Proceedings of Australasian Conference on Robotics and Automation, 3-5 Dec 2012, Victoria University of Wellington, New Zealand.

jmeSim: An Open Source, Multi Platform Robotics Simulator

Adam Haber, Matthew McGill, Claude Sammut
ARC Centre of Excellence in Autonomous Systems
School of Computer Science and Engineering, UNSW
Sydney, NSW, 2052, Australia.
{adamh,mmcgill,claude}@cse.unsw.edu.au

Abstract

Simulation of autonomous and teleoperated robots has become a staple of the robotics research community. Accurate physics simulation, high quality rendering, open source code, and multi platform implementability are essential qualities for simulation packages. Many simulation environments exist, each with a range of advantages and disadvantages, that address the above features to different degrees, but none achieves them all. Recently, the Robot Operating System (ROS) [Quigley *et al.*, 2009] has provided researchers with a standardised platform for robotics research, and has been widely adopted. This paper presents jmeSim, an open source, multi-platform robotic simulation package, which provides excellent graphical and physical fidelity, and provides tight integration with ROS. The simulation environment is described in detail, and several demonstrations of its capabilities are presented.

1 Introduction

Computer simulation has proven itself to be an invaluable tool in robotics research, particularly for mobile autonomous robots, for several reasons. Firstly, robots are fragile, potentially dangerous, and notoriously difficult to test and develop due to their inherent complexity. The ability to test and develop much of the control software in simulation can mitigate the risks and costs of hardware failure. A second drawback to developing software on robots themselves is that in most cases, robots come from offsite, at large expense. Simulation allows software to be developed and tested by researchers who either do not have access to, or are waiting on delivery of robots. An additional attraction of simulation is the ability to test robotic software in situations that are not feasible to create in reality, such as disaster scenarios, fires and the like. Finally, a truly invaluable advantage of simulation for robotics is the ability to run hundreds or thousands of trials for the purposes of *learning* robot control algorithms. This process is crucial to the development of complex articulated, particularly legged, robots, as hand coding control algorithms is extremely difficult in this case. Indeed, policies learned from trials in simulation are often applied to control physical robots, as long as the simulation has sufficient fidelity that the policy is transferable. Robot simulation software has improved to the point where high fidelity simulation of a large variety of sensors, complex robots, and rich environments is now possible. This has allowed the use of simulation to pervade all types of robotic research.

Robot software packages have tended to be customised and fine tuned to their particular niche. Recently, however, the free and open source Robot Operating System (ROS) [Quigley *et al.*, 2009] has become widely adopted, and is slowly bringing the beginnings of standardisation to the robotics research community [Cousins, 2010; Kamei *et al.*, 2012]. Integration with the ROS framework allows access to a large codebase of robotic software for perception, most notably the Point Cloud Library (PCL) [Rusu and Cousins, 2011], or navigation, reasoning, and any conceivable aspect of robotics. In light of this, it is desirable for robotic simulators to integrate with ROS to provide access to this software. The simulation package Gazebo [Koenig and Howard, 2004], is distributed with ROS, however, this package is currently limited to the Linux operating system.

An ideal simulation package would not only be easily integrated into the ROS framework, but would in addition, be multi platform, high fidelity in terms of graphics and physics rendering, and open source. High graphical fidelity in a robot simulation is not merely a matter of aesthetics. In robot simulation, the sensory input to the robots perceptual algorithms comes from whichever virtual sensors are provided by the simulation. For example, virtual cameras use the rendering engine of the simulator to provide their images. If the simulated camera image bears little resemblance to a real camera feed,

robotic vision algorithms cannot be tested in the simulation framework, rendering it incomplete. Similarly, high fidelity physics is essential, particularly in machine learning applications, where the robot refines a policy that based on the response of the simulated world to its actions. If the physics is not sufficiently realistic, there is little chance of transferring such a policy to a real robot.

Below, we review the attributes of several of the more popular simulation packages, and conclude that none of the currently available environments meet all of these key criteria. We then describe the development of a new simulation engine, jmeSim, which meets the four requirements we have enumerated above: it is integrated with ROS, multi platform, high quality, and open source.

2 Related Work

In this section, current simulation packages are evaluated in terms of their performance in the areas described above: the accuracy and quality of their graphics, physics simulation fidelity, quality of integration with ROS or competing robot software architectures, customisability, complement of robot or environment models, open source code, and multi platform capabilities. Our review of these simulators is summarised in Table 1.

