Autonomy Laboratory
Simon Frazer University
Vancouver, British Columbia, Canada

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

Robotics Laboratory
Stanford University
Stanford, California, USA

Player/Stage project

Stage

Version 1.3.3 User Manual
(compatible with Player 1.4rc2)

Richard T. Vaughan Andrew Howard Brian P. Gerkey

This document may not contain the most current documentation on Stage. For the latest documentation, consult the Player/Stage homepage:
`http://playerstage.sourceforge.net`

December 7, 2003

Contents

Chapter 1

Introduction

1.1 What is Stage?

Stage is a product of the Player/Stage Project, a collection of software tools to support research in autonomous robotics and intelligent sensor systems. Stage simulates a population of mobile robots, sensors and objects in a two-dimensional bitmapped environment. Stage is designed with multi-agent systems in mind, so it provides fairly simple, computationally cheap models of lots of devices rather than attempting to emulate any device with great fidelity. We have found this to be a useful approach.

Stage devices are usually controlled through *Player*; a networked robot server. Player provides a convenient interface to a set of device drivers for real robots and sensors. Stage simulates a population of devices and makes them available thorugh Player. Users write robot controllers and sensor algorithms as 'clients' to the Player 'server'. Typically, clients cannot tell the difference between the real robot devices and their simulated Stage equivalents (unless they try very hard). We have found that clients developed using Stage will work with little or no modification with the real robots and vice versa. Thus Stage allows rapid prototyping of controllers destined for real robots. Stage also allows experiments with realistic robot devices you don't happen to have.

Various sensors and actuators are provided, including sonar, scanning laser rangefinder, color-blob vision, odometry, grippers, bumpers/whiskers and mobile robot bases.

1.2 How to get Stage and related software

The main resource for Player/Stage is the project homepage:

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

Access to source code releases, access to the CVS development tree, bug tracking, user mailing lists, etc. is available from the Sourceforge project management page:

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

The current release is available as the source tarball Stage-<version>.src.tgz at:

http://sourceforge.net/project/showfiles.php?group_id=42445

Previous versions were significantly different; this manual does not apply to them.

1.3 What's in the Package?

In the release tarball you will find:

- C++ source code for 'stage' the simulation engine.
- Example environments and setup files.

1.4 Requirements

- Player [`http://playerstage.sourceforge.net`]
- libstdc++

Optional (but highly recommended):

- libRTK [`http://playerstage.sourceforge.net`]
  - X11R6, GTK++.

Stage was developed and tested under Linux kernel 2.4, glibc-2.2. Code is reasonable ANSII/POSIX so it should compile elsewhere. No promises, but people have found it to work on a variety of set-ups.

1.5 Ownership

Stage is released under the GNU General Public License. Stage programs, images, examples, source code and documentation are copyrighted by their their authors. The authors are:

Richard Vaughan [rtv@sourceforge.net]

Andrew Howard [inspectorg@sourceforge.net]

Brian Gerkey [gerkey@sourceforge.net]

Kasper Stoy [kaspers@robotics.usc.edu]

Boyoon Jung [boyoon@robotics.usc.edu]

Jakob Fredslund [jakobf@robotics.usc.edu]

See your name here by contributing devices, superior algorithms, bugfixes, examples, etc.

1.6 Bugs and feedback

This is constantly evolving research software. It is bound to contain bugs, despite developer and user testing. If you find something that appears to be a bug, or if some aspect of Stage's behavior seems wrong or non-intuitive, let us know. If you have a problem, please check the website and bug tracking logs. If you can't find an answer there, use the bug tracker to tell us about the problem. Better still, fix it and send us the patch. To stay in touch with the developers and other users, join the mailing lists.

When submitting bugs, include as much information as possible, including the Stage version, OS type and version, and any output messages. A detailed description of what happened will enable us (hopefully) to repeat and analyze the problem. Of course, there is NO WARRANTY on this software, and no guarantee that we will fix your problem. But we use Stage for our research and we want it to work properly, so we will do our best. Again *p*lease use the sourceforge bug tracking tools; that's the best way to see your problem solved.

1.7 What happened to...?

Previous releases contained some supplementary programs which have been removed during development of Stage-1.3.3. 'RTKStage' has been folded into Stage. Stage client mode makes 'XS' redundant. Stage server makes 'manager' redundant. Don't worry; things are better now.

