CHAPTER 6

Robots and Simulators

The previous chapters discussed many fundamental concepts of ROS. They may have seemed rather vague and abstract, but those concepts were necessary to describe how data moves around in ROS and how its software systems are organized. In this chapter, we will introduce common robot subsystems and describe how the ROS architecture handles them. Then, we will introduce the robots that we will use throughout the remainder of the book and describe the simulators in which we can most easily experiment with them.

Subsystems

Like all complex machines, robots are most easily designed and analyzed by considering one subsystem at a time. In this section, we will introduce the main subsystems commonly found on the types of robots considered in this book. Broadly speaking, they can be divided into three categories: *actuation*, *sensing*, and *computing*. In the ROS context, actuation subsystems are the subsystems that interact directly with how the robot's wheels or arms move. Sensing subsystems interact directly with sensor hardware, such as cameras or laser scanners. Finally, the computational subsystems are what tie actuators and sensing together, with (ideally) some relatively intelligent processing that allows the robot to perform useful tasks. We will introduce these subsystems in the next few sections. Note that we are not attempting to provide an exhaustive discussion; rather, we are trying to describe these subsystems just deeply enough to convey the issues typically faced when interacting with them from a software development standpoint.

Actuation: Mobile Platform

The ability to move around, or *locomote*, is a fundamental capability of many robots. It is surprisingly nuanced: there are many books written entirely on this subject!

However, broadly speaking, a mobile base is a collection of actuators that allow a robot to move around. They come in an astonishingly wide variety of shapes and sizes.

Although legged locomotion is popular in some domains in the research community, and camera-friendly walking robots have seen great progress in recent years, most robots drive around on wheels. This is because of two main reasons. First, wheeled platforms are often simpler to design and manufacture. Second, for the very smooth surfaces that are common in artificial environments, such as indoor floors or outdoor pavement, wheels are the most energy-efficient way to move around.

The simplest possible configuration of a wheeled mobile robot is called *differential drive*. It consists of two independently actuated wheels, often located on the centerline of a round robot. In this configuration, the robot moves forward when both wheels turn forward, and spins in place when one wheel drives forward and one drives backward. Differential-drive robots often have one or more *casters*, which are unpowered wheels that spin freely to support the front and back of the robot, just like the wheels on the bottom of a typical office chair. This is an example of a *statically stable* robot, which means that, when viewed from above, the center of mass of the robot is inside a polygon formed by the points of contact between the wheels and the ground. Statically stable robots are simple to model and control, and among their virtues is the fact that power can be shut off to the robot at any time, and it will not fall over.

However, *dynamically stable* or *balancing* wheeled mobile robots are also possible, with the term *dynamic* implying that the actuators must constantly be in motion (however slight) to preserve stability. The simplest dynamically stable wheeled robots look like (and often are literally built upon) Segway platforms, with a pair of large differential-drive wheels supporting a tall robot above. Among the benefits of balancing wheeled mobile bases is that the wheels contacting the ground can have very large diameters, which allows the robot to smoothly drive over small obstacles: imagine the difference between running over a pebble with an office-chair wheel versus a bicycle wheel (this is, in fact, precisely the reason why bicycle wheels are large). Another advantage of balancing wheeled mobile robots is that the footprint of the robot can be kept small, which can be useful in tight quarters.

The differential-drive scheme can be extended to more than two wheels and is often called *skid steering*. Four-wheel and six-wheel skid-steering schemes are common, in which all of the wheels on the left side of the robot actuate together, and all of the wheels on the right side actuate together. As the number of wheels extends beyond six, typically the wheels are connected by external *tracks*, as exemplified by excavators or tanks.

As is typically the case in engineering, there are trade-offs with the skid-steering scheme, and it makes sense for some applications, but not all. One advantage is that skid steering provides maximum traction while preserving mechanical simplicity

(and thus controlling cost), since all contact points between the vehicle and the ground are being actively driven. However, skid steering is, as its name states, constantly skidding when it is not driving exactly forward or backward.

In some situations, traction and the ability to surmount large obstacles are valued so highly that skid steering platforms are used extensively. However, all this traction comes at a cost: the constant skidding is tremendously inefficient, since massive energy is spent tearing up the dirt (or heating up the wheels) whenever the robot turns at low speeds. In the most extreme case, when trying to turn in place with one set of wheels turning forwards and the other turning backward, the wheels are skidding dramatically, which can tear up gentle surfaces and wear tires quickly. This is why excavators are typically towed to a construction site on a trailer!

The inefficiencies and wear and tear of skid steering are among the reasons why passenger cars use more complex (and expensive) schemes to get around. They are often called *Ackerman* platforms, in which the rear wheels are always pointed straight ahead, and the front wheels turn together. Placing the wheels at the extreme corners of the vehicle maximizes the area of the supporting polygon, which is why cars can turn sharp corners without tipping over and (when not driven in action movies) car wheels do not have to skid when turning. However, the downside of Ackerman platforms is that they cannot drive sideways, since the rear wheels are always facing forward. This is why parallel parking is a dreaded portion of any driver's license examination: elaborate planning and sequential actuator maneuvers are required to move an Ackerman platform sideways.

All of the platforms described thus far can be summarized as being *non-holonomic*, which means that they cannot move in *any* direction at any given time. For example, neither differential-drive platforms nor Ackerman platforms can move sideways. To do this, a *holonomic* platform is required, which can be built using *steered casters*. Each steered caster actuator has two motors: one motor rotates the wheel forward and backward, and another motor steers the wheel about its vertical axis. This allows the platform to move in any direction while spinning arbitrarily. Although significantly more complex to build and maintain, these platforms simplify motion planning. Imagine the ease of parallel parking if you could drive sideways into a parking spot!

As a special case, when the robot only needs to move on very smooth surfaces, a low-cost holonomic platform can be built using *Mecanum* wheels. These are clever contraptions in which each wheel has a series of rollers on its rim, angled at 45 degrees to the plane of the wheel. Using this scheme, motion in any direction (with any rate of rotation) is possible at all times, using only four actuators, without skidding. However, due to the small diameter of the roller wheels, it is only suitable for very smooth surfaces such as hard flooring or extremely short-pile carpets.

Because one of the design goals of ROS is to allow software reuse across a variety of robots, ROS software that interacts with mobile platforms virtually always uses a `Twist` message. A *twist* is a way to express general linear and angular velocities in three dimensions. Although it may seem easier to express mobile base motions simply by expressing wheel velocities, using the linear and angular velocities of the center of the vehicle allows the software to abstract away the kinematics of the vehicle.

For example, high-level software can command the vehicle to drive forward at 0.5 meters/second while rotating clockwise at 0.1 radians/second. From the standpoint of the high-level software, whether the mobile platform's actuators are arranged as differential-drive, Ackerman steering, or Mecanum wheels is irrelevant, just as the transmission ratios and wheel diameters are irrelevant to high-level behaviors.

The robots described in this book will only be navigating on flat, two-dimensional surfaces and are commonly called *planar robots*. However, expressing velocities in three dimensions allows path planning or obstacle avoidance software to be used by vehicles capable of more general motions, such as aerial, underwater, or space vehicles. It is important to recognize that even for vehicles designed for two-dimensional navigation, the general three-dimensional twist methodology is necessary to express desired or actual motions of many types of actuators, such as grippers, since they are often capable of three-dimensional motions when flying on the end of a manipulator arm even when the mobile base is constrained to the floor plane. Manipulators, in fact, comprise the other main application domain for robot actuators and will be discussed in the next section.

Actuation: Manipulator Arm

Many robots need to *manipulate* objects in their environment. For example, packing or palletizing robots sit on the end of a production line, grab items coming down the line, and place them into boxes or stacks. There is an entire domain of robot manipulation tasks called *pick and place*, in which manipulator arms grasp items and place them somewhere else. Security robot tasks include handling suspicious items, for which a strong manipulator arm is often required. An emerging class of *personal robots* hope to be useful in home and office applications, performing manipulation tasks including cleaning, delivering items, preparing meals, and so on.

As with mobile bases, there's astonishing variety in manipulator-arm subsystems across robots, with many trade-offs made to support particular application domains and price points.

Although there are exceptions, the majority of manipulator arms are formed by a *chain* of rigid *links* connected by *joints*. The simplest kinds of joints are single-axis revolute joints (also called "pin" joints), where one link has a shaft that serves as the axis around which the next link rotates, in the same way that a typical residential door rotates around its hinge pins. However, *linear* joints (also called *prismatic* joints)

are also common, in which one link has a *slide* or tube along which the next link travels, just as a sliding door runs sideways back and forth along its track.

A fundamental characteristic of a robot manipulator is the number of *degrees of freedom* (DOF) of its design. Often, the number of joints is equal to the number of actuators; when those numbers differ, typically the DOF is taken to be the lower of the two numbers. Regardless, the number of degrees of freedom is one of the most significant drivers of manipulator size, mass, dexterity, cost, and reliability. Adding DOF to the *distal* (far) end of a robot arm typically increases its mass, which requires larger actuators on the *proximal* (near) joints, which further increases the mass of the manipulator.

In general, six DOF are required to position the wrist of the manipulator arm in any location and orientation within its *workspace*, providing that each joint has full range of motion. In this context, *workspace* has a precise meaning: it is the space that a robot manipulator can reach. A subset of the robot's workspace, called the *dextrous workspace*, is the region in which a robot can achieve all positions and orientations of the end effector. Generally speaking, having a larger dextrous workspace is a good thing for robots, but unfortunately full (360-degree) range of motion on six joints of a robot is often extremely difficult to achieve at reasonable cost, due to constraints of mechanical structures, electrical wiring, and so on. As a result, seven-DOF arms are often used. The seventh DOF provides an extra degree of freedom that can be used to move the links of the arm while maintaining the position and orientation of the wrist, much as a human arm can move its elbow through an arc segment while maintaining the wrist in the same position. This "extra" DOF can help contribute to a relatively large dextrous workspace even when each individual joint has a restricted range of motion.

Research robots intended for manipulation tasks in human environments often have human-scale, seven-DOF arms, quite simply because the desired workspaces are human-scale surfaces, such as tables or countertops in home and office environments. In contrast, robots intended for industrial applications have wildly varying dimensions and joint configurations depending on the tasks they are to perform, since each additional DOF introduces additional cost and reliability concerns.

So far, we have discussed the two main classes of robot actuators: those used for locomotion, and those used for manipulation. The next major class of robot hardware is its sensors. We'll start with the sensor head, a common mounting scheme, and then describe the subcomponents found in many robot sensor heads.

Sensors

Robots must sense the world around them in order to react to variations in tasks and environments. The sensors can range from minimalist setups designed for quick installation to highly elaborate and tremendously expensive sensor rigs.

Many successful industrial deployments use surprisingly little sensing. A remarkable number of complex and intricate industrial manipulation tasks can be performed through a combination of clever mechanical engineering and *limit switches*, which close or open an electrical circuit when a mechanical lever or plunger is pressed, in order to start execution of a preprogrammed robotic manipulation sequence. Through careful mechanical setup and tuning, these systems can achieve amazing levels of throughput and reliability. It is important, then, to consider these *binary* sensors when enumerating the world of robotic sensing. These sensors are typically either "on" or "off." In addition to mechanical limit switches, other binary sensors include *optical limit switches*, which use a mechanical "flag" to interrupt a light beam, and *bump sensors*, which channel mechanical pressure along a relatively large distance to a single mechanical switch. These relatively simple sensors are a key part of modern industrial automation equipment, and their importance can hardly be overstated.

Another class of sensors return *scalar* readings. For example, a pressure sensor can estimate the mechanical or barometric pressure and will typically output a scalar value along some range of sensitivity chosen at time of manufacture. Range sensors can be constructed from many physical phenomena (sound, light, etc.) and will also typically return a scalar value in some range, which seldom includes zero or infinity!

Each sensor class has its own quirks that distort its view of reality and must be accommodated by sensor-processing algorithms. These quirks can often be surprisingly severe. For example, a range sensor may have a "minimum distance" restriction: if an object is closer than that minimum distance, it will not be sensed. As a result of these quirks, it is often advantageous to combine several different types of sensors in a robotic system.

However, many of the applications we will describe in this book are reliant on "rich" sensor data, which is a vague term that generally means that the robot's perception algorithms consider something more than a small number of binary or scalar sensors. Any configuration of sensing hardware is possible (and has likely been tried), but for convenience, aesthetics, and to preserve line-of-sight with the center of the workspace, it is common for robots to have a *sensor head* on top of the platform that integrates several sensors in the same physical enclosure. Often, sensor heads sit atop a *pan/tilt* assembly, so that they can rotate to a bearing of interest and look up or down as needed. The following several sections will describe sensors commonly found in robot sensor heads and on other parts of their bodies.

Visual cameras

Higher-order animals tends to rely on visual data to react to the world around them. If only robots were as smart as animals! Unfortunately, using camera data intelligently is surprisingly difficult, as we will describe in later chapters of this book. However,

cameras are cheap and often useful for teleoperation, so it is common to see them on robot sensor heads.

Interestingly, it is often more mathematically robust to describe robot tasks and environments in three dimensions (3D) than it is to work with 2D camera images. This is because the 3D shapes of tasks and environments are *invariant* to changes in scene lighting, shadows, occlusions, and so on. In fact, in a surprising number of application domains, the visual data is largely ignored; the algorithms are interested in 3D data. As a result, intense research efforts have been expended on producing 3D data of the scene in front of the robot.

When two cameras are rigidly mounted to a common mechanical structure, they form a *stereo camera*. Each camera sees a slightly different view of the world, and these slight differences can be used to estimate the distances to various features in the image. This sounds simple, but as always, the devil is in the details. The performance of a stereo camera depends on a large number of factors, such as the quality of the camera's mechanical design, its resolution, its lens type and quality, and so on. Equally important are the qualities of the scene being imaged: a stereo camera can only estimate the distances to mathematically discernable *features* in the scene, such as sharp, high-contrast corners. A stereo camera cannot, for example, estimate the distance to a featureless wall, although it can most likely estimate the distance to the corners and edges of the wall, if they intersect a floor, ceiling, or other wall of a different color. Many natural outdoor scenes possess sufficient texture that stereo vision can be made to work quite well for depth estimation. Uncluttered indoor scenes, however, can often be quite difficult.

Several conventions have emerged in the ROS community for handling cameras. The canonical ROS message type for images is `sensor_msgs/Image`, and it contains little more than the size of the image, its pixel encoding scheme, and the pixels themselves. To describe the *intrinsic distortion* of the camera resulting from its lens and sensor alignment, the `sensor_msgs/CameraInfo` message is used. Often, these ROS images need to be sent to and from OpenCV, a popular computer vision library. The `cv_bridge` package is intended to simplify this operation and will be used throughout the book.

Depth cameras

As discussed in the previous section, even though visual camera data is intuitively appealing, and seems like it should be useful somehow, many perception algorithms work much better with 3D data. Fortunately, the past few years have seen massive progress in low-cost *depth cameras*. Unlike the passive stereo cameras described in the previous section, depth cameras are *active* devices. They illuminate the scene in various ways, which greatly improves the system performance. For example, a completely featureless indoor wall or surface is essentially impossible to detect using

passive stereo vision. However, many depth cameras will shine a texture pattern on the surface, which is subsequently imaged by its camera. The texture pattern and camera are typically set to operate in near-infrared wavelengths to reduce the system's sensitivity to the colors of objects, as well as to not be distracting to people nearby.

Some common depth cameras, such as the Microsoft Kinect, project a *structured light* image. The device projects a precisely known pattern into the scene, its camera observes how this pattern is deformed as it lands on the various objects and surfaces of the scene, and finally a *reconstruction algorithm* estimates the 3D structure of the scene from this data. It's hard to overstate the impact that the Kinect has had on modern robotics! It was designed for the gaming market, which is orders of magnitude larger than the robotics sensor market, and could justify massive expenditures for the development and production of the sensor. The launch price of $150 was incredibly cheap for a sensor capable of outputting so much useful data. Many robots were quickly retrofitted to hold Kinects, and the sensor continues to be used across research and industry.

