1. How to start an Oculus project in Unity: <https://developer.oculus.com/documentation/unity/unity-gs-overview/>
   1. Alternative tutorial: <https://skarredghost.com/2019/06/08/how-get-started-oculus-quest-development-unity/>
   2. Important points:
      1. Download both the XR Plugin Management and Oculus VR Integration from the Asset Store
      2. Switch the build settings to Android, then Refresh the dropdown next to “Run Device” and click on the Quest 2 when it shows up (after you’ve connected it to your computer). There may be a build speed reason to change the texture compression option, but I’ve left it at the default.
      3. Project settings should be Android --> Oculus (only on Android, not on Windows)
   3. I originally started with the following packages from the Package Manager (Windows -> Package Manager): XR Plugin Management, OpenXR Plugin. In order to operate any GUI stuff, you’ll also need to add the XR Interaction Toolkit and possibly the Universal RP (Render Pipeline). I followed this tutorial to get the GUI stuff to work: https://learn.unity.com/course/create-with-vr?uv=2020.3
2. Using the Oculus:
   1. Connecting to your computer
      1. Use a data-carrying USB or USB-C cable (longer is more convenient)
      2. When you put on your headset, you’ll get a prompt asking you to confirm data transfer and then another prompt asking to allow your computer’s specific RSA fingerprint. You need to accept both. If the second prompt doesn’t appear, unplug and replug it in
      3. I have not had good luck asking the Oculus to remember my computer. It works better if I ask for the dialog boxes to reappear every time.
      4. I also haven’t had good luck enabling the Oculus Link connection. It should be irrelevant, but things seem to go better when I tell it not to try connecting.
   2. Oculus software for development:
      1. Oculus app (stupid, you don’t use it except to buy apps, but they insist)
      2. Oculus Developer Hub
         1. Once you’ve connected your headset, this app lets you keep your headset on even when you’re not wearing it, and remove the guardian visualization, both of which may be useful during debug. Go to the “Device Manager” tab and once you’ve connected, uncheck the “Proximity Sensor” and “Guardian” checkboxes
         2. Theoretically, it also allows you to download an .apk directly to your headset. I’ve had better luck with SideQuest.
   3. Downloading an already-existing .apk onto your headset using SideQuest
      1. Unity will let you launch a new apk straight from the editor by choosing “build + run” under your Build Settings, but once your APK is built, you don’t need to rebuild it in order to launch it on your Oculus. If it’s already downloaded, go to your list of existing apps. The default is to display only Unity apps, so click on the dropdown in the upper right corner from inside your headset and it should give you other options. Scroll all the way down to “Unknown Sources” and then you should be able to see your app. It might give you a warning about unverified sources; you built this yourself so just skip it.
      2. This allows you to play or restart an apk you’ve already installed. If you have the .apk but have not loaded it to your Oculus for some reason, open SideQuest and click the “upload” icon (it’s closer to the middle of the bar of icons on the top right). It’ll let you know if there have been any errors in uploading or if your Oculus can’t be found and you need to unplug and click through some dialogue boxes again.
3. Import a URDF:
   1. First, you need this in Unity: <https://github.com/Unity-Technologies/URDF-Importer>
   2. Create the URDF on a Linux machine:
      1. ROS command to create URDF from a .xacro in the Melodic distribution:

**“rosrun xacro xacro [name.xacro] > [name.urdf]”**

* + 1. You can use “check\_urdf” as a command afterwards to make sure it ran
    2. Biggest problem I’ve found if it doesn’t run: $ROS\_PACKAGE\_PATH doesn’t include the directory with your xacro file. (Some of the .xacro files have “find <pkg> in the first handful of lines; if the <pkg> path isn’t in the $ROS\_PACKAGE\_PATH, add it
       1. **export ROS\_PACKAGE\_PATH=$ROS\_PACKAGE\_PATH:/home/<usr>/ <path\_to\_your\_urdf\_directory>**
    3. Second biggest problem: not using the correct xacro file. Some are subfiles referenced by others; you want the top-most one.
  1. Transfer the URDF onto a Windows machine to import into a Unity project: Copy the entire folder into Assets/URDF/ in your Unity project--not just the URDF; you need meshes too, and you need to maintain the folder hierarchy that’s described in your .urdf file. Then right-click on the urdf you just made and click on “Import Robot from Selected URDF File” (which should be available once you finish importing the URDF-Importer and following the instructions in its README).

