# EyeSim VR
# User Manual

EyeSim VR Team

Apr. 21, 2026

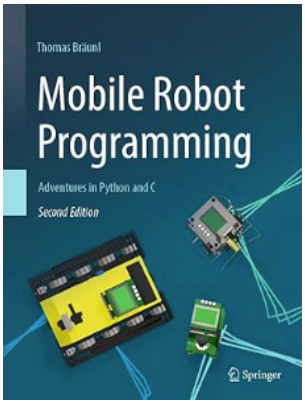
# 1 GENERAL INFORMATION

*EyeSim* lets users simulate multiple mobile robots, including driving robots, boats, submarines and robot manipulators in a realistic way. All robot programs use the RoBIOS-7 library for their sensors and actuators. See this web link and the Appendix of this manual:

```
https://roblab.org/eyebot/robios.html
```

Robot application programs can be written in either C, C++ or Python – and also C# for the Oculus Quest. The complied programs will then run in the EyeSim simulator.

For the setup, users can specify the robot environment either as a world file or a maze file an arbitrary number of objects and robots placed into the scene. All physical interactions between objects and robots will be realistically simulated by the underlying Unity-3D physics engine. For Further information, please refer to the textbook *"Mobile Robot Programming"*, Springer-Verlag, 2023 by Thomas Bräunl.




# 2 SYSTEM CONFIGURATION

Following is a table of tested OS and prerequisites for each OS. Please note that other system versions may also work, they are just not tested yet, if your system version cannot work properly, please file a bug according to section 5 or upgrade your system to the tested version.

The required supporting software should be installed in their default directory before start using the simulator.

| Operating Systems | OS Version | Prerequisites |
|---|---|---|
| Windows | Windows 8.1, 10 | None (includes cygwin) |
| Mac OS | 10.10.X | Install Xcode and Xquartz |
| Linux | 64bit | Install X11 library |

# 3 GETTING STARTED

## 3.1 Installation

**Mac OS:**

(1) Download and install XCode:
https://itunes.apple.com/au/app/xcode/id497799835?mt=12

(2) Download and install XQuartz: http://robotics.ee.uwa.edu.au/eyesim/ftp/aux/mac/
or https://www.xquartz.org/
Note: Your system default XQuartz app may need to be removed and reinstall from above link to include X11.

(3) Run following command in terminal to link the x11 library:
sudo ln -s /opt/X11/include/X11 /usr/local/include/X11

(4) Download and install EyeSim for macOS:
http://robotics.ee.uwa.edu.au/eyesim/ftp/

(5) Download and unzip eyesimX (EyeSim Examples):
https://robotics.ee.uwa.edu.au/eyesim/ftp/

**Windows:**

(1) Download and install EyeSim for Windows:
http://robotics.ee.uwa.edu.au/eyesim/ftp/

(2) Download cygwin.zip and Xming.zip from:
http://robotics.ee.uwa.edu.au/eyesim/ftp/aux/win/

(3) Unzip cygwin.zip to C:\Program Files (x86)\eyesim\cygwin

(4) Unzip Xming.zip to C:\Program Files (x86)\eyesim\Xming

(5) Download and unzip eyesimX (EyeSim Examples):
https://robotics.ee.uwa.edu.au/eyesim/ftp/

**Linux(64bit):**

(1) Install X11 library using following command:

```
sudo apt-get install libx11-dev
```

(2) Download the latest EyeSim for Linux: http://robotics.ee.uwa.edu.au/eyesim/ftp/

3) Unarchive the .tar.gz file and run the 'install.sh' script

4) Download and unzip eyesimX (EyeSim Examples):
https://robotics.ee.uwa.edu.au/eyesim/ftp/

## 3.2 System Menu

When the application is launched, a window as shown in Figure 3-1 will show up to let you select the resolution and graphics quality, and if ***windowed*** check box is ticked, the simulator will run in a window instead of full screen, you can view the key binds for control the simulator by choosing the **input** tab.

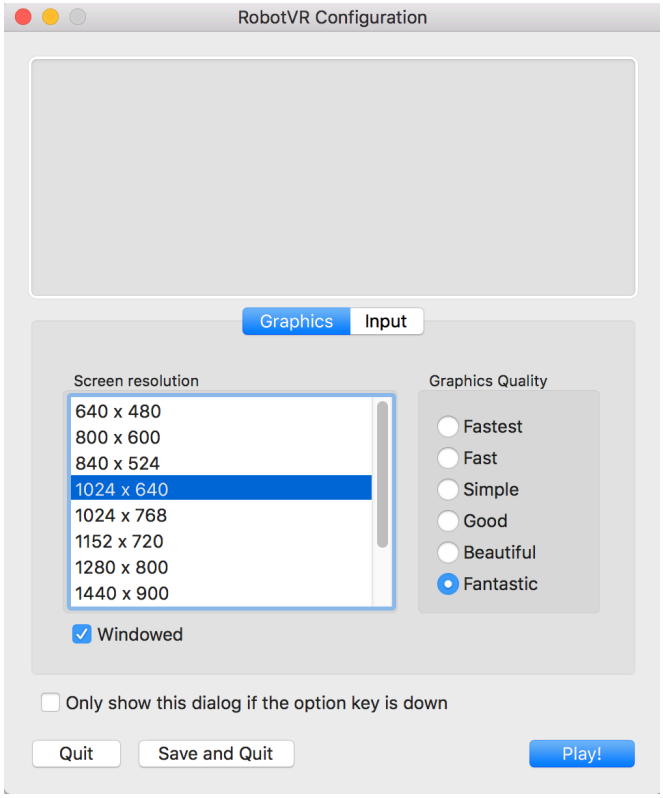



**Figure 3-1** Configuration window for Mac

The simulator main window shows after the ***Play!*** button is clicked, as shown in figure 3-2. The black plane in the middle is where the simulation will be, and the menus on the upper-left corner help to set up the simulation.