Although the Kinect is the most famous (and certainly the most widely used) depth camera in robotics, many other depth-sensing schemes are possible. For example, *unstructured light* depth cameras employ "standard" stereo-vision algorithms with random texture injected into the scene by some sort of projector. This scheme has been shown to work far better than passive stereo systems in feature-scarce environments, such as many indoor scenes.

A different approach is used by *time-of-flight* depth cameras. These imagers rapidly blink an infrared LED or laser illuminator, while using specially designed pixel structures in their image sensors to estimate the time required for these light pulses to fly into the scene and bounce back to the depth camera. Once this "time of flight" is estimated, the (constant) speed of light can be used to convert the estimates into a *depth image*.

Intense research and development is occurring in this domain, due to the enormous existing and potential markets for depth cameras in video games and other mass-market user-interaction scenarios. It is not yet clear which (if any) of the schemes discussed previously will end up being best suited for robotics applications. At the time of writing, cameras using all of the previous modalities are in common usage in robotics experiments.

Just like visual cameras, depth cameras produce an enormous amount of data. This data is typically in the form of *point clouds*, which are the 3D points estimated to lie on the surfaces facing the camera. The fundamental point cloud message is `sensor_msgs/PointCloud2` (so named purely for historical reasons). This message allows for unstructured point cloud data, which is often advantageous, since depth cameras often cannot return valid depth estimates for each pixel in their images. As

such, depth images often have substantial "holes," which processing algorithms must handle gracefully.

Laser scanners

Although depth cameras have greatly changed the depth-sensing market in the last few years due to their simplicity and low cost, there are still some applications in which *laser scanners* are widely used due to their superior accuracy and longer sensing range. There are many types of laser scanners, but one of the most common schemes used in robotics involves shining a laser beam on a rotating mirror spinning around 10 to 80 times per second (typically 600 to 4,800 RPM). As the mirror rotates, the laser light is pulsed rapidly, and the reflected waveforms are correlated with the outgoing waveform to estimate the time of flight of the laser pulse for a series of angles around the scanner.

Laser scanners used for autonomous vehicles are considerably different from those used for indoor or slow-moving robots. Vehicle laser scanners made by companies such as Velodyne must deal with the significant aerodynamic forces, vibrations, and temperature swings common to the automotive environment. Since vehicles typically move much faster than smaller robots, vehicle sensors must also have considerably longer range so that sufficient reaction time is possible. Additionally, many software tasks for autonomous driving, such as detecting vehicles and obstacles, work much better when multiple laser *scanlines* are received each time the device rotates, rather than just one. These extra scanlines can be extremely useful when distinguishing between classes of objects, such as between trees and pedestrians. To produce multiple scanlines, automotive laser scanners often have multiple lasers mounted together in a rotating structure, rather than simply rotating a mirror. All of these additional features naturally add to the complexity, weight, size, and thus the cost of the laser scanner.

The complex signal processing steps required to produce range estimates are virtually always handled by the firmware of the laser scanner itself. The devices typically output a vector of ranges several dozen times per second, along with the starting and stopping angles of each measurement vector. In ROS, laser scans are stored in `sensor_msgs/LaserScan` messages, which map directly from the output of the laser scanner. Each manufacturer, of course, has their own raw message formats, but ROS drivers exist to translate between the raw output of many popular laser scanner manufacturers and the `sensor_msgs/LaserScan` message format.

Shaft encoders

Estimating the motions of the robot is a critical component of virtually all robotic systems, with solutions ranging from low-level control schemes to high-level mapping, localization, and manipulation algorithms. Although estimates can be derived

from many sources, the simplest and often most accurate estimates are produced simply by counting how many times the motors or wheels have turned.

Many different types of *shaft encoders* are designed expressly for this purpose. Shaft encoders are typically constructed by attaching a marker to the shaft and measuring its motion relative to another frame of reference, such as the chassis of the robot or the previous link on a manipulator arm. The implementation may be done with magnets, optical discs, variable resistors, or variable capacitors, among many other options, with trade-offs including size, cost, accuracy, maximum speed, and whether the measurement is *absolute* or *relative* to the position at power-up. Regardless, the principle remains the same: the angular position of a marker on a shaft is measured relative to an adjacent *frame of reference*.

Just like automobile speedometers and odometers, shaft encoders are used to count the precise number of rotations of the robot's wheels, and thereby estimate how far the vehicle has traveled and how much it has turned. Note that *odometry* is simply a count of how many times the drive wheels have turned, and is also known as *dead reckoning* in some domains. It is *not* a direct measurement of the vehicle position. Minute differences in wheel diameters, tire pressures, carpet weave direction (really!), axle misalignments, minor skidding, and countless other sources of error are cumulative over time. As a result, the raw odometry estimates of *any* robot will drift; the longer the robot drives, the more error accumulates in the estimate. For example, a robot traveling down the middle of a long, straight corridor will *always* have odometry that is a gradual curve. Put another way, if both tires of a differential-drive robot are turned in the same direction at the exact same wheel velocity, the robot will never drive in a truly straight line. This is why mobile robots need additional sensors and clever algorithms to build maps and navigate.

Shaft encoders are also used extensively in robot manipulators. The vast majority of manipulator arms have at least one shaft encoder for every rotary joint, and the vector of shaft encoder readings is often called the *manipulator configuration*. When combined with a geometric model of each link of a manipulator arm, the shaft encoders allow higher-level collision-avoidance, planning, and trajectory-following algorithms to control the robot.

Because the mobility and manipulation uses of shaft encoders are quite different, the ROS conventions for each use are also quite different. Although the raw encoder counts may also be reported by some mobile-base device drivers, odometry estimates are most useful when reported as a *spatial transformation* represented by a `geometry_msgs/Transform` message. This concept will be discussed at great length throughout the book, but in general, a spatial transform describes one frame of reference relative to another frame of reference. In this case, the odometry transform typically describes the shaft encoder's odometric estimate relative to the position of the robot at power-up, or where its encoders were last reset.

In contrast, the encoder readings for manipulator arms are typically broadcast by ROS manipulator device drivers as `sensor_msgs/JointState` messages. The `JointState` message contains vectors of angles in radians, and angular velocities in radians per second. Since typical shaft encoders have thousands of discrete states per revolution, the ROS device drivers for manipulator arms are required to scale the encoders as needed, accounting for transmissions and linkages, to produce a `JointState` vector with standard units. These messages are used extensively by ROS software packages, as they provide the minimal complete description of the state of a manipulator.

That about covers it for the physical parts of a robot system. We now turn our attention to the "brains," where the robot interprets sensor data and determines how to move its body, and where we'll be spending most of our time in this book.

Computation

Impressive robotic systems have been implemented on computing resources ranging from large racks of servers down to extremely small and efficient 8-bit microcontrollers. Fierce debates have raged throughout the history of robotics as to exactly how much computer processing is required to produce robust, useful robot behavior. Insect brains, for example, are extremely small and power-efficient, yet insects are arguably the most successful life forms on the planet. Biological brains process data very differently from "mainstream" systems-engineering approaches of human technology, which has led to large and sustained research projects that study and try to replicate the success of bio-inspired computational architectures.

ROS takes a more traditional software-engineering approach to robotic computational architecture; as described in the first few chapters of this book, ROS uses a dynamic message-passing graph to pass data between software nodes, which are typically isolated by the POSIX process model. This does not come for free. It certainly requires additional CPU cycles to serialize a message from one node, send it over some interprocess or network communications method to another node, and deserialize it for another node. However, it is our opinion that the rapid prototyping and software integration benefits of this architecture outweigh its computational overhead.

Because of this messaging overhead and the emphasis on module isolation, ROS is not currently intended to run on extremely small microcontrollers. ROS can be (and has been) used to emulate and rapid-prototype minimalist processing paradigms. Typically, however, ROS is used to build systems that include considerable perceptual input and complex processing algorithms, where its modular and dynamically extensible architecture can simplify system design and operation.

ROS currently must run on top of a full-featured operating system such as Linux or Mac OS X. Fortunately, the continuing advance of Moore's law and mass-market

demand for battery-powered devices has led to ever-smaller and more power-efficient platforms capable of running full operating systems. ROS can run on small-form-factor embedded computer systems such as Gumstix, Raspberry Pi, or BeagleBone, among many others. Going up the performance and power curve, ROS has been widely used on a large range of laptops, desktops, and servers. Human-scale robots often carry one or more standard PC motherboards running Linux headless, which are accessed over a network link.

Complete Robots

The previous section described subsystems commonly found on many types of robots running ROS. Many of these robots used in research settings are custom built to investigate a particular research problem. However, there are a growing number of standard products that can be purchased and used "out of the box" for research, development, and operations in many domains of robotics. This section will describe several of these platforms, which will be used for examples throughout the rest of the book.

PR2

The PR2 robot was one of the original ROS target platforms. In many ways, it was the "ultimate" research platform for service-robotics software at the time of its release in 2010. Its mobile base is actuated by four steerable casters and has a laser scanner for navigation. Atop this mobile base, the robot has a telescoping torso that carries two human-scale seven-DOF arms. The arms have a unique passive mechanical counterbalance, which permits the use of surprisingly low-power motors for human-scale arms.

The PR2 has a pan/tilt head equipped with a wide range of sensors, including a "nodding" laser scanner that can tilt up and down independently of the head, a pair of stereo cameras for short and long distances, and a Kinect depth camera. Additionally, each forearm of the robot has a camera, and the gripper fingertips have tactile sensors. All told, the PR2 has two laser scanners, six cameras, a depth camera, four tactile arrays, and 1 kHz encoder feedback. All of this data is handled by a pair of computers in the base of the robot, with an onboard gigabit network connecting them to a pair of WiFi radios.

All of this functionality came at a price, since the PR2 was not designed for low cost. When it was commercially available, the PR2 listed for about $400,000.[1] Despite this financial hurdle, its fully integrated "out-of-the-box" experience was a landmark for research robots and is why PR2 robots are being actively used in dozens of research

1 All prices are approximate, as of the time of writing, and quoted in US dollars.

labs around the world. Figure 6-1 shows a PR2 running in the Gazebo simulator. Simulators will be discussed later in this chapter.

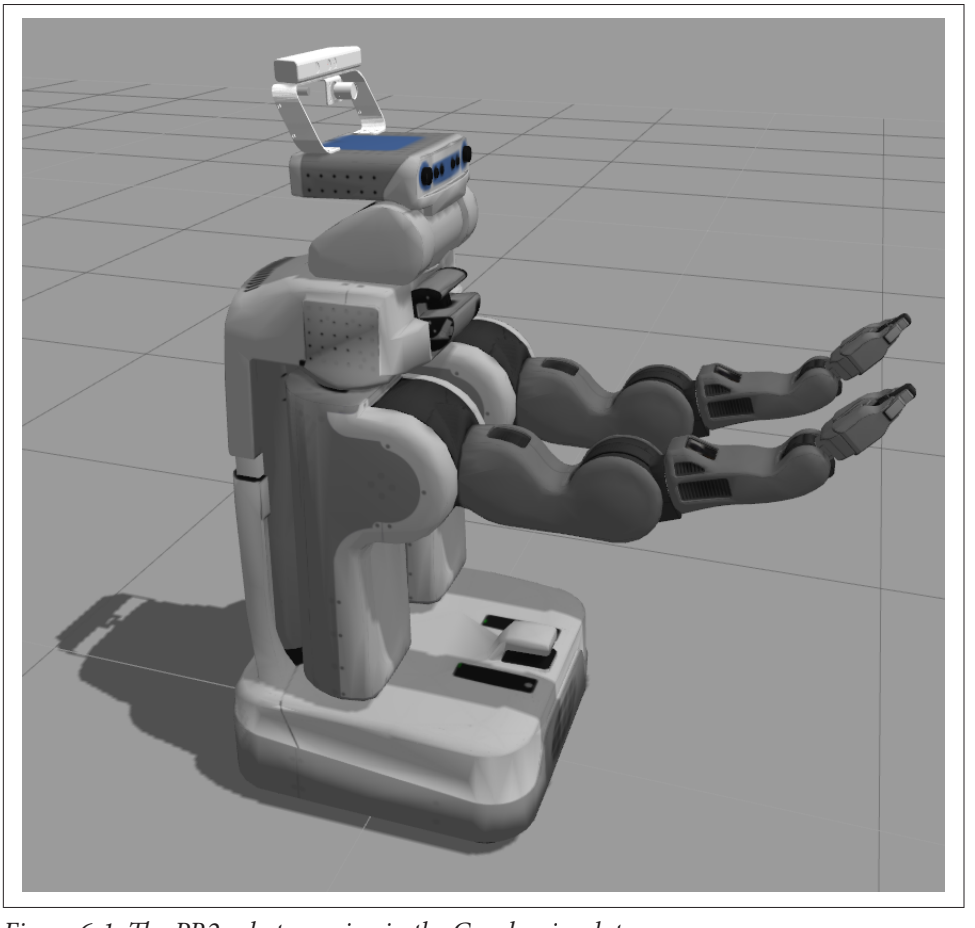

Figure 6-1. The PR2 robot running in the Gazebo simulator

Fetch

Fetch is a mobile manipulation robot intended for warehouse applications. The design team at Fetch Robotics, Inc. includes many of those who designed the PR2 robot, and in some ways the Fetch robot can be seen as a smaller, more practical and cost-effective "spiritual successor" of the PR2. The single-arm robot, shown in Figure 6-2 is fully ROS-based and has a compact sensor head built around a depth camera. The differential-drive mobile base has a laser scanner intended for navigation purposes and a telescoping torso. At the time of writing, the price of the robot has not been publicly released, but it is expected to be much more affordable than the PR2.

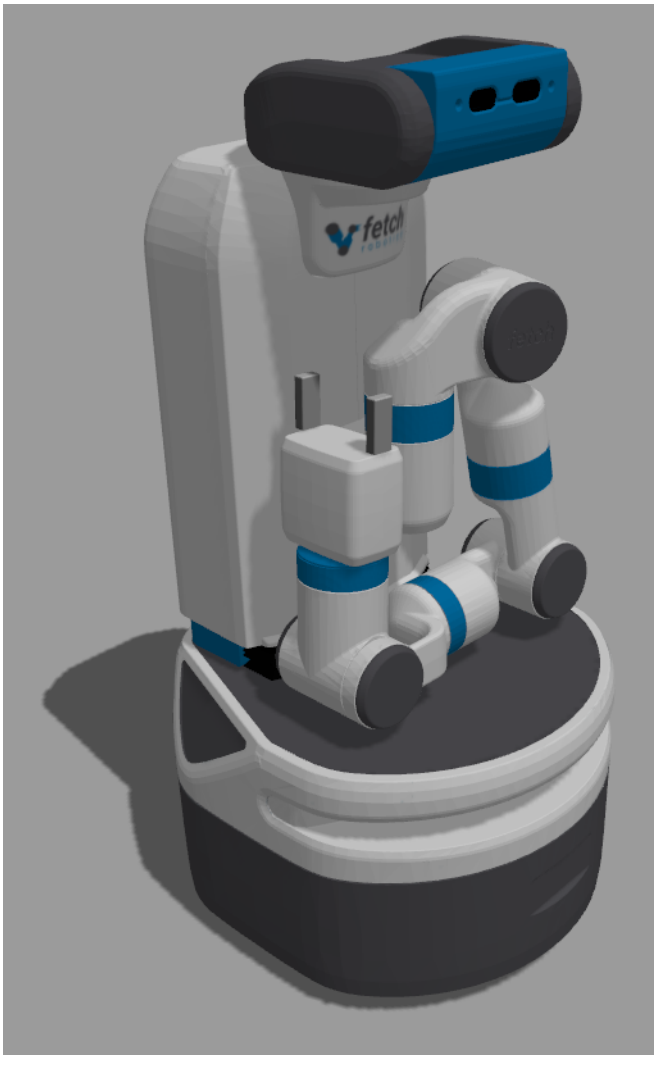

Figure 6-2. The Fetch robot running in the Gazebo simulator

Robonaut 2

The NASA/GM Robonaut 2 (Figure 6-3 is a human-scale robot designed with the extreme reliability and safety systems necessary for operation aboard the International Space Station. At the time of writing, the Robonaut 2 (a.k.a R2) aboard the space station is running ROS for high-level task control. Much more information is available at *http://robonaut.jsc.nasa.gov*.

