roboyVR

Release 0.0.1

Apr 22, 2018

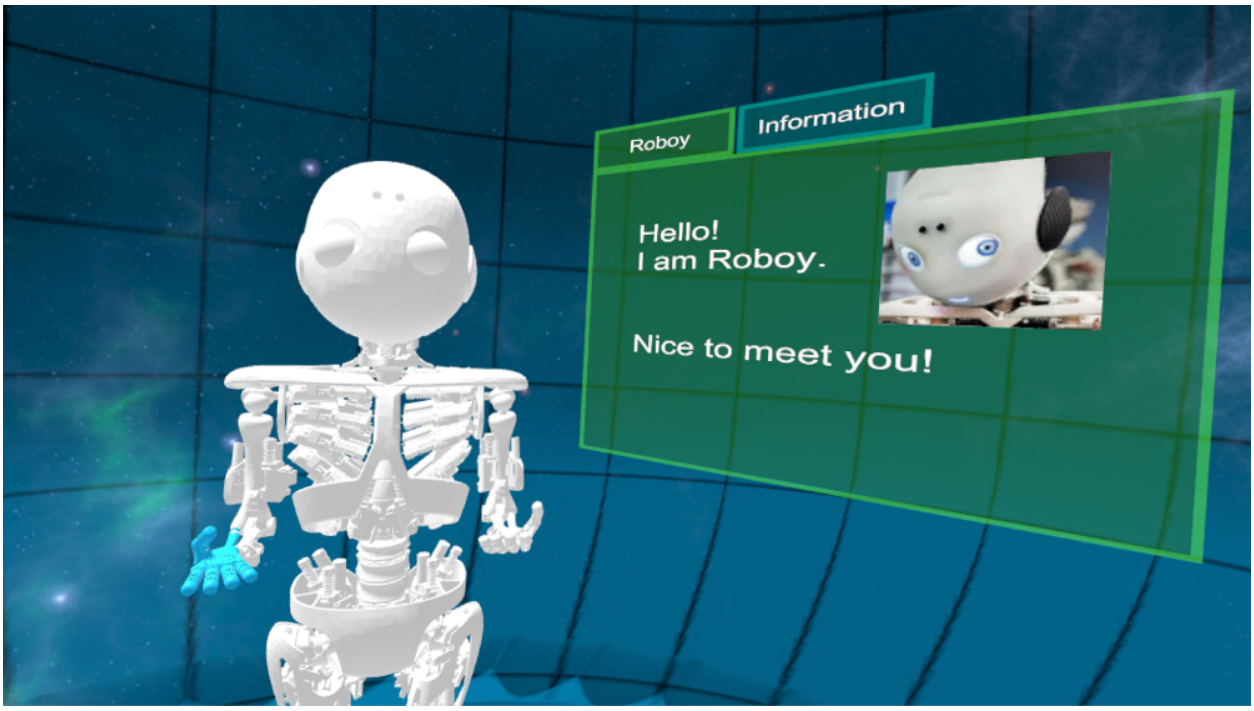
Installation and Usage

1 What is it? 3

2 How does it work? 5

3 Current status of the project 7

4 Contents: 9


Roboy
Information
Hello!
I am Roboy.
Nice to meet you!

CHAPTER 1

What is it?

RoboyVR is a virtual reality experience in which the user can experience and interact with Roboy (a humanoid robot) in a virtual environment. He or she can watch and study Roboy while he performs specific tasks, e.g. walking, waving, etc., while receiving additional information about the robot's state. The user can also interact with him, move him around, shove or pull him. Other features include BeRoboy, where the user can experience the virtual world through Roboy's eyes, see what he sees and move his body as if it was his or her own.

CHAPTER 2

How does it work?

The virtual environment is run using Unity, a game development platform. In the project, a model of Roboy in a virtual environment is provided and the user interaction is detected and processed here. Roboy and its behavior are simulated in Gazebo on a Linux OS and the relevant information such as positions and forces are shared using ROS, a tool for inter-platform communication. Forces or input from the user are sent from Unity to Gazebo, where these values are used to simulate Roboy in a physically accurate way. The resulting new positions are then provided back to Unity, which updates the positions of the displayed model. Additional information which is provided, such as forces applied to tendons, angles and notifications, are displayed in a custom User Interface.

CHAPTER 3

Current status of the project

Currently, Unity provides a model which adapts its position according to the values provided by a simulation. It can display additional information such as tendons and its forces as well as notifications when provided with these. Motor values are stored and processed, but not visualised as of now.

Older code previously implemented: BeRoboy enables the user to see through Roboy's eyes, the head and hand position of him or her are mapped to the model to move along with the user. Furthermore, different Roboy models can be downloaded from an online repository and will automatically be imported into Unity. A selection of these can be spawned in the virtual environment and later be removed if desired. Please notice that these functionalities might not be working correctly anymore.

CHAPTER 4

Contents:

4.1 Installation

As described on the main page, RoboyVR is a project which spans over multiple platforms to simulate Roboy and visualize all data. Unity is used to display the virtual environment and Roboy to the user, run on a Windows OS. The virtual reality is simulated using SteamVR and a HTC Vive headset together with two hand controllers. These are tracked with a lighthouse system.

> ROS is a server on which entities can publish and subscribe to certain topics and information, hosted on an Linux / Ubuntu platform. Roboy and its behavior is simulated in Gazebo which also runs on a Linux OS and publishes its values on ROS. Unity in turn subscribes to these topics to receive all updates. In Unity roboy is rendered and constantly updated concerning positions, rotations, etc.

With the help of a VR-Headset you can watch roboy move around in a virtual space.

This tutorial will help you setup roboyVR with all necessities it comes with.

4.1.1 Part 1: Setup an Ubuntu OS

A machine hosting the ROS server and running the gazebo simulation is needed, conveniently, both can be combined and executed on one machine. Virtual machines were not used - it is possible, that the setup works, out of perfomance reasons and stability, a separate machine is advised.

4.1.2 Part 2: Simulation Setup

Follow the setup instructions on the main Roboy repository.

Note: the setup.sh of gazebo is in /usr/share/gazebo-7/setup.sh and not in ../gazebo-7.0/..
Note: Export the gazebo paths AFTER the catkin_make because the devel directory is just created at this command.

On top of that it may be necessary to update the submodules of this repository:

```
cd /path-to-roboy-repository/
git submodule update --recursive --remote
```

There may also occur an error that says that you need to install the OpenPowerlink stack library. In that case follow the instructions on the OpenPowerlink Homepage. The OpenPowerlink folder lies in the *roboy_powerlink* folder.

4.1.3 Part 3: Unity Setup

1. Download Unity
   - (latest working version with roboyVR is 2017.1.0: https://unity3d.com/de/get-unity/download/archive)
2. Install Unity
   - During the install process make sure to check also the standalone build option.
   - Visual studio is recommended to use with Unity3D, as it is free and more user friendly than MonoDevelop (standard option).
3. Download this project
   - Clone this github repository (master branch) to your system: https://github.com/roboy/roboyVR.git
   - Command: git clone -b master https://github.com/roboy/roboyVR.git

4.1.4 Part 4: Setup of the Lighthouse Tracking System

A lighthouse tracking system is needed to track the user's movement in space for both controllers and the headset. The two base station should be able to see each other clearly with no viewing obstructions in their sights. They should be put up diagonal spanning a virtual room of two by five meters. For additional information take a look at this guide HTC Vive setup.

Install Steam/SteamVR in order to be able to use the Headset in the Untiy project.

4.1.5 Part 5: Blender & Python

- Install the latest version of Blender
- Install the latest version of Python
- Install the PyXB: Python XML Schema Bindings
- After installation, add the Python executable directories to the environment variable PATH in order to run Python. (Windows 10: http://www.anthonydebarros.com/2015/08/16/setting-up-python-in-windows-10/)

4.1.6 Part 6: ZED API and Unity Plugin

- Install the latest verstion of the API
- If the Unity plugin in the project is outdated then download the plugin from the same page and import it into the project to update it

4.2 Getting started

4.2.1 Part 1: Run rosbridge and roboySimulation

Before the simulation is run, model paths and IP addresses need to be sourced / specified and ROS messages defined. The following commands can be applied in any terminal, even if some might not be necessary, they don't do any harm and may save you from hardship later, thus executing all commands is advisable. IMPORTANT: Adapt paths to lead to /roboy-ros-control/ folder!

```
export GAZEBO_MODEL_PATH=/path/to/roboy-ros-control/src/roboy_models:$GAZEBO_MODEL_
↪PATH
export GAZEBO_PLUGIN_PATH=/path/to/roboy-ros-control/devel/lib:$GAZEBO_PLUGIN_PATH
export GAZEBO_RESOURCE_PATH=/path/to/roboy-ros-control/src/roboy_models:$GAZEBO_
↪RESOURCE_PATH
source /path/to/roboy-ros-control/devel/setup.bash
source /usr/share/gazebo-7/setup.sh
```

On the Linux/Ubuntu machine, the simulation and ROS server are run. There are two ways to start a simulation: launch files automatically take care of starting a ROS server, gazebo and other plugins and immediately start the simulation. Another option is to start ROS manually using the roslaunch command below. The simulation is run with rosrun. In case you need to change the port on which you want to operate (e.g. because it is already in use), adapt the value of the following command or in the respective launch and make sure to also change the value in the Unity scene.

```
#option 1:
roslaunch roboy_simulation roboy_moveable.launch

#option 2:
source path-to-roboy-ros-control/devel.setup.bash
roslaunch rosbridge_server rosbridge_websocket.launch port:=9090
rosrun roboy_simulation VRRoboy
```

NOTE: When running a simulation like VRRoboy / VRRoboy2, plugins which are defined in models WILL NOT be loaded - even though they are supposed to (at least expected from our side), therefore roboy_moveable was implemented in a launch file but is designed to do the same as VRRoboy2.

4.2.2 Part 2: Open the project in Unity

One main scene is designed to incorporate all project aspects, including a UI, the simulated Roboy model and more. on the Windows machine, start Unity and open this scene. The only exception poses the PaBi model, which is described in an extra chapter. Open the main scene in all other cases.

4.2.3 Part 3: Setup the scene

To receive and display the pose which is simulated on he Linux machine, both machines need to be situated in the same network and the ROS server's IP address and port needs to be registered in Unity. The following command shows which IP the ROS server is using.

```
echo $ROS_IP
```

You can reset the simulation with the **R** key or with both grip buttons on the Vive controller of the *GUI Hand*. You can also change the key in *RoboyManager*. **Notice: This is not working for all simulations**.

4.2.4 Part 4: Using RViz [Optional]

RViz is a tool to visualize the simulated Roboy. As soon as values are provided to RViz via ROS, it can display these. Additionally, it visualizes forces, as can be seen when visualising the roboy_moveable.launch simulation in this program.

To start Rviz, open a terminal and type **rviz**, hit enter after the ROS server started, otherwise it won't connect to the server. Set Fixed Frame to World (Displays->Fixed Frame) . Add a marker (Add(Button)->marker).

4.3 For Users and Developers

4.3.1 For the user

The user can simply start exploring the virtual environment without further ado. One of RoboyVR design goals is to have a user friendly and intuitive interface with which one can easily interact. Therefore, the user in the virtual environment does not need to be familiar with explicit commands or structures, he or she can simply explore the scene, experiment with all controllers and possible options without fear of issues or runtime failures. Yet, it does no harm to have a basic understanding of how the HTC Vive and its tracking mechanism work.

Putting on the head mounted display in a way that fits the user, he or she can adjust the distance from the lenses to the eyes as well as the distance between the lenses itself. These tweaks help immensely when it comes to maintaining a sharp field of view and having the full view frustrum.

4.3.2 For the developer

RoboyVR uses Unity3D to create an immersive and exciting virtual environment. Expierence with Unity is recommended. Unity natively relies on C#, so knowledge in this field is highly advised. Otherwise see Unity3D.

The Roboy simulations which run in Gazebo is written in C++, which normally load a world and model from an .sdf file, any additional plugins (written in C++) specified in these files and then simulate the given scene. For starting the simulation you should be familiar with Linux/Ubuntu.

The following links can be seen as a guideline, of course you can do the research by yourself.

- Unity provides a lot of tutorials for the editor and the API with code samples and videos.
- UnifyWiki has a lot of example scripts for all kind of extensions.
- StackOverflow is a forum where you can search for answers regarding your coding problems.
- UnityAnswers, similar to StackOverflow but only for Unity specific questions. The community is not as active and most questions are really basic, so bear that in mind.
- As we use ROS and our own custom messages, it is important to understand how ROS works and how ROS messages are built.
- Gazebo simulations rely on gazebo libraries, which are documented here

If you have any further questions about the project, feel free to contact the Roboy team, which will connect you with the people in charge.

4.4 Controllers

Two different controller types exist: One controller option is used as a tool for pointing and interacting with different objcets in the scene, while the second controller option is dedicated to different modes, providing the option to change modes and apply further settings in each mode.

The initial setup enables the user to choose which type of controller to have left and right, respectively.

Choose the pointing / tool controller

Fig. 4.1: Press the trigger of the prefered controller

4.5 Tool

Different tools can be selected which provide unique functionalities, such as grabbing pointing and shooting. Tools can be changed by pressing the buttons on the side of the controller designated for the tools. The following picture shows the selection wheel with which the current tool is selected.

4.5.1 Pointing Device

The pointing device is the standard device, it is the go-to tool when interacting with UI components and Roboy. It casts a laser, which highlights the pointed body part and gives haptic feedback when selecting something.

4.5.2 Hand

The hands are useful when interacting with the Roboy model, for example when pushing or pulling Roboy's arm. More information can be found under the point *Moving Roboy*.