1.8 Citations

If you find Stage and Player useful in your work, we would greatly appreciate your mentioning that fact in papers that you publish. We have presented papers on Player in peer-reviewed conferences; the following papers are the definitive references when citing Stage and Player: [3, 2, 1]:

- Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 2121–2427, Las Vegas, Nevada, October 2003.

- Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proc. of the Intl. Conf. on Advanced Robotics (ICAR)*, pages 317–323, Coimbra, Portugal, July 2003.

- Brian P. Gerkey, Richard T. Vaughan, Kasper Støy, Andrew Howard, Maja J Matarić and Gaurav S Sukhatme. Most Valuable Player: A Robot Device Server for Distributed Control. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1226–1231, Wailea, Hawaii, October 2001.

If you have space (and are feeling generous), you can also insert a footnote similar to the following:

> Stage and Player are freely available under the GNU General Public License from http://playerstage.sourceforge.net.

By including such acknowledgements, you do more than feed our egos and further our careers. You spread the word about the Player/Stage project, which will bring more users and developers, as well as please our funders, ensuring that we will continue hacking on the software.

1.9 Acknowledgements

Stage originated at the University of Southern California Robotics Labs. Support at USC has come from DARPA grant DABT63-99-1-0015 (MARS), NSF grant ANI-9979457 (SCOWR), DARPA contract DAAE07-98-C-L028 (TMR), ONR Grants N00014-00-1-0140 and N0014-99-1-0162, and JPL Contract No. 1216961. Development at HRL Laboratories is supported by a DARPA contract (SDR). Thanks to Doug Gage at DARP IPTO.

Thanks to our contributors and users, particularly the USC Robotics Lab students and alumni who have been generous with their advice, bug fixes, and contributions. These fine people have contributed code: Esben Østergård, Jakob Fredslund, Boyoon Jung, Jason K. Douglas, Kim Jinsuck, Gabe Sibley, and Dave Naffin. Contributed tools can be found on the website.

Chapter 2

Running Stage

2.1 Building and Installing Stage

You must install Player *before* building Stage. To build Stage with an interactive GUI, which is highly recommended, you must install libRTK before building Stage. Player and libRTK are availabe from the Player/Stage files page:

```
http://sourceforge.net/project/showfiles.php?group_id=42445
```

Build and install Player and libRTK following the instructions in the packages' README and INSTALL files. Now download thew Stage tarball and unpack it with:

```
$ tar xzvf Stage-1.2.tgz
```

Now follow the instructions for your release in the top-level README and INSTALL files.

2.2 Running Stage

Before running stage, first make sure the Player executable can be found in your path (Stage will automatically start an instance of Player, so it needs to know where to find the Player executable). For example, try:

```
$ which player
```

and make sure it returns a valid Player executable path, such as

```
~/player-1.3/bin/player
```

If not, set your PATH variable to include the Player binary directory. For example, in BASH do:

```
$ export PATH=$PATH:$HOME/player-1.2/bin
```

The general command line to run Stage is:

```
stage [options] <filename.world>
```

By convention, Stage configuration files end with a '.world' extension and are referred to as 'world files'. A world file specifies what stage must simulate. The user can override some of the world file settings at run time using the command-line options described below. If all is well, Stage will start up, load the world file, and spawn an Player. Each step causes a message on standard output, so a sample invocation and start up would look like this:

```
$ stage worlds/everything.world

 ** Stage  v1.3 **
[World worlds/everything.world]
[Server localhost:6601]
** Player v1.3 ** [Stage /tmp/stageIO.username.0]
```

At this point you should be able to interact with objects in the world with the GUI (try dragging things around) and access sensors and actuators through Player. Try using the Player client program (<player_root>/utils/playerv/playerv) to see the output from some sensors.

2.3 The World File

The world file is a description of the world that Stage must simulate. It describes robots, sensors, actuators, moveable and immovable objects. The world file can also be used to control many aspects of the simulation engine, such as its speed and fidelity. See Chapter 4 for a complete description of the world file format. Sample world files can also be found in the 'worlds' directory.

2.4 Command Line Arguments

Stage takes the following command-line options. Where an option can also be set in the configuration file, the command line option takes precedence.

`-n`
: No Player - do not spawn a Player. You can run Player manually in Stage mode with `player -stage <device dir>`. Useful for debugging Player or if you want to use an alternative interface to Stage devices.

`-g`
: Disables the Graphical User Interface.