Graphical user interface, text, application

Description automatically generated

Possible issues:

* + 1. The root folder isn’t always selected correctly. Reroute it to the folder above your project folder (i.e. “Assets/URDF/” instead of “Assets/URDF/anki\_description/meshes/\*dae”) and things should start importing fine.
    2. NB: max links = 63 (I think)
    3. I’ve gotten a world AABB error when the URDF is initialized too far from the world origin. Not 100% sure why this might be (“AABB” is a super descriptive error message), but come back to that/keep it in mind.
    4. You probably want to make your root link immovable in space so it doesn’t fall with gravity.

1. Control a URDF
   1. The built-in Controller.cs script can be enabled after importing. Here are the recommended parameters from the online tutorial (I modified speed because I’m impatient).



* 1. There are multiple versions of the Controller.cs file. I create an empty Child under my robot URDF in the SampleScene hierarchy and label it “Controller,” then disable the built-in Controller.cs and “Add Component” to this new Child. The current recommended option is “ControllerFullExploration.cs”:
     1. Controller.cs: default, auto-generated, permits control of the URDF joint by joint, using keyboard keys
     2. ControllerWithOculus.cs: modified script controls URDF joint by joint, using the Oculus joystick and buttons
     3. ControllerFromLogFile.cs: reads in a csv file generated by MoveIt (titled “corrected\_positions.csv” and moves the URDF to follow the trajectory thus documented. For information on how to use MoveIt to generate csv files, see **Section VI** below.
     4. ControllerFullExploration.cs: Instead of using a pre-planned MoveIt path to generate moment-by-moment full-body poses, target joint angles can be set and the robot can “drive” to them. In order to span a reasonable portion of the URDF workspace, this script generates a sequence of random joint angle combinations based on a random number seed, subsamples the list to a trajectory that can fit within 5 minutes, and then drives the robot URDF through that sequence of poses. This script also has modifications to allow it to take turns with other URDFs, so that the different robots are operated sequentially and their associated information is saved into different .csv files.
  2. Note: if anything breaks in your project, you’ll likely get crazy random motion from the robot. Backtrack and try again, or check to make sure all your GameObject references are populated and valid. I’ve had this behavior when I tried to call or access a Game Object in the project that was not available or active—even when it wasn’t the robot Game Object, but a TextMeshPro Debug Report or something. Other possible reasons for crazy motion: excessive forces, stiffness, or inappropriate damping levels on one or more URDF joints.
  3. Rotating your URDF for optimal viewing: This seems like a straightforward thing, but if you check the “Immovable” box under the Articulation Body component in one of your links (which is recommended, to keep your robot poised in midair instead of falling through the floor), then whatever your starting rotation is is what you’re stuck with. If you choose to rotate a joint distal to the immovable one, you’re likely in the realm of the joints you manually control, and will be stuck with setting “targets” in your X drive in order to reach the desired position. Setting the Force Limits, Stiffness and Damping to get to that value in a reasonable way without overshooting or getting crazy motion from the robot is a chore. Instead, I recommend setting the Y rotation of a base link (earlier than the immovable one) to be 180 degrees so that it faces away. If you want to have a robot that faces the viewer instead of away from the viewer (mirror image instead of facing the same way), this is the original position (without the 180 degree rotation) but you’ll probably have to reinstantiate the robot in order to get it. I could not manually rotate the robot during Play/Edit mode.