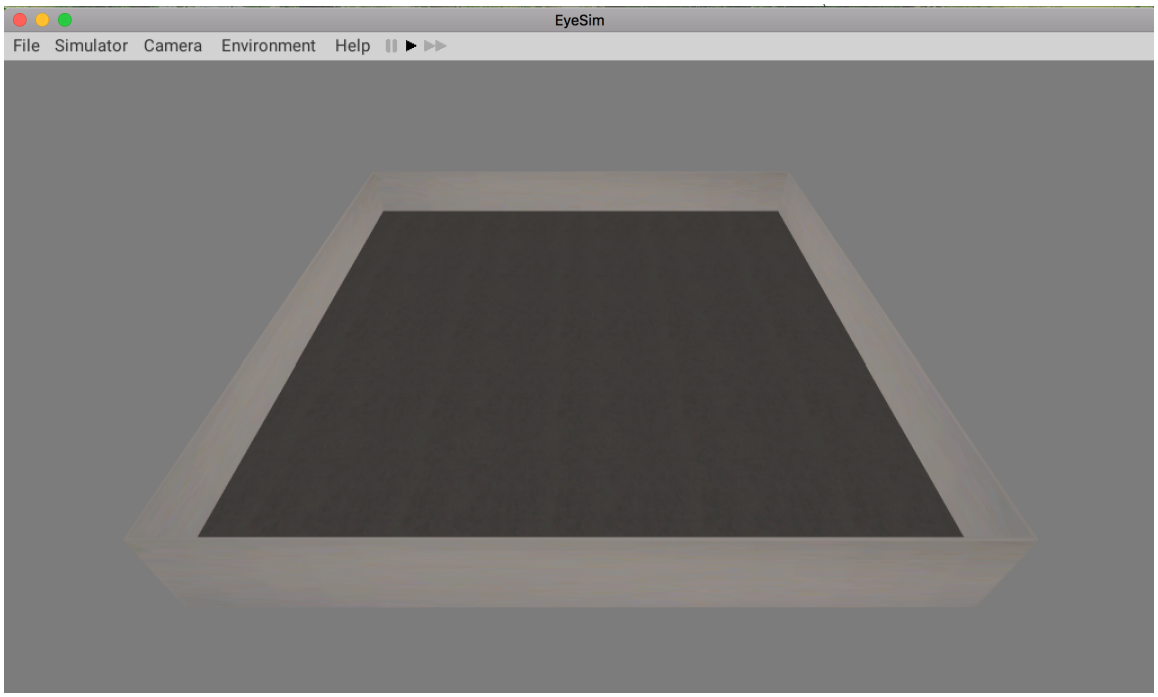



**Figure 3-2** Simulator main window

| File | Simulator | Camera | Environment | Misc | Help |
|---|---|---|---|---|---|
| Open Terminal | Add Robot | Birdseye View | Add Wall | 1v1 Soccer | RoBIOS API |
| Load Sim | Add Object | Follow Object | Remove Wall | 2v2 Soccer | View Log |
| Save Sim | Add Marker | Reset Camera | Paint Walls | | About |
| Load World | View Robots | | | | Changelog |
| Save World | View Objects | | | | |
| Create World | Save State | | | | |
| Reset World | Load State | | | | |
| Load Object | Pause | | | | |
| Settings | Resume | | | | |
| Exit | | | | | |

Table 3-1

The above table shows the menu functions of simulator, we can load world, save world, add objects and robots using these menus, detailed information related to each of the menu item will be introduced later.

# 4 USING THE SYSTEM

Please make sure you have all the required files/folders as described in section 3.1 before you proceed. Please make sure you launch the simulator application to perform following supported functionalities.

## 4.1 Adjust the Viewpoint

To adjust the simulation plane to a better viewpoint, you can use →←↑↓ arrow keys and **w, s, a, d** to move the plane around, and use the number key x and z to zoom in and out.

You can have a birds eye view of the simulation by choose from menu **Camera** -> **Birdseye View**. And you can reset the camera to normal viewpoint by **Camera** -> **Reset Camera**.

## 4.2 Load/Reset/Save/Create World

To save the current world, select **File -> Save World**, simulator will save current world settings in a file called **SavedWorld.wld** in the root of the EyeSim folder.

To load a world/maze file as simulation environment, in the main simulator window, select ***File***-> ***Load World*** submenu and then in the popped-up file selector, navigate to and click on the intended world or maze file (with wld or maz extension). The simulator will build simulation environment according to the selected file.

To reset the world/maze environment to the default one, select ***File***-> ***Reset World*** submenu. You can use the **File -> Create** World menu to adjust the dimensions of the current world.

## 4.3 Add/Remove Wall

To manually create a customized world, you can select **Environment** -> **Add Wall** to add wall, then you need to click two points in the simulator plane to set the starting point and ending point of the wall.

To remove any wall, you can choose **Environment** -> **Remove Wall**, and then click on the wall to be removed from environment, the selected wall will turn red when you move cursor on to it indicating it has been selected.

## 4.4 Place Robots/Objects/Markers

To place a robot in the simulator, select ***Simulator***-> ***Add Robot*** then selection a robot to add to the simulation.

To place an object (a can or a soccer ball) in the simulator, select ***Simulator***-> ***Add Object*** , then select the kind of object you want to put in the simulation.

Then you can move the cursor with the robot/object stuck on it to any valid place (robot/object is highlighted in green to indicate a valid placement spot, while in red not valid), and click your mouse to place it.

Though ***Simulator-> Add Marker***, you can add a colored marker point to the world in order to mark a position. You can also select a different color after the marker is placed, by double clicking on it and choose a color in the pop up inspector.