`-o`
: Output mode - enables console output showing timing and data throughput information

`-t <timeout in seconds>`
: Timeout - Stage will quit after simulating the specified amount of time. Useful for batch runs.

`-u <update period in seconds>`
: Stage will attempt to take this much real time (wall-clock time) to perform each update cycle. It does this by computing the cycle, then sleeping (or polling for input) for any remaining time. If the cycle's computation takes longer than the requested cycle time, Stage will run slower than requested. Default is 0.1 seconds.

`-v <simulation time step in seconds>`
: Stage will simulate the passing of this much time per update cycle. Default is 0.1 seconds. By changing the ratio of real (-u) and simulated (-v) time, you can make Stage run faster than, slower than, or approximately at real-time.

`-f`
: Fast mode - Stage will run as fast as possible; not attempting to match real time. Useful for batch runs. This is slightly more efficient than setting the desired update time to zero seconds (-u 0.0).

`-s`
: Stopped - Stage will start up with the clock stopped. You can start and stop the clock by sending Stage a SIGUSR1 signal (`killall -s USR1 stage` should do the trick; this command is provided as the script <stage_root>/tools/pause.).

2.5 Controlling the robots

The virtual robots in Stage are controlled through the Player. Demo controllers in various languages (currently C++, C, TCL & LISP) are included in the Player distribution.

Try using the Player example client `<player_root>/utils/playerv/playerv` to check that you can control Stage robots and read from their sensors. playerv is a very useful tool for testing and debugging your controller code.

Client libraries in other languages including Java and Python are also available. Check the website for the latest resources.

2.6 Starting and Stopping the clock

Stage's internal clock can be started and stopped by sending Stage a SIGUSR1, for example with the command:

```
killall -s USR1 stage
```

. The script `<stage_root>/tools/pause` executes this command.

Chapter 3

Using the Stage GUI

Stage presents a single conventional resizable window with a menu and a main display area showing a view of the world.

3.1 World view

The main display area shows the world, the simulated entities (objects and devices), and optionally, representations of the data generated by devices.

3.1.1 Mouse

The user can pan and zoom the world view and manipulate entities with the mouse:

Clicks on the background

Mouse action	Result
Left-click and drag	pan the window
Right-click and drag TOWARDS the center of the window	zoom in
Right-click and drag AWAY FROM the center of the window	zoom out

Clicks on entities

Mouse action	Result
Left-click and drag	move the entity
Right-click and drag	rotate the entity

3.1.2 Keyboard

The world view can also be panned and zoomed with the keyboard. The keybindings are:

Clicks on entities

Key	Action
<arrowkeys>	scroll the window
<ctrl><arrowkeys>	scroll the window in large increments
<shift><uparrow>	zoom in
<shift><downarrow>	zoom out

3.2 Menu

3.2.1 File Menu

Save	save current world state into worldfile
Export	export a picture of world state as an xfig file (needs work)
Quit	exit Stage

3.2.2 View Menu

Grid	toggle view of a 1m grid
Matrix	toggle view of underlying bitmap representation
Data menu	toggle visualizations of data generated by devices
Object menu	toggle visualizations of object bodies

3.2.3 Action Menu

Subscribe to all	toggle Player subscriptions to all devices, enabling data visualization for just about evetrything.

Chapter 4

The World File

The world file is used to describe the particular set of robots, sensors and objects to be simulated by Stage. Stage reads the world file on start-up and creates entities as indicated in the file. Stage may also write updated pose information into the world file when the user selects the **File:Save** menu option.

Note that the world file format has changed significantly from previous versions. The script `tools/worldfileconv.` will attempt to convert your old (pre Stage-1.2) world files to the new format.

The worldfile is given as the last argument when invoking Stage: `$ stage <worldfile>`. Stage searches for the specified file in these directories, in this order:

1. the current directory
2. the directory that contains the current world file
3. the directories listed in the STAGEPATH environment variable

STAGEPATH works like the PATH variable of most shells. It is a list of directories to be searched, separated by colons. An example of how to set STAGEPATH in the BASH shell:

```
$ export STAGEPATH=$HOME/robotdev/worlds:$HOME/build/stage-1.3/worlds
$ echo $STAGEPATH
/home/vaughan/robotdev/worlds:/home/vaughan/build/stage-1.3/worlds
```