1. Recording URDF and User Poses
   1. The PosPotRecorder.cs file can be inserted as many times as necessary into the Hierarchy within the Scene. (I attach it underneath every HandAnchor or EndEffector that I might want to record pose info for, but it has a public input field where you can drag the Transform or GameObject of your choice, so all the scripts could technically live under one GameObject if you wanted). It will record the Transform information (position and rotation about X/Y/Z axes) into a csv which is labeled according to the GameObject name.
      1. Note that the Transform information will be stored as a weird mixture of Euler angle and quaternion: Unity stores Transform data as quaternions in radians, but handles translations and rotations along the X/Y/Z axes and by degrees. Since MoveIt also uses radians, the Controller scripts are designed to transform inputs from the csv files by a factor of 180/pi in order to create the desired range of motion. However, this may be resulting in discontinuities in logged data. Double check to make sure you understand the nature of your input and output transforms.
   2. The JointRecorder.cs file generates files similarly to the PosPotRecorder, but instead of recording position and orientation information, it stores the joint angles for each of the numerically-labeled joints in the URDF.
      1. Note that this script runs continuously, and that the size of the target file is not based on the input controlpositions.csv file and cannot be known from the beginning. ~~Therefore, instead of storing all joint values into a variable that is then written into a csv at the end of a run, the <urdf\_name>\_JointMotion\_<run\_num>.csv file is opened immediately upon launch and values are stored in it throughout the run. This means that at the end of your data collection, you will have one opened but empty JointMotion.csv file that needs to be deleted. It also means that if you relaunch the apk freshly for any testing purposes, it will immediately overwrite the <urdf>\_JointMotion\_1.csv file if it exists. There are probably tidier ways of writing this but for now, just be careful.~~
   3. I had intended to transmit info to/from the Oculus Quest via server, as Jack did for the RoboVR Hackathon in October. Instead, I’m doing asynchronous transfer of info via .csv log files. If you need real-time info transfer, there is theoretically a ROS# communication pathway you can set up with Unity (see “Simulating Robots in Unity” from a Google search) but if that doesn’t work, I know we’ve at least got Jack’s method functioning once.

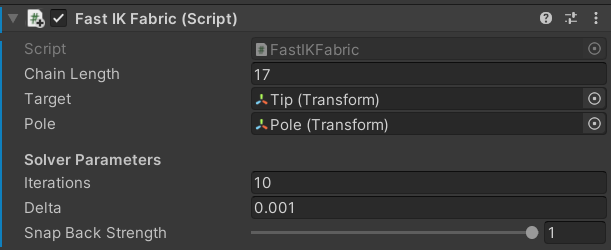
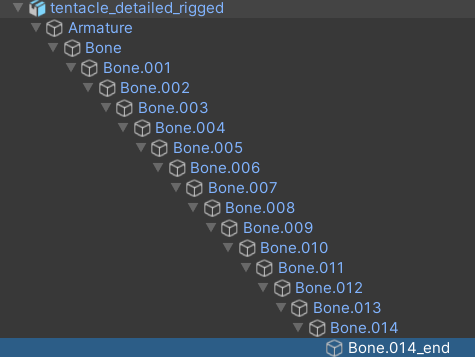
To transmit info to/from a server using JSONs:

* + 1. Current server setup is done via Go, modified from Jack Kolb’s starter script (see ServerScripts folder. Must be on the same wifi network as the server workstation in order to access.)

**> go run . (or go run main.go)**

* + 1. Jack has a tutorial on using python to generate servers, with a link at the bottom for how to include JSONS: (<https://kolb.dev/flask>)

1. Importing and Controlling the Tentacle-bot: The tentacle robot is in imitation of a soft-robot, and so it is controlled differently than a URDF with rigid links. There is a 3DOF and a 6DOF version of the prefab in the Assets folder, and each 3DOF segment has a tip and a base connected by 14 skeleton bones. Control of the tentacle will need to be done by controlling the location of the tip and its “pole” in 3D space, according to whatever control scheme you find reasonable, and then adding the Fast IK Fabric component (with its appropriate tip, pole, and chain length parameters—chain length = 17 for each 14-bone tentacle segment) to the end bone from the FastIK asset you can download from Unity. Proper settings for FastIKFabric.cs are shown in this screenshot:



Make the tip and pole spheres invisible after you’ve verified that everything’s working.