## 4.5 Inspect Robots/Objects/Markers

User can inspect the current status of any robot/object by double clicking on it, an inspector window will pop up.

**The robot inspector window:**

Figure 4-1 shows the inspector window of an object.

The **Camera** tab provides a camera image, and you can control the level of noise added to the image, by selecting **Salt and Pepper** and adjust the two parameters below.

The **PSD** tab shows the current psd readings of the robot, and you can also add errors to the PSD sensors by selecting **Error Enabled** and adjust the parameters of mean and std. Dev. of error. There's a toggle box called **Visualize Sensor**, when it is toggled, simulator will show the PSD sensor ray cast when performing the **PSDGet** command.

The **Driving** tab shows the current x,y coordinates of the robot and the rotation value Phi.

The **Control** tab can be used to select compiled simulation program files for simulation, and disconnect the control at any time.

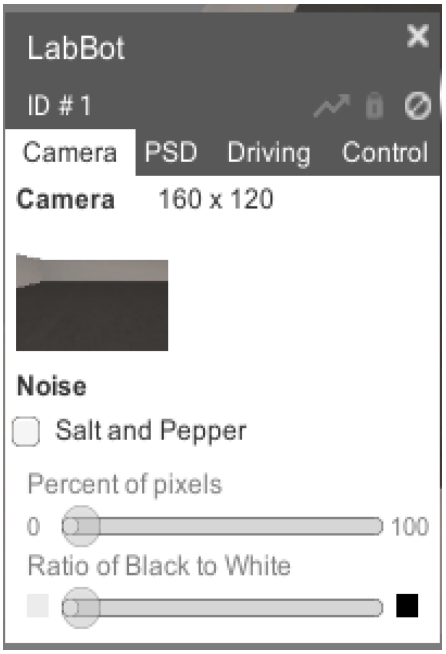



**Figure 4-1** Inspector window (robots)

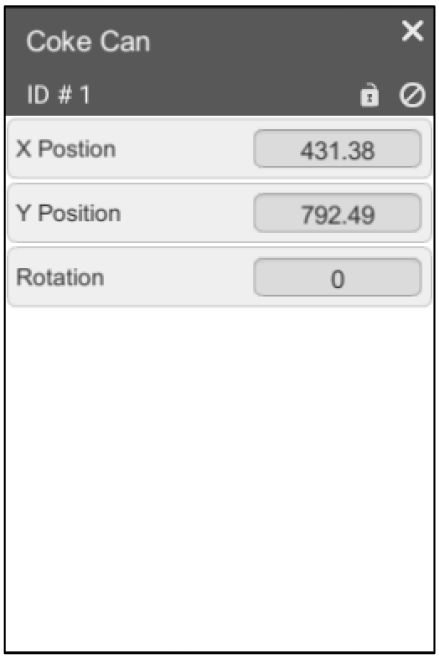



**Figure 4-2** Inspector window (objects)

**The object inspector window:**

Figure 4-2 shows the inspector window of an object, the information includes the name,

id of the inspected object, x and y coordinates of the object against the lower-left corner of the simulation plane, and the rotation parameter.

## 4.6 Relocate/Rotate a Robot/Object/Marker

You can move a robot, object or marker to any new valid spot after they have been added to the simulation.

To move a robot/object, you can double click on the target to open the inspector window, then click on the  icon so it becomes grey, a marker doesn't require this setting to move.

If you want to move the robot/object/marker, you can click on the target and drag your cursor to the desired spot, then click again to place it.

If you want to rotate the robot/object, you click on the target and drag until it is picked up and turned green, then use **-** and **=** key to control its rotation.

## 4.7 Relocate/Rotate a Robot/Object/Marker to Specified Value

Sometimes you want to move a robot/object/Marker to a specified position or rotate a robot/object to a specified degree.

To specify the x,y position or rotation degree of a robot/object/marker, you need to select submenu ***Simulator->Pause*** to make the position or rotation data editable, then double click on the target to open inspector window, you can see now you can input or change the position and rotation value of the target, after you typed in each desired value, you need to click elsewhere to make it work, after finishing the relocation, you should select ***Simulator->Resume*** to return to normal mode.

## 4.8 View all Robots/Objects

You can inspect all robots/objects in the simulation easier by selecting ***Simulator-> View Robots/View Objects*** submenu, this will create a list of all the robots/objects in the simulation as shown in figure 4-3. You can inspect the current status of any robot/object by simply clicking on the target in the list, and an inspector window of that target will pop up.

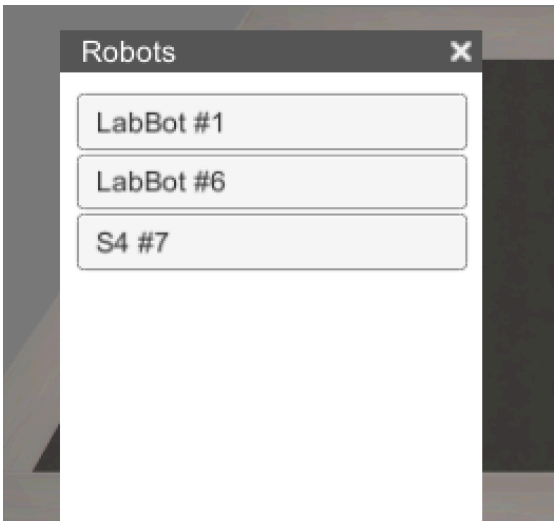



Figure 4-3 list of robots

### 4.9 Delete a Robot/Object/Marker

You can delete any robot/object/marker in the simulation by double clicking on the target to open the inspector window, then click the ⊘ icon to remove the target from the simulation.