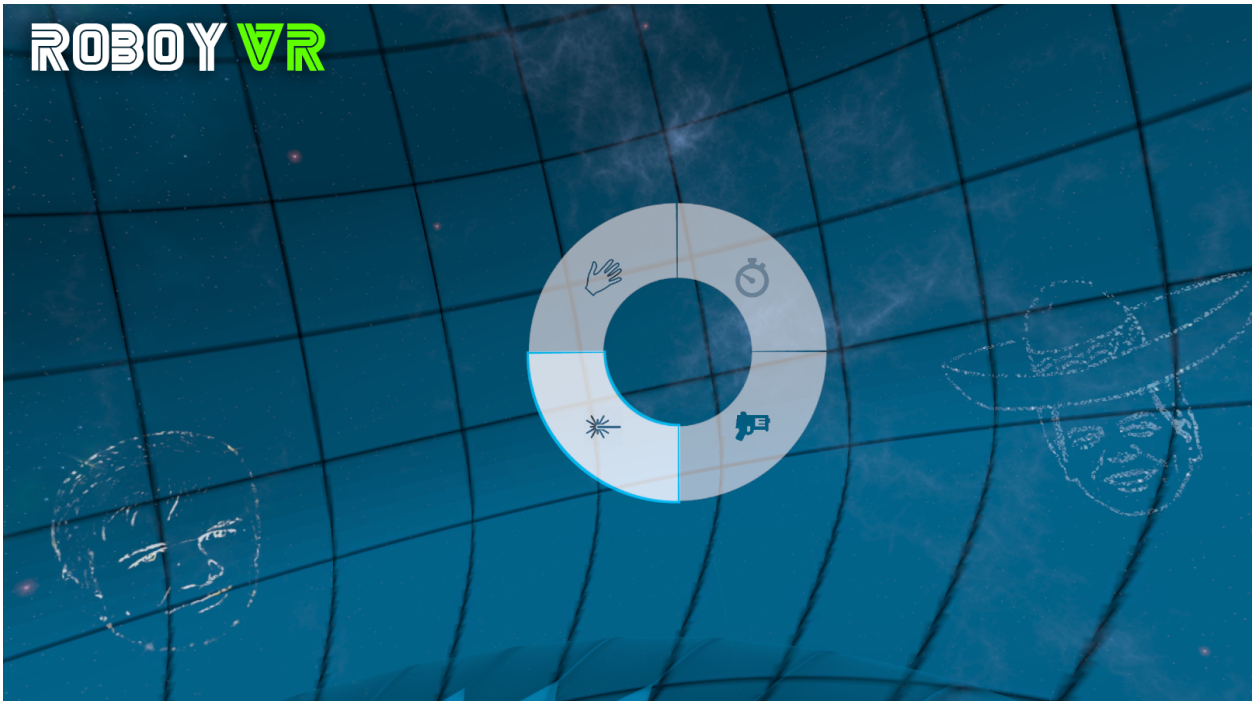


Fig. 4.2: The selection wheel offering different tools

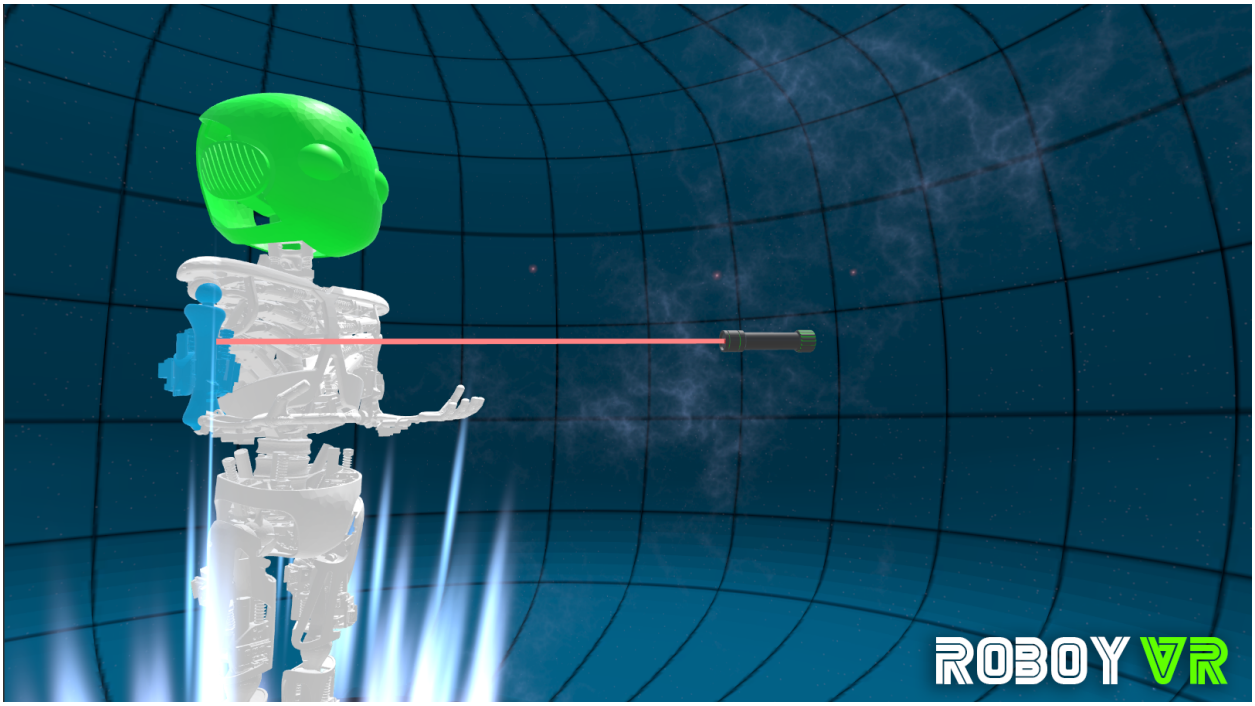


Fig. 4.3: Pointing Device to interact with Roboy and the UI

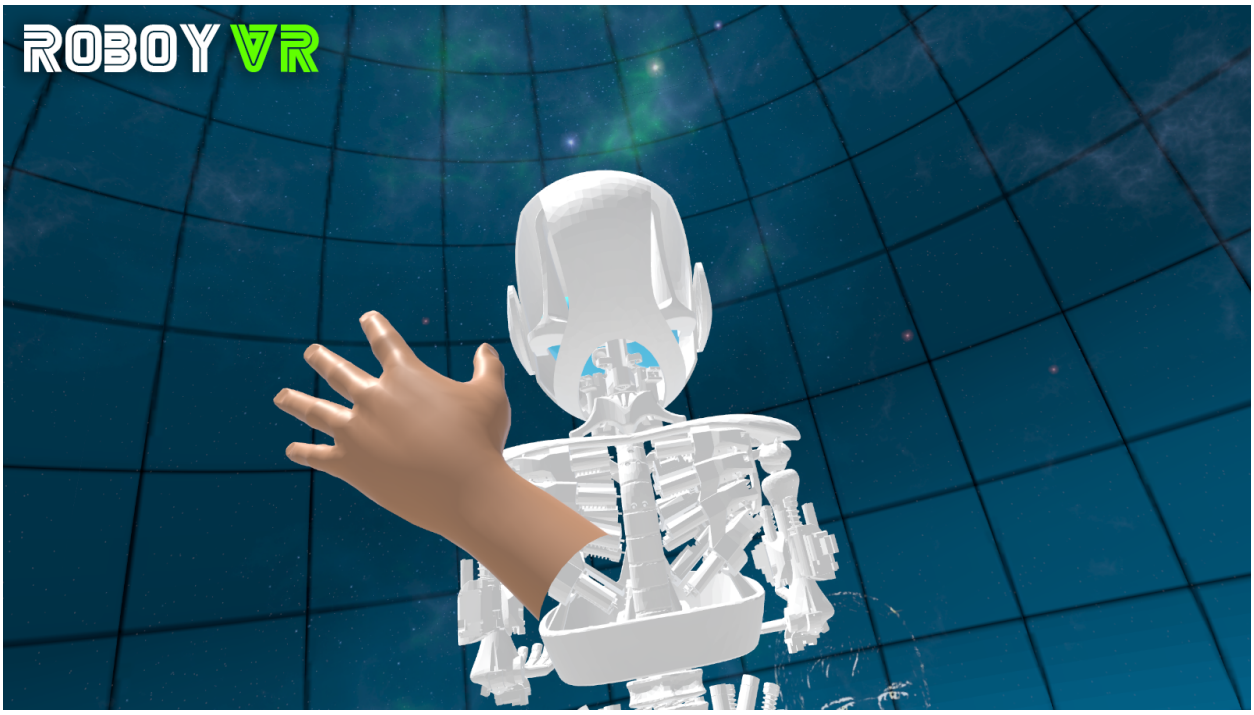


Fig. 4.4: Hand Tool to move Roboy around.

4.5.3 Watch

The whatch consists of a pocketwatch with one pointer, which can be manipulated by moving your finger over the touchpad representing the clock face. AAs of now there are no further functionalities. In the future it will be possible to control time, so to rewind the simulation and save/ load them on runtime.

4.5.4 Toy Gun

One tool which can be selected is a toy gun. When pressing the trigger, foam projectiles shoot in the direction of the pointed gun. If desired, these projectiles can inflict forces on hit Roboy parts.

4.6 Modes

Different modes can be chosen which provide distinct functionalities such as controlling Roboy, examining him from afar, or being in charge of Roboy's body and seing the world through his eyes. The following image shows the different mode options.

4.6.1 ViewSelection

This mode provides functionalities to change the current view from the initial user's point.

Choosing BeRoboy, the user is able to control the model and perceive the environment through the eyes of Roboy, either through his real eyes or his eyes in the running simulation. Further information can be found in the **BeRoboy** chapter.

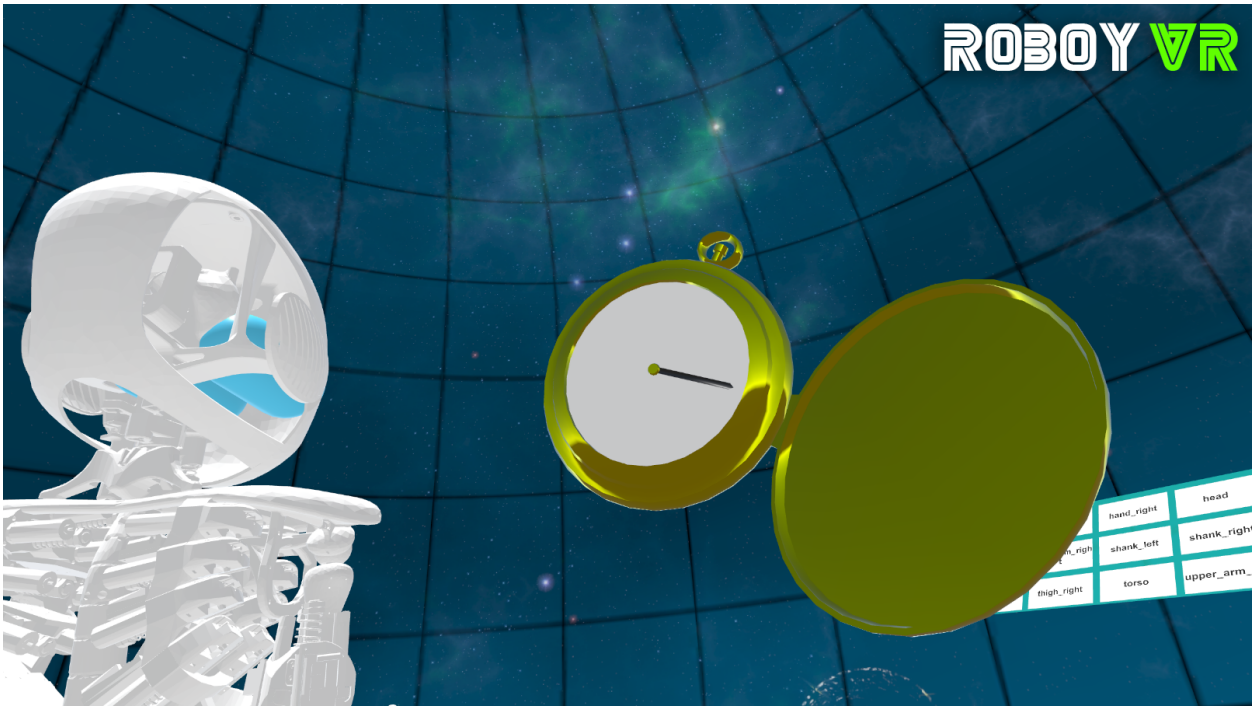


Fig. 4.5: Tool to alter flow of time.

