v.1

PR2 with Xylophone

cl3295 Chia-Jung Lin, jy2653 Jingwei Yang

1. Project Overview

In this project, PR2 can
1) pick up stick (https://www.youtube.com/watch?v=qgwbS4EpdAU)
2) move head to look for xylophone
3) move to standby position to play xylophone
(https://www.youtube.com/watch?v=NA59Ga1QzZA)
4) play xylophone (https://www.youtube.com/watch?v=6mOdmOH7q-k)
5) vocalize xylophone sound in Gazebo

2. Implementation

2.1. Models

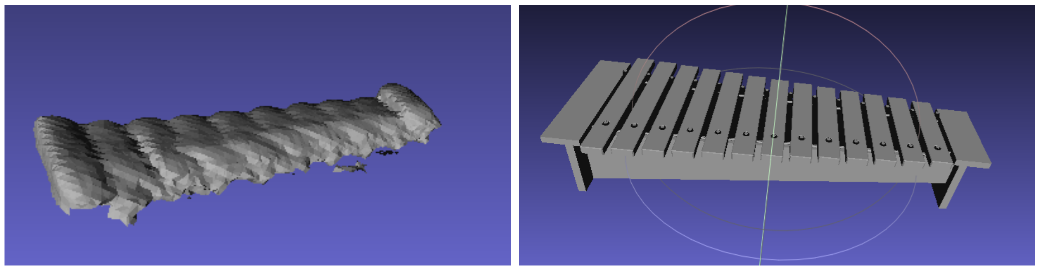


Fig.1: (a) mesh model captured with Kinect (b) 3D model downloaded from Blender community

The model we captured by Kinect was not easy to work with due to the uneven appearance. Thus, we downloaded 3D model from Blender community and usee it in all our experiments.

2.2. Pick Up Sticks

$ rosrun pickup_object pickup_stick.py

This step is not integrated in our final demonstration because of the unstableness state of holding the stick. Source code can be found in pickup_object package. The functions are implemented in both C++ and python. This part is mainly integrated over given sample codes[1]. The position and the orientation of the arm were hard-coded.

2.3. Vision Input

$ rosrun scene_segmenter scene_segmenter_node
$ rosrun scene_segmenter scene_client_node

2.3.1 Structure

The vision analysis function is implemented in scene_segmenter package.

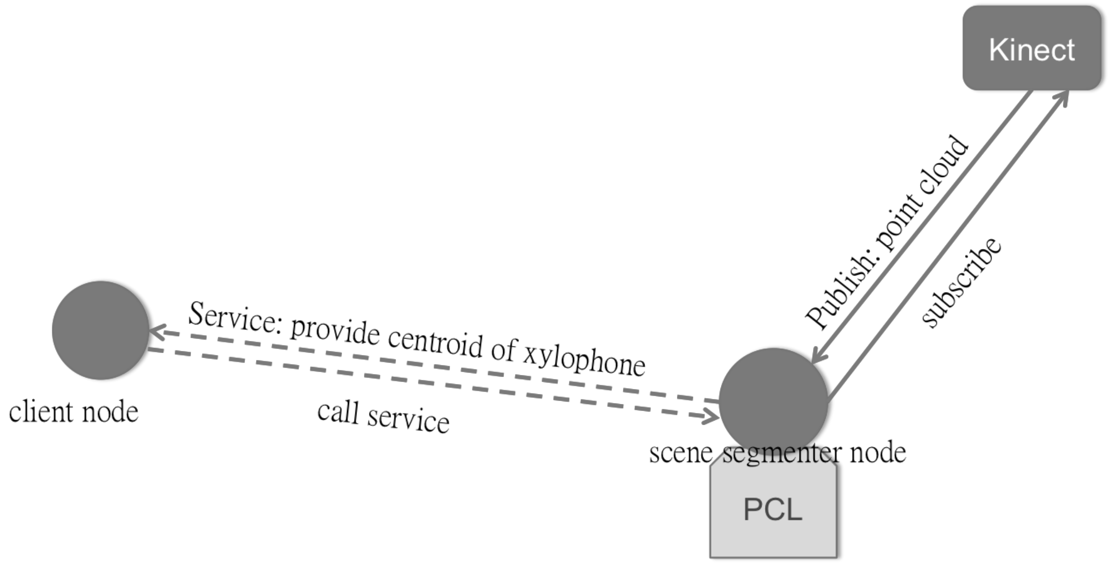


Fig. 2: Structure of scene_segmenter package

The structure can illustrated by Figure. 2. The scene_segmenter_node subscribes to kinect topic and constantly obtained the point cloud from Kinect. It offers a service called "get _xylophone_pose". When the client node calls this service, the scene_segmenter_node processes the point cloud and returns the centroid of the xylophone.

