Preprint of the paper which appears in:
European Conference on Mobile Robots (ECMR) 2021

Open-Source Tools for Efficient ROS and ROS2-based 2D Human-Robot Interface Development

Stefan Fabian and Oskar von Stryk

***Abstract*— 2D human-robot interfaces (HRI) are a key component of most robotic systems with an (optional) teleoperation component. However, creating such an interface is often cumbersome and time-consuming since most user interface frameworks require recompilation on each change or the writing of extensive boilerplate code even for simple interfaces. In this paper, we introduce five open-source packages, namely, the *ros(2)_babel_fish* packages, the *qml_ros(2)_plugin* packages, and the *hector_rviz_overlay* package. These packages enable the creation of visually appealing end-user or functionality-oriented diagnostic interfaces for ROS- and ROS2-based robots in a simple and quick fashion using the QtWidget or QML user interface framework. Optionally, rendering the interface as an overlay of the 3D scene of the robotics visualization tool *RViz* enables developers to leverage existing extensive data visualization capabilities.**

I. INTRODUCTION

Since its introduction by Quigley et al. [7], the robot operating system (ROS) has become the de facto standard middleware in the robotics community. ROS is used by a large variety of robotic systems from stationary to robots moving autonomously over land, air, and sea. Despite the fundamentally different applications, almost all of these share the necessity of having a human-robot interface (HRI). These HRIs can take vastly different forms. In service related fields, the focus is usually on natural interaction, e.g., using spoken language and gestures to communicate [3], and 2D interfaces are mainly used for diagnostics. However, in other areas where the robot is not seen as a separate entity but the safe extension of a human controller to a remote and possibly dangerous environment, 2D interfaces are an integral component of a robotic system's operation. While there have been some recent advances in augmented and virtual reality HRIs [10, 9], the current standard method in the field of rescue and inspection robotics – which this paper is focusing on – are 2D applications. Please note that we do not want to imply that AR/VR and 2D interfaces are mutually exclusive. In our opinion, robotic interfaces in the future will combine the advantages of these methods to maximize the situational awareness and reduce operator errors and the tools presented in this paper can not only be used for standalone 2D interfaces but also for 2D components of an AR/VR interface.

A common consensus across all of the mentioned application fields regarding HRIs – and human-machine interfaces in general – is that an HRI should be designed around the user [1]. In [4] Murphy and Tadokoro divide the users of HRIs into three groups: end-users, developers, and stakeholders / general public. These three groups have different requirements for the user interface. Currently, however, in the field of rescue robotics, the diagnostic interface used by the developers to control and debug the robot is often the same that the end-user is given to operate the robot during a mission which – according to Murphy and Tadokoro – the designers mistake as being satisfactory. The third group requires an interface to visualize and explain the robot's actions as well as the underlying scientific achievements.

Results from the DARPA Robotics Challenge (DRC) finals [5] attribute the success of competing teams to several characteristics which include limited human operator interaction, more robot autonomy for simple tasks, and having multiple specialized operators. Hence, the end-user group may even require multiple interfaces depending on the variety of tasks they will encounter. These results are not directly applicable to rescue force robot operators since participants in the DRC were mainly experienced roboticists. Nevertheless, it strongly indicates the necessity of easily modifiable and quickly creatable, capable HRIs to address (new) tasks with accessible, preferably autonomous, behaviors in research environments.

In the ROS community, the open-source visualization tool *RViz* is one of the major diagnostics and to a limited extent also control interfaces. *RViz* provides 3D visualizations for numerous types of sensor data and can be extended with visualizations for custom representations. However, it has neither been intended nor is it suited as an end-user control interface. For such purposes, it is highly complicated to use even for simple tasks. For example, changing the displayed camera costs the operator a significant amount of valuable time during deployment.

