Graph-SLAM Mapping of Multiple Gazebo Environments Using RTAB-Map and RGB-D Camera on Mobile Robot

Arnold Kompaniyets

Robotic Software Engineering, Term 2

Project 3

Abstract:

Implementing a previously-constructed differential wheeled robot, an RGB-D camera (in the form of a Kinect camera) and a Hokuyo laser scanner were implemented in adding mapping capabilities to said robot. A mapping node was added via the implementation of RTAB-Map (a graph-SLAM application), with a separate node constructed to transform RGB-D camera depth image data into laser scan format when necessary. Using the teleop twist keyboard package, the aforementioned robot was used to traverse two separate environments, one of a kitchen/dining room and one of a custom-made outdoor map. The success of each mapping run was analyzed via the RTAB-Map database viewer, where the number of loop closures and final derived map could be observed.

Introduction:

In the previous project, emphasis was primarily placed of proper localization of a robot in a provided environment. Said project aimed to tackle a

major challenge of mobile robotics; however, one specific component was already provided – the map. When considering mobile robotics in the real world, the other major challenge that arises beside localization is in constructing the aforementioned map. As such, this project aims to do just that. Using the mobile robot used for Project 2, the previously-placed front-facing camera was transformed to an RGB-D camera and used alongside the pre-existing Hokuyo laser scanner to allow for proper mapping of multiple Gazebo environments. Among the multiple methods available, this project utilized a graph-SLAM application, RTAB-Map, to derive the map. In the complete ROS package, the URDF and launch files of the mobile robot were combined with the appropriate "world" files, a Teleop node to allow for keyboard control of said robot, and lastly a "mapping" launch file which contained the RTAB-Map initialization. Upon completion of each map, the RTAB-Map database viewer was used to analyze the thoroughness and success of a given mapping run.

Background:

As stated above, alongside localization, mapping is a major challenge of mobile robotics. Regardless of a robot's desired task, a robot must be able to properly "see" its environment first, and this necessity is illustrated best when thinking about the most basic definition of a robot. A robot, at its core, is designed to execute an action, which requires an object to execute said action on. Without proper mapping, no robot can then accomplish its task, unless it is placed in a rigid environment with no possible deviations. This, of course, is not very realistic.

In the human mind, the task of mapping is performed unperceivably fast and continuously. Implementing a stereo camera, an audio interpreter, and pressure/texture sensors, the brain is capable of rapidly generating a local map regardless of where a person is placed. While it is not yet possible to truly see the algorithms that the mind uses to analyze sensory data and build its local map, the general outline described above is still followed in robotics. Using its own array of sensors, a robot must be able to use the provided stream of data to properly construct the local environment. The overall challenge of improvement, therefore, is two-fold: one in hardware and the other in software. Sensors, i.e. the hardware, need to be capable of providing extensive, high-quality data of an environment (in whichever physical spectrum they focus – light, sound, etc.). The higher quality the data that the hardware provides, the easier it is to distinguish features. The other side of mapping, the software, is perhaps even more important, because all that extensive sensory data needs to be parsed correctly. The mind of the robot needs to be capable of making the necessary distinctions and identifying the various features within. Various methods to accomplish this have been developed, but focus in this project will be placed on Graph-SLAM. This particular method of mapping makes an interconnected graph of robot poses and features, connecting further attained poses or features to said graph. As such, this method can very easily be compared to a problem of physics, where each feature/pose is a mass, and each mass is tied to others via flexible string. Depending on certainty, various strings can be tightened of loosened, which in turn refines and gives good form to a map. With adding complexity, the form of the map takes greater solidity, offering benefit to repeat exploration of a space.

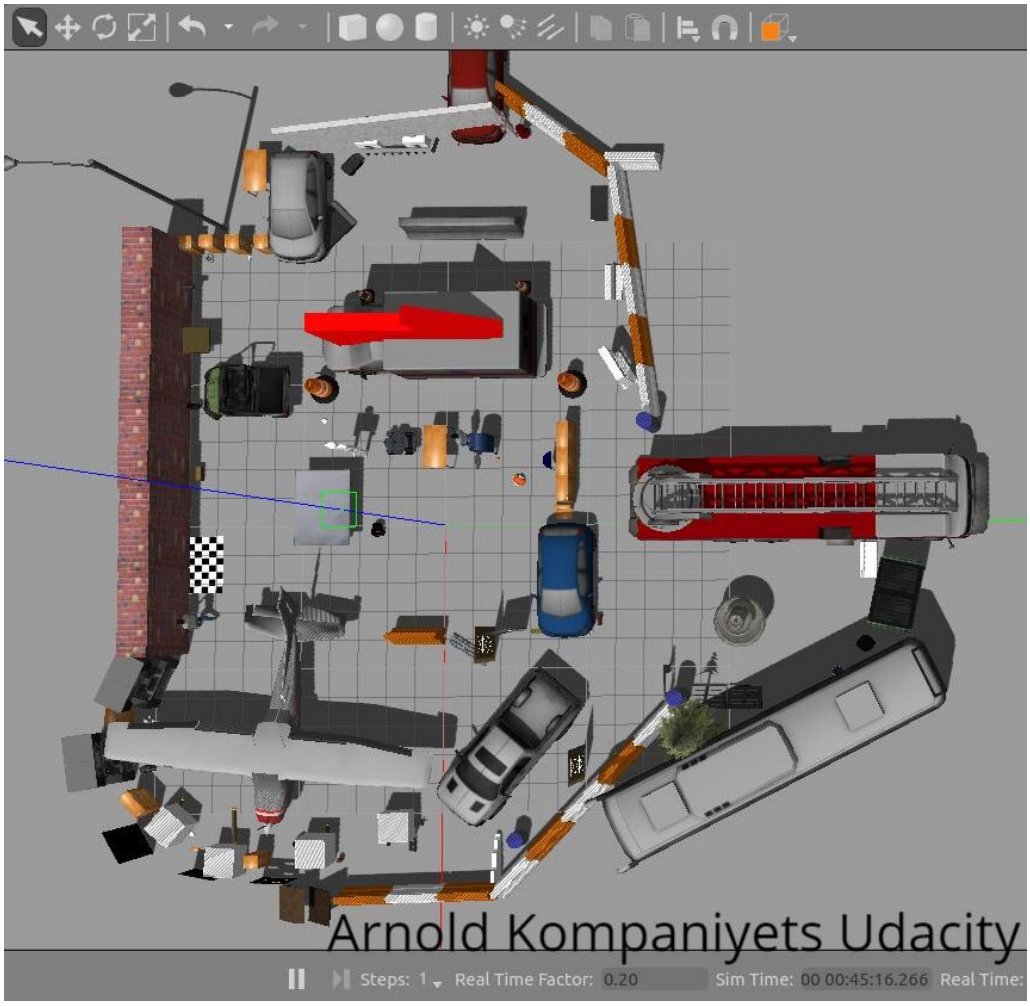
Taking concept to practical application, RTAB-Map is a ROS application with implements Graph-SLAM into robotics. This particular application only uses observed features and odometry data to construct the interconnected web described above. With the use of an RGB-D camera specifically, RTAB-Map looks at each incoming image and marks unique color patterns and combinations as "features". The application does not need to know what that feature actually is (and it may not be anything of extensive significance to the human eye), but as long as there is enough detail present to identify many features, the interconnected graph described above can be made correctly. In addition, the application uses the concept of loop closure, which allows for previously seen locations to be labeled as such and thus further linked.

Scene and Robot Configuration:

Scene:

In designing the custom world for this particular project, no specific structure or plan was followed, per se, other than trying to make the world both interesting to look at and, of course, very feature rich. As such, after opening up a blank world in Gazebo, the entire list of available models was viewed. Going down said list, models were randomly chosen and placed on the map in an arrangement that was interesting or somewhat comedic in nature. Care was taken to not overuse any specific models in excess. Due to the fact that the pixel arrangement, and therefore overall appearance, of each model is identical in Gazebo, there is always the risk that RTAB-Map thinks multiple copies of the same model are actually one and the same, thereby initiating loop closure.

This potentially posed a large issue in this simulation-based mapping project, so the direct means taken to avoid this was to only use a model once if possible. This was not always the case, especially when constructing the walls, so a secondary tactic used was to place different objects in and around the identical models, therefore hopefully preventing RTAB-Map from initiating unnecessary loop closures (i.e. placing a Coke can in front of one wall block and a table in front of the adjoining one). Lastly, after establishing a general map structure, extra models were added to allow the robot to have more defined paths throughout the map, versus just travelling in circles. The final project appeared as such in Gazebo:



Robot Configuration:

In constructing the robot to be used in this mapping project, the URDF framework was copied over from the previous project, making a robot that is a rectangular box, having two caster wheels below the base, two wheels attached at each side, a camera box at the front, and a Hokuyo laser on top. A change, however, was made to the Gazebo file for the robot, specifically to the camera controller. With this project requiring an RGB-D camera, the package used for the camera controller was changed to that of the Kinect ("libgazebo_ros_openni_kinect.so"). The resulting code appears as such:

```
<plugin name="camera_controller" filename="libgazebo_ros_openni_kinect.so">
  <alwaysOn>true</alwaysOn>
  <updateRate>10.0</updateRate>
  <cameraName>mapping_bot/camera1</cameraName>
  <imageTopicName>rgb/image_raw</imageTopicName>
  <depthImageTopicName>depth/image_raw</depthImageTopicName>
  <pointCloudTopicName>depth/points</pointCloudTopicName>
  <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>

<depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTopic
Name>
  <frameName>camera_optical</frameName>
  <pointCloudCutoff>0.4</pointCloudCutoff>
  <hackBaseline>0.07</hackBaseline>
  <distortionK1>0.0</distortionK1>
  <distortionK2>0.0</distortionK2>
  <distortionK3>0.0</distortionK3>
  <distortionT1>0.0</distortionT1>
  <distortionT2>0.0</distortionT2>
```

```
    <CxPrime>0.0</CxPrime>
    <Cx>0.0</Cx>
    <Cy>0.0</Cy>
    <focalLength>0.0</focalLength>
  </plugin>
```

The information entered was largely identical to that shown in the preceding lessons, with a few exceptions in naming. The camera name was changed to reflect the specific chosen topic (mapping_bot/camera1), and the frame name had to be changed as well. The latter change took quite a bit of time to figure out, as its importance was not initially explained within the lessons. Therefore, it was left as is to begin with; however, when first visualizing sensory data in Rviz, the RGB-D camera images were projected not straight forward as expected but instead straight up and at an odd angle. After a bit of experimentation in Rviz, to no success, it was realized that the Gazebo reference frame for the RGB-D camera was different from the URDF format entered in the XML file. At this point, an extra blank link was created in the robot's URDF file, called "camera_optical", and it was joined to the main camera link with a -90° change in both the roll and yaw, in order to reflect the change in frames from URDF to Gazebo. With the frame name in the Gazebo file changed to "camera_optical", the sensory data projections in Rviz were directed properly, directly in front of the robot.