2.3.2 Point Cloud Analysis

We use the Point Cloud Library (PCL)[2] to help analyzing the point cloud. We use the ClusterExtracter class in a github project[3]. The extraction technique is common among robot vision processing. First, use RANSAC to extract the biggest plane in the point cloud and extract it. Second, cluster the remain point cloud. We get the list of clustered point cloud and only keep the biggest clustered object. Because we only put a xylophone on the table, the remaining biggest cluster would be the xylophone. Then, we can compute the centroid of the xylophone.

Because Kinect is using its own coordinate system, we have to do point cloud transformation to obtain a position that is relative to PR2 in world coordinate system.

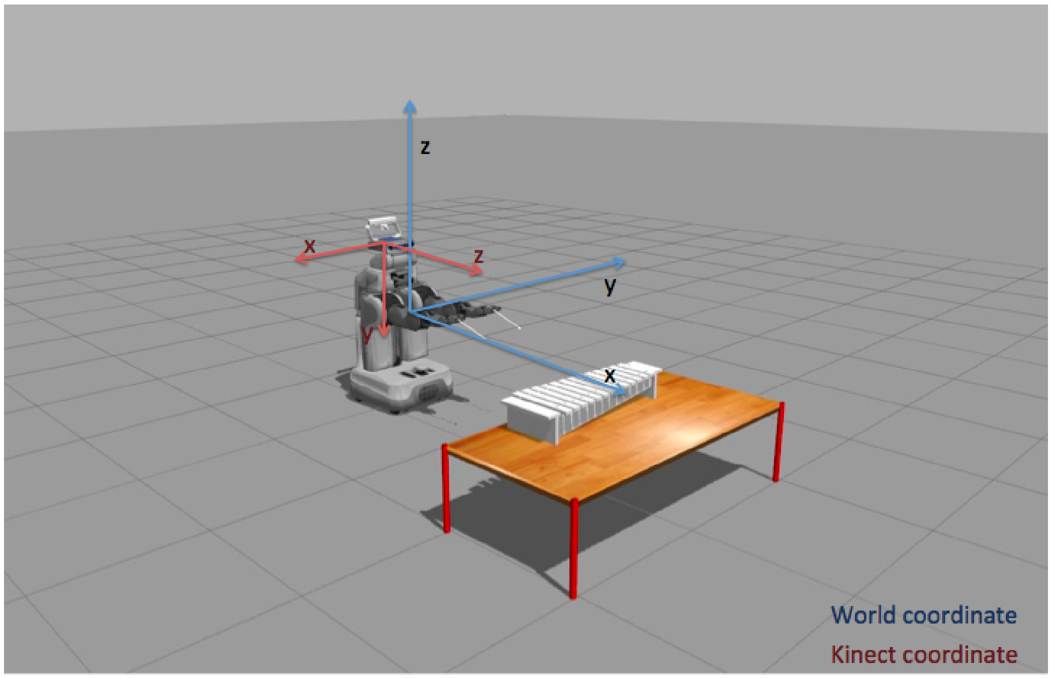


Fig 3: Two Coordinate Systems

Here we address some problem we met on point cloud transformation. The point cloud from Kinect is using different coordinate system, so we have to transform it into world coordinate to identify the correct location to head to. The transformation sample code on ROS official webpage did not work for us. However, another team in class can execute the same code segment successfully. We did not figure out what is causing the problem. However, we found other way around to solve this.

[Original Code Segment]

```
ros::Time now = ros::Time::now();
listener.waitForTransform("/base_link", current_cloud.header.frame_id, now, ros::Duration(3.0));
listener.transformPointCloud("/base_link",ptcld1,convertedPtcld1);
```

[Error Message]

Lookup would require interpolation into the past. Requested time [x] but the earliest data is at time [y]. When looking up transform from frame [A] to frame [B].

The function *transformPointCloud* in class *TransformListener* is implemented as:

```
void TransformListener::transformPointCloud(const std::string & target_frame, const
sensor_msgs::PointCloud & cloudIn, sensor_msgs::PointCloud & cloudOut) const
{
        StampedTransform transform;
        lookupTransform(target_frame, cloudIn.header.frame_id, cloudIn.header.stamp,
transform);
        transformPointCloud(target_frame, transform, cloudIn.header.stamp, cloudIn,
cloudOut);
}
```

The function looks up transformation at time header.stamp, but somehow in our experiment it could not find data at that time frame. Thus, we set the stamp of the cloud to 2 seconds later:

```
ptcld1.header.stamp = current_cloud.header + ros::Duration(2.0);
listener.transformPointCloud("/base_link", ptcld1,convertedPtcld1);
```

By manually delaying the transform time to now + ros::Duration(2.0), the transformation can be found.