In order to create a custom HRI, for every widely-used programming language there exist numerous frameworks enabling the development of human-machine interfaces (HMI) – of which HRIs are a subset. There are platform-dependent toolkits such as Windows Forms, Windows Presentation Foundation, Universal Windows Platform (Windows / C++ / C# / VB), Cocoa (macOS / Objective-C), Android (Java / C++) and platform-independent toolkits such as GTK (C++ / C#), Swing (Java), QtWidget, QML (C++ / Python / C#),

Stefan Fabian and Oskar von Stryk are with the Technical University of Darmstadt, Computer Science Department, Simulation, Systems Optimization and Robotics Group, Germany.
{fabian,stryk}@sim.tu-darmstadt.de

Research presented in this paper has been supported in parts by the LOEWE initiative (Hesse, Germany) within the emergenCITY center and by the German Federal Ministry of Education and Research (BMBF) within the subproject "Autonomous Assistance Functions for Ground Robots" of the collaborative A-DRZ project (grant no. 13N14861).
978-1-6654-1213-1/21/$31.00 ©2021 IEEE

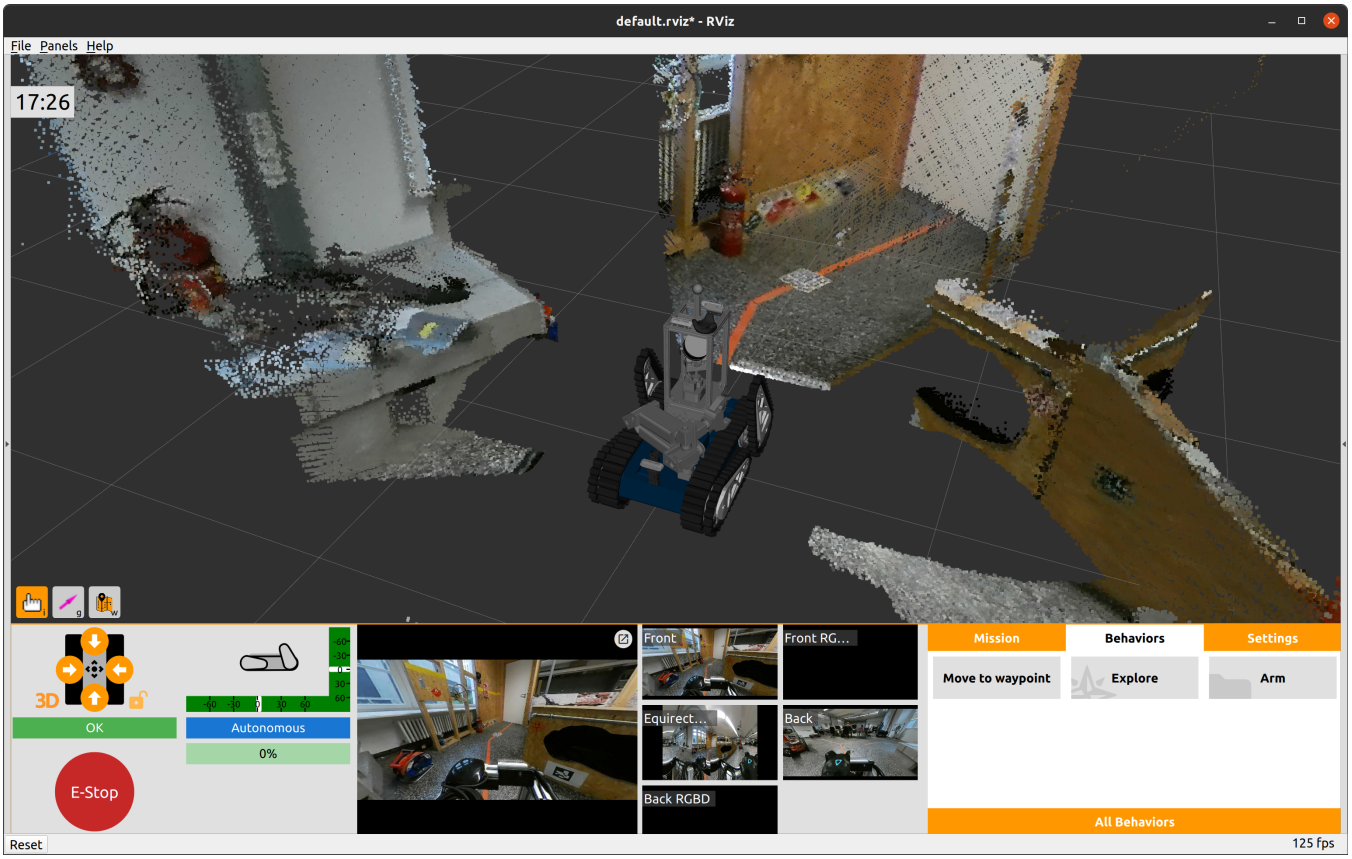

Fig. 1. An example of a human-robot interface created using the packages presented in this paper. The HRI is divided into two major parts: the virtual 3D scene (top) and a dashboard (bottom). The virtual 3D scene shows the robot and a colored pointcloud [6]. Overlaid are the current time at the top left and buttons to select the currently active *RViz* tool. The dashboard contains from left to right, top to bottom: controls for the view controller, an emergency stop button, a visualization of the robot's orientation and flipper positions, the currently active operating mode (autonomous, supervised, safe), the battery status, two multi-camera views, and a tab view with tabs for the mission if the robot is driving autonomously, the behaviors which is a collection of buttons that will trigger FlexBE [8] behaviors using ROS actions when clicked, and a third tab with general operating settings for the robot.

Web-based (JavaScript) and many more. In the robotics context, this number is only limited by the availability of ROS client libraries for the programming language. Officially, ROS supports C++ and Python but there exist community-developed client libraries for other languages such as Java and JavaScript. Most of these HMI frameworks require compilation for every change, rendering them unsuitable for rapid prototyping, and/or require large amounts of boilerplate code which inhibits quick visualizations, e.g., for diagnostics. Others, such as web-based applications are still lacking the performance and 3D rendering libraries available for native applications.