2.1 USARSim

USARSim is an open source, high fidelity robot simulation package originally based on the Unreal Game Engine 2.0, a commercially available game engine, although it has since migrated to version 3.0. Unreal 3.0 produces state of the art graphics rendering, and the NVIDIA PhysX physics simulation engine, which uses hardware acceleration. Thus, the graphics and physics fidelity is excellent in USARSim. There is also an open source module for integration with ROS, which has recently been developed [Balakirsky and Kootbally, 2012]. A further advantage of USARSim is that is it currently the simulation engine used for the RoboCup Rescue Virtual Robot Competition [Bastiaan, 2010]. The main drawbacks of USARSim are the fact that the Unreal game engine is commercially licensed, so users must purchase a copy of the game engine to use the simulation package, and the lowest levels of the simulation are opaque to the user.

2.2 Gazebo

Gazebo is an open source robotic simulation package [Koenig and Howard, 2004], which was developed as part of the well-known Player/Stage suite [Gerkey *et al.*, 2003], although it has since become tightly integrated with ROS. The Gazebo simulator is built on top of the Open Dynamics Engine (ODE) physics engine [Smith, 2012], which is an older physics engine, and does not have the multithreading or hardware acceleration capabilities of newer engines such as PhysX in USARSim, or Bullet [Coumans, 2012]. Thus physics performance in Gazebo can be somewhat slower, or less accurate than newer engines. Gazebo uses the open source OGRE rendering engine, which produces very good graphics fidelity. Although Gazebo is packaged with ROS, which is multiplatform, as mentioned above, Gazebo is currently Linux only, limiting its utility for Mac or Windows users.

2.3 Webots

Webots is a commercial robot simulation package that is perhaps the most developed and fully featured of those surveyed here [Michel, 1998]. Like Gazebo, it is built on ODE physics, although its excellent rendering capabilities are more impressive. It includes fully functional, high fidelity models of 15 different robotic platforms, a large library of indoor and outdoor objects, and can accept robot control code written in many different languages. Webots is cross platform, installable on Windows, Mac and Linux operating systems, includes a interface to ROS, and a particular strength is its scalability to large numbers of mobile robots. A software license for the full featured version of Webots currently retails for US$2300, rendering it prohibitively expensive for many researchers. Further, being commercially licensed means that reproducing results developed in Webots is impossible for those who have not purchased a license, and limits the customisability of the software itself.

Although the existing selection of robotic simulation tools, some of which are reviewed in the previous section, have many impressive capabilities, there is not yet a truly multi platform, open source alternative which provides the high fidelity simulation, and tight integration with ROS that is required in many applications. In order to address this need, we have developed jmeSim, which unlike any of the currently available simulation packages, meets all the requirements.

3 jmeSim

Like several other robot simulators, jmeSim is built on top of a game engine, specifically, the open source jMonkeyEngine3 Game Engine [Vainer, 2012]. Since both jMonkeyEngine3 and jmeSim are written in pure Java, jmeSim is architecture neutral: it can run on any operating system.

The jMonkeyEngine3 engine uses OpenGL for rendering, incorporating very high performance rendering abilities: shaders and per-pixel lighting, and an array of post-processing filters, which make for excellent graphical performance at real time frame rates on desktop machines. Qualitatively speaking, graphics performance of jMoneySim is as good or better than other robotics simulators, including commercial packages. Physics in

Simulation Package	USARSim	Gazebo	Webots	jmeSim
Graphics fidelity	Very good	Very Good	Excellent	Excellent
Physics accuracy	Excellent	Good	Good	Excellent
Integration with ROS	Yes	Yes	Yes	Yes
Environment and robot models	Extensive	Some	Comprehensive	Some
Licensing	Commercial	Open source	Commercial	Open source
Multi platform	No	No	Yes	Yes

Table 1: Summary of the capabilities of some popular robotic simulation packages.

