 0, -1]. The IMU’s Y axis is also different from the HMD, but specifying the X and Z axes constrains the Y axis.




Not only are the axes of the IMU different, but the IMU is not located at the origin of the HMD’s coordinate system. The IMU is located in the X/Y plane, meaning that the Z offset is 0 mm, but the X offset is -40 mm and the Y offset is +10 mm. Therefore, the required position value is [-0.040, 0.010, 0.0].

The accelerometer and gyroscope have offsets inherent in the device. Accelerometer offsets, for example, may start as low as 0.025g. However, they increase during the assembly process, and throughout the life of the part, to as high as 0.150g or more. Initial gyroscope offsets are typically 3°/s - 5°/s. The accelerometer and gyroscope offsets change over temperature. Accounting for that, SteamVR™ has internal calibration mechanisms that constantly trim out the error during use. To ensure that SteamVR™ can converge on the actual offsets as quickly as possible, the IMU member of the JSON file holds initial calibration data for offsets present at the time of manufacturing. Determining the correct values for these fields is easily accomplished

using the software utility described in **IMU Calibration**. The IMU calibration tool outputs a snippet of JSON to copy and paste into the IMU member of the JSON file.

IMU Calibrator Output:

```
Calibrating to gravity sphere, radius 9.8066
0.05265 accelerometer fit error (6 sample vectors x 8 subsamples per vector)

"acc_scale" : [ 0.998, 0.9982, 0.9912 ],
"acc_bias" : [ 0.04646, -0.04264, -0.2414 ],
"gyro_scale" : [ 1.0, 1.0, 1.0 ],
"gyro_bias" : [ 0.06343, 0.01029, -0.02168 ],
```

Example:

```
"imu" : {
   "acc_scale" : [ 0.998, 0.9982, 0.9912 ],
   "acc_bias" : [ 0.04646, -0.04264, -0.2414 ],
   "gyro_scale" : [ 1.0, 1.0, 1.0 ],
   "gyro_bias" : [ 0.06343, 0.01029, -0.02168 ],
   "plus_x" : [ 0, 1, 0 ],
   "plus_z" : [ 0, 0, -1 ],
   "position" : [ -0.040, 0.010, 0.0 ]
   }
```

#### “plus_x”

A unit vector [x, y, z] representing the direction of the IMU’s +X axis in the object’s coordinate system.

#### “plus_z”

A unit vector [x, y, z] representing the direction of the IMU’s +Z axis in the object’s coordinate system.

#### “position”

A coordinate [x, y, z] specifying the center of the IMU’s package in the object’s coordinate system.

> **Note:** A “plus_y” member is not required, because a right-handed coordinate system is assumed.

### “render_model”

Holds a string value that specifies the name of the subfolder within the SteamVR™ “rendermodels” folder that holds the default rendermodel for the object.

Example:

```
"render_model" : "ref_controller"
```

### “display_edid”

Description

Example:

```
"display_edid" : [ "", "" ]
```

## “direct_mode_edid_vid”

The integer display EDID vendor ID. This number must be whitelisted by NVIDIA to enable your display to operate in direct mode. After that, this value tells SteamVR™ which display to use when displaying VR to the HMD. Note that any text prefixes are omitted.

Example:

```
"direct_mode_edid_vid" : xxxxx
```

## “direct_mode_edid_pid”

The integer display EDID product ID. This number must be whitelisted by NVIDIA to enable your display to operate in direct mode. After that, this value tells SteamVR™ which display to use when displaying VR to the HMD. Note that any text prefixes are omitted.

Example:

```
"direct_mode_edid_vid" : xxxxx
```

## “device”

Description

Example:

```
  "device" : {
      "eye_target_height_in_pixels" : 1080,
      "eye_target_width_in_pixels" : 960,
      "first_eye" : "eEYE_LEFT",
      "last_eye" : "eEYE_RIGHT",
      "num_windows" : 1,
      "persistence" : 0.01666999980807304,
      "physical_aspect_x_over_y" : 0.8000000119209290
   }
```

### “eye_target_height_in_pixels”

This value indicates the vertical resolution of each eye’s display in the HMD.  When operating out of direct mode, SteamVR uses this value along with eye_target_width_in_pixels to determine which display to associate with the tracked HMD device.

