PyRovio: Python API for WowWee Rovio

Jonathan Bona, Michael Prentice
Department of Computer Science and Engineering
201 Bell Hall
University at Buffalo, The State University of New York
Buffalo, NY 14260-2000
jpbona@buffalo.edu, mjp44@buffalo.edu

May 8, 2009

Abstract

PyRovio is our Python implementation of the WowWee Rovio API. It allows direct control of a Rovio robot from Python programs. We have used PyRovio to implement Python-based actor-agents that participate in a live intermedia performance. We also use PyRovio as part of ongoing work in cognitive robotics to connect SNePS-based agent minds to their Rovio bodies in the world.

1 Introduction

The WowWee Rovio is commercially available telepresence robot. Its features include wheeled locomotion, a webcam, microphone and speaker, infrared object detection, 802.11g wireless, and more. The Rovio firmware includes an on-board web server that can be used to programmatically control the robot by any software that implements the API.

This paper describes our work on PyRovio, a Python implementation of the WowWee Rovio API. PyRovio makes it possible to control a Rovio robot directly from Python.

PyRovio has been used to control Rovio actor-agents participating in an intermedia theatrical performance.

We are also using PyRovio as part of ongoing cognitive robotics work. In this context it is used to connect cognitive agents' minds (implemented in the SNePS knowledge representation, reasoning, and acting system) to the Rovio body, which the mind then controls by sending commands (for navigation, etc) and receiving percepts. PyRovio plays the role of a perceptuo-motor layer in the MGLAIR architecture, implementing low-level (hardware) functionality in a way that is accessible to higher reasoning.

2 MGLAIR

MGLAIR (Modal Grounded Layered Architecture for Integrated Reasoning) [Hexmoor et al., 1993] [Shapiro et al., 2005] is an architecture for cognitive agents that divides agent design into a series of layers (Knowledge, Perceptuo-Motor, and Sensori-Actuator), each of which can be implemented independently of the other layers' implementation.

MGLAIR also separates different perceptual and efferent modalities into their own channels, which allows an agent to attend to different types of percepts, navigation tasks, etc based on that agent's own priorities and current goals. For instance, a robot tasked with visually locating a colorful ball while navigating in some environment should probably pay more attention to percepts in the *cliff detection* modality than even those *visual* percepts directly related to its primary goal (finding the ball).

By breaking agent design into layers, MGLAIR allows an agent's mind to be initially developed in the Knowledge Layer (KL) independently of its embodiment, so long as the KL has some basic information about the body's perceptual and efferent capabilities. It also makes it possible to create portable cognitive agents that can easily be moved from controlling a robot in the real world to controlling a simulated robot in a simulated world – and vice versa.

Figure 1 illustrates the MGLAIR architecture. The perceptuo-motor layer can be further divided into three distinct sublayers. PML sublayer a implements the Knowledge Layer's primitive actions independently of the agent's actual embodiment. PML sublayer b implements functions in the agent's PMLa for a particular embodiment. PML sublayer c makes sensori-actuator functionality available to higher-level layers in the architecture and is accessed by PMLb to pass messages and commands down to, and up from, the Sensori-Actuator Layer. The SAL (Sensori-Actuator Layer) directly controls the agent's body, including navigation and sensors. That is, it has functions to move in a particular direction, produce an utterance, and so on. The SAL depends heavily on the specific embodiment and the environment in which the agent is operating.

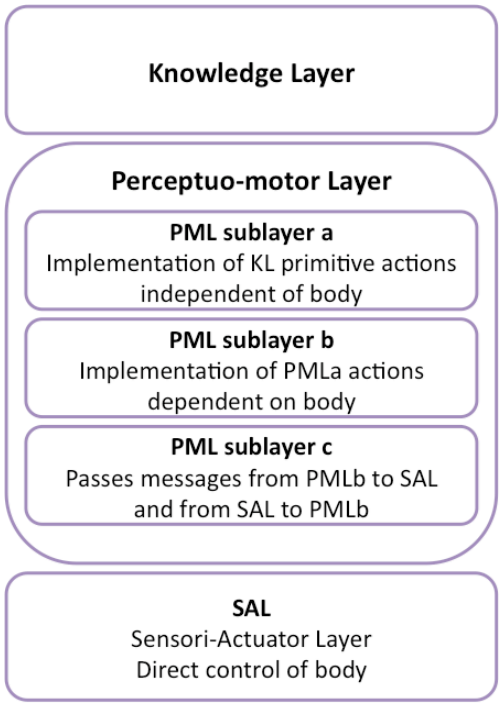

Figure 1: MGLAIR

In our MGLAIR-based SNePS/Rovio agents part of the Perceptuo-motor Layer is implemented in Python and uses PyRovio to connect to the SAL. This work is described in more detail in the sections that follow.