Fig. 4.6: Tool to shoot at Roboy

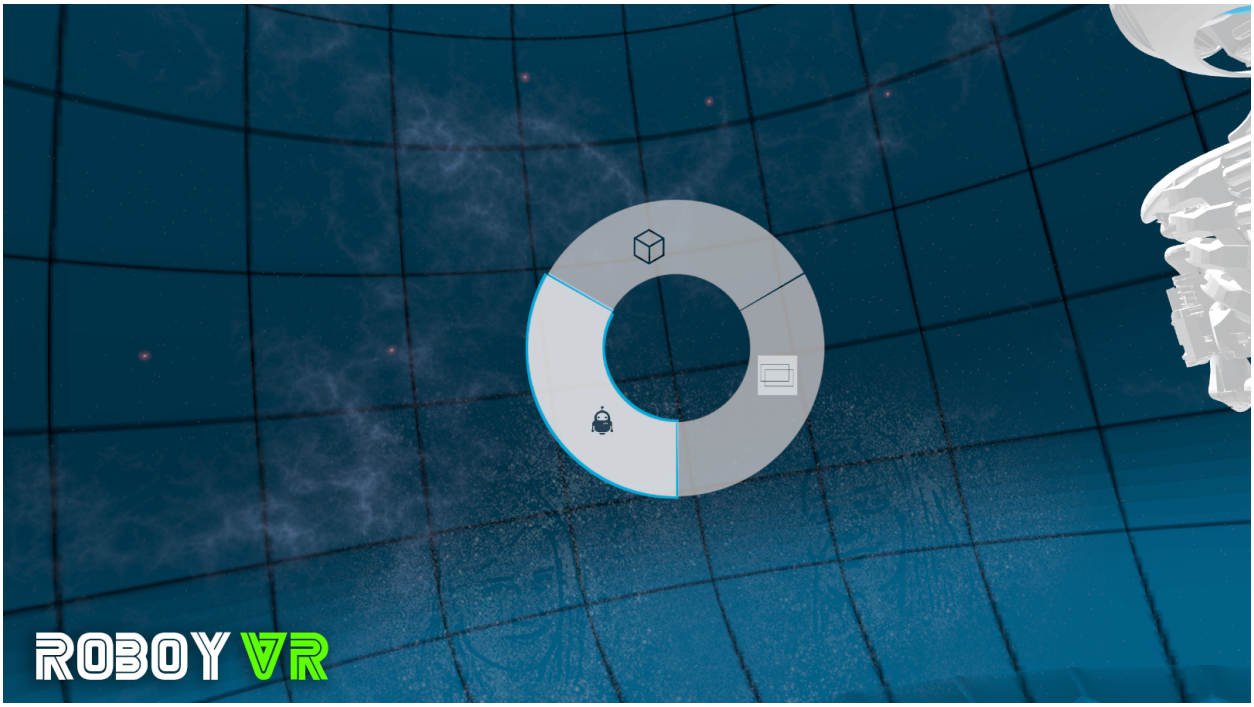


Fig. 4.7: Selection wheel to change between different modes

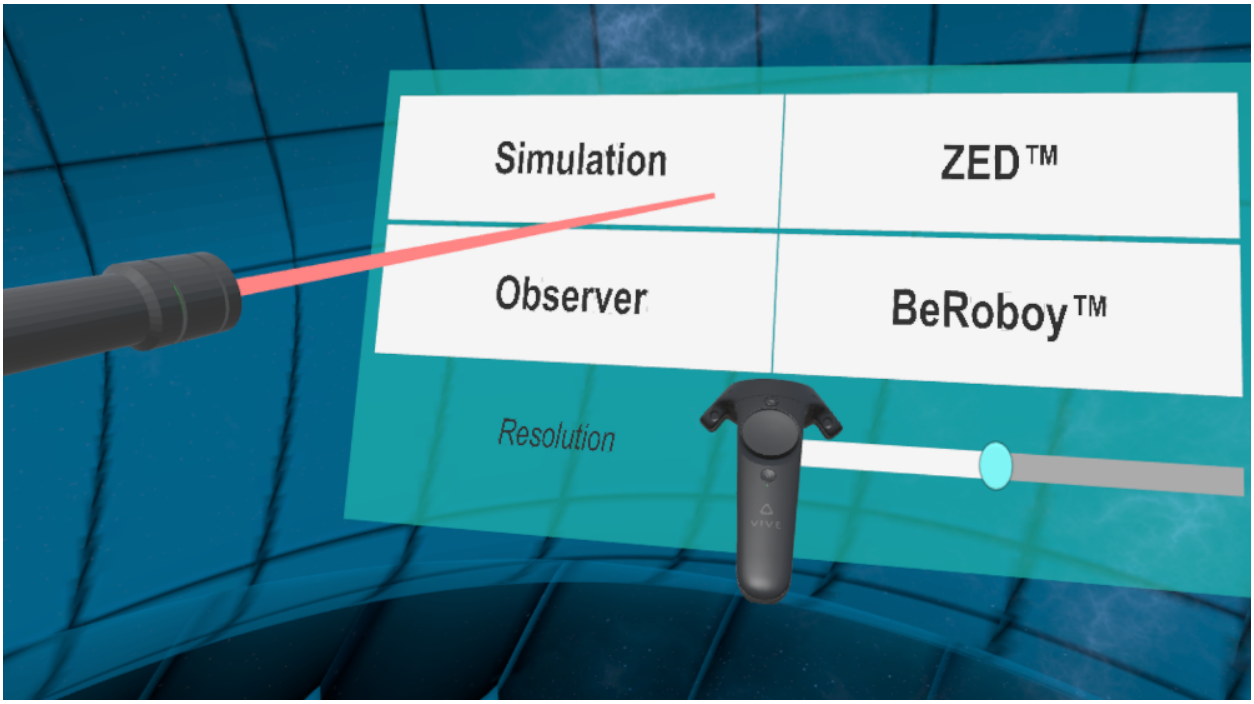


Fig. 4.8: Choose from different views including Observer and BeRoboy mode.

4.6.2 GUI Mode

This mode contains the user interface displaying information about Roboy. It is described in more detail in teh chapter **State Visualisation**.

4.6.3 Model Spawner

It is possible to insert and remove numerous models from the virtual world simply by point and click.

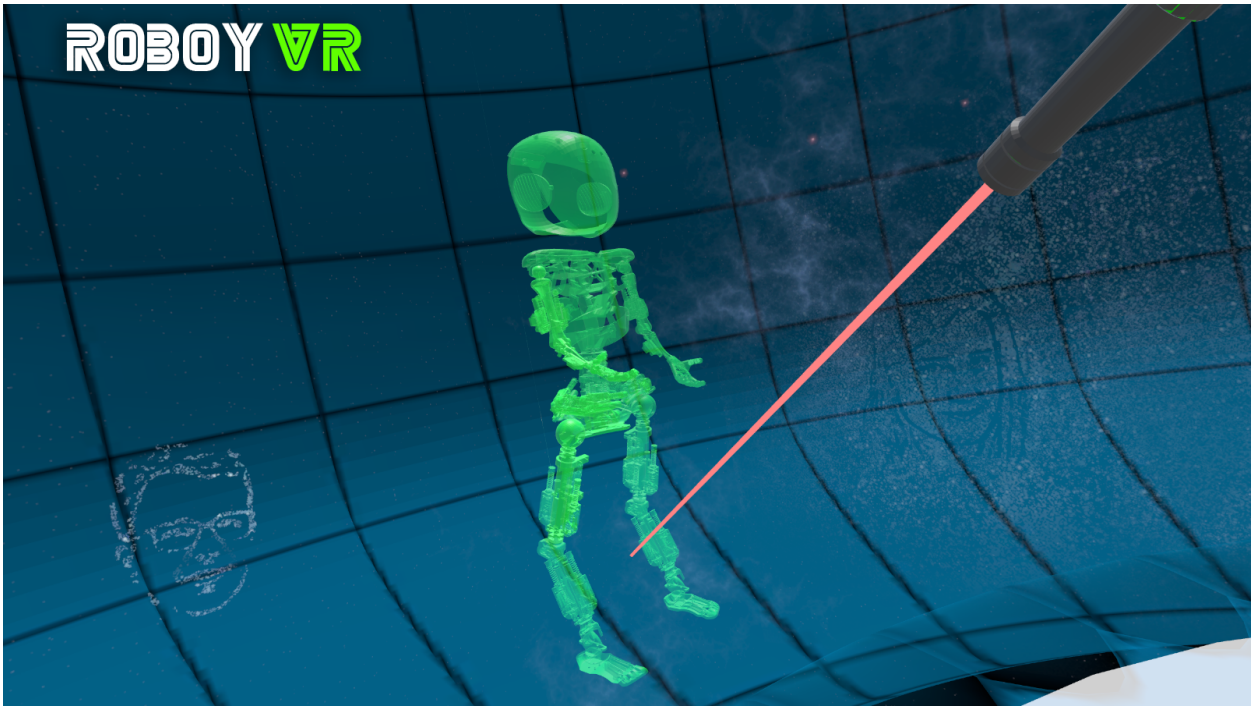


Fig. 4.9: Possible place for model to be placed

Just choose the model you want to insert and click on a free space on the ground.

Removing models is partly possible, even though it does not work with all of the models, e.g. the surprise model and PaBi will not work.

NOTICE: LEGACY PARTS This model will be inserted in the world of Unity but not in the gazebo simulation. Previous code implemented this function for certain models, but since this is legacy code, it is not expected to be working anymore. Additionally, the model poses were / are not updated by the simulation because of lacking message infrastructure.

4.7 Teleporting

Using the touchpad of the designated controller as a button and pressing down the lower end of it, the teleporting option becomes visible. By holding and pointing the controller around in space, the destination can be determined. In case the position is not possible, e.g. if it is outside the cave, it cannot be selected. If the spot is a valid destination, the user is spawned to this position as soon as he or she releases the touchpa.

Teleporting in a bit buggy as of now, the navigation mesh is in need of an update as we increased the scale on our VR cave. This was done to give the user more space to insert additional models in the environment.