### “eye_target_width_in_pixels”

This value indicates the horizontal resolution of each eye’s display in the HMD.  Since HMDs split a single display output feed into two physical displays, this value should be one half of the width of the display as it appears in your operating system’s display settings.  In this example, the HMD’s resolution is 1920x1080 split between both eyes. When operating out of direct mode, SteamVR uses this value along with eye_target_width_in_pixels to determine which display to associate with the tracked HMD device.

### “first_eye”

Description

### “last_eye”

Description

### “num_windows”

Description

### “persistence”

Description

### “physical_aspect_x_over_y”

Description

## “lens_separation”

Description

Example:

```
"lens_separation" : 0.06230000033974648
```

## “tracking_to_eye_transform”

The lens calibration information is stored in the HMD JSON file in the block “tracking_to_eye_transform”. This block is a two-element array where the two elements are the descriptions for the left and right eyes respectively. For each eye’s description there are several sub-blocks, described below.
We assume here that the lens is circularly symmetric and is kept in a fixed position parallel to the display panel. We will assume a pixel coordinate system with $(0,0)$ in the top left corner of the display, the +X axis extending to the right, and the +Y axis extending down. Let $(w,h)$ be the horizontal and vertical resolution of the display. Note that this coordinate system is independent for each eye, not a single coordinate system extending across both eyes.

Example: (Replace ellipsis with values described in each member below.)

```
  "tracking_to_eye_transform" : [
      {
         "distortion" : {...},
         "distortion_blue" : {...},
         "distortion_red" : {...},
         "extrinsics" : [...],
         "grow_for_undistort" : 0.0,
         "intrinsics" : [...],
         "undistort_r2_cutoff" : 1.50
      },
      {
         "distortion" : {...},
         "distortion_blue" : {...},
         "distortion_red" : {...},
         "extrinsics" : [...],
         "grow_for_undistort" : 0.0,
         "intrinsics" : [...],
         "undistort_r2_cutoff" : 1.50
      }
   ],
```

### “intrinsics”

The “intrinsics” block is a 3x3 matrix describing the linear projection that the lens implements after distortion is corrected. Five values are populated based on the focal length and the center of projection. These values can be calculated as follows.

Let $f$ be the focal length of the HMD lens, measured in pixels. Let $(c_x, c_y)$ be the location on the panel (in pixel coordinates) that the HMD lens is centered over. The “intrinsics” block 3x3 matrix is then:

$$M = \begin{bmatrix} \frac{2f}{w} & 0 & -\frac{2c_x}{w} + 1 \\ 0 & \frac{2f}{\mathrm{h}} & \frac{2c_y}{\mathrm{h}} - 1 \\ 0 & 0 & -1 \end{bmatrix}$$

Example:

```
"intrinsics" : [
            [ 1.250, 0.0, 0.0 ],
            [ 0.0, 1.0, 0.0 ],
            [ 0.0, 0.0, -1.0 ]
         ]
```

### “distortion”, “distortion_blue”, “distortion_red”

The “distortion” block is information describing any non-linear distortion in the HMD lens. This includes the lens centering information and a polynomial describing the lens distortion. For each eye, there are actually three distortion blocks, “distortion”, “distortion_red”, and “distortion_blue”, which provide the distortion information for each of the green, red, and blue color channels respectively.

Let $(c_x, c_y)$ be the location on the panel (in pixel coordinates) that the HMD lens is centered over, as mentioned above. The "center_x" and "center_y" values in the distortion block can be calculated as:

$$\text{“center\_x”} = \frac{c_x - \left(\frac{w}{2}\right)}{\left(\frac{w}{2}\right)}$$

$$\text{“center\_y”} = \frac{c_y - \left(\frac{\mathrm{h}}{2}\right)}{\left(\frac{w}{2}\right)}$$

Note that the denominator in both cases is $(w/2)$.

