RoboND RTAB - Map My World

Chris Rowe

Abstract—A ROS package is created for a mobile robot using a Hokuyo Laser Range Finder and a Kinect RGB-D Camera to apply the Real Time Appearance Based Mapping (RTAB) SLAM algorithm. Two environments are considered, one from the Gazebo model database and one designed from scratch. Using the RTAB-Map Database Viewer, 2D Occupancy Grid Maps and 3D Point Clouds are exported.

Index Terms—Udacity, SLAM, GraphSLAM, RTAB-Map

1 Introduction

Localization is the process by which a robot determines its local position within a known map. Given a map of the environment, the robot can use odometry and spatial sensory data to match the current local surroundings to an area of the provided map. Conversely, mapping is the process by which a robot uses perception of the local environment along a known path to generate a map. Simply, each process requires the output of the other to work. In many real world applications, neither is known. Subsequently, for a robot to be effective in the real world, it must be able to find a way around this causality dilemma. The solution is to perform both at the same time. This is known as Simultaneous Localization And Mapping (SLAM).

2 Background

Technically, in a static environment where the map is known, a robot would only require localization in order to navigate. In the case of robot vacuums, if we provided a schematic of the house including all the furniture, it should be able to easily localize itself based on that map. And this would work. But it would only work in the environment that the map was made from. If anything were to change, like a couch was repositioned, the robot, having no mapping capability, would be unable to navigate this new environment.

Implementing either localization or mapping individually is a difficult problem. So to perform them together drastically increases the problem's complexity. Estimating an environment with known poses and non-noisy measurements is called Mapping with Known Poses. At each pose, individual measurements at that time can be used to determine, rather than estimate the map. But because real data is noisy, the measurements become Gaussian, which exponentially increases the number of calculations.

A major difficulty in mapping is the large size of the hypothesis space, which is the space that contains all possible maps. Because the space is continuous - a spectrum of values - rather than discrete - finite values - the space is highly dimensional. However, the space can be represented discretely by approximating its value. This representation is known as an occupancy grid map, which divides the space into a grid consisting of cells. Each cell is a spatial representation of the physical area, which can be classified as occupied, free, or unknown. Thus, a high dimensional problem now becomes a set of one dimensional problems. But this dramatic reduction in complexity concurrently reduces its accuracy and range of applicability.

2.1 SLAM

Simultaneous Localization and Mapping is the most fundamental problem of robotics: mapping without known poses. Given measurements y, and commands u the goal is to estimate the combination of poses x, and map m. There are two variants of the SLAM problem.

The Online SLAM problem solves for only the instantaneous pose at the current step in time, x_t. The pose is independent from previous measurement and controls. This is represented mathematically as the posterior probability distribution

$$P(x_t, m|y_{1:t}, u_{1:t})$$

The Full SLAM problem solves for the entire path up until the current step in time $x_{1:t}$. This is represented mathematically as posterior probability distribution

$$P(x_{1:t}, m|y_{1:t}, u_{1:t})$$

2.2 GraphSLAM

GraphSLAM is an algorithm which solves the full SLAM problem, retrieving both the entire path and map. It does so with a graphical approach. There are two types of nodes and two types of soft constraints. Triangular nodes represent poses of the robot. Motion constraints connect two poses with a solid line. Star nodes represent features from the environment. Measurement constraints connect a feature and a pose via a dashed line.

Soft constraints can be visualized as a spring between two masses. In equilibrium, the forces on either mass is equal. The term soft represents the flexibility of the spring that is dependent upon the node configuration. When more nodes are connected, each spring will pull their respective masses. What we desire is the optimal configuration of the masses in which all forces are at their minimum, and all springs are relaxed. This is solved as a global graph optimization problem. In the analogy, the optimal configuration represents the most likely estimation of the map.

2.3 RTAB Mapping

Real Time Appearance Based mapping is a variation of GraphSLAM. RTAB uses RGB-D cameras to capture image and depth data concurrently, which is used to identify features in the environment. When encountering a match of features between two poses, loop closure occurs, which greatly increases the accuracy of the map. Otherwise, when similar features are detected at different poses and loop closure is not used, that location is identified as new, rather than one previously encountered, thus creating duplicates of the same area. RTAB is optimized for mapping large environments and due to its memory management, can perform loop closure in real time, rather than during post-processing. The resulting maps are occupancy grids and 3D point clouds.

3 ROS Package

3.1 Folders

`images` Contains screenshots of the worlds, mapping process, transform tree, node map, etc.
`config` Contains the configuration file for RViz.
`misc` Contains non-critical files used during testing.
`launch` Contains the launch files that start each node.
`models` Contains custom Gazebo model, and meshes for the Kinect, Hokuyo and green shell.
`urdf` Contains the XACRO and Gazebo files which define the physics and visualization in Gazebo.
`worlds` Contains the Gazebo world files for the given world and the custom world.
`src` Contains scripts `rtab_run` which opens and run each launch file in its own terminal, and `teleop` which enables keyboard teleoperation. Also includes the package `point_cloud_io` that can be used to visualize point clouds.

3.2 Robot Model

Upgraded from the previous project, **shelly**, is defined in her XACRO file `shelly.xacro`, in the **urdf** folder. The chassis is cylindrical with radius 0.1m and height 0.4m. The cylinder is oriented with a pitch angle of $-\pi/2$ (-1.5707) radians. The center of gravity was simplified into that of a homogeneous cylinder. Two centered casters, one in front and the other in the back allow the chassis to be balanced. **shelly** is differentially actuated by two centered wheels on either side.

A Kinect RGB-D camera is attached on the front face of the chassis, enabling depth perception in addition to regular RGB image data. A Hokuyo Laser Range-Finder is attached at the highest point of the chassis to ensure an unobstructed field of view for the lasers. Unlike the chassis, the two sensors are visualized using a mesh, both of which are in the **/models** folder.

The actuation of the wheels is done through a Gazebo plugin via the topic `/cmd_vel`, and odometry is published to the `/odom` topic. Actual RGB-D and LaserScan data are retrieved from Gazebo using their respective plugins. All three plugins are defined in the accompanying `shelly.gazebo` file in the same folder.

Note that, the physical characteristics of **shelly** are that of the cylindrical chassis in Fig: 1. As shown in the superimposition, Fig: 2, nothing physical (mass, collision or inertia, have changed. Visually, in Fig: 3, the cylindrical chassis and components were simply made invisible in the Gazebo simulation, and visually replaced with a Green Shell from Mario Kart. Hence, the name **shelly**.

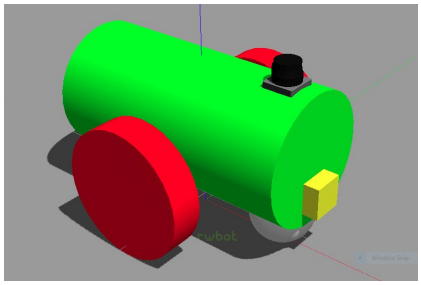

Fig. 1: shelly's Physical Collision

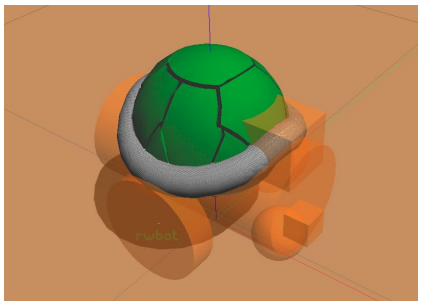

Fig. 2: shelly's Visual Superimposed Over Collision. The visual representation has changed, but **the collision has not.**

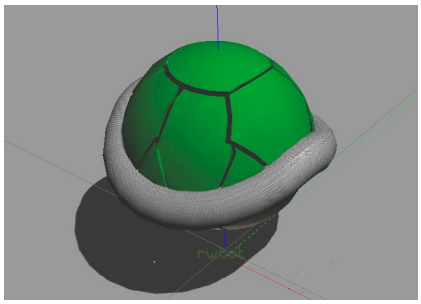

Fig. 3: shelly - Self Driving Mario Kart Shell [4]

3.3 Launch Files

For modularity, each node was given its own launch file.