### 4.10 Save Sim/Load Sim/Save State/Load State

During the process of your simulation, you may want to save the whole current simulation including the world, robots, objects and markers, you can select **File** -> **Save Sim**, simulator will automatically save the current simulation to a file called **SavedSim.sim**. You can restore the simulation at a later time by using **File** -> **Load Sim**, and select the SavedSim.sim file. Alternately, you can use **Similator** -> **Save State**, and **Simulator** -> **Load State** to quickly store and restore the simulation, without having to save and fetch from a sim file.

### 4.11 Create & compile programs

To compile example programs, go the subfolders under example folder, where you can find a file called **Makefile** together with other program files in C, type the following command in your terminal to compile all programs in this folder:

```
make
```

You will find one compiled file generated (in .x or .exe extension) for each program.

**To compile your robot application program:**

**Mac OS:**

Create a new folder, e.g. called "my" in eyesim folder. Then created a robot programe.g. in C in this folder (e.g. drive.c).

Open your mac terminal and **cd** to this folder. Type in following command to compile this program:

**gccsim -o drive.x drive.c**

The above command will generate an executable file called “drive.x” in this folder.

**Windows:**

Create a new folder, e.g. called **my** at any place you want, and create a program file called drive.c inside it.

In the simulator main window, select submenu ***File/Open Terminal*** , and the Cygwin terminal will pop up.

You can see your current folder is called **tmp** under Cygwin folder, you can navigate to your c disk using command:

**cd /cygdrive/c**

Then further navigate to your **my** folder, and run command.

**gccsim -o drive.x drive.c**

**Advanced Windows Users** (experience with cygwin / linux subsystem):

If you have an installed version of cygwin already on your machine, or are experienced using the Linux subsystem for Windows, you can use either of these to run your programs. The EyeSim libraries and header files are available by themselves.

**Linux:**

Create a new folder, e.g. called "my" in eyesim folder. Then created a robot program e.g. in C in this folder (e.g. drive.c).

Open your terminal and cd to this folder. Type in following command to compile the program:

**gcc -o drive.x drive.c - L ../lib -I ../include -leyesim -lX11 -lm**

The above command will generate an executable file called **drive.x** in this folder.

**Sample Program**: Here is a very simple robot program for your reference.

The program will drive a fixed distance of 600mm, then turn about 180˚.

```
#include "eyebot.h"

int main() {
  VWStraight(600, 30);
  VWWait();
  VWTurn(180,10);
  VWWait();
}
```

## 4.12 Start Simulation

**Requirements:**

There should be at least one robot in the simulation, if not, you should add a new robot to the simulation according to section 4.3.

There should be a compiled program file (with extension **.x**) placed in the folder. If not, you should first create a program file and have it compiled according to section 4.9.

There are two ways to control a robot.

**From command line:**

***For Mac OS/Linux:***

Open terminal and navigate the folder of any executable files you have compiled and run command: **./{your file name}** to run the executable programs.

***For Windows:***

Open the cygwin terminal and navigate (cd) to the directory where the executable file (e.g. drive.x) is located , then type: **./drive.x** in terminal window.

A response in the terminal as shown in figure 4-4 is expected to show, indicating the simulation is running, and you can watch the simulation in the application.

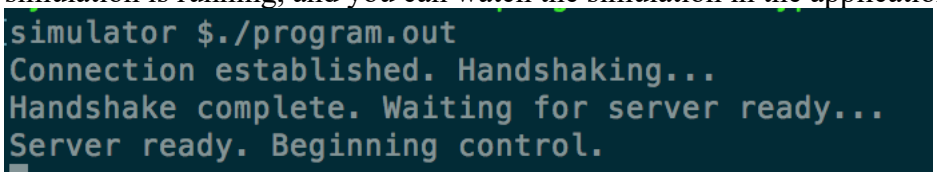



Figure 4-4 terminal response

**Note**: Multiple robots with different programs can be simulated concurrently. To start them, add them all to a ".sim" file.

**Select control program from robot inspector window:**

By selecting executable programs for each robot in the simulation, we can control multiple robots at the same time. Double click on any robot in the simulation, the inspector window will show up (as shown in figure 4-1). Click on the ***Select Control*** button, then navigate and select any compiled program file (with ".x" extension), the robot simulation will then start. To terminate the robot program, click on the ***Disconnect*** button in the inspector.

## 4.13 Pause/Resume/Speed up Simulation

You can pause and resume a simulation when the simulation is running, by selecting ***Simulator->Pause*** to pause the current simulation, and ***Simulator->Resume*** to resume

the simulation. You can also click on the icons on menu bar :  to pause, resume and speed up the simulation.

### 4.14 Add Trace to Robot

You can add trace to the robot during a simulation by double clicking on the target robot to show the inspector window, then click on the  icon on the upper right corner of the inspector window to may it highlighted instead of grey, you will see a green trace added to the route it has covered.

## 5 LOADING FILES

EyeSim supports the loading of the following external files:

- .robi files for custom Robots
- .esObj files for custom Objects
- .wld or .maz files for custom environments
- .sim files for preconfigured simulations

These files are standard text documents, with the appropriate extensions, and contain a series of commands to pass to the simulator. Each command begins with a keyword, as is followed by arguments separated by whitespace. An argument can be contained by double quotes (“”) if it contains a whitespace itself (such as a path to a file).