With the

4.1 Basic Syntax

A simple world file might look like this:

```
# This world file creates two robots with lasers.

environment
(
  file "cave.pnm"
  scale 0.03
)
```

```
position
(
  name "robot1" port 6665 pose [1 1 0]
  player ()
  laser ()
)

position
(
  name "robot2" port 6666 pose [2 1 0]
  player ()
  laser ()
)
```

This example shows the basic syntactic features of the world file format: comments, entities and properties. Comments are indicated by the # symbol; they may be placed anywhere in the file and continue to the end of the line. For example:

```
# This world file creates two robots with lasers.
```

Entities are indicated using `type ( ... )` entries; each such entry instantiates an entity of type `type`. For example:

```
position ( ... )
```

creates a single position device (a bare-bones mobile robot). Entities may be nested to indicate that one entity is a "child" of another; thus:

```
position ( player () laser() )
```

creates a single position device with a Player server and laser attached to it. Think of child entities as physically sitting on their parent. Entities have properties, indicated using `name value` pairs:

```
position ( name "robot1" port 6665 pose [1 1 0] ... )
```

This entry creates a position device named "robot1" attached to port 6665, with initial position $(1, 1)$ and orientation of 0. Property values can be either numbers (`6665`), strings (indicated by double quotes `"robot1"`) or tuples (indicated by brackets `[1 1 0]`).

4.2 Defining new entity types

The `define` statement can be used to define new types of entities. For example, the world file from the previous section can be re-written in a more concise form as follows:

```
# This world file creates two robots with lasers.
# It uses the 'define' construct to define a new type of entity.

environment ( file "cave.pnm" scale 0.03 )

define myrobot position ( player() laser() )

myrobot ( name "robot1" port 6665 pose [1 1 0] )
myrobot ( name "robot2" port 6666 pose [2 1 0] )
```

New entities are defined using `define newentity oldentity (...)`. For example, the line:

```
define myrobot position ( player() laser() )
```

defines a new `myrobot` entity type composed of the primitive `position`, `player` and `laser` entities. This entity may be instantiated using the standard syntax:

```
myrobot ( name "robot1" port 6665 pose [1 1 0] )
```

This entry creates a position device named `robot1` that has both `player` and `laser` devices attached.

4.3 Using include files

The `include` statement can be used to include entity definitions into a world file. For example, the world file from the previous section can be divided into an include file called `myrobots.inc`:

```
# This is an include file.
# It uses the 'define' construct to define a new type of entity.

define myrobot position ( player() laser() )
```

and a world file called `myworld.world`:

```
# This world file creates two robots with lasers.
# It uses the 'include' statement to include the robot definitions.

include "myrobots.inc"

environment ( file "cave.pnm" scale 0.03 )

myrobot ( name "robot1" port 6665 pose [1 1 0] )
myrobot ( name "robot2" port 6666 pose [2 1 0] )
```

The definitions are included using the `include "<filename>"` statement. Stage searches for <filename> in these directories, in this order:

1. the current directory
2. the directory that contains the current world file
3. the directories listed in the STAGEPATH environment variable

4.4 Units

The default units for length and angles are meters and degrees respectively. Units may be changed using the following global properties:
The following example uses millimeters rather than meters for the unit length unit:

Name	Values	Description
unit_length	"m" "cm" "mm"	Set the unit length to meters, centimeters or millimeters.
unit_angle	"degrees" "radians"	Set the unit angle to degrees or radians.

```
# This world file creates two robots with lasers.
# It uses the 'include' statement to include the robot definitions.

unit_length "mm"

include "myrobots.inc"

environment ( file "cave.pnm" scale 30 )

myrobot ( name "robot1" port 6665 pose [1000 1000 0] )
myrobot ( name "robot2" port 6666 pose [2000 1000 0] )
```

Be warned that the length specfication applies to the include files as well, so choose a unit length early and stick to it.

4.5 Examples

See the `examples` directory in the Stage distribution for more world file examples.

Chapter 5

Entity Reference

This chapter describes all of the models supported by Stage. All entity types defined in the world file are ultimately composed of one or more models. Each model is implemented by code in <stage_src>/src/models.

5.1 Properties and the type heirarchy

Each model has zero or more *properties* associated with it; these properties specify characteristics such as an object's shape or a sensor's range. Models are organized into a heriarchy, with sub-types inheriting properties from their parent type. All passive environmental objects (boxes, pucks and bitmaps) are derived from a basic `entity` type and are referred to as *objects. Similary, all sensors and actuator models are derived from a generic* `device` *type, and referred to as* devices. The `device` type is itself derived from `entity`, so they devices inherit all the standrd entity properties.