`rviz.launch` Launches RViz configured with `rtab.rviz`.
`teleop.launch` Launches a node that enables keyboard control using the provided `teleop` script.
`robot_description.launch` Launches the nodes that spawn **shelly** in Gazebo, `joint_state_publisher` and `robot_state_publisher` nodes which handle the transforms.
`world.launch` Launches the kitchen environment.
`rw.launch` Launches my Gazebo environment `rw.world`.

`mapping.launch` Launches the `rtabmap` node.

`rtabmapviz.launch` Launches a special RViz node that acts as a GUI during mapping, providing views of the generating point cloud, along with play/pause and export functionalities, and more.

4 World Creation

When considering the layout of the custom environment, each part needed to be designed to fulfill the requirement of the RTAB mapping method. Namely, the environment needed to be abundant in unique features. Since the success of RTAB mapping was dependent upon loop closures, the environment needed to be easily traversable. This would enable the robot to easily complete laps around the environment. Thus, the idea for a track was born.

First, a perimeter needed to be established. Using the Gazebo world editor, four walls were arranged in a square. To compare, the arrangement was duplicated twice with increasing lengths. The 20m, 15m and 10m length rooms were compared and ultimately the room 10m in length was chosen as the most practical to decorate with various features.

With the basic structure established, it was time to focus on adding features. The default colors of the wall were too plain. Discerning the features using only the color of the walls would be near impossible. To increase the amount of features, the default material was replaced by a customized material having a texture. Also, to ensure each wall was unique, a different material was used for each wall.

This was done by creating Gazebo `.material` files. These are scripts that need two things. First, a pattern in the form of a `PNG` image. Second, is the scale of the pattern. This determines how the pattern will be arranged on a given surface. Each wall was then assigned one of the custom materials. For all four walls, see figures 19 and 20.

Fig. 4: Wall with tech-themed material.

Next, the room was decorated. Among a selection of models in the Gazebo database, The most unique were picked and spread out along each of the walls. Four of the large models were each placed in a corner. Then, the rest of the models would be spread along the wall to fill up the space between corners.

Two models used in camera calibration were found and were put along the North and South walls.

Finally, to make it a real *track*, three models of barriers and an ATV were used to make a square in the middle, which would function as the inner walls of the track.

Fig. 5: Tech themed wall with added models.

Fig. 6: Calibration type models added to increase features.

Fig. 7: Inner boundary made from barriers and an ATV.

Fig. 8: The completed `rw.world`.

5 RESULTS

The robot was navigated using the keyboard around each world. Each run consisted of at least three laps around the environment. Upon completion, the RTAB-Map Database Viewer was used to export the generated occupancy grid map and point cloud. The rtab database and point cloud files can be found in the github project repository [1].

5.1 kitchen_dining.world

The traversal method that resulted in the most accurate map came from looping around each room once per lap, somewhat like an infinity symbol. This produced more loop closures compared to mapping each room completely before proceeding to the other. A total of **48 global loop closures** were found. The areas with the refrigerator and couch are where most loop closures were formed. See figures 21, 22 for examples of detected loop closures.

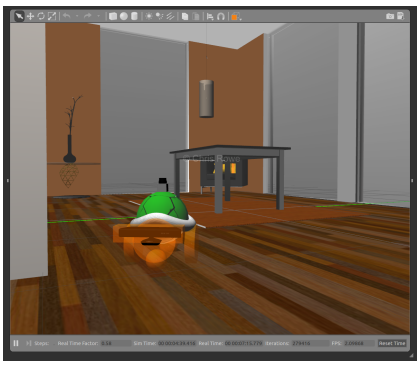

Fig. 9: **shelly** while mapping `kitchen_dining.world`.

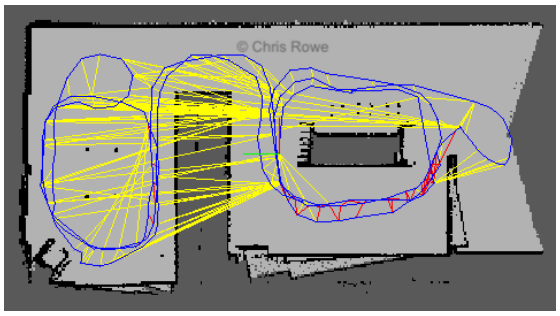

Fig. 10: Alternating room loop path used for mapping.

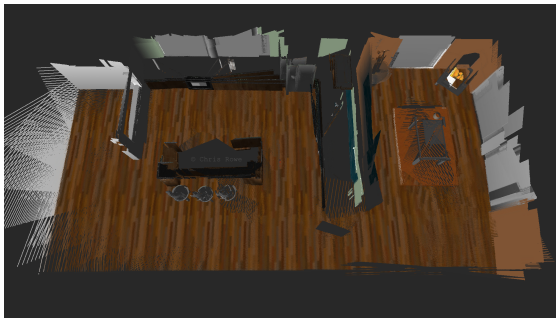

Fig. 11: Point cloud of `kitchen_dining.world`.

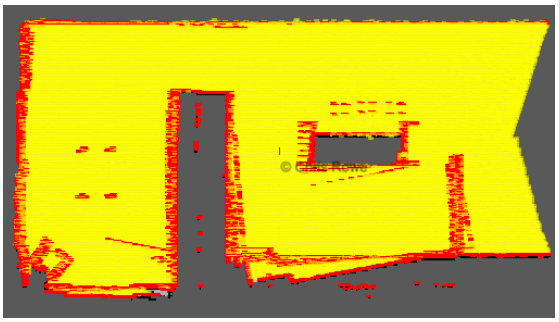

Fig. 12: Occupancy grid map of `kitchen_dining.world`.

5.2 rw.world

Shown in Fig: 13 is a Gazebo screen shot taken during a mapping run. The environment was traversed in laps in alternating between clockwise and anti-clockwise. In addition to providing more loop closures, the resolution of the various features in the environment increased with each lap. A total of **130 global loop closures** were found. It was found that the majority of loop closures tended towards features found in the corner areas of the environment. See figures 23, 24 for examples of detected loop closures.

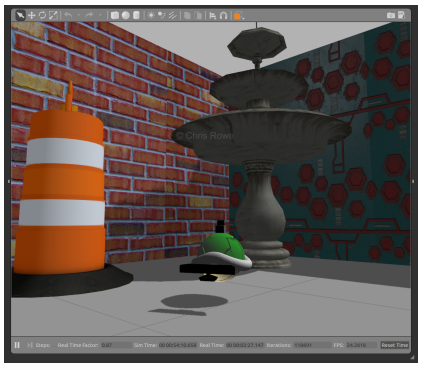

Fig. 13: **shelly** while mapping `rw.world`.

