dorodnic Merge pull request #5579 from krazycoder2k/patch-2 ...

7 contributors

Raw Blame

309 lines (246 sloc) 16.3 KB

T265 Tracking Camera

The **Intel® RealSense™ Tracking Camera T265** includes two greyscale cameras with fisheye lens, an IMU, and an Intel® Movidius™ Myriad™ 2 VPU. All of the V-SLAM algorithms run directly on the VPU, allowing for very low latency and extremely efficient power consumption (1.5W).

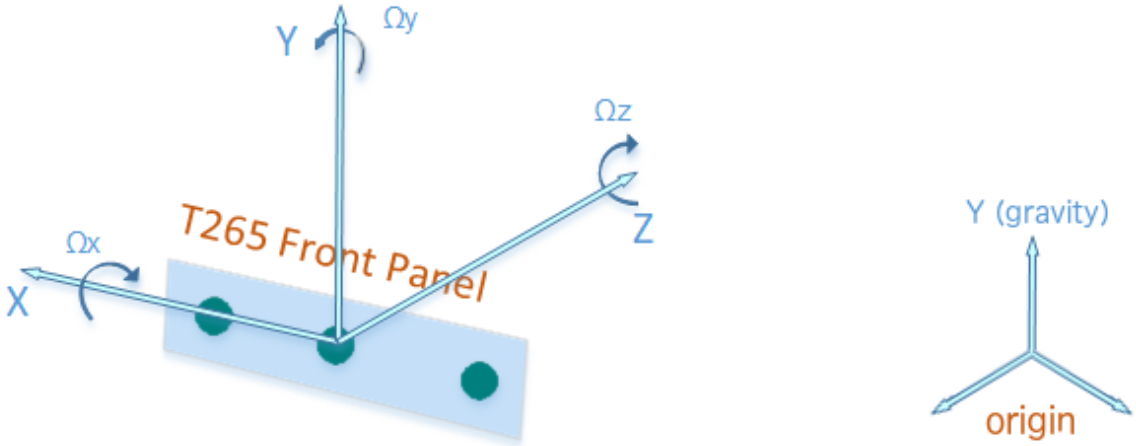
RealSense SDK currently supports T265 on Windows and Linux as well as via our ROS wrapper.

Notes / known issues

- For wheeled robots, odometer input is a requirement for robust and accurate tracking. The relevant APIs will be added to librealsense and ROS/realsense in upcoming releases. Currently, the API is available in the underlying device driver.
- The relocalization API should be considered unstable, and is planned to be updated in a future release
- Android support will come in a future release, as well as OpenVR integration.

Sensor origin and coordinate system

To aid AR/VR integration, the T265 tracking device uses the defacto VR framework standard coordinate system instead of the SDK standard:



1. Positive X direction is towards right imager
2. Positive Y direction is upwards toward the top of the device
3. Positive Z direction is inwards toward the back of the device