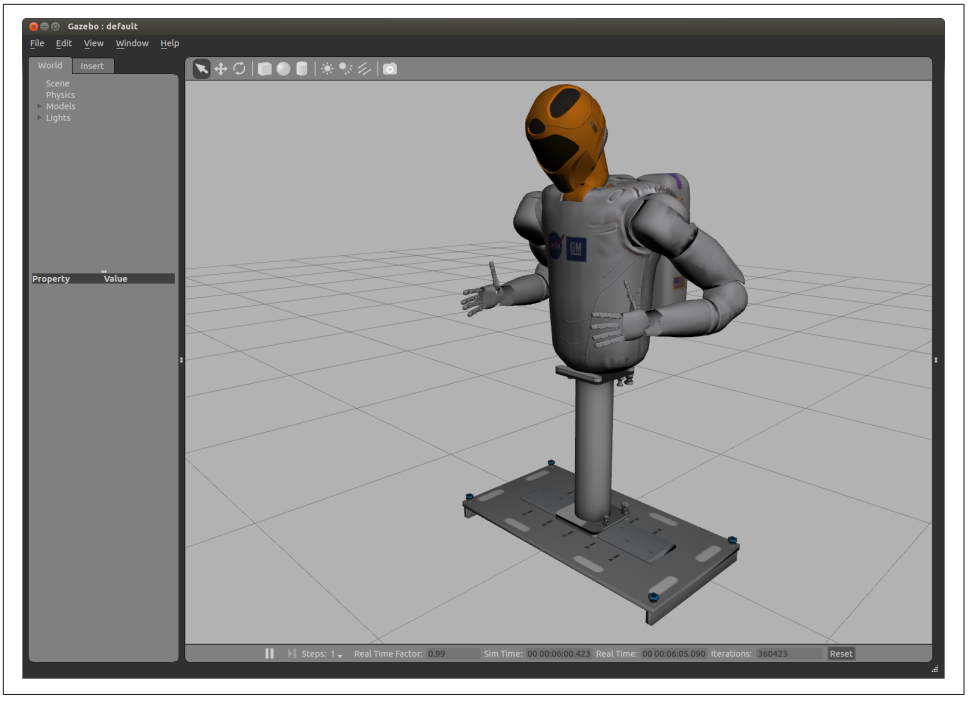

Figure 6-3. The NASA R2 robot running in the Gazebo simulator

TurtleBot

The TurtleBot was designed in 2011 as a minimalist platform for ROS-based mobile robotics education and prototyping. It has a small differential-drive mobile base with an internal battery, power regulators, and charging contacts. Atop this base is a stack of laser-cut "shelves" that provide space to hold a netbook computer and depth camera, and lots of open space for prototyping. To control cost, the TurtleBot relies on a depth camera for range sensing; it does not have a laser scanner. Despite this, mapping and navigation can work quite well for indoor spaces. TurtleBots are available from several manufacturers for less than $2,000. More information is available at *http://turtlebot.org*.

Because the shelves of the TurtleBot (pictured in Figure 6-4) are covered with mounting holes, many owners have added additional subsystems to their TurtleBots, such as small manipulator arms, additional sensors, or upgraded computers. However, the "stock" TurtleBot is an excellent starting point for indoor mobile robotics. Many similar systems exist from other vendors, such as the Pioneer and Erratic robots and thousands of custom-built mobile robots around the world. The examples in this book will use the TurtleBot, but any other small differential-drive platform could easily be substituted.

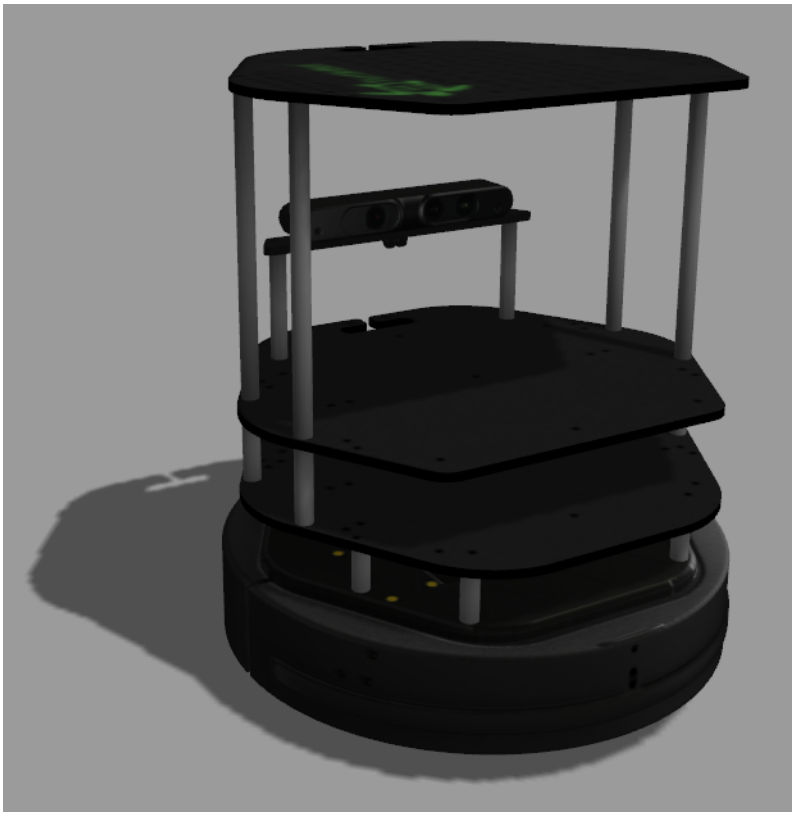

Figure 6-4. The TurtleBot robot running in the Gazebo simulator

Simulators

Although the preceding list of robots includes platforms that we consider to be remarkably low-cost compared to prior robots of similar capabilities, they are still significant investments. In addition, real robots require logistics including lab space, recharging of batteries, and operational quirks that often become part of the institutional knowledge of the organization operating the robot. Sadly, even the best robots break periodically due to various combinations of operator error, environmental conditions, manufacturing or design defects, and so on.

Many of these headaches can be avoided by using *simulated* robots. At first glance, this seems to defeat the whole purpose of robotics; after all, the very definition of a robot involves perceiving and/or manipulating the environment. Software robots, however, are extraordinarily useful. In simulation, we can model as much or as little of reality as we desire. Sensors and actuators can be modeled as ideal devices, or they can incorporate various levels of distortion, errors, and unexpected faults. Although

data logs can be used in automated test suites to verify that sensing algorithms produce expected results, automated testing of control algorithms typically requires simulated robots, since the algorithms under test need to be able to experience the consequences of their actions.

Simulated robots are the ultimate low-cost platforms. They are free! They do not require complex operating procedures; you simply spawn a `roslaunch` script and wait a few seconds, and a shiny new robot is created. At the end of the experimental run, a quick Ctrl-C and the robot vaporizes. For those of us who have spent many long nights with the pain and suffering caused by operating real robots, the benefits of simulated robots are simply magical.

Due to the isolation provided by the messaging interfaces of ROS, a vast majority of the robot's software graph can be run identically whether it is controlling a real robot or a simulated robot. At runtime, as the various nodes are launched, they simply find one another and connect. Simulation input and output streams connect to the graph in the place of the device drivers of the real robot. Although some parameter tuning is often required, ideally the *structure* of the software will be the same, and often the simulation can be modified to reduce the amount of parameter tweaks required when transitioning between simulation and reality.

As alluded to in the previous paragraphs, there are many use cases for simulated robots, ranging from algorithm development to automated software verification. This has led to the creation of a large number of robot simulators, many of which integrate nicely with ROS. The following sections describe two simulators that will be used in this book.

Stage

For many years, the two-dimensional *simultaneous localization and mapping* (SLAM) problem was one of the most heavily researched topics in the robotics community. A number of 2D simulators were developed in response to the need for repeatable experiments, as well as the many practical annoyances of gathering long datasets of robots driving down endless office corridors. Canonical laser range-finders and differential-drive robots were modeled, often using simple *kinematic* models that enforce that, for example, the robot stays plastered to a 2D surface and its range sensors only interact with vertical walls, creating worlds that vaguely resemble that of *Pac-Man* (see Figure 6-5). Although limited in scope, these 2D simulators are very fast computationally, and they are generally quite simple to interact with.

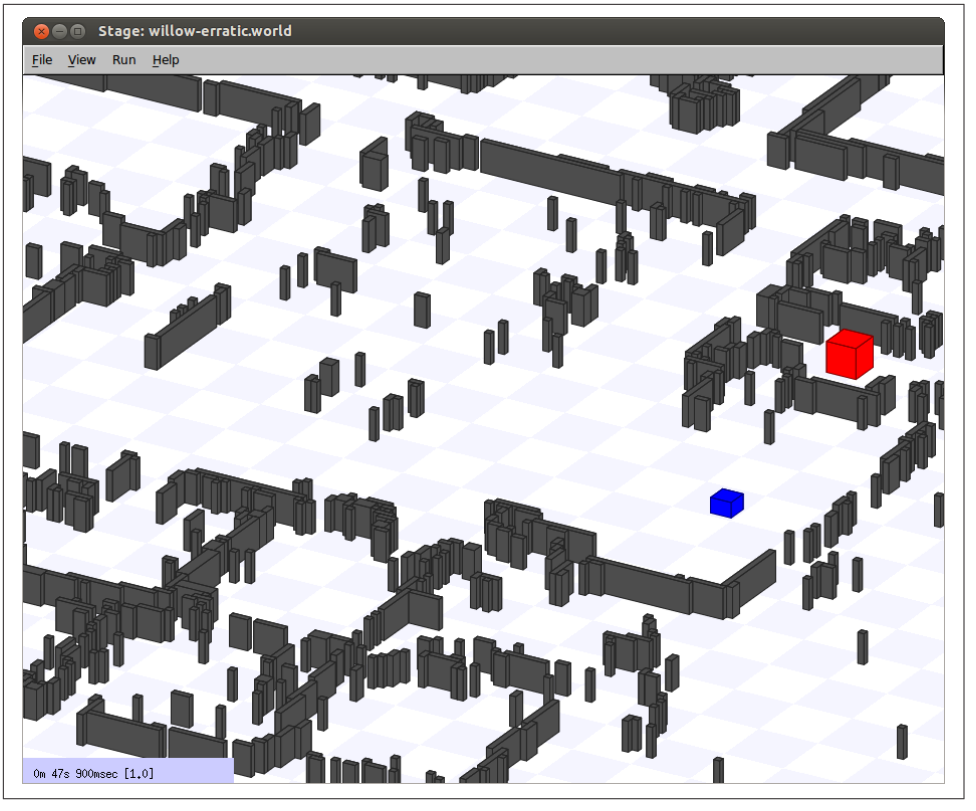

Figure 6-5. Typical screenshot of the Stage simulator

Stage is an excellent example of this type of 2D simulator. It has a relatively simple modeling language that allows the creation of planar worlds with simple types of objects. Stage was designed from the outset to support multiple robots simultaneously interacting with the same world. It has been wrapped with a ROS integration package that accepts velocity commands from ROS and outputs an odometric transformation as well as the simulated laser range-finders from the robot(s) in the simulation.

Gazebo

Although Stage and other 2D simulators are computationally efficient and excel at simulating planar navigation in office-like environments, it is important to note that planar navigation is only one aspect of robotics. Even when only considering robot navigation, a vast array of environments require nonplanar motion, ranging from outdoor ground vehicles to aerial, underwater, and space robotics. Three-dimensional simulation is necessary for software development in these environments.

In general, robot motions can be divided into *mobility* and *manipulation*. The mobility aspects can be handled by two- or three-dimensional simulators in which the environment around the robot is *static*. Simulating manipulation, however, requires a significant increase in the complexity of the simulator to handle the dynamics of not just the robot, but also the *dynamic* models in the scene. For example, at the moment that a simulated household robot is picking up a handheld object, contact forces must be computed between the robot, the object, and the surface the object was previously resting upon.

Simulators often use *rigid-body* dynamics, in which all objects are assumed to be incompressible, as if the world were a giant pinball machine. This assumption drastically improves the computational performance of the simulator, but often requires clever tricks to remain stable and realistic, since many rigid-body interactions become *point contacts* that do not accurately model the true physical phenomena. The art and science of managing the tension between computational performance and physical realism are highly nontrivial. There are many approaches to this trade-off, with many well suited to some domains but ill suited to others.

Like all simulators, Gazebo (Figure 6-6) is the product of a variety of trade-offs in its design and implementation. Historically, Gazebo has used the Open Dynamics Engine for rigid-body physics, but recently it has gained the ability to choose between physics engines at startup. For the purposes of this book, we will be using Gazebo with either the Open Dynamics Engine or with the Bullet Physics library, both of which are capable of real-time simulation with relatively simple worlds and robots and, with some care, can produce physically plausible behavior.

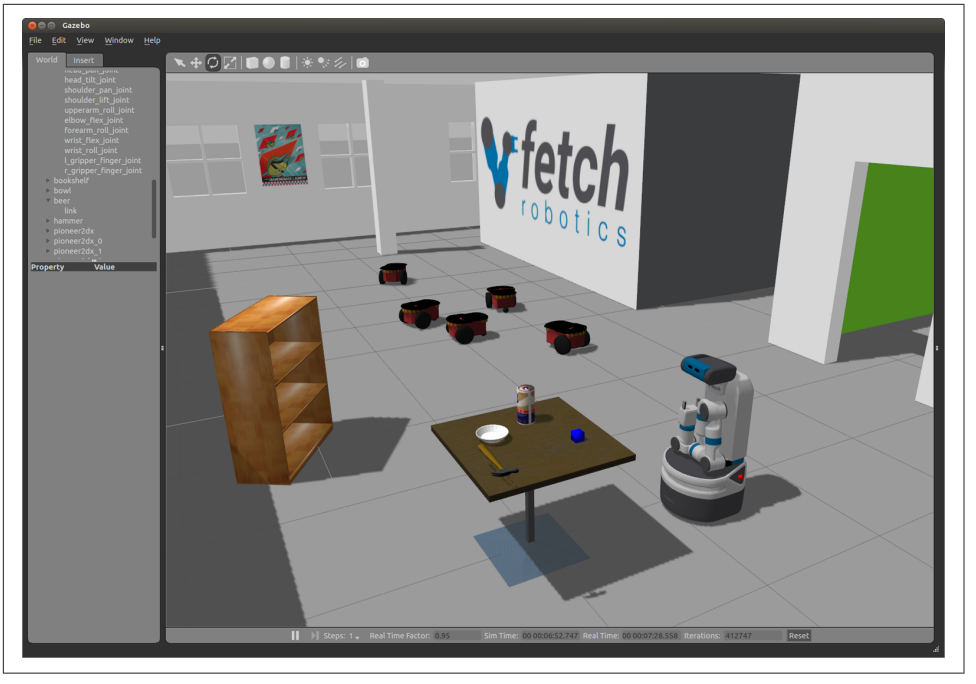

Figure 6-6. Typical screenshot of the Gazebo simulator

ROS integrates closely with Gazebo through the `gazebo_ros` package. This package provides a Gazebo *plugin* module that allows bidirectional communication between Gazebo and ROS. Simulated sensor and physics data can stream from Gazebo to ROS, and actuator commands can stream from ROS back to Gazebo. In fact, by choosing consistent names and data types for these data streams, it is possible for Gazebo to *exactly* match the ROS API of a robot. When this is achieved, all of the robot software above the device-driver level can be run identically both on the real robot, and (after parameter tuning) in the simulator. This is an enormously powerful concept and will be used extensively throughout this book.