The Qt Modeling Language (QML) aims to bridge this gap as a high-performance GPU-accelerated alternative to HTML-based HMIs. QML is a runtime-compiled declarative language based on the composition of primitive controls and layouts implemented in C++ and supports inline logic using a JavaScript engine that also supports interoperability with C++ code.

In this paper, we introduce five software packages to facilitate rapid prototyping of end-user, diagnostic, and presentation HRIs using QML and *RViz* (see Fig. 1). First, we introduce the *ros_babel_fish* and *ros2_babel_fish* packages which allow communication with ROS and ROS2 respectively using message definitions that don't need to be known at compile time. Then, we introduce *qml_ros_plugin* and *qml_ros2_plugin* which provide a QML module to connect QML-based HRIs to ROS and ROS2-based robots. Finally, we introduce *hector_rviz_overlay*, a package that enables the rendering of Qt-based (QtWidget and QML) HRIs on top of the 3D scene in *RViz*. Combined, these open-source packages will allow the robotics community to quickly develop visually appealing visualizations and performant control or diagnostic interfaces for their robotic demonstrators.

II. HUMAN-ROBOT-INTERFACE TOOLS

In this section, we introduce the packages presented in this paper. The tools consist of three separate components which will be introduced in detail in the following subsection. Bridging the connection from ROS to runtime evaluated languages such as the Qt Modeling Language (QML), the *ros(2)_babel_fish* packages enable the communication with messages, services, and actions that are not known at compile time. Based on these packages, the *qml_ros(2)_plugin* packages connect the HMI framework QML to the ROS framework, allowing ROS communication to be used in visually

TABLE I
FEATURES OF THE CURRENTLY AVAILABLE OPEN-SOURCE ROS COMMUNICATION INTROSPECTION PACKAGES.

	Subscribe	Publish	Services	Load
ros_babel_fish (Ours)	✓	✓	✓	✓
ros_msg_parser	✓			
variant_topic_tools	✓	✓		

rich HRIs. While this is already sufficient to build standalone HRIs, the *hector_rviz_overlay* package enables the integration of such interfaces directly on top of the 3D scene rendered by the popular open-source robot and sensor data visualization application RViz, giving access to the numerous proprietary or publicly available 3D data visualizations created by the robotics open-source community.