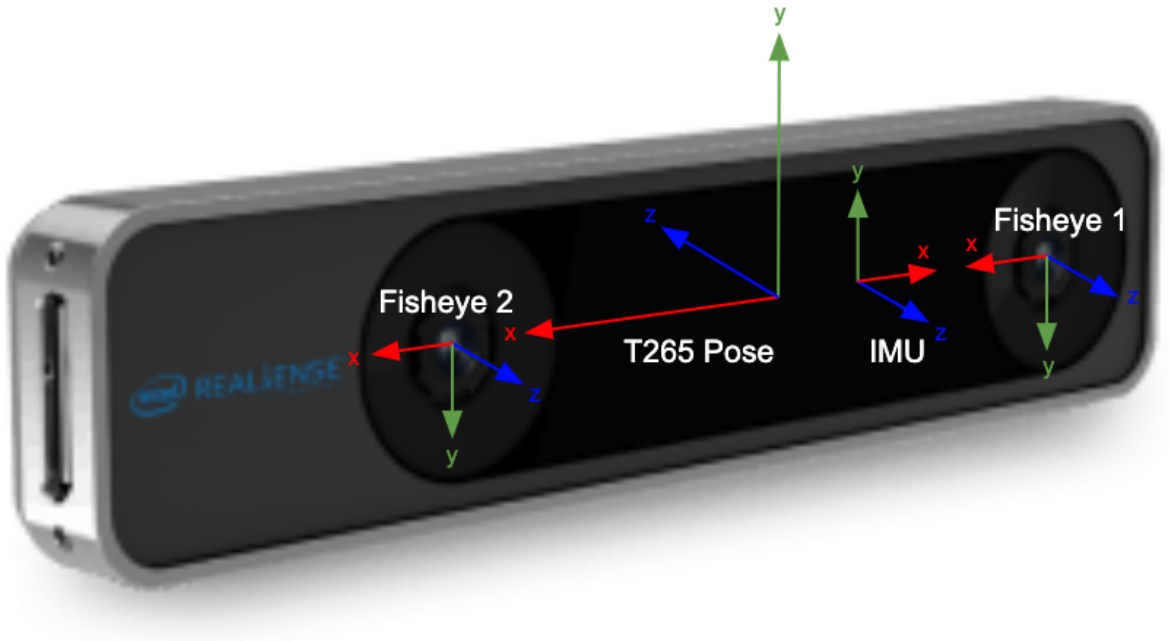
The center of tracking corresponds to the center location between the right and left monochrome imagers on the PCB.

When T265 tracking starts, an origin coordinate system is created and RealSense SDK provides T265 poses relative to it. World's Y axis is always aligned with gravity and points to the sky. World's X and Z axes are not globally set, but determined when tracking starts based on the initial orientation of the T265 device with the initial -Z world axis as the projection of the camera axis to the ground plane. This does mean that the initial yaw can seem random when the device is started in a downward facing configuration, say on the drone. All T265 (and librealsense) coordinate systems are right-handed.

Calibration

The T265's sensors (including the IMU) are calibrated on the production line, so no further calibration process is required (unlike the IMU on the D435i). Refer to the rs-ar-basic example to see how to get and use sensor intrinsics and extrinsics for the cameras and IMU.

For reference, this is the orientation of each sensor:



Timestamps

The T265 always reports the time for all sensors in `RS2_TIMESTAMP_DOMAIN_GLOBAL_TIME`. This uses a time synchronization mechanism between the host computer and the T265 to provide all sensor data timestamps in millisecond accurate host clock time.

Examples and tools

The following `librealsense` examples and tools work with T265 (see the full of C++ examples, Python examples and tools):

C++ Examples

- rs-pose - A basic pose retrieval example
- rs-pose-predict - Demonstrates pose prediction using librealsense global time and the callback API
- rs-pose-and-image - Demonstrates how to use tracking camera asynchroniously to obtain 200Hz poses and 30Hz images
- rs-pose-apriltag - Demonstrates how to compute Apriltag pose from T265 fisheye image stream
- rs-ar-basic - Shows how to use pose and fisheye frames to display a simple virtual object on the fisheye image
- rs-tracking-and-depth - Shows how to use the tracking camera together with a depth camera to display a 3D pointcloud with respect to a static reference frame

- rs-trajectory - This sample demonstrates how to draw the 3D trajectory of the device's movement based on pose data.
- rs-capture - 2D visualization of sensor data.
- rs-save-to-disk - Shows how to configure the camera and save PNGs
- rs-multicam - Work multiple cameras streams simultaneously, in separate windows
- rs-software-device Shows how to create a custom `rs2::device`
- rs-sensor-control - A tutorial for using the `rs2::sensor` API