Other Simulators

There are many other simulators that can be used with ROS, such as MORSE and V-REP. Each simulator, whether it be Gazebo, Stage, MORSE, V-REP, turtlesim, or any other, has a different set of trade-offs. These include trade-offs in speed, accuracy, graphics quality, dimensionality (2D versus 3D), types of sensors supported, usability, platform support, and so on. No simulator of which we are aware is capable of maximizing all of those attributes simultaneously, so the choice of the "right" simulator for a particular task will be dependent on many factors.

Summary

In this chapter, we've looked at the subsystems of a typical robot, focusing on the types of robots that ROS is most concerned with: mobile manipulation platforms. By now, you should have a pretty good idea of what a robot looks like, and you should be starting to figure out how ROS might be used to control one, reading data from the sensors, figuring out how to interpret that data and what to do, and sending commands to the acutators to make it move.

The next chapter ties together all of the material you've already read and shows you how to write code that will make a robot wander around. As discussed in this chapter, all of the code we will write in this book can be targeted either at real robots or at simulated robots. Onward!

CHAPTER 7

Wander-bot

The first chapters of this book introduced many of the abstract ROS concepts used for communication between modules, such as topics, services, and actions. Then, the previous chapter introduced many of the sensing and actuation subsystems commonly found in modern robots. In this chapter, we will put these concepts together to create a robot that can wander around its environment. This might not sound terribly earth-shattering, but such a robot is actually capable of doing meaningful work: there is an entire class of tasks that are accomplished by driving across the environment. For example, many vacuuming or other floor-cleaning tasks can be accomplished by cleverly designed and carefully tuned algorithms where the robot, carrying its cleaning tool, traverses its environment somewhat randomly. The robot will eventually drive over all parts of the environment, completing its task.

In this chapter, we will go step by step through the process of writing minimalist ROS-based robot control software, including creating a ROS package and testing it in simulation.

Creating a Package

First, let's create the workspace directory tree, which we will place in *~/wanderbot_ws*:

```
user@hostname$ mkdir -p ~/wanderbot_ws/src
user@hostname$ cd ~wanderbot_ws/src
user@hostname$ catkin_init_workspace
```

That's it! Next, it's just one more command to create a package in the new workspace. To create a package called `wanderbot` that uses `rospy` (the Python client for ROS) and a few standard ROS message packages, we will use the `catkin_create_pkg` command:

```
user@hostname$ cd ~/wanderbot_ws/src
user@hostname$ catkin_create_pkg wanderbot rospy geometry_msgs sensor_msgs
```

The first argument, `wanderbot`, is the name of the new package we want to create. The following arguments are the names of packages that the new package depends on. W must include these because the ROS build system needs to know the package dependencies in order to efficiently keep the builds up to date when source files change, and to generate any required installation dependencies when packages are released.

After running the `catkin_create_pkg` command, there will be a package directory called *wanderbot* inside the workspace, including the following files:

- *~/wanderbot_ws/src/wanderbot/CMakeLists.txt*, a starting point for the build script for this package
- *package.xml*, a machine-readable description of the package, including details such as its name, description, author, license, and which other packages it depends on to build and run

Now that we've created our `wanderbot` package , we can create a minimal ROS node inside of it. In the previous chapters, we were just sending generic messages between nodes, such as strings or integers. Now, we can send something robot-specific. The following code will send a stream of motion commands 10 times per second, alternating every 3 seconds between driving and stopping. When driving, the program will send forward velocity commands of 0.5 meters per second. When stopped, it will send commands of 0 meters per second. This program is shown in Example 7-1.

Example 7-1. Red light! Green light!

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist

cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1) ❶
rospy.init_node('red_light_green_light')

red_light_twist = Twist() ❷
green_light_twist = Twist()
green_light_twist.linear.x = 0.5 ❸

driving_forward = False
light_change_time = rospy.Time.now()
rate = rospy.Rate(10)

while not rospy.is_shutdown():
  if driving_forward:
    cmd_vel_pub.publish(green_light_twist) ❹
```

```
  else:
    cmd_vel_pub.publish(red_light_twist)
  if light_change_time > rospy.Time.now(): ❺
    driving_forward = not driving_forward
    light_change_time = rospy.Time.now() + rospy.Duration(3)
  rate.sleep() ❻
```

❶ The `queue_size=1` argument tells `rospy` to only buffer a single outbound message. In case the node sending the messages is transmitting at a higher rate than the receiving node(s) can receive them, `rospy` will simply drop any messages beyond the `queue_size`.

❷ The message constructors set all fields to zero. Therefore, the `red_light_twist` message tells a robot to stop, since all of its velocity subcomponents are zero.

❸ The `x` component of the linear velocity in a `Twist` message is, by convention, aligned in the direction the robot is facing, so this line means "drive straight ahead at 0.5 meters per second."

❹ We need to continually publish a stream of velocity command messages, since most mobile base drivers will time out and stop the robot if they don't receive at least several messages per second.

❺ This branch checks the system time and toggles the red/green light periodically.

❻ Without this call to `rospy.sleep()` the code would still run, but it would send far too many messages, and take up an entire CPU core!

A lot of Example 7-1 is just setting up the system and its data structures. The most important function of this program is to change behavior every 3 seconds from driving to stopping. This is performed by the three-line block reproduced here, which uses `rospy.Time` to measure the duration since the last change of behavior:

```
if light_change_time > rospy.Time.now():
  driving_forward = not driving_forward
  light_change_time = rospy.Time.now() + rospy.Duration(3)
```

Like all Python scripts, it is convenient to make it an executable so that we can invoke the script directly on the command line:

```
user@hostname$ chmod +x red_light_green_light.py
```

Now, we can use our program to control a simulated robot. But first, we need to make sure that the Turtlebot simulation stack is installed:

```
user@hostname$ sudo apt-get install ros-indigo-turtlebot-gazebo
```

We are now ready to instantiate a Turtlebot in the simulator. We'll use a simple world to start, by typing this in a new terminal window (remember to hit the Tab key often when typing ROS shell commands for autocompletion):

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

Figure 7-1 shows the initial TurtleBot world, in which a few obstacles are strewn about.

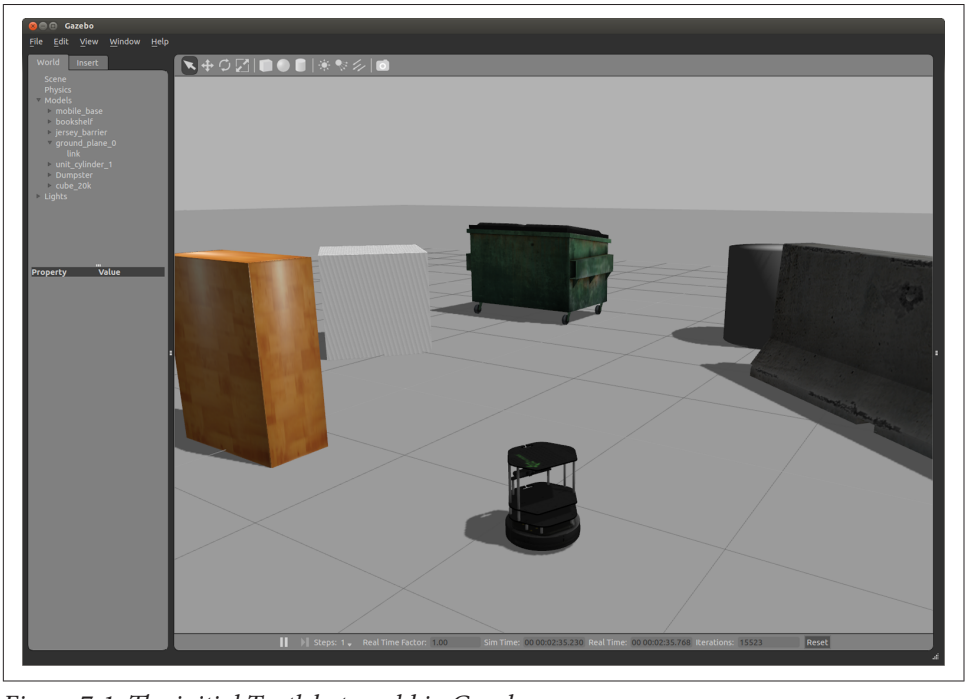

Figure 7-1. The initial Turtlebot world in Gazebo

Now, in a different terminal window, let's fire up our control node:

```
user@hostname$ ./red_light_green_light.py cmd_vel:=cmd_vel_mux/input/teleop
```

The `cmd_vel` remapping is necessary so that we are publishing our `Twist` messages to the topic that the Turtlebot software stack is expecting. Although we could have declared our `cmd_vel_pub` to publish to this topic in the *red_light_green_light.py* source code, our usual goal is to write ROS nodes that are as generic as possible, and in this case, we can easily remap `cmd_vel` to whatever is required by any robot's software stack.

When `red_light_green_light.py` is running, you should now see a Turtlebot alternating every second between driving forward and stopping. Progress! When you are

bored with it, just give a Ctrl-C to the newly created node as well as the TurtleBot simulation.

Reading Sensor Data

Blindly driving around is fun, but we typically want robots to use sensor data. Fortunately, streaming sensor data into ROS nodes is quite easy. Whenever we want to receive a topic in ROS, it's often helpful to first just echo it to the console, to make sure that it is actually being published under the topic name we expect and to confirm that we understand the data type.

In the case of Turtlebot, we want to see something like a laser scan: a linear vector of ranges from the robot to the nearest obstacles in various directions. To save on cost, sadly, the Turtlebot does not have a real laser scanner. It does, however, have a Kinect depth camera, and the Turtlebot software stack extracts the middle few rows of the Kinect's depth image, does a bit of filtering, and then publishes the data as `sensor_msgs/LaserScan` messages on the `scan` topic. This means that from the standpoint of the high-level software, the data shows up exactly like "real" laser scans on more expensive robots. The only difference is that the field of view is just narrower, and the maximum detectable range is quite a bit shorter than with typical laser scanners. To illustrate this difference in field of view, compare the Gazebo simulation rendering shown in Figure 7-2 to the actual simulated laser-scanner stream shown in Figure 7-3. Although the Turtlebot is able to perceive the obstacle directly in front of it, the obstacle on its right side is mostly out of view. Such are the trade-offs involved with using low-cost depth cameras as navigation sensors!

To start using the sensor data, we can just dump the `scan` topic to the console to verify that the simulated laser scanner is working. First, fire up a Turtlebot simulation, if one isn't already running:

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

Then, in another console, use `rostopic` to echo the topic:

```
user@hostname$ rostopic echo scan
```

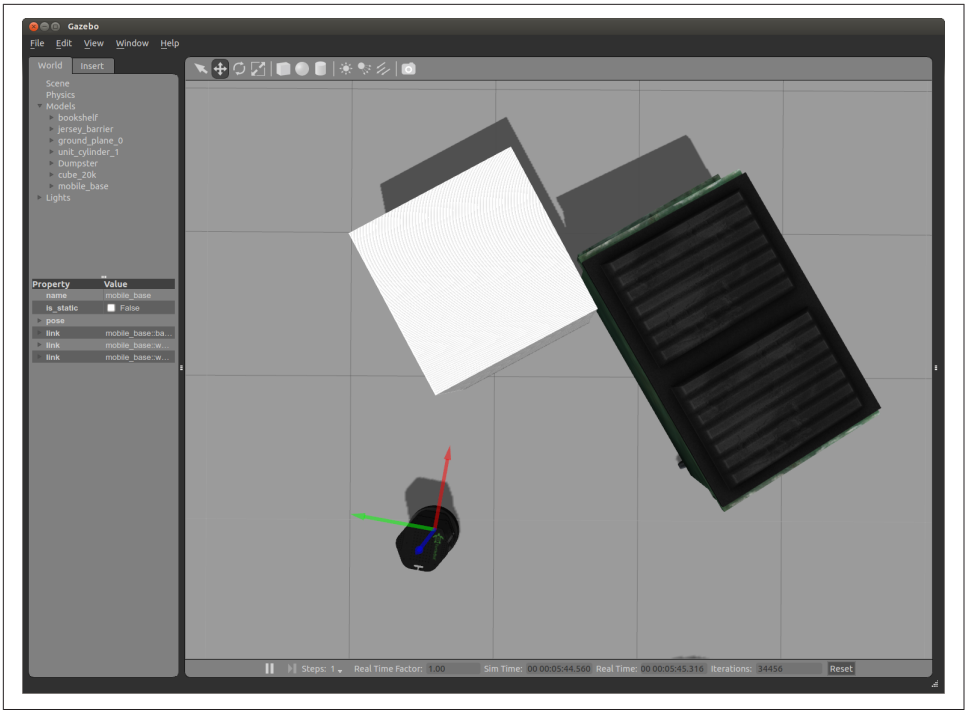

Figure 7-2. A bird's-eye Gazebo view of a Turtlebot in front of two obstacles

This will print a continuous stream of text representing the `LaserScan` messages. When you're bored, press Ctrl-C to stop it. Most of the text is the `ranges` member of the `LaserScan` message, which is exactly what we are interested in: the `ranges` array contains the range from the Turtlebot to the nearest object at bearings easily computed from the `ranges` array index. Specifically, if the message instance is named `msg`, we can compute the bearing for a particular range estimate as follows, where `i` is the index into the `ranges` array:

```
bearing = msg.angle_min + i * msg.angle_max / len(msg.ranges)
```

To retrieve the range to the nearest obstacle directly in front of the robot, we will select the middle element of the `ranges` array:

```
range_ahead = msg.ranges[len(msg.ranges)/2]
```

Or, to return the range of the closest obstacle detected by the scanner:

```
closest_range = min(msg.ranges)
```

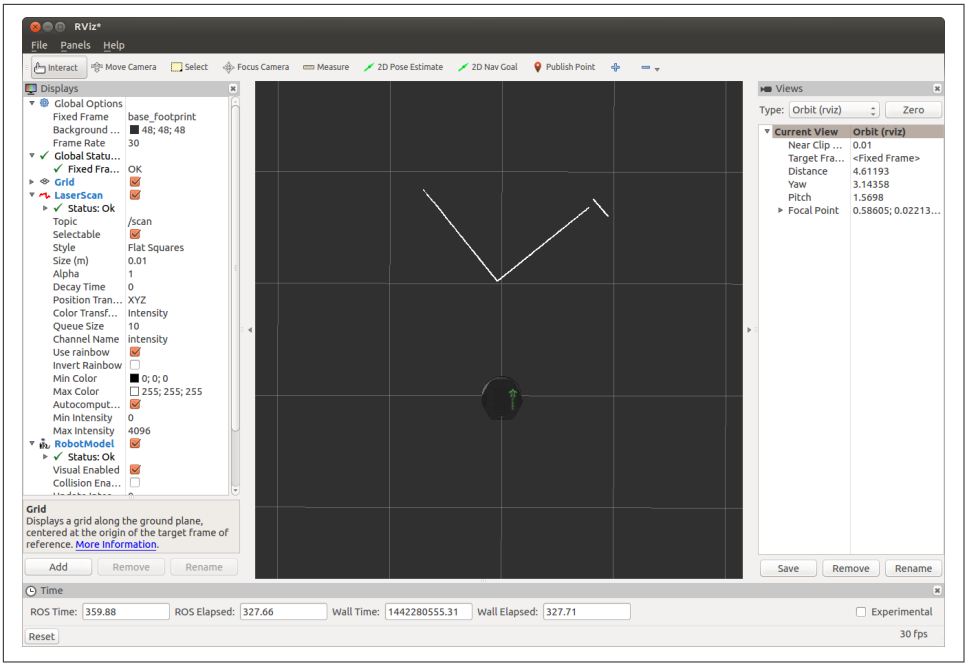

Figure 7-3. A bird's-eye view of the same scene as Figure 7-2, rendering the simulated laser scan extracted from the simulated Kinect data of the Turtlebot—the object directly in front of the robot is visible, but the object to its right is mostly out of view

This signal chain is deceptively complex: we are picking out elements of an *emulated* laser scan, which is itself produced by picking out a few of the middle rows of the Turtlebot's Kinect depth camera, which is itself generated in Gazebo by backprojecting rays into a simulated environment! It's hard to overemphasize the utility of simulation for robot software development.

Example 7-2 is a complete ROS node that prints the distance to an obstacle directly in front of the robot.

Example 7-2. range_ahead.py

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan

def scan_callback(msg):
  range_ahead = msg.ranges[len(msg.ranges)/2]
  print "range ahead: %0.1f" % range_ahead

rospy.init_node('range_ahead')
```

```
scan_sub = rospy.Subscriber('scan', LaserScan, scan_callback)
rospy.spin()
```

This little program shows how easy it is to connect to data streams in ROS and process them in Python. The `scan_callback()` function is called each time a new message arrives on the `scan` topic. This callback function then prints the range measured to the object directly in front of the robot by picking the middle element of the `ranges` field of the `LaserScan` message:

```
    def scan_callback(msg):
      range_ahead = msg.ranges[len(msg.ranges)/2]
      print "range ahead: %0.1f" % range_ahead
```

We can experiment with this program in Gazebo by dragging and rotating the Turtlebot around in the world. Click the Move icon in the Gazebo toolbar to enter Move mode, and then click and drag the Turtlebot around the scene. The terminal running `range_ahead.py` will print a continually changing stream of numbers indicating the range (in meters) from the Turtlebot to the nearest obstacle (if any) directly in front of it.

