Proceeding of the IEEE
International Conference on Information and Automation
Hailar, China, July 2014

From ROS to Unity: leveraging robot and virtual environment middleware for immersive teleoperation*

R. Codd-Downey, P. Mojiri Forooshani, A. Speers, H. Wang and M. Jenkin
York Centre for Field Robotics and Lassonde School of Engineering
York University, Toronto, Ontario, M3J 1P3, Canada
{*robert, pmojirif, huiwang, speers, jenkin*}*@cse.yorku.ca*

***Abstract*— Virtual reality systems are often proposed as an appropriate technology for the development of teleoperational interfaces for autonomous and semi-autonomous systems. In the past such systems have typically been developed as "one off" experimental systems in part due to a lack of common software systems for both robot software development and virtual environment infrastructure. More recently, common frameworks have begun to emerge for both robot control (e.g., ROS) and virtual environment display and interaction (e.g., Unity). Here we consider the task of developing systems that integrate these two environments. A yaml-based communications protocol over web sockets is used to glue the two software environments together. This allows each system to be controlled using standard software toolkits independently while providing a flexible interface between these two infrastructures.**

***Index Terms*— robotics, teleoperation, virtual reality.**

I. Introduction

There has long been an interest in developing virtual environment-based interface technologies for the control and supervision of autonomous and semi-autonomous devices. The basic concept can be traced back to early work in telexistence (see [1]): By providing an operator with a perceptual environment that is consistent with the experience that they would have experienced had they been co-located with the remote device then operator performance would be improved. Virtual reality would seem an appropriate technology to provide these essential perceptual cues. Given the potential such an interface might provide, a number of different experimental and operational systems have been developed. For example, systems have been developed for space operations (e.g., [2]), off-road vehicles (e.g,, [3] and unmanned aerovehicles (e.g., [4]), to name but a few. Although these and other systems have had their successes, one aspect that has limited the adoption more generally of the concept has been the lack of a common and easily accessible software infrastructure for both robot control and virtual reality systems. Fortunately, the last few years have seen the development and adoption of a number of standard software systems for both robot control and virtual reality infrastructure. Here we explore how these advances can be exploited in the development of a virtual reality-based teleoperational interface for autonomous robots.

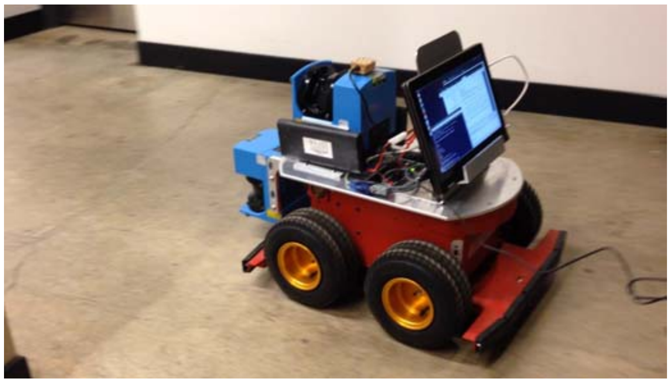
*The financial support of the NSERC Canadian Network in Field Robotics is gratefully acknowledged.

A. *Autonomous robot middleware*

There have been a number of attempts to develop a standard robot middleware. Early efforts at developing such software infrastructures for robot control include Ayllu [5], Player [6] and COLBERT/Saphira [7]. Although these efforts advanced our understanding of the requirements for autonomous robot middleware, for a number of reasons these systems did not find wide-spread adoption in the academic and industrial communities. Different systems had different detractions, but often systems were targeted at specific hardware platforms/sensor platforms, or had limited computing hardware/operating system or language support. Over the last decade the ROS (Robot Operating System) [8] has emerged as a standard software middleware for the development of autonomous systems. Within ROS, overall robot control is modelled as a collection of asynchronous processes (known as nodes) that communicate via message passing. Although ROS has been ported to a number of different hardware platforms and bindings exist for a number of different languages, support is primarily targeted towards Ubuntu, with software libraries targeted at C++ and Python. A very limited level of support exists for lower performance devices (e.g., Android platforms) but the limited memory footprint on such devices makes development and deployment more complex.

