AR Tag Tutorial

Austin Buchan

October 22, 2014

This document provides a tutorial on using AR (Augmented Reality) Tags to track the 3D location of markers using camera images in a ROS environment. We provide examples for using the tags with a webcam, and using the hand and head cameras on Baxter.

1 Introduction

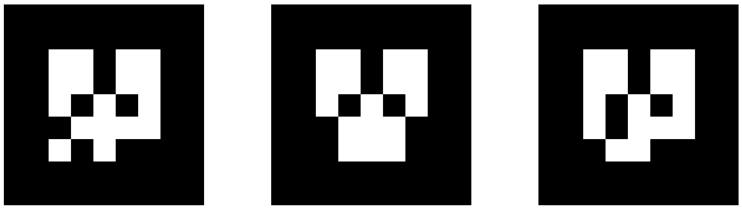

Figure 1: Example AR Tags

An AR Tag is usually a square pattern printed on a flat surface, such as the patterns in Figure 1. The corners of these tags are easy to identify from a single camera perspective, so that the homography to the tag surface can be computed automatically. The center of the tag also contains a unique pattern to identify multiple tags in an image. When the camera is calibrated and the size of the markers is known, the pose of the tag can be computed in real-world distance units.

There are several ROS packages that can produce pose information from AR tags in an image, we will be using `ar_track_alvar` [1]. If you are planning on using AR Tags, read through the documentation carefully before using this tutorial.

2 Webcam Tracking

2.1 Setup

1. Download the package to the `src` directory of a ROS workspace with

   ```
   git clone https://github.com/sniekum/ar_track_alvar.git
   ```

2. Download the AR Tag Resources zip file from the Piazza website, and unzip this to the `launch` directory of the `ar_track_alvar` package.

3. Edit `webcam_track.launch` the update the the `camera_info_url` parameter to have the full path to the `usb_cam.yml`.

   IMPORTANT NOTE: You need to leave the `file:///` in front of the path to the yml file. The parameter is expecting a web URL, but the `file:///` tells it to look in the local file system.

[1] `http://wiki.ros.org/ar_track_alvar`

`pwd` will print the full path to the current directory.

4. If any other parameters have changed, such as the name of the webcam, make sure they are consistent in the launch file.
5. Run `catkin_make` from the workspace (this may take a while).
6. Find or print some AR Tags. There should be a class set of 4 in Cory 119 and SDH 133. Please only use these for testing, and leave them unmodified so others can use them. The `ar_track_alvar` documentation has instructions for printing more tags that you can use in your project.

2.2 Calibration

If you use a camera other than the lab webcams, you will need to run the camera calibration yourself. This process collects images of a standardized grid, and computes calibration parameters to correctly map camera pixels to straight rays in the world. This information is stored in a YAML file (`*.yml`), which we have provided for the lab webcams.

I used the `camera_calibration` [2] package to calibrate our webcams by following the monocular calibration tutorial. To do this you will need to download and make the the package. An important step here is selecting the correct branch of of the package to use. The lab computers are running ROS Groovy, so after cloning the repository you would switch branches with:

```
git checkout groovy-devel
```

You will then need to wave a calibration grid around the camera view until it collects enough images to calculate. There is a grid with 1 inch squares available in Cory 119, or you can create one using the instructions in the camera calibration documentation. When this is complete, commiting the calibration should work. Try this first on your platform, and see if you get the success message "writing calibration data to ..." in the console.

However, on the lab computers the program tended to hang after clicking the commit button, without updating any information. To work around this, you can click the save button instead to produce a zip file of the calibration information. This contains a `ost.txt` file with the info for the YAML file. The `camera_calibration_parsers` [3] package can then convert this to a YAML file (check the doc page). You then need to update the `usb_cam` node `camera_info_url` to point to this new file.