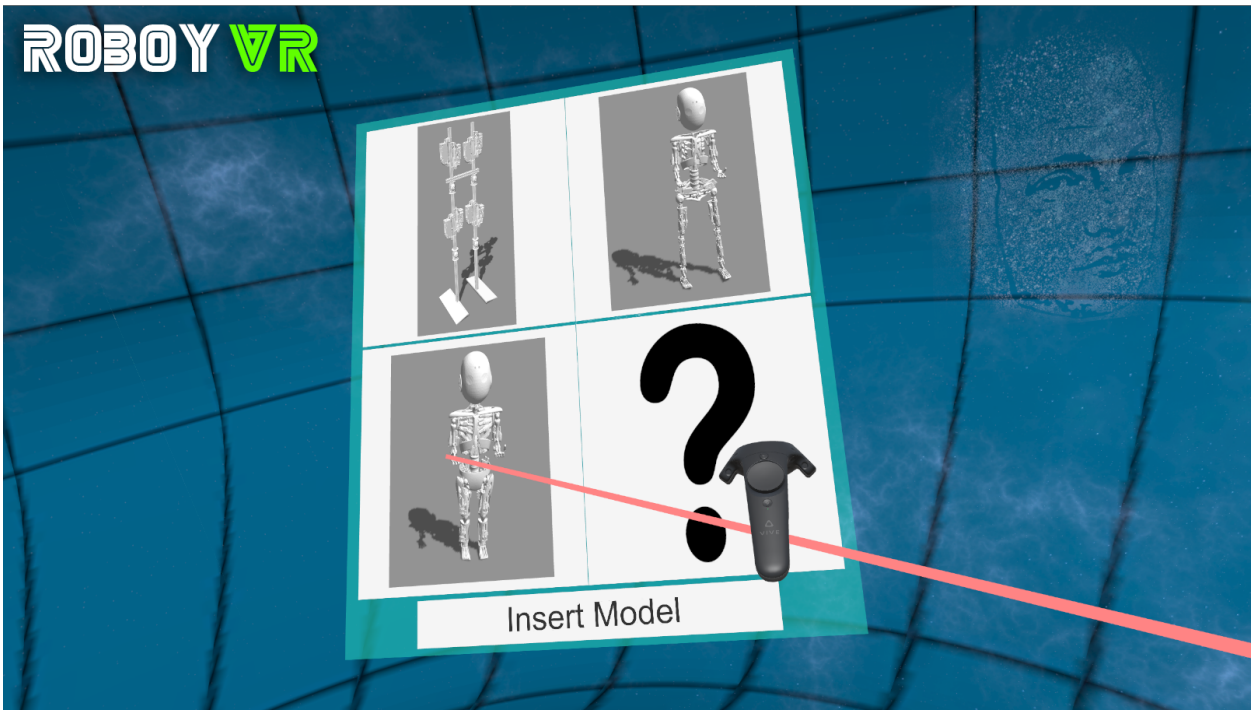


Fig. 4.10: Selection panel to choose a model which is to be inserted

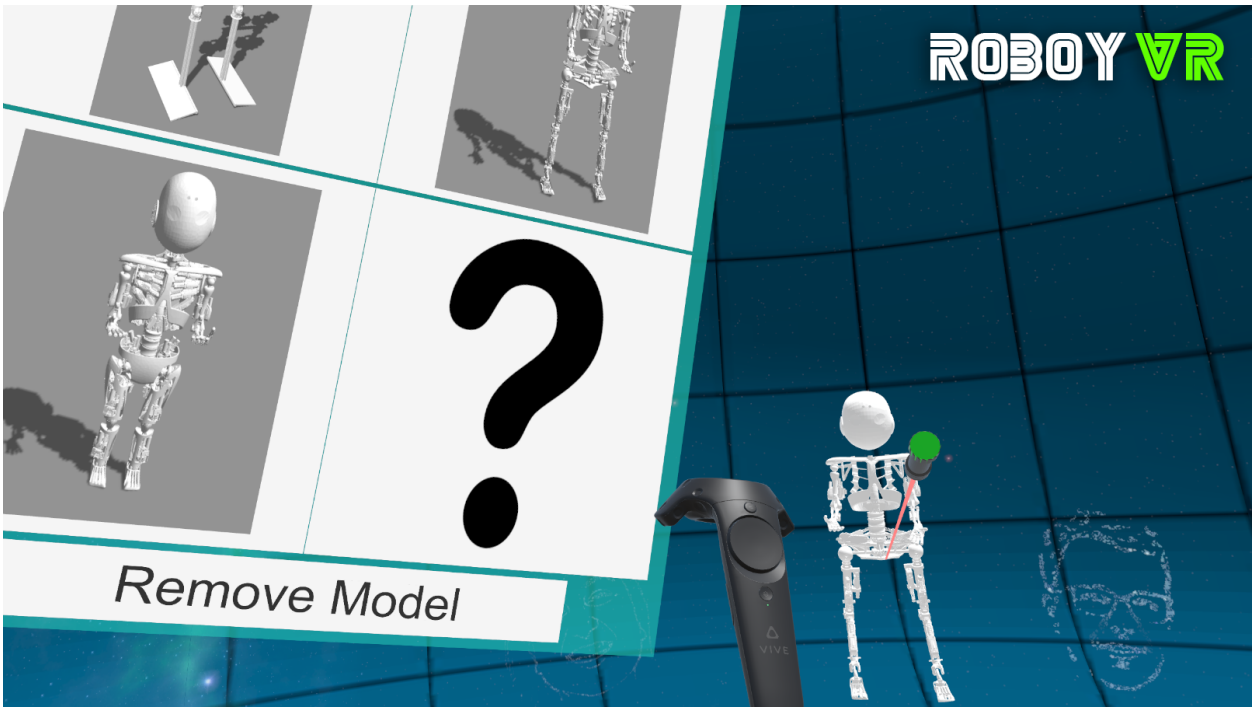


Fig. 4.11: With remove enabled, delete a model by clicking on it

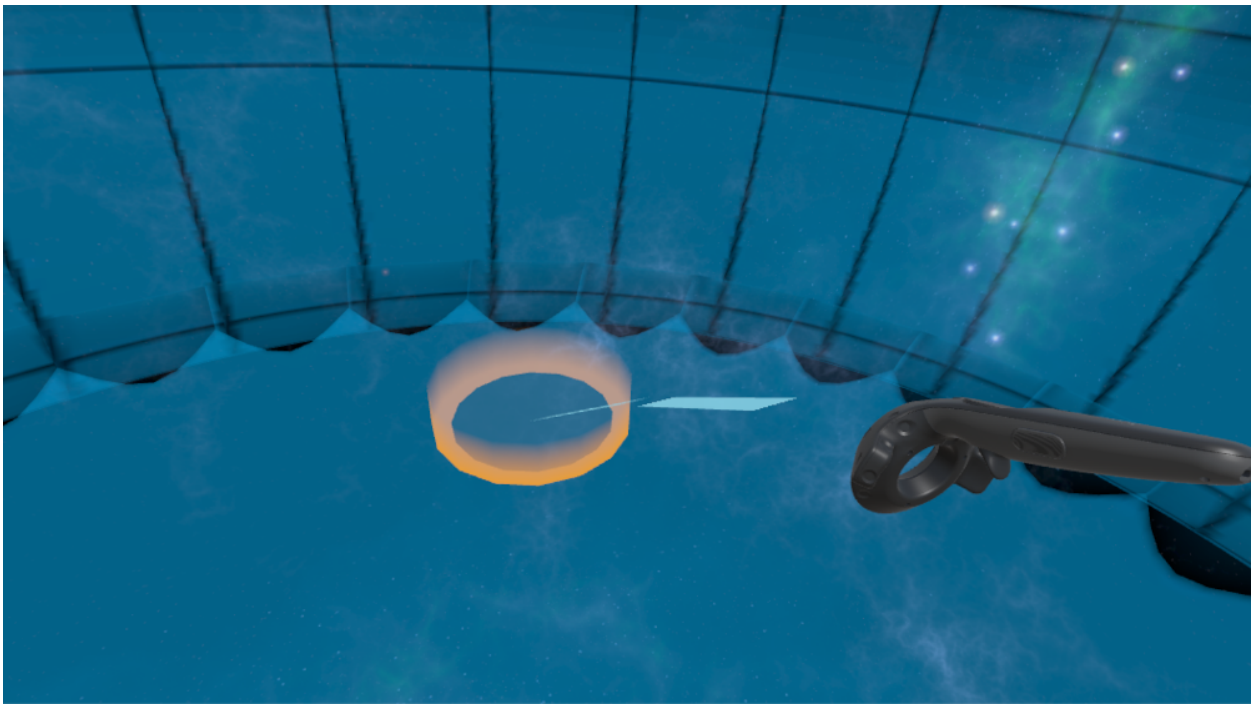


Fig. 4.12: Press down on the touchpad to teleport to a specific position.

4.8 Introduction

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

4.8.1 What is it?

BeRoboy is designed to allow the user to be in charge of Roboy and experience the world and virtual environment through his eyes. With BeRoboy you can control various different versions of Roboy. This includes a Roboy in VR, in a gazebo simulation and even the real one. The user can make Roboy move, jump'n'jive, rock'n'roll and more!

4.8.2 How does it work?

BeRoboy is utilizing the full capabilities of HTC's Vive headset and lighthouse tracking to accurately capture the user's pose. This data is then converted to determine the positions and rotations Roboy needs to adopt. Corresponding commands are then send in a format that Roboy understands and which he is able to process. After receiving those messages Roboy changes its state/ pose/ etc. When the user establishes a link with either simulated or real Roboy, BeRoboy provides video/ camera streems from the respective environment. This serves the purpose to give the user feedback in what way his actions affect the surroundings of Roboy.

4.9 User's Manual

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

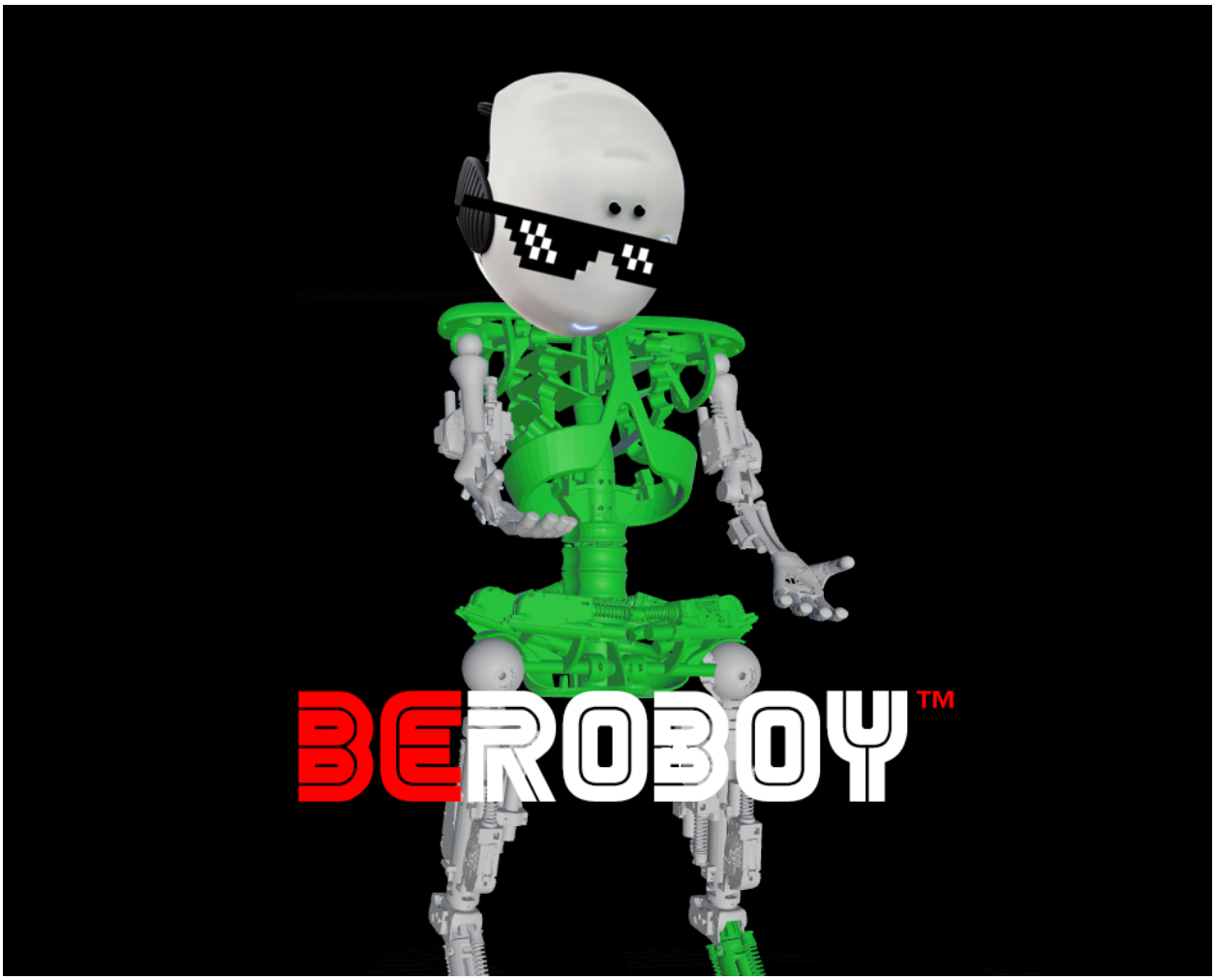


Fig. 4.13: BeRoboy is the next step in the world of virtual reality robots.

This manual will descripe the steps required to start BeRoboy.

4.9.1 Starting Gazebo

Start the Ubuntu machine and open a terminal. See further information about starting a simulation under the point **Getting Started** - this includes necessary exports before a launch file is executed.

Start the launch file which starts Gazebo with the Roboy and a Camera ROS node

```
roslaunch roboy_simulation roboy_camera.launch
```

If yout want the insert/remove feature to work, also launch this in a seperate terminal, calling all pre-simulation commands again.

```
rosrun roboy_simulation VRRoboy
```

4.9.2 Starting Unity

1. Start the unity project and set up the ROS server connection as described in **Getting Started**.
2. Start the scene.
3. After the controller assignment, you can switch between various view modes via a selection menu in the scene.

4. Enjoy!

4.9.3 View Scenarios

You can choose between the following four view scenarios, each of them offering different things to explore!

I. Gazebo Simulation

II. Real Roboy (ZED)

III. Observing Gentleman

IV. VR Roboy

4.10 Developer's Manual

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

Gazebo Simulation

Example for a launch file: This launch file would load camera.world and set also some start parameters for the gazebo simulation, for example it would start it in a not paused stated ("paused" set to "false").

```
<launch>
        <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find roboy_simulation)/worlds/camera.world"/>
        <arg name="paused" value="false"/>
        <arg name="use_sim_time" value="true"/>
        <arg name="gui" value="true"/>
```

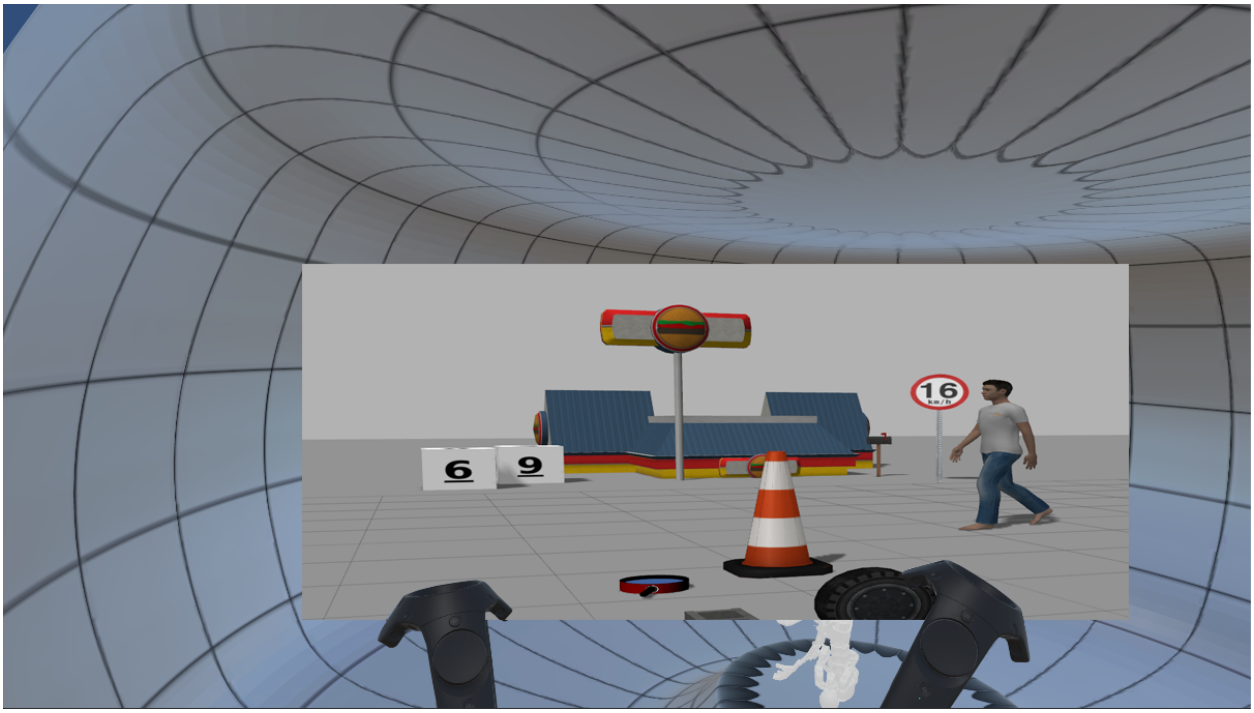


Fig. 4.14: Take control over the simulation Roboy and see what he does in gazebo.

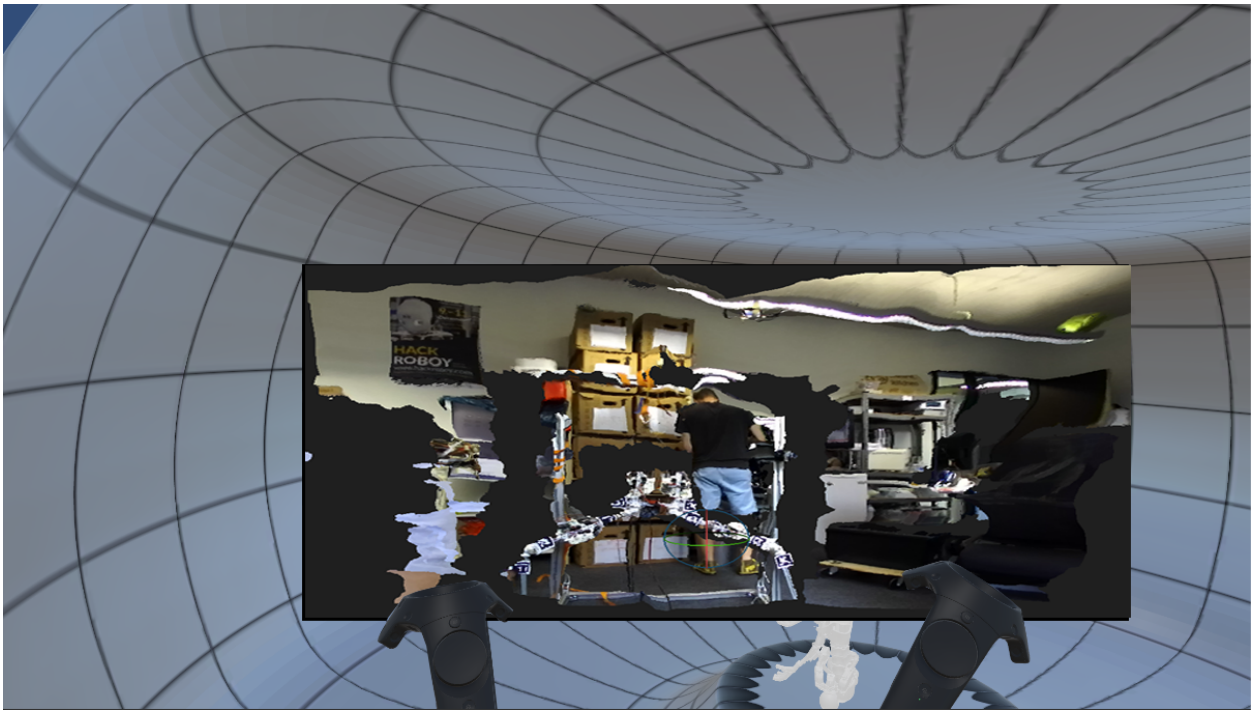


Fig. 4.15: Look through the eyes of the real Roboy and control him in real life.