Note that both the `entity` and `device` types are *abstract* (in C++ parlance) since they cannot be instantiated directly.

5.2 Object Summary

The following table lists all of the *objects* (models of passive environmental objects) supported by Stage. They are described in detail in the following pages. See Section 5.3 for a list of Stage's *devices*.

Type	Parent type	Description
`entity`	None	A generic entity, which has shape, exent and color - cannot be instantiated directly.
`bitmap`	`entity`	A pattern of fixed obstacles loaded from a PNM bitmap file. Used to import building floorplans, walls, boulders, etc.
`box`	`entity`	A fixed obstacle (can be moved by the user but not the robot).
`puck`	`entity`	A movable object (can be moved by both the user and the robot).

5.2.1 The `entity` base type

Parent

`<none>`

Description

All Stage models are derived from the generic `entity` type. While this type can not be instantiated directly, all descendent types inherit its properties.

Properties

Name	Values	Description
`name`	`string`	A descriptive name for this entity; used by the GUI.
`pose`	`[x y a]`	Initialse pose (position and orientation).
`shape`	`"rect"` `"circle"`	Entity shape; entities can be either rectangular or circular.
`size`	`[sizex sizey]`	Entity dimensions.
`mass`	`float`	mass for puck collision model; defaults to be immoveably massive.
`color`	`string`	Descriptive color (e.g. `"red"` or `"blue"`); only colors listed in the X11 color database should be used (look for `rgb.txt` in your X installation).
`fiducial_id`	`int`	The id returned by a `fiducialfinder` scanning this object. In the range 0-255, where 0 means the object will not be detected as a fiducial.
`obstacle_return`	`"visible"` `"invisible"`	Specifies whether or not this entity will be treated as a fixed obstacle for the purposes of collision detection. Derived types will set this to a sensible default.
`sonar_return`	`"visible"` `"invisible"`	Specifies whether or not this entity will be detected by sonar sensors. Derived types will set this to a sensible default.
`vision_return`	`"visible"` `"invisible"`	Specifies whether or not this entity will be seen by cameras; the color is specified by the `color` property. Derived types will set `vision_return` to a sensible default.
`laser_return`	`"bright"` `"visible"` `"invisible"`	Specifies whether or not this entity will be seen be laser range finders; the `"bright"` value indicates that the entity is a retro-reflector (and hence produces a very intense return in the laser).
`idar_return`	`"IDARTransparent"` `"IDARReflect"` `"IDARReceive"`	Specifies the behavior when hit by an IDAR beam.
`interval`	`seconds`	Specifies the interval between model updates in seconds. Smaller intervals can increase simulation fidelity at cost of CPU time.

Defaults

Name	Value
name	""
shape	"none"
size	[0 0]
pose	[0 0 0]
color	"black"
obstacle_return	"invisible"
sonar_return	"invisible"
vision_return	"invisible"
laser_return	"invisible"
gripper_return	"invisible"
idar_return	"IDARTransparent"
id	-1
mass	1000.0
interval	0.1

5.2.2 The `box` object

Parent

`entity`

Description

The most simple object type, the `box` is used to create a rectangular or circular obstacle.

Properties

Name	Values	Description
<none>		

Defaults

Name	Value
`shape`	`"rect"`
`size`	`[1.0 1.0]`
`color`	`"yellow"`
`obstacle_return`	`"visible"`
`sonar_return`	`"visible"`
`vision_return`	`"visible"`
`laser_return`	`"visible"`
`idar_return`	`"IDARReflect"`

5.2.3 The `bitmap` object

Parent

`entity`

Description

The `bitmap` type is used to load a pattern of fixed obstacles (walls, rocks, trees, furniture, etc) defined in a PNM bitmap file. Since v1.3 a world file can specify as many bitmaps as desired, and bitmap respects the `pose`, `color` and most other properties.

Properties

Name	Values	Description
`file`	`string`	The name of the image file describing the fixed obstacles; non-black image pixels will be treated as obstacles, black image pixels will be treated as empty space. Only pnm and gzipped pnm images are supported.
`scale`	`float`	The image scale, in meters/pixel (or units-lengths/pixel if some another unit of length is specified).

Defaults

Name	Value
`color`	`"black"`
`obstacle_return`	`"visible"`
`sonar_return`	`"visible"`
`vision_return`	`"visible"`
`laser_return`	`"visible"`
`idar_return`	`"IDARReflect"`
`file`	`""`
`scale`	`0`

Notes

Both the `file` and `scale` properties *must* be specified.