Following this, the launch folder was modified to add a mapping file, which appeared as such:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<launch>
  <!-- Arguments for launch file with defaults provided -->
  <arg name="database_path"     default="rtabmap.db"/>
  <arg name="rgb_topic"   default="/mapping_bot/camera1/rgb/image_raw"/>
  <arg name="depth_topic" default="/mapping_bot/camera1/depth/image_raw"/>
  <arg name="camera_info_topic" default="/mapping_bot/camera1/rgb/camera_info"/>

  <!-- Mapping Node -->
  <group ns="rtabmap">
    <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen" args="--
delete_db_on_start">

      <!-- Basic RTAB-Map Parameters -->
      <param name="database_path"       type="string" value="$(arg database_path)"/>
      <param name="frame_id"            type="string" value="robot_footprint"/>
      <param name="odom_frame_id"       type="string" value="odom"/>
      <param name="subscribe_depth"     type="bool"   value="true"/>
      <param name="subscribe_scan"      type="bool"   value="true"/>

      <!-- RTAB-Map Inputs -->
      <remap from="scan"            to="/scan"/>
      <remap from="rgb/image"       to="$(arg rgb_topic)"/>
       <remap from="depth/image"     to="$(arg depth_topic)"/>
       <remap from="rgb/camera_info" to="$(arg camera_info_topic)"/>

       <!-- RTAB-Map Output -->
       <remap from="grid_map" to="/map"/>

          <!-- Rate (Hz) at which new nodes are added to map -->
      <param name="Rtabmap/DetectionRate" type="string" value="1"/>
```

```
      <!-- 2D SLAM -->
      <param name="Reg/Force3DoF" type="string" value="true"/>

      <!-- Loop Closure Detection -->
    <!-- 0=SURF 1=SIFT 2=ORB 3=FAST/FREAK 4=FAST/BRIEF 5=GFTT/FREAK
6=GFTT/BRIEF 7=BRISK 8=GFTT/ORB                9=KAZE -->
    <param name="Kp/DetectorStrategy" type="string" value="8"/>

<!-- Maximum visual words per image (bag-of-words) -->
          <param name="Kp/MaxFeatures" type="string" value="400"/>

      <!-- Used to extract more or less SURF features -->
              <param name="SURF/HessianThreshold" type="string" value="100"/>

  <!-- Loop Closure Constraint -->
<!-- 0=Visual, 1=ICP (1 requires scan)-->
      <param name="Reg/Strategy" type="string" value="0"/>

<!-- Minimum visual inliers to accept loop closure -->
      <param name="Vis/MinInliers" type="string" value="50"/>

      <!-- Set to false to avoid saving data when robot is not moving -->
      <param name="Mem/NotLinkedNodesKept" type="string" value="false"/>

