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 -->
      3. Project settings should be Android --> Oculus (only on Android, not on Windows)
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: <https://github.com/Unity-Technologies/URDF-Importer>
   1. 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
     1. The root folder isn’t always selected correctly. Reroute it to the folder above your project folder 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:
     1. Controller.cs: default, auto-generated, permits control using keyboard keys
     2. ControllerWithOculus.cs: modified script controls URDF using Oculus joystick and buttons
     3. ControllerFromLogFile.cs: reads in a csv file generated by MoveIt 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.
  2. Note: if anything breaks in your project, you’ll likely get crazy random motion from the robot. Backtrack and try again. 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 like a TextMeshPro Debug Report or something.

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. Using MoveIt to identify joint limits and generate sample trajectories:
   1. Important note: sourcing the setup.bash file in the melodic moveit workspace will overwrite the $ROS\_PACKAGE\_PATH so that it cannot see your urdf file. Even if you successfully exported/created it earlier. 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**

* 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**

* 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.py 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. Convert positions.txt into a csv file, with brackets and headers removed. Save as “corrected\_positions.csv”. The controller code in Unity (ControllerFromLogFile.cs) will import the sequence of positions and transmit them to the robot every 0.2 seconds. This can be manually visualized by dragging the “target” slider under the xDrive of each individual link while running the code in Unity.
  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:** 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

* + 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. 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 above). 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.
2. 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. 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 there are ugly discontinuities in the rotation data that we need to resolve before doing GPR (why? Didn’t you just say we were storing quaternions? Where do the discontinuities come from?).

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