3 PyRovio Implementation

The Rovio is controlled using HTTP commands to a web server hosted on the robot. The robot itself connects to a network using the wireless 802.11g protocol. This allows for controlling the robot and accessing its sensory data from any client with an internet connection and in any language. The authors' code for controlling the Rovio robot is implemented in a Python module. The programmatic interface consists of two main parts: the Rovio class and the RovioController class.

The Rovio class wraps HTTP calls to a Rovio and returns the appropriate responses. It also provides some additional Python-specific support. The RovioController class provides an instruction queue for running

multiple commands over a period of time.

The code's purpose and design choices are introduced, and some example usage is provided.

3.1 Rovio Class

The Rovio class provides a low-level Python interface to the Rovio. Some state information is kept and some pre- and post-processing of Rovio input and output takes place to return structures appropriate to Python; however, this class otherwise mirrors the Rovio's web-based API [WowWee, 2008]. It provides the lowest-level perceptuo-motor interface to the sensori-actuator layer hosted on the Rovio.

An instance of the Rovio class provides an interface to one Rovio. This class wraps the HTTP requests with Python method calls. The Rovio API is mirrored as faithfully as possible; however, some convenience functions have been used. For example, the movement commands are implemented as separate methods rather than parameters to the ManualDrive method in the Rovio API.

A Rovio is simply a wireless node acting as a webserver. To connect to it, set the hostname of the Rovio to connect to using the host property. Set the IP address or host of the Rovio webcam itself using the Rovio API using setHTTP (this functionality has not yet been implemented in the PyRovio code). After using SetHTTP, the user is required to then set the host property to the same address in order to continue controlling the same Rovio object.[1]

The following Python exceptions are provided: ConnectError, ResponseError, ParamError, and OutOfRangeError. They inherit from the base class RovioError. A ConnectError is raised when the specified Rovio webserver cannot be found or connected to. The ResponseError is designed to be used at a higher level to indicate a failure state returned by the API, but is unused.[2] ParamError and OutOfRangeError are used to indicate that an attribute or input has been improperly set. Parameter checking in the Python code is limited and relies upon the internal, firmware version-dependent parameter checks implemented on the Rovio in order to better provide a future-proof interface.[3] The advantage of lazy evaluation of server connection and parameter checking lies in both future-proofing the code, and providing a flexible interface for programmers. The disadvantage to relying on the Rovio connection is a certain lack of feedback provided by the Python code.

The movement commands provide for directional and camera movement and are implemented using the Rovio API's ManualDrive function (as distinct from navigation system path-based movement). Sensory information is provided by get_report, get_MCU_report and get_status to provide sensory information on the motor movements, navigation information, battery levels, and so forth. Camera data is provided through get_image. A continuously-updating motion JPEG (MJPEG) is provided by the Rovio robot but not yet supported in PyRovio. Real-time streaming of video and audio is supported by WowWee on Windows machines only at this point, and is not yet supported in the PyRovio code.

The Rovio has a built-in navigation system that much of the built-in API relies on; however, the authors' experience with the NorthStar^{TM}navigation system has been at-best unreliable. In the proper environment, such a path-based navigation system could be quite helpful in triggering needed actions at a low level; however, a vision and reasoning based navigation system implemented on an external client would provide a more flexible platform and is an area of future work that would greatly benefit users of the Rovio as a cognitive robotics platform.

3.2 RovioController Class

The RovioController class provides an event queue for stringing multiple actions together. Once the dispatch thread is started, it continuously checks its event queue and dispatches Rovio actions at a user-defined

[1] This was an arbitrary design decision in making the Rovio class.