There are a number of reasons why ROS has emerged as a standard software middleware for robot software. First, the software makes few assumptions about hardware or networking – although it does rely on reliable TCP/IP communication. Second, the software model of independent communicating agents allows for great modularity, including the integration of software across multiple hardware sites – including remote sites. Third, there is support for a large number of robots and sensors. Finally, there exists a large number of software tools and libraries for common robot tasks.

Figure 1 illustrates a ROS-based robot system. Figure 1(a) shows a robot based on the P3-AT platform developed by Mobile Robots. The robot relies on skid-steering, and is controlled by a PC104 computer mounted inside the robot proper. The robot exposes the sensors/locomotion system in terms of a small number of ROS nodes. A standard laptop mounted on top of the robot runs Ubuntu, and hosts two laser scanners one mounted in the horizontal plane, the other in the vertical, and communicates with the ROS environment


(a) A laser-equipped robot

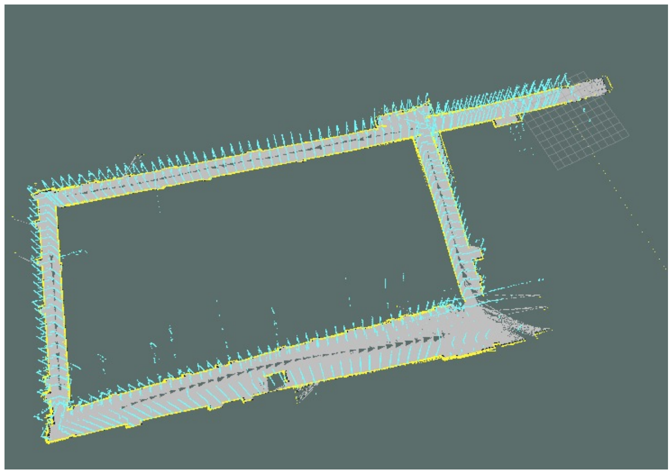

(b) A typical laser sensor-based map

Fig. 1. A typical indoor autonomous robot and the map that can be constructed using it. (a) The robot is equipped with two laser scanners that enables the robot to reconstruct both the wall structure as well as the vertical structure of the environment (b).

within the robot. The entire ROS infrastructure is visible to an external operator through wifi.

Figure 1(b) shows an environmental map obtained with the robot operating indoors at York University. This map relies on the horizontal plane laser and odometry information to solve the SLAM (simultaneous localization and mapping) problem, and then uses the solution to SLAM to integrate the horizontal and vertical laser data into a single point cloud representation of the environment.

ROS provides a number of visualization tools – including the one used to generate Figure 1(b). Although these tools are quite powerful they were not designed to support sophisticated virtual reality infrastructure (e.g., HMD's, head trackers, and the like).

B. *Virtual reality middleware*

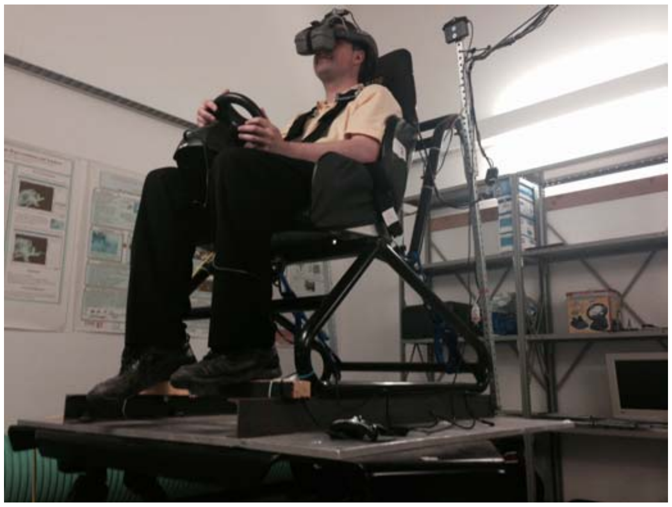
Just as robot middleware has matured over the past decade, there has been a similar evolution in the development of software systems to support virtual reality installations. Early efforts to develop VR-based infrastructure were hampered by specialized hardware and the lack of standardization of sophisticated trackers, rendering systems, physics engines and the like. Early VR systems such as VE [9] and FLow VR [10], were typically tied to specific hardware systems and rendering libraries (e.g., SGI hardware, OpenGL). Support for trackers and other input devices was limited, and considerable effort was required to support different VR display technologies and rendering structures.