### 5.1 .robi Files

A .robi file specifies the parameters for a custom robot. Any line that begins with a # is treated as a comment, and is not processed by EyeSim. The keywords and corresponding arguments are as follows:

| **Keyword** | **Arguments Specification**<br>**Example** |
|---|---|
| drive | One of the following:<br>DIFFERENTIAL_DRIVE<br>ACKERMANN_DRIVE<br>OMNI_DRIVE |
| name | Name of the robot<br>MyRobot |
| mass | Mass in kg, followed by position center of mass in mm (kg x y z)<br>5 0 30 -50 |
| speed | Maximum linear velocity in mm/s<br>600 |
| turn | Maximum rotational velocity in deg/s<br>300 |

| | |
|---|---|
| model | Path to a .obj model, x y z offset (in mm), rotation about x y z axis (in degrees)<br>"\Robots\Models\LabBot.obj" 14 0 0 0 90 0 |
| axis | Distance between the center of the robot, and the center of the axis (vertical, horizontal in mm)<br>22.7 10.8 |
| psd | Id number, PSD name, position relative to robot's center x y z (in mm), and rotation x y z (in deg)<br>1 PSD_LEFT 30 0 80 0 90 0 |
| camera | Camera position relative to robot x y z (in mm), default pan and tilt (in deg), max pan and tilt (in deg)<br>40 50 70 0 0 90 90<br>For standard robots, SERVO1 and SERVO2 can be used for pan and tilt of the camera (see example program pan-tilt). |
| wheel | Wheel diameter (in mm), maximum rotational velocity (in deg/s), encoder ticks per revolution, distance between wheels (track, in mm)<br>45 3600 540 70 |
| lidar | position relative to robot's center x y z (in mm), and rotation x y z (in deg), angular range [1…360], tilt angle relative to driving plane (in deg, between -90 and 90), number of LIDAR points<br>0 0 0 0 0 0 180 10 360 |
| thruster | Id number, thruster name, thruster diameter, max speed, position relative to robot's center x y z (in mm), and rotation x y z (in deg)<br>1 THRUSTER_LEFT 180 1000 -320 290 -170 90 0 0 |
| fin | Id number, fin name, axis, max angle, size x y z (mm), and position relative to robot's center x y z (mm)<br>1 FIN_UPPER Y 90 10 100 100 0 250 -190 |
| buoyancy | Volume of robot (m^3)<br>0.012 |
| turn_offset | Offset value<br>-3 |

The first non-comment line of a .robi file must be the drive keyword and arguments. After a robot is loaded, it will be added to the Add Robot submenu (under Simulator), specified by the name parameter.

## 5.2 .esObj files

Custom objects can be loaded with .esObj files. These are simple world objects that interact physically with the robots. The keywords for this type of file is as follows:

| **Keyword** | **Arguments Specification**<br>**Example** |
|---|---|
| name | Name of the object<br>Bottle |
| obj | Path to a .obj file<br>"\Objects\Models\Bottle.obj" |
| scale | Scale of the object (modifies model size, positive number)<br>0.1 |
| mass | Mass (in kg), center of mass x y z (in mm)<br>1 0 0 0 |
| collider capsule | Center of capsule x y z (in mm), radius (in mm), height (in mm), vertical axis (character x, y, or z)<br>0 0 0 1 3 y |
| collider sphere | Center of sphere x y z (in mm), radius (in mm)<br>0 0 0 1 |
| collider box | Center of box x y z (in mm) side length of box x y z (in mm)<br>0 0 0 2 2 2 |
| buoyancy | Volume of object (m^3)<br>0.0012 |
| fixed | N/A |

An object consists of multiple colliders, to allow more complicated objects to be created. All the positions are relative to the center of the model.

### 5.3 .wld and .maz files

Custom environments can be loaded with .wld and .maz files. A .wld file consists of a floor, walls, and an optional floor texture. The floor can be specified by the keyword floor followed by the width and height in mm:

```
floor 2000 2000
```

Or by the width and height keywords

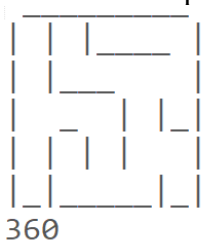
```
width 2000
height 2000
```

A texture to apply to the floor is specified by the keyword floor_texture, followed by a path to a .png file.

Walls have no keyword, and are simply lines with 4 numbers: x1 y1 x2 y2 eg:

```
0 0 1000 1000
```

Will create a diagonal wall from (0, 0) to (1000,1000).
Maze files, specified by .maz, contain an ASCII representation of what the maze looks like, using the characters | and _, with the very last line being the wall size in mm. An example of a maze:




### 5.4 .sim files

A sim file is used to store configurations, for quickly repeating experiments, by setting up the environment and objects. A sim file can consist of the following keywords:

| **Keyword** | **Arguments Specification**<br>**Example** |
|---|---|
| world | Path to a .wld or .maz file<br>"\Worlds\MyWorld.wld" |
| robi | Path to a .robi file<br>"\Robots\MyRobot.robi" |
| object | Path to an .esObj file<br>"\Objects\MyObject.esObj" |
| object_name | Name of an actual object, position x y (in mm) rotation phi (in deg), path to executable (if a robot)<br>labbot 1000 1000 0 "\Programs\Drive.x"<br>can 1500 1500 0 |
| marker | Position of marker x y (in mm), color of marker r g b (0 to 255)<br>1000 1000 255 0 255 |
| settings | Toggle various in-game settings.<br>VIS – Enable PSD visualization for all robots<br>TRACE – Enable path tracing for all robots<br>Parameters can be used together, e.g.:<br>'settings VIS TRACE' |

Note that for object_name, the keyword is the name of the object itself (ie labbot, s4, can, etc.) This can also be the name of a robot or object that has been loaded previously with the robi or obj keywords.