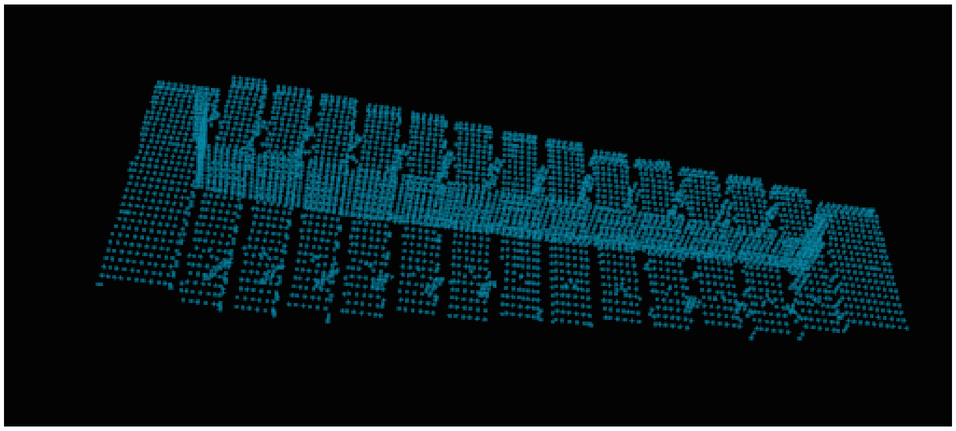
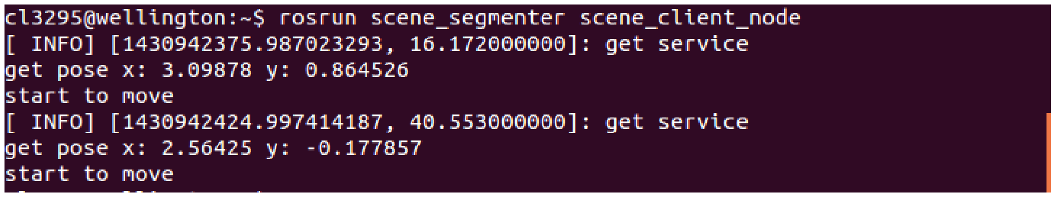


Fig. 4: Point Cloud of Xylophone

2.3.3 Move!

After the client node gets the position, it decides the direction and distance that PR2 should move. We predefine the correct 'relative distance' to the xylophone. PR2 should arrive at the this correct position to play the xylophone. After several experiments, we found that when the distance is large, the estimation of position of the xylophone is less accurate. Therefore, we run two iterations and let PR2 check and adjust its direction in the second iteration. Our setting is confined to that the xylophone should be placed at the side of the table closer to the PR2, so PR2's final position will have the same orientation (facing out). PR2 gets the relative position of the xylophone, and moves in y direction first, and then x direction.



```
cl3295@wellington:~$ rosrun scene_segmenter scene_client_node
[ INFO] [1430942375.987023293, 16.172000000]: get service
get pose x: 3.09878 y: 0.864526
start to move
[ INFO] [1430942424.997414187, 40.553000000]: get service
get pose x: 2.56425 y: -0.177857
start to move
```