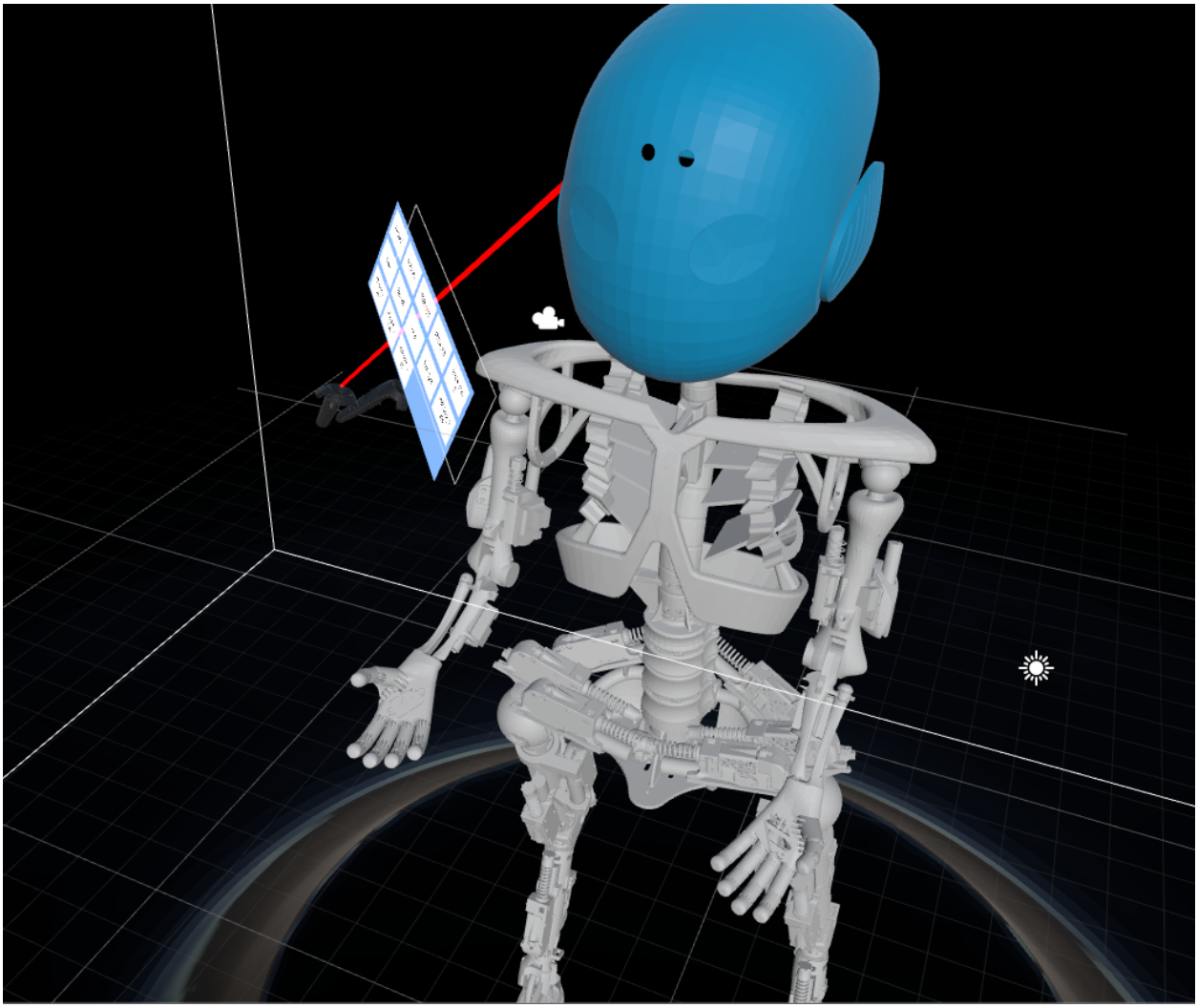


Fig. 4.16: Sit back, relax, take a look at Roboy from a safe distance and watch him do some stuff.

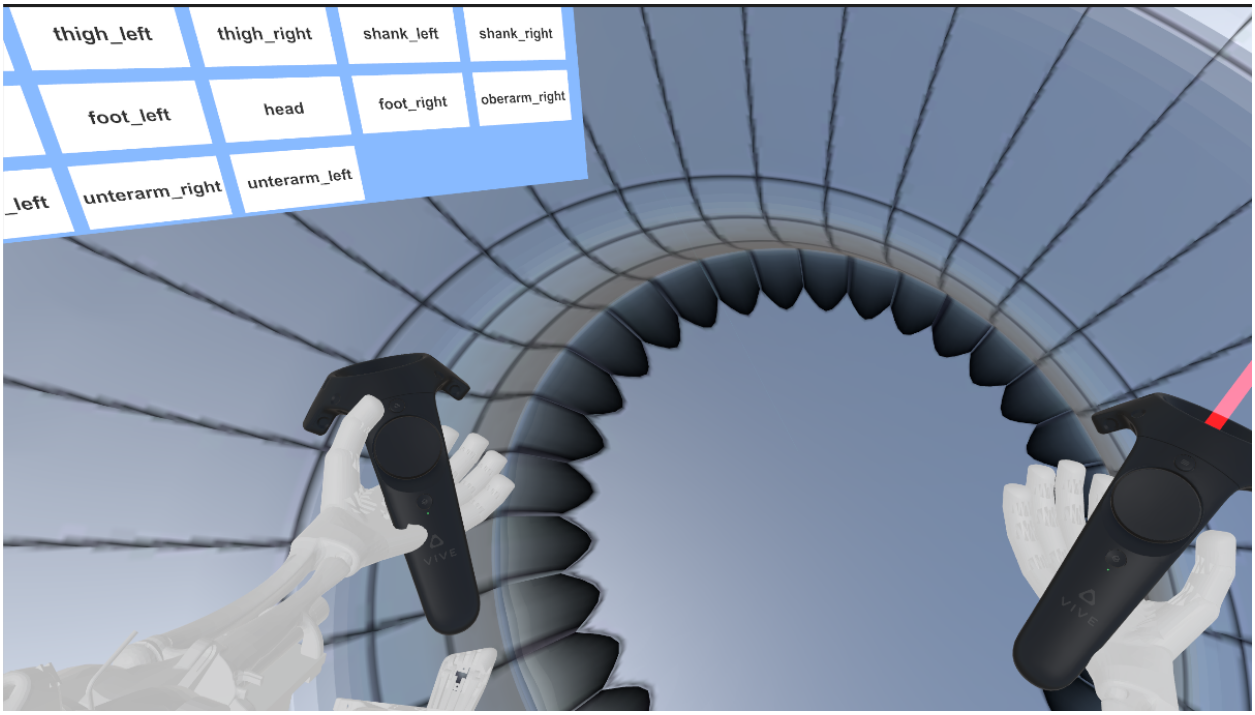


Fig. 4.17: Slip into the role of the true VR Roboy, cause mayhem or look cute, you decide.

```
        <arg name="headless" value="false"/>
        <arg name="debug" value="false"/>
        </include>
        <include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch"/>
</launch>
```

Example for a world (camera.world) file: In this case the world file contains a ground plane, the legs with upper body roboy model and a light source, a sun.

```
<world name="default">

<!-- A ground plane -->
<include>
    <uri>model://ground_plane</uri>
</include>
<!--PabiRoboy -->
<include>
    <uri>model://Roboy_with_camera_simplified</uri>
</include>
<!--Sun -->
<include>
    <uri>model://sun</uri>
</include>


<!-- Focus camera on tall pendulum -->
<gui fullscreen='0'>
    <camera name='user_camera'>
        <pose>4.927360 -4.376610 3.740080 0.000000 0.275643 2.356190</pose>
        <view_controller>orbit</view_controller>
```

```
    </camera>
</gui>
</world>
```

4.10.1 Model Configuration

If you want to see a camera feed from a gazebo simulation you need to have a *camera sensor* that captures images and publishes them via messages over a ros bridge. Those messages a standard sensor messages. You can refer to a *gazebo plugin* that has already been implemented. It is re- commended to attach this sensor to a position close to the models head because you want to its POV to maximize the POV experience. To implement such a thing, just open the model.sdf of the specific model you want to have in the simulation and add the following section.

```
<sensor type="camera" name="camera">
      <update_rate>3.0</update_rate>
      <camera name="head">
        <pose>0 1.25 0 -1.5707963267948966 -1.5707963267948966 0</pose>
        <horizontal_fov>1.6962634</horizontal_fov>
        <image>
          <width>640</width>
          <height>480</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.1</near>
          <far>100</far>
        </clip>
        <noise>
          <type>gaussian</type>
          <!-- Noise is sampled independently per pixel on each frame.
               That pixel's noise value is added to each of its color
               channels, which at that point lie in the range [0,1]. -->
          <mean>0.0</mean>
          <stddev>0.007</stddev>
        </noise>
      </camera>
      <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>0.0</updateRate>
        <cameraName>roboy/camera</cameraName>
        <imageTopicName>image_raw</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>camera_link</frameName>
        <hackBaseline>0.07</hackBaseline>
        <distortionK1>0.0</distortionK1>
        <distortionK2>0.0</distortionK2>
        <distortionK3>0.0</distortionK3>
        <distortionT1>0.0</distortionT1>
        <distortionT2>0.0</distortionT2>
      </plugin>
</sensor>
```

The *pose* determines where the camera will be looking at and which perspective it will be publishing messages from. In order to publish images the camera sensor needs a plugin attached to it, in this case its a standard plugin-in, the ros camera from the gazebo library. The *width* and *height* tag determine the *resolution* of the published images, the update rates is crucial to how many images are sent in one second (25 means, 25 updates per second).

In order to control Roboy in the simulation via ROS messages, the respective model needs to have the suiting plugin.

```
#include "roboy_simulation/BeRoboyPlugin.hpp"
#include <math.h>

using namespace std;
using namespace gazebo;

GZ_REGISTER_MODEL_PLUGIN(BeRoboyPlugin)

BeRoboyPlugin::BeRoboyPlugin() : ModelPlugin() {}

BeRoboyPlugin::~BeRoboyPlugin(){}

void BeRoboyPlugin::Load(physics::ModelPtr _parent, sdf::ElementPtr _sdf)
{
        // get the model
        model = _parent;
        // bind the gazebo update function to OnUpdate
        updateConnection = event::Events::ConnectWorldUpdateBegin(boost::bind(&
↪BeRoboyPlugin::OnUpdate, this, _1));
        // get all joints and the initial pose
        physics::Joint_V jointVector = model->GetJoints();
        initPose = model->GetWorldPose();

        // Init ros if it is has not been initialized
        if(!ros::isInitialized())
        {
                int argc = 0;
                char **argv = NULL;
                ros::init(argc, argv, "BeRoboy");
        }

        // Create ros node
        nh = ros::NodeHandlePtr(new ros::NodeHandle("BeRoboy"));
        spinner = boost::shared_ptr<ros::AsyncSpinner>(new ros::AsyncSpinner(1));
        spinner->start();

        jointCommand_sub = nh->subscribe("/roboy/middleware/JointCommand", 1, &
↪BeRoboyPlugin::JointCommand, this);
        setPosition_sub = nh->subscribe("/roboy/middleware/Position", 1, &
↪BeRoboyPlugin::SetPosition, this);
        pose_pub = nh->advertise<roboy_communication_middleware::Pose>("/roboy/
↪simulation/"+ _parent->GetName() +"_pose", 1);
        hip_sub = nh->subscribe("/roboy/middleware/DarkRoom/sensor_location", 1, &
↪BeRoboyPlugin::DarkRoomSensor, this);
        for(auto joint = jointVector.begin(); joint != jointVector.end(); joint++)
        {
                // Test if joint type is revolute
                if((*joint)->GetType() != 576)
                        continue;
                // replace whitespace with underscore in the names
                string _modelName = model->GetName();
                string jointName = (*joint)->GetName();
                string _jointName = jointName;
                boost::algorithm::replace_all(_modelName, " ", "_");
                boost::algorithm::replace_all(_jointName, " ", "_");
                joints.push_back(jointName);
                jointAngles[jointName] = (*joint)->GetAngle(0).Radian();
```

```
        }
}

void BeRoboyPlugin::publishPose()
{
        roboy_communication_middleware::Pose msg;
        for(auto link:model->GetLinks()){
                msg.name.push_back(link->GetName());
                math::Pose p = link->GetWorldPose();
                msg.x.push_back(p.pos.x);
                msg.y.push_back(p.pos.y);
                msg.z.push_back(p.pos.z);
                p.rot.Normalize();
                msg.qx.push_back(p.rot.x);
                msg.qy.push_back(p.rot.y);
                msg.qz.push_back(p.rot.z);
                msg.qw.push_back(p.rot.w);
        }
        pose_pub.publish(msg);
}

void BeRoboyPlugin::JointCommand(const roboy_communication_
↪middleware::JointCommandConstPtr &msg){
        for(uint i=0;i<msg->link_name.size();i++){
                jointAngles[msg->link_name[i]] = msg->angle[i];
        }
}

void BeRoboyPlugin::SetPosition(const roboy_communication_
↪middleware::PositionConstPtr &msg){
        math::Vector3 pos(msg->x, msg->y, msg->z);
                gazebo::math::Pose p(pos, initPose.rot);
        initPose = p;
}

void BeRoboyPlugin::DarkRoomSensor(const roboy_communication_
↪middleware::DarkRoomSensorConstPtr &msg)
{
        int hipIDPos = -1;
        for(int i = 0; i < msg->ids.size(); i++)
        {
        // hip id of the sensor should be 4
        if(msg->ids[i] == hipID)
        {
                hipIDPos = msg->ids[i];
                break;
        }
        }
        if(hipIDPos == -1)
        return;

        // move the position of the model
        math::Quaternion modelRot = model->GetWorldPose().rot;
        math::Vector3 modelPos = math::Vector3(msg->position[hipIDPos].x, msg->
↪position[hipIDPos].y, msg->position[hipIDPos].z);
        initPose = math::Pose(math::Pose(modelPos, modelRot));
}
```

```
void BeRoboyPlugin::OnUpdate(const common::UpdateInfo &_info)
{
        // make the model stationary
        model->SetWorldPose(initPose);
        // set velocity and force to zero and force for every saved joint and set␣
↪angle to saved value
        for(auto it = joints.begin(); it != joints.end(); it++)
        {
                model->GetJoint(*it)->SetVelocity(0, 0);
                model->GetJoint(*it)->SetForce(0, 0);
                model->GetJoint(*it)->SetPosition(0, jointAngles[*it]);
        }
        initPose = model->GetWorldPose();
        publishPose();

}
```

4.10.2 Unity Scene

In Unity you need to establish a *Rosbridge* in order to be able to communicate with the various types of Roboy, e.g. the simulation one or the real one. Both of them are sending their camera feed as *Image messages* of the type sensor_msgs/Image. Therefore you need also a suiting *subscriber* in Unity to be able to receive the messages correctly and parse them afterwards in the right manner.

