Player/Stage project

USC Robotics Research Laboratory
University of Southern California
Los Angeles, California, USA

Gazebo

Version 0.4.0 User Manual

Andrew Howard
ahoward@usc.edu

Nathan Koenig
nkoenig@usc.edu

May 31, 2004

Contents

1 Introduction **1**
- 1.1 What is Gazebo? . . . 1
- 1.2 Stage and Gazebo . . . 1
- 1.3 Getting Gazebo . . . 1
- 1.4 System Requirements . . . 2
- 1.5 Bugs . . . 2
- 1.6 License . . . 2
- 1.7 Acknowledgments . . . 2

I User Guide **3**

2 General Usage **4**
- 2.1 Installing Third-Party Dependencies . . . 4
- 2.2 Building and Installing Gazebo . . . 4
- 2.3 Starting Gazebo . . . 5
- 2.4 Working with Player . . . 5
- 2.5 Command Line Options . . . 6

3 The World File **7**
- 3.1 Key Concepts and Basic Syntax . . . 7
- 3.2 Graphical User Interface . . . 8
- 3.3 Canonical World File Layout . . . 8
- 3.4 Coordinate Systems and Units . . . 9

4 Working with Player **10**
- 4.1 Setting up the Simulation . . . 10
- 4.2 Example: Using a Single Robot . . . 10
- 4.3 Example: Using a Single Robot with the VFH driver . . . 11
- 4.4 Example: Using Multiple Robots . . . 12
- 4.5 Example: Using Multiple Robots with `--gazebo-prefix` . . . 14

5 Non-model Reference **16**
- 5.1 Global Paramamerters . . . 16

6 Model Reference **17**
6.1 `AvatarHeli` . . . 19
6.1.1 Overview . . . 19
6.1.2 `libgazebo` Interfaces . . . 19
6.1.3 Player Drivers . . . 19
6.1.4 World File Attributes . . . 19
6.1.5 Body Attributes . . . 19
6.2 `Blimp` . . . 20
6.2.1 Overview . . . 20
6.2.2 `libgazebo` Interfaces . . . 20
6.2.3 Player Drivers . . . 20
6.2.4 World File Attributes . . . 20
6.2.5 Body Attributes . . . 20
6.3 `ClodBuster` . . . 21
6.3.1 Overview . . . 21
6.3.2 `libgazebo` Interfaces . . . 21
6.3.3 Player Drivers . . . 21
6.3.4 World File Attributes . . . 21
6.3.5 Body Attributes . . . 21
6.4 `Factory` . . . 22
6.4.1 Overview . . . 22
6.4.2 `libgazebo` Interfaces . . . 22
6.4.3 Player Drivers . . . 22
6.4.4 World File Attributes . . . 22
6.4.5 Body Attributes . . . 22
6.5 `GarminGPS` . . . 23
6.5.1 Overview . . . 23
6.5.2 `libgazebo` Interfaces . . . 23
6.5.3 Player Drivers . . . 23
6.5.4 World File Attributes . . . 23
6.5.5 Body Attributes . . . 23
6.6 `GroundPlane` . . . 24
6.6.1 Overview . . . 24
6.6.2 `libgazebo` Interfaces . . . 24
6.6.3 Player Drivers . . . 24
6.6.4 World File Attributes . . . 24
6.6.5 Body Attributes . . . 24
6.7 `LightSource` . . . 25
6.7.1 Overview . . . 25
6.7.2 `libgazebo` Interfaces . . . 25
6.7.3 Player Drivers . . . 25
6.7.4 World File Attributes . . . 25
6.7.5 Body Attributes . . . 25
6.8 `MapExtruder` . . . 26
6.8.1 Overview . . . 26
6.8.2 `libgazebo` Interfaces . . . 26
6.8.3 Player Drivers . . . 26

6.8.4 World File Attributes . . . 26
6.8.5 Body Attributes . . . 26
6.9 ObserverCam . . . 27
6.9.1 Overview . . . 27
6.9.2 World File Attributes . . . 27
6.9.3 Windows . . . 27
6.9.4 Body Attributes . . . 28
6.10 Pioneer2AT . . . 29
6.10.1 Overview . . . 29
6.10.2 libgazebo Interfaces . . . 29
6.10.3 Player Drivers . . . 29
6.10.4 World File Attributes . . . 29
6.10.5 Body Attributes . . . 29
6.11 Pioneer2DX . . . 30
6.11.1 Overview . . . 30
6.11.2 libgazebo Interfaces . . . 30
6.11.3 Player Drivers . . . 30
6.11.4 World File Attributes . . . 30
6.11.5 Body Attributes . . . 30
6.12 Pioneer2Gripper . . . 31
6.12.1 Overview . . . 31
6.12.2 libgazebo Interfaces . . . 31
6.12.3 Player Drivers . . . 31
6.12.4 World File Attributes . . . 31
6.12.5 Body Attributes . . . 31
6.13 Pioneer2Sonars . . . 32
6.13.1 Overview . . . 32
6.13.2 libgazebo Interfaces . . . 32
6.13.3 Player Drivers . . . 32
6.13.4 World File Attributes . . . 32
6.13.5 Body Attributes . . . 32
6.14 PointSet . . . 33
6.14.1 Overview . . . 33
6.14.2 libgazebo Interfaces . . . 33
6.14.3 Player Drivers . . . 33
6.14.4 World File Attributes . . . 33
6.14.5 Body Attributes . . . 33
6.15 SegwayRMP . . . 34
6.15.1 Overview . . . 34
6.15.2 libgazebo Interfaces . . . 34
6.15.3 Player Drivers . . . 34
6.15.4 World File Attributes . . . 34
6.15.5 Body Attributes . . . 34
6.16 Shrimp . . . 35
6.16.1 Overview . . . 35
6.16.2 libgazebo Interfaces . . . 35
6.16.3 Player Drivers . . . 35

6.16.4 World File Attributes . . . 35
6.16.5 Body Attributes . . . 35
6.17 SickLMS200 . . . 36
6.17.1 Overview . . . 36
6.17.2 libgazebo Interfaces . . . 36
6.17.3 Player Drivers . . . 36
6.17.4 World File Attributes . . . 36
6.17.5 Body Attributes . . . 36
6.18 SimpleSolid . . . 37
6.18.1 Overview . . . 37
6.18.2 libgazebo Interfaces . . . 37
6.18.3 Player Drivers . . . 37
6.18.4 World File Attributes . . . 37
6.18.5 Body Attributes . . . 37
6.19 SonyVID30 . . . 38
6.19.1 Overview . . . 38
6.19.2 libgazebo Interfaces . . . 38
6.19.3 Player Drivers . . . 38
6.19.4 World File Attributes . . . 38
6.19.5 Body Attributes . . . 38
6.19.6 Windows . . . 39
6.20 Terrain . . . 40
6.20.1 Overview . . . 40
6.20.2 libgazebo Interfaces . . . 40
6.20.3 Player Drivers . . . 40
6.20.4 World File Attributes . . . 40
6.20.5 Body Attributes . . . 40
6.21 TotemPole . . . 41
6.21.1 Overview . . . 41
6.21.2 libgazebo Interfaces . . . 41
6.21.3 Player Drivers . . . 41
6.21.4 World File Attributes . . . 41
6.21.5 Body Attributes . . . 41
6.22 TruthWidget . . . 42
6.22.1 Overview . . . 42
6.22.2 libgazebo Interfaces . . . 42
6.22.3 Player Drivers . . . 42
6.22.4 World File Attributes . . . 42
6.22.5 Body Attributes . . . 42
6.23 WheelChair . . . 43
6.23.1 Overview . . . 43
6.23.2 libgazebo Interfaces . . . 43
6.23.3 Player Drivers . . . 43
6.23.4 World File Attributes . . . 43
6.23.5 Body Attributes . . . 43

II Developer Guide **44**

7 Gazebo Architecture **45**

8 Adding a New Model **46**
8.1 Model Source Files . . . 46
8.2 Registering the Model . . . 46
8.3 Working with GNU Autotools . . . 47

9 `libgazebo` **49**
9.1 Introduction . . . 49
9.2 Architecture . . . 49
9.3 Devices and Interfaces . . . 49
9.4 Using `libgazebo` . . . 50
9.5 Building Programs With `libgazebo` . . . 51

10 `libgazebo` Interface Reference **52**
10.1 `camera` . . . 53
10.2 `factory` . . . 55
10.3 `fiducial` . . . 57
10.4 `gps` . . . 59
10.5 `laser` . . . 61
10.6 `position` . . . 63
10.7 `power` . . . 65
10.8 `ptz` . . . 67
10.9 `truth` . . . 69

A Platform Specific Build Information **71**
A.1 Mac OS X . . . 71

B Coding Standards and Conventions **72**
B.1 Gazebo . . . 72
B.2 `libgazebo` . . . 73

Chapter 1

Introduction

1.1 What is Gazebo?

Gazebo is a multi-robot simulator for outdoor environments. Like Stage, it is capable of simulating a population of robots, sensors and objects, but does so in a three-dimensional world. It generates both realistic sensor feedback and physically plausible interactions between objects.

Gazebo is normally used in conjunction with the Player device server. Player provides an abstracted, network-centric mechanism (a server) through which robot controllers (clients) can interact with real robots and sensors. Gazebo works in conjunction with Player, providing simulated sensor data in the place of real sensor data. Ideally, client programs cannot tell the difference between real devices and the Gazebo simulation of those devices.

