# Simulating sensors in Gazebo

Wyatt Newman
July, 2015

An impressive and useful capability of Gazebo is simulation of sensors. In this document, we describe how to augment our robot model with a 3-D sensor (an emulated "Kinect" sensor). This sensor provides a "point cloud" of 3-D points, which may be registered with corresponding color information from a color camera. Rviz can be used to display the results of the emulated sensor.

This document borrows from the on-line ROS tutorial here:
http://gazebosim.org/tutorials?tut=ros_gzplugins.

**Virtual sensor, theory of operation:**
The computation required for sensor emulation can conceptualized as follows. The Kinect camera projects a speckle pattern of infrared light, and this pattern is distorted as it is projected onto 3-D surfaces. The resulting structured-light pattern is sensed by an infrared camera, and by interpretation of the distortion, depth can be inferred from the infra-red camera image. Consider pixel (i,j) from the image plane of the infrared camera. Photons reaching this pixel must have originated from some small patch (in the limit, a point) along a line of sight determined by the pixel coordinates and the equivalent pin-hole of the camera focal point. This much geometry implies the azimuth and elevation angles of the light source relative to the image plane frame. If, in addition, distance to the light source can be inferred from the structured-light projection, then we can deduce 3-D coordinates corresponding to illuminatino of the (i,j) pixel of the infrared camera. Repeating for all pixels, we obtain a large number of 3-D points in the "point cloud."

In Gazebo, we define a "world" that includes CAD-modeled solid objects. Each object has a respective coordinate frame, and each such object frame has a known transform relative to the "world" frame. In addition, the simulated sensor has an image-plane frame, and we must be able to define the transform between the world frame and this image frame. If the sensor is mounted on a movable link, then the transform from sensor to world will require kinematic computations that are a function of movable joint angles.

Assuming we are able to specify the sensor frame with respect to the world frame, Gazebo can compute 3-D coordinates corresponding to each pixel in the simulated sensor using line-of-sight computations. Each pixel of the sensor, together with coordinates of the focal point, defines a corresponding ray in space. Tracing along any such ray, one eventually encounters either a solid object capable of light reflection or a sensor-range limit. For the first line-of-sight intersection with a solid, the resulting intersection coordinates correspond to the 3-D point of our simulated 3-D sensor. Repeating for all pixels in the sensor provides a point cloud that emulates a real sensor. This virtual-sensor computation can be repeated at a specified rate—but since this computation is very intensive, one should be careful to specify reasonable update rates for the available computing platform.

Based on the above description, it should be clear that updates of the virtual sensor values would take into account motion of the sensor or of objects in the world. The resulting sensor data can be visualized in Rviz.

**Adding a Kinect sensor to the robot model URDF:** The package minimal_robot_description contains a file "minimal_robot_w_sensor.launch," which is our previous minimal robot model with the addition of a Kinect sensor. First, we include the physical characteristics, including the visual, collision-model and inertial properties.  These are entered (crudely) with the following lines:

```
<!-- TRY ADDING a Kinect sensor: first the visual/collision/inertial box -->
    <link name="kinect_link">
      <collision>
          <origin xyz="0 0 0" rpy="0 0 0"/>
          <geometry>
              <!-- coarse model; a simple box -->
              <box size="0.05 0.2 0.05"/>
          </geometry>
      </collision>

      <visual>
          <origin xyz="0 0 0" rpy="0 0 0" />
          <geometry>
              <box size="0.05 0.2 0.05" />
          </geometry>
          <material name="kinect_gray">
              <color rgba="0.2 0.2 0.2 1.0"/>
          </material>
      </visual>

      <inertial>
          <mass value="0.1" />
          <origin xyz="0 0 0" rpy="0 0 0"/>
          <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001" />
      </inertial>
  </link>
```