* 1. I coded the tip (or “target”) and pole controls to follow [cartesian/spherical/curvature…?] coordinates with a distance limit, and a restriction that the pole always makes the tentacle bend upward, instead of horizontally (like a willow tree instead of a snake). This is the purpose of the TentacleController script, which means it functions differently than the normal Controller scripts attached to every other robot prefab. Note that the Controller scripts are attached to each prefab, so you shouldn’t need to attentively switch any particular type of controller on and off. The PosRotRecorder script is also placed at report the location of the tip and pole and base (of each segment) instead of at the location of the different links or end-effectors.
  2. In order to make the tentacle follow the tip/target and the pole, don’t forget that you need *both* the TentacleController (at the tentacle root) and the Fast IK Fabric component (at the tip—or at least, the tip of each segment).
  3. Note that you might need to add something to smooth the transition between the bottom and the base. Right now I have the base and top separate for ease of control, but I’m not sure if the parenting will work right between the two so that they sit on top of each other sensibly. Trying to have all of them combined has issues with “loop in parent” errors in Blender so far. (Test with moving base and see if top comes along for the ride first. Answer: it does, but there is a lack of deformation at that juncture. Maybe put a ball there to mark the junction point and cover up the slop, like )

1. Communication between the Oculus and MoveIt is done via .csv logfiles in the “persistent data path directory” (specific paths for deployment on Oculus and testing on Windows are mentioned below). Properly-labeled CSV files will be read by the Oculus from that directory, and labeled CSV files with joint variables, hand pose and end-effector pose are saved into that directory as well.
   1. Persistent data path for Windows (my machine only): C:\Users\jmoln\AppData\LocalLow\DefaultCompany\NonAnthroHandsUserStudy
   2. Persistent data path for Oculus:

This PC\Quest 2\Internal shared storage\Android\data\com.DefaultCompany.NonAnthroHandsUserStudy\files

* 1. NOTE: The files that should be loaded onto the Oculus (besides the APK) for the app to work are in a consolidated directory, ~/Documents/Projects/NonAnthroHands\_User\_Study/Files\ needed\ on\ Oculus/

1. Using MoveIt to identify joint limits, generate sample trajectories, and generate IK-based joint commands when end-effector movement is pre-specified:
   1. Important note: After you source the setup.bash file in the melodic moveit workspace *(which you should do),* run this line of code manually right before you launch the moveit setup assistant:

**> export ROS\_PACKAGE\_PATH=$ROS\_PACKAGE\_PATH:/home/<usr>/<path\_to\_urdf\_directory>**

**> roslaunch moveit\_setup\_assistant setup\_assistant.launch**Sourcing the setup.bash file will overwrite the $ROS\_PACKAGE\_PATH so that it cannot see your urdf file, even if you successfully exported/created it earlier. DON’T FORGET THIS CRUCIAL STEP.

* 1. You can follow the tutorial to set up your config file here: <https://docs.ros.org/en/melodic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html> Don’t forget to save your configuration in a catkin workspace, under a folder you can reference later.

- Key command to launch the set-up assistant to create the config file:

**> roslaunch moveit\_setup\_assistant setup\_assistant.launch**

* + 1. Note that you will want to look at the default start pose in Unity (after you press “play” and the joints settle) and put the joint angles into one of the poses you save in your config file. This will allow you to make plans that start from the URDF’s natural start pose, rather than having an abrupt jump at the beginning (position control) or getting the robot stuck against one of its joint limits (velocity control). Doing this also allowed me to both plan and execute your chosen path (originally, Execution would fail because the robot start state wasn’t considered valid).
  1. Command that opens MoveIt for generating paths between poses:

**> roslaunch kinova\_moveit\_config demo.launch rvis\_tutorial:=true***(OR j2s6s300\_config OR whatever the name of your config folder is.*

*Do not include the backslash afterwards. It will not run, even with the proper $ROS\_PACKAGE\_PATH)*I found that the ROS\_PACKAGE\_PATH updates that I had made earlier weren’t enough to get RViz to open with MoveIt. It threw errors until I added the path down to my launch folder, along with all the subpaths down to that level.

