Autobots VIP Fall 2022
Final Report

Anthony Miyaguchi
acmiyaguchi@gatech.edu

2022-12-11

Contents

Initial Expectations

I joined Autobots Vertically Integrated Project (VIP) this fall because I wanted to get an opportunity to try out research as part of my master's studies at Georgia Tech. It is my second semester in the Online Masters in Computer

Science (OMSCS) program, so I have been cautious not to over-commit the workload since I am working full-time in addition to my studies. I signed up for one credit hour to gauge whether it would be worth spending three credit hours on the project next semester. I knew I wouldn't be able to build anything substantial this semester, but it would lay down a foundation for potential future work.

My goals for this semester were to understand the problem space for assistive autonomous robots and to clearly articulate what challenges exist in a more constrained and concrete task. As a remote graduate student, I knew I would be primarily semi-independent with my learning experience and work. I expected hands-on experience with the toolchain for simulating a robotic task. I was also interested in collaboration and interaction with other students in the section, but I knew this would take a lot of work to do remotely.

Overview of Progress

I also chose to work on Vision-Based Manipulation (VisMan). The semester primarily focused on working through the VisMan Learning Adventures in the wiki. I've been able to run a manipulator (both Edy and Handy), and camera (RealSense D435) in Gazebo. I wrote functional unit tests to verify the camera's functionality. I have not progressed beyond tabletop calibration.

The first few weeks of the semester involved installing Linux, ROS, and Gazebo on my computer. I chose to install ArchLinux, and as a result spent time filing issues and making small pull requests to fix bugs in upstream dependencies. I learned about ROS and models by making a few patches to the simData and simData_imgSaver repositories under the IVALab organization. I could load models used in the experiments and learned how to export models for downstream catkin projects.

I configured a camera and an environment to demonstrate the pick-and-place task for the remainder of the semester. To understand camera calibration, I had to read through the first three weeks of ECE4850 notes. I was also able to create a surface to be used for tabletop calibration. I leave the completion of this task to next semester.

Notable Achievement

One of the most significant milestones this semester was building a catkin package that includes unit tests to verify the functionality of an RGB-D camera in a simulated environment. The ability to unit-test that a camera is notable because it requires several prerequisites. First, we can install and run arbitrary ROS packages from the source. These are installed by cloning git repositories into the workspace or using the system package manager (i.e., `pacman` via the AUR) to install it on the path. We added the following packages to our workspace:

```
git clone --recurse-submodules git@github.com:rickstaa/realsense-ros-gazebo.git
git clone git@github.com:machinekoder/ros_pytest.git
```

After installing the packages, we can verify the camera in Gazebo and RViz.

```
roslaunch realsense2_description view_d435_model_rviz_gazebo.launch
```

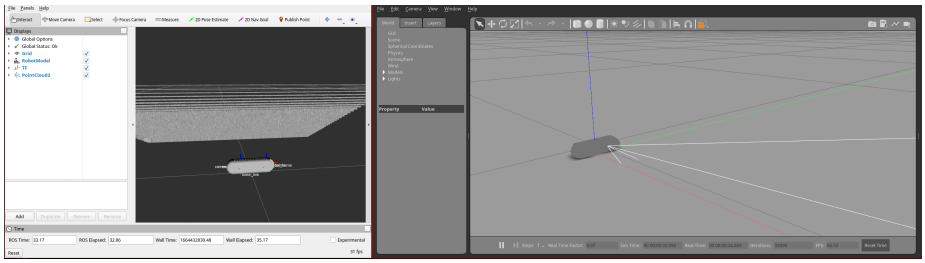


Figure 1: d435 gazebo rvis

We add a red box to the environment so that we have something to observe. We add a new **sdf** model to the project directory. We write a script to obtain color and depth images, which runs on delay as a node in a launch file for testing the camera.