Gazebo also has a Rotate tool that will (by default) rotate a model about its vertical axis. Both the Move and Rotate tools will immediately affect the output of the *range_ahead.py* program, since the simulation (by default) stays running while models are being dragged and rotated.

Sensing and Actuation: Wander-bot!

We have now written *red_light_green_light.py*, which causes Turtlebot to drive *open-loop*, and *range_ahead.py*, which uses the Turtlebot's sensors to estimate the range to the nearest object directly in front of the Turtlebot. We can put these two capabilities together and write *wander.py*, shown in Example 7-3, which will cause the Turtlebot to drive straight ahead until it sees an obstacle within 0.8 meters or times out after 30 seconds. Then, the Turtlebot will stop and spin to a new heading. It will continue doing those two things until the end of time or Ctrl-C, whichever comes first.

Example 7-3. wander.py

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

def scan_callback(msg):
  global g_range_ahead
  g_range_ahead = min(msg.ranges)

g_range_ahead = 1 # anything to start
```

```
scan_sub = rospy.Subscriber('scan', LaserScan, scan_callback)
cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
rospy.init_node('wander')
state_change_time = rospy.Time.now()
driving_forward = True
rate = rospy.Rate(10)

while not rospy.is_shutdown():
  if driving_forward:
    if (g_range_ahead < 0.8 or rospy.Time.now() > state_change_time):
      driving_forward = False
      state_change_time = rospy.Time.now() + rospy.Duration(5)
  else: # we're not driving_forward
    if rospy.Time.now() > state_change_time:
      driving_forward = True # we're done spinning, time to go forward!
      state_change_time = rospy.Time.now() + rospy.Duration(30)
  twist = Twist()
  if driving_forward:
    twist.linear.x = 1
  else:
    twist.angular.z = 1
  cmd_vel_pub.publish(twist)

  rate.sleep()
```

As will always be the case with ROS Python programs, we start by importing `rospy` and the ROS message types we'll need: the `Twist` and `LaserScan` messages. Since this program is so simple, we'll just use a global variable called `g_range_ahead` to store the minimum range that our (simulated) laser scanner detects in front of the robot. This makes the `scan_callback()` function very simple; it just copies out the range to our global variable. And yes, this is horrible programming practice in complex programs, but for this small example, we'll pretend it's OK.

We start the actual program by creating a subscriber to `scan` and a publisher to `cmd_vel`, as we did previously. We also set up two variables that we'll use in our controller logic: `state_change_time` and `driving_forward`. The `rate` variable is a helpful construct in `rospy`: it helps create loops that run at a fixed frequency. In this case, we'd like to run our controller at 10 Hz, so we construct a `rospy.Rate` object by passing 10 to its constructor. Then, we call `rate.sleep()` at the end of our main loop; each time through, `rospy` will adjust the amount of actual sleeping time so that we run at something close to 10 Hz on average. The actual amount of sleeping time will depend on what else is being done in the control loop and the speed of the computer; we can just call `rospy.Rate.sleep()` and not worry about it.

The actual control loop is kept as simple as possible. The robot is in one of two states: `driving_forward` or `not driving_forward`. When in the `driving_forward` state, the

robot keeps driving until it either sees an obstacle within 0.8 meters or times out after 30 seconds, after which it transitions to the `not driving_forward` state:

```
if (g_range_ahead < 0.8 or rospy.Time.now() > state_change_time):
  driving_forward = False
  state_change_time = rospy.Time.now() + rospy.Duration(5)
```

When the robot is in the `not driving_forward` state, it simply spins in place for five seconds, then transitions back to the `driving_forward` state:

```
if rospy.Time.now() > state_change_time:
  driving_forward = True # we're done spinning, time to go forward!
  state_change_time = rospy.Time.now() + rospy.Duration(30)
```

As before, we can quickly test our program in a Turtlebot simulation. Let's start one up:

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

Then, in a separate console, we can make `wander.py` executable and run it:

```
user@hostname$ chmod +x red_light_green_light.py
user@hostname$ ./wander.py cmd_vel:=cmd_vel_mux/input/teleop
```

The TurtleBot will wander around aimlessly, while avoiding collisions with obstacles it can see. Hooray!

Summary

In this chapter, we first created an open-loop control system in *red_light_green_light.py* that started and stopped the Turtlebot based on a simple timer. Then, we saw how to read the information from the Turtlebot's depth camera. Finally, we closed the loop between sensing and actuation by creating Wander-bot, a program that causes the Turtlebot to avoid obstacles and randomly wander around its environment. This brought together all of the aspects of the book thus far: the streaming data transport mechanisms of ROS, the discussion of robot sensors and actuators, and the simulation framework of Gazebo. In the next chapter, we will start making things more complex by listening to user input, as we create Teleop-bot.

PART II

Moving Around Using ROS

CHAPTER 8
Teleop-bot

The previous section covered fundamental concepts in ROS, provided a brief overview of subsystems common to many robots, and finished with Wander-bot, a program that drove a Turtlebot around aimlessly. In this section of the book, we will show how to build a series of robots that become more and more sophisticated in their motions, culminating with a state-of-the-art 2D navigation system. We will then conclude this section by showing how to move manipulator arms using common ROS packages.

This chapter will describe how to drive a robot around via *teleoperation*. Although the term "robot" often brings up images of *fully autonomous* robots that are able to make their own decisions in all situations, there are many domains in which close human guidance is standard practice due to a variety of factors. Since teleoperated systems are, generally speaking, simpler than autonomous systems, they make a natural starting point. In this chapter, we will construct progressively more complex teleoperation systems.

As discussed in the previous chapter, we drive a Turtlebot by publishing a stream of `Twist` messages. Although the `Twist` message has the ability to describe full 3D motion, when operating differential-drive planar robots, we only need to populate two members: the linear (forward/backward) velocity, and the angular velocity about the vertical axis, which can also be called *yaw rate* and is simply the measure of how quickly the robot is spinning. From those two fields, it is then an exercise in trigonometry to compute the required wheel velocities of the robot as a function of the spacing of the wheels and their diameter. This calculation is usually done at low levels in the software stack, either in the robot's device driver or in the firmware of a microcontroller onboard the robot. From the teleoperation software's perspective, we simply command the linear and angular velocities in meters per second and radians per second, respectively.

Given that we need to produce a stream of velocity commands to move the robot, the next question is, how can we elicit these commands from the robot operator? There are a wide variety of approaches to this problem, and naturally we should start with the simplest approach to program: keyboard input.

Development Pattern

Throughout the remainder of the book, we will encourage a development pattern that makes use of the ROS debugging tools wherever possible. Since ROS is a distributed system with topic-based communications, we can quickly create testing environments to help our debugging, so that we are only starting and stopping a single piece of the system every time we need to tweak a bit of code. Structuring our software as a collection of very small message-passing programs makes it easier and more productive to insert ROS debugging tools into these message flows.

In the specific case of producing Teleop-bot velocity commands, we will write two programs: one that listens for keystrokes and then broadcasts them as ROS messages, and one that listens for those keystroke ROS messages and outputs `Twist` messages in response. This extra layer of indirection helps isolate the two functional pieces of this system, as well as making it easier for us, or anyone else in the open source community, to reuse the individual pieces in a completely different system. Creating a constellation of small ROS nodes often will simplify the creation of manual and (especially) automated software tests. For example, we can feed a canned sequence of keystroke messages to the node that translates between keystrokes and motion commands, comparing the output motion command with the previously defined "correct" response. Then, we can set up automated testing to verify the correct behavior as the software evolves over time.

Once we have decided the highest-level breakdown of how a task should be split into ROS nodes, the next task is to write them! As is often the case with software design, sometimes it helps to create a *skeleton* of the desired system that prints console messages or just publishes dummy messages to other nodes in the system. However, our preferred approach is to build the required collection of new ROS nodes incrementally, with a strong preference for writing *small* nodes.

Keyboard Driver

The first node we need to write for keyboard-Teleop-bot is a keyboard driver that listens for keystrokes and publishes them as `std_msgs/String` messages on the `keys` topic. There are many ways to perform this task. Example 8-1 uses the Python `termios` and `tty` libraries to place the terminal in raw mode and capture keystrokes, which are then published as `std_msgs/String` messages.

Example 8-1. key_publisher.py

```
#!/usr/bin/env python
import sys, select, tty, termios
import rospy
from std_msgs.msg import String

if __name__ == '__main__':
  key_pub = rospy.Publisher('keys', String, queue_size=1)
  rospy.init_node("keyboard_driver")
  rate = rospy.Rate(100)
  old_attr = termios.tcgetattr(sys.stdin)
  tty.setcbreak(sys.stdin.fileno())
  print "Publishing keystrokes. Press Ctrl-C to exit..."
  while not rospy.is_shutdown():
    if select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]:
      key_pub.publish(sys.stdin.read(1))
    rate.sleep()
  termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
```

This program uses the `termios` library to capture raw keystrokes, which requires working around some quirks of how Unix consoles operate. Typically, consoles buffer an entire line of text, only sending it to programs when the user presses Enter. In our case, we want to receive the keys on our program's standard input stream as soon as they are pressed. To alter this behavior of the console, we first need to save the attributes:

```
old_attr = termios.tcgetattr(sys.stdin)
tty.setcbreak(sys.stdin.fileno())
```

Now, we can continually poll the `stdin` stream to see if any characters are ready. Although we could simply block on `stdin`, that would cause our process to not fire any ROS callbacks, should we add any in the future. Thus, it is good practice to instead call `select()` with a timeout of zero, which will return immediately. We will then spend the rest of our loop time inside `rate.sleep()`, as shown in this snippet:

```
if select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]:
  key_pub.publish(sys.stdin.read(1))
rate.sleep()
```

Finally, we need to put the console back into standard mode before our program exits:

```
termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
```

To test if the keyboard driver node is operating as expected, three terminals are needed. In the first terminal, run `roscore`. In the second terminal, run the `key_publisher.py` node. In the third terminal, run `rostopic echo keys`, which will print any and all messages that it receives on the `keys` topic to the console. Then, set focus back to the second terminal by clicking on it or using window manager

shortcuts such as Alt-Tab to switch between terminals. Keystrokes in the second terminal should cause `std_msgs/String` messages to print to the console of the third terminal. Progress! When you're finished testing, press Ctrl-C in all terminals to shut everything down.

You'll notice that "normal" keys, such as letters, numerals, and simple punctuation, work as expected. However, "extended" keys, such as the arrow keys, result in `std_msgs/String` messages that are either weird symbols or multiple messages (or both). That is expected, since our minimalist `key_publisher.py` node is just pulling characters one at a time from `stdin`—and improving `key_publisher.py` is an exercise left to the motivated reader! For the remainder of this chapter, we will use just alphabetic characters.

Motion Generator

In this section, we will use the common keyboard mapping of *w*, *x*, *a*, *d*, *s* to express, respectively, that we want the robot to go forward, go backward, turn left, turn right, and stop.

As a first attempt at this problem, we'll make a ROS node that outputs a `Twist` message every time it receives a `std_msgs/String` message that starts with a character it understands, as shown in Example 8-2.

Example 8-2. keys_to_twist.py

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }

def keys_cb(msg, twist_pub):
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key
  vels = key_mapping[msg.data[0]]
  t = Twist()
  t.angular.z = vels[0]
  t.linear.x  = vels[1]
  twist_pub.publish(t)

if __name__ == '__main__':
  rospy.init_node('keys_to_twist')
  twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
  rospy.Subscriber('keys', String, keys_cb, twist_pub)
  rospy.spin()
```

This program uses a Python dictionary to store the mapping between keystrokes and the target velocities:

```
key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }
```

In the callback function for the `keys` topic, incoming keys are looked up in this dictionary. If a key is found, the target velocities are extracted from the dictionary:

```
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key
  vels = key_mapping[msg.data[0]]
```

In an effort to prevent runaway robots, most robot device drivers will automatically stop the robot if no messages are received in a few hundred milliseconds. The program in the previous listing would work, but only if it had a continual stream of keypresses to continually generate `Twist` messages for the robot driver. That would be exciting for a few seconds, but once the euphoria of "Hey, the robot is moving!" wears off, we'll be searching for improvements!

Issues such as robot firmware timeouts can be tricky to debug. As with everything in ROS (and complex systems in general), the key for debugging is to find ways to divide the system into smaller pieces and discover where the problem lies. The `rostopic` tool can help in several ways. As in the previous section, start three terminals: one with `roscore`, one with `key_publisher.py`, and one with `keys_to_twist.py`. Then, we can start a fourth terminal for various incantations of `rostopic`.

First, we can see what topics are available:

```
user@hostname$ rostopic list
```

This provides the following output:

```
/cmd_vel
/keys
/rosout
/rosout_agg
```

The last two items, `/rosout` and `/rosout_agg`, are part of the general-purpose ROS logging scheme and are always there. The other two, `cmd_vel` and `keys`, are what our programs are publishing. Now, let's dump the `cmd_vel` data stream to the console:

```
user@hostname$ rostopic echo cmd_vel
```

Each time a valid key is pressed in the console with `key_publisher.py`, the `rostopic` console should print the contents of the resulting `Twist` message published by `keys_to_twist.py`. Progress! As always with ROS console tools, simply press Ctrl-C to exit. Next, let's use `rostopic hz` to compute the average rate of messages:

```
user@hostname$ rostopic hz cmd_vel
```

The `rostopic hz` command will compute an average of the rate of messages on a topic every second and print those estimates to the console. With `keys_to_twist.py`, this estimate will almost always be zero, with minor bumps up and down each time a key is pressed in the keyboard driver console.