* 1. Once you have your chosen start (default) and end pose, press “plan” in the “MotionPlanning/Planning” tab at the bottom left of the RViz screen. You can export the positions of the joints for your planned path by saving the joint positions to a “test.bag” rosbag and then a “test.yaml” file with the following commands:   
     **> rosbag record --duration=10 --output-name=test.bag /move\_group/display\_planned\_path**

***> rostopic echo -b bag\_name.bag -p /move\_group/display\_planned\_path > test.csv***

***(above is deprecated; does not capture all information)***  
**> time ros\_readbagfile test.bag /move\_group/display\_planned\_path | tee test.yaml**

**> awk '/positions/ {print}' test.yaml > positions.txt**

(above requires installation of ros\_readbagfile.py; see <http://wiki.ros.org/ROS/Tutorials/reading%20msgs%20from%20a%20bag%20file>)

* + 1. *Cool things you can manipulate (and probably should!):* You can get MoveIt to do way cooler and more useful stuff than just what the default planner provides (interpolates smoothly between all joint start and end angles simultaneously). Get a little more control over the gestures by clicking the “Use Cartesian Path” checkbox in the Planning tab (if such a solution exists—this will attempt to make the end-effector move in a straight line instead of swinging in arcs the way the default planner will do), and by adding Scene Objects in the appropriate tab. You can change the dimensions of the boxes when they’re first created, which allows you to set planes that constrain the planner’s motion to zones that you want (if, for instance, you want to keep the robot moving through a human-like workspace).
    2. Note that once you add constraints, the planner no longer defaults to the same planned trajectory every time. Monitor the animation as you store it to the test.bag file; only transfer and save if it’s the motion you want.
  1. Convert positions.txt into a csv file, with brackets and headers removed. (The PositionFileTxtToCSV” python notebook will do this for you.) Save as “<urdf\_name>\_corrected\_positions\_<num>.csv”. The controller code in Unity (ControllerFromLogFile.cs) will import the sequence of joint positions and transmit them to the robot every 0.2 seconds. These joint positions can be manually visualized by dragging the “target” slider under the xDrive of each individual link while running the app in Unity’s Editor.
     1. File editing: Check to make sure that RViz didn’t save multiple trajectories, just the one you want. (If there are more than one back to back, it may be that the “current” position of the robot in RViz is different than your starting position. That would make the second motion the one you want. Occasionally it is the last motion, instead. You can watch the trajectories in Unity to confirm that they’re the ones you want.)
     2. Also do a test run in Unity’s editor to make sure that the invisible end-of-line and end-of-document characters are correct. You’ll notice if the “record” and “play” buttons never re-enable, and the robot is frozen in its final position. To correct this, select all empty boxes in Excel and re-delete them.
     3. Use Unity to also check that the speed of your motion is desirable. Most of the time that it runs too quickly, it’s because it’s using the default planner (that smoothly interpolates between starting and ending joint angles) and has segmented the trajectories too sparsely. You can manually interpolate more finely in Excel. It’s kinda a pain, but I haven’t found a straightforward way of checking for this except manually.
  2. You can do the same with velocities if you want. The default URDF controller in Unity uses velocities, but even after scaling my velocities to be in deg/sec instead of rad/sec, I didn’t see much motion when I tried to upload “targetVelocity” values instead of “target” [position] values with my code.
  3. Move the “corrected\_positions.csv” file to *“C:\Users\<username>\AppData\LocalLow\DefaultCompany\NonAnthroHandsUserStudy”* if you need Unity to be able to read the file during Play, and place a copy in *“C:\Quest 2\Internal shared storage\Android\data\com.DefaultCompany.NonAnthroHandsUserStudy\files”* for the app to access when deployed on the Oculus
  4. Using MoveIt to create a trajectory that spans multiple waypoints:

**DEPRECATED:** If the trajectory is not based on end-effector position and thus does not require IK, it’s easier to systematically insert joint angle drive targets into Unity than to collect thousands of waypoints from MoveIt for a csv-driven URDF trajectory. Alternatively, you can make multiple trajectories and set the start point of one to be the end-point of another, and then sequence them manually.