Gazebo can also be controlled through a low-level interface (`libgazebo`). This library included to allow third-party developers to easily integrate Gazebo into their own (non-Player) robot device servers or architectures.

Last but not least, Player is Open Source and Free Software, released under the GNU General Public License. If you don't like how something works, change it. And please send us your patch!

1.2 Stage and Gazebo

The Player/Stage project provides two multi-robot simulators: Stage and Gazebo. Since Stage and Gazebo are both Player-compatible, client programs written using one simulator can usually be run on the other with little or no modification. The key difference between these two simulators is that whereas Stage is designed to simulate a very large robot population with low fidelity, Gazebo is designed to simulated a small population with high fidelity. Thus, the two simulator are complimentary, and users may switch back and forth between them according to their needs.

1.3 Getting Gazebo

Gazebo is release in source form through the Player/Stage website:

`http://playerstage.sourceforge.net`

Check the downloads page for the latest software releases, and check the documentation page for the latest version of this manual.

1.4 System Requirements

Gazebo is primarily developed for x86/Linux systems using GCC and GNU autotools. It can, however, be ported fairly easily to Posix-like systems with X11 and OpenGL extensions (it is known to run more-or-less out-of-the-box on Apple's OS X, for example).

For best performance, users should also ensure that they are using hardware accelerated display drivers; try:

```
$ glxinfo
```

and check for "direct rendering: Yes". Please, please don't ask the Gazebo developers how to get hardware acceleration working for your particular graphics card; you should be able to figure this out by consulting various on-line sources.

1.5 Bugs

This software is provided WITHOUT WARRANTY. Nevertheless, if you find something that doesn't work, or there is some feature you would like to see, you can submit a bug report/feature request through the Player/Stage homepage:

```
http://playerstage.sourceforge.net
```

Include a detailed description of you problem and/or feature request, and information such as the Player version and operating system. Make sure you also select the "gazebo" category when reporting bugs.

1.6 License

1.7 Acknowledgments

Gazebo is written by Nate Koenig and Andrew Howard. This work is supported by DARPA grant DABT63-99-1-0015 (MARS). Thanks also to SourceForge.net for project hosting.

Part I

User Guide

Chapter 2

General Usage

2.1 Installing Third-Party Dependencies

Gazebo relies on a number of third-party libraries, most of which will probably be installed on your system by default. You may, however, have to install the following additional packages *before* installing Gazebo:

- libXML2: pretty much all distributions will have this package.
- OpenDynamicsEngine (ODE): most distributions have this package (including RedHat, Gentoo, Fink); if in doubt, you can build it yourself from the sources here:
  http://opende.sourceforge.net/ode.html
- Geospatial Data Abstraction Library (GDAL): this is less common, although there is a Fink package. Build it from the sources here:
  http://remotesensing.org/gdal/

2.2 Building and Installing Gazebo

The Gazebo source tarball can be obtained from
http://playerstage.sourceforge.net/
After unpacking the tarball, read the generic instructions in `README` and `INSTALL`. If you don't feel like reading those files, the following should suffice in most cases:

```
$ ./configure
$ make
$ make install
```

Gazebo will be installed in the default location: `/usr/local/`. The `configure` script accepts a number of options for customizing the build process, including changing the install location and adding/removing device drivers. For example, to change the install location for Gazebo to `~/local` in your home directory, use:

```
$ ./configure --prefix /home/<username>/local
```

Please read the FAQ entry on local installations, available here:
http://playerstage.sourceforge.net/faq.html

To see a complete list of build options, use:

```
$ ./configure --help
```

If you are going to use Gazebo with Player, note that Gazebo must be installed *before* Player.

2.3 Starting Gazebo

The Gazebo server can be started as follows:

```
$ gazebo <worldfile>
```

where `<worldfile>` is the file containing the description of the world and everything in it. Sample world files can be found in the `worlds` directory of the source distribution, or in the installed version under `/usr/local/share/gazebo/worlds/` (default install). For example:

```
$ gazebo /usr/local/share/gazebo/worlds/example1.world
```

will create a simple world with a single robot. Gazebo will display a window with a view of the simulated world; the camera viewpoint can be changed by dragging around with the mouse.

2.4 Working with Player

The Player device server treats Gazebo in exactly the same way that it treats real robot hardware: as a device that is a source of data and a sink for commands. Users must therefore run Player seperately, and point it at an running instance of Gazebo. Player has a number of specific drivers, such as `gz_position` and `gz_laser` that can be used to interact with Gazebo models.

For example, after starting Gazebo as per the above example, run Player like this:

```
$ player -g default /usr/local/share/player/config/gazebo.cfg
```

Player will output a message indicating that is has connected with the simulation:

```
libgazebo msg : opening /tmp/gazebo-<username>-default-sim-default
```

Users can now interact with the simulated robot exactly as the would a real robot. Try running `playerv`, for example:

```
$ playerv --position:0 --laser:0
```

This will pop up the standard Player viewer utility. You should see an outline of the robot and the laser scan. Use the mouse to pan and zoom. You can driver the robot around by selecting the ”command” option from the menu, and then dragging the little cross hairs to where you want the robot to go. You should see the the robot in the Gazebo window moving at the same time.

See Chapter 4 for examples of typical Player/Gazebo configurations, and consult the Player manual for information on specific Player drivers.

2.5 Command Line Options

Gazebo recognizes the following command line options.

Argument	Meaning
-v	Print the version string.

Chapter 3

The World File

The world file contains a description of the world to be simulated by Gazebo. It describes the layout of robots, sensors, light sources, user interface components, and so on. The world file can also be used to control some aspects of the simulation engine, such as the force of gravity or simulation time step.

Gazebo world files are written in XML, and can thus be created and modified using a text editor. Sample world files can be found in the `worlds` directory of the source distribution, or in the installed version (default install) under

`/usr/local/share/gazebo/worlds/`

3.1 Key Concepts and Basic Syntax

The world consists mainly of *model* declarations. A model can be a robot (e.g. a Pioneer2AT or SegwayRMP), a sensor (e.g. SICK LMS200), a static feature of the world (e.g. MapExtruder) or some manipulable object. For example, the following declaration will create a Pioneer2AT named "`robot1`":

```
<model:Pioneer2AT>
  <id>robot1</id>
  <xyz>0 0 0.40</xyz>
  <rpy>0 0 45</rpy>
</model:Pioneer2AT>
```

Associated with each model is a set of *attributes* such as the model's position `<xyz>` and orientation `<rpy>`; see Section 6 for a complete list of models and their attributes.

Models can also be composed. One can, for example, attach a scanning laser range-finder to a robot:

```
<model:Pioneer2AT>
  <id>robot1</id>
  <xyz>0 0 0.40</xyz>
  <rpy>0 0 45</rpy>
  <model:SickLMS200>
    <id>laser1</id>
    <parentBody>chassis</parentBody>
    <xyz>0.15 0 0.20</xyz>
    <rpy>0 0 0</rpy>
  </model:SickLMS200>
```

```
</model:Pioneer2AT>
```

The <parentBody> tag indicates which part of the robot the laser should be attached to (in this case, the chassis rather than the wheels). The <xyz> and <rpy> tags describe the laser's position and orientation with respect to this body (see Section 3.4 for a discussion of coordinate systems). Once attached, robot and the laser form a single rigid body.

3.2 Graphical User Interface

While Gazebo can operate without a GUI, it is often useful to have a viewport through which the user can inspect the world. In Gazebo, such user interface components are provided through special models (such as the ObserverCam) that can be declared and configured in the world file. See the documentation on the ObserverCam model for details.

3.3 Canonical World File Layout

The "standard" layout for the world file is can be seen from the examples included in the worlds directory. The basic components are as follows:

- XML meta-data, start of world block, more XML meta-data:

```
<?xml version="1.0"?>
<gz:world
  xmlns:gz='http://playerstage.sourceforge.net/gazebo/xmlschema/#gz'
  ...
```

- Global parameters:

```
<params:GlobalParams>
  <gravity>0.0 0.0 -9.8</gravity>
</params:GlobalParams>
```

- GUI components:

```
<model:ObserverCam>
  <id>userCam0</id>
  <xyz>0.504 -0.735 0.548</xyz>
  <rpy>-0 19 119</rpy>
  <window>
    <title>Observer</title>
    <size>640 480</size>
    <pos>0 0</pos>
  </window>
</model:ObserverCam>
```

- Light sources (without lights, the scene will be very dark):

```
<model:LightSource>
  <id>light0</id>
  <xyz>0.0 0.0 10.0</xyz>
</model:LightSource>
```

- Ground planes and/or terrains:

```
<model:GroundPlane>
  <id>ground1</id>
</model:GroundPlane>
```

- Robots, objects, etc.:

```
<model:Pioneer2AT>
  <id>robot1</id>
  <xyz>0 0.0 0.5</xyz>
  <model:SickLMS200>
    <id>laser1</id>
    <xyz>0.15 0 0.20</xyz>
  </model:SickLMS200>
</model:Pioneer2AT>
```

- End of world block:

```
</gz:world>
```

A detailed list of models and their attributes can be found in Chapters 5 and 6.