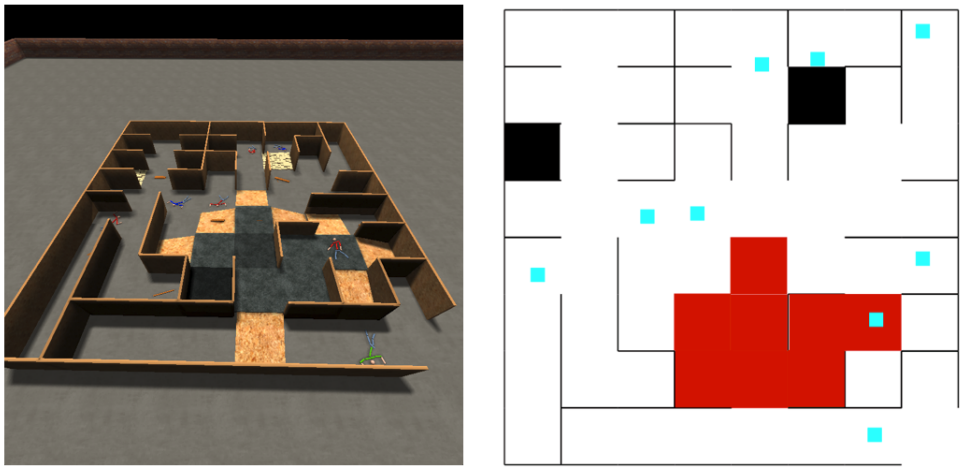

Figure 1: Environment generation in jmeSim. Right: A vector graphics world file specifying locations of victims, raised areas, and obstacles in blue, red, and black, respectively. Left: The jmeSim environment generated by this world file.

jmeSim is performed by jBullet, a Java port of the Bullet Physics library [Coumans, 2012]. Bullet physics provides a multi-threaded dynamics solver which provides excellent accuracy, speed, and stability, outperforming other open source and even some commercially licensed physics engines [Boeing and Bräunl, 2007]. The engine runs as two threads, one for physics updates, and another to render the scene. Notably, Bullet provides facility for convex hull generation for complex meshes, which means that mesh accurate collisions can be modelled for shapes that are not easily expressible as a collection of primitives. This is a decided advantage over older physics engines such as ODE, which does not have this capacity natively. Although a quantitative analysis of the CPU requirements of jmeSim is beyond the scope of this paper, the multi threaded Bullet physics allow simulations involving complex articulated robots and hundreds of physics objects to run at realistic frame rates on standard desktop hardware.

jMonkeyEngine3 provides a collection of low level libraries for object culling, rendering, and texturing, so that simulator objects can be specified in a high level manner. jmeSim takes advantage of these capabilities by providing an abstract interface for defining robots, world objects, and environment configurations. Robot initial locations and sensor capabilities are specified in a configuration file, and environmental descriptions are specified using vector graphics files, which can be drawn by hand, or generated programmatically, to populate the world with rooms, corridors and other objects. Since our research is primarily in the area of mobile rescue robots, the simulation package includes a module for batch generation of random mazes, including obstacles and victims, based on a modified version of Prim's algorithm [Wiener, 2011]. An example of a generated vector graphics file and the corresponding simulation environment produced is shown in Figure 1.

jmeSim contains a model of a wheeled rescue robot, with a four degree-of-freedom sensor head, containing an array of sensors. The robot model, along with a photo

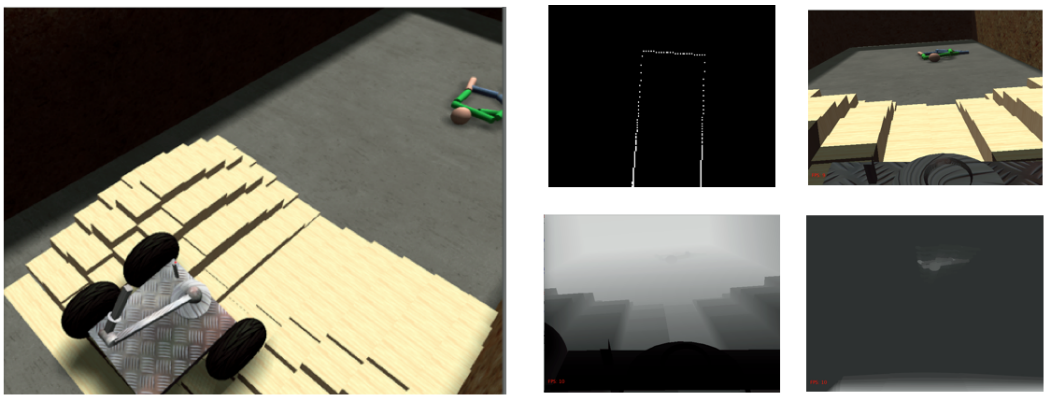

Figure 2: Sensor data in jmeSim. On the left, the main camera view shows a simulated robot traversing some rough terrain, approaching a victim lying on the ground. Simulated sensor data feeds clockwise from top right: colour camera, thermal camera, depth camera, and laser range finder.

of the real robot it is based on, is shown in Figure 4(d). As in many simulation packages, jMonkeyEngine allows specification of complex articulated physical objects in terms of primitive geometric objects, thus any conceivable robot model could be included into jmeSim.