2.3 Visualizing results

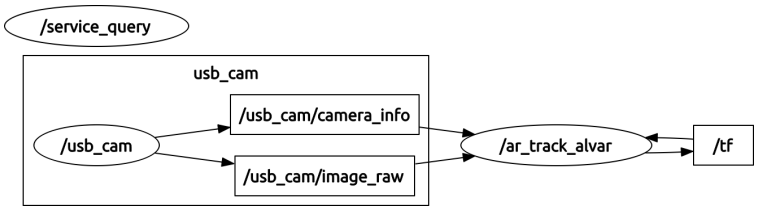

Figure 2: RQT Graph using AR Tags

Once the tracking package is installed, and the camera calibration is complete, you can run tracking by launching `webcam_track.launch`. You should see topics `/visualization_marker` and `/ar_pose_marker` being published. They are only updated when a marker is visible, so you will need to have a marker in the field of view of the camera to get messages.

Running `rqt_graph` at this point should produce something like Figure 2. As this graph shows, the tracking node also updates the `/tf` topic to have the positions of observed markers published in the TF Tree.

[2] `http://wiki.ros.org/camera_calibration`
[3] `http://wiki.ros.org/camera_calibration_parsers`

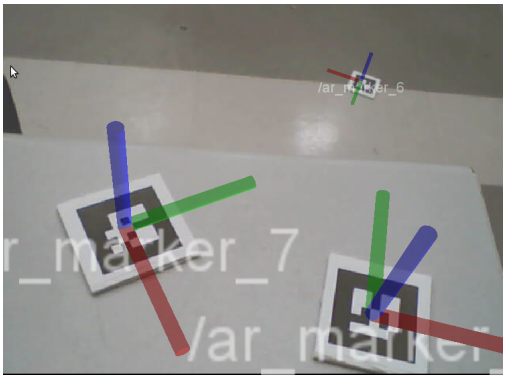

Figure 3: Tracking AR Tags with webcam

To get a sense of how this is all working, you can use RViz to overlay the tracked positions of markers with camera imagery. With the camera and tracking node running, start RViz with:

```
rosrun rviz rviz
```

From the Displays panel in RViz, add a "Camera" display. Set the Image Topic of the Camera Display to the appropriate topic (`/usb_cam/image_raw` for the starter project), and set the Global Options Fixed Frame to `usb_cam`. You should now see a separate docked window with the live feed of the webcam.

Finally, add a TF display to RViz. At this point, you should be able to hold up an AR Tag to the camera, and see coordinate axes superimposed on the image of the tag in the camera display. Figure 3 shows several of these axes on tags using the lab webcams. Making the marker scale smaller and disabling the Show Arrows option can make the display more readable. This information is also display in the 3D view of RViz, which will help you debug spatial relationships of markers for your project.

Alternatively, you can display the AR Tag positions in RViz by adding a Marker Display to RViz. This will draw colored boxes representing the AR Tags.

3 Baxter Tracking

The cameras on Baxter were calibrated at the factory, so we only need to run a tracking node for each camera that we want information from. You can copy the section of the previous launch file that starts the tracking node, setting the `cam_image_topic`, `cam_info_topic`, and `output_frame` parameters appropriately. Adding the Robot Model and then marker or TF data to RViz a great way to see how the markers are being identified around Baxter.

Some important notes that I found were necessary to get this working:

- The computer connected to Baxter is running ROS Hydro. You need to check out the right branch of the tracking repo with `git checkout hydro-devel`.
- You will need to have a unique name for each tracking node.
- So far I have only got tracking to work work at 1280x800 resolution. See the Baxter Camera Control documentation [4] for information on starting and setting the cameras.
- Use the `_axis` frames for the hand camera output frames.
- The head camera orientation seems a bit wonky, stay tuned to see if we figure out why this is.

[4] `https://github.com/RethinkRobotics/sdk-docs/wiki/Camera-Control-Example`

- When a tag is visible from multiple cameras, the visualizer will constantly display only the last published information. This will make the display appear jumpy. If you want a single pose estimate of a tag from multiple views, you may have to use the `/ar_pose_marker` topic relative to each camera frame and average them yourself.