3.4 Coordinate Systems and Units

By convention, Gazebo uses a right-handed coordinate system, with x and y in the plane, and z increasing with altitude. Most models are designed such that the are upright (with respect to the z axis) and pointing along the positive x axis. The tag `<xyz>` is used to indicate an object's position (x, y and z coordinates); the tag `<rpy>` is used to indicate an objects orientation (Euler angles; i.e., roll, pitch and yaw). For example, `<xyz>1 2 3</xyz>` indicates a translation of 1 m along the x-axis, 2 m along the y-axis and 3 m along the z-axis; `<rpy>10 20 30</rpy>` indicates a rotation of 30 degrees about the z-axis (yaw), followed by a rotation pf 20 degrees about the y-axis (pitch) and a rotation of 10 degrees about the x-axis (roll).

Unless otherwise specified, the world file uses SI units (meters, seconds, kilograms, etc). The following idioms should also be noted:

- Angles and angular velocities are measured in degrees and degrees/sec, respectively.

Chapter 4

Working with Player

The Player device server treats Gazebo in exactly the same way that it treats real robot hardware: as a device that is a source of data and a sink for commands. A key advantage of this approach is that users may mix Player *abstract* drivers with simulation drivers. Thus, for example, drivers such as VFH (Vector Field Histogram) and AMCL (adapative Monte-Carlo localization), will work equally well with simulated and real robots. In the following sections, we describe some basic scenarios that demonstate this interaction.

4.1 Setting up the Simulation

The basic steps for setting up and running a combined Player/Gazebo simulation are as follows.

1. Write the Gazebo world file.
2. Start Gazebo.
3. Write the corresponding Player configuration file(s).
4. Start Player(s).
5. Start client program(s).

Note that Gazebo must be started *before* the Player server, and that the Player server must *re-started* whenever Gazebo is re-started.

4.2 Example: Using a Single Robot

Consider the case of a single robot with a scanning laser range-finder. The following Gazebo world file snippet will create a Pioneer2DX robot with SICK LMS200 laser.

```
<model:Pioneer2DX>
  <id>robot1_position</id>
  <xyz>0 0 0.40</xyz>
  <rpy>0 0 45</rpy>
  <model:SickLMS200>
    <id>robot1_laser</id>
    <xyz>0.15 0 0.20</xyz>
    <rpy>0 0 0</rpy>
```

```
    </model:SickLMS200>
  </model:Pioneer2DX>
```

The corresponding snippet of the Player configuration file should look like this:

```
  position:0
  (
    driver "gz_position"
    gz_id "robot1_position"
  )

  laser:1
  (
    driver "gz_laser"
    gz_id "robot1_laser"
  )
```

To run this simulation, start Gazebo with:

```
  $ gazebo <myworld>
```

where `<myworld>` is the name of the Gazebo world file, and start Player with

```
  $ player -g default <myconfig>
```

where `<myconfig>` is the name of the Player configuration file. Client programs can connect to the simulated devices devices on the default Player port 6665.

4.3 Example: Using a Single Robot with the VFH driver

Abstract devices can be mixed freely with simulated devices in the Player server. Thus, for example, it is possible to use the VFH (Vector Field Histogram) driver with a simulated robot. The following Gazebo world file snippet will create a Pioneer2DX robot with SICK LMS200 laser.

```
  <model:Pioneer2DX>
    <id>robot1_position</id>
    <xyz>0 0 0.40</xyz>
    <rpy>0 0 45</rpy>
    <model:SickLMS200>
      <id>robot1_laser</id>
      <xyz>0.15 0 0.20</xyz>
      <rpy>0 0 0</rpy>
    </model:SickLMS200>
  </model:Pioneer2DX>
```

The corresponding snippet of the Player configuration file should look like this:

```
  position:0
  (
    driver "gz_position"
    gz_id "robot1_position"
```

```
)

laser:0
(
  driver "gz_laser"
  gz_id "robot1_laser"
)

position:1
(
  driver "vfh"
  <vfh driver settings>
)
```

Note that the configuration file is exactly as per the first example; we have simply added another device to the Player server. The VFH driver will use the simulated robot chassis and laser exactly as it would a real robot chassis and laser.

To run this simulation, start Gazebo with:

```
$ gazebo <myworld>
```

where `<myworld>` is the name of the Gazebo world file, and start Player with

```
$ player -g default <myconfig>
```

where `<myconfig>` is the name of the Player configuration file. Client programs can connect to the server on the default Player port 6665.

4.4 Example: Using Multiple Robots

There are a number of ways to work with multiple robots. The simplest way is to use multiple instances of the Player server (one for each robot being simulated). The following Gazebo world file snippet will create a pair of Pioneer2DX robots with SICK LMS200 lasers.

```
<model:Pioneer2DX>
  <id>robot1_position</id>
  <xyz>0 0 0.40</xyz>
  <rpy>0 0 45</rpy>
  <model:SickLMS200>
    <id>robot1_laser</id>
    <xyz>0.15 0 0.20</xyz>
    <rpy>0 0 0</rpy>
  </model:SickLMS200>
</model:Pioneer2DX>

<model:Pioneer2DX>
  <id>robot2_position</id>
  <xyz>0 0 0.40</xyz>
  <rpy>0 0 45</rpy>
```

```
    <model:SickLMS200>
      <id>robot2_laser</id>
      <xyz>0.15 0 0.20</xyz>
      <rpy>0 0 0</rpy>
    </model:SickLMS200>
  </model:Pioneer2DX>
```

Since there will be two instances of the Player server, two different configuration files are required. For the first robot:

```
position:0
(
  driver "gz_position"
  gz_id "robot1_position"
)

laser:1
(
  driver "gz_laser"
  gz_id "robot1_laser"
)
```

and for the second robot:

```
position:0
(
  driver "gz_position"
  gz_id "robot2_position"
)

laser:1
(
  driver "gz_laser"
  gz_id "robot2_laser"
)
```

Note that these files are identical apart from the `gz_id` property. In general, however, the simulated robots may be heterogeneous, in which case the Player configuration files will differ substantially.

To run this simulation, start Gazebo with:

```
$ gazebo <myworld>
```

where `<myworld>` is the name of the Gazebo world file, and start two instances of Player with

```
$ player -p 7000 -g default <myconfig1>
```

and

```
$ player -p 7001 -g default <myconfig2>
```

where `<myconfig1>` and `<myconfig2>` are the two Player configuration files. Client programs can connect to the robots `robot1` and `robot2` through ports 7000 and 7001, respectively.

4.5 Example: Using Multiple Robots with `--gazebo-prefix`

When the simulated robots are homogeneous, one may simplify the process somewhat by employing the `--gazebo-prefix` flag with Player. The following Gazebo world file snippet will create a pair of Pioneer2DX robots with SICK LMS200 lasers.

```
<model:Pioneer2DX>
  <id>robot1_position</id>
  <xyz>0 0 0.40</xyz>
  <rpy>0 0 45</rpy>
  <model:SickLMS200>
    <id>robot1_laser</id>
    <xyz>0.15 0 0.20</xyz>
    <rpy>0 0 0</rpy>
  </model:SickLMS200>
</model:Pioneer2DX>

<model:Pioneer2DX>
  <id>robot2_position</id>
  <xyz>0 0 0.40</xyz>
  <rpy>0 0 45</rpy>
  <model:SickLMS200>
    <id>robot2_laser</id>
    <xyz>0.15 0 0.20</xyz>
    <rpy>0 0 0</rpy>
  </model:SickLMS200>
</model:Pioneer2DX>
```

Since the robots are identical, we can write one Player configuration file for both:

```
position:0
(
  driver "gz_position"
  gz_id "_position"
)

laser:1
(
  driver "gz_laser"
  gz_id "_laser"
)
```

Note that the `gz_id` values are incomplete; we will add *prefix* to these values when the Player server is started.

To run this simulation, start Gazebo with:

```
$ gazebo <myworld>
```

where `<myworld>` is the name of the Gazebo world file, and start two instances of Player with

```
$ player -p 7000 -g default --gazebo-prefix robot1 <myconfig>
```

and

```
$ player -p 7001 -g default --gazebo-prefix robot 2 <myconfig>
```

where `<myconfig>` is the common Player configuration file. Client programs can connect to the robots `robot1` and `robot2` through ports 7000 and 7001, respectively.

Chapter 5

Non-model Reference

5.1 Global Paramamerters

General parameters used by Gazebo.

<params:GlobalParams>

Attribute	Type	Default	Description
<gravity>	3-vector	(0,0,-9.8)	Gravity vector

Chapter 6

Model Reference

Gazebo currently supports the following models.

- `AvatarHeli`: USC AVATAR Helicopter.
- `Blimp`: A dirigible.
- `ClodBuster`: Car-like Non-Holonomic 4-wheeled Robot with Ackerman Steering.
- `Factory`: A tool to dynamically create models.
- `GarminGPS`: Basic GPS device model.
- `GroundPlane`: A flat plane that represents the ground.
- `LightSource`: A single point light.
- `MapExtruder`: Linearly extrudes a 2D image into 3D geometry.
- `ObserverCam`: A gods-eye camera (GUI component for user interaction).
- `Pioneer2AT`: ActivMedia Pioneer2AT (“all terrain”) robot with 4 wheels and skid steering.
- `Pioneer2DX`: ActivMedia Pioneer2DX (indoor) robot with 2 drive wheels and a castor.
- `Pioneer2Gripper`: ActivMedia Pioneer2 Gripper.
- `Pioneer2Sonars`: ActivMedia Pioneer2 sonar ring.
- `PointSet`: A model to visualize a point cloud.
- `SegwayRMP`: custom modified Segway RMP (“Robot Mobility Platform”) robot.
- `SickLMS200`: the ubiquitous SICK scanning laser range-finder.
- `Shrimp`: A robot designed after the Shrimp robot by BlueBotics.
- `SimpleSolid`: Simple solid shapes, such as boxes and spheres.
- `SonyVID30`: color video camera with controllable pan-tilt-zoom.
- `Terrain`: A terrain triangle mesh model.