A. *ros_babel_fish*

The *ros_babel_fish* package is an introspection and runtime generation package for ROS communication written in C++. It supports messages, services, and actions both as a listener (or subscriber in ROS terms) and a broadcaster (publisher). Since ROS sends the message definition – which specifies the message's structure – with the message, it can decode any received message and will return it in the form of the tree structure depicted in Fig. 2 and explained in more detail at the end of this subsection. Messages sent using *ros_babel_fish* do not need to be known at compile time. If a message of the type that is being sent has been received earlier, the message definition does not need to be available on the host machine, otherwise, *ros_babel_fish* will look up the message definition from the ROS workspace and fail if it is not available. In contrast to other external solutions such as the *rosbridge_suite*, *ros_babel_fish* parses the message directly in the node with a lazy copy mechanism which results in a significantly reduced overhead in both inter-process communication and memory usage. To achieve this, *ros_babel_fish* parses the message definition and creates a template for the message. When a message is received, this template is used to determine the location of each message member in the raw buffer of the received message. While trivial values are copied, arrays only store the information necessary to access the message content in the buffer and will only copy the data if it is modified.

The existing available open-source C++ introspection packages such as *ros_msg_parser*[1] and *variant_topic_tools*[2] are compared to our package in Table I. While all solutions can subscribe to unknown messages, only *variant_topic_tools* and our package can publish messages. Unique features of *ros_babel_fish* include the ability to call and provide services and publish at compile time unknown messages by loading their definition from disk at runtime.

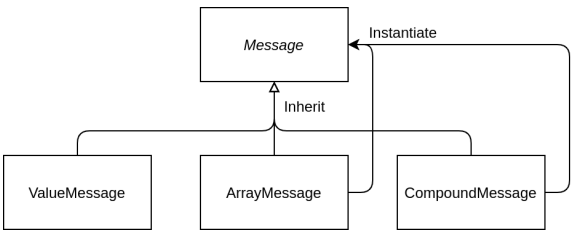

Fig. 2. The structure of the message representation in *ros_babel_fish*

The text definition in the ROS message description specification format is parsed using regular expressions to build a hierarchical template that converts messages to the tree structure depicted in Fig. 2 where each node is one of the following:

1) ValueMessage: This is always a leaf of the tree built from the three message types. It is used to represent a primitive value of a ROS message, e.g., a string or numeric member. For example, the members *x*, *y*, *z* of *geometry_msgs/Point* will be represented as `ValueMessage<double>`.

2) ArrayMessage: This may be a leaf if the values in the array are primitive types, otherwise, it will be an array of *Message*. Arrays of primitives such as the byte array member *data* of *sensor_msgs/Image* are extracted using the previously described lazy-copy mechanism, allowing the introspection into large image messages with minimal overhead.

3) CompoundMessage: The *CompoundMessage* represents the composite ROS message. It has a mapping of string keys representing the names of the message members to *Message* values. For example, the *geometry_msgs/Point* message would be represented as a *CompoundMessage* with the keys *x*, *y*, and *z* each mapping to a *ValueMessage* with the value for each member.

To publish a message, this tree structure is serialized back into a raw memory buffer which can be sent using a standard ROS publisher. For more information and examples to help in getting started, we refer to the GitHub project for *ros_babel_fish*[3].

B. *ros2_babel_fish*

Analogous to *ros_babel_fish*, *ros2_babel_fish* provides introspection and runtime generation for ROS2. In contrast to ROS, ROS2 does not specify a serialization method and instead leaves this to several middleware implementations using a middleware abstraction layer. These implementations differ greatly in the protocol and serialization format they are using. The only other C++ introspection library available for ROS2 to the authors knowledge is *ros2_introspection*[4]. *ros2_introspection* is the ROS2 version of *ros_msg_parser* and is limited to the subscription of messages serialized using the Fast-CDR serialization used by the Fast-DDS middleware.