Next, we need to specify how this new sensor "link" is connected to a parent link.  In the present example, we connect the sensor to the tip of our movable link, tilted so that it can view the ground when the robot is upright.  This is specified as follows:

```
  <!--now set the position/orientation of the above box: -->
  <joint name="kinect_joint" type="fixed">
      <axis xyz="0 1 0" />
      <!--origin xyz="0.55 0.03 1.585" rpy="0.06 0.95 0"/ -->
      <origin xyz="0.2 0.0 0.95" rpy="0.0 1.2 0"/>
      <parent link="link2"/>
      <child link="kinect_link"/>
  </joint>
<!-- end wsn adding kinect link; -->
```

Next, sensor "plug-in" code is referenced, which performs the high-speed, tedious computations of depth camera emulation.  This is specified in the URDF with the following lines:

```
  <!-- wsn--add kinect sensor emulation-->
<gazebo reference="kinect_link">
    <sensor type="depth" name="openni_camera_camera">
        <always_on>1</always_on>
        <visualize>true</visualize>
        <camera>
            <horizontal_fov>1.047</horizontal_fov>
            <image>
                <width>640</width>
                <height>480</height>
                <format>R8G8B8</format>
            </image>
            <depth_camera>

            </depth_camera>
            <clip>
                <near>0.1</near>
                <far>100</far>
            </clip>
        </camera>
        <plugin name="kinect_controller" filename="libgazebo_ros_openni_kinect.so">
            <alwaysOn>true</alwaysOn>
            <updateRate>10.0</updateRate>
            <cameraName>kinect</cameraName>
            <frameName>kinect_sensor_frame</frameName>
            <imageTopicName>rgb/image_raw</imageTopicName>
            <depthImageTopicName>depth/image_raw</depthImageTopicName>
            <pointCloudTopicName>depth/points</pointCloudTopicName>
            <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>
            <depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTopicName>

            <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>
    </sensor>
</gazebo>
```

The specifications above reference the library "libgazebo_ros_openni_kinect.so", which contains the plug-in code for depth-camera emulation.  Details of the depth camera to be emulated include min and max range, number of horizontal and vertical pixels, field of view, update rate, inclusion of a 24-bit

RGB camera, and distortion coefficients.

Emulation of the depth camera is specified to be with respect to a sensor frame called “kinect_sensor_frame”. The physical placement of the sensor has been specified via the above links and joints, and this determines the field of view of the camera. However, the sensor data is reported with respect to an optical frame, and we need to specify the transform between this optical frame and some known frame (one that has a specified kinematic chain to the world frame). To include this additional information, we need to add a line to the launch file. The launch file for our robot with sensor is “minimal_robot_w_sensor.launch, which contains:

```
<launch>
<param name="robot_description"
textfile="$(find minimal_robot_description)/minimal_robot_w_sensor.urdf"/>

 <!-- wsn: transform for kinect -->
<node pkg="tf" type="static_transform_publisher" name="kinect_calib" args="0 0 0 -0.500 0.500
-0.500 0.500 kinect_link kinect_sensor_frame 10"/>

</launch>
```

The first 2 lines of this launch file cause the robot model to be loaded onto the parameter server (as was done previously). The additional line launches the “static_transform_publisher” node from the “tf” (transform) package. The numerical arguments specify the transform using <x,y,z>, <r,p,y> between the kinect_link frame and kinect_sensor_frame. With this additional information, rviz can compute how to display the sensor data, transformed into world coordinates.

**Running the minimal robot with sensor:**
Running the minimal robot with the kinect sensor, and displaying the results in rviz involves the following steps (which can be started in separate terminals):

```
roscore
rosrun gazebo_ros gazebo
rosrun gazebo_ros spawn_model  -file minimal_robot_w_sensor.urdf -urdf -model one_DOF_robot
rosrun minimal_joint_controller minimal_joint_controller
roslaunch minimal_robot_description minimal_robot_w_sensor.launch
rosrun robot_state_publisher robot_state_publisher
rosrun rviz rviz
```

All of the above can be run in separate terminals, or (more conveniently) using a single launch file as:

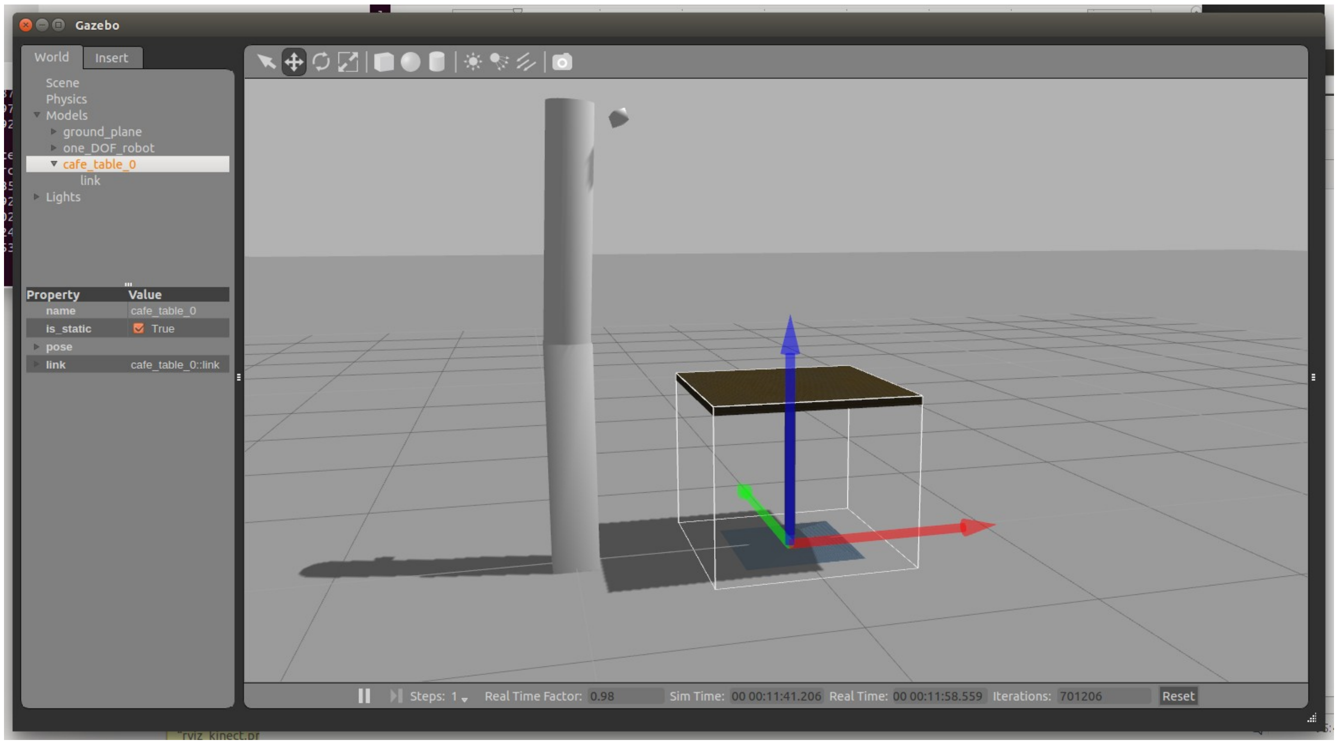
```
roslaunch minimal_robot_description minimal_robot.launch
```

To use the above launch file, minimal_joint_controller.cpp was modified to include a test of:

```
service_ready = ros::service::exists("/gazebo/apply_joint_effort",true);
```

Since the minimal controller might start running before Gazebo offers its services, attempts by the controller to use these services before they are advertised results in the controller node crashing. By testing for the existence of these services, the controller node is tolerant of variations in start-up timing among the various nodes.

With the above nodes running, one can add additional items to the robot's “world” via Gazebo. From “Gazebo”, select the “insert” tab, and a list of existing models will appear. From this list, the model “cafe table” was selected. Gazebo initially places the new model on the ground plane and allows one to slide x/y with a mouse, then click on the desired (rough) location. The location of the model can be refined by selecting the model (upon which the model will be highlighted with a white, wireframe box), then clicking the “translate” icon in the top menu-bar of Gazebo. The scene then looks like the following.