Figure 2 illustrates the challenges involved in developing an effective VR standard software middleware. Figure 2(a) shows a HMD-based VR system that utilizes a motion base to provide physical motion cues coupled with stereo video streams. Although tools such as ROS's rviz can certainly provide visual simulations of the robot's environment, rviz lacks the infrastructure necessary to easily permit integration of head trackers, motion bases and the like. Figure 2(b) illustrates another potential interaction technology, an immersive visual display. In IVY [9], each of the external surfaces of the display is rear projected using stereo video streams. Head position is tracked in order to update the visual displays to simulate different viewing positions/directions. Although IVY has used a range of different software infrastructures to provide a coherent visual display, it is critical to observe that each of the external video signals must be generated with controlled viewpoints and these signals must be frame locked. Again, tools such as rviz are not designed with these types of constrains in mind.

A number of different software middlewares/libraries now exist that support the generation of complex visual scenes coupled with interfaces to multiple screens/projectors, trackers, physics engines, audio generation, user interface devices and physical motion bases, and the like. Many of these systems integrate "game engines" such as Ogre [11] and Unity [12] with external libraries and other tools. Typically such game engines provide some scripting tool or other mechanism to enable developers to extend the basic infrastructure available with the game engine itself. A number of external system, e.g., MiddleVR [13] exist that provide support for specialized VR hardware and also provide appropriate links to a game engine for content management.

Although a number of different game engines/external library systems exist, Unity offers a number of advantages for the development of an immersive teleoperational system. First, it has support for a wide range of different platforms. Second, it provides a number of different scripting options. Finally, it provides well supported libraries for access to external VR hardware through systems such as MiddleVR.

In Unity, the basic software module is the `GameObject`. A GameObject provides an encapsulation of the visual appearance of an object and hooks to the physics engine, along with logic for interaction, animation and the like. Unity – like many game engines – operates in a frame-driven manner, where


(a) Motion-based VR

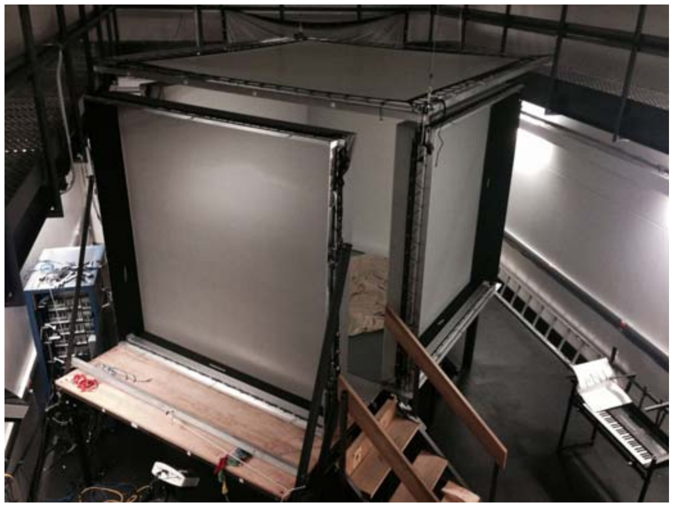

(b) Immersive projective VR

Fig. 2. Extremes in virtual reality systems. (a) shows a HMD-based VE system that uses a 6 DOF motion base to provide physical motion cues. (b) shows a six-sided immersive projective environment. Each of the walls, floor and ceiling can be rear projected using stereo video streams.

the need to visually refresh the visual display is critical and drives the basic software infrastructure. Although this model is appropriate for game engine systems and virtual reality hardware, this model does not mesh well with the software infrastructure found in robot control systems such as ROS.

C. Summary

Although there exists middleware for robots and virtual reality systems that are well targeted for their respective tasks, the two software environments are not easily integrated. ROS, the de facto standard for robot software (at least in the research domain) is based on a message passing paradigm. The typical game software infrastructure (as typified by Unity) is tightly tied to the rendering loop and uses game objects as the basic primitive. Integrating these two environments requires dealing with the following realities

- ROS messages must be mapped to events that can be processed as part of the basic rendering loop in the visualization system.
- Mapping user intent as presented to the virtual reality system must be mapped into the appropriate ROS message(s) within the robot software environment.
- It is critical to deal with the liveliness requirement of the rendering environment. Typically, the process of servicing the rendering loop must be completed sufficiently quickly – often in less than 1/30'th of a second – in order to retain the liveliness of the virtual reality interface.