# 6 ROBOT MANIPULATORS

Apart from driving and swimming robot, EyeSim can also simulate robot manipulators. The popular UR5 robot is predefined in the library. In a ".sim" file, a UR 5 robot can be placed with a single line command, e.g. at position 1m, 1m, zero rotation and executable program file "joints.x":

```
UR5 1000 1000 0 joints.x
```

The application program can then set each individual robot joint to an angle in range [0, 255] by using the "SEVOSet" command. All joints are labeled from 1, so for the UR5 the joints are [1, 6]. E.g. for setting joint 4 to its middle position (128), use the command:

```
SERVOSet(4, 128);
```

New manipulator designs can be created through a custom ".robi" file. Each manipulator joint is specified there with four parameters in accordance to the Denavit-Hartenberg (DH) Notation: ***d, theta, r, alpha***

The following example is for a **single** link with **two degrees of freedom (green)** and **two constant values (blue)**:

```
8000(-4000|2000), 0(0|180), 0(0|0), 90(0|0)
```

means:

- **d**: **initial** length is 800.0 mm, **linear** joint (relative).
  **variable** length is between 8,000–4,000=4,000 and 8,000+2,000=10,000
- **theta**: **initial** angle is 0°, **angular joint** (absolute).
  **variable** angle is between 0 and 180 degrees
- **r**: **initial** length is 0.0 mm long
  **no range** difference (0|0), therefore **constant** distance
- **alpha**: **initial** angle of 90°
  **no range** difference (0|0), therefore **constant** angle

**Note:**

- Use a colon symbol ":" to separate subsequent joints.
- Use keyword "gripper" to add a standard gripper as end-effector
- 

The following ".robi" script file creates a **4-dof** robot (1 linear and 3 rotary joints), the first link/joint has 2 dof as shown above (length *d* and angle *theta*), the following two link/joints each have 1 dof rotation *theta*).

```
drive MANIPULATOR_DRIVE
name "custom"
parameters 8000(-4000|2000), 0(0|180), 0(0|0), 90(0|0) :
           4000(0|0),        0(0|180), 0(0|0), 90(0|0) :
           2000(0|0),        0(0|180), 0(0|0), 0(0|0)
gripper
```

To move a joint, use the command SERVOSet: Each limb has 4 parameters (*d, theta, r, alpha*), so IDs [1..4] are for the first link/joint, IDs [5..8] are for the second link/joint, and so on.

**Examples:**

- To set 'd' of the first limb, its ID would be 1.
- To set 'theta' of the third limb, its ID would be 10.
- To set the gripper of a 3-joint manipulator, its ID would be 13 (3*4 + 1).

All parameters can be set between 1 and 255, these scale between the range set upon creation. For example,

```
SERVOSet(1, 255);
```

will extend 'd' of the first limb to its maximum length (10000 in the example).

```
SERVOSet(13, 1);
```

will open the gripper, assuming it is a 3-joint robot.

**Mounting Manipulators onto Robot Vehicles**

Mounting of a manipulator is done in the sim file:

1) LabBot 1194 780 89 circles.x
2) Manipulator LittleArm.robi 0 0 0 -45 0 0 mount 1 theta_set.x

First, the vehicle is created in line (1). The Manipulator is then added in line (2). It refers to the description file "LittleArm.robi", which contains its specifications.
The next 6 parameters determine the relative position and angle (x, y, z, alpha, beta, gamma) on the mounted vehicle. The command 'mount 1' tells the system onto which robot vehicle the manipulator is being mounting. Here it mounts the manipulator to the first robot defined in the simulation.

# 7 BUG REPORTING

If you have detected any bug or considered any function missing or needed to be improved while using this EyeSim-VR simulator, you are highly encouraged to report the bug or suggestion to us for further corrections and improvements.
We are using Bugzilla as an online bug-reporting system, you will see the main page of Bugzilla as shown in figure 5-1 by accessing following link:
http://robotics.ee.uwa.edu.au/bugzilla/.


**Figure 5-1** Bugzilla

## 7.1 Create an Account/login

### Create account

If it is the first time you are using this system, you should open a new account by clicking on the ***Open a New Account*** icon, then follow the steps to create your account. (note: sometimes it may take a few minutes for Bugzilla to send the registration email, please be patient). you will automatically get logged in after you have created your account.

### Log in

You can log in directly if you have an account through the ***LogIn*** menu.

## 7.2 File a Bug

After logging in, you can click the File a Bug icon to report a bug. Then after clicking on **Robot-VR** as selected product, you will see a report form as shown in figure 5-2. Please select the specific component that is related to the bug, and **Severity** of the bug, **Hardware type** of the computer you are using and the **OS**.

In the **Summary** section, you are supposed to write a succinct sentence summarizing the bug.

In the **Description** section, the details of the bug, like when it occu4rs, what is the setup of environment, what are the previous actions you have performed, what you suspect has led to the bug should be presented.

In the **Attachment** section, you can add attachment like a snapshot of the bug, the error message or some other supporting files. Then you can click on Submit Bug button to submit the bug report.