The “type” field specifies the mathematical form of the distortion function, and the “coeffs” field specifies coefficients for use in that function. Let $P$ be a pixel location on the HMD display panel, and let $R$ be the distance in pixels between $P$ and the lens center location $P$. Let $f$ be the focal length of the HMD lens in pixels, as mentioned above. Let be the angle at which pixel $P$ appears through the lens. For a lens with no distortion, the relationship between these terms would be:

$$R = f * tan(\theta)$$

The most common form of distortion function is a function of $R$ (so, radially symmetric) that is applied as a multiplicative factor. In order to keep the values in a limited range the input to the function is also normalized, currently by dividing by $(w/2)$. For a distortion function $D$ this gives:

$$R * D\left(\frac{R}{\left(\frac{w}{2}\right)}\right) = f * tan(\theta)$$

We support several different distortion function types and it is fairly simple to add new types. Two common examples are: cubic polynomials in the (normalized) radius squared, and rational cubic polynomials in the (normalized) radius squared:

$$D(x) = 1 + C_0x^2 + C_1x^4 + C_2x^6$$

$$D(x) = \frac{1}{1 + C_0x^2 + C_1x^4 + C_2x^6}$$

For the first example the “type” field is DISTORT_POLY3, and for the second it is DISTORT_DPOLY3. In both cases the first three elements of the “coeffs” field are populated with $C_0$,$C_1$,$C_2$ and the remaining five elements are set to 0.0 (the length of the “coeffs” array is currently fixed at eight elements).

Example:

```
"distortion" : {
            "center_x" : 0.0,
            "center_y" : 0.0,
            "coeffs" : [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ],
            "type" : "DISTORT_DPOLY3"
         }
```

### “extrinsics”

The “extrinsics” block is a 3x4 matrix describing the rotation and translation of the lens+panel system relative to the user. For most normal usage this should simply express the relationship between the lens centers due to a default stereo separation (IPD). If we let $S$ be the default stereo separation in meters, then this matrix is:

$$\begin{bmatrix} 1 & 0 & 0 & \pm\frac{S}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

With the upper-right value being positive for the left eye and negative for the right eye.

Example:

```
"extrinsics" : [
```

```
            [ 1.0, 0.0, 0.0, -0.03115000016987324 ],
            [ 0.0, 1.0, 0.0, 0.0 ],
            [ 0.0, 0.0, 1.0, 0.0 ]
         ]
```

### “grow_for_undistort”

The “grow_for_undistort” is not a critical parameter and can (for now) be set to 0.6.

Example:

```
"grow_for_undistort" : 0.6
```

### “undistort_r2_cutoff”

The “undistort_r2_cutoff” field is not a critical parameter and can (for now) be set to 1.5.

Example:

```
"undistort_r2_cutoff" : 1.50
```

## “type”

A string value “Lighthouse_HMD”

This value is always “Lighthouse_HMD,” even when the device_class is set to controller.