- `TotemPole`: A vertical pole of simple solids, used as a visual fiducial.
- `TruthWidget`: A magical device for getting and seting model poses.
- `WheelChair`: A wheelchair robot.

6.1 AvatarHeli

Authors

Srikanth Saripalli srik(at)usc.edu

6.1.1 Overview

The AvatarHeli model simulates the USC AVATAR helicopter, a Bergen Industrial Twin RC helicopter (http://www.bergenrc.com/IndustrialTwin.asp).

6.1.2 libgazebo Interfaces

AvatarHeli currently does not support any interfaces.

6.1.3 Player Drivers

No player drivers are available.

6.1.4 World File Attributes

AvatarHeli models can be instantiated using the <model:AvatarHeli> tag. The following attributes are supported.

<model:AvatarHeli>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation

6.1.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.2 Blimp

Authors

Pranav Srivastava pranav(at)seas.upenn.edu

6.2.1 Overview

The Blimp model simulates a UPenn Blimp.

6.2.2 libgazebo Interfaces

Blimp supports the libgazebo position and position3d interface.

6.2.3 Player Drivers

Basic motor control and odometry information is available through the gz_position or gz_position3d driver.

6.2.4 World File Attributes

Blimp models can be instantiated using the <model:Blimp> tag. The following attributes are supported.

<model:Blimp>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<is3d>	true/false	false	Does this use the position3d interface?

6.2.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.3 ClodBuster

Authors

Pranav Srivastava pranav(at)seas.upenn.edu

6.3.1 Overview

The ClodBuster model simulates the UPenn ClodBuster robot (4 wheel drive robot with Ackerman steering).

6.3.2 libgazebo Interfaces

ClodBuster supports the libgazebo position interface.

6.3.3 Player Drivers

Basic motor control and odometry information is available through the gz_position driver.

6.3.4 World File Attributes

ClodBuster models can be instantiated using the <model:ClodBuster> tag. The following attributes are supported.

<model:ClodBuster>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<raw_encoder_position>	true/false	false	Flag indicating return of raw wheel encoder values.

6.3.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.4 Factory

Authors

Chris Jones cvjones(at)usc.edu, Nathan Koenig nkoenig(at)usc.edu

6.4.1 Overview

The Factory model maintains no physical characteristics, but instead allows the user to dynamically create models at runtime. A model is instantiated by passing an XML character string to a Factory model. The character string use the same model format as the .world files. Note that the string should *only* contain the model description, and no other tags. The XML string is passed into Gazebo through the gz_factory interface which has the same interface as the Player speech device.

6.4.2 libgazebo Interfaces

Factory supports the libgazebo factory interface.

6.4.3 Player Drivers

A new model is created using the gz_factory driver.

6.4.4 World File Attributes

Factory models can be instantiated using the <model:Factory> tag. The following attributes are supported.

<model:Factory>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string

6.4.5 Body Attributes

This model does not have a physical representation.

6.5 GarminGPS

Authors

Pranav Srivastava pranav(at)seas.upenn.edu

6.5.1 Overview

The GarminGPS model simulates GPS information by using the flat-world approximation. It doesn't account for GPS occlusion or loss of signal, etc.

6.5.2 libgazebo Interfaces

GarminGPS supports the libgazebo gps interfaces.

6.5.3 Player Drivers

The GPS position are available through the gz_gps driver.

6.5.4 World File Attributes

GarminGPS models can be instantiated using the <model:GarminGPS> tag. The following attributes are supported.

<model:GarminGPS>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<origin>	3-vector	(0,0,0)	coordinates of the origin in lat-long-alt

6.5.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.6 GroundPlane

Authors

Nathan Koenig nkoenig(at)usc.edu

6.6.1 Overview

The GroundPlane model simulates a flat infinite plane.

6.6.2 libgazebo Interfaces

GroundPlane has no libgazebo interface.

6.6.3 Player Drivers

There are no player drivers.

6.6.4 World File Attributes

GroundPlane models can be instantiated using the <model:GroundPlane> tag. The following attributes are supported.

<model:GroundPlane>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<normal>	3-vector	(0,0,1)	Direction of the normal vector
<height>	double	0	Distance along the z-axis to place the plane
<color>	3-vector	(0,0,0)	RGB color
<texture2D>	string	NULL	Texture image filename
<cfm>	double	0.01	Constraint Force Mixing parameter

6.6.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.7 LightSource

Authors

Andrew Howard ahoward(at)usc.edu, Nathan Koenig nkoenig(at)usc.edu

6.7.1 Overview

The LightSource model simulates a single point light.

6.7.2 libgazebo Interfaces

LightSource has no libgazebo interface.

6.7.3 Player Drivers

There are no player drivers.

6.7.4 World File Attributes

LightSource models can be instantiated using the <model:LightSource> tag. The following attributes are supported.

<model:LightSource>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<ambientColor>	4-vector	(0.2 0.2 0.2 1.0)	Ambient color
<diffuseColor>	4-vector	(0.8 0.8 0.8 1.0)	Diffuse color
<specularColor>	4-vector	(0.2 0.2 0.2 1.0)	Specular color
<attenuation>	3-vector	(1.0 0.0 0.0)	Constant, Linear, Quadratic attenuation

6.7.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.8 MapExtruder

Authors

Nathan Koenig nkoenig(at)usc.edu

6.8.1 Overview

The MapExtruder model extrudes a 2D image into 3D geometry. This is useful for generating simple maps. The bottom left point in the image is mapped to the position specified by the pos tag.

6.8.2 libgazebo Interfaces

MapExtruder currently has no libgazebo interface.

6.8.3 Player Drivers

There are no player drivers.

6.8.4 World File Attributes

MapExtruder models can be instantiated using the <model:MapExtruder> tag. The following attributes are supported.

<model:MapExtruder>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<imageFile>	string	NULL	PPM image file to extrude
<negative>	true/false	false	Use the inverse color values
<threshold>	integer	200	Value over which a pixel is considered filled space
<width>	float	0.1	Width of the geometries
<height>	float	1	Height of the geometries, along z-axis
<scale>	float	0.1	Scale the map up or down
<color>	3-vector	(0,0,0)	RGB color
<errBound>	float	5	Allowable error in the geometries
<halign>	left/center	left	Horizontal alignment
<valign>	bottom/center	bottom	Vertical alignment

6.8.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.9 ObserverCam

Authors

Nathan Koenig nkoenig(at)usc.edu, Andrew Howard ahoward(at)usc.edu

6.9.1 Overview

The ObserverCam model provides a interactive viewport through which users can view the simulation (think of it as a fly god's eye).

6.9.2 World File Attributes

ObserverCam models can be instantiated using the <model:ObserverCam> tag. The following attributes are supported.

<model:ObserverCam>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<updatePeriod>	float	0.1	Seconds between refresh
<displayRays>	true/false	false	Display laser rays
<saveFrames>	string	NULL	Directory to store image frames
<shadeModel>	flat/smooth	flat	OpenGL shading model
<polygonMode>	point/line/fill	fill	OpenGL polygon mode
<lens>			Camera lens attributes
<hfov>	degrees	60	Horizontal field of view
<nearClip>	meters	0.20	Distance to near clip plane
<farClip>	meters	1000.0	Distance to far clip plane

6.9.3 Windows

The following are definitions for the ObserverWindow. The window attribute block must be a child of <**model:ObserverCam**>.

<window:X11GLWindow>

Attribute	Type	Default	Description
<id>	string	NULL	Window ID string
<title>	string	NULL	Window title string
<size>	2-vector	(100,100)	Initial size of the window
<moveScale>	float	0.1	Mouse movement scale factor

Mouse Commands

Button	Vertical Move	Horizontal Move
Left	Rotate along X axis	Rotate along Z axis
Middle	Translate along Z axis	Translate along X axis
Right	Translate along Y axis	Translate along X axis

6.9.4 Body Attributes

This model does not have a physical representation.

6.10 Pioneer2AT

Authors

Andrew Howard ahoward(at)usc.edu, Nathan Koenig nkoenig(at)usc.edu

6.10.1 Overview

The Pioneer2AT model simulates the ActivMedia Pioneer2AT ("all terrain") robot with 4 wheels and skid steering. Currently, it does not simulate the robot's sonar ring.

6.10.2 libgazebo Interfaces

Pioneer2AT supports the libgazebo position interface.

6.10.3 Player Drivers

Basic motor control and odometry information is available through the gz_position driver.

6.10.4 World File Attributes

Pioneer2AT models can be instantiated using the <model:Pioneer2AT> tag. The following attributes are supported.

<model:Pioneer2AT>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation

6.10.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.11 Pioneer2DX

Authors

Andrew Howard ahoward(at)usc.edu

6.11.1 Overview

The Pioneer2DX model simulates the ActivMedia Pioneer2DX (indoor) robot with 2 drive wheels and a castor. Currently, it does not simulate the robot's sonar ring.

6.11.2 libgazebo Interfaces

Pioneer2DX supports the libgazebo position interface.

6.11.3 Player Drivers

Basic motor control and odometry information is available through the gz_position driver.