Python examples

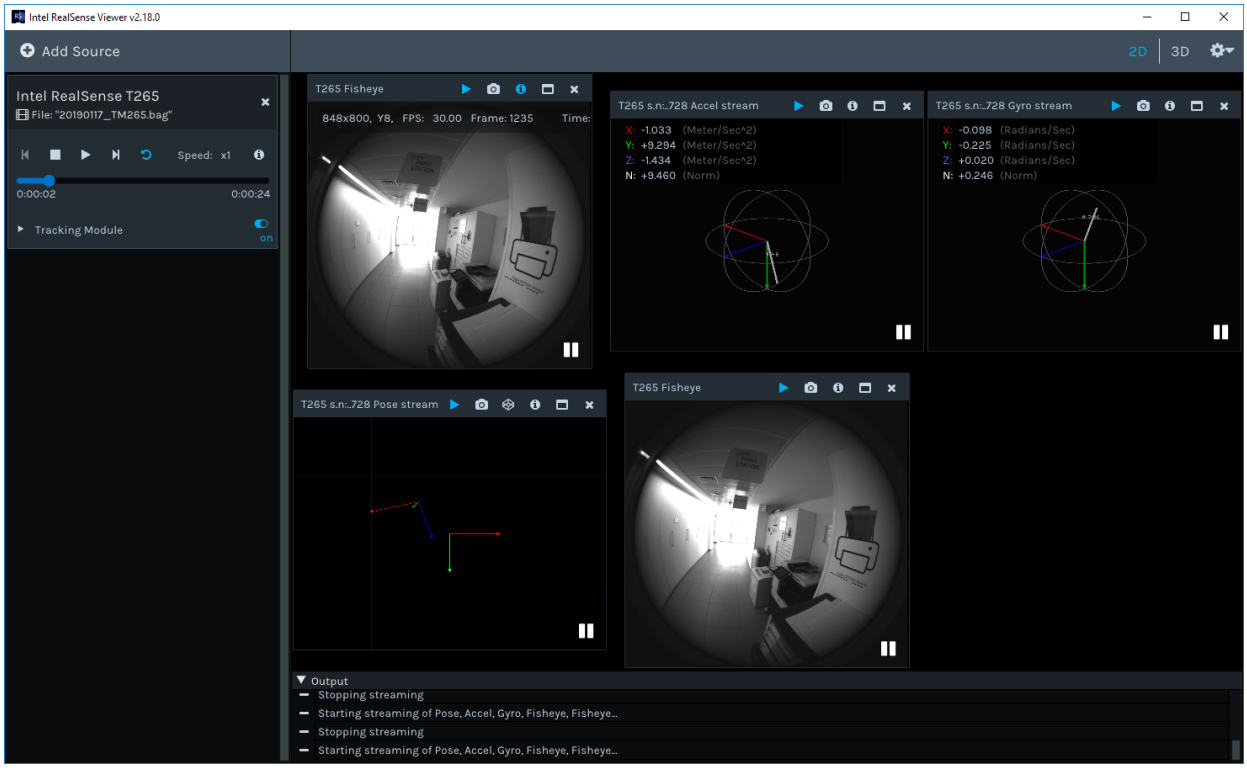
- T265 Basic - Demonstrates how to retrieve pose data from a T265
- T265 Coordinates - This example shows how to change coordinate systems of a T265 pose
- T265 Stereo - This example shows how to use T265 intrinsics and extrinsics in OpenCV to asynchronously compute depth maps from T265 fisheye images on the host.