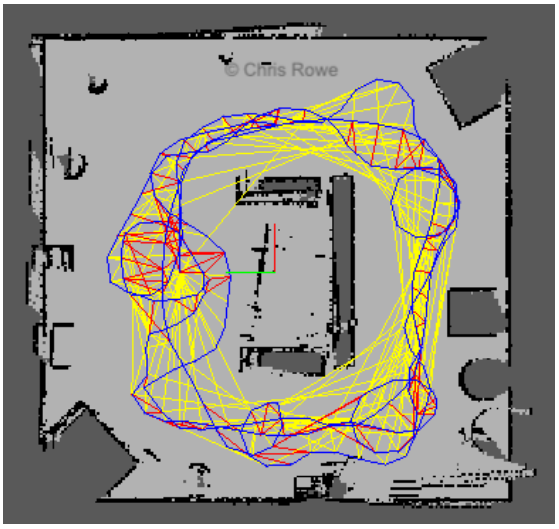

Fig. 14: Alternating between clockwise and anti-clockwise loops.

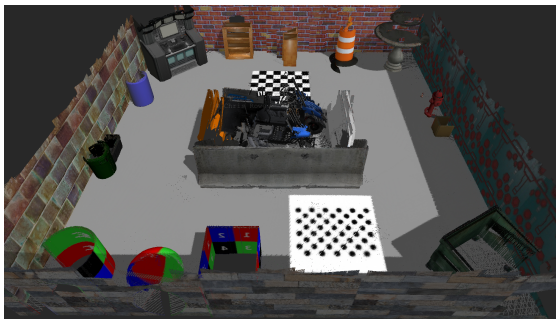

Fig. 15: Point cloud of `rw.world`.

Fig. 16: Occupancy grid map of `rw.world`.

6 Discussion

Both worlds were successfully mapped and produced mostly accurate 2D occupancy grid maps. There are areas where some of the 2D passes were out of sync. For example in Fig: 12, there is a duplicate of the lower wall. Also, in Fig: 16, the shapes in the lower left corner were duplicated.

In the custom world, it was difficult to capture all sides of each object. Loops were done by alternating between clockwise and anti-clockwise. Thus, most of each object, except for the backs, was captured. But in switching between loop directions, the robot needed to make tight turns. It was found that turning in place causes the map to become completely distorted.

After browsing through the github wiki for the ROS package [3], it was found that the distortion can be avoided by changing the `Reg/Strategy` parameter of the `rtabmap` node to **1**. This parameter determines the constraints for loop closures. In mapping the custom world, switching from `visual` (0) to `iterative closest point` (1) kept the mapping stable when turning in place.

Overall, the 3D point clouds generated were noticeably close to the actual environment. Except for the top surfaces of higher objects, and the backs of objects against the wall, the point clouds were quite close to their real counterparts as seen in figures 25, 26, 27 and 28.

7 Future Work

Hardware Deployment

Plans are being made for applying RTAB-Map on a robot based on the Jetson TX2 using the ZED stereo camera. The hallways in my engineering building are quite narrow, so mapping would involve turning in place. It will be interesting to see how that problem manifests itself when real world data is used.

References

[1] *Project Github Repository*, https://github.com/rwbot/RoboND-RTAB-SLAM-Project
[2] Sebastian Thrun, Wolfram Burgard and Dieter Fox, *Probabilistic Robotics*, The MIT Press, 2005
[3] *Introlab RTABMap Wiki*, Iterative Closest Point
[4] *Maze-Man*, Mario Power Ups, 3D Warehouse Sketchup

Appendix

The following is a list of figures provided for reference:

Fig 17 - Transform Tree of the `shelly` robot.

Fig 18 - RQT Node and Topic Map.

Fig 19 - Walls created in Gazebo using custom material.

Fig 20 - Arranging unique objects across the walls.

Fig 21 - Loop Closure Detected from refrigerator and counter features in `kitchen_dining.world`.

Fig 22 - Loop Closure Detected from sofa features in `kitchen_dining.world`.

Fig 23 - Loop Closure Detected from console, brick, and bookshelf features in `rw.world`.

Fig 24 - Loop Closure Detected from features found in the shapes with numbers and colors in `rw.world`.

Fig 25 - Point cloud generated from RTAB-Map Database Viewer of `kitchen_dining.world`.

Fig 26 - Point cloud generated from RTAB-Map Database Viewer of `kitchen_dining.world`.

Fig 27 - Point cloud generated from RTAB-Map Database Viewer of `rw.world`.

Fig 28 - Point cloud generated from RTAB-Map Database Viewer of `rw.world`.

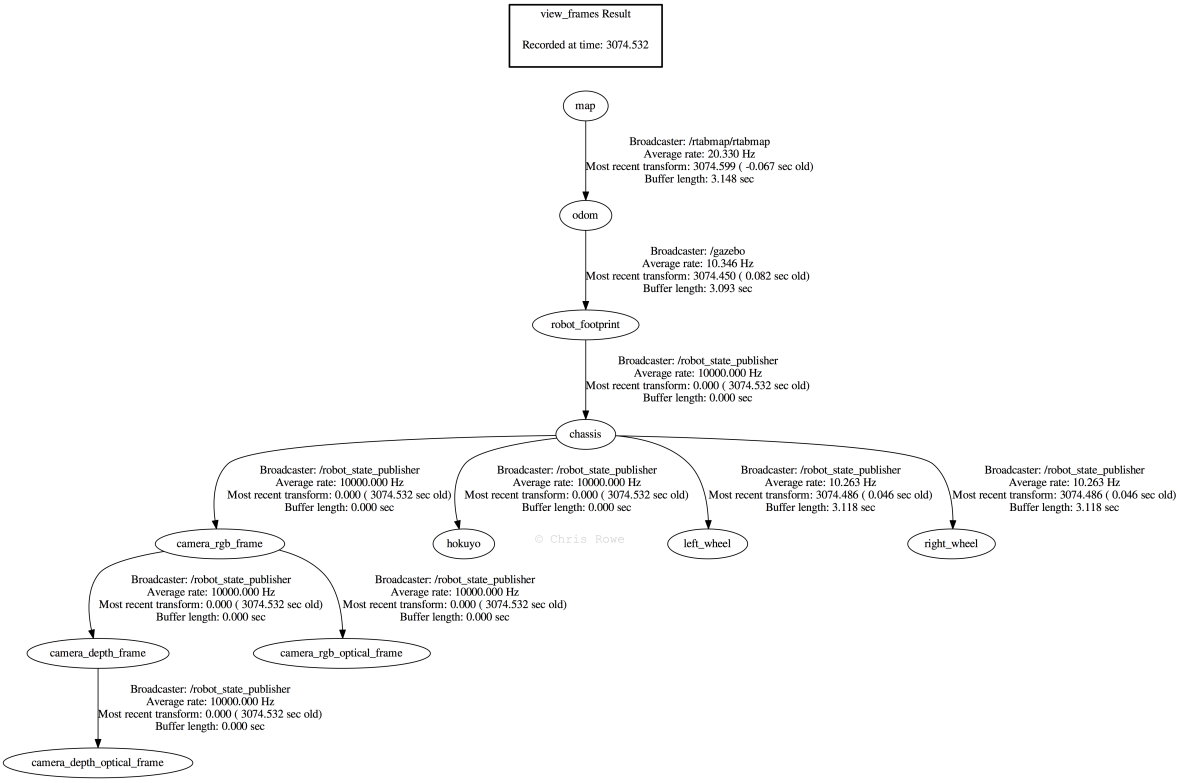

Fig. 17: Transform Tree of the `shelly` robot.