6.11.4 World File Attributes

Pioneer2DX models can be instantiated using the <model:Pioneer2DX> tag. The following attributes are supported.

<model:Pioneer2DX>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation

6.11.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.12 Pioneer2Gripper

Authors

Carle Cote Carle.Cote(at)USherbrooke.ca

6.12.1 Overview

The Pioneer2Gripper model simulates the ActivMedia Pioneer2 Gripper. Every object in the world that can be picked up by a gripper must set the <**canBeGrip**> to true.

6.12.2 libgazebo Interfaces

Pioneer2Gripper supports the libgazebo gripper interface.

6.12.3 Player Drivers

Control of the gripper position is available through the gz_gripper driver.

6.12.4 World File Attributes

Pioneer2Gripper models can be instantiated using the <model:Pioneer2Gripper> tag. The following attributes are supported.

<**model:Pioneer2Gripper**>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation

6.12.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.13 Pioneer2Sonars

Authors

Carle Cote Carle.Cote(at)USherbrooke.ca

6.13.1 Overview

The Pioneer2Sonars model simulates the ActivMedia Pioneer2 Sonar Ring. The sonar ring consists of 15 ray proximity sensors, each of which simulates a single sonar.

6.13.2 libgazebo Interfaces

Pioneer2Sonars supports the libgazebo sonar interface.

6.13.3 Player Drivers

Control of the sonars is available through the gz_sonars driver.

6.13.4 World File Attributes

Pioneer2Sonars models can be instantiated using the <model:Pioneer2Sonars> tag. The following attributes are supported.

<model:Pioneer2Sonars>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	(0,0,0)	Position of the ray
<dir>	3-vector	(0,0,0)	Direction of the ray
<range>	meters	0	Initial sensor range

6.13.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.14 PointSet

Authors

Andrew Howard ahoward(at)usc.edu

6.14.1 Overview

The PointSet model visualizes a, usually dense, point cloud.

6.14.2 libgazebo Interfaces

PointSet has no libgazebo interface.

6.14.3 Player Drivers

There are no player drivers.

6.14.4 World File Attributes

PointSet models can be instantiated using the <model:PointSet> tag. The following attributes are supported.

<model:PointSet>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<pointFile>	string	NULL	Point cloud data file

6.14.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.15 SegwayRMP

Authors

Andrew Howard ahoward(at)usc.edu

6.15.1 Overview

The SegwayRMP model simulates RMP modification of the Segway HT.

Those of you that already have one of these will know what this is; for everyone else: the Segway RMP (Robotic Mobility Platform) is a two-wheeled, dynamically stabilized robot made from a modified Segway HT (Human Transporter). This is an interesting beast: it is large, fast and heavy, and will fall down alarmingly if you turn the power off.

The SegwayRMP model includes a PID inverted pendulum controller written by Marin Kobilarov mkobilar(at)usc.edu. The dynamics are roughly similar to those of the real robot, but users should not expect a close correlation.

6.15.2 libgazebo Interfaces

SegwayRMP supports the libgazebo position interface. IMU data is not yet available.

6.15.3 Player Drivers

Basic motor control and odometry information is available through the gz_position driver. IMU data is not yet available.

6.15.4 World File Attributes

SegwayRMP models can be instantiated using the <model:SegwayRMP> tag. The following attributes are supported.

<model:SegwayRMP>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation

6.15.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.
topPlate	Top plate

6.16 Shrimp

Authors

Stijn Opheide stijn.opheide(at)kotnet.org, Jef Marien jef.marien(at)student.kuleuven.ac.be, Koen Jans koen.jans(at)student.kuleuven.ac.be

6.16.1 Overview

The Shrimp model simulates the BlueBotics Shrimp robot. This model is currently under development.

6.16.2 libgazebo Interfaces

Shrimp supports the libgazebo position interface.

6.16.3 Player Drivers

Basic motor control and odometry information is available through the gz_position driver.

6.16.4 World File Attributes

Shrimp models can be instantiated using the <model:Shrimp> tag. The following attributes are supported.

<model:Shrimp>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation

6.16.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.17 SickLMS200

Authors

Andrew Howard ahoward(at)usc.edu, Nathan Koenig nkoenig(at)usc.edu

6.17.1 Overview

The SickLMS200 model simulates the ubiquitous SICK scanning laser range-finder.

6.17.2 libgazebo Interfaces

SickLMS200 supports the libgazebo laser and fiducial interfaces.

6.17.3 Player Drivers

Range and intensity data is available through the gz_laser driver. Fiducial information (ID, range, bearing and orientation) is available through the gz_fiducial driver. Note that, at present, only SimpleSolid models can be fiducials.

6.17.4 World File Attributes

SickLMS200 models can be instantiated using the <model:SickLMS200> tag. The following attributes are supported.

<model:SickLMS200>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<rangeCount>	int	361	The number of range readings to generate.
<rayCount>	int	91	The number of actual rays to generate.
<maxRange>	float	8.192	Maximum laser range (m).
<minRange>	float	0.20	Minimum laser range (m).
<scanPeriod>	float	0.200	The interval between successive scans (s).

Note that when the number of rays is less than the number of range readings, the "missing" range readings will be interpolated. Reducing the number of rays is a good way to save CPU cycles (at the expense of simulation fidelity).

6.17.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.18 SimpleSolid

Authors

Andrew Howard ahoward(at)usc.edu

6.18.1 Overview

The SimpleSolid model creates simple solid objects, such as spheres and boxes. The solids have mass, and will interact with other objects.

6.18.2 libgazebo Interfaces

SimpleSolid does not support any interfaces.

6.18.3 Player Drivers

There are no Player drivers for SimpleSolid.

6.18.4 World File Attributes

SimpleSolid models can be instantiated using the <model:SimpleSolid> tag. The following attributes are supported.

<model:SimpleSolid>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<shape>	string	NULL	Shape description. Can be any of: sphere, box, cylinder.
<size>	float	1.0	Sphere diameter.
<size>	tuple	(0.1, 1.0)	Cylinder diameter and height.
<size>	tuple	(1.0, 1.0, 1.0)	Box width, length and height.
<mass>	float	1.0	Mass of the object (kg).
<color>	tuple	(0, 0, 0, 1)	Color of the object (including alpha, if transparency is enabled.
<retro>	float	0	Retro-reflectivity level (0 = not a retro-reflector, 1 = perfect retro-re
<fiducial>	integer	-1	Fiducial ID (-1 = not a fiducial).
<transparent>	true/false	false	Sets transparency.
<texture2D>	string	NULL	Texture image filename
<cfm>	double	0.01	Constraint Force Mixing parameter

6.18.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.19 SonyVID30

Authors

Nathan Koenig nkoenig(at)usc.edu, Andrew Howard ahoward(at)usc.edu, Pranav Srivastava pranav(at)seas.upenn.edu

6.19.1 Overview

The SonyVID30 model simulates the Sony VID 30 pan-tilt-zoom camera. This model will probably be merged with the MonoCam model in future releases.

6.19.2 libgazebo Interfaces

SonyVID30 supports the libgazebo ptz interface.

6.19.3 Player Drivers

The camera position (pan and tilt) can be controlled by setting pan/tilt using the gz_ptz driver.

6.19.4 World File Attributes

SonyVID30 models can be instantiated using the <model:SonyVID30> tag. The following attributes are supported.

<model:SonyVID30>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<gain>	double	1.0	Motion gain
<zoomLimits>	tuple	(10.0, 60.0)	Zoom limits
<imageSize>	tuple	(320, 240)	Size of the image in pixels
<updatePeriod>	float	0.1	Seconds between refresh
<displayRays>	true/false	false	Display laser rays
<saveFrames>	string	NULL	Directory to store image frames
<shadeModel>	flat/smooth	flat	OpenGL shading model
<polygonMode>	point/line/fill	fill	OpenGL polygon mode
<lens>			Camera lens attributes
<hfov>	degrees	60	Horizontal field of view
<nearClip>	meters	0.20	Distance to near clip plane
<farClip>	meters	1000.0	Distance to far clip plane

6.19.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.
head	The camera head (the bit that moves.)

6.19.6 Windows

The following are definitions for the windows. The window attribute block must be a child of <**model:SonyVID30**>.

<**window:X11GLWindow**>

Attribute	Type	Default	Description
<id>	string	NULL	Window ID string
<title>	string	NULL	Window title string

6.20 Terrain

Authors

Nathan Koenig nkoenig(at)usc.edu

6.20.1 Overview

The Terrain model generates a triangle mesh from a 2D image or other data file, such as a DEM. The generated mesh has a variable triangle density corresponding to a user defined error bound. A high error bound allows for more error in the terrain representation, resulting in fewer triangles.

6.20.2 libgazebo Interfaces

Terrain has no libgazebo interface.

6.20.3 Player Drivers

There are no player drivers.

6.20.4 World File Attributes

Terrain models can be instantiated using the <model:Terrain> tag. The following attributes are supported.

<model:Terrain>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<vertexSpacing>	double	1.0	Meters between the vertices of the mesh.
<elevationScale>	double	1.0	Elevation scale factor in meters.
<color>	3-vector	(0,0,0)	RGB color
<texture2D>	string	NULL	Texture image filename
<textureRepeat>	tuple	(1.0 1.0)	Texture repetition in X and Y directions.
<cfm>	double	0.01	Constraint Force Mixing parameter

6.20.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.21 TotemPole

Authors

Andrew Howard ahoward(at)usc.edu

6.21.1 Overview

The TotemPole model generates a stack of simple solids to create a visual fiducial.