* + 1. Don’t forget to export your up-to-date ROS\_PACKAGE\_PATH with the path to your config folder as well as your URDF. This should be the config folder where you saved all miscellaneous files after completing your URDF setup in the MoveIt SetUp Assistant.
    2. Launch Rviz in one terminal window:

**> roslaunch [kinova\_moveit]\_config demo.launch rvis\_tutorial:=true**

* + 1. Launch the trajectory planner in another:

**> rosrun [kinova\_moveit]\_config move\_group\_grid\_search.py**

* + - 1. You can change the joint angle combinations and order in the move\_group\_python\_interface.py, under the go\_to\_joint\_state() function. Comment out all other tutorial functions under the main() function. I found that three joint angles per dof (pi/3, pi, and 5pi/3) was few enough that even with 3^6 combinations, all motions can be run in a reasonable time frame, and the selected joint angles also avoid hitting joint limits near 360deg and 0deg. Grab the output angles by doing the usual rosbag command on the /joint\_states topic.

1. FastIK:
   1. Installation is a pain. I followed the tutorial here: <http://docs.ros.org/en/melodic/api/moveit_tutorials/html/doc/ikfast/ikfast_tutorial.html> using the git package here (<https://github.com/crigroup/openrave-installation>) to install the OpenRave software package. (I think the tutorial has been updated in the last week so you can go straight to docker and it’ll take care of the installation for you)
   2. Download collada\_urdf:

> sudo apt-get install ros-melodic-collada-urdf

* 1. Once again, check your $ROS\_PACKAGE\_PATH variable to make sure it can find your urdfs before you turn them into .dae files with this command:

> rosrun collada\_urdf urdf\_to\_collada “$MYROBOT\_NAME”.urdf “$MYROBOT\_NAME”.dae

1. Dynamic time warping:
   1. I followed this tutorial, with its relevant github: <https://www.kdnuggets.com/2022/05/dynamic-time-warping-algorithm-time-series-explained.html> . Look for NAH\_DTW\_for\_JA.ipynb in the github. (I made two files, one for end-effector DTW alignment and one that includes alignment for joint positions. I realized that it’s better to make mappings to the joint angles, since that’s how Unity will control the robot, rather than generating end-effector poses that you have to put back into MoveIt (and which leave an unresolved DOF mismatch for some URDFs).)
   2. Note that if you have extra data at the front or back end, your DTW will have weird mappings there. It’s good practice to truncate all recordings that do not contain actual data.
   3. Note that if you install Conda in a Unix environment, it will break the paths you’ve made for all your ROS/MoveIt operations. To deactivate (activate) a Conda environment (it will be on by default—look for the *(base)* in front of your user name at the command line), type:

**> conda deactivate**

1. GPR: look for GPR\_for\_joint\_angles.ipynb in the github
   1. Note that Unity stores Transform rotations as quaternions, but controls them via Euler angle rotations. This is why the control script in the Unity requires a 180/pi factor to be multiplied against the same rotation values it just exported into the log file in order to animate the URDF. But it also means that Unity sometimes hiccups and outputs odd discontinuities in the rotation data that we need to resolve before doing GPR. They’re artifacts, not data. There is a function within the python code that will smooth them out.
   2. Also note that according to 2018-era literature (quoted from https://stackoverflow.com/questions/50185399/multiple-output-gaussian-process-regression-in-scikit-learn) , multiple-output GPRs really are equivalent (or should be) to multiple single-output GPRs, and thus for the sake of getting accurate std\_dev info, we should train GPRs for each joint angle separately.

Weird plugin bugs: <https://github.com/Unity-Technologies/Unity-Robotics-Hub/issues/215>

Motivation for doing multiple single-output GPRs instead of one attempted multiple-output GPR: <https://stackoverflow.com/questions/50185399/multiple-output-gaussian-process-regression-in-scikit-learn>