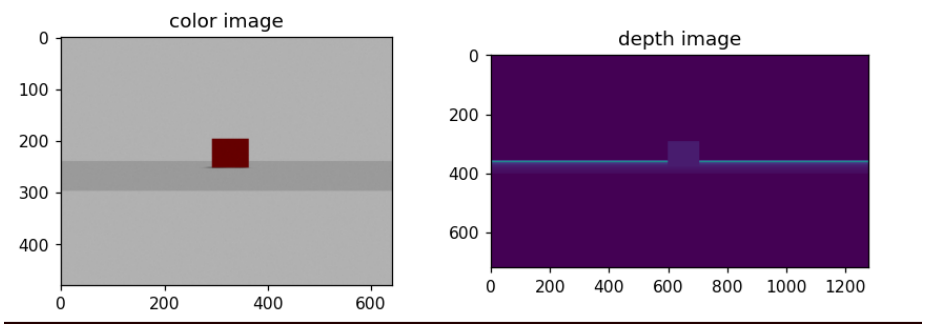


Figure 2: bridge block

To wrap everything together, we create a test directory that takes advantage of catkin tests and the **pytest** framework.

```
<!-- tests/camera_box.test -->
<launch>
  <include file="$(find visman_learning_adventure)/launch/camera_box.launch">
    <arg name="GUI" value="false" />
  </include>

  <param name="test_module" value="$(find visman_learning_adventure)/tests" />
  <test
```

```
    test-name="camera_box"
    pkg="ros_pytest"
    type="ros_pytest_runner"
    args="-k camera_box -vvv -s"
    time-limit="60.0"
  />
</launch>
```

Because it's just a launch file, we can include other launch files to set up the environment before running the test node. The test does two things: first, it checks that the camera can detect red pixels from the OpenCV bridge, and second, the number of red pixels monotonically increases as the box moves toward the camera.

```
$ catkin test --this

[visman_learning_adventure.rosunit-camera_box/test_camera_can_see_red_box][passed]
[visman_learning_adventure.rosunit-camera_box/test_camera_red_increases_when_box_moves_close

SUMMARY
 * RESULT: SUCCESS
 * TESTS: 2
 * ERRORS: 0
 * FAILURES: 0
```

These are simple tests, but verifies a valid environment and correct behavior. This milestone was significant because it helped me build a good mental model around how to build software in ROS and Gazebo from the ground up.

Refer to the report appendix or the camera documentation on GitHub for more details.

Future Work

Here are a few things I'd like to do if I had three more months to work on this project. The ultimate goal is to build foundational knowledge that I might use in the future as an academic or professional and to have concrete deliverables that I can present at the Spring 2023 OMSCS showcase.

VisMan Learning Adventure Skeleton Code

I want to finish the VisMan learning adventure and consolidate my work into a skeleton that can serve as an exercise for a future student. The exercises in the wiki are challenging (and achievable). However, it would aid learning and ramp-up to have a starting point for the tasks that require an understanding in robotics or computer vision theory. A skeleton project would formalize the

exercises with enough code to get started quickly. For example, consider the following task in the camera section of the learning series:

> Connect to the depth camera and display the streaming RGB and depth images.

We might provide a future student with an empty launch file with a python node that runs a script that displays an empty plot in a loop. The student would then have to complete the following:

1. Add the appropriate ROS nodes to instantiate a camera and ensure it writes to the ROS topic.
2. Add the appropriate code to read from the OpenCV bridge and show the image through the screen via `matplotlib`.

One challenge with this approach is that modules build on each other, which makes it difficult to write later modules without spoiling solutions for earlier modules. I need to finish the series for myself before I'm sure how best to simplify skeleton code while leaving the conceptual challenges intact.

Collaboration with the Multi-Robot Coordination Team

One of the main goals established this semester was to work toward a pick-and-place task. Pick-and-place is one of the fundamental tasks of a vision-based manipulator, and it provides a relatively good context for understanding research topics in the area.

It would be nice to collaborate with the multi-robot coordination team, such as building a simulation environment that they could use for the manipulation portion of their payload delivery task. There is existing work from previous semesters to build a model of the fourth floor of the TSRB building, which is a helpful starting point.

Literature Review and Posing a Research Problem

The last thing I want to do is a literature review. I've gotten a good sense of ROS and Gazebo, so I want to expand my knowledge and understanding of the state-of-the-art in the vision-based manipulation space. After reading a few papers, I would like to propose a research problem. I would deliver this as a presentation or a report with a summary of techniques for a particular research problem. If time permits, I would like to study or reproduce an implementation of an algorithm using skills that I've developed in earlier portions of the project. I will likely move on to focus on the rest of my master's program after the end of the school year, but it would be an interesting exercise to think through what it would take to solve a problem in the area.