To avoid this strong dependency on the used middleware, *ros2_babel_fish* makes use of the message introspection packages introduced in ROS2 that allow introspection directly

[1] https://github.com/facontidavide/ros_msg_parser
[2] https://github.com/ANYbotics/variant
[3] https://github.com/StefanFabian/ros_babel_fish
[4] https://github.com/facontidavide/ros2_introspection

into the deserialized C++ message class instances. The message type is either provided or determined from the advertised topic. Once the type is known, *ros2_babel_fish* dynamically loads the type introspection support from the respective message library. This means as opposed to the behavior in ROS1, we need the message binaries to be available at runtime not only for publishing but also for subscribing to advertised topics. The benefit of this approach is that it keeps us independent of the middleware that is being used since we are operating on the layer above the middleware abstraction. In theory, this would also allow using *ros2_babel_fish* with zero-copy shared memory middlewares or intra-process communication since it wraps the message class directly. This is not yet implemented, however. Essentially, this means that the memory wrapped using *ros2_babel_fish*'s introspection structure – that is largely the same as in *ros_babel_fish*, with the addition of fixed length and bounded arrays which were introduced in ROS2 – can be casted directly to an instance of the actual C++ message class.

The main differences to *ros_babel_fish* are that *ArrayMessages* do not wrap a raw buffer but an instance of a C++ container class, the type of which depends on whether it is a fixed-length, bounded or unbounded array. Also, since *ros2_babel_fish* operates on a layer above the serialization, it is not necessary to convert the message before publishing.

For more information and examples to help in getting started, we refer to the GitHub project for *ros2_babel_fish*[5].

C. qml_ros_plugin and qml_ros2_plugin

Existing QML modules such as *ROS QML Plugin*[6] and *ros_qml*[7] only offer limited ROS support. The *ROS QML Plugin* does not support general message publishing and subscription but only a fixed subset which is limited to *TF*, *geometry_msgs/Pose*, *std_msgs/Empty*, *std_msgs/String*, and displaying a ROS image topic as a QML image. While it theoretically can display a live video feed from a ROS camera, it uses a hack to reload the source of an image at a fixed refresh rate. This leads to additional latency and wastes computation resources as frames are not presented as they are received but at a fixed rate and reloaded even if no new message has been received. The *ros_qml* plugin supports publishing and subscribing to any message type but it does not support services, actions, or include methods to allow the displaying of image messages which would require special handling. For ROS2, to the authors knowledge, there exist no publicly available modules offering any ROS2 integration.

Our contribution, the *qml_ros_plugin* package and the *qml_ros2_plugin* package support all features of the existing modules except *ROS QML Plugin*'s ability to publish QML items as ROS images. Additionally, it can also be interfaced with JavaScript to call services (both synchronous and asynchronous) or actions using the previously introduced *ros_babel_fish* and *ros2_babel_fish* packages. To interface

[5]https://github.com/LOEWE-emergenCITY/ros2_babel_fish
[6]https://github.com/severin-lemaignan/ros-qml-plugin
[7]https://github.com/bgromov/ros_qml

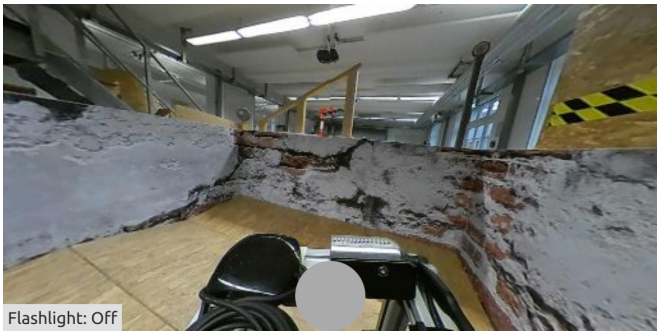

Fig. 3. The remote control user interface declared in Listing 1. It displays a camera feed, a grey joystick area that can be used for steering with mouse or touch input, and a button to toggle a flashlight using a ROS service.

with QML, the message introspection tree-structure is converted to a QML compatible representation. *ValueMessages* are converted to *QVariants* (Qt's implementation of variant types), *CompoundMessages* to *QVariantMaps* which represent a mapping from string keys to *QVariant* values, and *ArrayMessages* are wrapped using a custom wrapper type implementing a lazy-copy abstraction only converting values when requested.