Tools

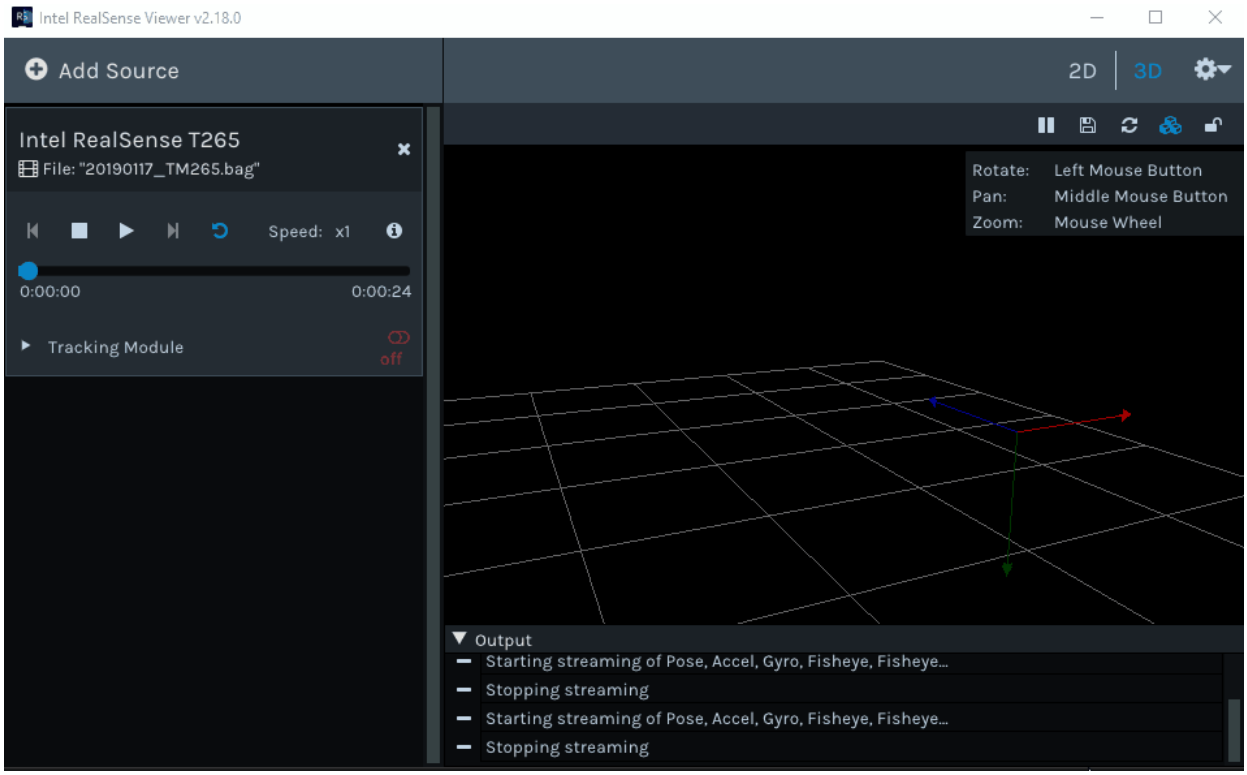
- enumerate-devices - Console application providing information about connected devices
- recorder - Simple command line raw data recorder
- data-collect - Collect statistics about various streams.

Realsense viewer

As with all Realsense cameras, T265 works with realsense-viewer. It provides visualization of all sensor streams, including a 3D visualization for pose samples:



The IMU and tracking data streams are fully compatible with SDK's embedded recorder utility.



If you want to record raw sensor data including images in ROS `.bag` format, we recommend using recorder

API

The pose and IMU data are treated by the SDK like any other supported sensor. Therefore the sensor access and invocation API calls are similar to those of the depth/rgb sensors of D400 and SR300:

```
rs2::pipeline pipe;

rs2::config cfg;
cfg.enable_stream(RS2_STREAM_GYRO);
cfg.enable_stream(RS2_STREAM_ACCEL);
cfg.enable_stream(RS2_STREAM_POSE);

pipe.start(cfg);

while (app) // Application still alive?
{
    rs2::frameset frameset = pipe.wait_for_frames();

    // Find and retrieve IMU and/or tracking data
    if (rs2::motion_frame accel_frame = frameset.first_or_default(RS2_STRE
    {
        rs2_vector accel_sample = accel_frame.get_motion_data();
        //std::cout << "Accel:" << accel_sample.x << ", " << accel_sample.
        //...
    }

    if (rs2::motion_frame gyro_frame = frameset.first_or_default(RS2_STREA
    {
        rs2_vector gyro_sample = gyro_frame.get_motion_data();
        //std::cout << "Gyro:" << gyro_sample.x << ", " << gyro_sample.y <
        //...
    }

    if (rs2::pose_frame pose_frame = frameset.first_or_default(RS2_STREAM_
    {
        rs2_pose pose_sample = pose_frame.get_pose_data();
        //std::cout << "Pose:" << pose_sample.translation.x << ", " << pos
        //...
    }
}
```

FAQ

Is this a depth camera?