Fig. 5: Terminal Output of Two-Staged Moving

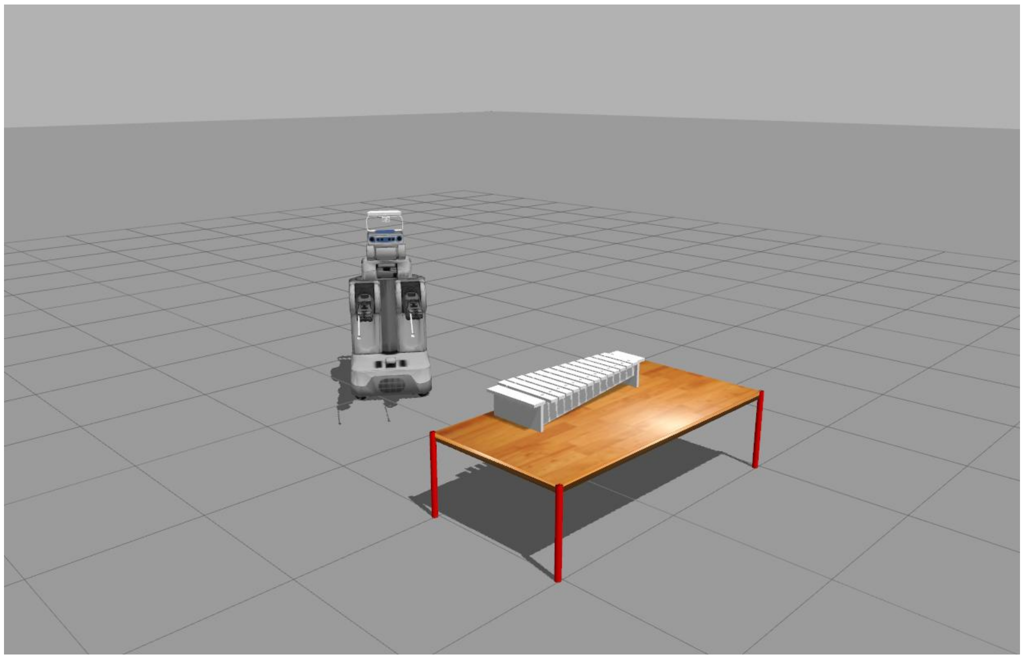


Fig.6: PR2 moving to destination

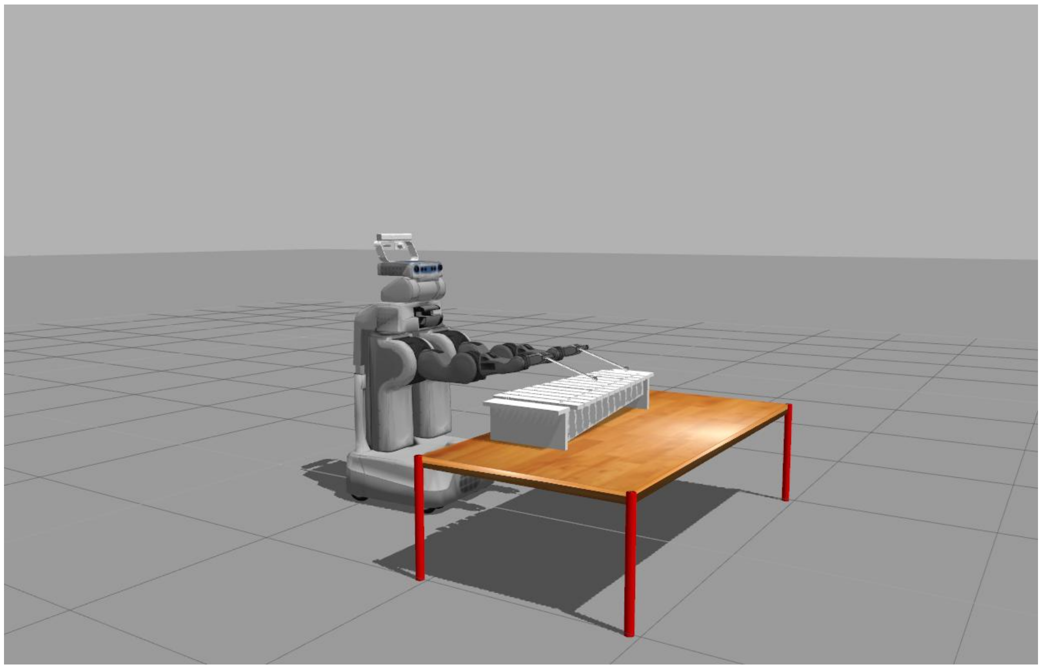


Fig. 7: PR2 at correct destination

2.3.4 Experiment

In this part, we run 10 times of the moving procedure, and observe the final position of PR2. The start position of PR2 is randomly assigned. For this set of experiment, the ideal final position is x= 0.9855 and y = 0.00.

#	x	y	yaw
1	1.1877	-0.0312	0.045
2	0.9410	0.0065	0.053
3	1.0247	0.0809	-0.020
4	1.0858	-0.0616	0.010
5	1.1215	0.0370	-0.039
6	1.1529	0.0842	0.015
7	1.1807	0.0120	0.009
8	1.1748	0.0528	-0.017
9	1.1628	-0.1443	0.066
10	1.1815	0.0501	0.113
Avg	1.1213	0.0086	0.0315
Std	0.0776	0.0673	0.0474

The standard deviation for x is near 0.08, so PR2 usually gets too forward and its gripper gets stuck by the xylophone. Lifting its hand higher might be able to relieve this problem. Even we do two iterations, PR2 does not arrive at the exact same position every time. Sometimes the rotation angle is not exactly 90 degrees, so when PR2 turns and then turns back to front, it is not actually facing front ('yaw' attribute not equals to 0). Because of the same reason, PR2 sometimes moves slightly astray, and then arrive at the wrong destination. This problem can be relieved by adding more iterations. However, we think there is some limit and because of this randomness, PR2 fails to play the correct node in the last step.