Image message in Unity

```
namespace ROSBridgeLib
{
        namespace sensor_msgs
        {
                public class ImageMsg : ROSBridgeMsg
                {
                        ...
                        ...

                        public ImageMsg(JSONNode msg){...}

                        public ImageMsg(HeaderMsg header, byte[] data){...}

                        public byte[] GetImage(){...}

                        public static string GetMessageType(){...}

                        public override string ToString(){...}
                        public override string ToYAMLString(){...}
                }
        }
}
```

Image Subscriber in Unity

```
namespace ROSBridgeLib
{
        public class RoboyCameraSubscriber : ROSBridgeSubscriber
        {
                public new static string GetMessageTopic()
```

```
                {
                        return either "/roboy/camera/image_raw" or "/zed/rgb/image_
↪raw_color"
                }

                public new static string GetMessageType()
                {
                        return "sensor_msgs/Image";
                }

                public new static ROSBridgeMsg ParseMessage(JSONNode msg)
                {
                        //ImageMsg from sensor messages lib
                        return new ImageMsg(msg);
                }

                public new static void CallBack(ROSBridgeMsg msg)
                {
                        ImageMsg image = (ImageMsg)msg;
                        //ReceiveMessage respectively either for the simulation or␣
↪zed image
                        BeRoboyManager.Instance.ReceiveMessage(image);
                }

        }
}
```

After getting the ros bridge connection right and being able to receive image messages as well as reading them correctly the camera feeds should be displayed and rendered at at suited position. For this purpose this unity scene uses a *canvas in camera space*. Attached to this canvas are various image planes (unity ui images) that can wrap up the received messages.

There is also a *View Selection Manager* embedded to the BeRoboy™ scene, it is used to fluently switch from one view to another. This manager is responsible for the procedures after a button on the *3D selection menu* is pressed. When a certain button is invoked by onClick() the state of various different game objects needs to manipulated (mostly enabling or disabling them). A View Selection Manager always needs the desired references in order to set them, if they not already come preconfigured.

Receiving Images Info

Depending on what images you want to receive, you need to set the size of the color arrays in the BeRoboyManager class. m_colorArraySample = new Color [width*height]

In addition you also need to set the texture size in Awake() respectively m_texSample = new Texture2D(width, height)

4.11 Use Cases

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

The following use cases should demonstrate how the Unity side of BeRoboy deals with certain scenarios, it should show further which procedure calls are happening in the scene, that a developer needs to be aware of.

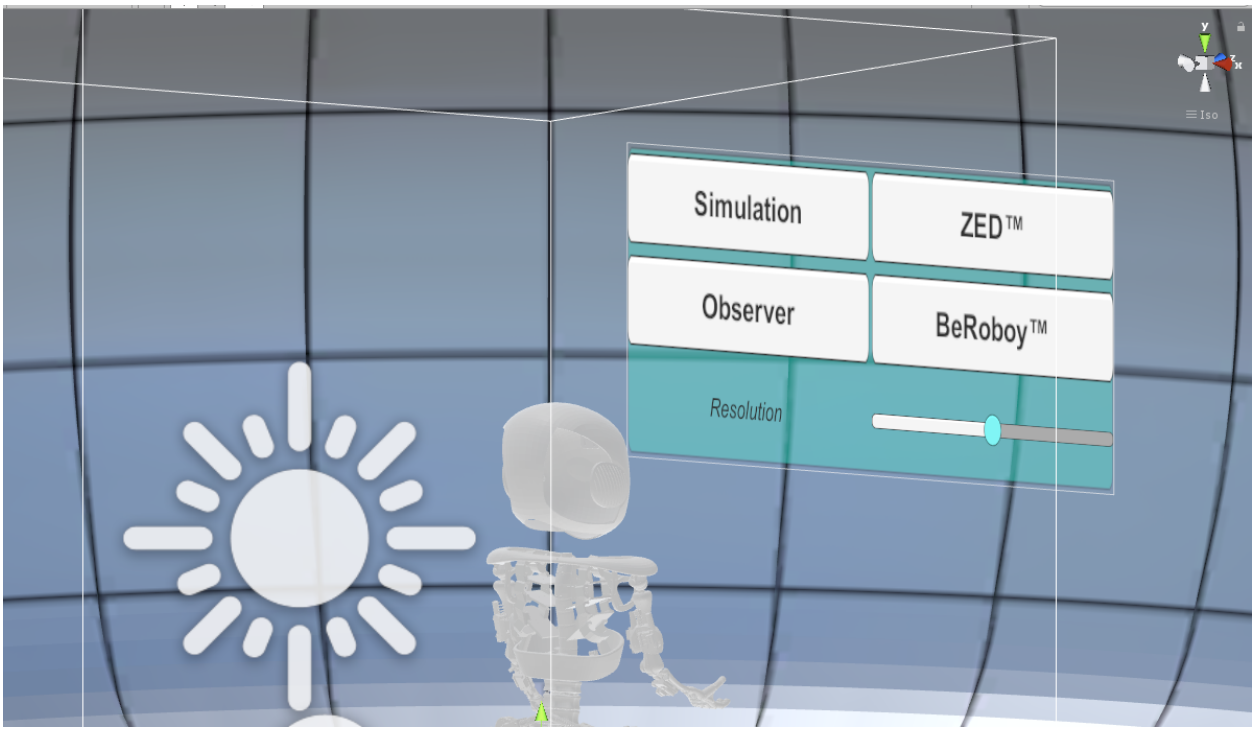


Fig. 4.18: After clicking on one of the buttons, the View Selection Manager takes the necessary steps to change to the respective view.

4.11.1 Switching between views

4.12 Current State

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

As of now only virtual Roboys can be manipulated and controlled which are serving as subjects in the gazebo simulation. Some positions and rotations are not displayed properly, as translations and further positions and values changed.

4.13 Introduction

4.13.1 Usage

The Virtual Reality user interface will display all given and desired data in a structured environment to help both developers and external visitors to gain further insight into Roboy and how he works. For the developer side, it is important to display the data in a coherent way to clearly communicate the current state of Roboy to aid in implementation and debugging scenarios. Goals for the visitor include designing a visually appealing interface which does not overload the user with unnecessary and misleading information, but provides selected information and explanations to satisfy the visitor's interest. Important aspects include structuring and grouping the given data in sets, between which the user can change and which he can activate dynamically, providing an intuitive control system which does not need further explanation and visualizing the given data in a clear and understandable way.

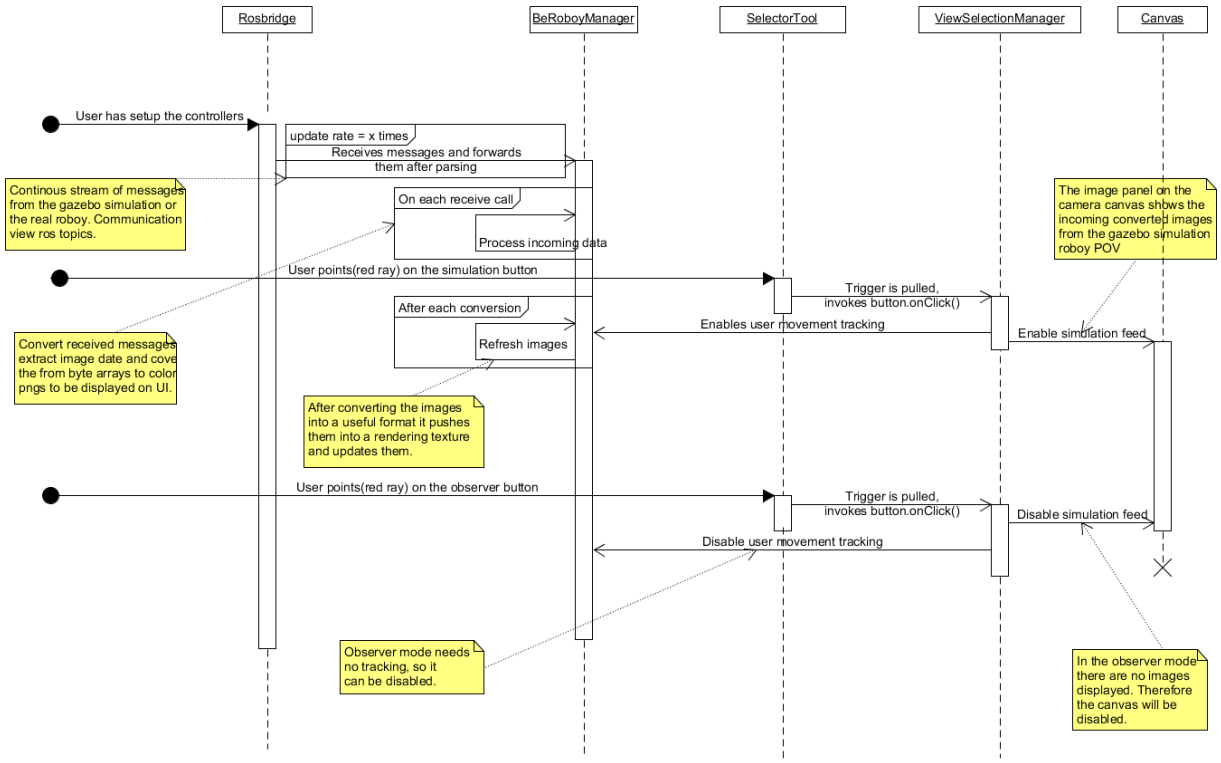
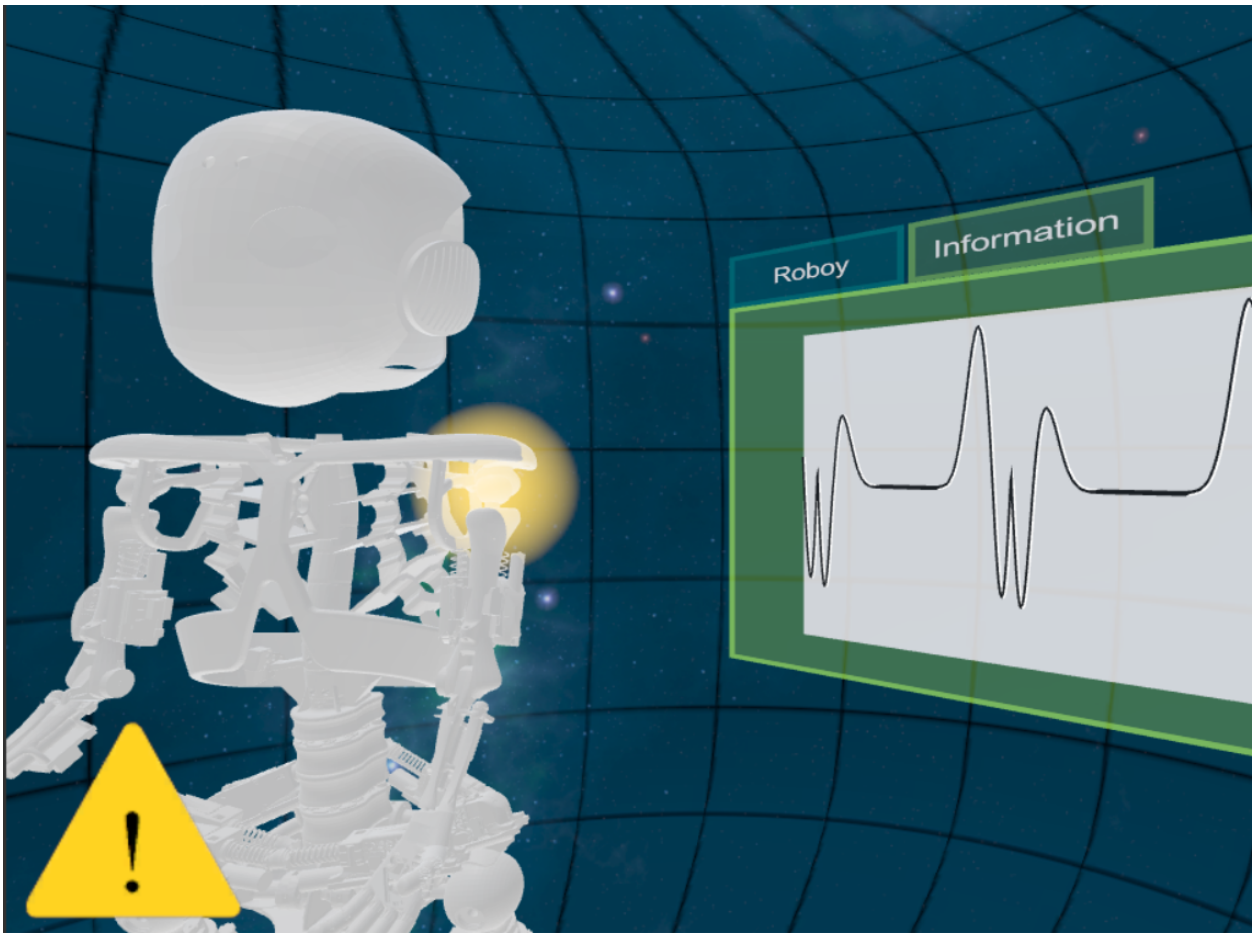


Fig. 4.19: The user first switches to the simulation view, but gets all exhausted and switches to the relaxing observer view mode.