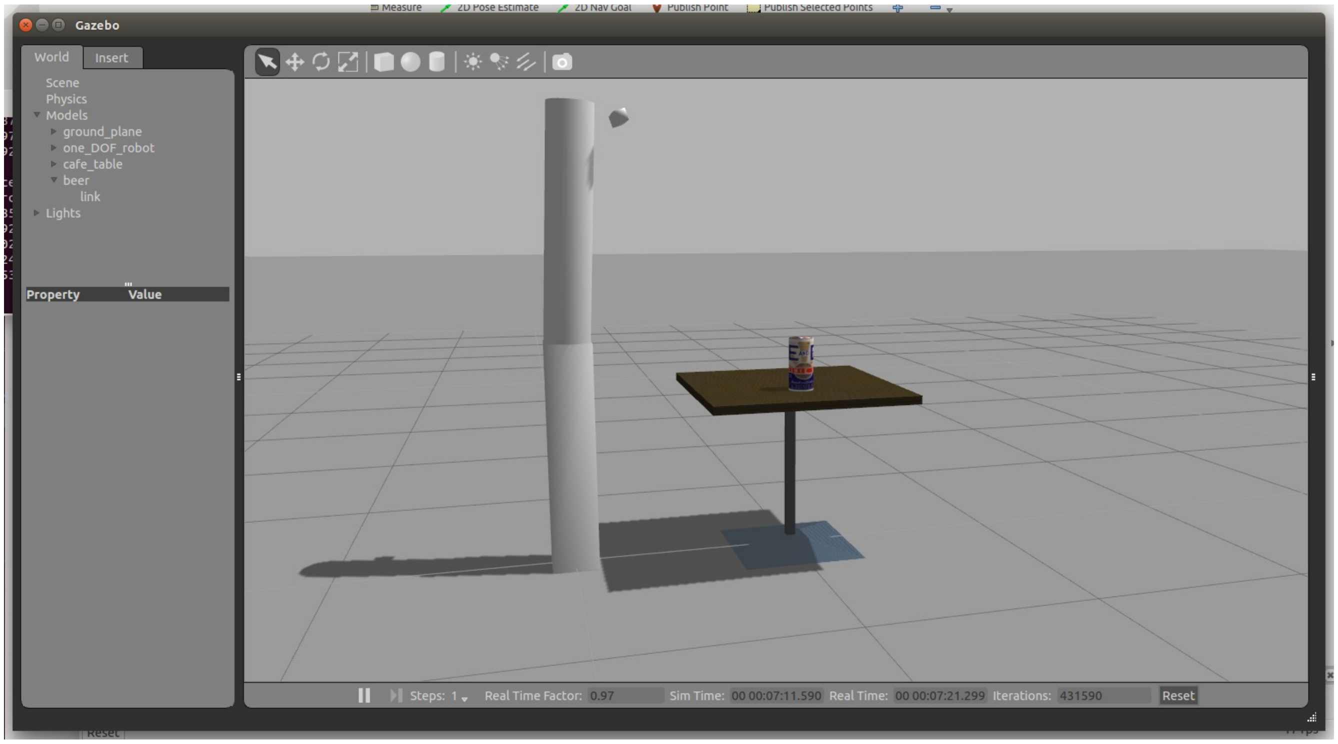


| Property | Value |
|---|---|
| name | cafe_table_0 |
| is_static | ✓ True |
| pose | |
| link | cafe_table_0::link |

```
Steps: 1   Real Time Factor: 0.98   Sim Time: 00 00:11:41.206   Real Time: 00 00:11:58.559   Iterations: 701206
```

The 3 axes associated with the table model offer “handles” with which one can translate the table. By hovering over one of the axes with a mouse, the selected axis will display bolder, and one can click/drag the model along the desired direction. The “rotation mode” icon (next to the translation-mode icon) allows one to rotate the the model about each of 3 axes.