2.4. Hold Stick

When PR2 hit the xylophone in the Gazebo, the stick would slip away from PR2's hands. We have tried to improve the fraction factor between the stick and finger, but it still does not work.

Brief Solution:

In reality, we could bind the stick with PR2's finger. The force of striking the key is not strong enough to drive the stick away from PR2's hands. After consulting with our TA, we decide to create a joint to bind the stick and PR2's finger together.

Challenge 1: The PR2 description folder is under the management of super user of CLIC. To create a new fixed joint means we have to edit on the description files of PR2. However the URDF and related mesh files are in the folder "/opt/ros/hydro/share/pr2_description", which is managed by the super user. We have no right to edit on it.

Brief Solution: When we type the roslaunch in the command line, the setup would firstly search the related resources in the current working directory. So we could have our own version PR2 description in current working directory.

Challenge 2: The CS student account has very limited storage space. The PR2 description folder would take up 64MB in total. We have only 50MB left and we need to have extra storage for recording the log of running ROS.

We solve this problem by using the concept of "symlink" (soft link) from Operating System. We find out that the sub-folder "meshes" actually take 55MB in total. Actually, the mesh files are the models of each link in the PR2, they should be kept stable. All we need is to change a file called "urdf/gripper_VO", which only take up 36KB.

Brief Solution: Create a folder called pr2_dscription under current working directory, "`~/robot-test/src/pr2_description`". Create a symlink to meshes folder and copy other directories under the current pr2_description folder. And then we can edit on the gripper_VO file.

Command: `ln -s /opt/ros/hydro/share/pr2_description/meshes    meshes`

Challenge 3: Which type of joint to create? And we should attach the stick to which existing link of PR2.

Brief Solution: To figure out the solution, we have explored many tutorials to understand the links and joints of PR2's arm. We have learned that the joints are classified into two types: revolute and fixed. We need to create a fixed joint to bind the `{side}_gripper_l_finger_tip_link` with `{side}_stick_link`.

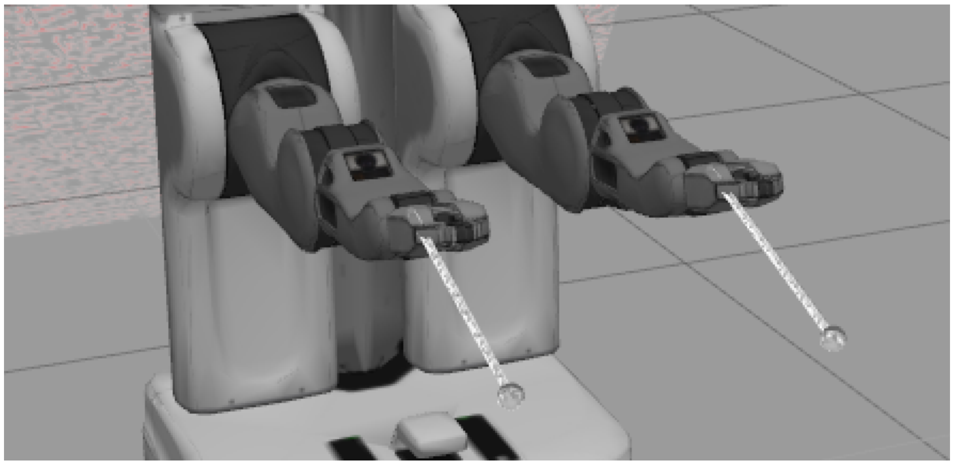


Fig. 8: Stick link attached to a fixed joint of PR2

2.5. Hit Xylophone

$ rosrun hit_xylophone process_notes.py

Problem 5.1: We need to have a fine-grained motion plan for striking each note. Usually, we use "MoveIt!" to achieve the goal position of a control group. Through this way, we just need to specify the target position of a control group. However, this functionality face two challenges in our project.

Challenge 1: The resultant motion plan of "MoveIt!" is quite random. Our PR2 is assigned to play with the inputting notes, thus the interval between two consecutive notes should be almost the same. If we just use "MoveIt!" to randomly generate the striking motion plan, the quality of playing could not be guaranteed.