Roboy
Information

4.13.2 Structure

In general, the Scene contains two types of objects:

- Scene related Objects: Roboy, the Background, the Camera
- UI related Objects: Canvases, screens, container objects, UI elements such as panels, buttons, images

The UI has three main layers:

- Front-end: Containing all UI objects and displaying data
- Core: Containing logic, updating methods and operating on given data set
- Back-end: Provider of data through a connection to ROS

4.13.3 Current Implementation

See the user manual for further information.

4.14 User's Manual

4.14.1 Set-up

The Scene can be run in the Unity Editor. Simply double click on the UIScene, Unity will start and load the scene. The play button in the top-centre starts it. The scene works with and without the connected SteamVR headset and controller, though it does not change the camera or control panels without these interaction methods, as these are the only input method. The Screen is displayed in a window in Unity and in the glasses. Since there is no connection to ROS so far, this aspect does not need to be considered. Both the play buttons as well as the game window can be seen highlighted in the screenshot below.

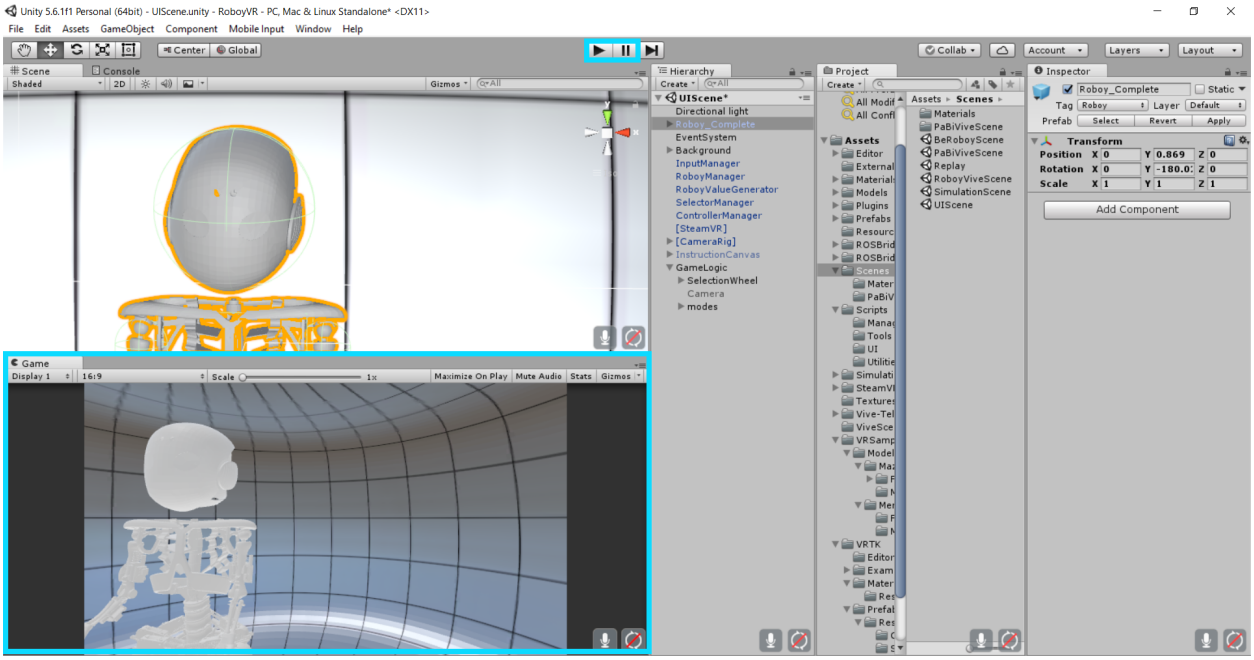


Fig. 4.20: Unity Editor

4.14.2 SteamVR

The hardware of the computer needs to support Virtual Reality applications, additionally SteamVR needs to be installed. For further information on how to set up the VR headset and controller, follow the instructions.

4.14.3 UI Features

The user interface as of now implements different modes focussing on certain data sets. - The Overview provides general information about Roboy, a graph plotting a heartbeat-like function and displays notifications concerning the state of roboy. These relate to a cretain body part, which is highlighted using a halo and a warning sign near the respective area. Different types include error, warning, debug and information notifications. - The Middleware mode visualizes tendons as colored lines which change color depending on the applied force and update their positions depending on the movement of the robot. - In Control mode, additional information about notifications is given. A list provides all currently detected status messages, which display additional information when clicked. - The cognition mode does not contain any visualizations of specific data types. Only template screens are provided in this sceen.

The user can initially decide, which controller is his main controller by using the trigger at the underside of the controller. Controllers are used to help the user navigate and control the environment as well as interact with the data sets. A selection wheel is used to change between modes, all other interaction methods only include point-and-click and click-and-hold using a raycaster displazing a red beam. Not all UI elements are interactive just yet, but this is to be changed in future updates. The main controller touchpad can then be used to change between different modes (Cognition, Overview, Middleware and Control). By turning the head with the headset, the camera can be rotated. By moving around the room, the camera position can be changed. Beware of possible obstacles and boundaries given by the physical surroundings.

.

4.15 Developer Manual

4.15.1 General

The UI design goal was to create a modular and robust UI which does not rely on continuous data input. Due to the fact, that the Virtual Reality scenes will later be merged and therefore the setup will change, it was advantageous to not create one definite UI structure already containing all the desired elements. Instead, a modular, easy-to-adjust base was designed, which can be integrated in other scenes without much effort.

Scripts: User Interaction Scripts: These scripts were already provided by the previous implementation and only needed to be adapted at most. This is the case for the SelectorTool, which now additionally provides information when a button is triggered and held and later released.

Game objects: General game objects, which belong to the scene but not the UI, include Roboy, a cave and the camera rig containing the SteamVR controllers and headset.

4.15.2 Use Case

The following use case depicts an example activity, where a user changes the currently selected mode by touchpad.

4.15.3 UI Implementation and Structure

The UI can be structured in three basic layers:

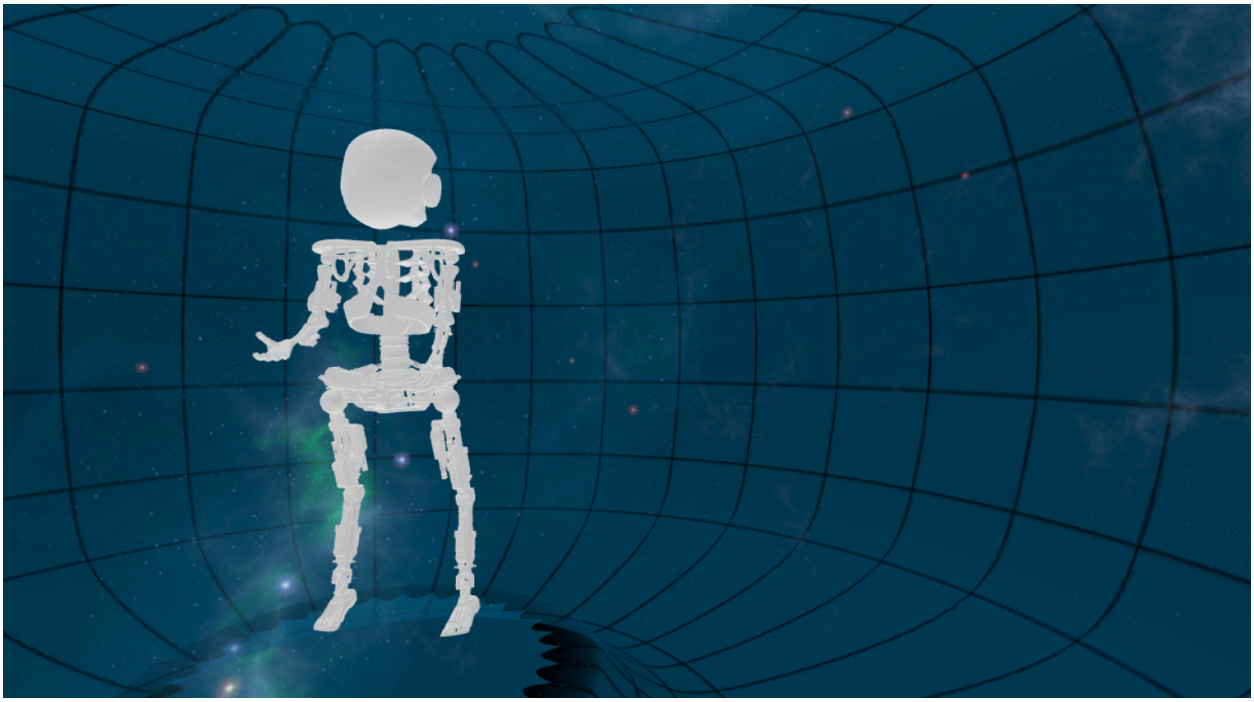


Fig. 4.21: Roboy in a cave

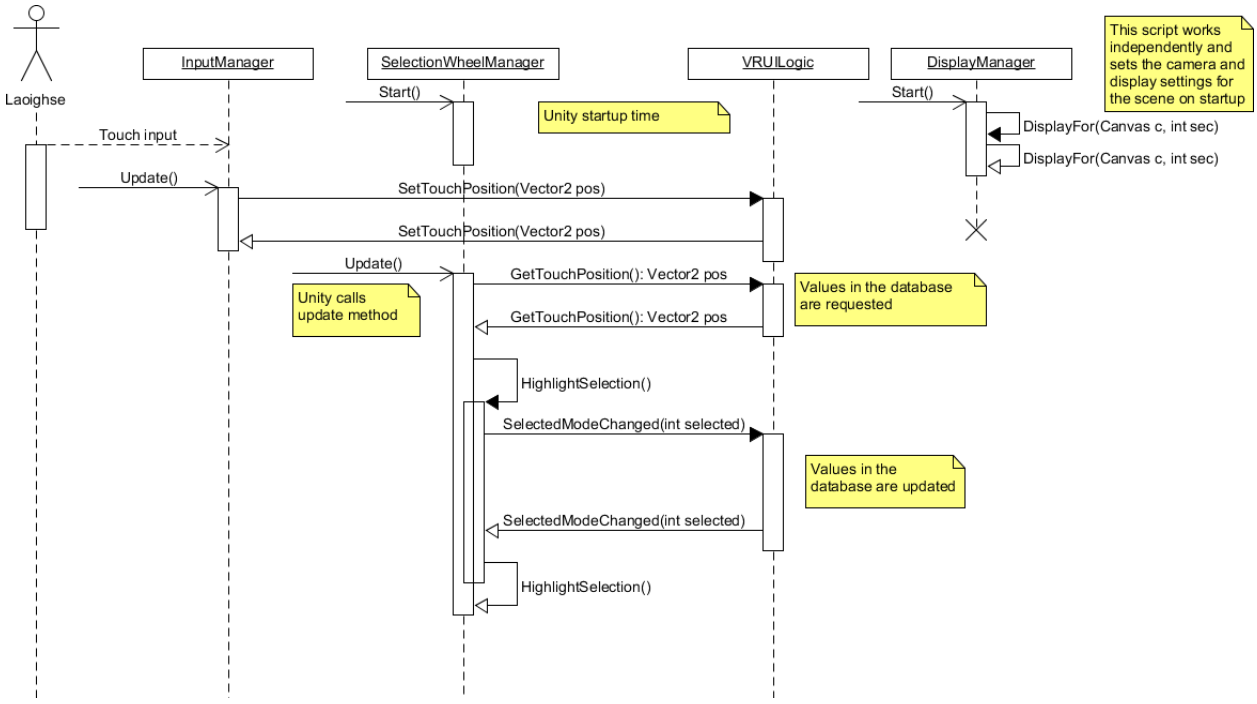


Fig. 4.22: Sequence of method calls after user touch input initialized by Unity's update function

Front-End

In this layer, objects, modes, and their respective items covering screens, labels, screen overlays and UI are contained. It serves as the frontend towards the user. The respective UI elements display the data they are given, but do not actively change or adapt to changes.

Scripts: DisplayManager: This script automatically detects the number of connected displays on startup and uses the first to display the main camera on the main screen, and second camera on the latter. In case only one screen is found, it continues with the normal setup. It needs to be noted that the SteamVR glasses are not considered to be a Display by Unity.