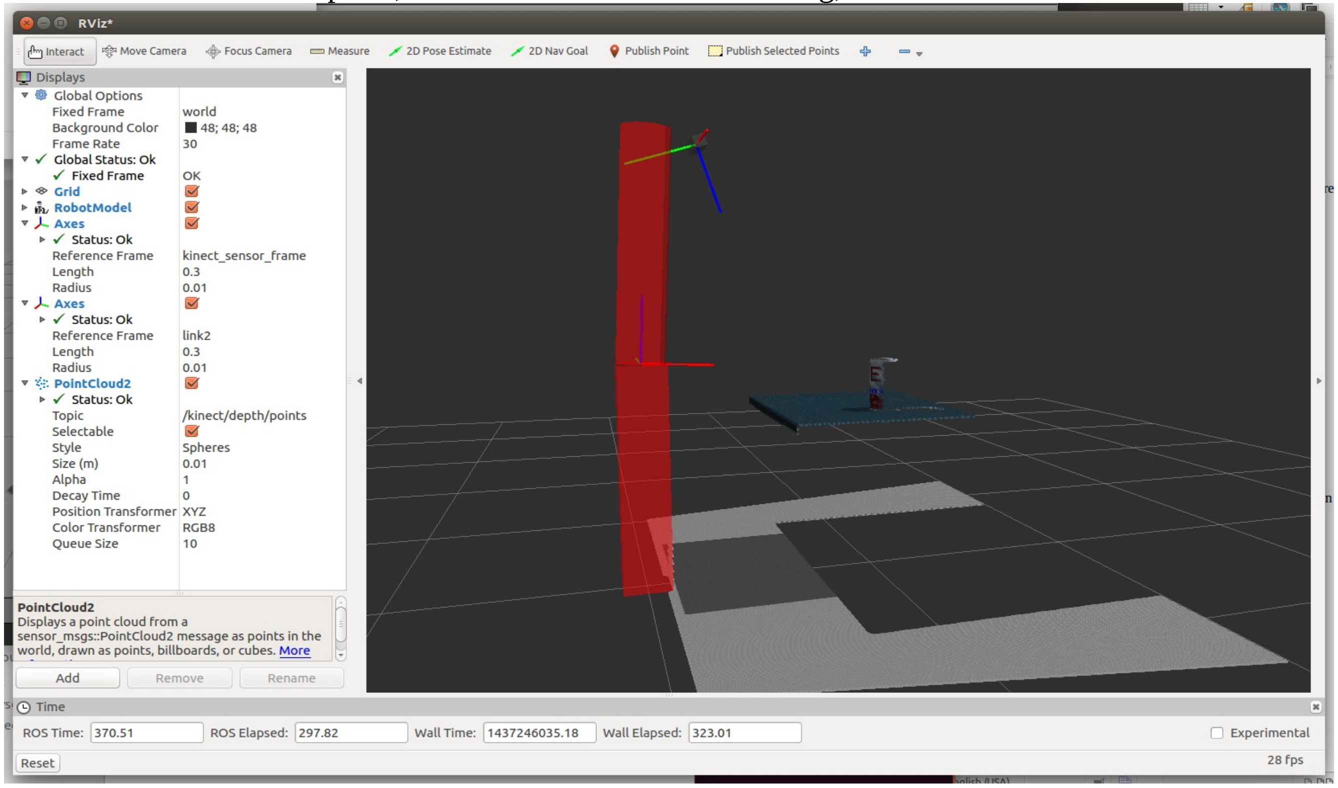
In the scene below, a model of a can has been placed on the table. Initially, the can is placed by Gazebo on the ground plane. By selecting the can and using the translation mode, the can may be placed on the table.



Steps: 1 Real Time Factor: 0.97 Sim Time: 00 00:07:11.590 Real Time: 00 00:07:21.299 Iterations: 431590 Reset

In placing the can on the table, it is helpful to suspend the dynamic simulation, else the can falls to the ground under the influence of gravity while trying to relocate it. To suspend simulation, click the "pause" button in the menu bar at the bottom of the Gazebo scene. After moving the can to the table top, re-enable the dynamic simulation by clicking the "play" icon, which will appear when the simulation is suspended.