Thoughts on Experience

The discussion sessions were an excellent opportunity to learn a variety of problem spaces and solutions from the literature. Out of the presentations, I enjoyed topics about planning and navigation the most, such as the graph-based planner for cave exploration and sequential scene understanding and manipulation (SUM). I enjoyed learning about different approaches and gaining a breadth around autonomous robotics. I was one of the first and last presenters, so my presentation was heavily focused on what I had built, but it would be interesting to do a research oriented presentation too. As a side-note, it would be nice to consolidate all the paper references into a single document reference.

I also enjoyed building out the catkin project and learning ROS and Gazebo. It was challenging, but I now feel comfortable reading and modifying other packages for my own use cases. I also had the opportunity to contribute to open-source projects like eigenpy. I also appreciated design choices made with ROS, coming in with the perspective of a professional software engineer. Building out my package from scratch was an excellent exercise for learning about robotics software development.

I spent most of the budgeted time working on the learning adventures and got stuck a few times. After spending a few hours reading through the documentation, forums, issue trackers, and source code, I resolved the issues myself. However, it would have been nice to have the opportunity to ask questions in office hours. It was challenging to reach out with questions about office hours in TSRB because I am remote. Regardless, the project's structure was a good fit for me since I could work and learn at my own pace. I was initially concerned about being the only remote member of the group and the only graduate student. The Zoom meetings and Trello board worked well for me; I could attend the discussion sessions synchronously and update my learning progress asynchronously.

Documentation

I have put all relevant documentation and code on GitHub at acmiyaguchi/autobots_visman. It is the canonical source for work that I have done in Autobots.

My progress can be tracked in the Fall 2022 VisMan planning card in Trello. The two primary tickets that I worked on this semester were the "Add demo of the virtual camera with readings (color/depth)" and "Model the task workspace plane" cards. These cards have links to the relevant GitHub issues and pull requests. Other tickets in the central planning card capture other miscellaneous work that I've done, such as pull requests I've made to ivaHandy, ivaEdy, and the simData repositories.

The 2022-09-20 presentation summarizes work toward the beginning of the semester. The 2022-11-29 presentation summarizes work up to the end of the semester. Both of these presentations are found on the shared Autobots drive.

Appendix

Working with Cameras

For this section, I worked through adding a RGB-D (color and depth) camera into my simulation environment while working through a vision-based manipulation tutorial. I chose an appropriate camera to simulate, installed the appropriate packages, and wrote roslaunch files to test out various aspects of the camera.

Configuring the RGB-D camera

I took a look at adding a Kinect or Realsense D435i into the environment, since these are the cameras available in the lab. After reading through a nice summary in GraspKpNet, I decided to go with the RealSense.

> We are supporting two types of cameras: Kinect Xbox 360 and Realsense D435. Personally I will recommend Realsense since Kinect is pretty old and its driver isn't very stable.

I went ahead and installed the realsense-ros-gazebo package into my workspace, which includes the following catkin packages:

- `realsense_gazebo_plugin` - plugin for generating simulated sensor readings
- `realsense_description` - description/model for interacting with the camera
- `realsense_camera` - ros module for interacting with a physical camera

The fork contains a few handy launch files, and is much easier than the alternative packages that are available. I had to install the following dependencies to build the package:

- ros-noetic-ddynamic-reconfigure
- librealsense-git

The first is a build system dependency (because `rosdep` is not an option on Arch), while the second contains the drivers for the Realsense camera. This is actually not necessary since I only need the description files to work in simulation, but it is cleaner to build all the dependencies instead of adding an exclude to the builds in my workspace.

Once installed, we can test the camera in both rviz and gazebo:

```
roslaunch realsense2_description view_d435_model_rviz_gazebo.launch
```

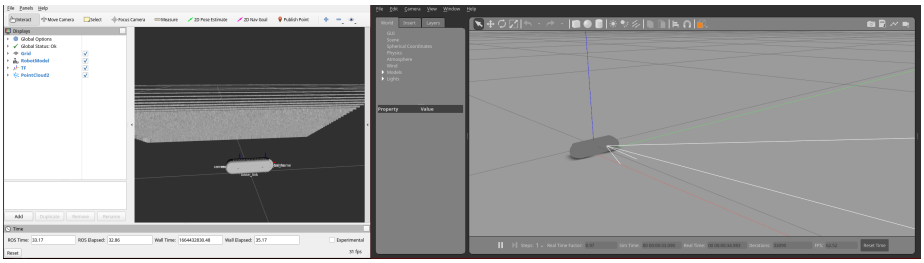


***figure**: Rviz on the left shows the sensor readings from the D435 RGB-D camera. Gazebo on the right shows the camera situation in a simulated world.*