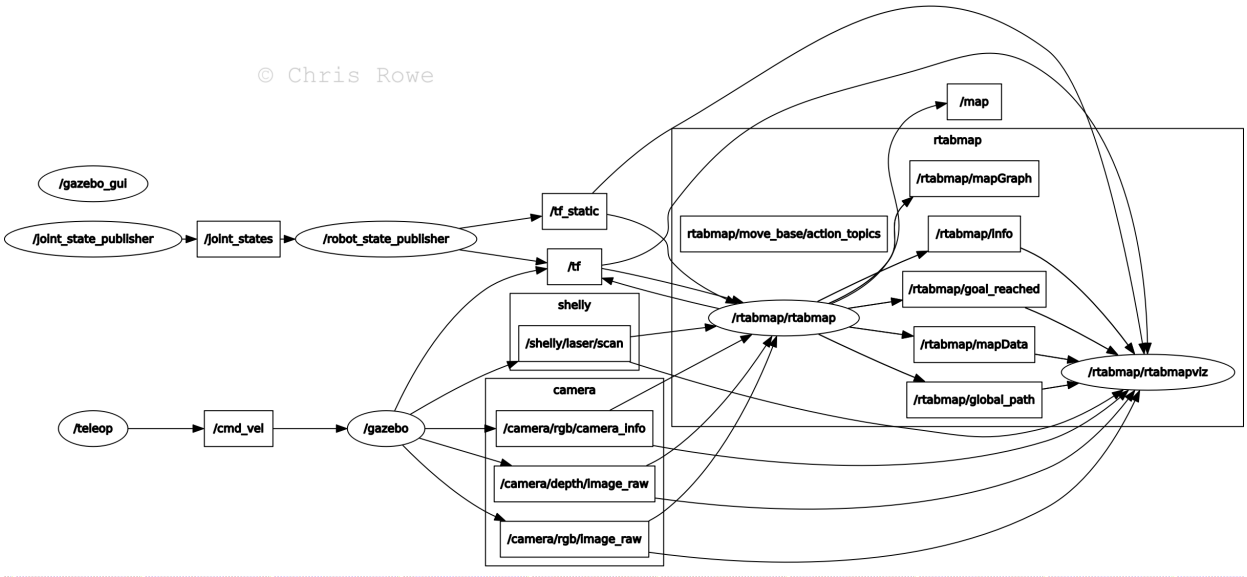

Fig. 18: RQT Node and Topic Map.

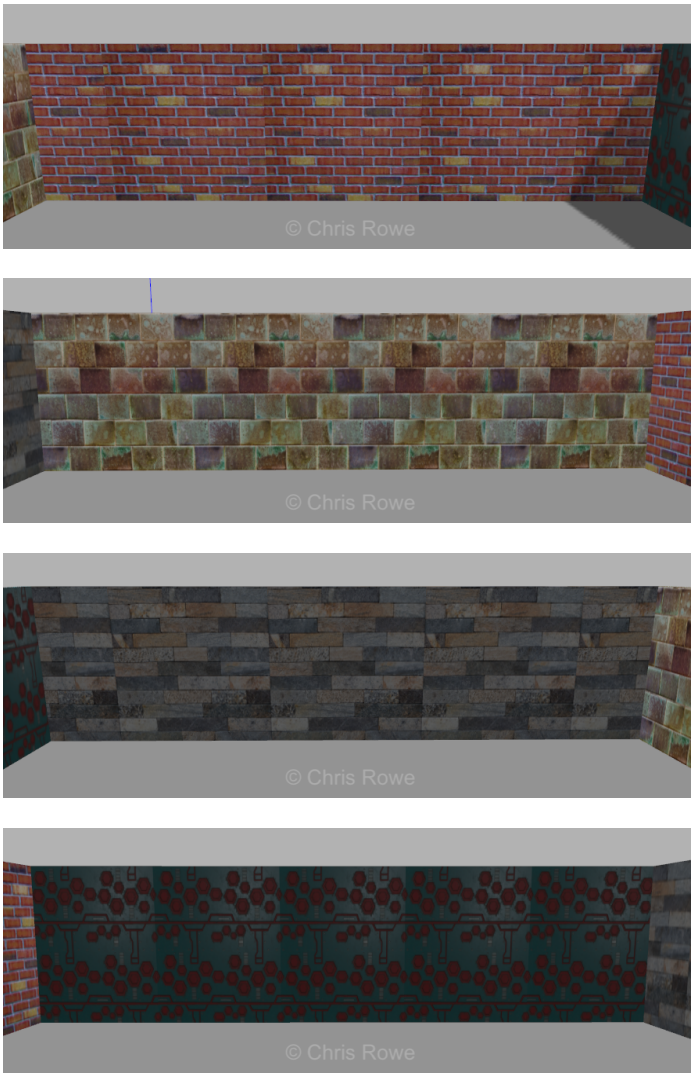

Fig. 19: Walls created in Gazebo using custom material.

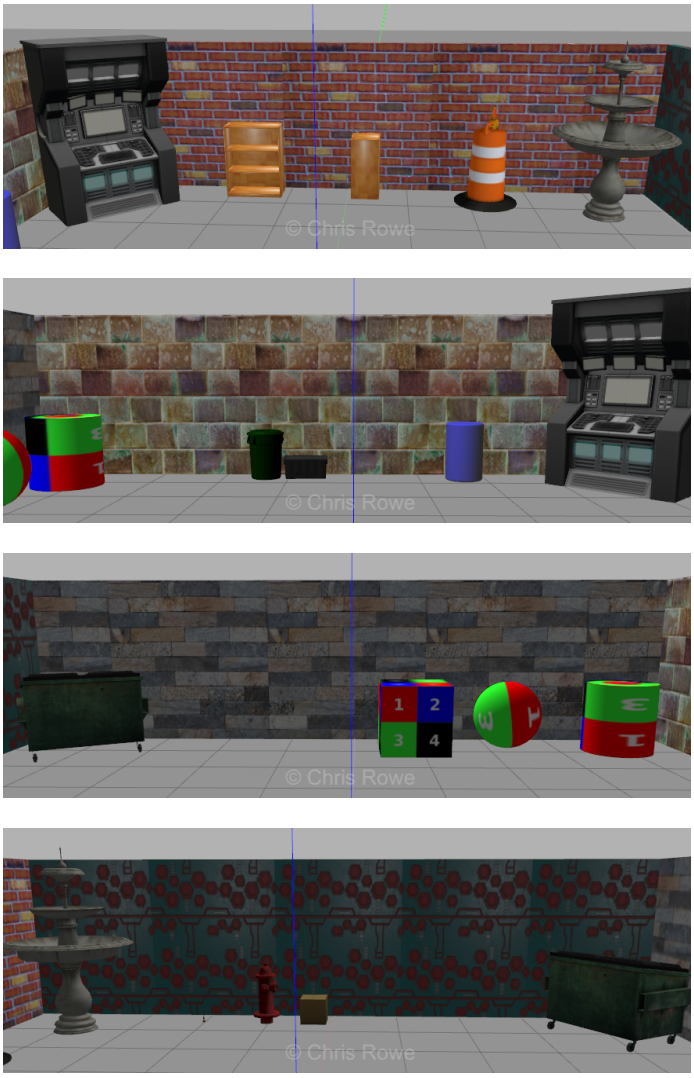

Fig. 20: Arranging unique objects across the walls.