    </node>
  </group>
</launch>
```

The structure of this mapping launch file was virtually identical in structure to the template given in preceding lessons, with only a few changes made similar in style to the Gazebo file. Specifically, the topic names given to the mapping node were changed to reflect the names implemented by the specific robot chosen. Additionally, the 'loop closure detection method' value was changed, many times actually, in various attempts to find the best choice for a given map. Along with this parameter, the 'loop closure constraint' and 'minimum inliers to accept loop closure' parameters were changed frequently during the mapping optimization phase of the project.

Furthermore, an addition to the main launch file proved to be necessary to properly map the provided kitchen dining map. In the first page of the project section, a method was described of using an existing ROS package to convert RGB-D depth image values to "laser scan" data in order to allow RTAB-Map to function fully without an actual laser scanner present. However, it was also stated that since the Hokuyo laser was already present on the robot to begin with, doing the conversion would not be necessary for this project. This statement proved to be true in mapping the custom world, but not for the initial provided example. In loading up the kitchen dining world and looking at Rviz, the laser scan data would not come in from Hokuyo. Many different changes were tried in Rviz and the parameters/naming within the URDF and launch files were checked repeatedly. When nothing erroneous could be found, a simple browse through the Gazebo model information showed that the kitchen dining model was only a visual element and contained no collision parameters. As such, the Hokuyo laser scanner was working perfectly fine, there was simply nothing for it to detect.

In order to allow RTAB-Map to still construct an occupancy map of the kitchen dining environment, the following node was added to the main launch file:

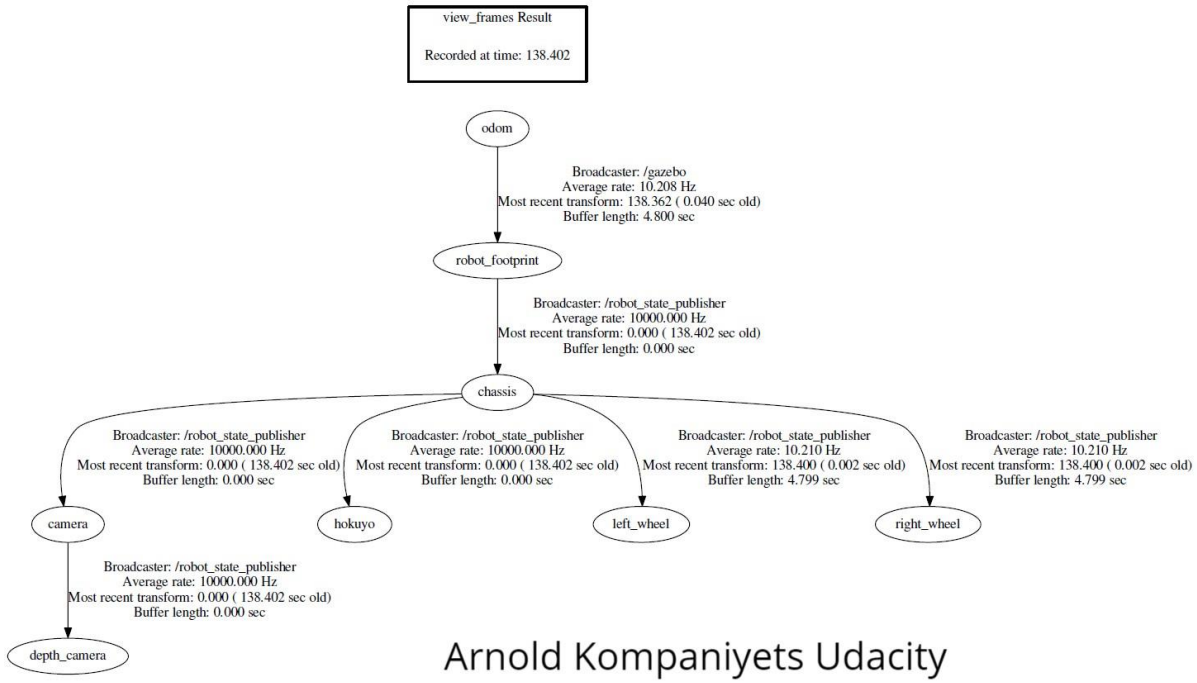
```
  <node name="depth_to_laser" pkg="depthimage_to_laserscan"
type="depthimage_to_laserscan">
    <remap from="image" to="/mapping_bot/camera1/depth/image_raw"/>
    <remap from="scan" to="/scan"/>
    <remap from="camera_info" to="/mapping_bot/camera1/depth/camera_info"/>
    <param name="output_frame_id" type="string" value="camera"/>
  </node>
```

As previously described, this node invokes a pre-made ROS package that subscribes to the RGB-D camera's depth image stream and outputs sensory data in the form of a laser scanner (renamed simply to "/scan"). Adding a bit to the overall confusion of reference frames, this particular ROS package prefers to use the frame of the URDF structure, so the output frame had to be identified as the initial camera link of the robot's URDF file, versus the "camera_optical" link used to visualize the remainder of the camera data. With this node added, RTAB-Map was able to use the purely visual model of the kitchen dining object and still construct an occupancy grid.