Adding a red box to the environment

The next step was to create a custom launch file that can be used to test out the sensor readings using an object of my choice. I decided to pull in code from a couple of different sources:

- ivaLab/simData_imgSaver/models/box/description.sdf - an initial box description, the repository was generally a good reference for how I should set up my own models directory
  - [Gazebo] - Adding Color and Textures to a Model also followed this tutorial to make the box red
- rickstaa/realsense-ros/realsense2_description/launch/view_d435_model_rviz_gazebo.launch - the launch file for rviz and gazebo

One particularly useful trick is to set up the model path in `package.xml` with the following lines:

```
<export>
    <!-- https://answers.gazebosim.org/question/6568/uri-paths-to-packages-in-the-sdf-model-
    <!-- gazebo_ros_paths_plugin automatically adds these to
        GAZEBO_PLUGIN_PATH and GAZEBO_MODEL_PATH when you do this export inside
        the package.xml file. You can than use URIs of type model://my_package/stuff. -->
    <gazebo_ros
    gazebo_plugin_path="${prefix}/lib"
    gazebo_model_path="${prefix}"
  />
</export>
```

Then the box can be spawned via `gazebo_ros` without having to muck around with environment variables before launching gazebo:

```
  <param
    name="model_description"
    textfile="$(find visman_learning_adventure)/models/box/description.sdf"
  />
  <arg name="distance" default="0.3" />
  <node
```

```
    name="model_spawner"
    pkg="gazebo_ros"
    type="spawn_model"
    respawn="false"
    output="screen"
    args="-sdf -x $(arg distance) -y 0.01 -z 0.02 -param model_description -model box"
  />
```

The full launch file can be viewed under visman_learning_adventure/launch/camera_box.launch.

```
roslaunch visman_learning_adventure camera_box.launch
```

Like before, we get a gazebo and rviz window with the camera sensor and model.
We also get a red box placed directly in front of the camera.

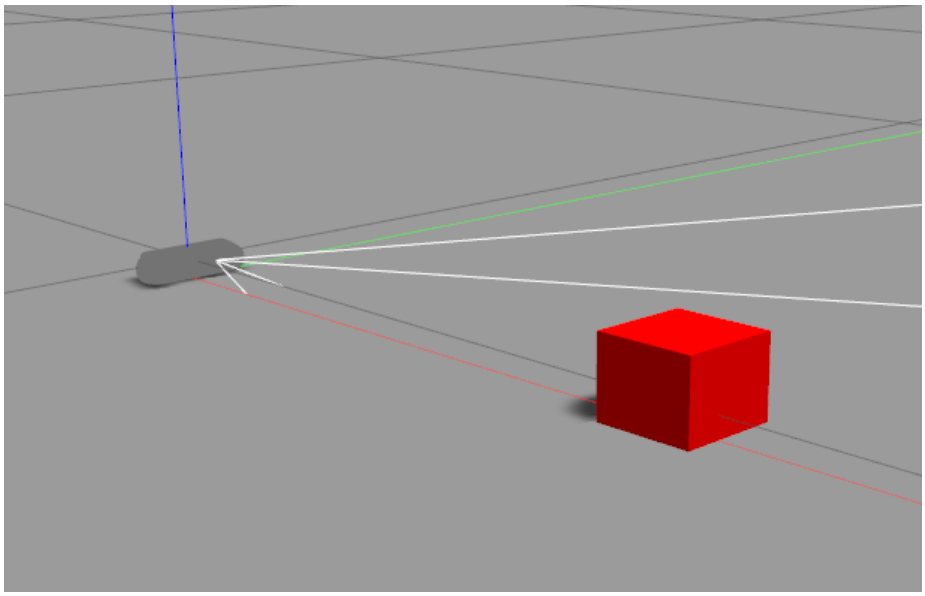


Figure 3: gazebo block

figure: *a red block positioned in front of the camera*

figure: *a depth reading of the block in front of the camera*

We can pass arguments into the launch to adjust the distance:

```
roslaunch visman_learning_adventure camera_box.launch distance:=0.5
```

This sets the position of the box relative to the origin (which happens to be the location of the camera).