[2] The higher-level interface raising ResponseError is a priority for future work.

[3] Versions 3 and 5 of the Rovio firmware, the two main versions in use by current users, contain many differences. The Python interface functions with both; however, several firmware-dependent functions will return different values. This is indicated in the Python documentation strings for each function.

interval.[4]

Events are added to the queue via enqueue, which accepts a duration (in milliseconds), a command, and parameters for the command. Commands can be chained together ahead of time and added with enqueue_all. The queue can be cleared with clear. To clear the queue and add a new command—for example, to

In the future, the RovioController class will provide more higher-level functionality and more descriptive feedback from the low-level API Python implementation.

3.3 Using the PyRovio Interface

The PyRovio implementation is available for current use.[5] The following is a sample run to illustrate how the PyRovio code is used from within an interactive Python shell.[6]

```
In [1]: import rovio

In [2]: v = rovio.Rovio('vanessa', '192.168.10.18')

In [3]: vc = rovio.RovioController(v)

In [4]: v.forward()
Out[4]: 0

In [5]: v.head_middle()
Out[5]: 0

In [6]: v.get_status()
Out[6]: {'Cmd': 'nav', 'raw_state': 0, 'responses': 0, 'state': 'idle'}

In [7]: v.get_report()
Out[7]:
{'Cmd': 'nav',
 ...
 'x': 7133,
 'y': -1747}

In [8]: v.get_MCU_report()
Out[8]: '0E0100000000000000000003897400'

In [9]: vc.start()

In [10]: vc.enqueue_all([[1000, v.forward], [1, v.rotate_left, [1, 3]],
                         [1000, v.backward], [1, v.head_down]])

In [11]: v.get_image()
Out[11]: '\xff\xd8\xff\xdb\x00C\x00\x13\r\x10\x1d\x10\r\x10\x13\x1a ...'
```

Lines 1–3 illustrate setting up the Python objects used for controlling the Rovio. Lines 4 & 5 demonstrate movement and camera commands. A return value of 0 (defined in the constant rovio.SUCCESS) indicates

[4]The wait interval is provided to avoid overwhelming the Rovio webserver with HTTP connections.

[5]The most recent PyRovio codebase is available via git or for independent download at `http://github.com/mprentice/pyrovio`. For running on the UB graduate server, it is available on Nickelback in /projects/mjp44/pyrovio/src. This may be moved to /projects/mjp44/pyrovio in the future.

[6]Using iPython for Python 2.6.

a successful command. Lines 6–8 show how to get sensory information from the Rovio.[7] To run the event queue, the RovioController thread must be started as in line 9. Line 10 shows how to queue multiple commands for execution (1000 = 1 second). The get_image command returns a JPEG-compressed image from the camera.

4 WoyUbu Performance

We have used PyRovio to implement simple Python-based actor-agents as part of WoyUbu, which is a collaborative intermedia performance presented in March 2009 by the University at Buffalo Intermedia Performance Studio and collaborators. WoyUbu is a "mashup" of Georg Buchner's 'Woyzeck' and Alfred Jarry's 'Ubu Roi'. The performance integrates live human actors, film, computer animation and video games, puppets, robots, and more. In one scene a group of Rovios battles an opposing group of iRobots. The battle consists primarily of a sequence of choreographed advances and retreats and other motions. For instance, when the Rovios are "attacking", they randomly move in small circles and raise and lower their webcams.

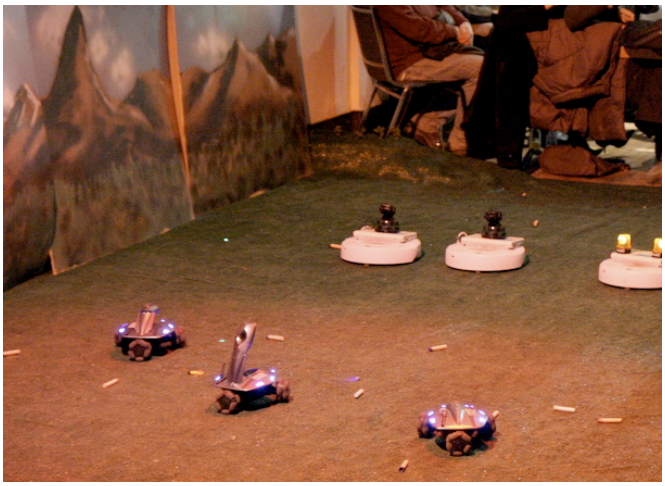

Figure 2: WoyUbu – Rovios battle iRobot Creates. Photo by Dave Pape.