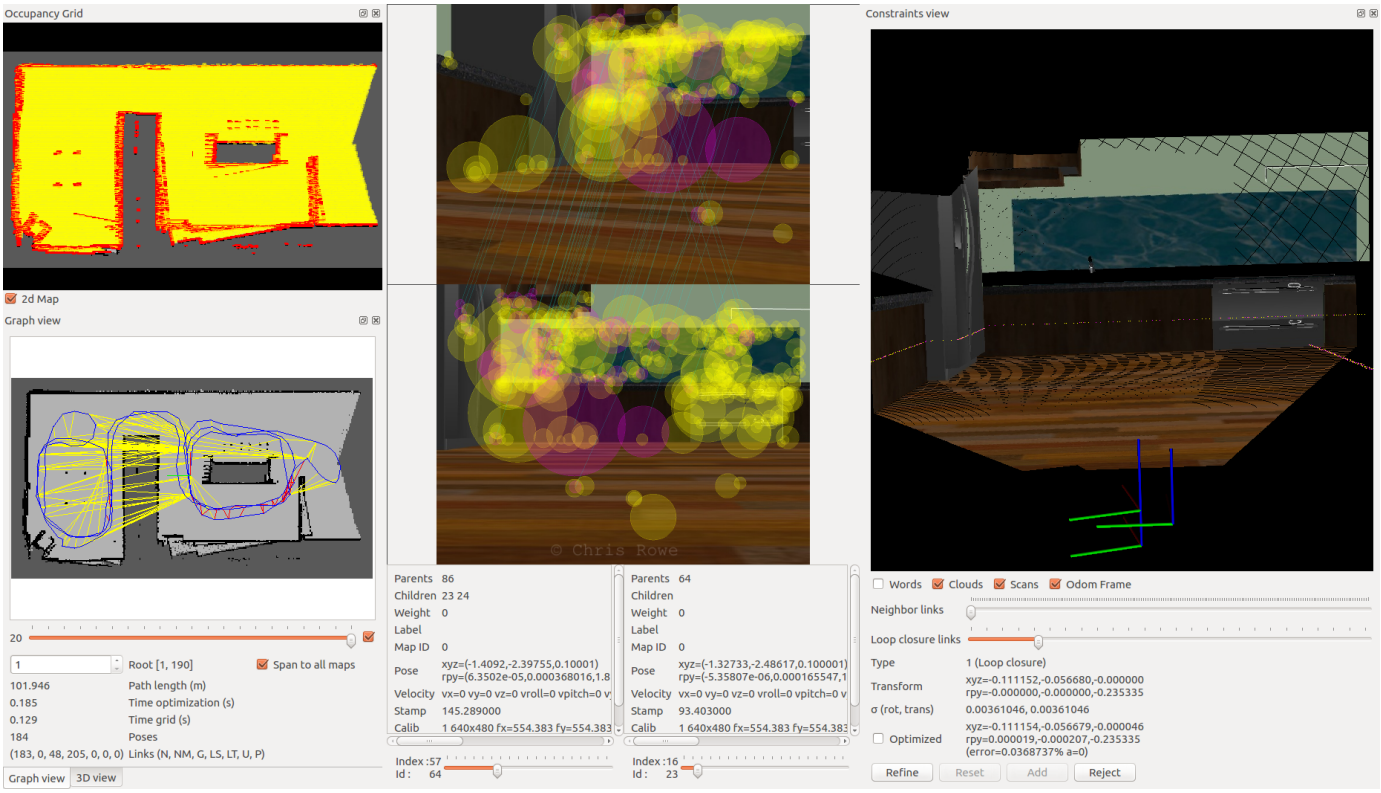

Fig. 21: Loop Closure Detected from refrigerator and counter features in `kitchen_dining.world`.

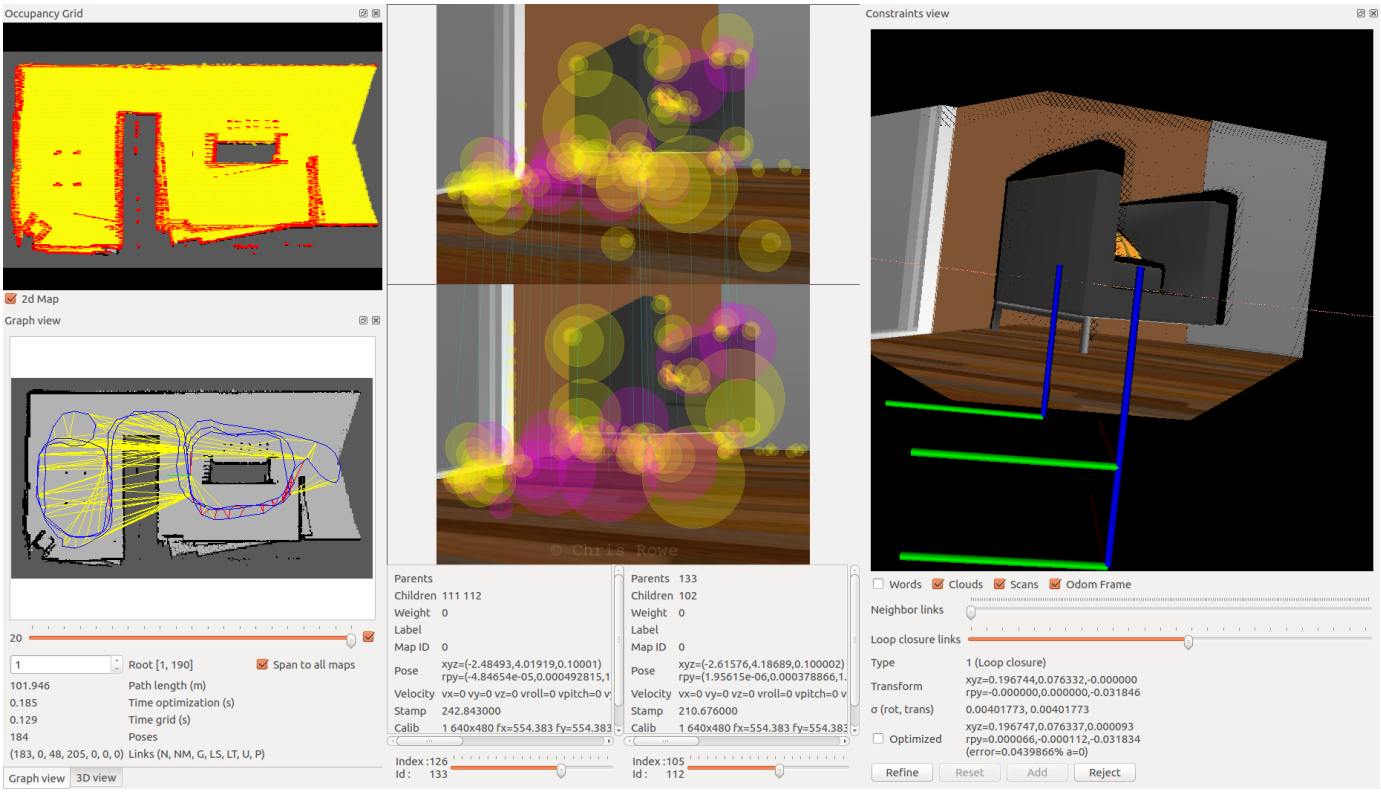

Fig. 22: Loop Closure Detected from sofa features in `kitchen_dining.world`.