5.2.4 The `puck` object

Parent

`entity`

Description

The most simple object type, the `box` type is used to create a rectangular or circular obstacle.

Properties

Name	Values	Description
`friction`	`float`	Coefficient of friction for the puck sliding model.

Defaults

Name	Value
`shape`	`"circle"`
`size`	`[0.08 0.08]`
`color`	`"green"`
`obstacle_return`	`"visible"`
`vision_return`	`"visible"`
`gripper_return`	`"visible"`
`puck_return`	`"visible"`
`friction`	`0.05`
`mass`	`0.2`
`interval`	`0.01`

5.3 Device Summary

This table lists all of the *devices* (Player-controllable sensors and actuators) supported by Stage. See the Player User Manual for details of the physical devices and the interfaces used to access them. The Stage devices are described in detail on the following pages.

Type	Parent type	Description
`device`	`entity`	A generic device type (has a port number, device index, etc).
`broadcast`	`device`	Allows clients to communicate with one another through Stage/Player; messages sent to the broadcast device will be received by all other broadcast devices.
`gps`	`device`	GPS receiver; *disabled*
`gripper`	`device`	1-DOF gripper (open/close).
`laser`	`device`	Scanning laser range finder.
`fiducialfinder`	`device`	Find fiducials (formerly Laser beacon detector).
`position`	`device`	A bare-bones differential-drive mobile robot with odometry.
`omniposition`	`device`	An omnidirectional mobile robot with odometry.
`ptz`	`device`	Pan-tilt-zoom camera.
`sonar`	`device`	Sonar range-finder array.
`truth`	`device`	Thee truth device can be used to get and set the pose of entities in the simulator; setting the pose of a truth device will 'teleport' the device's parent to a new location.
`blobfinder`	`device`	Vision-based color blob detector (formerly visiondevice.
`idar`	`device`	Infrared Data and Ranging device, based on the HRL Labs Pherobot main sensor.
`idarturret`	`device`	Array of IDARs accessed together for efficiency.

5.3.1 The generic `device` type

Parent

`<none>`

Interface

`<none>`

Description

This is a base type for all device models. The properties are specify how the device is addressed through the Player interface.

Properties

Name	Values	Description
`port`	`int`	The port to which this device is attached; if not specified, the parent device's port will be used.
`index`	`int`	The index for this device; used to distinguish between multiple instances of the same type of device (on a single robot). Will generally be set to 0.

Defaults

Name	Value
`port`	`<6665 or inherited from parent>`
`index`	`0`

Notes

Typically the `port` is specified for the top-level device in each robot (usually a `position` device). Descendant devices inherit the port number. If you have more than one device of a given type on a robot, be sure to give them unique `index` numbers.

5.3.2 The `broadcast` device

Parent

`device`

Interface

`comms`

Description

The `broadcast` device allows clients to communicate with one another through Stage/Player; messages sent to one broadcast device will be received by all other broadcast devices.

Properties

Name	Values	Description
<none>		

Defaults

Name	Value
<none>	

5.3.3 The `gps` device (*disabled*)

Parent

`device`

Interface

`gps`

Description

The `gps` *is temporarily disabled, pending implementation of a proper GPS model, or at least one that returns GPS-style coordinates.*

Properties

Name	Values	Description
<none>		

Defaults

Name	Value
<none>	

5.3.4 The `idar` device

Parent

`device`

Interface

`idar`

Description

The `idar` device simulates the Infrared Data and Ranging sensor used by HRL Labs Pherobot. This device beams out short data strings which can be received by another IDAR, or by reflection from an obstacle. Received signal strength gives an approximate range.

Properties

Name	Values	Description
`<none>`		

Defaults

Name	Value
`idar_return`	`"IDARReceive"`
`size`	`[0.03 0.02]`
`shape`	`"ShapeRect"`

5.3.5 The `idarturret` device

Parent

`device`

Interface

`idarturret`

Description

The `idarturret` device simulates an array of `idar` devices, combined for efficiency.

Properties

Name	Values	Description
`<none>`		

Defaults

Name	Value
`<none>`	

5.3.6 The `gripper` device

Parent

`device`

Interface

`gripper`

Description

The `gripper` device simulates a 1-DOF gripper (i.e., the gripper can open and close), with inner and outer break beams.