II. Linking ROS and Unity

Here we describe the process we followed to integrate a Unity-based virtual reality teleoperational interface with a ROS-based ground contact robot (specifically the robot shown in Figure 1). As described above, this robot utilizes ROS as its operational middleware with all computing and sensing performed on board using a collection of computers running Linux. Tests to date have concentrated on operation on a flat ground plane, but when fully implemented, it is anticipated that the robot will operate over uneven terrain and that the vehicle roll and pitch will be communicated to the operator through physical roll and pitch induced by the motion platform shown in Figure 2(a). Preliminary testing utilizes less sophisticated virtual reality hardware – from the immersive projective environment shown in Figure 2(b), through tracked HMD's to simple laptop-based displays.

Utilizing a software infrastructure that can deploy to these and other hardware infrastructures is critical. Although a number of different visualization/virtual reality toolkits would meet this requirement, here we have chosen to use Unity. Unity provides the flexibility of deploying/developing the software on a range of different platforms and libraries exist to provide control/interaction of the various virtual reality interaction suits available to us.

Operationally, this teleoperated robot runs in two phases: a mapping phase and a teleoperational phase. During the mapping phase a map of the robot's environment is acquired. The process of accomplishing this is beyond the scope of this manuscript, but it could be manual – using a map acquired by hand – or it could be automatic – using some standard SLAM algorithm. The map shown in Figure 1(b) was obtained using the later technology, but the process is not critical to the work presented here. This map could take many different forms, but here we assume a 3D point cloud representation in some global coordinate frame. During the teleoperational stage the robot is driven by the user through a virtual reality-based teleoperational interface written in Unity. Both the ROS and Unity worlds have access to the pre-computed map which is augmented with real-time telemetry from the vehicle, local point clouds obtained by the vehicle as it moves, and onboard tilt/pitch sensors which will be used to drive the orientation of the motion base relative to gravity. (See Figure 3.)

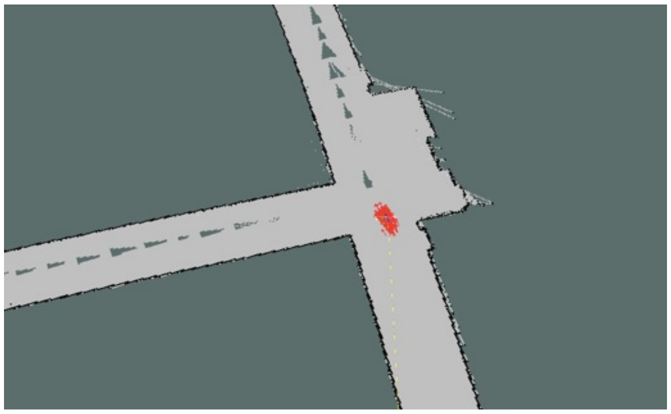


Fig. 3. Robot localization during the teloperational phase. During teleoperation the user utilizes the virtual reality interface to control the robot. The robot and the user have access to the map as well as an autonomous localization process running within the ROS environment.

A. Interfacing with the ROS message passing system

Within ROS, messages are passed using an internal protocol based on TCP/IP sockets. This is a sophisticated protocol designed to provide reliable transport between asynchronous nodes within ROS. Although it would be possible to interface with this protocol directly, much of the ROS infrastructure is of little interest to the teleoperational interface. Instead of dealing with the complexity of the ROS world an alternative is to expose the interesting portion of the ROS infrastructure through the `rosbridge` framework [14]. Rosbridge provides a mechanism within which messages generated within the ROS environment are exposed to an external agent and within which an external agent can inject messages into the ROS environment. This process utilizes the WebSocket [15] protocol as a communication layer which means that provided the external agent has network access to the ROS environment it can be physically located anywhere. A number of libraries exist that support the WebSocket protocol using the Unity scripting environment.