ROS camera streams are handled separately and do not use a type introspection library. Instead, an *ImageTransportSubscriber* takes a *QAbstractVideoSurface* which allows the camera frames to be rendered using a standard QML *VideoOutput* item. The *ImageTransportSubscriber* subscribes to the camera using the default ROS method provided by the *image_transport* package and converts the received image to an image format that is compatible with the used *VideoOutput* if necessary. This allows the stream to update whenever (and only once) a new image is received. The *ImageTransportSubscriber* also provides statistics that can be read from QML such as the received framerate, the processing latency introduced by the potentially necessary conversion and the network latency. The latter requires the clock of the robot and the computer running the user interface to be synchronized.

To reduce the network load when displaying previews of many cameras, our image subscribers can be configured to refresh only at a specified interval – e.g., 5 seconds – and use a simple load-balancing algorithm to ensure that the updates of the camera images do not occur at the same time which would result in large network usage spikes. Multiple *ImageTransportSubscribers* for the same camera share a single subscription reducing the CPU load caused by the required image format conversions.

To illustrate how quickly a standalone robot control user interface can be built using QML and the presented module, we provide the 36 lines of code in Listing 1 that create a minimal remote control user interface for a ground robot. The resulting application is shown in Fig. 3 and features a video stream and a gray circle that can be used to steer the robot with the mouse or touch. To steer the robot, while pressing the left mouse button on the circle at the bottom center, movement along the vertical axis results in

a forward/backward movement whereas movement on the horizontal axis results in a rotation around the yaw axis. The movement commands are published directly to the `/cmd_vel` topic as a *geometry_mgs/Twist* message.

Listing 1. A quite minimal remote control user interface

```
import QtQuick 2.2
import QtQuick.Controls 2.5
import QtMultimedia 5.10
import Ros 1.0
Item {
  Component.onCompleted: Ros.init('qml_example_node')
  ImageTransportSubscriber {
    id: imageSubscriber
    topic: "/camera360/pinhole_front/image_rect_color"
  }
  VideoOutput { anchors.fill: parent; source: imageSubscriber }
  Rectangle {
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.bottom: parent.bottom
    width: 120; height: 120
    radius: 60
    color: '#aaaaaa'
    MouseArea {
      anchors.fill: parent
      property var pub: Ros.advertise("geometry_msgs/Twist",
                                      "/cmd_vel", 1)
      onPositionChanged: {
        var x = -mouse.x / (width / 2) + 1
        var y = -mouse.y / (height / 2) + 1
        pub.publish({linear: {x: y}, angular: {z: x}})
      }
    }
  }
  Button {
    anchors.bottom: parent.bottom
    property bool isOn: true
    text: 'Flashlight: ' + (isOn ? 'On' : 'Off')
    onClicked: Service.call('/flashlight', 'std_srvs/SetBool',
                            {data: (isOn = !isOn)})
  }
}
```

Additional short getting started examples, as well as documentation, are available on the respective GitHub projects[8].

D. hector_rviz_overlay and hector_rviz_plugins

HRIs for end-users and, even more, diagnostic HRIs require the visualization of the robot's acquired and aggregated data. Building such visualizations is time-consuming, hence, for prototypes and internal diagnostic HRIs it is preferable to build upon existing highly configurable software such as *RViz* for which many data visualizations are freely available. However, the rendering capabilities of *RViz* are mostly limited to 3D visualizations and while it is possible to add 2D components such as text or buttons that trigger autonomous behaviors, they will be outside of and take away screen space from the 3D view. The *hector_rviz_overlay*[9] package enables GPU-accelerated rendering of QML- and QtWidget-based HRIs overlayed on top of the *RViz* 3D scene. It injects itself into the rendering pipeline to ensure that the overlay is drawn after each frame of the 3D view is rendered, and filters the input events received by *RViz* to support interaction with the rendered overlay(s).

For QML overlays the *hector_rviz_overlay* package also supports live reloading which automatically reloads the overlay if one of the QML files changed. In addition, it provides an interface that can be used to interact with *RViz* directly. For example, retrieve and change the currently active tool, add properties for the user to modify that will be saved to and restored from the *RViz* configuration, and to track the position of a 3D coordinate in the 2D viewport, e.g., to display hints or context-based popups for an object in the 3D scene.