With the extra models in place, the Rviz view is more interesting, as seen below.



| | |
|---|---|
| Global Options | |
| Fixed Frame | world |
| Background Color | ■ 48; 48; 48 |
| Frame Rate | 30 |
| ✓ Global Status: Ok | |
| ✓ Fixed Frame | OK |
| Grid | ✓ |
| RobotModel | ✓ |
| Axes | ✓ |
| ✓ Status: Ok | |
| Reference Frame | kinect_sensor_frame |
| Length | 0.3 |
| Radius | 0.01 |
| Axes | ✓ |
| ✓ Status: Ok | |
| Reference Frame | link2 |
| Length | 0.3 |
| Radius | 0.01 |
| PointCloud2 | ✓ |
| ✓ Status: Ok | |
| Topic | /kinect/depth/points |
| Selectable | ✓ |
| Style | Spheres |
| Size (m) | 0.01 |
| Alpha | 1 |
| Decay Time | 0 |
| Position Transformer | XYZ |
| Color Transformer | RGB8 |
| Queue Size | 10 |

**PointCloud2**
Displays a point cloud from a sensor_msgs::PointCloud2 message as points in the world, drawn as points, billboards, or cubes. More

ROS Time: 370.51 ROS Elapsed: 297.82 Wall Time: 1437246035.18 Wall Elapsed: 323.01

Reset 28 fps

The rviz display above includes a display item “PointCloud2” with topic set to “/kinect/depth/points.” This topic name follows from our URDF specifications of:

```
<cameraName>kinect</cameraName>
```

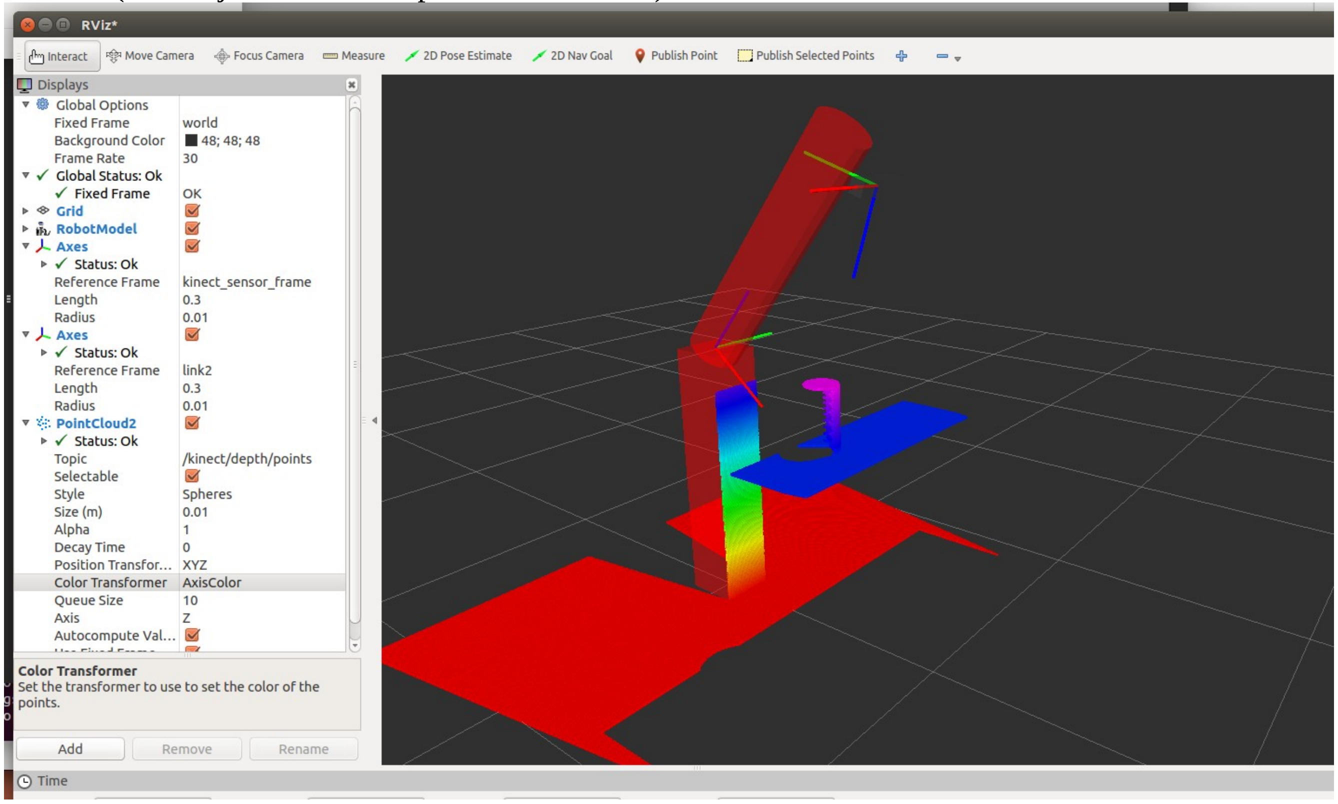
and:

```
<pointCloudTopicName>depth/points</pointCloudTopicName>
```