The rosbridge framework communicates ROS messages to and from the external world in the form of yaml (YAML Ain't Markup Language) [16] strings. Such yaml strings can be used directly by an external agent but perhaps the most convenient way is to use JSON [17] to map yaml strings to instances of objects within the unity scripting environment. There have been a number of successful efforts to use this particular approach to interface various devices with a ROS including interfacing lightweight computing devices such as Android phones and tablets [18].

The process of communicating between ROS and Unity involves transmitting yaml-encoded messages through a websockets. yaml messages are collections of keyword-value pairs. The protocol is quite straightforward although not without its concerns as we will see shortly. Messages are easily crafted. For example, to request that ROS transmit all clock messages from the ROS environment to the Unity environment, the Unity environment would simply transmit through the websocket protocol

```
{"op": "subscribe",
 "topic": "/clock",
 "type": "rosgraph_msgs/Clock"}.
```

Responses from the ROS environment will appear asynchronously through the websocket connection. Too inject (publish) a particular message to the ROS environment the Unity environment first advertises that it will be publishing messages of a particular type on a particular topic

```
{"op": "advertise",
 "topic": "/unity/joy",
 "type": "sensor_msgs/Joy"}
```

and then publishes values when desired using

```
{"op": "publish",
 "topic": "/unity/joy",
 "msg": msg}.
```

where msg is a yaml string encoding a sensor_msgs/Joy message. One issue that must be addressed in terms of servicing the message protocol is that the rendering process within Unity assumes that any script associated with rendering in the scene will complete quickly. Given the potential for delay in sending/receiving data from the ROS environment, this essentially requires an asynchronous process to deal with the remote ROS environment. In the current implementation this is accomplished through a separate Unity thread that handles the rosbridge message stream.

In the current implementation the static environmental map is not transmitted through the rosbridge interface, but rather is pre-stored as a binary file and loaded by a script within Unity. This provides substantive performance improvement over using the rosbridge interface. YAML is a seven bit clean protocol and although this has a number of advantages it does mean that large data structures – the map may contain over one hundred thousand points in the point cloud – can take substantive time to transmit. The use of a file stored on both the Unity and ROS file systems addresses this particular issue. The use of the relatively memory inefficient YAML messages during teleoperation is less of an issue as the actual amount of data being transmitted is relatively small.

During teleoperation the Unity environment displays the pre-loaded map (a point cloud) as a particle system. The operator uses a joystick or other input device to command the robot. Input messages are converted into corresponding ROS `sensor_msgs/Joy` messages and injected into the ROS control environment on the robot causing it to move. As the robot moves a pose estimation process running on the robot maintains an estimate of the robot's pose with respect to the global map that is common to both the ROS and Unity environments. Sensor data collected by the robot is converted into a point cloud in a global frame of reference in the ROS environment and this information is subscribed to by the Unity environment. This causes the information to be communicated by rosbridge automatically to the Unity environment where

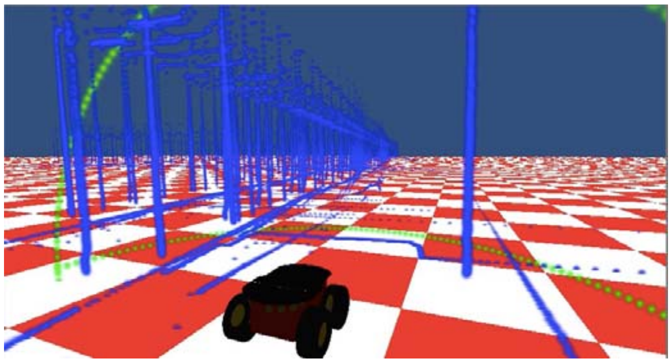


Fig. 4. ROS integrated with Unity. The Unity world showing the robot as well as the world point map displayed as a point cloud.

it is converted into a second point cloud and displayed along with the global map information. Errros in the pose estimation process can be seen as discrepancies between the mapped environment and returned sensor data. Should the user wish, they can provide hints to the localization process within the ROS environment. Such hints are captured as user interaction events within Unity, transmitted via rosbridge to the ROS environment, where they become local corrections to the pose estimation process. These localization hints can prove necessary if the autonomous localization system fails to maintain an accurate estimate of the robot's current pose.