RViz 2 ports of the presented packages are not available at the time of writing.

III. APPLICATIONS

Applications for the packages presented in this paper are manifold and include standalone or *RViz* based end-user HRIs such as the one depicted in Fig. 1. This interface example provides a virtual 3D view of the robot and its environment, its camera streams, status visualizations, and buttons to control the robot's behavior and the view controller for the 3D view. The latter has been released as part of our open-source *hector_rviz_plugins*[10] package. Using the view controller's ROS interface and the *qml_ros_plugin* package, the interface can move the camera to different positions relative to the robot and lock it on to the robot which keeps the camera at the same location relative to the robot. The rendering of the QML interface in Fig. 1 takes on average $10.88\,\mathrm{ms}$ (CPU time: $3.07\,\mathrm{ms}$) with a standard deviation of $2.85\,\mathrm{ms}$ ($1.61\,\mathrm{ms}$) on a 15W i7-8550U notebook processor with integrated graphics. For comparison, a $60\,\mathrm{Hz}$ refresh rate requires a frame time of at most $16\,\mathrm{ms}$.

In the scientific context, a less obvious but equally important application is in quickly creating visualizations for the demonstration or debugging of algorithms. An example of such an application is depicted in Fig. 4 where it is used to

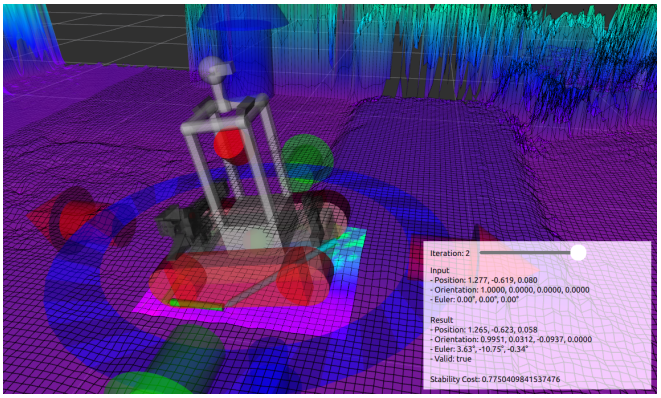

Fig. 4. A basic interactive visualization of an iterative pose prediction. The iteration can be selected using a slider. For the selected iteration, the input and output orientation and the stability value are displayed.

[8] https://github.com/StefanFabian/qml_ros_plugin
https://github.com/StefanFabian/qml_ros2_plugin