If the simulation software is implemented in a standalone configuration, control code for robots can be written in Java. However, the simulation is designed for integration with other packages to define robot control, in particular, ROS, which is language neutral, allowing control code to written in any required formalism. Integration of ROS and jmeSim is described in detail in later sections.

jmeSim provides an array of sensors which provide data typically needed by an autonomous mobile robot. The sensors built in to jmeSim are laser range finder, sonar range finder, colour camera, 3D / depth camera, thermal camera, wheel odometry, and inertial measurement unit for orientation sensing. Simple sensors such as sonar, and LIDAR are modelled using physics collision detection, and ray casts respectively. More advanced sensors such as cameras and depth cameras require rendering the simulation scene from the perspective of the simulated camera, and jMoneySim takes advantage of the multithreaded rendering capabilities of jMonkeyEngine3 to produce high frame rates. The colour camera is produced in exactly the same way as the main scene image, however the depth camera is slightly more complex. For each virtual camera in the scene, OpenGL, the rendering engine used by jmeSim, stores the distances of every world object in view in order to produce a visualisation with objects appropriately occluded. These values are stored in an array called the depth buffer or Z-buffer. The depth camera image in jmeSim is rendered by reading the depth buffer, and constructing an image using the values. A detailed description of the method can be found in [Sheh, 2010].

We are particularly interested in using jmeSim for simulating autonomous robotic rescue, thus it is important to model a thermal camera, which is essential for detecting body heat. As such, jmeSim includes a thermal camera sensor, which assigns objects with a temperature, and renders their brightness in a greyscale image accordingly. To simulate the image of a real thermal camera, a Gaussian blur is applied to the image. The sensor output from the colour, depth, and thermal cameras can be seen in Figure 2.

4 Integration with ROS

As discussed above, ROS has been gaining steadily in popularity amongst the research community, because of its open source, free, and multi platform nature. For this reason it is important to be able to integrate jmeSim with ROS. This is achieved through the recently developed rosjava framework [Kohler, 2012], a pure Java implementation of ROS. A system built using ROS is composed of a number of independent processes, connected in a peer-to-peer topology, which communicate via publish-subscribe message passing. Each process is referred to as a ROS node. To integrate with ROS, jmeSim runs a rosjava node which can act as an interface between the Java simulation and other ROS processes in the system. The rosjava node runs in a separate thread, and allows messages to pass back and forth from the simulation to a ROS system. A schematic of the connectivity of ROS and jmeSim is shown in Figure 3.

4.1 Demonstration 1: Simultaneous Localisation and Mapping

One of the advantages of integration with ROS is that it provides access to a large variety of user contributed algorithms. As a demonstration of the integration of

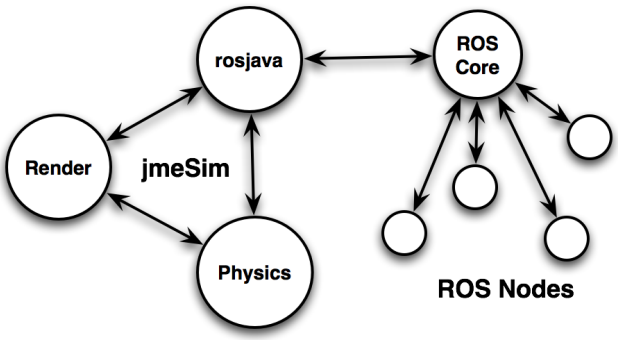

Figure 3: ROS-jmeSim connectivity. jmeSim runs as three threads, one for solving physics calculations, one performs rendering, and a third runs a rosjava node that publishes sensor data, and subscribes to control data.

jmeSim, we consider a common task in mobile robotics, simultaneous localisation and mapping (SLAM). For this task, a robot must interpret the data from its sensors, generally a laser range finder, to produce a map of an unknown environment within which it must simultaneously localise itself. The environment to be mapped in this instance is a typical open plan office, shown in Figure 4(c).