Additionally, in order to traverse each of the worlds selected, the constructed robot needed to be controlled via keyboard. To do this, ROS' teleop functionality was invoked. After running '$ apt-get update' in the terminal window of the project workspace, teleop functionality was added via the command '$ sudo apt-get install ros-kinetic-teleop-twist-keyboard'. With this package installed, a new terminal could be opened, running the command

'$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py' to allow complete control of the robot via keyboard.

When viewing the final robot model in regards to all the frames of reference, the following graph was constructed:

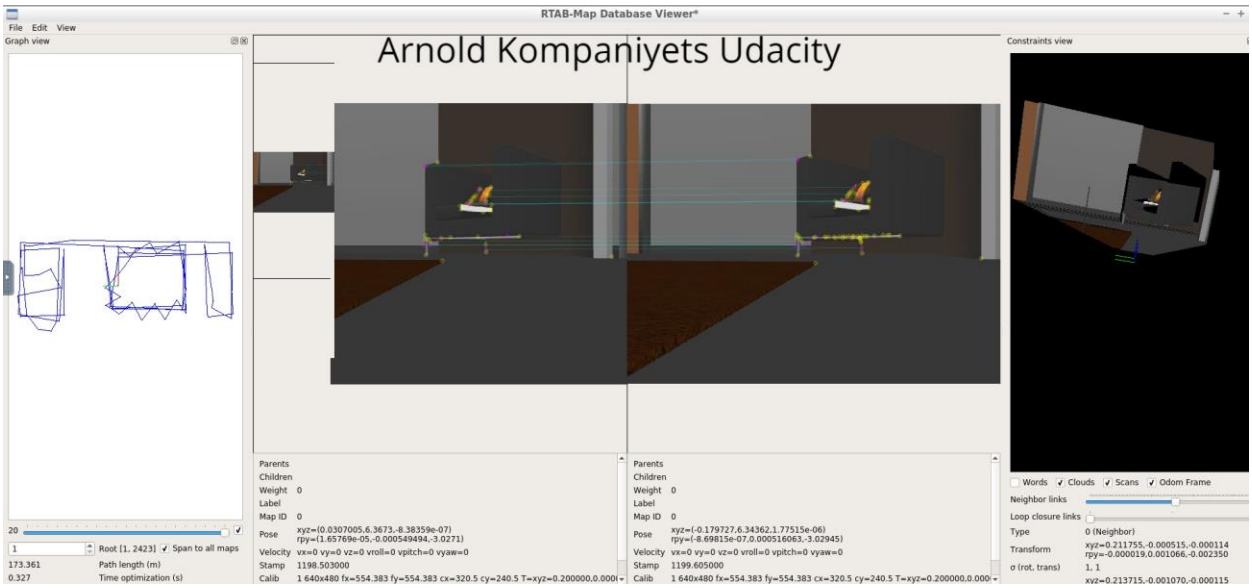


Results:

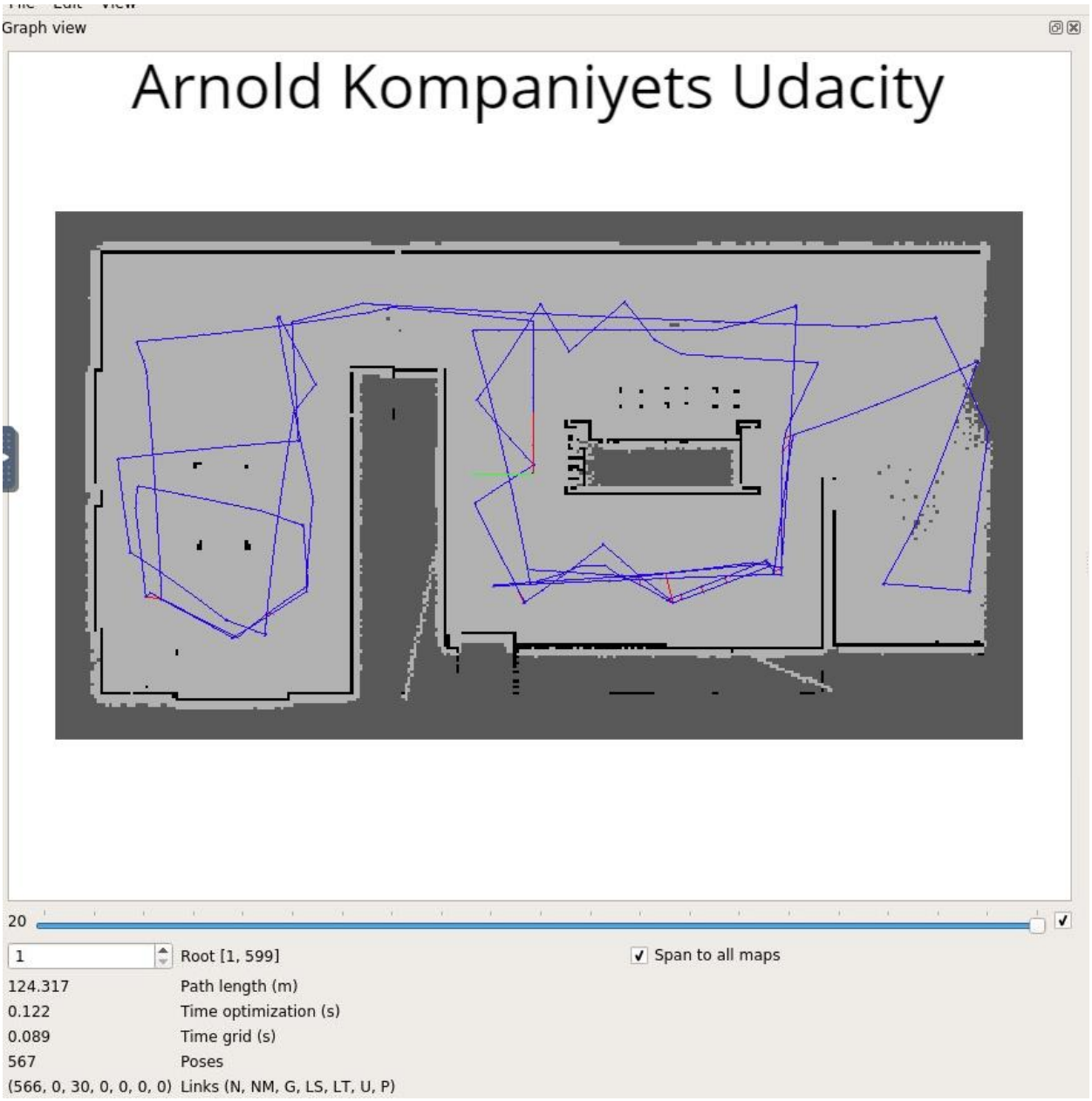
Kitchen Dining Map:

From the first run in mapping this provided world, the primary task in success was in optimizing the loop closure parameters. Being a simulated environment with virtually perfect sensors, the RGB-D images projected were always correct, and the odometry data displaying the chosen path never went astray. With such relatively perfect hardware, loop closure actually proved to not only be unnecessary, but often detrimental if not tuned correctly.

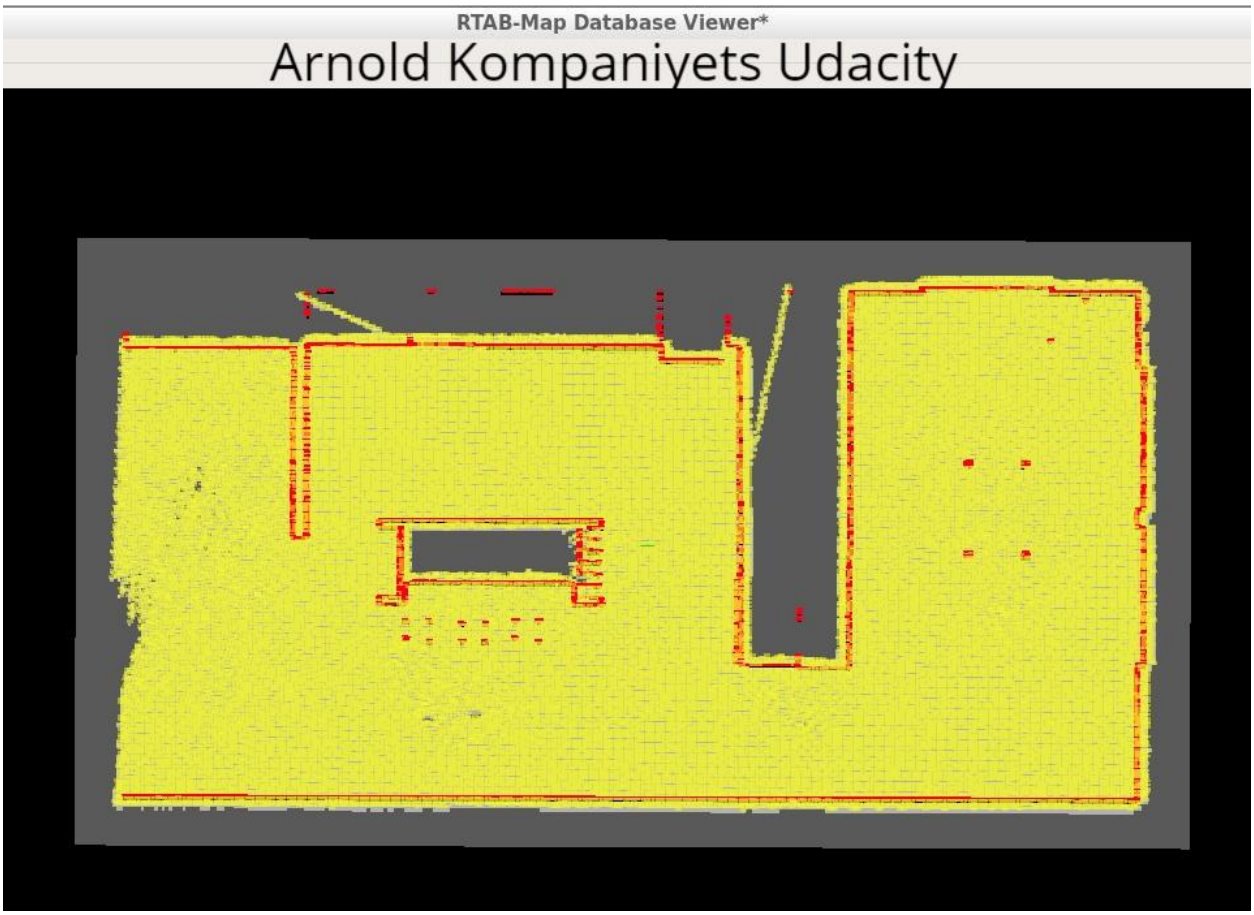
Mapping the kitchen dining world resulted in two databases. The first run involved using the FAST/FREAK method, resulting in the following completed run:



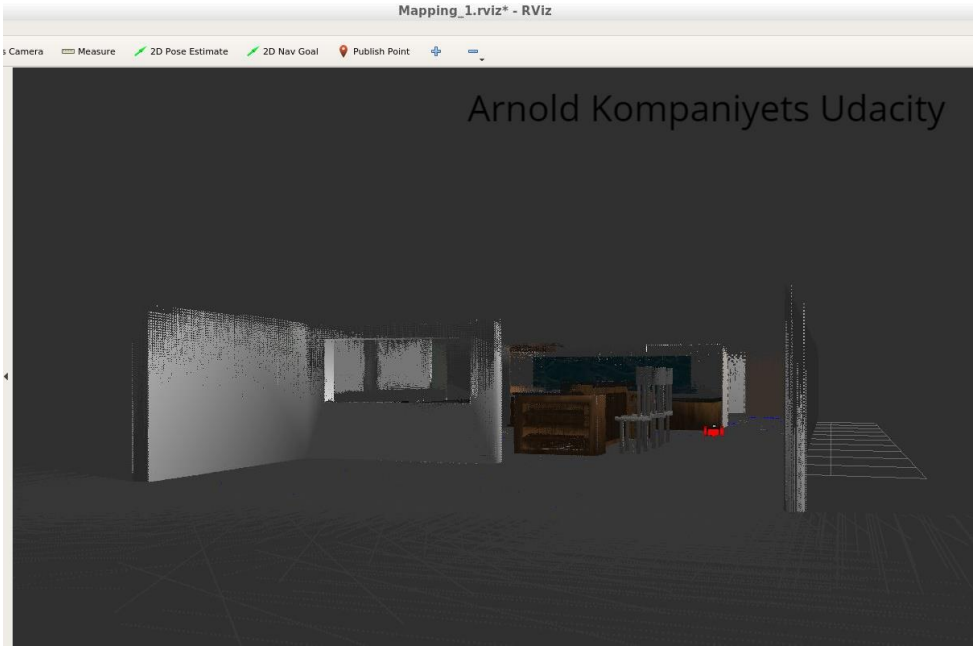
This run resulted in 3 global loop closures and appeared to locate an appropriate number of features to provide an accurate map of the environment. However, this run did attempt to use the Hokuyo laser scanner, but the purely visual map did not allow for mapping of the occupancy grid. Therefore, another run was done, this time changing the loop closure method to GFTT/ORB and adding the "depth image to laser scan" node. This second run resulted in 30 global loop closures and produced the following result:

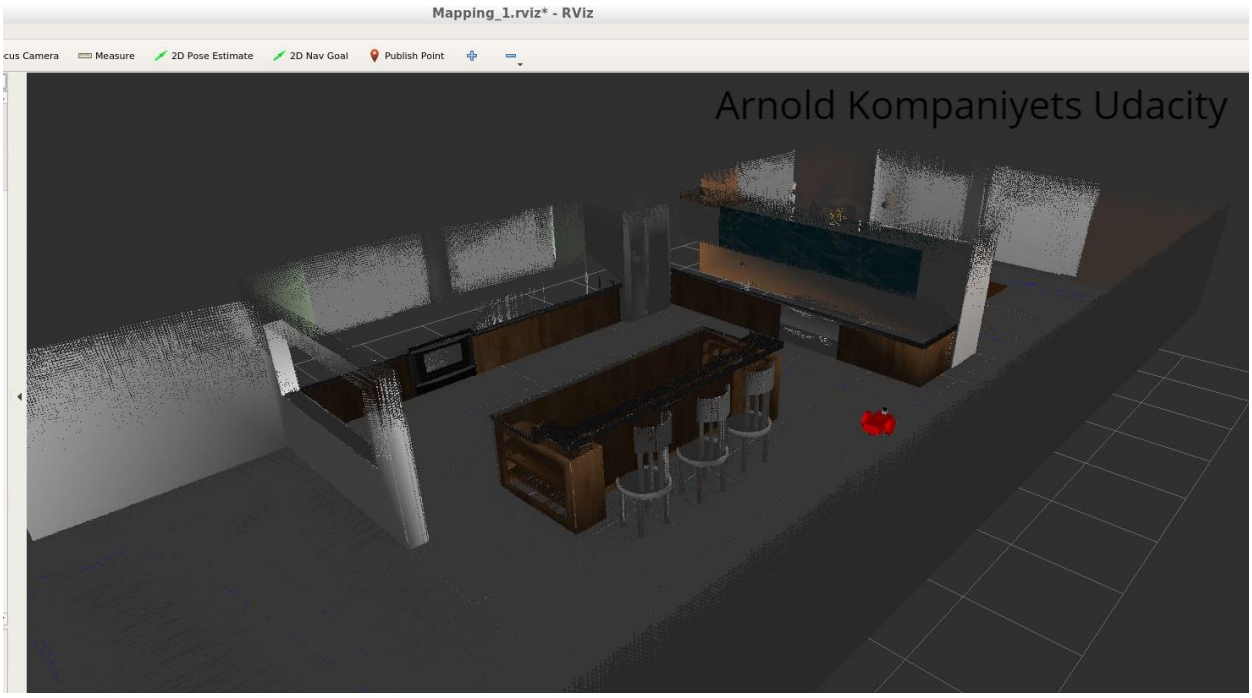


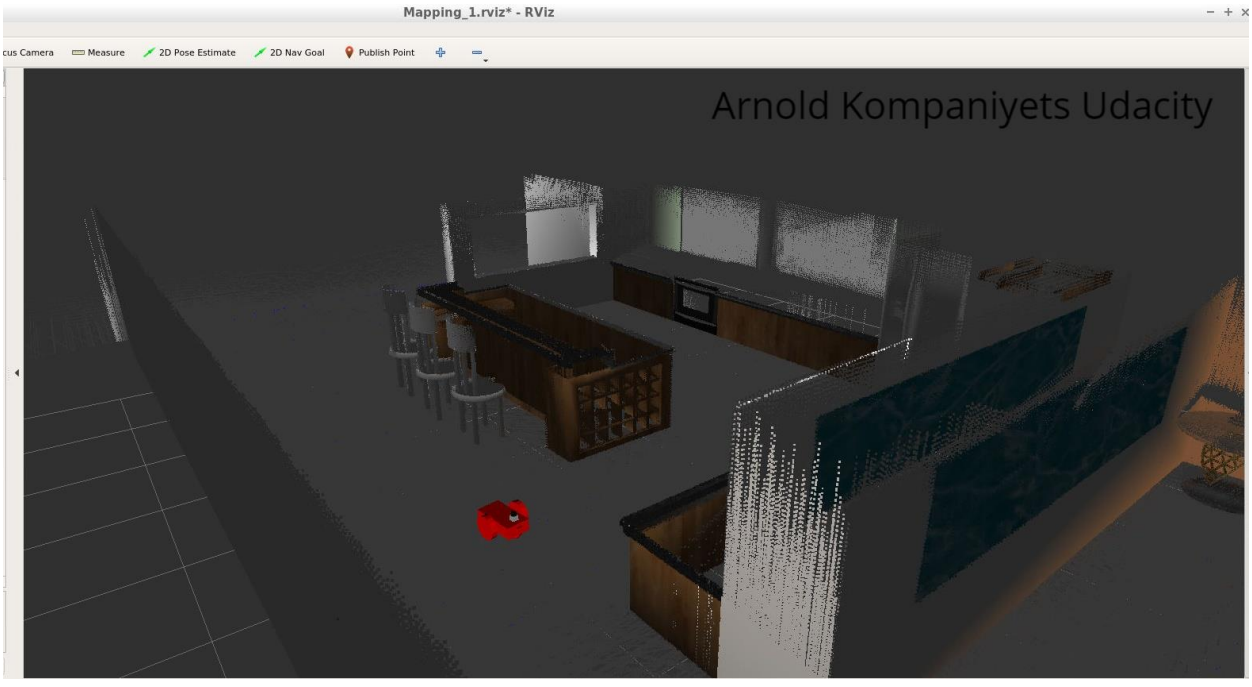
This specific graph was able to show both the robot's path and where the various walls and objects were located. The occupancy grid was now filled as well, appearing as such:

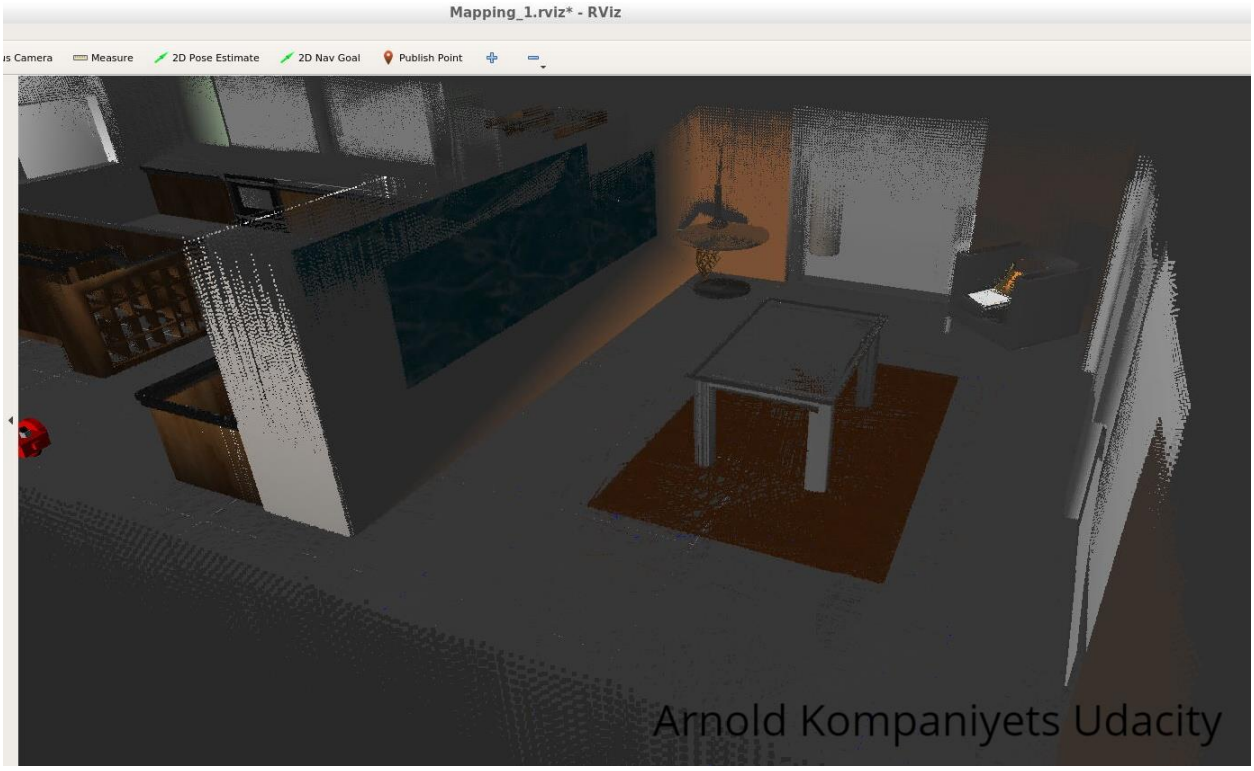


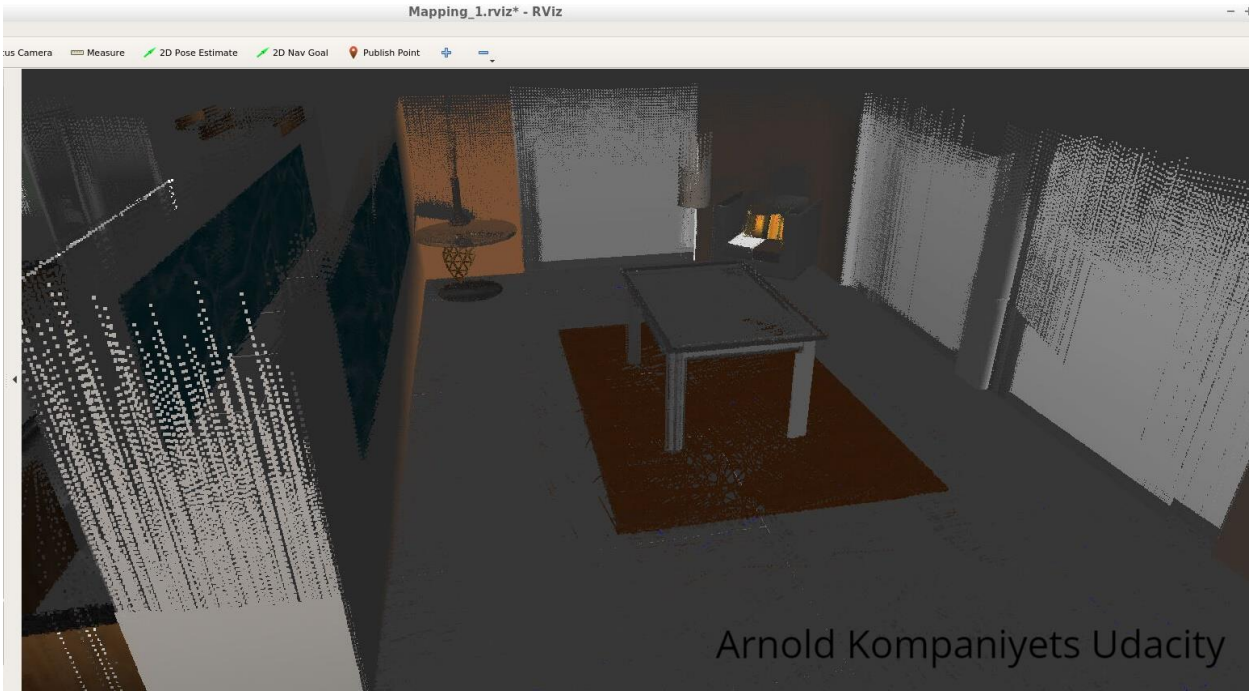
After completing the entirety of mapping, the Rviz map appeared as such:



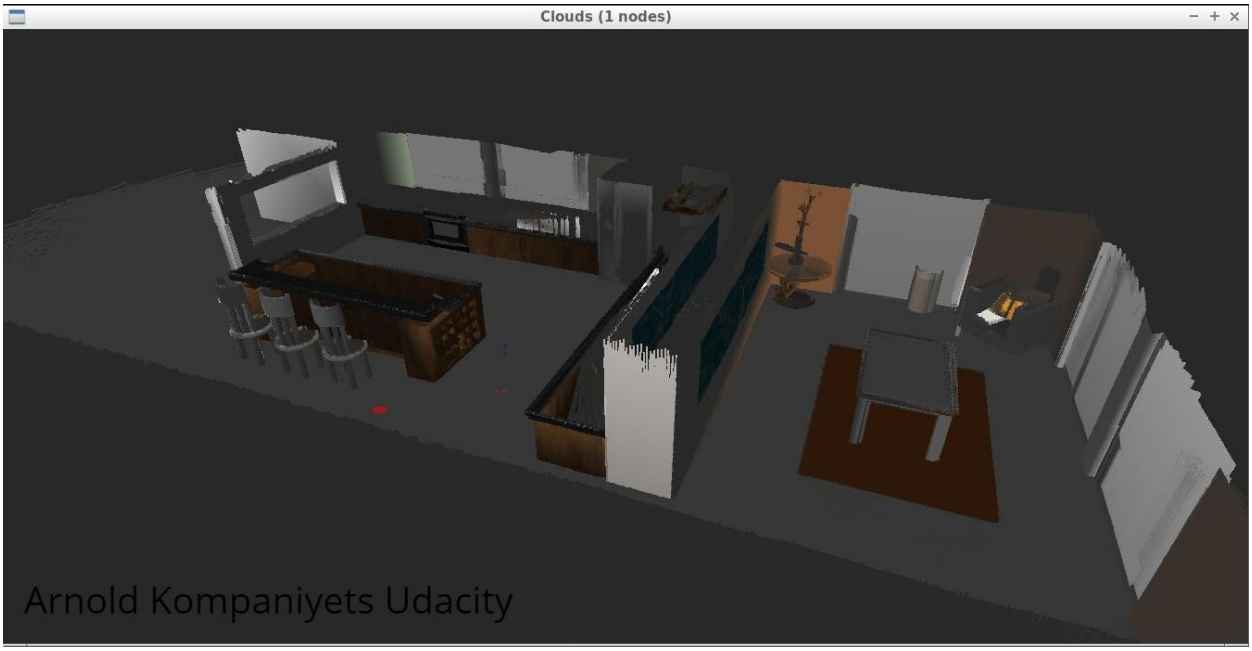

Mapping_1.rviz* - RViz
cus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Arnold Kompaniyets Udacity