6.21.2 libgazebo Interfaces

TotemPole has no libgazebo interface.

6.21.3 Player Drivers

There are no player drivers.

6.21.4 World File Attributes

TotemPole models can be instantiated using the <model:TotemPole> tag. The following attributes are supported.

<model:TotemPole>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation
<shape>	box/cylinder	cylinder	Simple Sold shape.
<diameter>	double	0.10	Diameter of the totem pole in meters.
<steps>	integer	1	Number of shapes used in the pole.
<step_##_height>	double	0.02	Height of the ##th shape.
<step_##_barcode>	binary string	NULL	Barcode of ##th shape.

6.21.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

6.22 TruthWidget

Authors

Chris Jones cvjones(at)usc.edu

6.22.1 Overview

The TruthWidget is a magical device for querying and modifying the true pose of objects in the simulator. It has no simulated body, but can be attached to other models to learn or change their pose.

6.22.2 libgazebo Interfaces

TruthWidget supports the libgazebo truth interface.

6.22.3 Player Drivers

As of Player 1.4rc2, this model has no Player support.

6.22.4 World File Attributes

TruthWidget models can be instantiated using the <model:TruthWidget> tag. The following attributes are supported.

<model:TruthWidget>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation

6.22.5 Body Attributes

This model does not have a physical representation.

6.23 WheelChair

Authors

Pranav Srivastava pranav(at)seas.upenn.edu

6.23.1 Overview

The WheelChair model simulates a robotic wheelchair.

6.23.2 libgazebo Interfaces

WheelChair supports the libgazebo position interface.

6.23.3 Player Drivers

Basic motor control and odometry information is available through the gz_position driver.

6.23.4 World File Attributes

WheelChair models can be instantiated using the <model:WheelChair> tag. The following attributes are supported.

<model:WheelChair>

Attribute	Type	Default	Description
<id>	string	NULL	Model ID string
<xyz>	3-vector	$(0, 0, 0)$	Model position
<rpy>	Euler angles	$(0, 0, 0)$	Model orientation

6.23.5 Body Attributes

The following bodies are used by this model.

Name	Description
canonical	The canonical (default) body.

Part II

Developer Guide

Chapter 7

Gazebo Architecture

TODO

Chapter 8

Adding a New Model

This chapter describes the basic steps for creating a new model. It describes how the source files should be laid out, how to register models with the server and how to work with GNU Autotools.

> **N.B. These instructions assume you are working from CVS, not a source snap-shot or distribution.**

Developers should consult Chapter 7 for detailed information on model class implementation. Developers are also advised to read Appendix B for Gazebo coding standards and conventions.

8.1 Model Source Files

Source code for models is located under `server/models/`, with a separate directory for each model. Model directory names should match model class names, i.e., the `SickLMS200` model is found the `server/models/Sic` directory. Layout of files within the model directory is at the developers discretion; by convention, however, models are comprised of a single header file containing the model class declarations, and one or more source files containing the class definitions. Thus, the `SickLMS200` model is comprised of `SickLMS200.hh` and `SickLMS200.cc`.

The recommended way to create a new model is to copy an existing model with similar functionality, and perform some judicious search-and-replacing. In addition to changing the class name (e.g. from `SickLMS200` to `MyModel`), developers must also change the model's naked `New` function. E.g., `NewSickLMS200()` becomes `NewMyModel()`. This function is used to create instances of the model class at run-time.

8.2 Registering the Model

Models must be registered with the server. Registration is handled by the `ModelFactory` class, which can be found in `server/ModelFactory.cc`. Registration is a two step process:

1. Add a declaration for the new models creation function; e.g.:

```
#if INCLUDE_MYMODEL
extern Model* NewMyModel( World *world );
#endif
```

The `INCLUDE` macro will be defined automagically by Autotools (see below) if and only if the model should be included in the build.

2. In the `ModelFactory::NewModel()` function, add a clause for the new model, e.g.:

```
#ifdef INCLUDE_MYMODEL
  if (strcmp(classname, "MyModel") == 0)
    return NewMyModel(world);
#endif
```

This line allows the server to look up the model by name and construct an appropriate class instance at run-time.

8.3 Working with GNU Autotools

Gazebo uses GNU Autotools to managing the build process; while Autotools can be daunting for newcomers, the rewards are well worth the effort. When using Autotools, there are two key notions to bear in mind:

- Project-level configuration is controlled by `configure.in` found in the project top-level directory.
- Module-level configuration is controlled by `Makefile.am` found in every sub-directory.

These configuration files are used to *generate* the `Makefile`'s that will ultimately control the build process (developers should never manipulate `Makefile`'s directly).

The basic process for adding a new model to the Autotools setup is as follows.

1. Create `Makefile.am` for the new model:
   - Copy `Makefile.am` from another model into the new model directory.
   - Modify all entries describing the model name. For example, if you copied `Makefile.am` from the `SickLMS200` model, replace all references to `sicklms200` with `mymodelname`.
   - Modify the `SOURCES` line to list the new model's source files. Be sure to include header files in this list (so they will included in the final distribution).
2. Modify `Makefile.am` in the `server/models` directory.
   - Add the new model directory to the `SUBDIRS` line.
3. Modify `configure.in` in the top-level directory.
   - In the "model tests" section, add a `GAZEBO_ADD_MODEL` entry for the new model. The arguments are: model name (lower case), model path, whether or not the model should be included in the default build, a list of libraries to check for and a list of headers to check for. The model will not be built unless all of the library and header dependencies are satisfied.
   - In the "create makefiles" section, add the path of the `Makefile` that needs to be created.
4. Re-generate the `Makefile`'s:
   - From the top-level directory, run:

```
$ ./bootstrap
```

with whatever arguments you would normally pass to `configure`.

Running `make` will now build the new driver.

Chapter 9

libgazebo

9.1 Introduction

External programs can use `libgazebo` to interact with the Gazebo simulator. `libgazebo` is a simple C library that allows other programs to peek and poke into a running simulation; through this library, programs may read sensor data from and send commands to simulated devices. The `player` device server, for example, uses `libgazebo` in this way.

Normal users will interact with Gazebo via `player` and need not be aware of this library. This chapter is included primarily to aid those who either wish to add a new type of interface to the simulator, or who wish to write their own custom control programs (by-passing `player` entirely).

9.2 Architecture

`libgazebo` uses interprocess communication (IPC) to allow a program to exchange data with a running simulator. Hereafter, we shall refer to the simulator as the *server* and the program as the *client*. Thus, for example, when using `player` to interact with Gazebo, Gazebo is the server and `player` is the client.

The details of the underlying IPC are entirely hidden by `libgazebo` , which should be treated as a black box for passing data back and forth to the server. Currently, the only limitation users need be aware of is that both server and client must reside on the same machine; they cannot be run in separate locations across a network.

9.3 Devices and Interfaces

`libgazebo` makes the familiar distinction between *devices* and *interfaces*. In Gazebo, a *device* is a fully parameterized model, representing a particular real-world object such as a Pioneer2AT robot or a SICK LMS200 laser. An interface, on the other hand, is a general specification for an entire class of devices, such as *position* or *laser*. Thus, a Pioneer2AT device will present a *position* interface, while the SICK LMS200 will present a *laser* interface.

The complete set of interfaces supported by `libgazebo` is described in Chapter 10. Note, however, that the set of interfaces defined by `libgazebo` should not be confused with those defined by `player` . Currently, these two sets of interfaces are fairly similar, but there is no guarantee that they will remain so in future.

9.4 Using `libgazebo`

Client programs must connect both to a specific server and a specific set of devices within that server. The following code snippet illustrates the general behavior:

```
// main.c - works with example1.world
// compile with:
//    g++ -Wall -c -o main.o main.c
//    g++ main.o main -L. -lgazebo
// Thanks to Kevin Knoedler for various fixes.

#include <stdio.h>
#include <gazebo.h>

int main (inst argc, char *argv[])
{
  // Create a client object
  client = gz_client_alloc();

  // Connect to the server
  gz_client_connect(server_id);

  // Create a position interface
  position = gz_position_alloc();

  // Connect to device on server
  gz_position_open(position, client, device_id);

  // Lock it
  gz_position_lock(position, 1);

  // Print the current odometric position
  printf("%0.3f %0.3f\n",
         position->data->odo_pose[1], position->data->odo_pose[1]);

  // Unlock it
  gz_position_unlock(position);

  // Close the interface and free object
  gz_position_close(position);
  gz_position_free(position);

  // Close the connection and free the client object
  gz_server_disconnect(client);
  gz_server_free(client);

  return 0;
}
```

Error checking has been removed from this example for the sake of clarity. There are several points to note:

- The value of `server_id` variable must correspond to the server ID specified in the Gazebo world file. By default, the ID is `default`, but other values may be specified if the user is running multiple instances of Gazebo simultaneously.
- The `device_id` variable specifies a unique device ID; the value of this variable must correspond to the device ID specified in the Gazebo world file.
- Any attempt to connect to a device using an unsupported interface will result in the `open` function returning an error. For example, if the device specified by `device_id` is actually a laser, attempting to open it using `gz_position_open()` will result in an error.
- Do not use the `create` and `destroy` functions associated with each interface; these are reserved for use by Gazebo only. Use `open` and `close` instead.