For comparison, a real robot equipped with a SICK laser, and running a ROS system, was driven by tele operation around the physical office. Within the ROS system, two nodes run separate algorithms to solve the SLAM problem. Firstly, a metric based iterative closest point (MBICP) position tracker matches successive laser scans to determine the position and orientation of the robot as it moves. This position and orientation data, known as the robot pose, is passed via ROS to a second node, running an implementation of the FastSLAM algorithm [Milstein *et al.*, 2011]. The FastSLAM algorithm generates a map of the environment. The map produced is shown in Figure 4(a).

An environment with the same floor plan is then produced within the jmeSim environment by drawing a vector graphics file by hand. A simulated robot is created in the simulation, equipped, just like the physical version, with a laser range finder. The laser data is sent via the rosjava node described above to the same ROS nodes running MBICP and FastSLAM, and the system produces a map of the simulated environment, which is shown in Figure 4(b). The similarity of the two maps qualitatively demonstrates the fidelity of the simulated laser sensor, and more importantly, demonstrates the utility of connectivity to ROS.

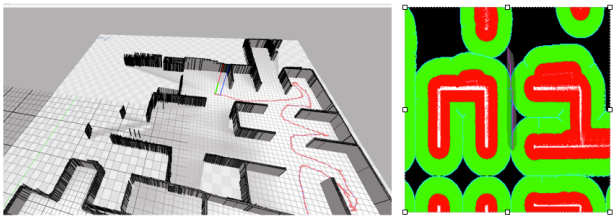

Figure 5: Autonomous navigation in jmeSim. Left: the path of a robot as it autonomously navigates through a jmeSim maze environment. Right: The output of path planning software aiding the robots navigation. Both algorithms are running in ROS.

4.2 Demonstration 2: Autonomous navigation

Demonstration 1 demonstrated communication only in the direction from jmeSim to ROS, in the form of laser range data. In our second demonstration we consider an example in which messages are passed in both directions. In this setting, we consider another common task in mobile robotics, in which a robot must autonomously navigate through a maze like environment, such as that shown in Figure 1.

Again, a ROS system running several autonomy algorithms receives laser data from jmeSim, except this time messages from ROS are return to jmeSim, in the form of linear and angular velocities, to control the motors on the simulated robot. In this example, the navigation algorithms used are the software from the RoboCup rescue team CASualty, with whom we share a lab. The algorithm combines a reactive controller which performs a wall follow, with an A* path planner that can send the robot to new terrain when it becomes stuck. The algorithms are described in detail in [Sheh *et al.*, 2009]. Figure 5 shows the robot's path as it autonomously navigatesg through the maze, along with the route chosen by the A* planning algorithms running in ROS.

These types of demonstrations show the tight integration between jmeSim and ROS. Of particular note in these examples is that the ROS code that produced these results is the same code which runs to control the real robot, and interpret its sensor data. The incoming laser data is indistinguishable to ROS from real data, and the motor commands it sends to jmeSim to drive the simulated robot are the same that it sends to the real autonomous robot. This is invaluable for testing and development of perception and navigation algorithms, so that they can be transferred from the simulation to the robotic application with no or minimal changes.