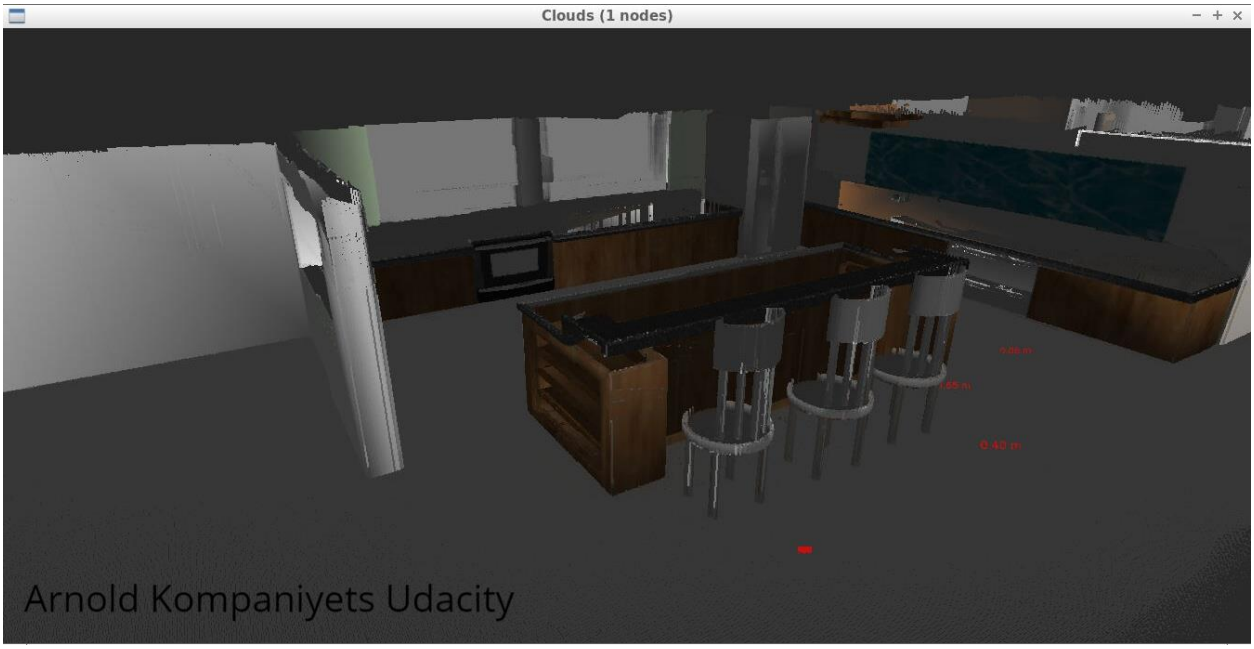

Mapping_1.rviz* - RViz
cus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Arnold Kompaniyets Udacity

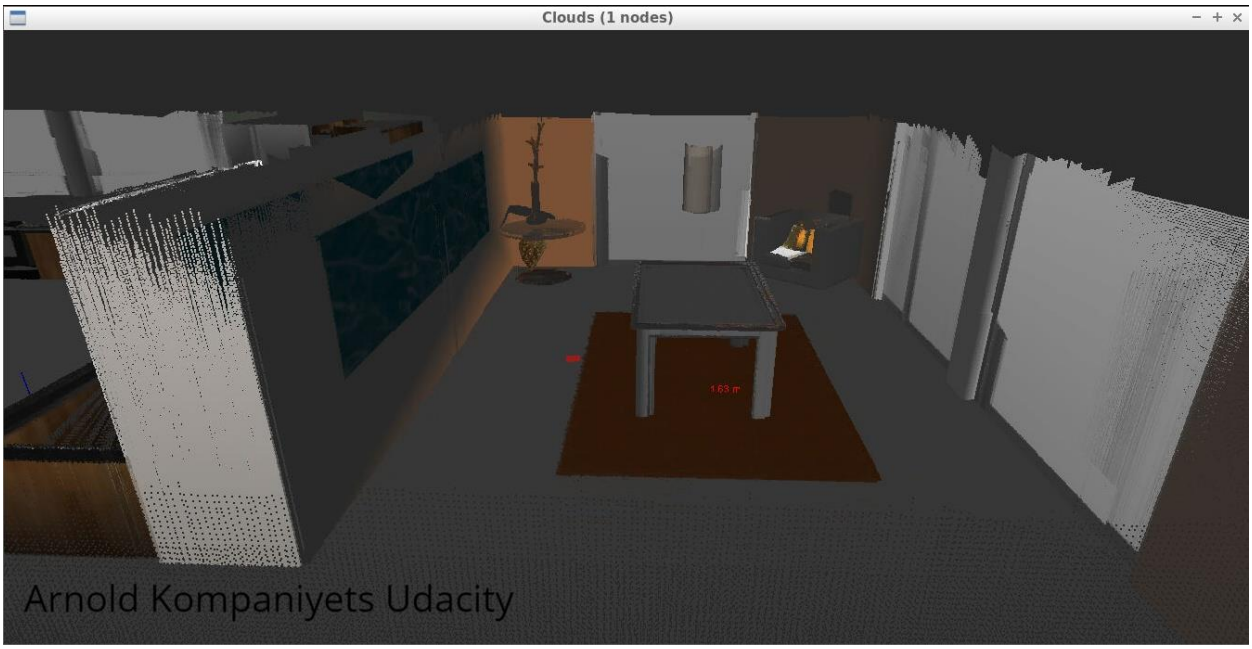

Mapping_1.rviz* - RViz
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Arnold Kompaniyets Udacity


Mapping_1.rviz* - RViz
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Arnold Kompaniyets Udacity

This mapping was able to accurately re-draw the kitchen dining map provided, with minimal, if any, overlapping or shadow-objects present. After reconstructing the RTAB-Map data in the database viewer, the following result was attained:



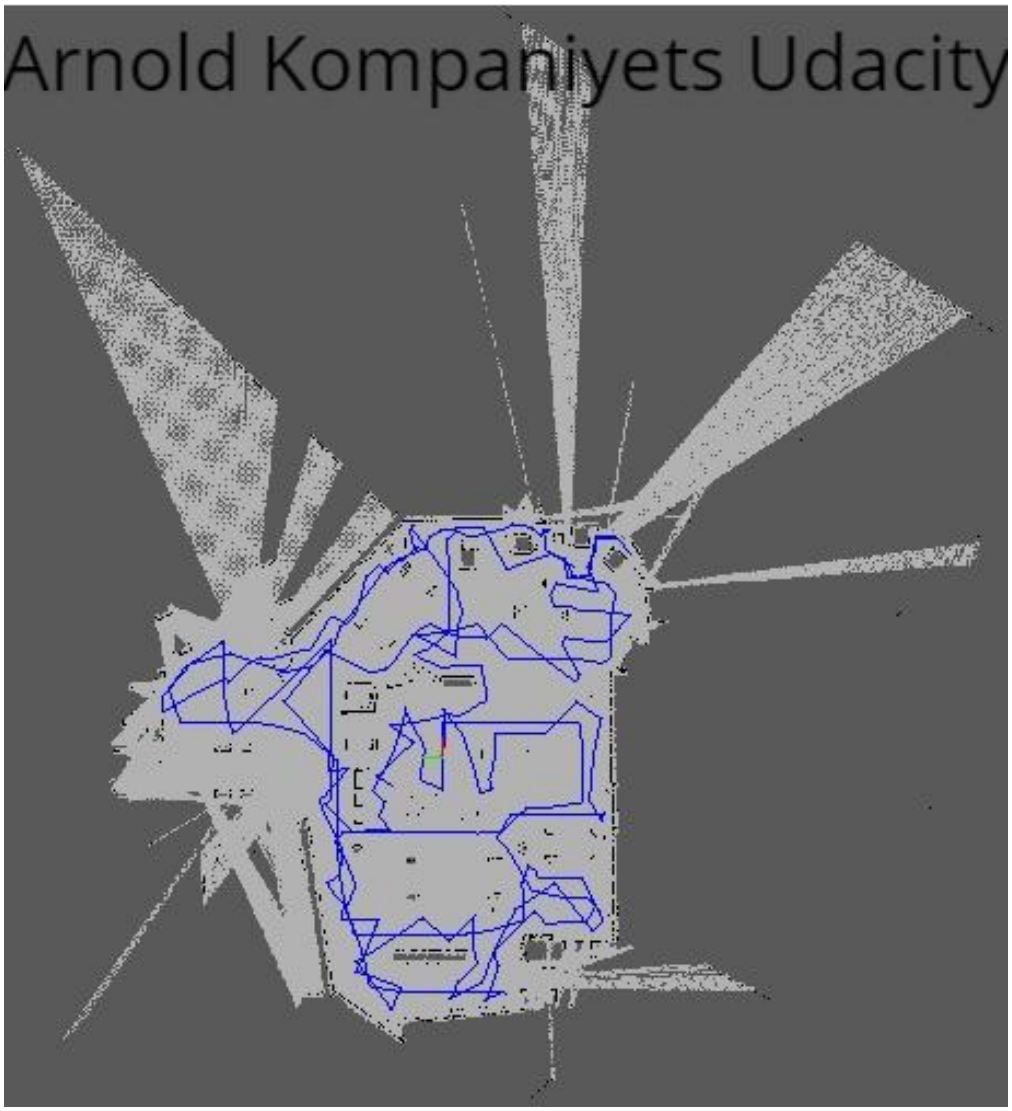




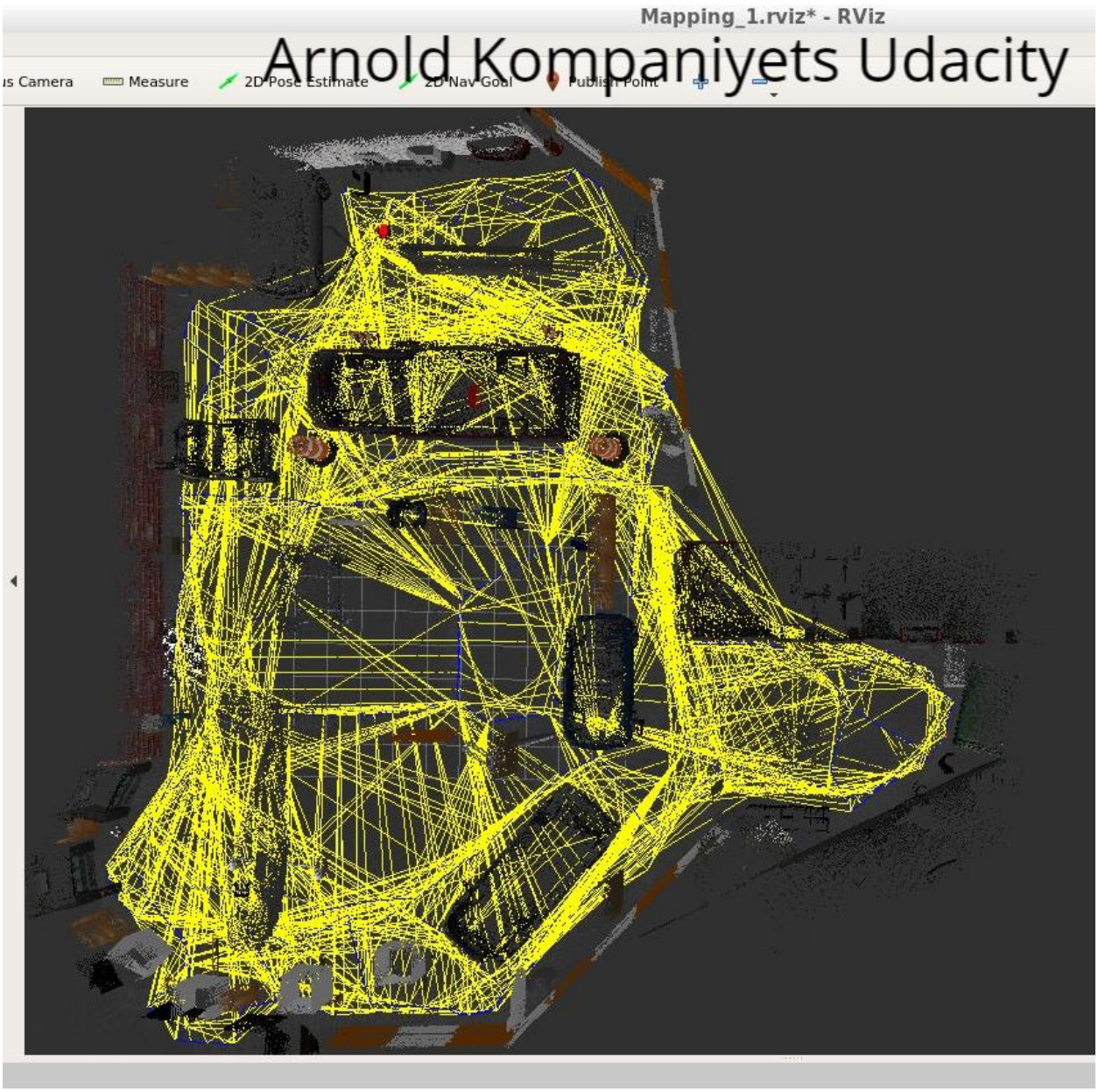
Custom Map:

With the custom map, attaining an appropriate number of minimum inliers for loop closure proved to be much more difficult. Both loop closure methods described previously developed the tendency to initiate many unnecessary loop closures, making correct mapping incredibly difficult. With the custom map possessing so many more features to draw from, RTAB-Map began to draw connections when absolutely non necessary. Eventually, the minimum number of inliers was raised to 250 (using closure method GFTT/ORB). This resulted in greatly minimized overlapping and shadow-objects, keeping each unique object in a solid, singular form. However, the issue of unnecessary loop closure remained prevalent. As an example, RTAB-Map had difficulty separating the right side of the green ATV from its left side. At this stage, the loop closure constraint was moved from visual to ICP, which fixed the dilemma. By creating extensive local feature association and loop

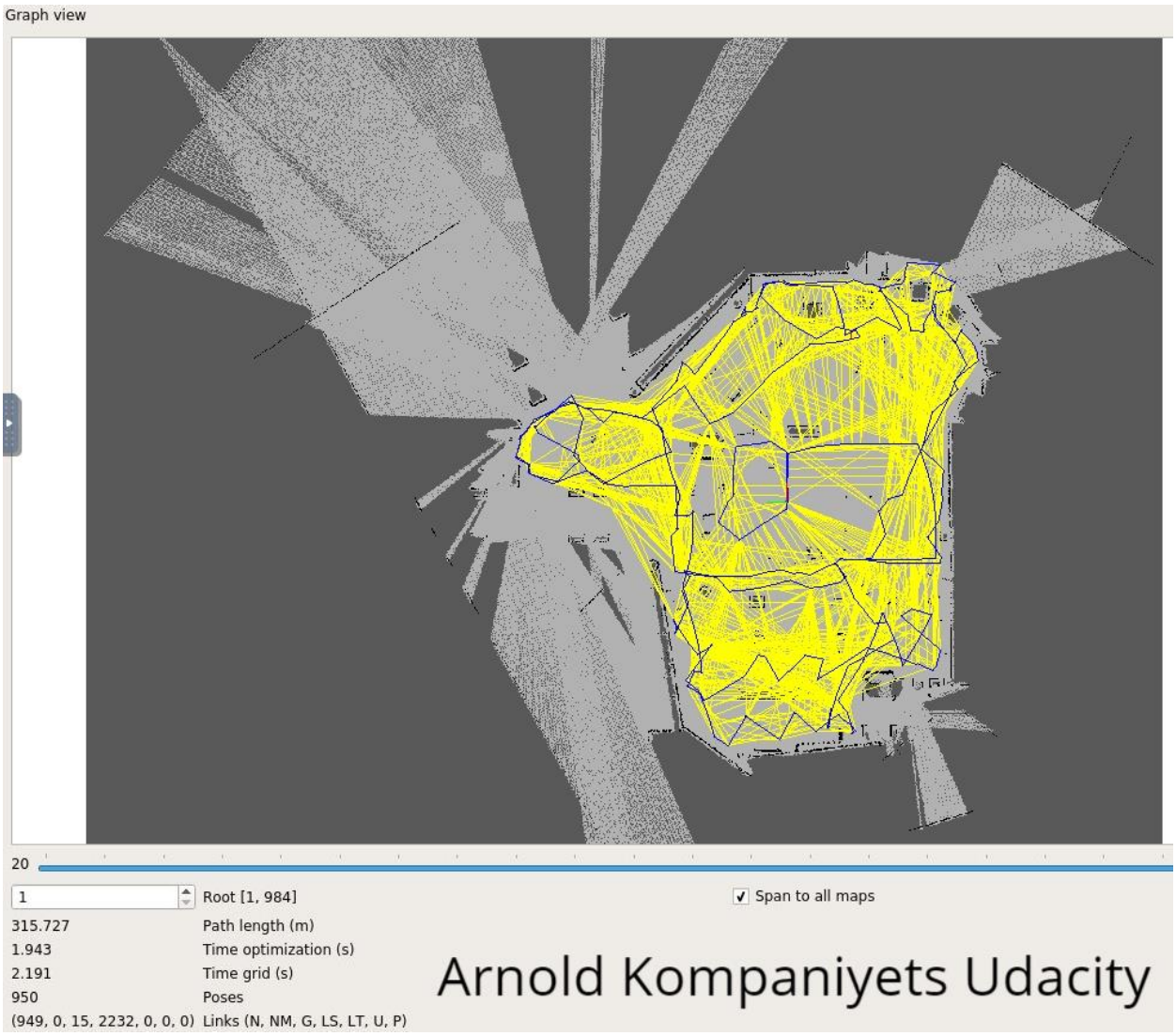
closures, RTAB-Map was constrained from radically altering the already-traversed map. The entire path taken was rather extensive:



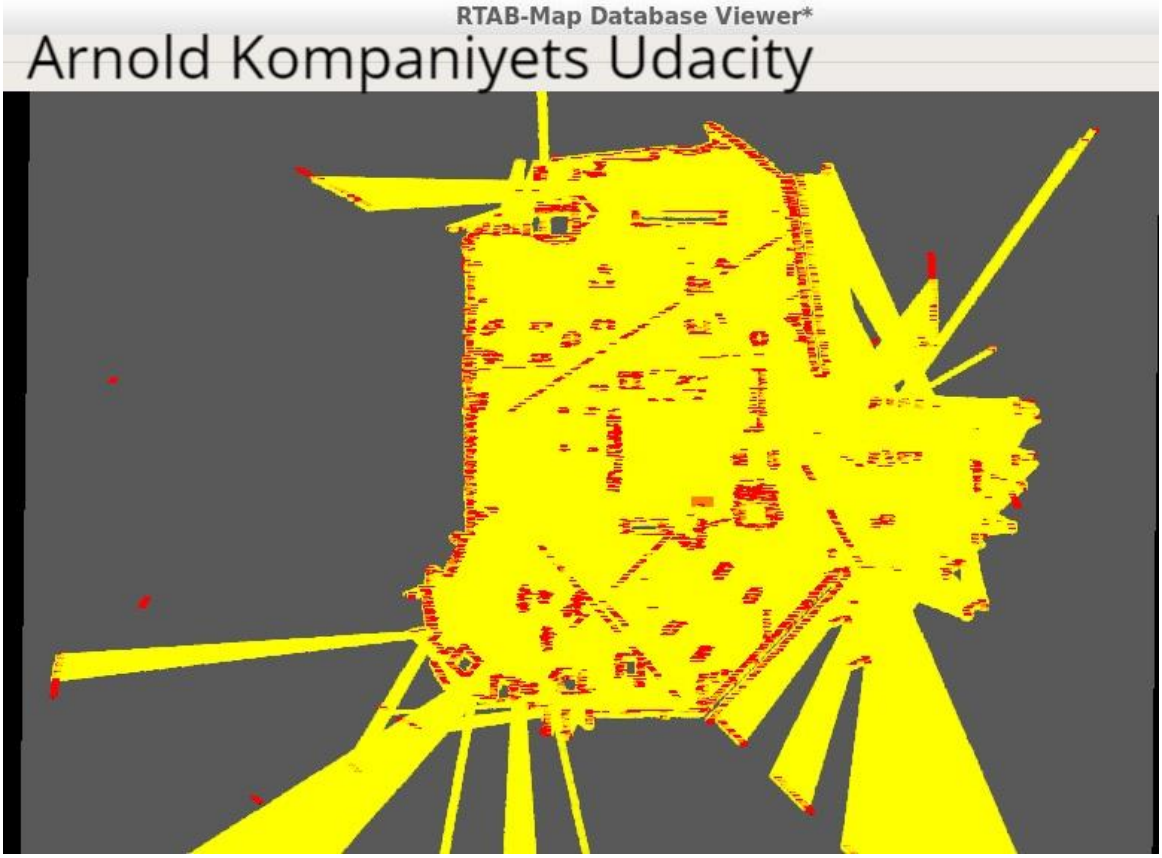
Therefore, as expected, the total number of local connections was extensive as well (illustrated by yellow lines below):



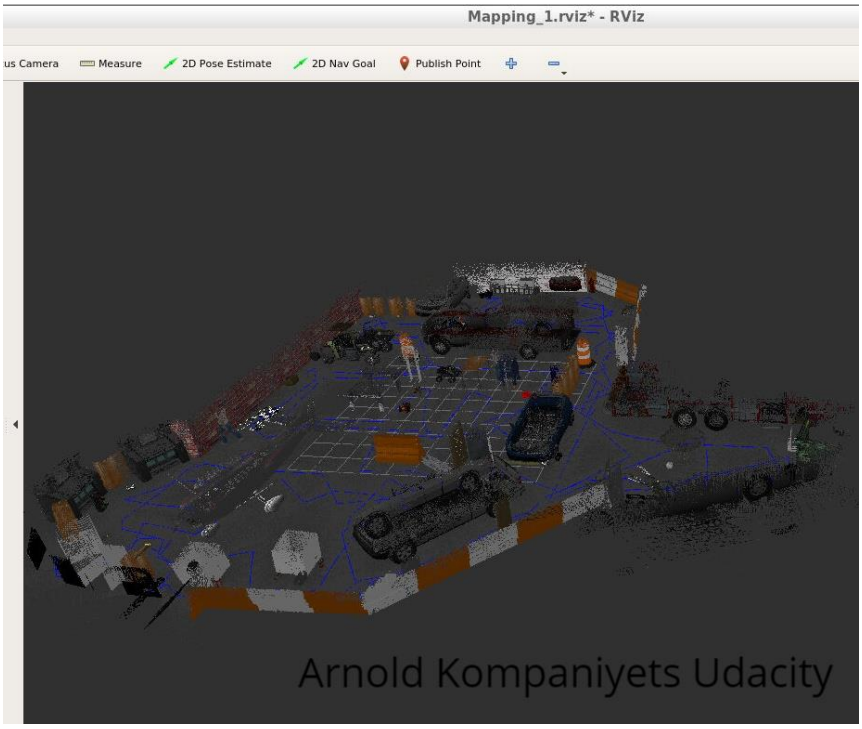
In the RTAB-Map database viewer, the overall path and local links were illustrated together as such:

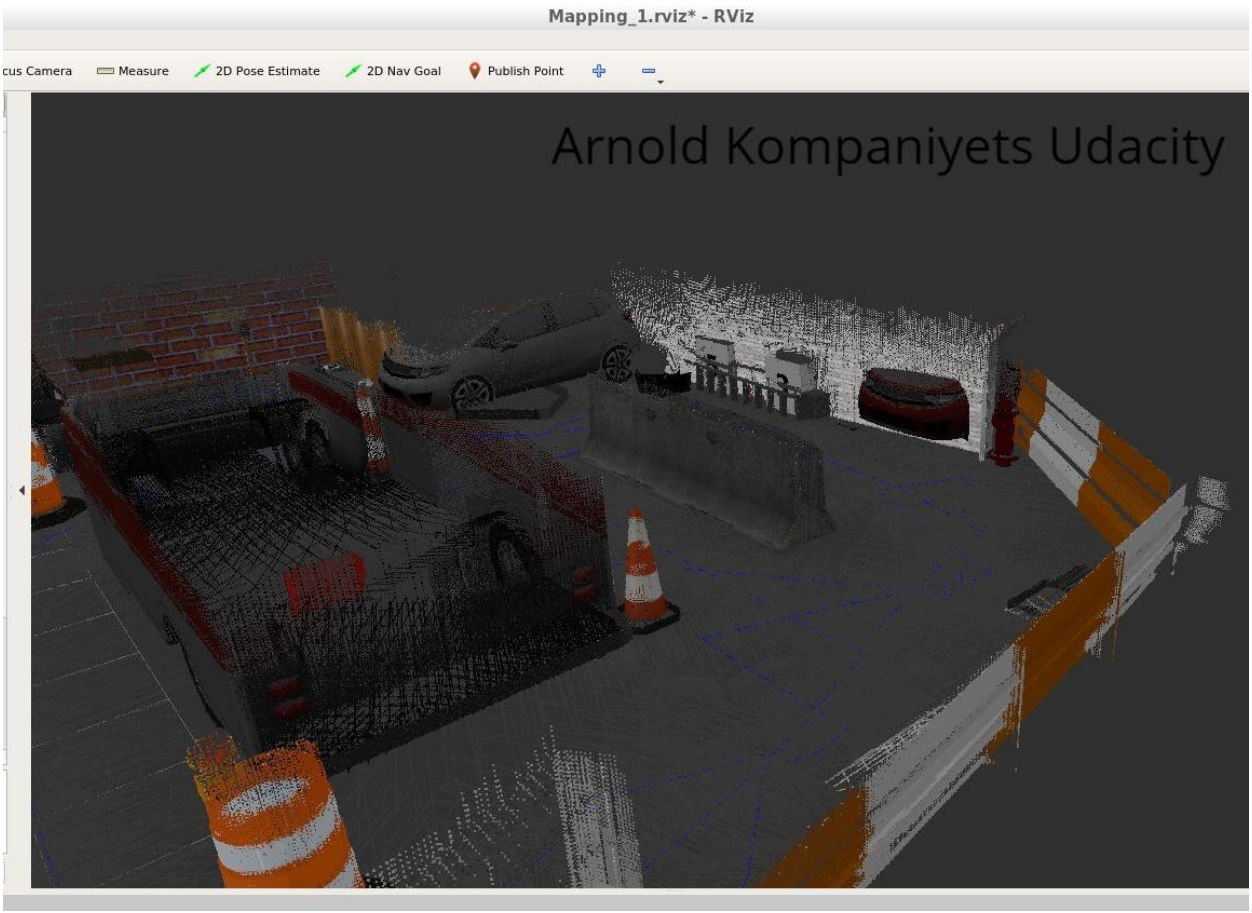


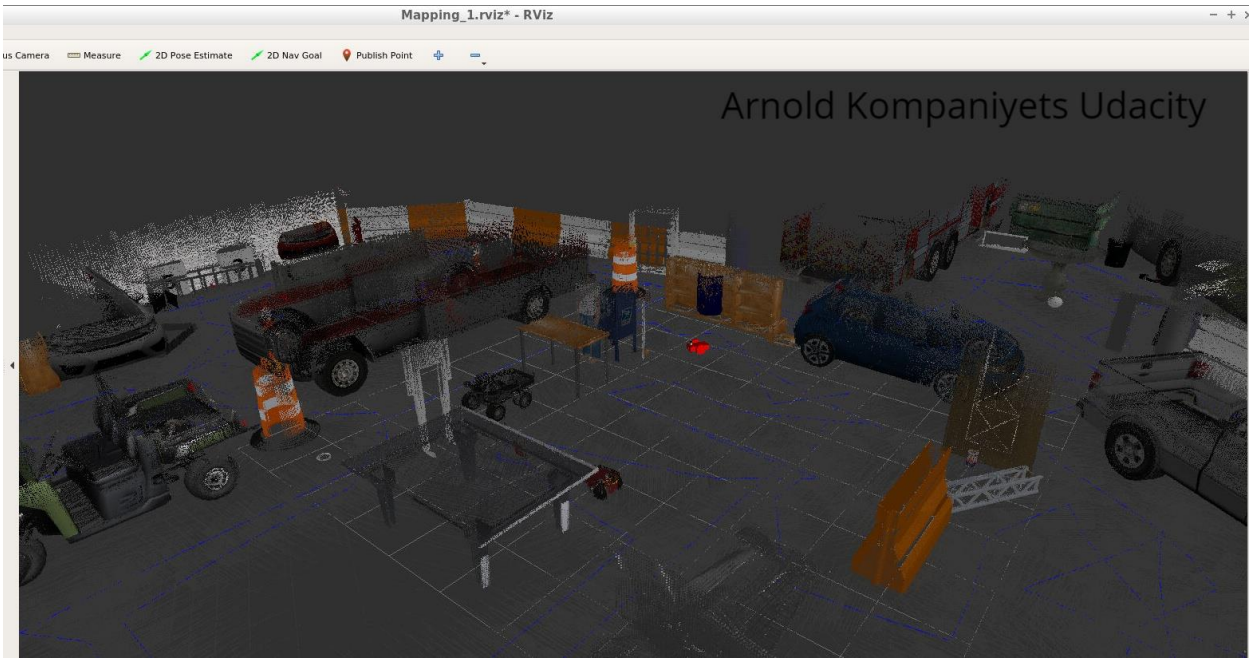
Additionally, as can be seen above, a total of 15 global loop closures were performed, and, since the map implemented contained objects with collision parameters, the Hokuyo laser was used in constructing the occupancy grid:

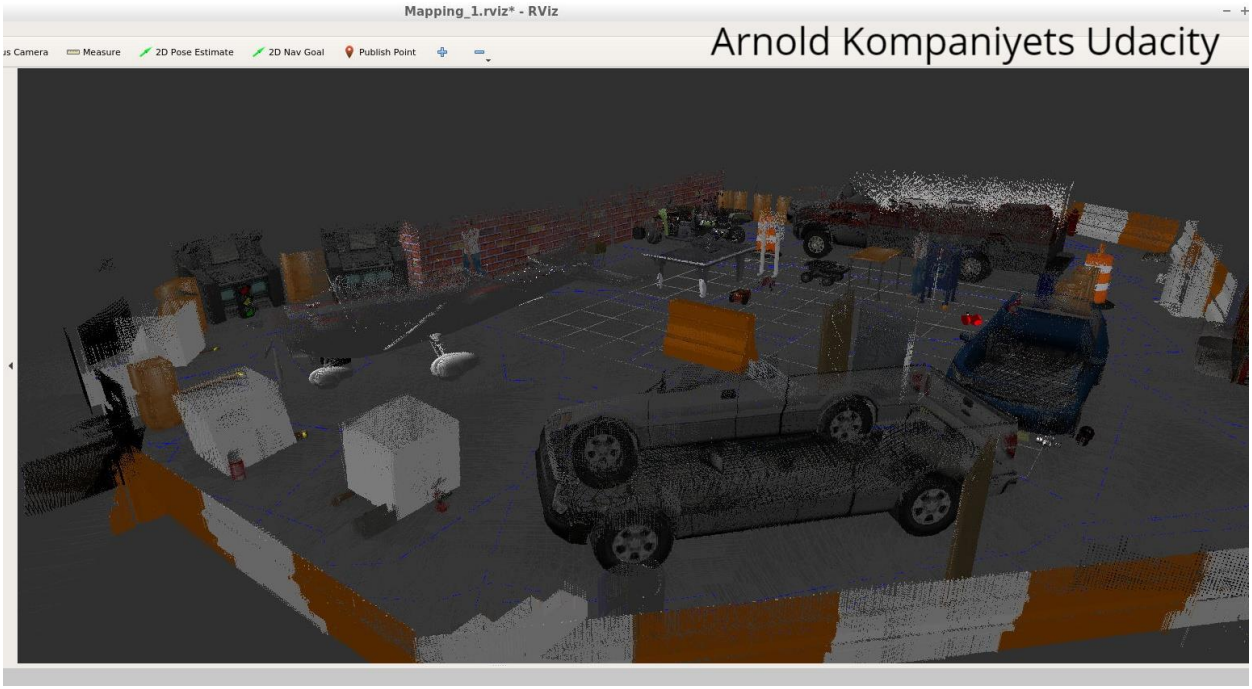


At the conclusion of mapping, the Rviz database appeared as such:

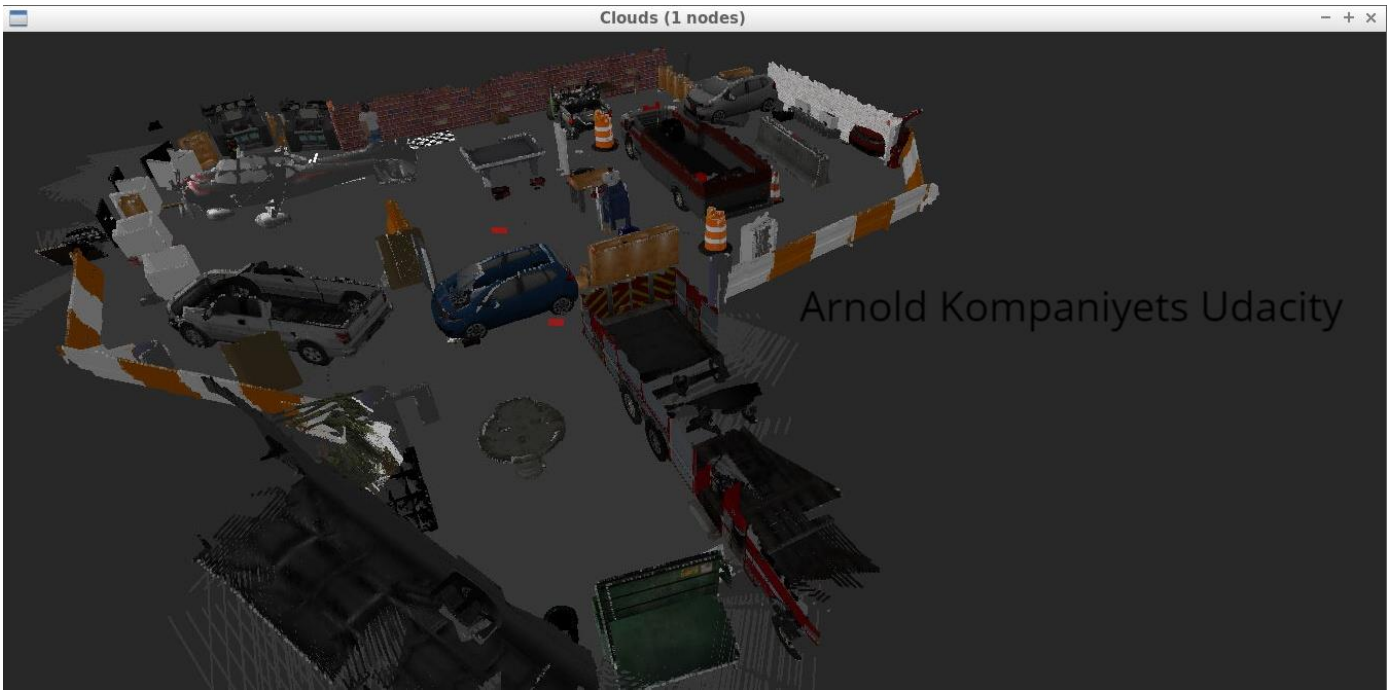



Mapping_1.rviz* - RViz
cus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Arnold Kompaniyets Udacity


Mapping_1.rviz* - RViz
us Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Arnold Kompaniyets Udacity



After reconstructing the above information in the RTAB-Map database viewer, the following map was made:



Discussion:

In mapping both worlds shown above, the greatest amount of time was spent in fine-tuning the few mentioned parameters of the mapping launch file. Regardless of the loop closure technique chosen, the key seemed to be in the minimum number of inliers allowed before initiating loop closure. If the parameter was set to require such a high number of minimum features that even nearly identical images would not meet such intense standards, no loop closures would be done (which was actually desired when sensors function perfectly). On the other hand, if the minimum number of features was set too low, then loop closure would be initiated with images that looked even slightly similar, drawing unnecessary connections between far-away locations or creating shadow images of various objects, simply due to slightly varying paths being connected as one.

When considering the loop closure method specifically, a few were chosen at random. The default method of "SURF" proved to be too erratic in its nature (initiating loop closure rapidly for certain features but much slower for others). The FAST/FREAK and GFTT/ORB methods proved to be the most consistent in feature identification, although quite different in their sensitivities to minimum features for loop closure. The FAST/FREAK method was much less sensitive, so the minimum number of inliers was lowered to 10 when mapping the kitchen dining world. Even then, only three global loop closures were done. The GFTT/ORB method was far more sensitive to features, initiating loop closure much more quickly. Therefore, when mapping the kitchen dining world, the minimum number of inliers had to be raised to 50 in

order to prevent excessive loop closure and overlapping. Even so, just a few laps around the world with this mapping method resulted in 30 global loop closures.

After identifying the two methods that seemed to function most consistently for all objects used, it largely became a matter of setting the most appropriate number of minimum inliers. It was quite interesting to discover just how detrimental global loop closure proved to be in this specific project. In the lessons preceding the project, loop detection and closure was illustrated as the one, sure-fire way to eliminate overlapping and shadow-objects (such as a single chair appearing as three overlapping chairs). With the highly accurate sensors used in simulation, loop closure proved entirely unnecessary and was actually the source of object overlapping. If not for the project requirement of having at least three global loop closures, both worlds would have been mapped more accurately without any global loop closures. Of course, in the real world, where odometry data can be highly error-prone, loop closure would be very efficient, just not very much so in this project.