No, the T265 does not offer any Depth information. It can provide position and orientation (6-DOF), IMU (accelerometer and gyroscope), and two monochrome fisheye streams.

Can I use it together with a depth camera?

Yes, this device can be used with RealSense (and most likely any 3rd-party) depth cameras. It is passive and comes with an IR-cut filter.

How can I use the camera intrinsics and extrinsics?

The T265 uses the Kanalla-Brandt distortion model. This model is supported in OpenCV as part of the fisheye model. There is a python example which uses the camera intrinsics and extrinsics to compute depth from T265 images on the host using OpenCV. There is also a C++ example which uses the camera intrinsics and extrinsics to display a virtual object.

What platforms are supported?

The T265 is supported via librealsense on Windows, Linux, macOS and via ros-realsense. Android support are planned but not scheduled yet, as well as OpenVR integration.

What are the system requirements?

Any board with USB2 capable of receiving pose data at 200 times a second should be sufficient. The device was validated on Intel NUC platform. To access the fisheye streams, USB3 is required.

Will the same SLAM be available for the D435i as a software package?

We are still looking into it, please stay tuned.

What is wheel odometry and how does it help T265 navigate?

Wheel odometry is the use of sensors to measure how much a wheel turns, it can be used to estimate changes in a wheeled robots position. T265 has wheel odometry support built in, allowing it to use the data from these sensors to refine the position data of the robot. Providing robotic wheel odometer or velocimeter data over USB to TM2 and the corresponding calibration data will make the tracking much more robust on wheeled robots, which otherwise can experience many tracking failures. The calibration file format is described in the section below. We consider odometer input to be a requirement for robust tracking on wheeled robots.

Can T265 work indoors and outdoors?

Yes, T265 can work both indoors and outdoors. Just like a person, it can be blinded by light that shines straight into its eyes, and it can’t see in absolute darkness.

Do multiple T265 devices interfere with each other?

No, you can use as many T265 devices in a space as you like.

Are there any T265 specific options?

Yes, there are.

- `RS2_OPTION_ENABLE_MAPPING` - The internal map allows the device to recognize places it's been before so that it can give a consistent pose for that location. The map has a fixed finite size and will keep the most often/recently seen locations. Without this it will be operating completely open loop (i.e. just doing VIO) and will have more drift. The device will use the map to close modest loops and recover from small drifts. Enabling this option will not cause pose jumps without `RS2_OPTION_ENABLE_POSE_JUMPING`.
- `RS2_OPTION_ENABLE_POSE_JUMPING` - This option allows the device to discontinuously jump its pose whenever it discovers its pose is inconsistent with one it has given before. For example, after walking in a circle or covering the camera (getting a small drift) and uncovering it. Currently this will only affect the translation and not the rotation nor any of the velocities or accelerations.
- `RS2_OPTION_ENABLE_RELOCALIZATION` - This allows the device to solve the Kidnapped Robot Problem, i.e. it will allow connecting the current map to a loaded map or connecting the current map to itself after accumulating a large drift, say after closing a large loop or covering the camera and walking around. This is independent of jumping the pose. When fooled this feature can lead to much larger errors than are likely via the basic mapping.
- `RS2_OPTION_ENABLE_MAP_PRESERVATION` - This option will allow the device to preserve its current map across stop/start calls. The device will act as if the map was saved after stop and loaded back before the subsequent start. This was the default in versions <= 2.29.0.