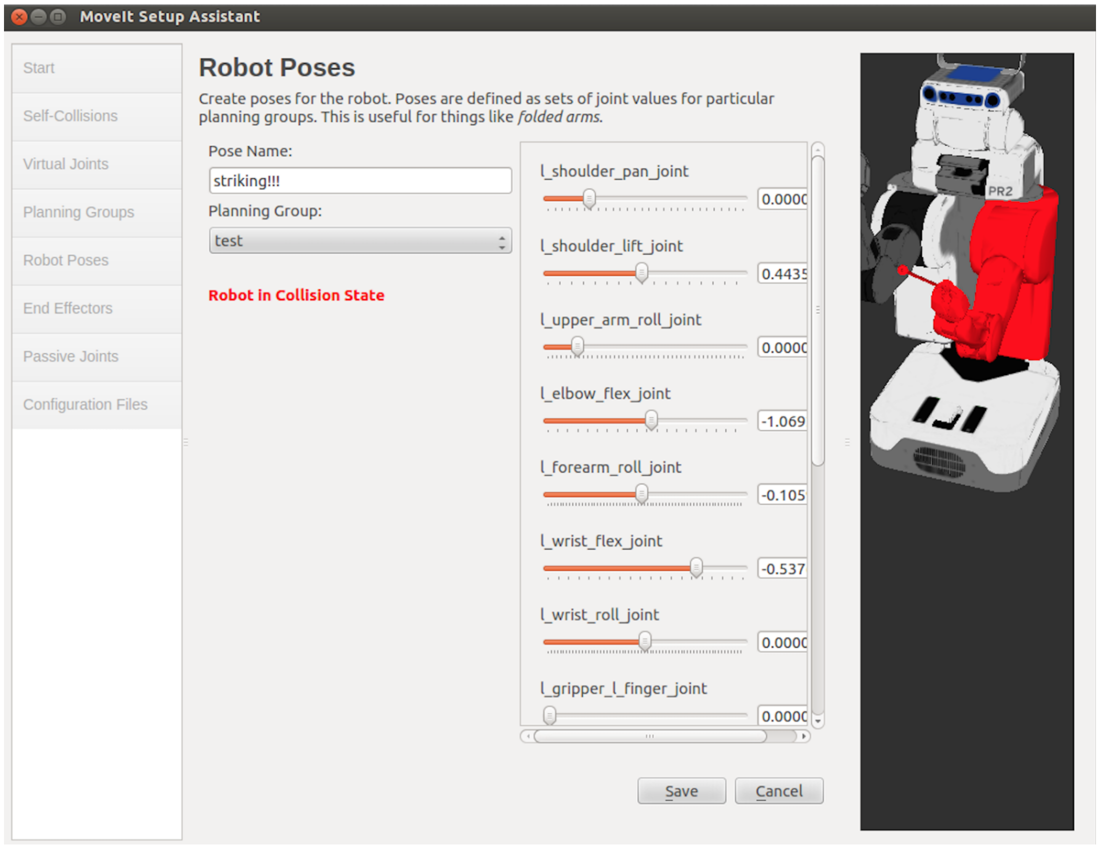
Challenge 2: The control unit of "MoveIt!" is arm group under our project. The "MoveIt" could not meet our requirement of fine-grained control over each joints, thus we could not achieve the goal of striking different keys. We need to specify our own motion plan for playing xylophone in fine grained scale.

Challenge 3 : The joints of arms on PR2 is more complicated than those of other parts. To specify the motion of striking a key, we need to collaboratively adjust many joints at the same time.

Challenge 4: Even we can figure out the group of joints we need to adjust with, the orientation of each joint should also be precisely manipulated, so as to strike each tiny key precisely.

Brief Solution:

After days of exploring the joints and valid orientation angles, we find it's quite difficult to adjust on those joints. But, we finally figure out a very useful tool called "PR2/Setup Assistant". With the help of this tool, we could try to create a planning group with the links and joints from PR2 description. We could intuitively adjust the angle of those joints, and thus to make fine adjustment over each joints to achieve our striking goal. In our motion planning of striking key, we need to adjust shoulder_pan_joint, *shoulder_lift_joint, upper_arm_roll_joint, elbow_flex_joint, forearm_roll_joint, wrist_flex_join and wrist_roll_joint*, even for one single playing motion.



Robot Poses

Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like *folded arms*.

Pose Name: striking!!!

Planning Group: test

Robot in Collision State

l_shoulder_pan_joint	0.0000
l_shoulder_lift_joint	0.4435
l_upper_arm_roll_joint	0.0000
l_elbow_flex_joint	-1.069
l_forearm_roll_joint	-0.105
l_wrist_flex_joint	-0.537
l_wrist_roll_joint	0.0000
l_gripper_l_finger_joint	0.0000