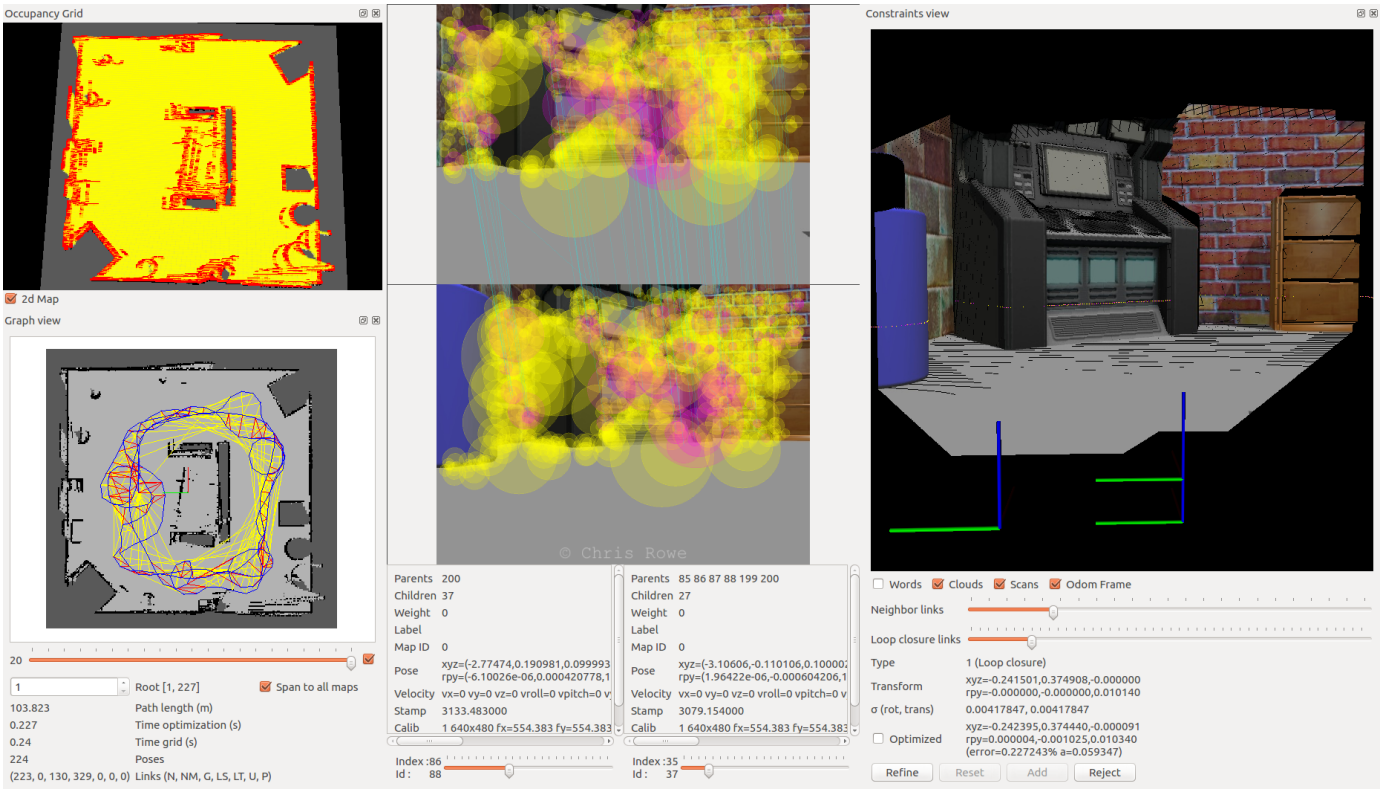

Fig. 23: Loop Closure Detected from console, brick, and bookshelf features in `rw.world`.

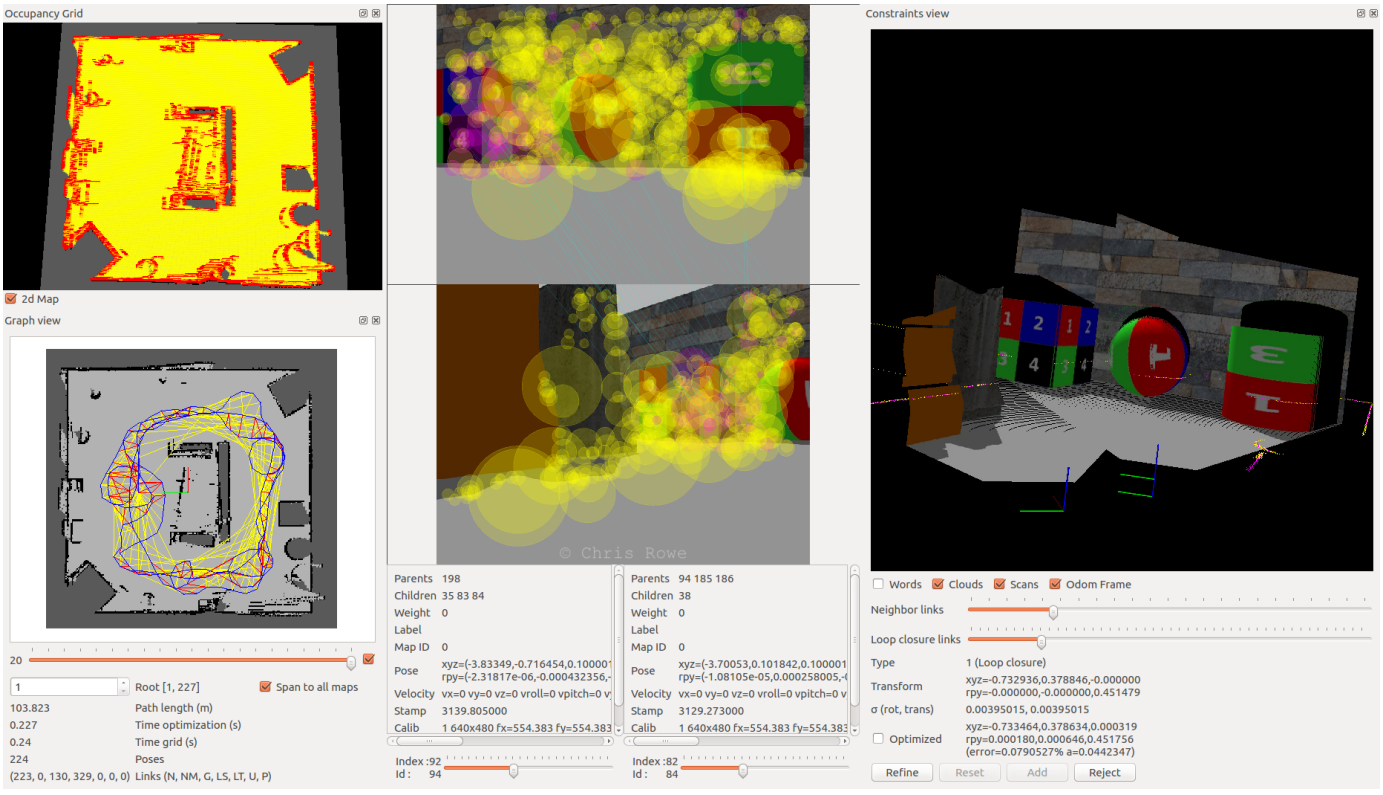

Fig. 24: Loop Closure Detected from features found in the shapes with numbers and colors in `rw.world`.