The `rostopic` tools are your friends! Virtually every ROS programming and (especially) debugging session includes some usage of `rostopic` to rapidly introspect the system and verify that data is flowing as expected.

To make this node useful for robots that require a steady stream of velocity commands, we will output a `Twist` message every 100 milliseconds, or at a rate of 10 Hz, by simply repeating the last motion command if a new key was not pressed. Although we could do something like this by using a `sleep(0.1)` call in the `while` loop, this would only ensure that the loop runs *no faster* than 10 Hz; the timing results would likely have quite a bit of variance since the scheduling and execution time of the loop itself are not taken into account. Because computers have widely varying clock speeds and overall computational performance, the exact amount of CPU time that a loop would need to sleep to maintain a particular update rate is not knowable ahead of time. Looping tasks are thus better accomplished with the ROS *rate* construct, which continually estimates the time spent processing the loop to obtain more consistent results, as shown in Example 8-3.

Example 8-3. keys_to_twist_using_rate.py

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }
g_last_twist = None

def keys_cb(msg, twist_pub):
  global g_last_twist
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key
  vels = key_mapping[msg.data[0]]
  g_last_twist.angular.z = vels[0]
  g_last_twist.linear.x  = vels[1]
  twist_pub.publish(g_last_twist)
```

```
if __name__ == '__main__':
  rospy.init_node('keys_to_twist')
  twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
  rospy.Subscriber('keys', String, keys_cb, twist_pub)
  rate = rospy.Rate(10)
  g_last_twist = Twist() # initializes to zero
  while not rospy.is_shutdown():
    twist_pub.publish(g_last_twist)
    rate.sleep()
```

Now, when the `keys_to_twist_using_rate.py` node is running, we will see a quite consistent 10 Hz message stream when we run `rostopic hz cmd_vel`. This can be seen using a separate console running `rostopic echo cmd_vel`, as in the previous section. The key difference between this program and the previous one is the use of `rospy.Rate()`:

```
      rate = rospy.Rate(10)
      g_last_twist = Twist() # initializes to zero
      while not rospy.is_shutdown():
        twist_pub.publish(g_last_twist)
        rate.sleep()
```

When debugging low-dimensional data, such as the velocity commands sent to a robot, it is often useful to plot the data stream as a time series. ROS provides a command-line tool called `rqt_plot` that can accept *any* numerical data message stream and plot it graphically in real time.

To create an `rqt_plot` visualization, we need to send `rqt_plot` the exact message field that we want to see plotted. To find this field name, we can use several methods. The simplest is to look at the output of `rostopic echo`. This is always printed in YAML, a simple whitespace-based markup format. For example, `rostopic echo cmd_vel` will print a series of records of this format:

```
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

Nested structures are indicated by whitespace: first, the `linear` field structure has field names `x`, `y`, `z`; this is followed by the `angular` field structure, with the same members.

Alternatively, we can discover the topic data type using `rostopic`:

```
user@hostname$ rostopic info cmd_vel
```

This will print quite a bit of information about the topic publishers and subscribers, as well as stating that the `cmd_vel` topic is of type `geometry_msgs/Twist`. With this data type name, we can use the `rosmsg` command to print the structure:

```
user@hostname$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

This console output shows us that the `linear` and `angular` members of the `Twist` message are of type `geometry_msgs/Vector3`, which has fields named `x`, `y`, and `z`. Granted, we already knew that from the `rostopic echo` output, but `rosmsg show` is sometimes a useful way of obtaining this information when we don't have a data stream available to print to the console.

Now that we know the topic name and the names of the fields, we can generate streaming plots of the linear velocity that we are publishing by using slashes to descend into the message structure and select the fields of interest. As mentioned previously, for planar differential-drive robots, the only nonzero fields in the `Twist` message will be the x-axis linear (forward/backward) velocity and the z-axis (yaw) angular velocity. We can start streaming those fields to a plot with a single command:

```
user@hostname$ rqt_plot cmd_vel/linear/x cmd_vel/angular/z
```

This plot will look something like Figure 8-1 as keys are pressed and the stream of velocity commands changes.

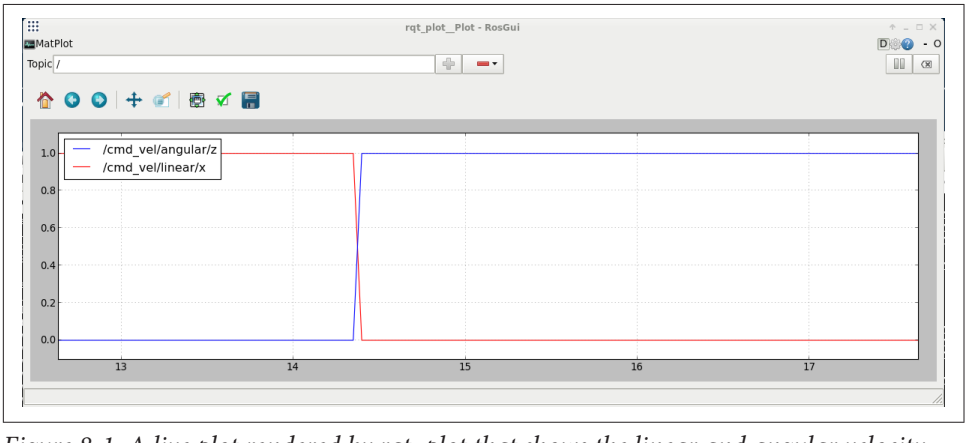

Figure 8-1. A live plot rendered by rqt_plot that shows the linear and angular velocity commands over time

We now have a pipeline built where pressing letters on the keyboard will send velocity commands to a robot, and we can view those velocities in a live plot. That's great! But there's a lot of room for improvement. First, notice in the previous plot that our velocities are always either 0, –1, or +1. ROS uses SI units throughout, which means that we are asking our robot to drive forward and backward at one meter per second and turn at one radian per second. Unfortunately, robots run at greatly varying speeds in different applications: for a robotic car, one meter per second is very slow; however, for a small indoor robot navigating a corridor, one meter per second is actually quite fast. We need a way to *parameterize* this program, so that it can be used with multiple robots. We'll do that in the next section.

Parameter Server

We can improve the *keys_to_twist_using_rate.py* program by using ROS *parameters* to specify the linear and angular velocity scales. Of course, there are countless ways that we can give parameters to programs. When developing robotic systems, it is often useful to set parameters in a variety of ways: at the command line when debugging, in `roslaunch` files, from graphical interfaces, from other ROS nodes, or even in separate parameter files to cleanly define behavior for multiple platforms or environments. The ROS master, often called `roscore`, includes a *parameter server* that can be read or written by all ROS nodes and command-line tools. The parameter server can support quite sophisticated interactions, but for our purposes in this chapter, we will only be setting parameters at the command line when running our teleoperation nodes.

The parameter server is a generic key/value store. There are many strategies for how to name parameters, but for our teleoperation node, we want a *private* parameter name. In ROS, a private parameter name is still publicly accessible; the notion of "private" simply means that its full name is formed by appending the parameter name to the node's name. This ensures that no name clashes can occur, because node names are always unique (see "Names, Namespaces, and Remapping" on page 22). For example, if our node name is `keys_to_twist`, we can have private parameters named `keys_to_twist/linear_scale` and `keys_to_twist/angular_scale`.

To set private parameters on the command line at the time the node is launched, prepend the parameter name with an underscore and set its value using `:=` syntax, as follows:

```
./keys_to_twist_parameterized.py _linear_scale:=0.5 _angular_scale:=0.4
```

This would set the `keys_to_twist/linear_scale` parameter to 0.5 and the `keys_to_twist/angular_scale` parameter to 0.4, immediately before the node is launched. These parameter values are then returned by the `has_param()` and `get_param()` calls, as shown in Example 8-4.

Example 8-4. keys_to_twist_parameterized.py

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }
g_last_twist = None
g_vel_scales = [0.1, 0.1] # default to very slow

def keys_cb(msg, twist_pub):
  global g_last_twist, g_vel_scales
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key
  vels = key_mapping[msg.data[0]]
  g_last_twist.angular.z = vels[0] * g_vel_scales[0]
  g_last_twist.linear.x  = vels[1] * g_vel_scales[1]
  twist_pub.publish(g_last_twist)

if __name__ == '__main__':
  rospy.init_node('keys_to_twist')
  twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
  rospy.Subscriber('keys', String, keys_cb, twist_pub)
  g_last_twist = Twist() # initializes to zero
  if rospy.has_param('~linear_scale'):
    g_vel_scales[1] = rospy.get_param('~linear_scale')
  else:
    rospy.logwarn("linear scale not provided; using %.1f" %\
                  g_vel_scales[1])

  if rospy.has_param('~angular_scale'):
    g_vel_scales[0] = rospy.get_param('~angular_scale')
  else:
    rospy.logwarn("angular scale not provided; using %.1f" %\
                  g_vel_scales[0])

  rate = rospy.Rate(10)
  while not rospy.is_shutdown():
    twist_pub.publish(g_last_twist)
    rate.sleep()
```

At startup, this program queries the parameter server using `rospy.has_param()` and `rospy.get_param()`, and outputs a warning if the specified parameter was not set:

```
    if rospy.has_param('~linear_scale'):
      g_vel_scales[1] = rospy.get_param('~linear_scale')
    else:
      rospy.logwarn("linear scale not provided; using %.1f" %\
                    g_vel_scales[1])
```

This warning is printed using the ROS logging system, which has a few benefits over a standard Python `print()` call. First, the ROS logging calls, such as `logwarn()`, `loginfo()`, and `logerror()`, print colorized text to the console. That may sound inconsequential, but it actually can be quite useful when watching or scrolling for warnings or errors in a noisy console stream. The ROS logging calls can also (optionally) be routed to a centralized console of warnings and errors, so that the warning and error streams from large, complex collections of nodes can be monitored more easily.

The warning text produced by `rospy.logwarn()` also prepends a timestamp:

```
[WARN] [WallTime: 1429164125.989] linear scale not provided. Defaulting to 0.1
[WARN] [WallTime: 1429164125.989] angular scale not provided. Defaulting to 0.1
```

The `get_param()` function optionally accepts a second parameter, serving as a default parameter when the parameter key is not available on the parameter server. In many cases, using this optional second parameter can shorten code and provides an appropriate level of functionality. For general-purpose nodes such as `keys_to_twist.py` that want a parameter to be explicitly defined, however, using `has_param()` to determine the existence of an explicit parameter definition can be useful.

The syntax to use `keys_to_twist_parameterized.py` with explicit command-line parameters is as follows:

```
./keys_to_twist_parameterized.py _linear_scale:=0.5 _angular_scale:=0.4
```

The resulting stream of `Twist` messages is scaled as expected: for example, pressing *w* (move forward) in the console running `key_publisher.py` will result in a stream of these messages appearing in the `rostopic echo cmd_vel` output:

```
linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

Each time we launch `keys_to_twist_parameterized.py`, we can specify the desired maximum velocities for our robot. Even more conveniently, we can put these parameters into launch files so that we don't have to remember them! But first, we need to deal with the physics problem of finite acceleration, which we will address in the next section.

Velocity Ramps

Unfortunately, like all objects with mass, robots cannot start and stop instantaneously. Physics dictates that robots accelerate gradually over time. As a result, when a robot's wheel motors try to instantly jump to a wildly different velocity, typically something bad happens, such as skidding, belts slipping, "shuddering" as the robot repeatedly hits electrical current limits, or possibly even something breaking in the mechanical driveline. To avoid these problems, we should *ramp* our motion commands up and down over a finite amount of time. Often lower levels of robot firmware will enforce this, but in general, it's considered good practice not to send impossible commands to robots. Example 8-5 applies ramps to the outgoing velocity stream, to limit the instantaneous accelerations that we are asking of the motors.

Example 8-5. keys_to_twist_with_ramps.py

```
#!/usr/bin/env python
import rospy
import math
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [ 0, -1],
                'a': [ 1, 0], 'd': [-1,  0],
                's': [ 0, 0] }
g_twist_pub = None
g_target_twist = None
g_last_twist = None
g_last_send_time = None
g_vel_scales = [0.1, 0.1] # default to very slow
g_vel_ramps = [1, 1] # units: meters per second^2

def ramped_vel(v_prev, v_target, t_prev, t_now, ramp_rate):
  # compute maximum velocity step
  step = ramp_rate * (t_now - t_prev).to_sec()
  sign = 1.0 if (v_target > v_prev) else -1.0
  error = math.fabs(v_target - v_prev)
  if error < step: # we can get there within this timestep-we're done.
    return v_target
  else:
    return v_prev + sign * step  # take a step toward the target

def ramped_twist(prev, target, t_prev, t_now, ramps):
  tw = Twist()
  tw.angular.z = ramped_vel(prev.angular.z, target.angular.z, t_prev,
                            t_now, ramps[0])
  tw.linear.x = ramped_vel(prev.linear.x, target.linear.x, t_prev,
                           t_now, ramps[1])
  return tw
```

```
def send_twist():
  global g_last_twist_send_time, g_target_twist, g_last_twist,\
         g_vel_scales, g_vel_ramps, g_twist_pub
  t_now = rospy.Time.now()
  g_last_twist = ramped_twist(g_last_twist, g_target_twist,
                              g_last_twist_send_time, t_now, g_vel_ramps)
  g_last_twist_send_time = t_now
  g_twist_pub.publish(g_last_twist)

def keys_cb(msg):
  global g_target_twist, g_last_twist, g_vel_scales
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key
  vels = key_mapping[msg.data[0]]
  g_target_twist.angular.z = vels[0] * g_vel_scales[0]
  g_target_twist.linear.x  = vels[1] * g_vel_scales[1]

def fetch_param(name, default):
  if rospy.has_param(name):
    return rospy.get_param(name)
  else:
    print "parameter [%s] not defined. Defaulting to %.3f" % (name, default)
    return default

if __name__ == '__main__':
  rospy.init_node('keys_to_twist')
  g_last_twist_send_time = rospy.Time.now()
  g_twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
  rospy.Subscriber('keys', String, keys_cb)
  g_target_twist = Twist() # initializes to zero
  g_last_twist = Twist()
  g_vel_scales[0] = fetch_param('~angular_scale', 0.1)
  g_vel_scales[1] = fetch_param('~linear_scale', 0.1)
  g_vel_ramps[0] = fetch_param('~angular_accel', 1.0)
  g_vel_ramps[1] = fetch_param('~linear_accel', 1.0)

  rate = rospy.Rate(20)
  while not rospy.is_shutdown():
    send_twist()
    rate.sleep()
```

The code is a bit more complex, but the main lines of interest are in the `ramped_vel()` function, where the velocity is computed under the acceleration constraint provided as a parameter. Each time it is called, this function takes a step toward the target velocity, or, if the target velocity is within one step away, it goes directly to it:

```
def ramped_vel(v_prev, v_target, t_prev, t_now, ramp_rate):
  # compute maximum velocity step
  step = ramp_rate * (t_now - t_prev).to_sec()
  sign = 1.0 if (v_target > v_prev) else -1.0
  error = math.fabs(v_target - v_prev)
  if error < step: # we can get there within this timestep-we're done.
    return v_target
  else:
    return v_prev + sign * step  # take a step toward the target
```

At the command line, the following incantation of our teleop program will produce reasonable behavior for the Turtlebot:

```
user@hostname$ ./keys_to_twist_with_ramps.py _linear_scale:=0.5\
    _angular_scale:=1.0_linear_accel:=1.0 _angular_accel:=1.0
```

The motion commands we are sending the Turtlebot are now physically possible to achieve, as shown in Figure 8-2, since they take nonzero time to ramp up and down. Using the `rqt_plot` program as shown previously, we can generate a live plot of the system:

```
user@hostname$ rqt_plot cmd_vel/linear/x cmd_vel/angular/z
```

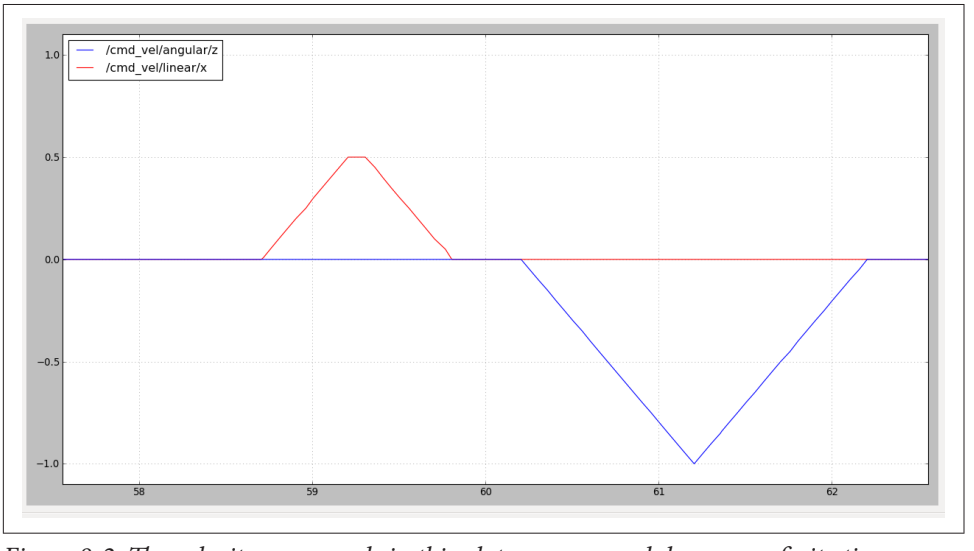

Figure 8-2. The velocity commands in this plot ramp up and down over finite time, allowing this trajectory to be physically achievable

To reiterate: even if we were to give instantaneously changing or "step" commands to the Turtlebot, somewhere in the signal path, or in the physics of the mechanical system, the step commands would be slowed into ramps. The advantage to doing this in higher-level software is that there is simply more visibility into what is happening, and hence a better understanding of the behavior of the system.

Let's Drive!

Now that we have reasonable `Twist` messages streaming from our teleop program over the `cmd_vel` topic, we can drive some robots. Let's start by driving a Turtlebot. Thanks to the magic of robot simulation, we can get a Turtlebot up and running with a single command:

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

This will launch a Gazebo instance with a world similar to that shown in Figure 8-3, as well as emulating the software and firmware of the Turtlebot behind the scenes.

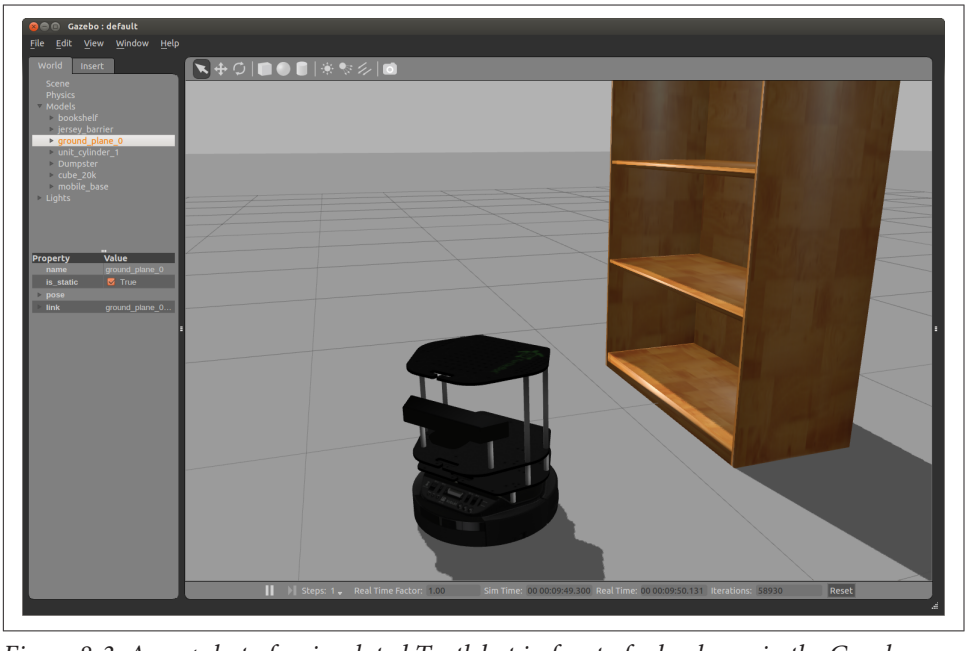

Figure 8-3. A snapshot of a simulated Turtlebot in front of a bookcase in the Gazebo simulator

Next, we want to run our teleop program, which broadcasts `Twist` messages on the `cmd_vel` topic:

```
user@hostname$ ./keys_to_twist_with_ramps.py
```

However, if we do this, it won't work! Why? Because the Turtlebot looks for its `Twist` motion messages on a different topic. This is an extremely common problem to debug in distributed robotic software systems, or in any large software system, for that matter. We will describe a variety of tools for debugging these types of problems in a later chapter. For now, however, to make the Turtlebot simulator work, we need to publish `Twist` messages to a topic named `cmd_vel_mux/input/teleop`. That is, we need to *remap* our `cmd_vel` message so that they are published on that topic instead. We can use the ROS remapping syntax to do this on the command line, without changing our source code:

```
user@hostname$ ./keys_to_twist_with_ramps.py cmd_vel:=cmd_vel_mux/input/teleop
```

We can now drive the Turtlebot around in Gazebo using the *w*, *a*, *s*, *d*, *x* buttons on the keyboard. Hooray!

This style of teleoperation is similar to how remote-controlled cars work: the teleoperator maintains line-of-sight with the robot, sends motion commands, observes how they affect the robot and its environment, and reacts accordingly. However, it is often impossible or undesirable to maintain line-of-sight contact with the robot. This requires the teleoperator to visualize the robot's sensors and see the world through the "eyes" of the robot. ROS provides several tools to simplify development of such systems, including `rviz`, which will be described in the following section.

rviz

`rviz` stands for *ROS visualization*. It is a general-purpose 3D visualization environment for robots, sensors, and algorithms. Like most ROS tools, it can be used for any robot and rapidly configured for a particular application. For teleoperation, we want to be able to see the camera feed of the robot. First, we will start from the configuration described in the previous section, where we have four terminals open: one for `roscore`, one for the keyboard driver, one for `keys_to_teleop_with_rates.py`, and one that ran a `roslaunch` script to bring up Gazebo and a simulated TurtleBot. Now, we'll need a fifth console to run `rviz`, which is in its own package, also called `rviz` :