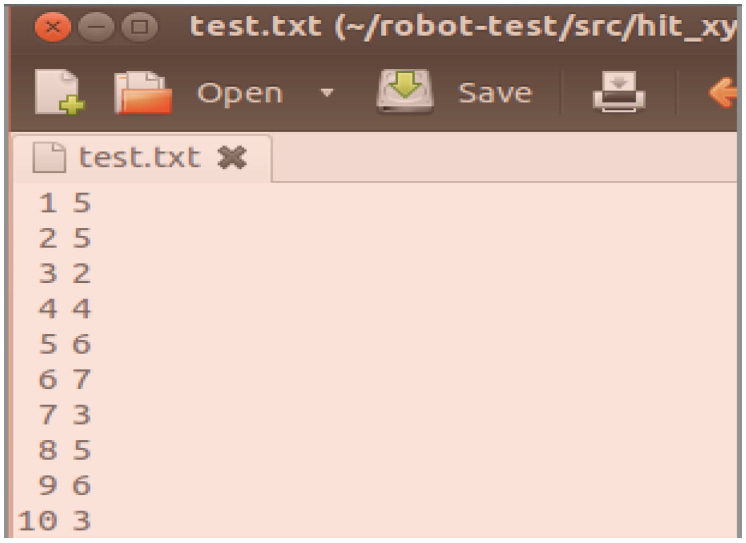
Fig. 9: Our planning group of PR2

Problem 5.2: It's easy to hard code the movement of PR2's arms, thus PR2 could only plays one song. Apparently, our group's PR2 should not behave in that way. The principle of generalization is at the very center of our design and implementation. We want to PR2 to play any song based on the inputting note files.

Challenge 1: Since we could not predict the order and length of our input, we could not implement the activity of playing xylophone as a single function. And we should have a service node to push the new message for movement.

Brief Solution:

We design a set of motion associated with playing each note. Once we read a new note from the file, we retrieve related motion plan and execute it. Thus, our PR2 could play with any legal inputs.



```
1 5
2 5
3 2
4 4
5 6
6 7
7 3
8 5
9 6
10 3
```

Fig. 10: The inputting file of notes

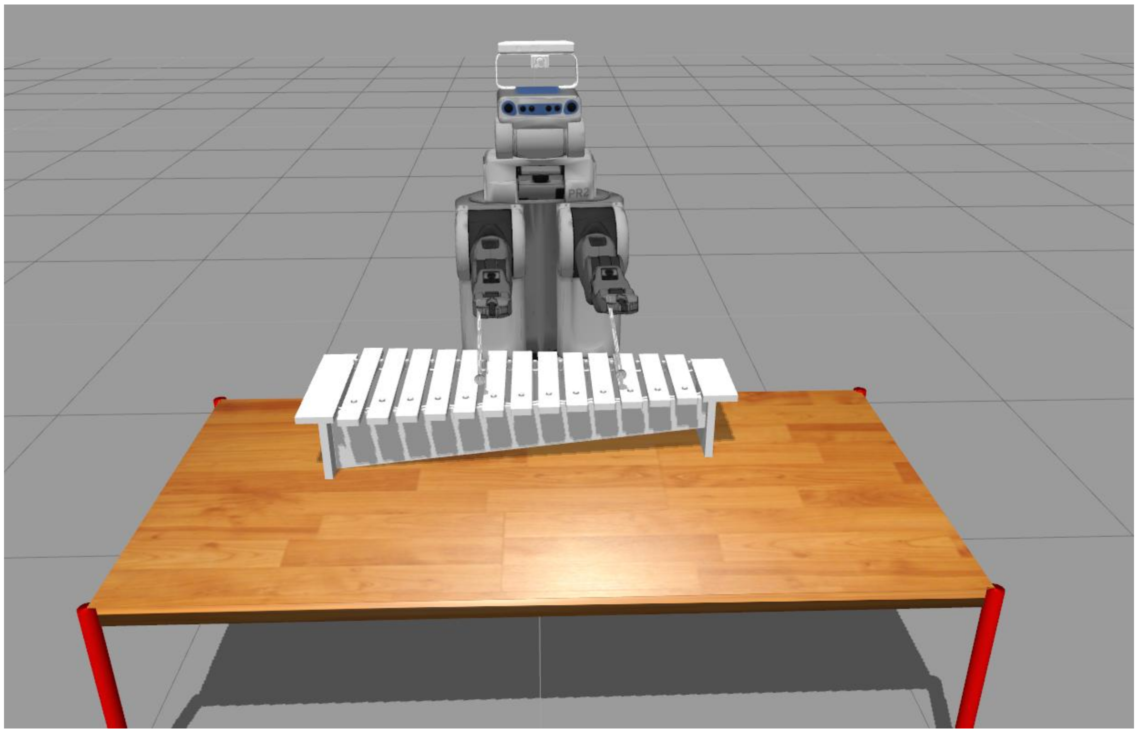


Fig. 11: PR2 playing xylophone

2.6. Vocalize the playing

Problem:

Since it's not convenient to access real PR2, we want to vocalize the playing of PR2 in our Gazebo. When PR2 finishes the motion of 1striking a key, our system should be able to respond with the sound of the note. This addition would definitely empower the simulation effects of project. To explore out the way of vocalizing the playing in Gazebo, we have tried many ways, and find out a very effective one.