Properties

Name	Values	Description
`consume`	`"true"` `"false"`	If `true`, the gripper will consume pucks, ie. they will disappear from the world; if `false`, the gripper can hold only one puck at a time and the puck will be dropped when the gripper opens.

Defaults

Name	Value
`consume`	`"true"`
`size`	`[0.08 0.3]`

5.3.7 The `laser` device

Parent

`device`

Interface

`gps`

Description

The `laser` device simulates a scaning laser range-finder with a 180° field-of-view (the SICK LMS200 to be exact).

Properties

Name	Values	Description
`min_res`	`float`	The angular resolution of a range sample in degrees
`max_range`	`float`	The max range of a scan beam in meters

Defaults

Name	Value
`min_res`	`0.25`
`max_range`	`8.0`
`size`	`[0.155 0.155]`
`color`	`"blue"`

5.3.8 The `fiducialfinder` device

Parent

`device`

Interface

`fiducial`

Description

The `fiducialfinder` locates models that have their `fiducial_id` property set. It returns the identity, range, bearing and orientation of the detected fiducials. Currently a `fiducialfinder` device must be the child of a `laser` device, though this constraint will be relaxed in future releases.

Properties

Name	Values	Description
<none>		

Defaults

Name	Value
<none>	

5.3.9 The `position` device

Parent

`device`

Interface

`position`

Description

The `position` device simulates a bare-bones differential-drive mobile robot.

Properties

Name	Values	Description
`<none>`		

Defaults

Name	Value
`<none>`	

5.3.10 The `omniposition` device

Parent

`device`

Interface

`position`

Description

The `omniposition` device simulates an omnidirectional mobile robot base.

Properties

Name	Values	Description
<none>		

Defaults

Name	Value
<none>	

5.3.11 The `ptz` device

Parent

`device`

Interface

`ptz`

Description

The `ptz` device simulates a pan-tilt-zoom camera. This device has the following properties.

Properties

Name	Values	Description
`lens`	`"normal"` `"wide"`	Select lens type: either normal (60° field-of-view) or wide (120° field-of-view).

Defaults

Name	Value
`lens`	`"normal"`

5.3.12 The `sonar` device

Parent

`device`

Interface

`gps`

Description

The `sonar` device simulates an array of sonar range sensors.

Properties

Name	Values	Description
`scount`	`numsonars`	The number of sonar transducers.
`spose[i]`	`[x y th]`	The pose of each transducer number `i`.

Defaults

Name	Value
`scount`	`8`
`spose[0]`	`[0 0 0]`
`spose[1]`	`[0 0 45]`
`spose[2]`	`[0 0 90]`
`spose[3]`	`[0 0 135]`
`spose[4]`	`[0 0 180]`
`spose[5]`	`[0 0 225]`
`spose[6]`	`[0 0 270]`
`spose[7]`	`[0 0 315]`

Notes

The default sonar array has 8 evenly spaced beams covering 360 degrees. The example file worlds/pioneer.inc provides sonar device configurations that match the Pioneer2 and AmigBot robots.

5.3.13 The `truth` device

Parent

`device`

Interface

`truth`

Description

The `truth` device allows clients to get and set the pose of simulated entities. Setting the pose of the `truth` device will 'teleport' the device's parent to a new location.

Properties

Name	Values	Description
<none>		

Defaults

Name	Value
<none>	

5.3.14 The `blobfinder` device

Parent

`device`

Interface

`blobfinder`

Description

The `blobfinder` device simulates the ACTS color blob detector. If the `blobfinder` device is a child of a `ptzdevice` it respects the field of view of the PTZ.

Properties

Name	Values	Description
`channels`	`["color0" "color1" ...]`	The color detected by each channel in the vision device. Descriptive color names from the X11 color database should be used (e.g. `"red"` or `"blue"`). Look for `rgb.txt` in your X installation).

Defaults

Name	Value
`channels`	`["red" "green" "blue" "yellow" "cyan" "magenta"]`

Bibliography

[1] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. pages 317–323, Coimbra, Portugal, July 2003.

[2] Brian P. Gerkey, Richard T. Vaughan, Kasper Støy, Andrew Howard, Gaurav S Sukhtame, and Maja J Matarić. Most Valuable Player: A Robot Device Server for Distributed Control. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1226–1231, Wailea, Hawaii, October 2001.

[3] Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 2121–2427, Las Vegas, Nevada, October 2003.