Appendix

Wheel odometry calibration file format

Below is a sample calibration file with inline comments (not a valid file format! Remove "//comments" to make valid) that explain each field. Please note that this is an intermediate solution and a more unified interface will be coming.

```
{
    "velocimeters": [ // array of velocity sensors (currently max. 2
supported)
        {
            "scale_and_alignment": [ // 3-by-3 matrix, row-major
```

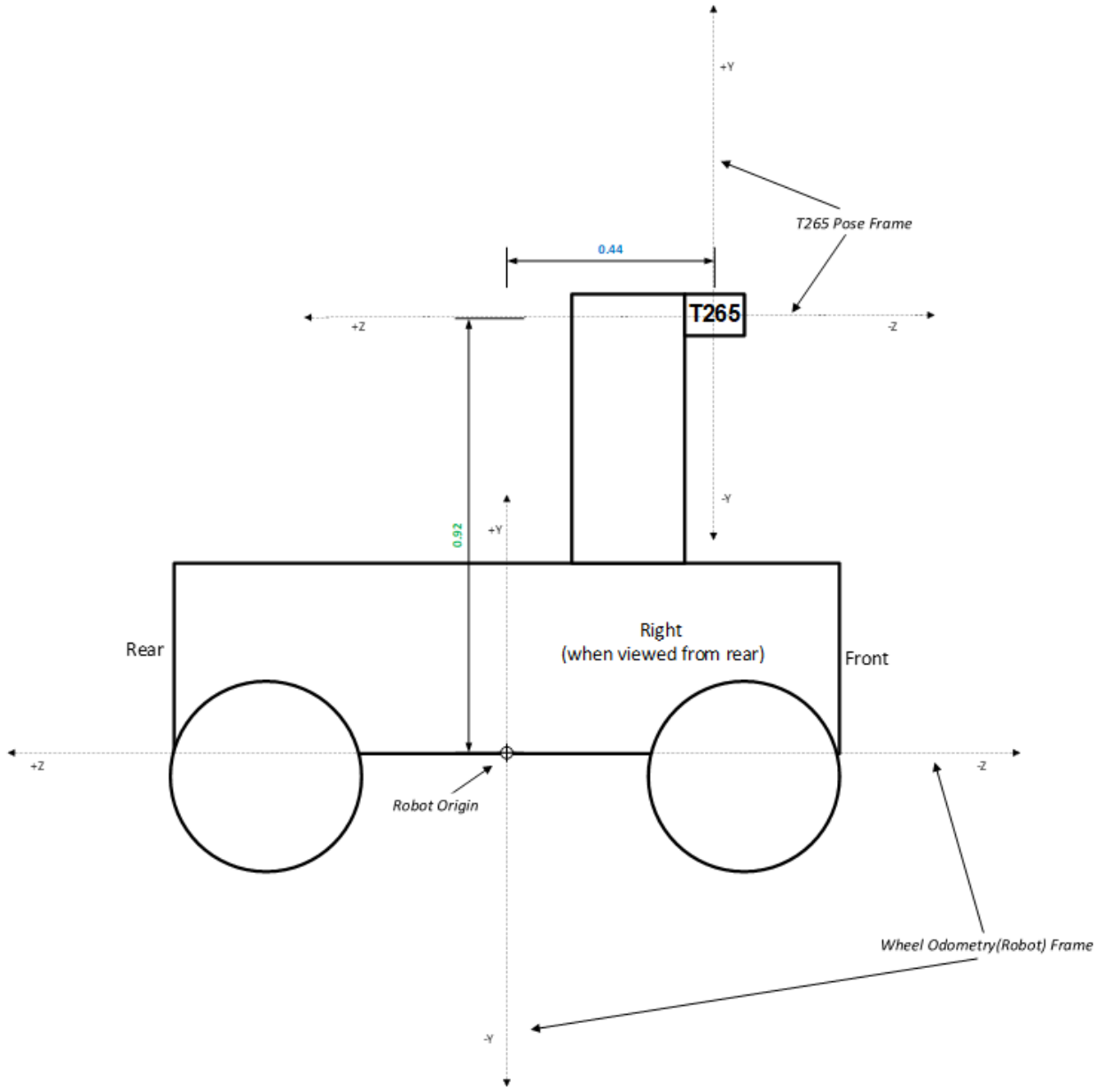
```
    (multiplies measurement)
                    1.0,
                    0.0000000000000000,
                    0.0000000000000000,
                    0.0000000000000000,
                    1.0,
                    0.0000000000000000,
                    0.0000000000000000,
                    0.0000000000000000,
                    1.0
                ],
                "noise_variance": 0.004445126050420168, // measurement
    covariance (to be determined/tuned)
                "extrinsics": { // relative transformation of sensor frame
    w.r.t. T265 body frame
                    "T": [ // translation (in meters)
                        -0.5059,
                        -0.6294,
                        -0.6873
                    ],
                    "T_variance": [ // currently not used, please ignore
                        9.999999974752427e-7,
                        9.999999974752427e-7,
                        9.999999974752427e-7
                    ],
                    "W": [ // orientation in axis-angle representation,
    axis x angle (in rad)
                        -1.1155,
                        -1.1690,
                        -1.2115
                    ],
                    "W_variance": [ // currently not used, please ignore
                        9.999999974752427e-5,
                        9.999999974752427e-5,
                        9.999999974752427e-5
                    ]
                }
            }
        ]
    }
```