Challenge 1: How to make the computers in CLIC play sounds along with the playing of PR2? There is no vocalization library in CLIC machine, through which we can directly control the sound card of computer. I have tried to use a library called PortAudio to interact with the sound card, and use python module PyAudio to control the playing. However, the experience is quite painful. Since we have no right to edit global share folder, we have to create a virtual environment for the Python module PyAudio. And we also have to make the PortAudio is callable in our virtual environment, which means it should be globally accessible. I have failed to find out the C library to also provide the mechanism of virtual environment, thus the PortAudio is not accessible globally.

Brief Solution: There is a ROS package called "sound_play", which is designed for robots to speak out. We decide to build our sounds playing mechanism based on this package. However, the computers in CLIC only pre-install the simple version of ROS, which does not include the "sound_play" package. To make things worse, the computers in CLIC also miss some dependency packages for it. To fix this problem, we go through ROS source code and figure out related packages. The most important git repository is audio_common [4]. In this repository, the package "audio_common_msgs" is necessary for the proper function of "sound_play" package. We finally design and build our sound mechanism through following architecture.

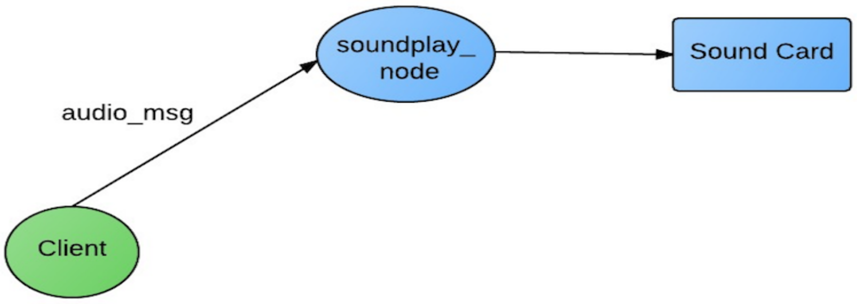
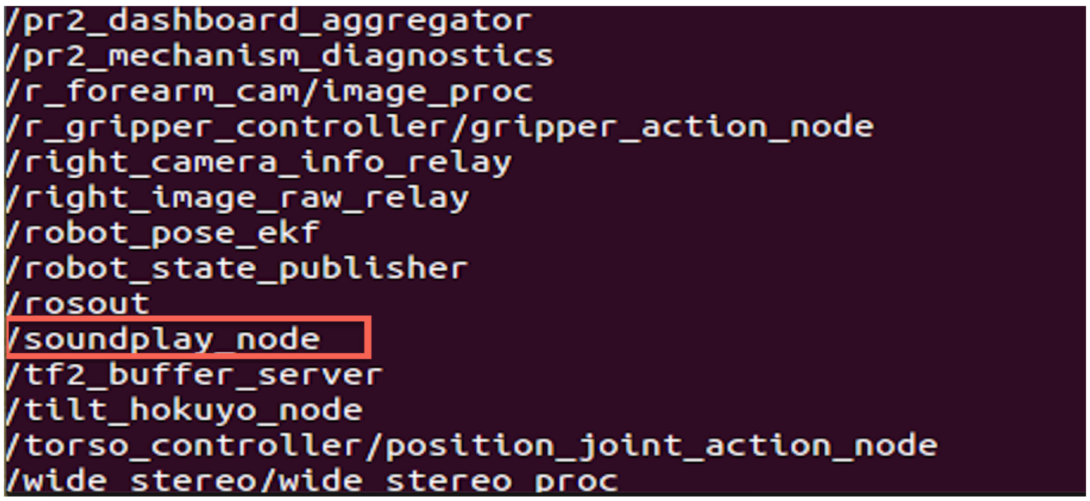


Fig. 12: The soundplay_node for vocalizing the playing



```
/pr2_dashboard_aggregator
/pr2_mechanism_diagnostics
/r_forearm_cam/image_proc
/r_gripper_controller/gripper_action_node
/right_camera_info_relay
/right_image_raw_relay
/robot_pose_ekf
/robot_state_publisher
/rosout
/soundplay_node
/tf2_buffer_server
/tilt_hokuyo_node
/torso_controller/position_joint_action_node
/wide_stereo/wide_stereo_proc
```

Fig. 13: The nodes list after launching our Gazebo environment

3. Reference

[1] Class Sample Code https://github.com/HumanoidRobotics/challenge_problem_1
[2] Point Cloud Library http://pointclouds.org/
[3] project scene_segmenter https://github.com/CURG/scene_segmenter
[4] audio_common https://github.com/ros-drivers/audio_common