The timing of the robots' actions in the battle depends on cues in the performance (lines delivered by live human actors), and the timing varies from performance to performance. In an ideal scenario, the robots' bodies would be controlled by and providing percepts to cognitive agents implemented in a system like SNePS, and the agents would have full vision and speech recognition software that allows them to interact with each other and with the human actors. Given the current state of voice recognition technology and machine vision, this was not a practical way of implementing our robotic actor-agents for this performance. Instead, the robots depend on a human stage manager to recognize cues and notify the robots when a cue has been delivered. Each robot has a set of "scripts" that dictate its response to each cue. These scripts are written in English fragments that the robots can process, and contain commands like "move forward for 5 seconds", and "turn 180 degrees".

The stage manager communicates to the robots that a particular cue has been delivered by pressing a button on a Java GUI that has a button for each cue. This software is connected to the robots over a wireless network using an MGLAIR-compatible framework that simulates each agents' PMLb. The (PyRovio-based) PMLc receives messages from the stage management software in exactly the same way as it would receive messages that originated in a cognitive agents' Knowledge Layer.

[7] get_MCU_report is firmware-dependent and not well-documented. The return value shown is from firmware version 5.

5 SNePS Rovio

We are using PyRovio in cognitive robotics work that combines cognitive agents' minds implemented in SNePS with the Rovio robotic hardware. Each SNePS/Rovio agent's mind is connected to its body using MGLAIR, and has the ability to receive and reason about percepts that originate in the robot's sensors, and to control the robot's navigation and other capabilities.

5.1 Architecture

Our SNePS/Rovio MGLAIR agents' Knowledge Layers are implemented in SNePS. PMLa and PMLb are implemented as Lisp functions that run in the same image as SNePS and are accessed by it. The Lisp PMLb includes functions that interact with the PMLc. The Python PMLc and the Lisp PMLb interact via TCP/IP sockets, with a separate socket for each modality (*vision*, *navigation*, etc). These are often connected using an external Java-based server that manages connecting all the sockets for various modalities. The details of that connecting server are not discussed here. The PMLc is implemented in Python and uses PyRovio to interact with the SAL (Rovio firmware) via HTTP requests as per the Rovio API.

The composition of the MGLAIR layers as used by our SNePS/Rovio agents is illustrated in **Figure 3**.

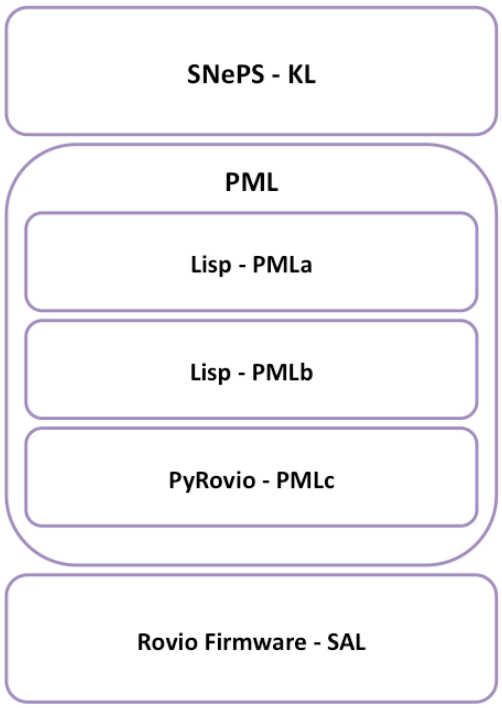

Figure 3: SNePS/Rovio MGLAIR

5.2 Obstacle Detection

Work on SNePS/Rovio agents is still ongoing, and some of the robot's capabilities are currently unused or incompletely used by agents. Here we discuss infrared obstacle detection as an example of how changes in the environment are perceived and become represented by propositions that are asserted in the Knowledge Layer.