Extrinsic Calibration for Wheeled Odometry Examples

- All calibration metrics are relative to T265 origin frame. I.e.: They are offsets/rotations *from* the T265 origin *to* the robot's origin. Said another way, we transform a point from frame B (Robot Odometry) to frame A (T265 Pose) or A_p = H_AB * B_p, where A_p is the point expressed in frame A, B_p is the point expressed in frame B, and H_AB is the corresponding homogeneous transformation.

- In order for the T265 to consume and use odometry data correctly it must know where the robot's origin is relative to itself. This is accomplished by feeding a calibration (json) file into the librealsense API with this initial data.

In this basic sample robot setup, for simplicity, we assume the X-axis of the camera and robot's coordinate system are aligned (+X coming out of the screen) and only concern ourselves with the Y and Z axis.

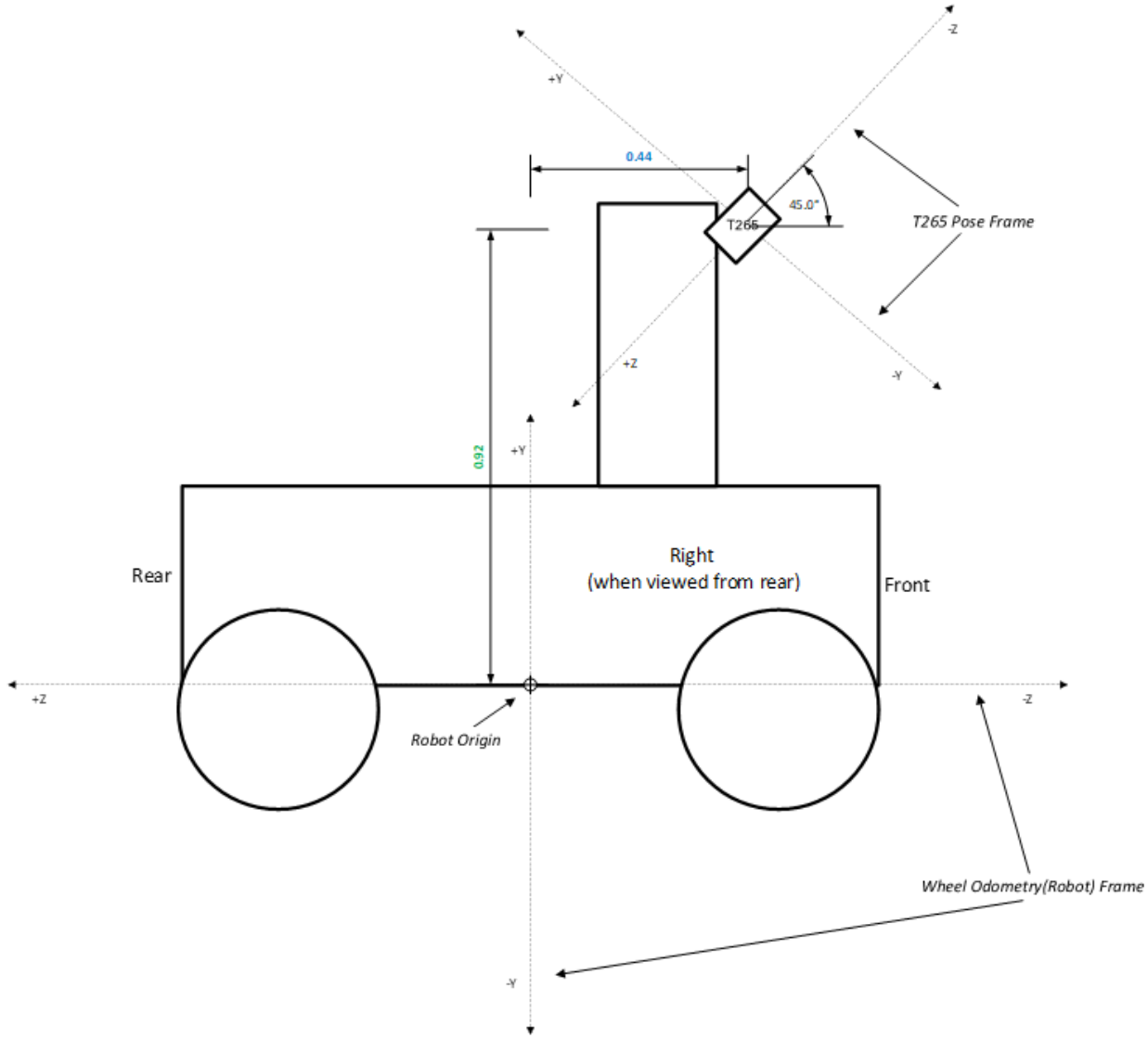


Corresponsing JSON:

```
{
   "velocimeters": [
       {
           "scale_and_alignment": [
               1.0,
               0.0,
               0.0,
               0.0,
               1.0,
```

```
                    0.0,
                    0.0,
                    0.0,
                    1.0
                ],
                "noise_variance": 0.00004445126050420168,
                "extrinsics": {
                    "T": [
                        0.0,      // No Translation.
                        -0.92,   // 0.92m below (-Y) the camera.
                        0.44    // 0.44m behind (+Z) the camera.
                    ],
                    "T_variance": [
                        9.999999974752427e-7,
                        9.999999974752427e-7,