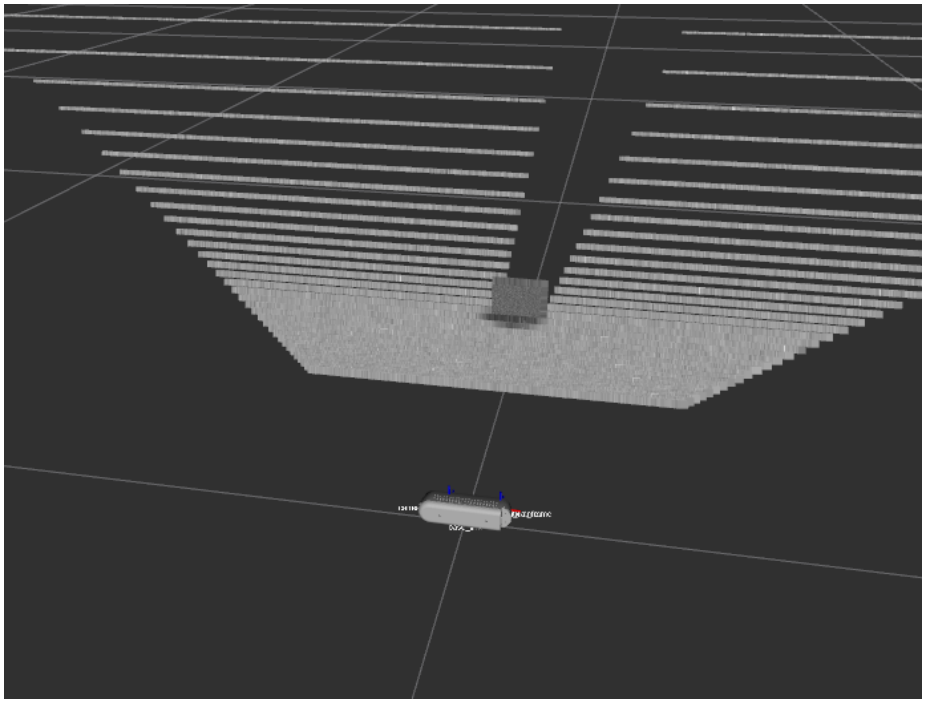


Figure 4: rviz block

Interacting with the camera and box topic

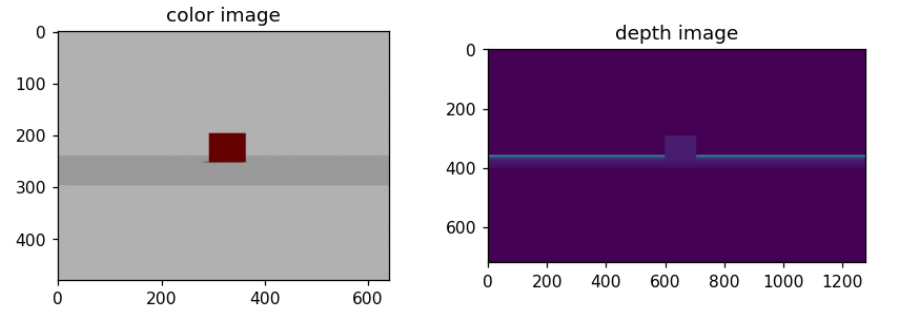
Now that we have the environment set up, we can start to interact with it. First, we launch `camera_box.launch` in a process, and open a terminal where we can interact with ros.

```
$ rostopic list

/camera/color/camera_info
/camera/color/image_raw
/camera/color/image_raw/compressed
/camera/color/image_raw/compressed/parameter_descriptions
/camera/color/image_raw/compressed/parameter_updates
...
```

We are interested in `/camera/color/image_raw` and `/camera/depth/image_raw` to display the image. With the OpenCV bridge, this is straightforward and can be run from a REPL:

```
import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import matplotlib.pyplot as plt

# required setup
rospy.init_node("realsense_subscriber")

# get a message from the topic containing a color image
# the first argument is set by the topic namespace passed into the included sensor gazebo la
# the second argument allows us to deserialize the object
msg = rospy.wait_for_message("/camera/color/image_raw", Image)

# decode the message and plot
bridge = CvBridge()
img = bridge.imgmsg_to_cv2(msg, msg.encoding)

# show the image
plt.imshow(img)
plt.show()
```

We demonstrate the use of the bridge inside of `realsense_image_snapshot.py` as part of the demo launch file.