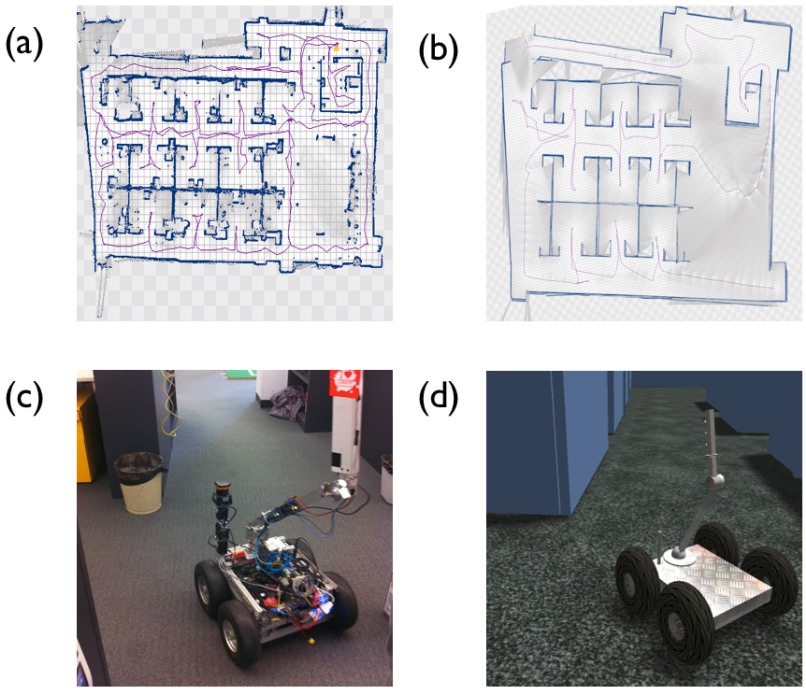

Figure 4: Real and simulated SLAM of an office environment. Clockwise from top left: The SLAM map generated by a real robot running FastSLAM in ROS, the SLAM map generated by the same code using the laser data from jmeSim, the simulated and real emu robots during SLAM runs.

5 Conclusion

This paper presents the jmeSim environment, an open source, multi-platform robotics simulation platform. The simulation provides high fidelity graphics and physics, batch environment generation, and a rich sensor suite. An implementation of the simulation demonstrating the integration with ROS has been demonstrated. This close integration can significantly reduce the work required for the testing and development of robotic software. Along the metrics we have described in this paper, this simulation package represents a significant improvement over currently available simulation frameworks.

Acknowledgements

This work was partially supported by the Australian Research Council Centre of Excellence for Autonomous Systems.

References

[Balakirsky and Kootbally, 2012] S Balakirsky and Zeid Kootbally. USARSim/ROS: A Combined Framework for Robotic Control and Simulation. *Proceedings of the ASME 2012 International Symposium On Flexible Automation*, 2012.

[Bastiaan, 2010] C Bastiaan. Virtual victims in USARSim. Master's thesis, Universiteit van Amsterdam, 2010.

[Boeing and Bräunl, 2007] A Boeing and T Bräunl. Evaluation of Real-Time Physics Simulation Systems. *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, 2007.

[Coumans, 2012] Erwin Coumans. Bullet Physics Homepage. http://bulletphysics.org/, 2012.

[Cousins, 2010] Steve Cousins. Welcome to ROS Topics. *IEEE Robotics & Automation Magazine*, 17(1):13–14, 2010.

[Gerkey *et al.*, 2003] B Gerkey, R T Vaughan, and A Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. *Proceedings of the 11th International Conference on Advanced Robotics*, 2003.

[Kamei *et al.*, 2012] K Kamei, S Nishio, N Hagita, and M Sato. Cloud Networked Robotics. *Network*, 2012.

[Koenig and Howard, 2004] N Koenig and A Howard. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[Kohler, 2012] Damon Kohler. rosjava: An Implementation of ROS in Pure Java with Android Support. http://code.google.com/p/rosjava/, 2012.

[Michel, 1998] O Michel. Webots: Symbiosis between Virtual and Real Mobile Robots. *Virtual Worlds*, pages 254–263, 1998.

[Milstein *et al.*, 2011] Adam Milstein, Matthew McGill, Timothy Wiley, Rudino Salleh, and Claude Sammut. A Method for Fast Encoder-Free Mapping in Unstructured Environments. *Journal of Field Robotics*, 28(6):817–831, 2011.

[Quigley *et al.*, 2009]
M Quigley, K Conley, and B Gerkey. ROS: an Open-Source Robot Operating System. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) Workshop on Open Source Software*, 2009.

[Rusu and Cousins, 2011] R Rusu and S Cousins. 3D is here: Point Cloud Library (PCL). *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[Sheh *et al.*, 2009] R Sheh, A Milstein, M McGill, R Salleh, and B Hengst. Semi-Autonomous Robots for RoboCup Rescue. *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*, 2009.

[Sheh, 2010] Raymond Ka-Man Sheh. *Learning Robot Behaviours by Observing and Envisaging*. PhD thesis, University of New South Wales, November 2010.

[Smith, 2012] R Smith. Open Dynamics Engine (ODE). http://www.ode.org, 2012.

[Vainer, 2012] Kirill Vainer. jMonkeyEngine 3.0 — Java OpenGL Game Engine. http://jmonkeyengine.com/, 2012.

[Wiener, 2011] Elliot J. Wiener. Maze Generators. http://www.ccs.neu.edu/home/snuffy/maze/, 2011.