9.5 Building Programs With `libgazebo`

All public functions are declared in `gazebo.h` and defined in `libgazebo.a`. The default Gazebo install script will put these in sensible places, so your compiler should pick them up automatically.

Chapter 10

`libgazebo` Interface Reference

`libgazebo` currently supports the following interfaces:

- `camera` : supports cameras such as the SonyVID30.
- `factory` : supports adding models at runtime via a Factory model.
- `fiducial` : supports simple fiducial finders, such as those based on retro-reflective barcodes.
- `gps` : supports (D)GPS receivers.
- `laser` : supports scanning laser range finders, such as the SICK LMS200.
- `position` : supports basic mobile robot platforms, such as the Pioneer2AT.
- `power` : reports battery levels.
- `ptz` : supports pan-tilt-zoom camera heads, such as the SonyVID30.
- `truth` : reports true pose of objects in the simulator.

A detailed reference for each of these interfaces is included in the following sections.

10.1 camera

The camera interface allows clients to read images from a simulated camera. This interface gives the view of the world as the camera onboard a robotic platform would see it. This can be fed to image processing algorithms such as the CMVision blobfinder to recover blobs. Useful for testing visual-servoing type stuff.

```
// Camera interface
typedef struct _gz_camera_data_t
{
  // Common data structure
  gz_data_t head;

  // Data timestamp
  double time;

  // Image dimensions
  unsigned int width, height;

  // Image depth (bits: 8, 16, 24 or 32)
  int depth;

  // Image pixel data
  unsigned int image_size;
  unsigned char image[GAZEBO_MAX_IMAGE_SIZE];

} gz_camera_data_t;

// The camera interface
typedef struct _gz_camera_t
{
  // General interface info
  gz_iface_t *iface;

  // Pointer to interface data area
  gz_camera_data_t *data;

} gz_camera_t;

// Create an interface
extern gz_camera_t *gz_camera_alloc();

// Destroy an interface
extern void gz_camera_free(gz_camera_t *self);

// Create the interface (used by Gazebo server)
extern int gz_camera_create(gz_camera_t *self, gz_server_t *server, const char *id);

// Destroy the interface (server)
extern int gz_camera_destroy(gz_camera_t *self);

// Open an existing interface (used by Gazebo clients)
extern int gz_camera_open(gz_camera_t *self, gz_client_t *client, const char *id);
```

```
// Close the interface (client)
extern int gz_camera_close(gz_camera_t *self);

// Lock the interface.  Set blocking to 1 if the caller should block
// until the lock is acquired.  Returns 0 if the lock is acquired.
extern int gz_camera_lock(gz_camera_t *self, int blocking);

// Unlock the interface
extern void gz_camera_unlock(gz_camera_t *self);
```

10.2 factory

The `factory` interface allows clients to send XML strings to a factory in order to dynamically create models.

```
// Factory interface
typedef struct _gz_factory_data_t
{
  // Common data structures
  gz_data_t head;

  // Data timestamp
  double time;

  // String describing the model to be instantiated
  uint8_t string[4096];

} gz_factory_data_t;

// The position interface
typedef struct _gz_factory_t
{
  // General interface info
  gz_iface_t *iface;

  // Pointer to interface data area
  gz_factory_data_t *data;

} gz_factory_t;

// Create an interface
extern gz_factory_t *gz_factory_alloc();

// Destroy an interface
extern void gz_factory_free(gz_factory_t *self);

// Create the interface (used by Gazebo server)
extern int gz_factory_create(gz_factory_t *self, gz_server_t *server, const char *id);

// Destroy the interface (server)
extern int gz_factory_destroy(gz_factory_t *self);

// Open an existing interface (used by Gazebo clients)
extern int gz_factory_open(gz_factory_t *self, gz_client_t *client, const char *id);

// Close the interface (client)
extern int gz_factory_close(gz_factory_t *self);

// Lock the interface.  Set blocking to 1 if the caller should block
// until the lock is acquired.  Returns 0 if the lock is acquired.
extern int gz_factory_lock(gz_factory_t *self, int blocking);
```

```
// Unlock the interface
extern void gz_factory_unlock(gz_factory_t *self);
```

10.3 fiducial

The `fiducial` interface allows clients to determine the identity, range, bearing and orientation (relative to some sensor) of objects in the world. For example, this interface is used by the SickLMS200 model to return data from simulated retro-reflective barcodes.

```
// Data for a single fiducial
typedef struct _gz_fiducial_fid_t
{
  // Fiducial id
  int id;

  // Fiducial range, bearing and orientation
  double pose[3];

} gz_fiducial_fid_t;

// Fiducial data
typedef struct _gz_fiducial_data_t
{
  // Common data structures
  gz_data_t head;

  // Data timestamp
  double time;

  // Observed fiducials
  int fid_count;
  gz_fiducial_fid_t fids[GZ_FIDUCIAL_MAX_FIDS];

} gz_fiducial_data_t;

// The fiducial interface
typedef struct _gz_fiducial_t
{
  // General interface info
  gz_iface_t *iface;

  // Pointer to interface data area
  gz_fiducial_data_t *data;

} gz_fiducial_t;

// Create an interface
extern gz_fiducial_t *gz_fiducial_alloc();

// Destroy an interface
extern void gz_fiducial_free(gz_fiducial_t *self);

// Create the interface (used by Gazebo server)
extern int gz_fiducial_create(gz_fiducial_t *self, gz_server_t *server, const char *id);
```

```
// Destroy the interface (server)
extern int gz_fiducial_destroy(gz_fiducial_t *self);

// Open an existing interface (used by Gazebo clients)
extern int gz_fiducial_open(gz_fiducial_t *self, gz_client_t *client, const char *id);

// Close the interface (client)
extern int gz_fiducial_close(gz_fiducial_t *self);

// Lock the interface.  Set blocking to 1 if the caller should block
// until the lock is acquired.  Returns 0 if the lock is acquired.
extern int gz_fiducial_lock(gz_fiducial_t *self, int blocking);

// Unlock the interface
extern void gz_fiducial_unlock(gz_fiducial_t *self);
```

10.4 gps

The gps interface allows the user to receive GPS (Latitude/Longitude/Altitude) information for the robot platform on which the GPS device is mounted.

```
// GPS interface
typedef struct _gz_gps_data_t
{
  // Common data structure
  gz_data_t head;

  // Data timestamp
  double time;

  // Latitude and longitude, in degrees.
  double latitude;
  double longitude;

  // Altitude, in meters.
  double altitude;

  // UTM coordinates (meters)
  double utm_e, utm_n;

  // Number of satellites in view.
  int satellites;

  // Fix quality
  int quality;

  // Horizontal dilution of position (HDOP)
  double hdop;

  // Errors
  double err_horz, err_vert;

  // ?
  double origin_longitude, origin_latitude, origin_altitude;

} gz_gps_data_t;

// The GPS interface
typedef struct _gz_gps_t
{
  // General interface info
  gz_iface_t *iface;

  // Pointer to interface data area
  gz_gps_data_t *data;

} gz_gps_t;
```

```
// Create an interface
extern gz_gps_t *gz_gps_alloc();

// Destroy an interface
extern void gz_gps_free(gz_gps_t *self);

// Create the interface (used by Gazebo server)
extern int gz_gps_create(gz_gps_t *self, gz_server_t *server, const char *id);

// Destroy the interface (server)
extern int gz_gps_destroy(gz_gps_t *self);

// Open an existing interface (used by Gazebo clients)
extern int gz_gps_open(gz_gps_t *self, gz_client_t *client, const char *id);

// Close the interface (client)
extern int gz_gps_close(gz_gps_t *self);

// Lock the interface.  Set blocking to 1 if the caller should block
// until the lock is acquired.  Returns 0 if the lock is acquired.
extern int gz_gps_lock(gz_gps_t *self, int blocking);

// Unlock the interface
extern void gz_gps_unlock(gz_gps_t *self);
```

10.5 laser

The laser interface allows clients to read data from a simulated laser range finder (such as the SICK LMS200). Some configuration of this device is also allowed.

```
// Laser data
typedef struct _gz_laser_data_t
{
  // Common data structures
  gz_data_t head;

  // Data timestamp
  double time;

  // Range scan angles
  double min_angle, max_angle;

  // Angular resolution
  double res_angle;

  // Max range value
  double max_range;

  // Range readings
  int range_count;
  double ranges[GZ_LASER_MAX_RANGES];
  int intensity[GZ_LASER_MAX_RANGES];

} gz_laser_data_t;

// The laser interface
typedef struct _gz_laser_t
{
  // General interface info
  gz_iface_t *iface;

  // Pointer to interface data area
  gz_laser_data_t *data;

} gz_laser_t;

// Create an interface
extern gz_laser_t *gz_laser_alloc();

// Destroy an interface
extern void gz_laser_free(gz_laser_t *self);

// Create the interface (used by Gazebo server)
extern int gz_laser_create(gz_laser_t *self, gz_server_t *server, const char *id);

// Destroy the interface (server)
extern int gz_laser_destroy(gz_laser_t *self);
```

```
// Open an existing interface (used by Gazebo clients)
extern int gz_laser_open(gz_laser_t *self, gz_client_t *client, const char *id);

// Close the interface (client)
extern int gz_laser_close(gz_laser_t *self);

// Lock the interface.  Set blocking to 1 if the caller should block
// until the lock is acquired.  Returns 0 if the lock is acquired.
extern int gz_laser_lock(gz_laser_t *self, int blocking);

// Unlock the interface
extern void gz_laser_unlock(gz_laser_t *self);
```