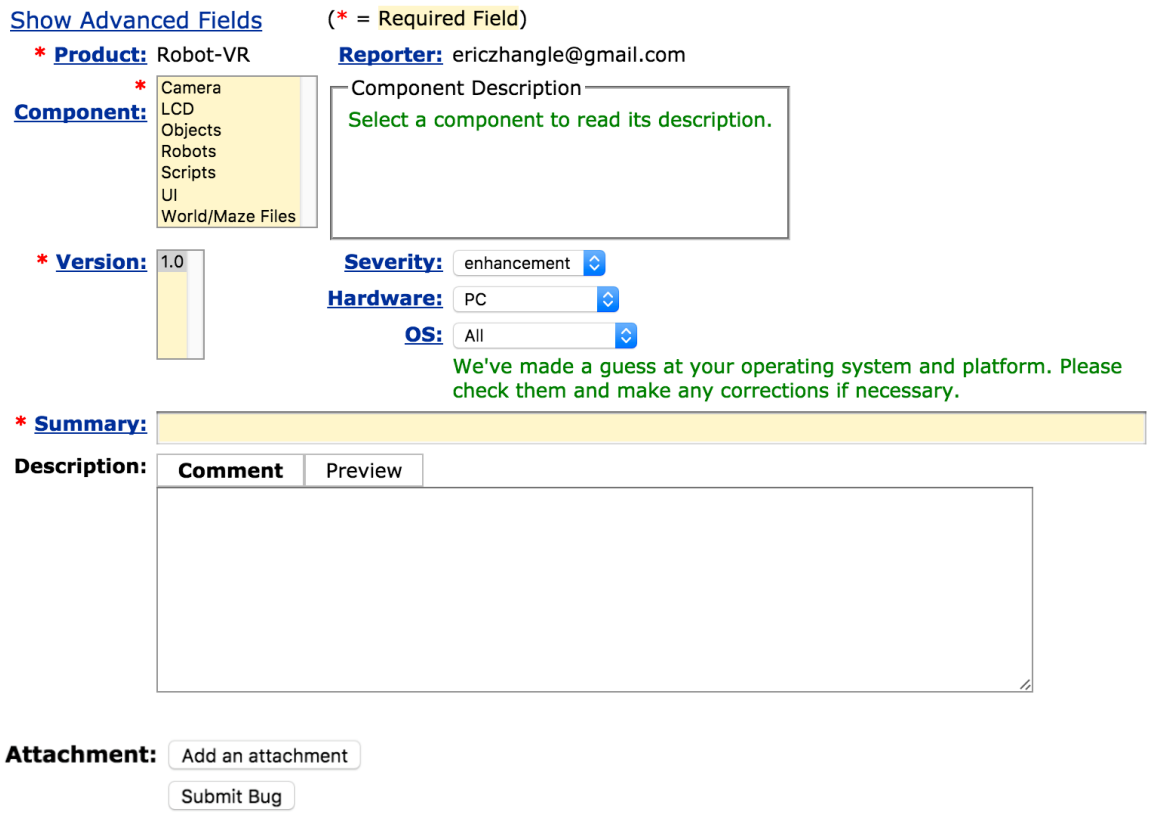



**Figure 5-2** Bug report form

# Appendix A — RoBIOS-7 Library Functions

See latest update on: http://roblab.org/eyebot/Robios7.html

## LCD Output

```
int LCDPrintf(const char *format, ...);                          // Print string and arguments on LCD
int LCDSetPrintf(int row, int column, const char *format, ...);  // Printf from given position
int LCDClear(void);                                             // Clear the LCD display and display buffers
int LCDSetPos(int row, int column);                  // Set cursor position in pixels for subsequent printf
int LCDGetPos(int *row, int *column);                // Read current cursor position
int LCDSetColor(COLOR fg, COLOR bg);                 // Set color for subsequent printf
int LCDSetFont(int font, int variation);             // Set font for subsequent print operation
int LCDSetFontSize(int fontsize);                    // Set font-size (7..18) for subsequent print operation
int LCDSetMode(int mode);                           // Set LCD Mode (0=default)
int LCDMenu(char *st1, char *st2, char *st3, char *st4); // Set menu entries for soft buttons
int LCDMenuI(int pos, char *string, COLOR fg, COLOR bg); // Set menu for i-th entry with color [1..4]
int LCDGetSize(int *x, int *y);                          // Get LCD resolution in pixels
int LCDPixel(int x, int y, COLOR col);                    // Set one pixel on LCD
COLOR LCDGetPixel (int x, int y);                         // Read pixel value from LCD
int LCDLine(int x1, int y1, int x2, int y2, COLOR col);   // Draw line
int LCDArea(int x1, int y1, int x2, int y2, COLOR col, int fill); // Draw filled/hollow rectangle
int LCDCircle(int x1, int y1, int size, COLOR col, int fill);     // Draw filled/hollow circle
int LCDImageSize(int t);                                 // Define image type for LCD (default QVGA; 0,0; full)
int LCDImageStart(int x, int y, int xs, int ys);     // Define image start position and size
int LCDImage(BYTE *img);                             // Print color image at screen start pos. and size
int LCDImageGray(BYTE *g);                           // Print gray image [0..255] black..white
int LCDImageBinary(BYTE *b);                          // Print binary image [0..1] white..black
int LCDRefresh(void);                                // Refresh LCD output
```

## Keys

```
int KEYGet(void);               // Blocking read (and wait) for key press (returns KEY1..KEY4)
int KEYRead(void);              // Non-blocking read of key press (returns NOKEY=0 if no key)
int KEYWait(int key);           // Wait until specified key has been pressed (use ANYKEY for any key)
int KEYGetXY (int *x, int *y);  // Blocking read for touch at any position, returns coordinates
int KEYReadXY(int *x, int *y);  // Non-blocking read for touch at any position, returns coordinates
```

```
Key Constants:  KEY1..KEY4, ANYKEY, NOKEY
```

# Camera

```
int CAMInit(int resolution);    // Change camera resolution (will also set IP resolution)
int CAMRelease(void);          // Stops camera stream
int CAMGet(BYTE *buf);         // Read one color camera image
int CAMGetGray(BYTE *buf);     // Read gray scale camera image
```

For the following functions, the Python API differs slightly as indicated.

```
def CAMGet    ()  -> POINTER(c_byte):
def CAMGetGray() -> POINTER(c_byte):
```