Example:

```
"type" : "Lighthouse_HMD"
```

## Example JSON File


```
{
   "device" : {
      "eye_target_height_in_pixels" : 1080,
      "eye_target_width_in_pixels" : 960,
      "first_eye" : "eEYE_LEFT",
      "last_eye" : "eEYE_RIGHT",
      "num_windows" : 1,
      "persistence" : 0.01666999980807304,
      "physical_aspect_x_over_y" : 0.8000000119209290
   },
   "device_class" : "controller",
   "device_pid" : 8192,
   "device_serial_number" : "LHR-F8DE9EBE",
   "device_vid" : 10462,
   "display_edid" : [ "", "" ],
   "lens_separation" : 0.06230000033974648,
   "lighthouse_config" : {
      "channelMap" : [ 17, 15, 13, 21, 19 ],
      "modelNormals" : [
         [ 0, 0, -1 ],
         [ -0.13309992849826813, 0.11159992963075638, -0.98479938507080078 ],
         [ 0.11159992963075638, 0.13309992849826813, -0.98479938507080078 ],
         [ 0.13309992849826813, -0.11159992963075638, -0.98479938507080078 ],
         [ -0.11159992963075638, -0.13309992849826813, -0.98479938507080078 ]
      ],
      "modelPoints" : [
         [ -0.0015368221793323755, 0.017447538673877716, -0.0040629836730659008 ],
         [ -0.046612702310085297, 0.039085414260625839, 0.011825915426015854 ],
         [ 0.039518974721431732, 0.046799946576356888, 0.011834526434540749 ],
         [ 0.046315468847751617, -0.038777932524681091, 0.011671472340822222 ],
         [ -0.03922756016254425, -0.046778313815593719, 0.011606470681726933 ]
      ]
   },
   "manufacturer" : "",
   "model_number" : "",
   "render_model" : "lighthouse_ufo",
   "revision" : 3,
   "tracking_to_eye_transform" : [
      {
         "distortion" : {
            "center_x" : 0.0,
            "center_y" : 0.0,
            "coeffs" : [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ],
            "type" : "DISTORT_DPOLY3"
         },
         "distortion_blue" : {
            "center_x" : 0.0,
            "center_y" : 0.0,
            "coeffs" : [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ],
            "type" : "DISTORT_DPOLY3"
         },
```

```
      "distortion_red" : {
         "center_x" : 0.0,
         "center_y" : 0.0,
         "coeffs" : [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ],
         "type" : "DISTORT_DPOLY3"
      },
      "extrinsics" : [
         [ 1.0, 0.0, 0.0, 0.03115000016987324 ],
         [ 0.0, 1.0, 0.0, 0.0 ],
         [ 0.0, 0.0, 1.0, 0.0 ]
      ],
      "grow_for_undistort" : 0.0,
      "intrinsics" : [
         [ 1.250, 0.0, 0.0 ],
         [ 0.0, 1.0, 0.0 ],
         [ 0.0, 0.0, -1.0 ]
      ],
      "undistort_r2_cutoff" : 1.50
   },
   {
      "distortion" : {
         "center_x" : 0.0,
         "center_y" : 0.0,
         "coeffs" : [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ],
         "type" : "DISTORT_DPOLY3"
      },
      "distortion_blue" : {
         "center_x" : 0.0,
         "center_y" : 0.0,
         "coeffs" : [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ],
         "type" : "DISTORT_DPOLY3"
      },
      "distortion_red" : {
         "center_x" : 0.0,
         "center_y" : 0.0,
         "coeffs" : [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ],
         "type" : "DISTORT_DPOLY3"
      },
      "extrinsics" : [
         [ 1.0, 0.0, 0.0, -0.03115000016987324 ],
         [ 0.0, 1.0, 0.0, 0.0 ],
         [ 0.0, 0.0, 1.0, 0.0 ]
      ],
      "grow_for_undistort" : 0.0,
      "intrinsics" : [
         [ 1.250, 0.0, 0.0 ],
         [ 0.0, 1.0, 0.0 ],
         [ 0.0, 0.0, -1.0 ]
      ],
      "undistort_r2_cutoff" : 1.50
   }
],
```

```
   "head" : {
      "plus_x" : [ 1, 0, 0 ],
      "plus_z" : [ 0, 0, 1 ],
      "position" : [ 0, 0, 0 ]
   },
   "imu" : {
      "acc_bias" : [ 0, 0, 0 ],
      "acc_scale" : [ 1, 1, 1 ],
      "gyro_bias" : [ 0, 0, 0 ],
      "gyro_scale" : [ 1, 1, 1 ],
      "plus_x" : [ 0, 1, 0 ],
      "plus_z" : [ 1, 0, 0 ],
      "position" : [ 0.0034600000362843275, 0.0013079999480396509,
0.077326998114585876 ]
   },
    "type" : "Lighthouse_HMD"
}
```