On the left we get a color image of the box, and on the right we get a depth image of the box and horizon.

We can also manipulate the box in this environment. We follow this tutorial on moving the model from the command-line, and eventually translate this into a set of unit tests.

```
rosservice call /gazebo/get_model_state "model_name: box"
```

We get a yaml representation of the message as a result.

```
header:
  seq: 1
  stamp:
    secs: 163
    nsecs: 765000000
  frame_id: ""
pose:
  position:
    x: 0.3
    y: 0.01
    z: 0.02
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
twist:
  linear:
    x: 0.0
    y: 0.0
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 0.0
success: True
```

```
status_message: "GetModelState: got properties"
```

Now we can simply publish to the the `/gazebo/set_model_state` topic to move the box by a certain amount. We can do this one of two ways on the command line. The first is to use `rosservice` like we did above:

```
$ rosservice call /gazebo/set_model_state "
model_state:
  model_name: box
  pose:
    position:
      x: 0.3
"

success: True
status_message: "SetModelState: set model state done"
```

The other is to publish a message directly to the topic:

```
$ rostopic pub -1 /gazebo/set_model_state gazebo_msgs/ModelState "
model_name: box
pose:
  position:
    x: 0.5
"

publishing and latching message for 3.0 seconds
```

Either way works for quick testing, but we'll prefer to use python where possible.

Unit testing the camera

With the mechanics of subscribing and publishing to topics out of the way, we put together a unit test that performs the following:

- checks that we can subscribe to the `camera` topic and see the red box
- checks that as we move the box away from the origin, the amount of red is monotonically decreasing

This test helps verify that the environment is set up correctly. Inside of the `visman_learning_adventure` package directory, we can run the following command:

```
$ catkin test --this

...
[visman_learning_adventure.rosunit-camera_box/test_camera_can_see_red_box][passed]
[visman_learning_adventure.rosunit-camera_box/test_camera_red_increases_when_box_moves_close

SUMMARY
```

```
* RESULT: SUCCESS
* TESTS: 2
* ERRORS: 0
* FAILURES: 0
```

See tests/camera_box.test and the test_camera_box.py for the implementation of the unit tests.

We take use python_ros, which will need to be added to the catkin workspace. I personally find it much easier to use pytest over the nose or unittest packages. There are a few things that I found interesting while putting together the tests:

- the launch file for the test is like any other launchfile; here we pass arguments into our `camera_box.launch` file so we can run the tests without launching the gui interfaces for gazebo and rviz.
- tests were a great way to try out the rpc mechanisms in ros, but there is a downside of having to run the entire suite of tests alongside the 5-10 second startup time for launching the test.
- it's not easy to get standard out from the tests, which is unfortunate because printing ends up being a useful part of testing things. It might be useful to try the `--pdb` flag alongside the `-k <filter>` flags next time I'm writing a test.

Overall, it's reassuring that I can put in a simple validation of the environment, which I hope to incorporate into an docker image and maybe even continuous integration at a later date.

references

- https://classic.gazebosim.org/tutorials?tut=ros_gzplugins
- https://classic.gazebosim.org/tutorials?tut=ros_depth_camera&cat=connect_ros
- https://classic.gazebosim.org/tutorials?tut=color_model
- https://github.com/ivalab/GraspKpNet/blob/1f7546bc36e26e5c70877c21bfec33cf1ad97330/readme/experi
- https://github.com/ivalab/aruco_tag_saver_ros
- https://github.com/ivaROS/turtlebot_description/blob/06968fca967b5615be3344f2b8843dff9856bd9e/urd
- https://github.com/IntelRealSense/realsense-ros
- https://github.com/IntelRealSense/realsense-ros/issues/1453#issuecomment-710226770
- https://github.com/pal-robotics/realsense_gazebo_plugin
- https://github.com/rickstaa/realsense-ros-gazebo
- https://varhowto.com/how-to-move-a-gazebo-model-from-command-line-ros/
- https://answers.gazebosim.org//question/22125/how-to-set-a-models-position-using-gazeboset_model_state-service-in-python/
- https://answers.ros.org/question/261782/how-to-use-getmodelstate-service-from-gazebo-in-python/
- http://wiki.ros.org/rostest/Writing

- https://answers.ros.org/question/115526/is-test_depend-actually-used-for-anything/
- https://github.com/machinekoder/pytest-ros-node-example
- https://github.com/machinekoder/ros_pytest