The second largest intake of time proved to be in making 'tf' function properly. It was initially very confusing to have the RGB-D data being recorded from the front but projected straight up (and at an angle too). There was no immediately seen point of trouble and so it took a bit of time to discover the source of this dilemma. Even after this discovery, there was no source found to identify the actual rotational changes in perspective between URDF and Gazebo, so various changes had to be implemented before the correct rotation of frames was made.

Besides the issue of unnecessary loop closure and time commitment in arranging reference frames, no other significant issues arose in this particular

project. The teleop node ran without problems, since the default /cmd_vel topic was the one implemented in this project (requiring no change to the teleop Python file), and the mapping node initiated without trouble once all of the topics were properly named.

Between the two worlds, performance of RTAB-Map seemed fairly consistent. As mentioned above, the minimum number of inliers for loop closure did need to be changed (since the custom map was much more feature rich), but RTAB-Map was able to, nonetheless, generate an accurate map of both worlds. The one other point of mention was the usefulness of the loop closure constraint. When using ICP in the kitchen dining map, the added local loop closures were not at all helpful; in fact, using this loop closure constraint caused the mapped path to very frequently rotate about the Z-axis. This rotation kept the derived map from being clear and instead led to excessive overlapping and shadow-objects. With the custom map, though, using ICP proved to be the only means to keep unnecessary global loop closures from happening. The total path taken would fluctuate ever-so-slightly around the Z-axis, but the significantly larger size of the overall map and path taken kept this from being an issue.

Conclusion/ Future Work:

Using RTAB-Map in a simulated environment proved to be relatively simple and effective, once the necessary parameters were tuned appropriately. As enjoyable as it was to successfully and accurately map an environment in Gazebo, the real challenge and feeling of success would derive from correctly mapping the real world. Having a strong interest in the field of medical/surgical