III. Summary and future work

Building virtual reality-based teleoperational interfaces for autonomous systems involves bridging two very different software technologies. ROS-enabled robots utilize a software architecture based on message passing, while virtual reality systems are typically tied to a rendering loop. This paper describes a system and general approach to linking these two disparate technologies. Rather than developing a VR node within ROS, rosbridge and websockets are used to provide a mechanism within which the VR infrastructure can listen to and inject ROS messages into the ROS environment.

The use of websockets as the underlying communication protocol allows the ROS and VR environments to operate on completely separate hardware. It only requires the two systems to be coupled by a reliable network connection.

Experiments reported here were conducted indoors and using relatively simple display systems. Ongoing work is investigating deploying the robot outdoors and using a motion platform to cue vehicle roll/pitch orientation relative to gravity to an operator controlling the robot from the motion platform..

References

[1] S. Tachi, K. Tanie, K. Komoriya, and M. Kaneko, "Tele-existence (i): design and evaluation of a visual display with a sensation of presence," in *Proc. Fifth CISM-IFToMM Symposium*, 1984, pp. 206–215.

[2] E. Stoll, M. Wilde, and C. Pong, "Using virutal reality for human-assisted in-space robotic assembly," in *Proc. World Congress on Engineering and Computer Science*, 2009.

[3] M. A. Steffen, J. D. Will, and N. Murakami, "Use of virtual reality for teloperation of autonomous vehicles," in *American Society of Agricultural and Biological Engineers Biological Sensorics Conference*, 2007.

[4] B. E. Walter, J. S. Knutzon, A. V. Sannier, and J. H. Oliver, "Virtual uav ground control station," in *Proc. 3rd AIAA "Unmanned Unlimited" Technical Conference*, Chicago, IL, 2004.

[5] B. B. Werger, "Ayllu: distributed port-arbitrated behavior-based control," in *Distributed Autonomous Robotic Systems 4*, L. E. Parker, G. Bekey, and J. Barhen, Eds. Knoxville, TN: Springer-Verlag, 2000, pp. 24–34.

[6] B. P. Gerkey, R. T. Vaughan, K. Stoy, A. Howards, G. S. Sukhatme, and K. J. Mataric, "Most valuable player: a robot device server for distributed control," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, Wailea, HI, 2001, pp. 1226–1231.

[7] K. Konolige, "Colbert: A language for reactive control in saphira," in *PRoc. German Conf on Artificial Intelligence*, Freiburg, Germany, 1997, pp. 31–52.

[8] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *Proc. Open-Source Software workshop at the International Conference on Robotics and Automation (ICRA)*, 2009.

[9] M. Robinson, J. Laurence, J. Zacher, , A. Hogue, R. Allison, L. R. Harris, M. Jenkin, and W. Stuerzlinger, "Growing ivy: Builtind the immersive visual environment at york," in *Proc. Int. Conf. on Augmented Reality and Telexistence*, Tokyo, Japan, 2001.

[10] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert, "Flowvr: a middleware for large scale virtual reality applications," in *Proc. 10th Int. EuroPar Conference*, 2004, pp. 497–505.

[11] Ogre, "Ogre," http://www.ogre3d.com, 2014, accessed May 8, 2014.

[12] U. Technologies), "Unity," http://unity3d.com, 2014, accessed May 8, 2014.

[13] i'm in VR, "middlevr," http://www.www.imin-vr.com, 2014, accessed May 8, 2014.

[14] Brown University, "rosbridge," http://www.rosbridge.org, Feb. 2013, accessed February 10, 2013.

[15] I. Hickson, "The WebSocket API," W3C Working Draft 16 August 2010, W3C.

[16] S. Ben-Kilko, "Yaml ain't markup language," http://www.yaml.org, accessed March 12, 2013.

[17] D. Corckford, "The application/json media type for JavaScript object notation JSON," Network Working Group RFC 4627, 2006.

[18] A. Speers, P. Forooshani, M. Dicke, and M. Jenkin, "Lightweight tablet devices for command and control of ros-enabled devices," in *Proc. 2013 Int. Conf. on Advanced Robotics (ICAR)*, Montevideo, Uruguay, 2013.