## Image Processing

```
int   IPSetSize(int resolution);              // Set IP resolution using CAM constants (also set by CAMInit)
int   IPReadFile(char *filename, BYTE* img);   // Read PNM file, return 3:color, 2:gray, 1:b/w, -1:error
int   IPWriteFile(char *filename, BYTE* img);       // Write color PNM file
int   IPWriteFileGray(char *filename, BYTE* gray);  // Write gray scale PGM file
void  IPLaplace(BYTE* grayIn, BYTE* grayOut);       // Laplace edge detection on gray image
void  IPSobel(BYTE* grayIn, BYTE* grayOut);                    // Sobel edge detection on gray image
void  IPCol2Gray(BYTE* imgIn, BYTE* grayOut);                  // Transfer color to gray
```

```
void  IPGray2Col(BYTE* imgIn, BYTE* colOut);                    // Transfer gray to color
void  IPRGB2Col (BYTE* r, BYTE* g, BYTE* b, BYTE* imgOut);      // Transform 3*gray to color
void  IPCol2HSI (BYTE* img, BYTE* h, BYTE* s, BYTE* i);         // Transform RGB image to HSI
void  IPOverlay(BYTE* c1, BYTE* c2, BYTE* cOut);                // Overlay c2 onto c1, all color images
void  IPOverlayGray(BYTE* g1, BYTE* g2, COLOR col, BYTE* cOut); // Overlay gray image g2 onto g1, using col
COLOR IPPRGB2Col(BYTE r, BYTE g, BYTE b);                       // PIXEL: RGB to color
void  IPPCol2RGB(COLOR col, BYTE* r, BYTE* g, BYTE* b);         // PIXEL: color to RGB
void  IPPCol2HSI(COLOR c, BYTE* h, BYTE* s, BYTE* i);           // PIXEL: RGB to HSI for pixel
BYTE  IPPRGB2Hue(BYTE r, BYTE g, BYTE b);                       // PIXEL: Convert RGB to hue (0 for gray)
void  IPPRGB2HSI(BYTE r, BYTE g, BYTE b, BYTE* h, BYTE* s, BYTE* i); // PIXEL: Convert RGB to hue, sat, int
```

## Distance Sensors

```
int PSDGet(int psd);                            // Read distance value in mm from PSD sensor [1..6]
int PSDGetRaw(int psd);                         // Read raw value from PSD sensor [1..6]
int LIDARGet(int distance[]);                   // Measure distances in [mm]; default 360Â° and 360 points
int LIDARSet(int range, int tilt, int points);  // range [1..360Â°], tilt angle down, number of points
```

### Servos and Motors

```
int SERVOSet(int servo, int angle);             // Set servo [1..14] position to [1..255] or power down (0)
int SERVOSetRaw (int servo, int angle);         // Set servo [1..14] position bypassing HDT
int SERVORange(int servo, int low, int high);   // Set servo [1..14] limits in 1/100 sec
int MOTORDrive(int motor, int speed);           // Set motor [1..4] speed in percent [-100 ..+100]
int MOTORDriveRaw(int motor, int speed);        // Set motor [1..4] speed bypassing HDT
int MOTORPID(int motor, int p, int i, int d);   // Set motor [1..4] PID controller values [1..255]
int MOTORPIDOff(int motor);                     // Stop PID control loop
int MOTORSpeed(int motor, int ticks);           // Set controlled motor speed in ticks/100 sec
int ENCODERRead(int quad);                      // Read quadrature encoder [1..4]
int ENCODERReset(int quad);                     // Set encoder value to 0 [1..4]
```

### V-Omega Drive System

```
int VWSetSpeed(int linSpeed, int angSpeed);     // Set fixed linSpeed  [mm/s] and [degrees/s]
int VWGetSpeed(int *linSspeed, int *angSpeed);  // Read current speeds [mm/s] and [degrees/s]
int VWSetPosition(int x, int y, int phi);       // Set robot position to x, y [mm], phi [degrees]
int VWGetPosition(int *x, int *y, int *phi);    // Get robot position as x, y [mm], phi [degrees]
int VWStraight(int dist, int lin_speed);        // Drive straight, dist [mm], lin. speed [mm/s]
int VWTurn(int angle, int ang_speed);           // Turn on spot, angle [degrees], ang. speed [degrees/s]
int VWCurve(int dist, int angle, int lin_speed);// Drive Curve, dist [mm], angle (orientation change)
[degrees], lin. speed [mm/s]
int VWDrive(int dx, int dy, int lin_speed);     // Drive x[mm] straight and y[mm] left, x>|y|
int VWRemain(void);                             // Return remaining drive distance in [mm]
int VWDone(void);                               // Non-blocking check whether drive is finished (1) or not
(0)
int VWWait(void);                               // Suspend current thread until drive operation has finished
int VWStalled(void);                            // Returns number of stalled motor [1..2], 3 if both stalled
```

### Radio Communication

```
int RADIOInit(void);                            // Start radio communication
int RADIOGetID(void);                           // Get own radio ID
int RADIOSend(int id, char* buf);               // Send string (Null terminated) to ID destination
int RADIOReceive(int *id_no, char* buf, int size); // Wait for message, then fill in sender ID and data,
returns number of chars received
int RADIOCheck(void);                           // Check if message is waiting: 0 or 1 (non-blocking); -1 if
error
int RADIOStatus(int IDlist[]);                  // Returns number of robots (incl. self) and list of IDs in
network
```

```
int RADIORelease(void);                         // Terminate radio communication
```

ID numbers match last byte of robots' IP addresses.

For the following functions, the Python API differs slightly as indicated.

```
def RADIOReceive(int size) -> [id_no, buf]  # max 1024 Bytes
def RADIOStatus()          -> int IDList[]  # max  256 entries
```

## Simulation-Only Commands

The following commands are only available in simulation, not on readl robots.

```
void SIMGetRobot (int id, int *x, int *y, int *z, int *phi);
void SIMSetRobot (int id, int  x, int  y, int  z, int  phi);
void SIMGetObject(int id, int *x, int *y, int *z, int *phi);
void SIMSetObject(int id, int  x, int  y, int  z, int  phi);
int  SIMGetRobotCount()
int  SIMGetObjectCount()
```

## RoBIOS-Only Commands

Simulation is happening at an intermediate level, so the following low-level RoBIOS functions are not available in simulation: Quadrature encoder, Serial communication, Analog and Digital I/O, IRTV (infrared TV remote).