[9] https://github.com/tu-darmstadt-ros-pkgs/hector_rviz_overlay

[10] https://github.com/tu-darmstadt-ros-pkgs/hector_rviz_plugins

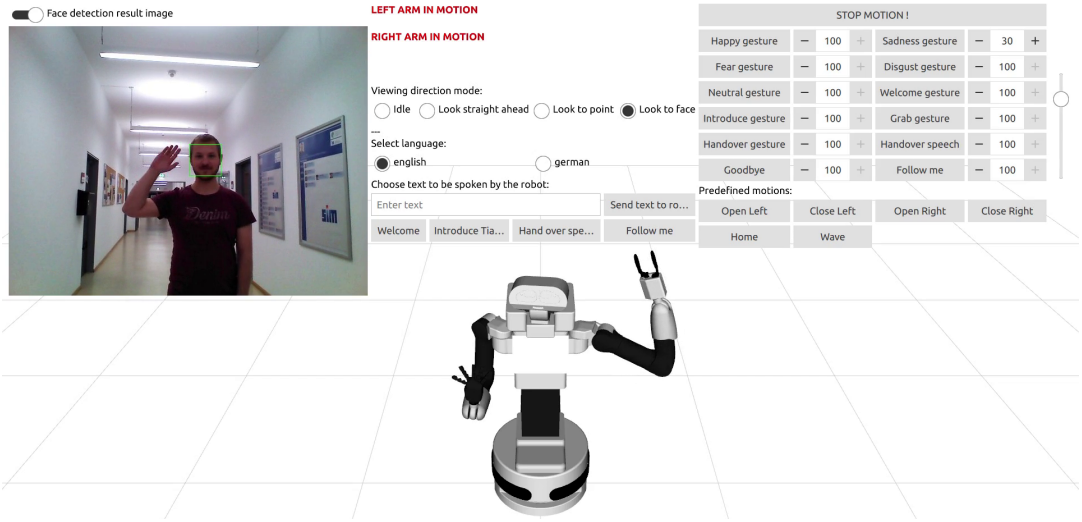

Fig. 5. A demonstration interface for a new robot platform for research in interaction of humans with humanoid service robots that was quickly created using the tools introduced in this paper.

investigate and visualize the iterative geometric robot pose prediction method developed in [2]. Commonly, iterative methods are investigated by logging relevant data and either using console commands or using start parameters to set the iteration that is investigated. While these methods can be implemented quickly, they are cumbersome to use and need to be well documented to avoid confusion. Using these packages, an overlay can be quickly constructed that displays information, such as the orientation and stability published using ROS messages, and allows to change the displayed iteration, using a slider that upon change will call a service requesting the selected iteration.

Another important application is the demonstration of the robotic system to important stakeholders such as politicians or research funding commission members. Especially, if a new platform that is not yet fully integrated with the software stack or the scientific achievements are not yet in a presentable state, the packages introduced in this paper can be used to quickly create a basic user interface to demonstrate some of the robot's skills. An example of such a user interface is depicted in Fig. 5. In approximately 500 lines of QML code written by a researcher without any previous experience in QML, it allows controlling the robot's state, trigger gestures and custom speech output, and visualize the robot's sensors and motor state.

IV. CONCLUSION

In this paper, we have presented a novel software framework that allows ROS and ROS2 users to efficiently create capable and visually rich HRIs for a very general range of robotic systems and applications that can run with high refresh rates even on weak hardware. The existing capabilities of *RViz* can be utilized and easily extended with custom ROS data visualizations and interactive triggers for autonomous behaviors.

V. ACKNOWLEDGMENT

We would like to thank all members of Team Hector for their support in testing the user interface framework. Especially, we thank Karim Barth for his help in designing and implementing our user interfaces with the presented tools. We also thank Jérôme Kirchhoff for creating an application example and his help in creating the demonstration video.

REFERENCES

[1] J. Adams. "Critical Considerations for Human-Robot Interface Development". In: 2002.

[2] Stefan Fabian, Stefan Kohlbrecher, and Oskar Von Stryk. "Pose Prediction for Mobile Ground Robots in Uneven Terrain Based on Difference of Heightmaps". In: *2020 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. DOI: 10.1109/SSRR50563.2020.9292574.

[3] Thomas Kollar et al. "A Multi-modal Approach for Natural Human-Robot Interaction". In: *Social Robotics*. 2012.

[4] Robin R. Murphy and Satoshi Tadokoro. "User Interfaces for Human-Robot Interaction in Field Robotics". In: *Disaster Robotics: Results from the ImPACT Tough Robotics Challenge*. DOI: 10.1007/978-3-030-05321-5_11.

[5] Adam Norton et al. "Analysis of human–robot interaction at the DARPA Robotics Challenge Finals". In: *The International Journal of Robotics Research* 36 (2017).

[6] Martin Oehler and Oskar von Stryk. "A Flexible Framework for Virtual Omnidirection Vision to Improve Operator Situation Awareness". In: *2021 European Conference on Mobile Robots (ECMR)*. 2021.

[7] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*.

[8] Philipp Schillinger, Stefan Kohlbrecher, and Oskar von Stryk. "Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics". In: *IEEE International Conference on Robotics and Automation*. 2016.

[9] Daniel Szafir. "Mediating human-robot interactions with virtual, augmented, and mixed reality". In: *International Conference on Human-Computer Interaction*. 2019.

[10] Veiko Vunder et al. "Improved Situational Awareness in ROS Using Panospheric Vision and Virtual Reality". In: *2018 11th International Conference on Human System Interaction (HSI)*. IEEE. DOI: 10.1109/HSI.2018.8431062.