                        9.999999974752427e-7
                    ],
                    "W": [
                        0.0,
                        0.0,
                        0.0
                    ],
                    "W_variance": [
                        9.999999974752427e-5,
                        9.999999974752427e-5,
                        9.999999974752427e-5
                    ]
                }
            }
        ]
    }
```

In this example, we take the above setup and add a rotation around the camera's X-axis.



Corresponding JSON:

```
{
   "velocimeters": [
       {
           "scale_and_alignment": [
               1.0,
               0.0,
               0.0,
               0.0,
               1.0,
               0.0,
               0.0,
               0.0,
               1.0
           ],
           "noise_variance": 0.00004445126050420168,
           "extrinsics": {
               "T": [
                   0.0,      // No Translation.
                   -0.3394,  // 0.3394m below (-Y) the camera.
                   0.9617    // 0.9617m behind (+Z) the camera.
               ],
```

```
                "T_variance": [
                    9.999999974752427e-7,
                    9.999999974752427e-7,
                    9.999999974752427e-7
                ],
                "W": [
                    -0.7853982,  // -0.78rad (-pi/4) around the camera
    x-axis.
                    0.0,
                    0.0
                ],
                "W_variance": [
                    9.999999974752427e-5,
                    9.999999974752427e-5,
                    9.999999974752427e-5
                ]
            }
        }
    ]
}
```