10.6 position

The position interface allows clients to send commands to and read odometric data from simulated mobile robot bases, such as the Pioneer2AT or ATRV Jr.

```
// Position interface
typedef struct _gz_position_data_t
{
  // Common data structures
  gz_data_t head;

  // Data timestamp
  double time;

  // Pose (usually global cs); rotation is specified by euler angles
  double pos[3];
  double rot[3];

  // Velocity; rotation is specified by euler angles
  double vel_pos[3];
  double vel_rot[3];

  // Motor stall flag
  int stall;

  // Enable the motors
  int cmd_enable_motors;

  // Commanded robot velocities (robot cs); rotation is specified by euler angles
  double cmd_vel_pos[3];
  double cmd_vel_rot[3];

} gz_position_data_t;

// The position interface
typedef struct _gz_position_t
{
  // General interface info
  gz_iface_t *iface;

  // Pointer to interface data area
  gz_position_data_t *data;

} gz_position_t;

// Create an interface
extern gz_position_t *gz_position_alloc();

// Destroy an interface
extern void gz_position_free(gz_position_t *self);

// Create the interface (used by Gazebo server)
```

```
extern int gz_position_create(gz_position_t *self, gz_server_t *server, const char *id);

// Destroy the interface (server)
extern int gz_position_destroy(gz_position_t *self);

// Open an existing interface (used by Gazebo clients)
extern int gz_position_open(gz_position_t *self, gz_client_t *client, const char *id);

// Close the interface (client)
extern int gz_position_close(gz_position_t *self);

// Lock the interface.  Set blocking to 1 if the caller should block
// until the lock is acquired.  Returns 0 if the lock is acquired.
extern int gz_position_lock(gz_position_t *self, int blocking);

// Unlock the interface
extern void gz_position_unlock(gz_position_t *self);
```

10.7 power

The power interface allows clients to read battery levels from simulated robots.

```
// Power interface
typedef struct _gz_power_data_t
{
  // Common data structures
  gz_data_t head;

  // Data timestamp
  double time;

  // Battery levels (volts)
  double levels[10];

} gz_power_data_t;

// The power interface
typedef struct _gz_power_t
{
  // General interface info
  gz_iface_t *iface;

  // Pointer to interface data area
  gz_power_data_t *data;

} gz_power_t;

// Create an interface
extern gz_power_t *gz_power_alloc();

// Destroy an interface
extern void gz_power_free(gz_power_t *self);

// Create the interface (used by Gazebo server)
extern int gz_power_create(gz_power_t *self, gz_server_t *server, const char *id);

// Destroy the interface (server)
extern int gz_power_destroy(gz_power_t *self);

// Open an existing interface (used by Gazebo clients)
extern int gz_power_open(gz_power_t *self, gz_client_t *client, const char *id);

// Close the interface (client)
extern int gz_power_close(gz_power_t *self);

// Lock the interface.  Set blocking to 1 if the caller should block
// until the lock is acquired.  Returns 0 if the lock is acquired.
extern int gz_power_lock(gz_power_t *self, int blocking);

// Unlock the interface
extern void gz_power_unlock(gz_power_t *self);
```

10.8 ptz

The ptz interface allows clients to control the pan, tilt and zoom angles on a camera head such as the Sony VID30.

```
// Ptz interface
typedef struct _gz_ptz_data_t
{
  // Common data structure
  gz_data_t head;

  // Data timestamp
  double time;

  // Measured orientation (radians)
  double pan, tilt;

  // Measured field of view (radians)
  double zoom;

  // Commanded orientation (radians)
  double cmd_pan, cmd_tilt;

  // Commanded field of view (radians)
  double cmd_zoom;

} gz_ptz_data_t;

// The ptz interface
typedef struct _gz_ptz_t
{
  // Common data structure
  gz_data_t head;

  // General interface info
  gz_iface_t *iface;

  // Pointer to interface data area
  gz_ptz_data_t *data;

} gz_ptz_t;

// Create an interface
extern gz_ptz_t *gz_ptz_alloc();

// Destroy an interface
extern void gz_ptz_free(gz_ptz_t *self);

// Create the interface (used by Gazebo server)
extern int gz_ptz_create(gz_ptz_t *self, gz_server_t *server, const char *id);

// Destroy the interface (server)
```

```
extern int gz_ptz_destroy(gz_ptz_t *self);

// Open an existing interface (used by Gazebo clients)
extern int gz_ptz_open(gz_ptz_t *self, gz_client_t *client, const char *id);

// Close the interface (client)
extern int gz_ptz_close(gz_ptz_t *self);

// Lock the interface.  Set blocking to 1 if the caller should block
// until the lock is acquired.  Returns 0 if the lock is acquired.
extern int gz_ptz_lock(gz_ptz_t *self, int blocking);

// Unlock the interface
extern void gz_ptz_unlock(gz_ptz_t *self);
```

10.9 truth

The truth interface is useful for getting and setting the ground-truth pose of objects in the world; currently, it is supported only by the TruthWidget model.

```
// Truth data
typedef struct _gz_truth_data_t
{
  // Common data structures
  gz_data_t head;

  // Data timestamp
  double time;

  // True object position (x, y, x)
  double pos[3];

  // True object rotation (roll, pitch, yaw)
  double rot[3];

  // New command (0 or 1)?
  int cmd_new;

  // Commanded object position
  double cmd_pos[3];

  // Commanded object rotation
  double cmd_rot[3];

} gz_truth_data_t;

// The truth interface
typedef struct _gz_truth_t
{
  // General interface info
  gz_iface_t *iface;

  // Pointer to interface data area
  gz_truth_data_t *data;

} gz_truth_t;

// Create an interface
extern gz_truth_t *gz_truth_alloc();

// Destroy an interface
extern void gz_truth_free(gz_truth_t *self);

// Create the interface (used by Gazebo server)
extern int gz_truth_create(gz_truth_t *self, gz_server_t *server, const char *id);

// Destroy the interface (server)
```

```
extern int gz_truth_destroy(gz_truth_t *self);

// Open an existing interface (used by Gazebo clients)
extern int gz_truth_open(gz_truth_t *self, gz_client_t *client, const char *id);

// Close the interface (client)
extern int gz_truth_close(gz_truth_t *self);

// Lock the interface.  Set blocking to 1 if the caller should block
// until the lock is acquired.  Returns 0 if the lock is acquired.
extern int gz_truth_lock(gz_truth_t *self, int blocking);

// Unlock the interface
extern void gz_truth_unlock(gz_truth_t *self);
```

Appendix A

Platform Specific Build Information

A.1 Mac OS X

Gazebo has been successfully built on Mac OS X 10.2, using the following steps.

1. Install Apple's X11 and X11 developer packages; these can be obtained from Apple's website with a little bit of scrounging.
2. Install Fink; this can be obtained from http://fink.sourceforge.net/
3. Use Fink to install other packages, such as `libxml2`, `gdal` and `gdal-dev`.
4. Install ODE as per normal, then run `ranlib` on the installed library:

   ```
   $ ranlib [path_to_ode_lib]/libode.a
   ```

5. Set the compiler include path. In bash shell:

   ```
   $ export CPATH=/usr/X11R6/include:/sw/include:/sw/include/gdal1
   $ export CPPFLAGS=-no-cpp-precomp
   ```

   Note the extra include paths (Fink sometimes installs things in weird places).

6. Configure and build Gazebo as per normal.

Appendix B

Coding Standards and Conventions

This appendix sets out the basic coding standards and conventions used in Gazebo and `libgazebo`.

B.1 Gazebo

Gazebo is written in a C++.

File Naming Conventions

- Source files are camel-capped; e.g., `Pioneer2AT.cc`.
- C++ header files are suffixed with `.hh`, C++ source files are suffixed with `.cc`; e.g., `Model.hh`, `Model.cc`.
- Directory names are lower case, with the exception of model directories, which are camel-capped.

Coding Conventions

- Class and function names are camel-capped with a leading upper-case character; e.g. `BoxGeom`, `SetGravity()`.
- Variable names are camel-capped with a leading lower-case character, and do not include any type encoding (no Hungarian); e.g. `myVariable`.
- Class variables and methods are always accessed explicitly: e.g. `this->MyMethod()` and `this->myVariable`.
- Standard indentation is two spaces (no tab characters).
- Matching braces are aligned, e.g.:

```
if (condition)
{
  do_something();
  do_something_else();
}
```

Other C++ "features"

Use of the following C++ "features" is *strongly* discouraged:

- Templates and the STL.
- Multiple inheritance.

B.2 `libgazebo`

For maximum compatibility and portability, `libgazebo` is written in C.

File Naming Conventions

- Source files are lower-case with underscores; e.g., `gz_laser.c`.
- Header files are suffixed with `.h`, C source files are suffixed with `.c`; e.g., `gazebo.h, libgazebo.c`.
- Source files for interfaces are prefixed by `gz_`; e.g., `gz_position.c`.

Coding Conventions

- Public functions (i.e., those included in `gazebo.h` are prefixed by `gz_`; e.g., `gz_laser_alloc()`.
- Function names are lower-case with underscores; e.g., `my_function()`.
- Variable names are lower-case with underscores, and do not include any type encoding (i.e., no Hungarian); e.g., `my_variable`.
- Standard indentation is two spaces (no tab characters).
- Matching braces are aligned, e.g.:

```
if (condition)
{
  do_something();
  do_something_else();
}
```