```
user@hostname$ rosrun rviz rviz
```

`rviz` can plot a variety of data types streaming through a typical ROS system, with heavy emphasis on the three-dimensional nature of the data. In ROS, all forms of data are attached to a *frame of reference*. For example, the camera on a Turtlebot is attached to a reference frame defined relative to the center of the Turtlebot's mobile base. The *odometry* reference frame, often called `odom`, is taken by convention to have

its origin at the location where the robot was powered on, or where its odometers were most recently reset. Each of these frames can be useful for teleoperation, but it is often desirable to have a "chase" perspective, which is immediately behind the robot and looking over its "shoulders." This is because simply viewing the robot's camera frame can be deceiving—the field of view of a camera is often much narrower than we are used to as humans, and thus it is easy for teleoperators to bonk the robot's shoulders when turning corners. A sample view of `rviz` configured to generate a chase perspective is shown in Figure 8-4. Observing the sensor data in the same 3D view as a rendering of the robot's geometry can make teleoperation more intuitive.

Like many complex graphical user interfaces (GUIs), `rviz` has a number of *panels* and *plugins* that can be configured as needed for a given task. Configuring `rviz` can take some time and effort, so the *state* of the visualization can be saved to configuration files for later reuse. Additionally, when closing `rviz`, by default the program will save its configuration to a special local file; the next time `rviz` is run, it will then instantiate and configure the same panels and plugins.

The default, unconfigured `rviz` window will appear as shown in Figure 8-5. It can be disconcerting at first, since there is nothing there! In the next few pages, we will show how to add various streams to `rviz` to end up with the visualization shown in Figure 8-4.

The first task is to choose the frame of reference for the visualization. In our case, we want a visualization perspective that is attached to the robot, so we can follow the robot as it drives around. On any given robot, there are many possible frames of reference, such as the center of the mobile base, various links of the robot's structure, or even a wheel (note that this frame would continually flip around and around, making it rather dizzying as a vantage point for `rviz`). For the purposes of teleoperation, we will select a frame of reference attached to the optical center of the Kinect depth camera on the Turtlebot. To do this, click in the table cell to the right of the "Fixed Frame" row in the upper-left panel of `rviz`. This will pop up the menu shown in the following screenshot, which contains all transform frames currently broadcasting in this ROS system. For now, select `camera_depth_frame` in the pop-up menu, as shown in Figure 8-6. Selecting the fixed frame for visualization is one of the most important configuration steps of `rviz`.

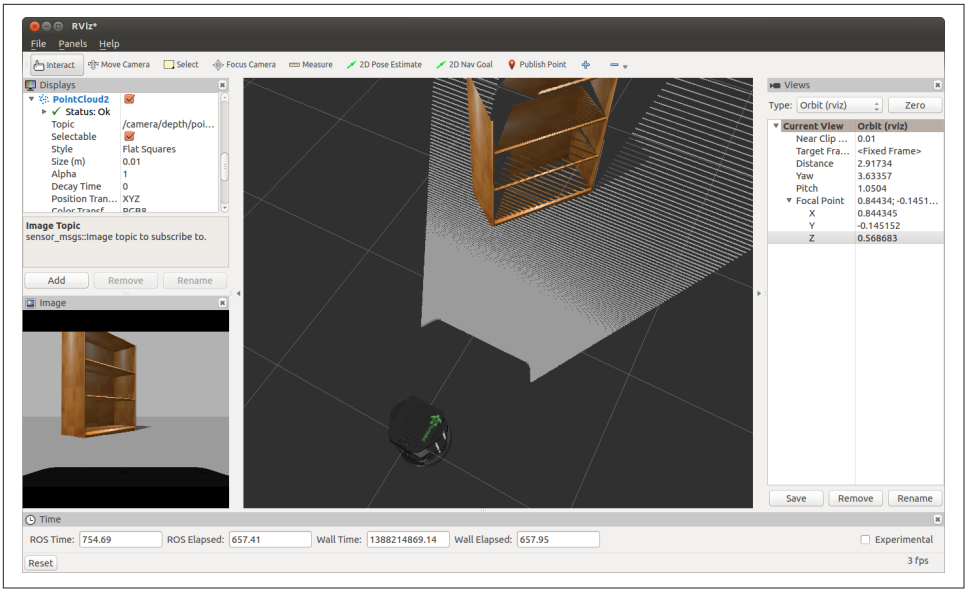

Figure 8-4. rviz configured to render the Turtlebot geometry as well as its depth camera and 2D image data

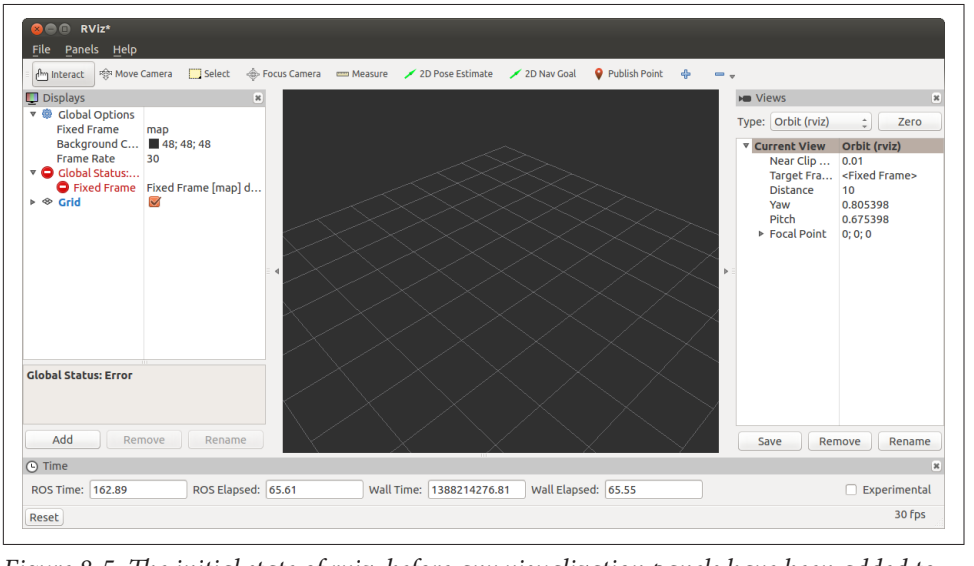

Figure 8-5. The initial state of rviz, before any visualization panels have been added to the configuration

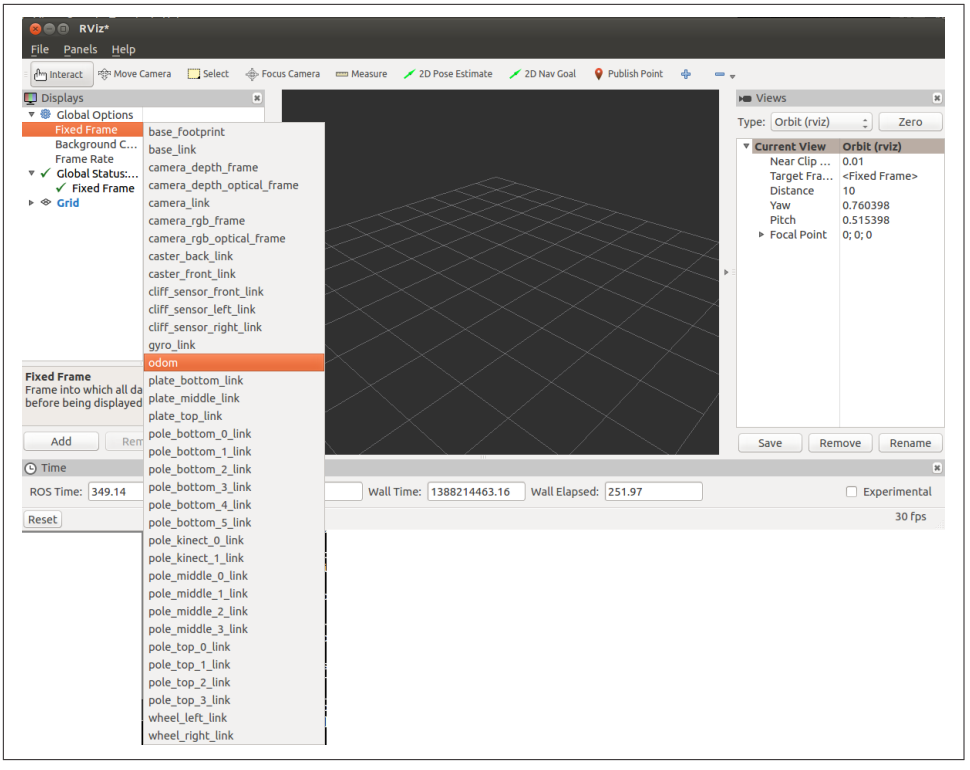

Figure 8-6. The fixed frame pop-up menu

Next, we want to view the 3D model of the robot. To accomplish this, we will insert an instance of the *robot model* plugin. Although the Turtlebot has no moving parts (other than its wheels) that we need to visualize, it is still useful to see a rendering of the robot to improve situational awareness and a get sense of scale for teleoperation. To add the robot model to the `rviz` scene, click the "Add" button on the lefthand side of the `rviz` window, approximately halfway down. This will bring up a dialog box, shown in Figure 8-7, that contains all of the available `rviz` plugins for various data types.

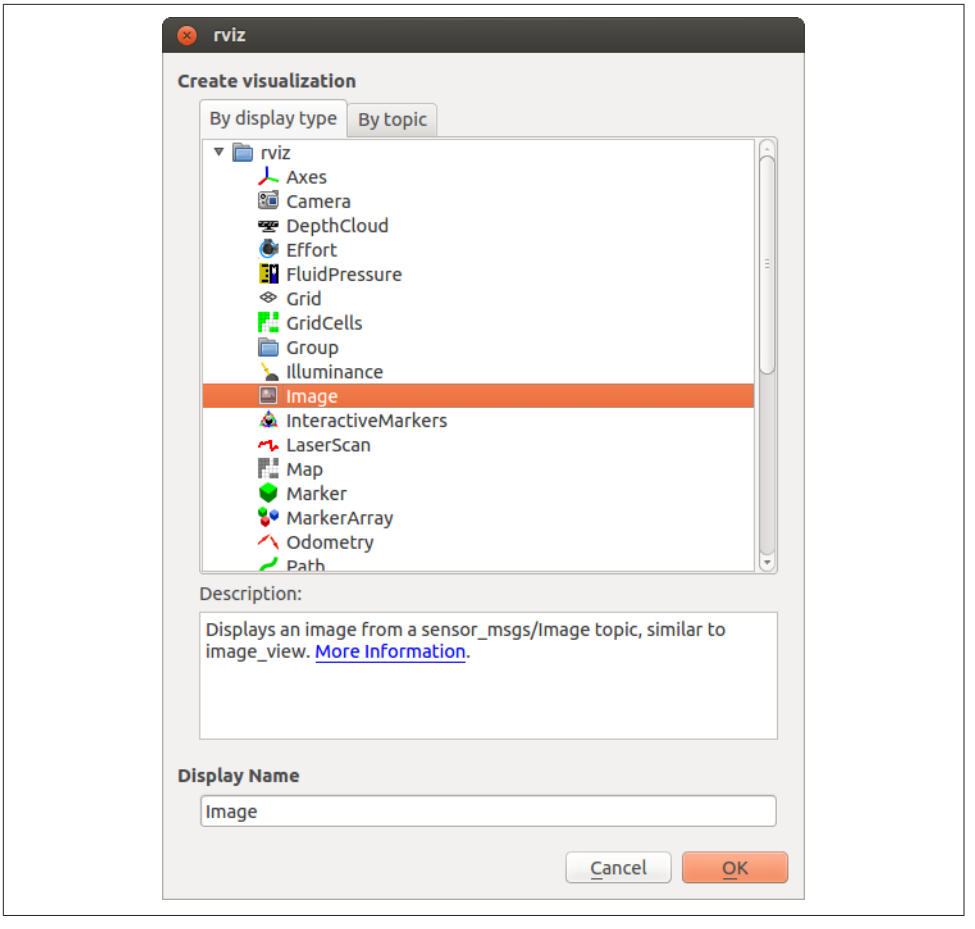

Figure 8-7. rviz dialog box used to select the data type that is currently being added to the visualization

From this dialog box, select "RobotModel" and then click "OK". Plugin instances appear in the tree-view control at the left of the `rviz` window. To configure a plugin, ensure it is expanded in the tree view. Its configurable parameters can then be edited. For the Robot Model plugin, the only configuration typically required is to enter the name of the robot model on the parameter server. However, since the ROS convention is for this to be called `robot_description`, this is autofilled and typically "just works" for single-robot applications. This will produce an `rviz` visualization similar to that shown in Figure 8-8, which is centered on a model of the Turtlebot.

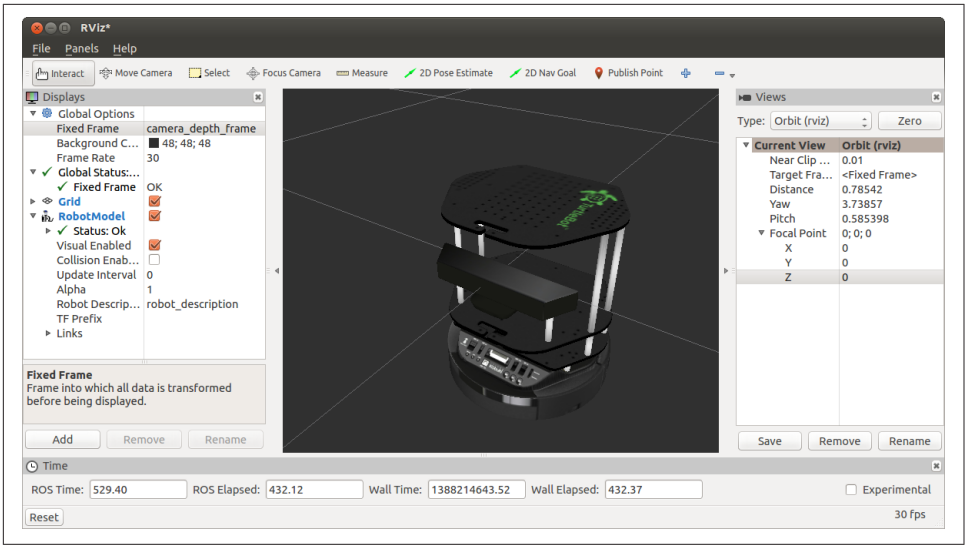

Figure 8-8. A Turtlebot model added to rviz

In order to teleoperate the Turtlebot reasonably, we need to plot its sensors. To plot the depth image from the Kinect camera on the Turtlebot, click "Add" and then select "PointCloud2" from the plugin dialog box, near the lower-left corner of `rviz`. The PointCloud2 plugin has quite a few options to configure in the tree-view control in the left pane of `rviz`. Most importantly, we need to tell the plugin which topic should be plotted. Click the space to the right of the "Topic" label, and a drop-down box will appear, showing the PointCloud2 topics currently visible on the system. Select /camera/depth/points, and the Turtlebot's point cloud should be visible, as shown in Figure 8-9.

The Kinect camera on the Turtlebot also produces a color image output, in addition to its depth image. Sometimes it can be useful for teleoperation to render both the image and the point cloud. `rviz` provides a plugin for this. Click "Add" near the lower-left corner of `rviz`, and then select "Image" from the plugin dialog box. As usual, this will instantiate the plugin, and now we need to configure it. Click on the whitespace to the right of the "Image Topic" label of the Image plugin property tree, and then select `/camera/rgb/image_raw`. The camera stream from the Turtlebot should then be plotted in the left pane of `rviz`, as shown in Figure 8-10.

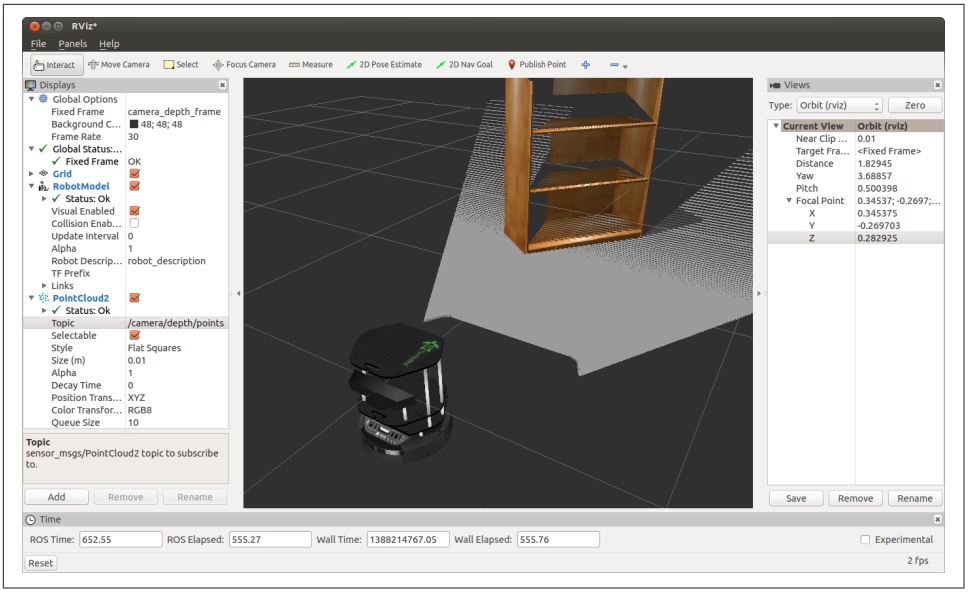

Figure 8-9. The Turtlebot's depth camera data has been added to the visualization

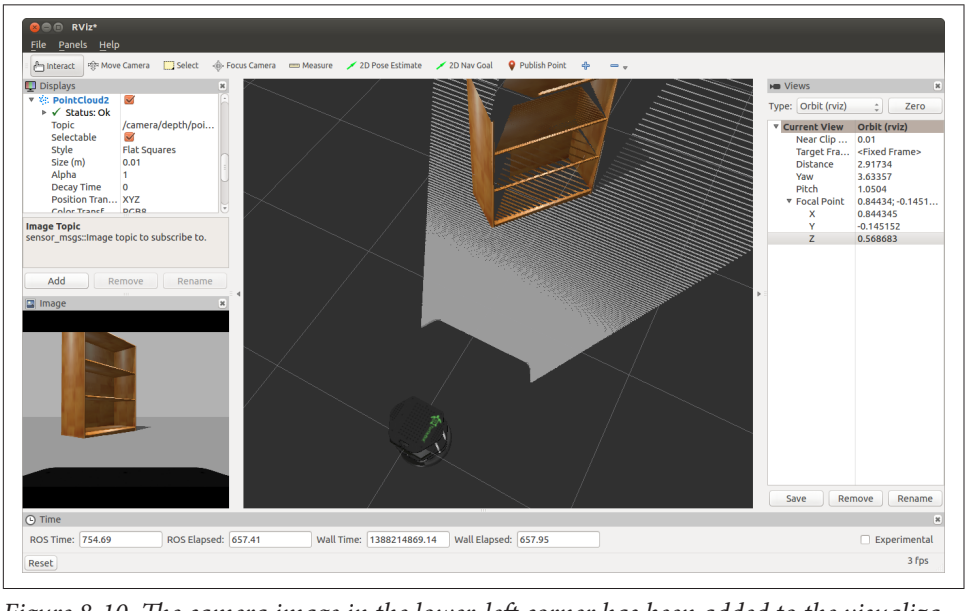

Figure 8-10. The camera image in the lower-left corner has been added to the visualization, allowing teleoperators to see the first-person perspective as well as the third-person perspective of the main window

The `rviz` interface is panelized and thus can be easily modified to suit the needs of the application. For example, we can drag the Image panel to the righthand column of the `rviz` window and resize it so that the depth image and camera image are similarly sized. We can then rotate the 3D visualization so that it is looking at the point cloud data from the side, which could be useful in some situations. An example panel configuration is shown in Figure 8-11.

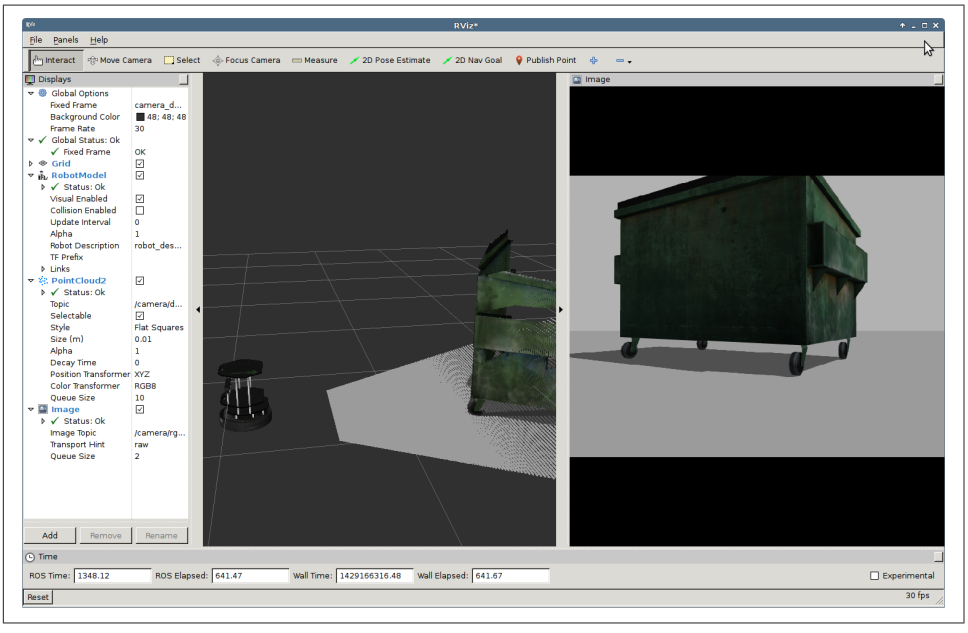

Figure 8-11. rviz panels can be dragged around to create different arrangements—here, the left panel has the third-person renderings of the depth camera data, and the visual camera is shown in the right panel

Alternatively, we can rotate the 3D scene so that it has a top-down perspective, which can be useful for driving in tight quarters. An example of this "bird's-eye" perspective is shown in Figure 8-12.

These examples just scratch the surface of what `rviz` can do! It is an extremely flexible tool that we will use throughout the remainder of the book.

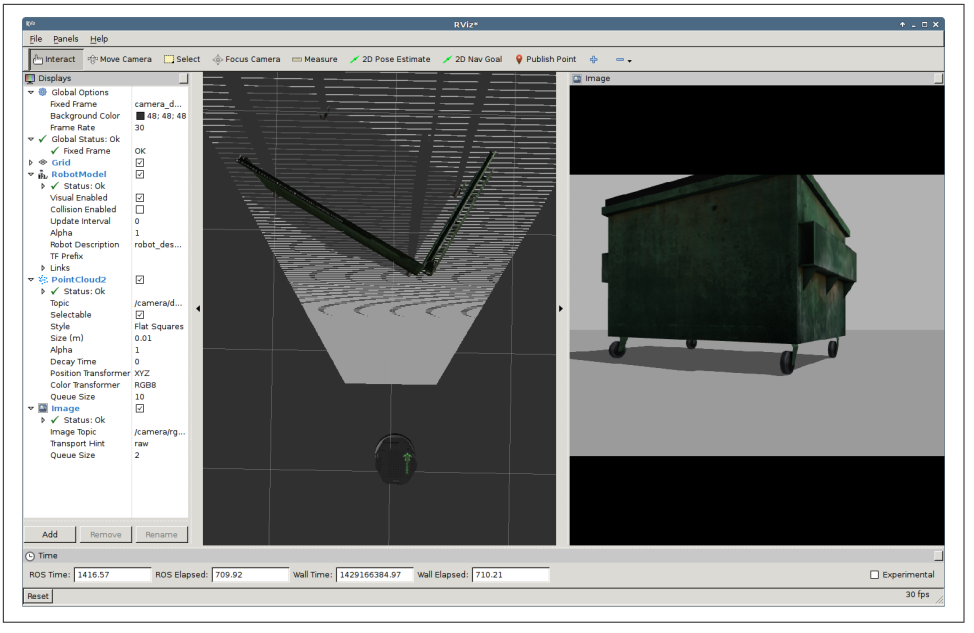

Figure 8-12. Rotating the perspective of the 3D view to create a "bird's eye" view of the environment

Summary

This chapter developed a progressively more complex keyboard-based teleoperation scheme and then showed how to connect the resulting motion commands to a Turtlebot. Finally, this chapter introduced `rviz` and showed how to quickly configure `rviz` to render point cloud and camera data, to create a teleoperation interface for a mobile robot.

Although teleoperated robots have many important applications, it is often more convenient or economical for robots to drive themselves. In the next chapter, we will describe one approach for building 2D maps, which is a necessary step for robots to start driving themselves.