The PMLc uses PyRovio to continually poll the robot for sensor data. When a significant change is detected (e.g. an obstacle is detected when none was previously detected), PMLc sends a message reporting

the state of the relevant sensor to the PMLb, which passes the message to the PMLa, which adds the percept to a percept queue for the relevant modality. The agent's KL regularly checks the queues for new percepts, and its beliefs are updated to reflect the new information about the world that the percepts represent.

For instance, when the robot's infrared sensor detects that an obstacle has appeared in front of it the PML will "notice" within milliseconds, and pass that information up to the KL, where it it is perceived and becomes asserted as the proposition `Blocked()`. When an obstacle disappears from in front of the robot, that information is also sent up through the MGLAIR layers and results in the proposition `Blocked()` being disbelieved and its negation, ∼`Blocked()` being asserted.

5.3 Navigation Acts

PyRovio implements Rovio primitive actions like turning some number of degrees, moving forward/ backward/ left/ right some number of units or for some number of seconds, and so on. We have made these capabilities available to the SNePS acting system by attaching SNePS primitive acts for navigation. These simple acts can be invoked directly by the user or as part of more complicated actions an agent might take when reasoning about and interacting with its environment. For instance, the act `navigate(Turn_90)`, when performed by the agent, will cause the robot to turn 90 degrees clockwise.

5.4 SNePS Obstacle Detection Agent

We have created a very simple proof-of-concept SNePS/Rovio agent with the following behavior: When the robot has an obstacle in front of it and that obstacle is removed, the agent moves forward two units.

The main SNeRE policy that encodes this behavior is the following:

```
;;; Whenever I stop being blocked by an obstacle, move forward two units.
wheneverdo(~Blocked(), navigate(Forward_2)).
```

6 Future Work

6.1 PyRovio

Currently PyRovio provides an easy way of controlling the Rovio and accessing many of its features from Python. Some of the more advanced functions in the Rovio API have not yet been implemented in PyRovio and we are working to create and publish a stable, full-featured, and well-documented public release.

6.2 Agents

The only SNePS/Rovio agents we have completed so far have been very simple proofs of concept. Now that we have successfully used SNePS and the SNeRE acting system to access the Rovio's sensors, reason about the state of the world, and act in the world, we are ready to begin developing more complicated and interesting agents.

WoyUbu will be performed at one or more festivals in the coming months and the portions of the performance involving robots would be improved if the robots were controlled by cognitive agents implemented in a system like SNePS, which can explicitly represent goals, plans, and complex actions. We plan to develop SNePS agents for WoyUbu to replace the simplistic Python code that currently controls the robotic bodies in the performance.

6.3 Vision

The Rovio's on-board webcam makes it an ideal platform for robotics vision projects. PyRovio includes basic functions to retrieve images from the camera, but we have not yet implemented cognitive agents that process and use this data. Our agents, including those involved in WoyUbu, would perform better at basic tasks such as navigation if they had access to the visual information captured by their cameras.

7 Conclusions

We have developed a Python implementation of the Rovio API that can be used by stand-alone Python programs to directly control the Rovio. PyRovio has already proven itself useful in supporting the work of Buffalo's Intermedia Performance Studio. We expect it to also be of use to the Rovio developer community when a final release is publicly distributed. Finally, it also can be and has been used as part of the Perceptuo-Motor Layer of SNePS MGLAIR agents, a simple example of which is provided here. PyRovio provides a starting point for the development of more complex cognitive robotics agents.

References

[Hexmoor et al., 1993] Hexmoor, H., Lammens, J., and Shapiro, S. (1993). *Embodiment in GLAIR: A Grounded Layered Architecture with Integrated Reasoning for Autonomous Agents*. State University of New York at Buffalo, Dept. of Computer Science.

[Shapiro et al., 2005] Shapiro, S., Anstey, J., Pape, D., Nayak, T., Kandefer, M., and Telhan, O. (2005). MGLAIR agents in virtual and other graphical environments. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 20, page 1704. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

[WowWee, 2008] WowWee (2008). Api specification for rovio, version 1.2. Technical report, WowWee Group Limited.