robotics, it became quickly apparent that successful use of RTAB-Map could prove very useful in mapping the human body. While surgical robots are currently still almost entirely controlled by the surgeon, it is certainly the hope of the future to have robots capable of traversing the human body on their own, independently carrying out desired actions. The benefit of carrying out surgical procedures outside the OR and without the need for direct supervision of an entire medical team would be enormous, but this would only be possible with very accurate mapping; after all, if a surgical robot needs to properly map the outline of the aorta and inferior vena cava in order to operate in the abdomen, mapping would need to be accurate down to the millimeter (if not desiring to risk significant internal bleeding).

In determining the sensors needed for independently-functioning surgical robots, numerous options would be available. Of course, using RGB-D cameras would be best for the purposes of collecting data to be viewed later by the medical team, but this may not be best for the most accurate mapping, especially since the robot's sensors would often be in a fluid environment. Laser scanners, or perhaps even sound-based or IR sensors, could prove to be much more beneficial in mapping the human body.

Other than the issue of sensor type, a larger issue of surgical robot mapping would simply be in regard to size. Besides trying to fit the sensors themselves onto a robot capable of compactly fitting inside the human body, an even larger issue would be in fitting a CPU (or GPU) powerful enough to carry out the process of data interpretation. Of course, fitting all of this into such a small package doesn't seem possible with current technological capability, meaning that most of the robot would currently still need to be outside the

body. With time, though, it is definitely the expectation that a robot capable of successful mapping would be small enough to enter the human body without the need to break any of the body's major barriers or membranes, eliminating the need for extensive sterilization and the OR. As an intermediate, it is certainly possible to at least have sensors small enough to fit on the surgical robot, with the derived sensory data transferred to an external computer via Bluetooth.

All in all, the advancement of medical robotics is a rapidly-progressing and extensive field, but mapping is sure to play an enormous role in many, if not most, applications in this area of robotics. The human body is a constantly changing and varied organism, different from one to another and cluttered with miniscule components that significantly affect the body as a whole. If the field of robotics hopes to take a significant place in medicine, robots will need to be capable of mapping any part of the human body without error, down to fractions of a millimeter. Maybe this isn't quite a possibility yet, but with hard work and perseverance, this will surely be reality in just a few years' time.