Additionally, two “Axes” items are displayed, showing the link2 frame and the kinect_sensor_frame. Note that the sensor frame defines its z-axis as pointing out from the camera along its optical axes. This sensor frame orientation is used by the sensor data, and the resulting 3-D points must be transformed into the fixed frame (world) of the Rviz display.

In the Rviz scene, one can see part of the table top, part of the beer can and part of the ground plane. The ground plane is view is limited by the field of view (and range) of the Kinect sensor, and part of this view is occluded by the table top. Part of the table top is visible in the scene, with the distal edge beyond the field of view. Part of the beer can is visible, corresponding to surfaces that have a clear line of sight from the sensor. Part of the table top is occluded by the beer can, creating the equivalent of a shadow.

In the Rviz scene below, the can is moved closer to the robot, and link2 of the robot is tilted slightly forward (via the joint controller position command).



| Displays | |
|---|---|
| Global Options | |
| Fixed Frame | world |
| Background Color | ■ 48; 48; 48 |
| Frame Rate | 30 |
| ✓ Global Status: Ok | |
| ✓ Fixed Frame | OK |
| Grid | ✓ |
| RobotModel | ✓ |
| Axes | ✓ |
| ✓ Status: Ok | |
| Reference Frame | kinect_sensor_frame |
| Length | 0.3 |
| Radius | 0.01 |
| Axes | ✓ |
| ✓ Status: Ok | |
| Reference Frame | link2 |
| Length | 0.3 |
| Radius | 0.01 |
| PointCloud2 | ✓ |
| ✓ Status: Ok | |
| Topic | /kinect/depth/points |
| Selectable | ✓ |
| Style | Spheres |
| Size (m) | 0.01 |
| Alpha | 1 |
| Decay Time | 0 |
| Position Transfor... | XYZ |
| Color Transformer | AxisColor |
| Queue Size | 10 |
| Axis | Z |
| Autocompute Val... | ✓ |

**Color Transformer**
Set the transformer to use to set the color of the points.

From this vantage point, the far side of the can is visible, but the near side is not. Also, at the new camera angle, the robot sees part of its own body, and it sees less of the table top. In the above display, parameters of the PointCloud2 item are set to “AxisColor” for “Color Transformer”, which displays the 3-D points with false color corresponding to z-height. Red points are at ground level, whereas the top of the can is violet, and elevations in between vary with the color spectrum seen on link1 of the robot.

**Conclusion**:  An impressive capability of Gazebo is that one can emulate sensors.  A depth camera (Kinect) is described in these notes, showing how to include the sensor in a URDF model.  The resulting simulated sensor data may be visualized in Rviz.  This capability allows one to develop sensor-based controls using simulation.  Subsequently, such code can be applied to a physical system, typically with only minor tuning required.  Key sensors known to Gazebo include: color cameras, depth cameras, LIDAR scanners, rotating or wobbling LIDAR scanners and ultrasound sensors.

In demonstrating the minimal robot with a virtual sensor, Rviz visualization required some additional transform information provided by the “robot_state_publisher” node and by “tf.”  Explanation of these nodes requires understanding “transforms,” which is introduced next.