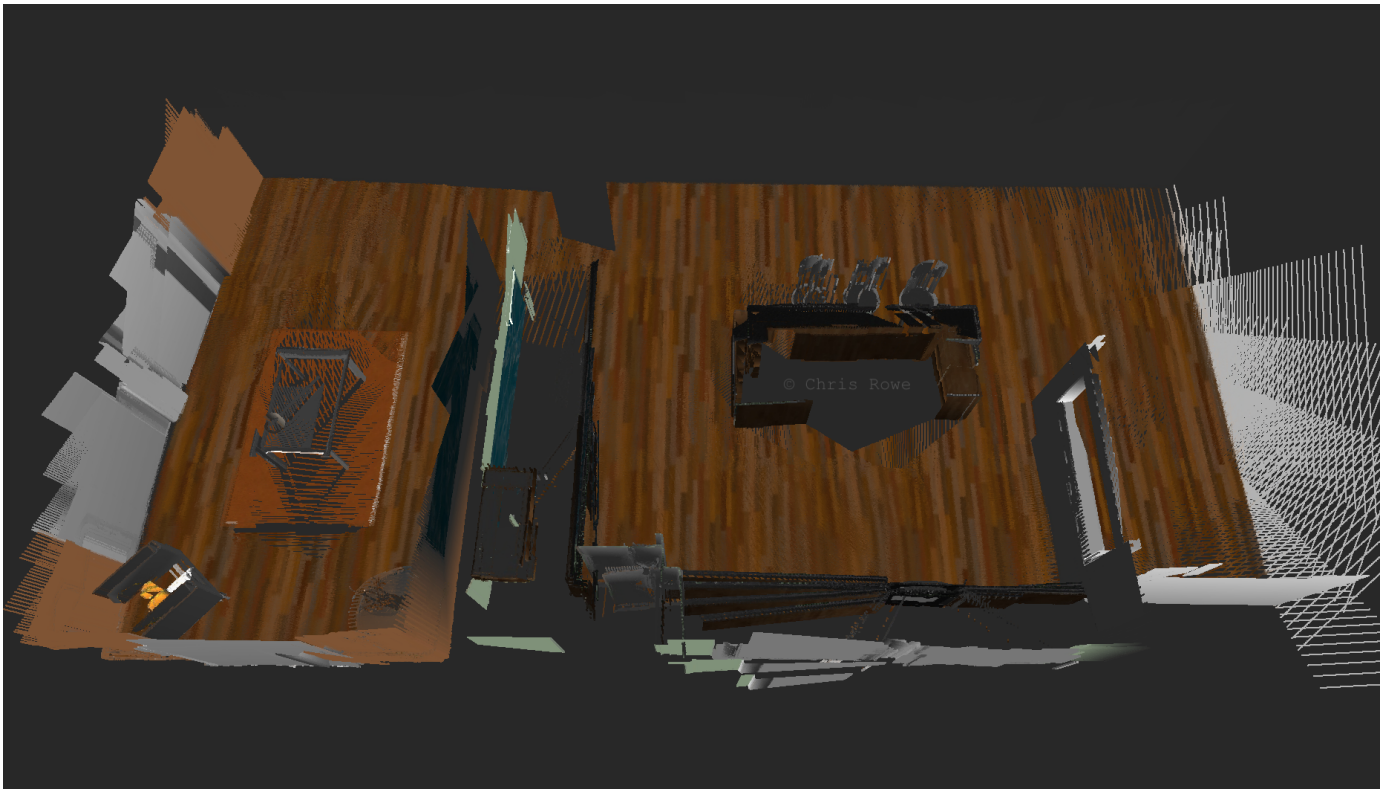

Fig. 25: Point cloud generated from RTAB-Map Database Viewer of `kitchen_dining.world`.

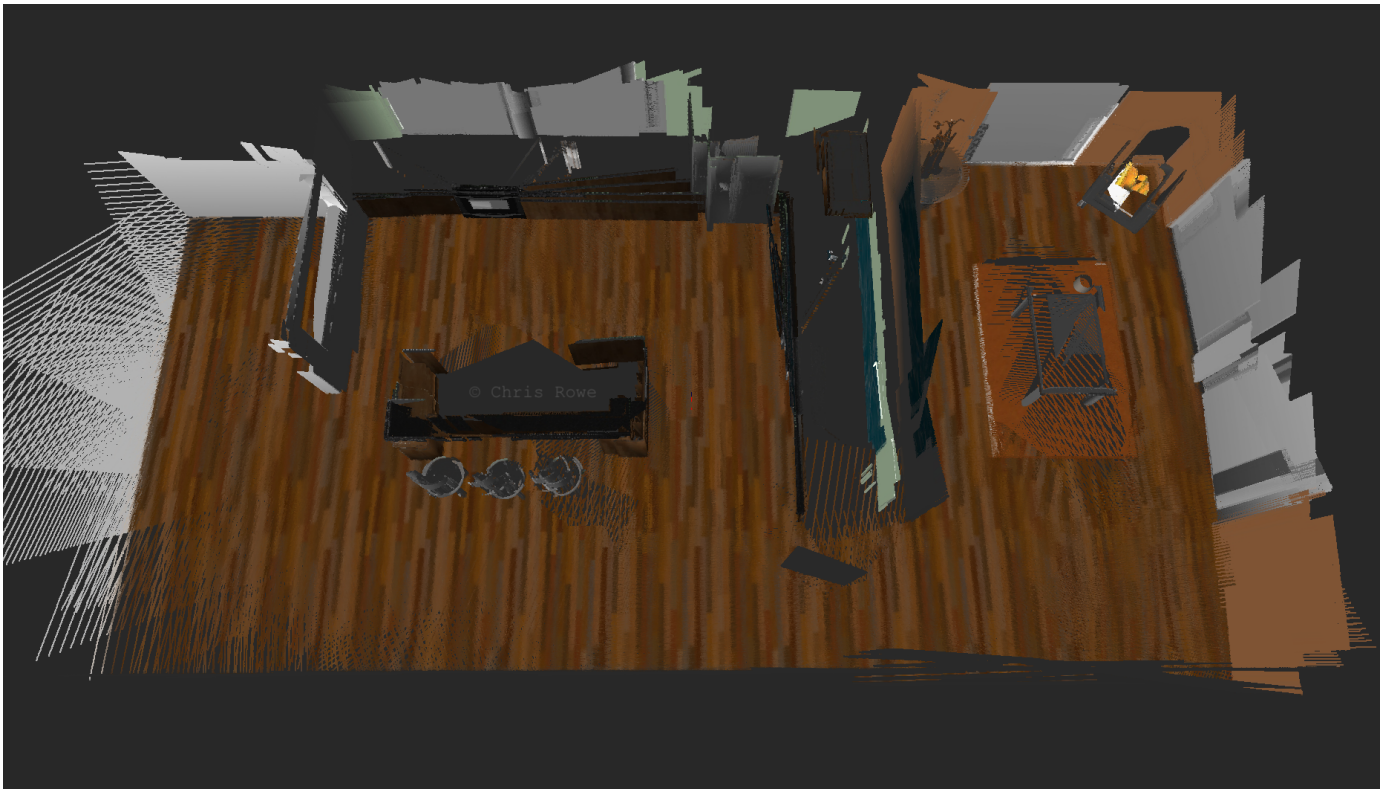

Fig. 26: Point cloud generated from RTAB-Map Database Viewer of `kitchen_dining.world`.

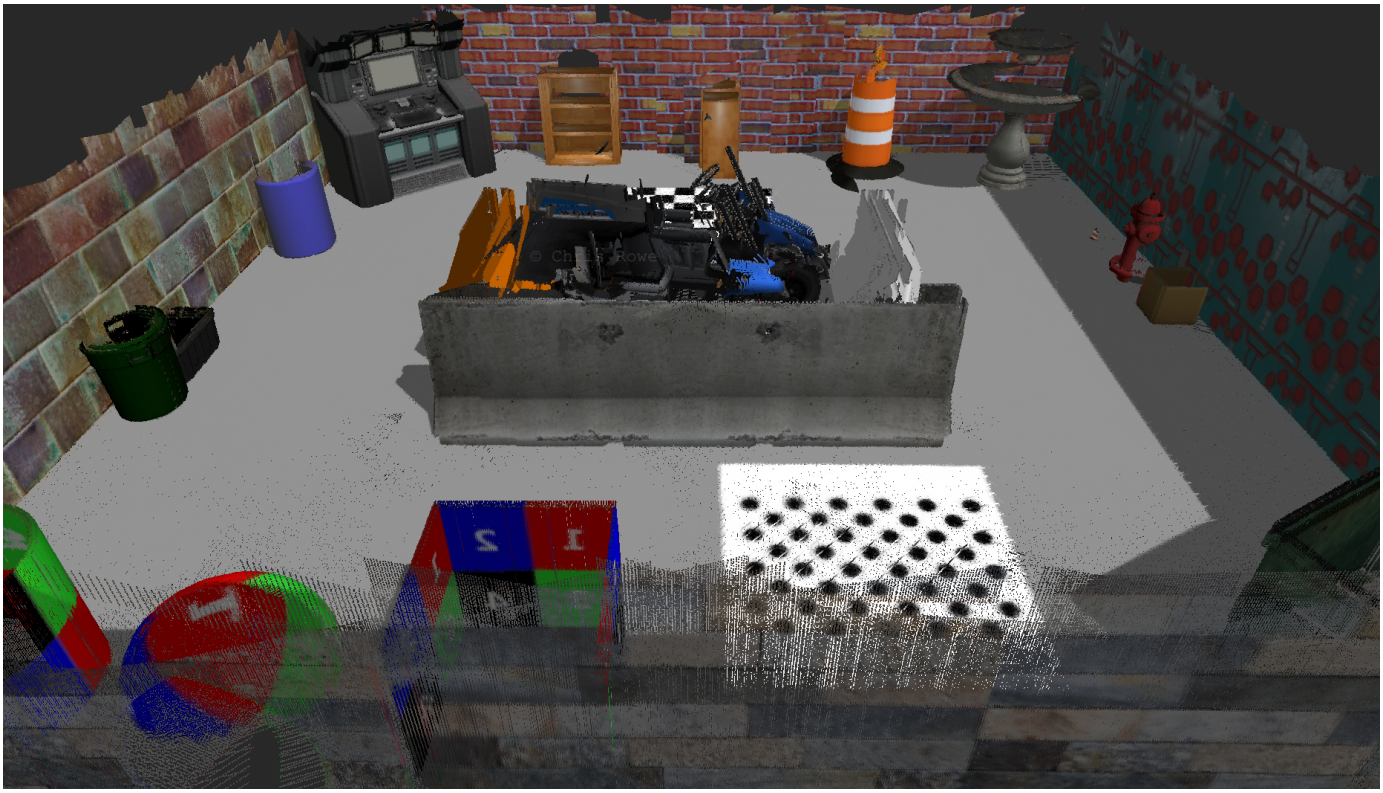

Fig. 27: Point cloud generated from RTAB-Map Database Viewer of `rw.world`.

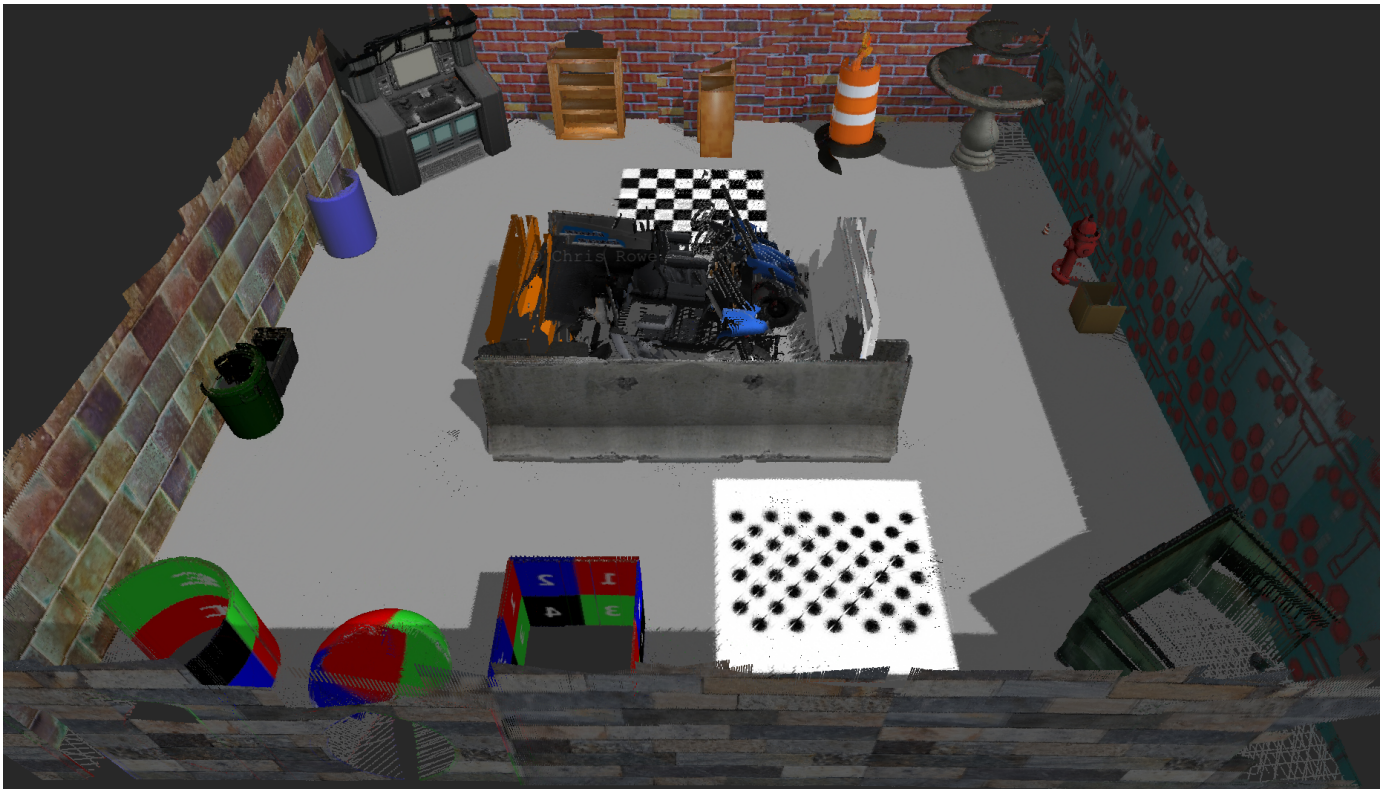

Fig. 28: Point cloud generated from RTAB-Map Database Viewer of `rw.world`.