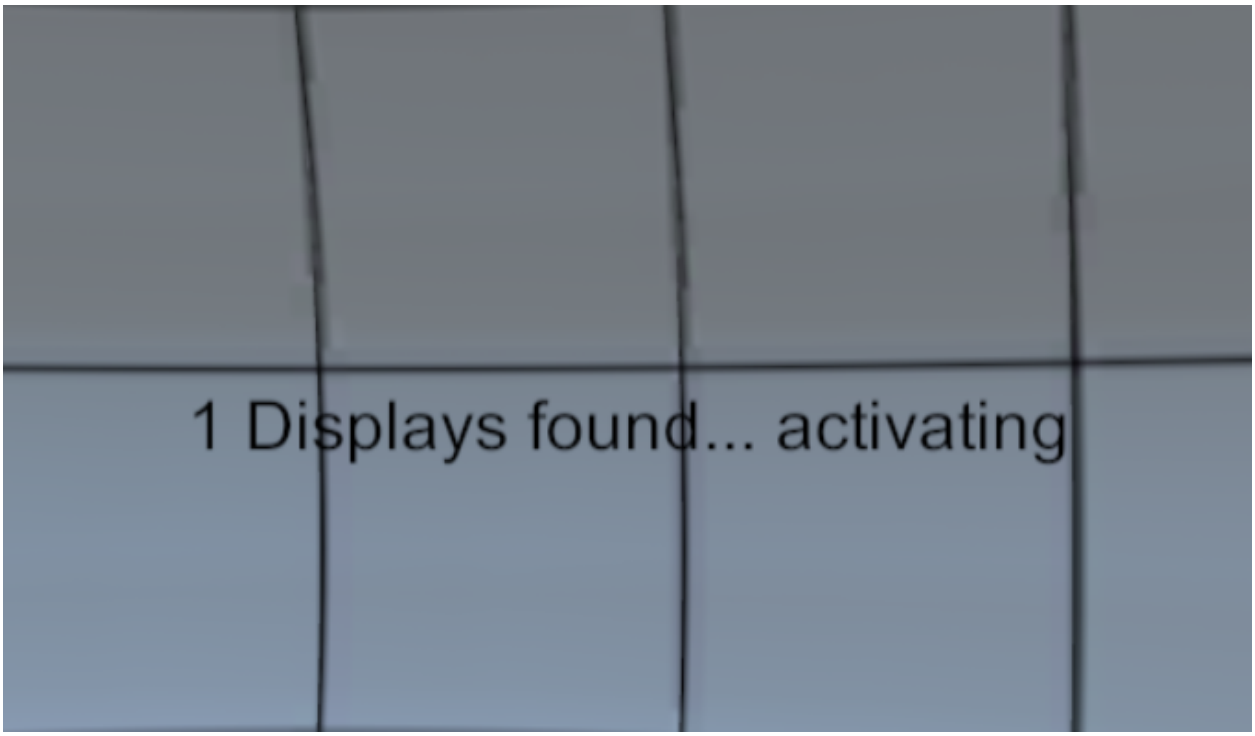


Fig. 4.23: Display message

ModeManager: Four different scripts manage the displayed data in the respective mode. They are attached to the respective game object which represents the mode and contains all mode-related elements. The scripts are:

MiddlewareManager: Since tendons automatically display themselves, no other functionalities are implemented yet.

OverviewManager: This Manager provides an examplary hearbeat function and plots it.

ControlManager: This Manager provides a list of notifications on one of the panels and adds new list items as soon as it is notified of new notifications.

CognitionManager: Since no data type, which this mode would handle, is implemented yet, no functionalities are provided.

ScreenTabManager: This script manages the correct display of the tab contents, since the objects may overlap or be covered or interfer with each other. It deactivates all content, that is not rendered last, the last tab page is fully activated.

RayOrder: This script is needed to adapt the order of canvases and the ray visible from the selector tool. The attribute in question cannot be changed in the Editor, that's why this script was created.

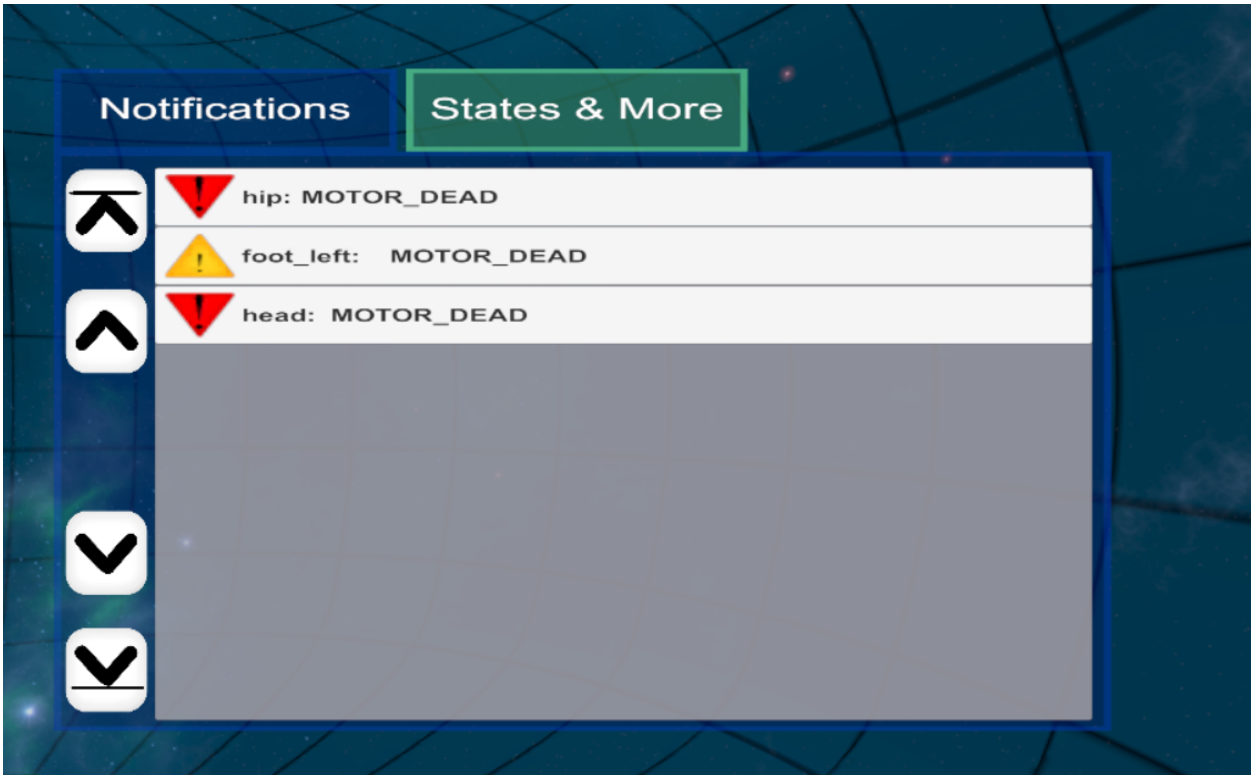


Fig. 4.24: List of notifications

RotateWithView: This script rotates the specified component around a predefined pivot point. The rotation that is applied comes from the y axis of the user's headset, which leads to the object remaining at the same height but rotating in a horizontal circle around the point.

ScrollViewFunctionalities: For a ScrollView component, this script provides scrolling functionalities, including events and the methods performing the action. These are to be triggered by buttons which shall immplement these functionalities. It needs references to the scrollView object that is to be moved. Only vertical scrolling is implemented.

Notification: This script is attached to gameobject and together they represent a notification object. It contains all needed attributes and functions.

NotificationManager: Since some notifications and visualization methods always need to be displayed (no matter which mode) this script handles the display of the icons and halos created for each notification. It is attached to a UI game object which is always enabled.

NotificationListButton: This script initializes all values and fields of the item in the notification list and additionally manages OnClick() behaviour where it creates an additional info screen for the notification. This is currently in use in the control mode.

AdditionalInfoScreen: This script initializes all values and fields and manages the functionalities of the additional screen that is provided in control mode.

Tendon: This script is attached to a gameobject and together they represent a tendon object. It contains all needed attributes and function.

ExampleFunctions: This script proides exemplary data to test certain visualization techniques and the UI in general. This includes four tendons and repeatedly spawned notifications for now.

Game Ojects: modes: this empty game object contains all modes the user can choose with the selection wheel. These

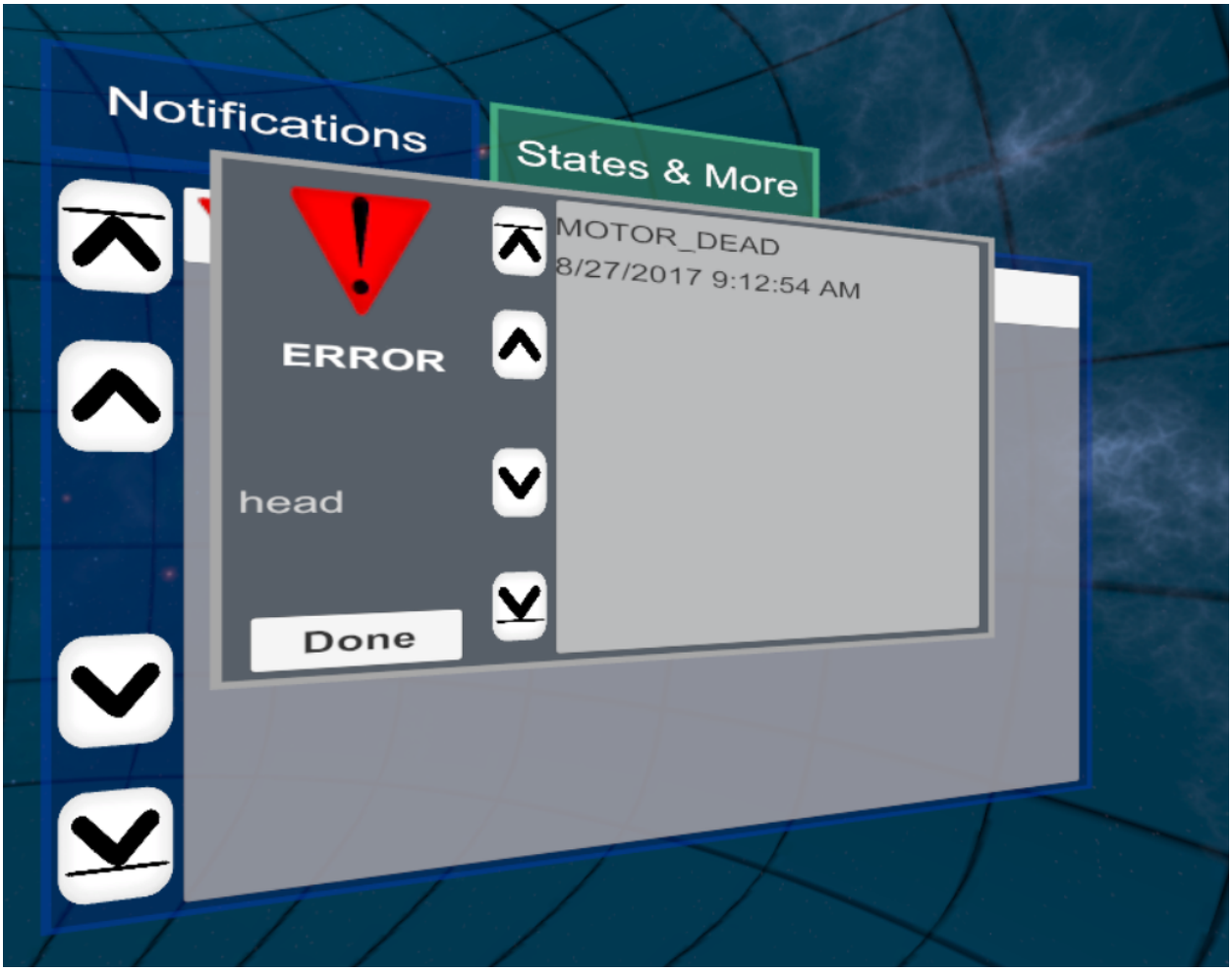


Fig. 4.25: Additional screen displaying extended information about a notification

are dis- and enabled dynamically by the script UILogic.

Canvases: Each mode contains an individual canvas which is activated together with the parent (container) object. Canvas Render "Screen Space - Overlay" needs to be selected and the in-game camera belonging to the VR headset to display the Camera there.

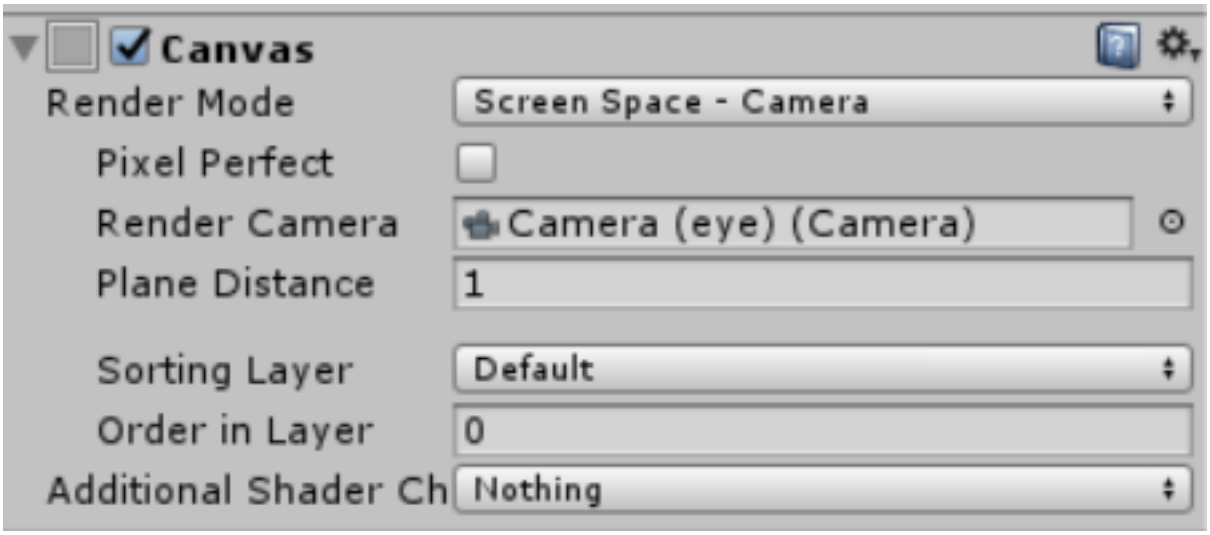


Fig. 4.26: Canvas settings for VR headsets

Note that even though the canvas is stretched to fit the screen size, the display of the headset extends further than the user's view frustrum. In the picture below, the original canvas size can be seen as well as the actual view frustrum, which the user can comfortably perceive without too much strain on the eyes.

Notifications: Theses are stored in a notification container. Notification visualization is managed for each body part in the BodyPart.cs script, as multiple errors, warnings and so on can be linked to one gameobject.

Tendons: These are visualized using a lineRenderer. All tendons are stored in one container object, which itself is a child of the middleware mode.

Skyboxes: Two different skyboxes were created to be used as the background: the Roboy skybox containing Roboy as a constellation in the night sky, RoboyAngels contains the faces of all Roboy angels. They can be set in the VRUILogic.

Core

This layer covers the UI logic and certain modemanager. It displays the selected modes, provides the frontend with the given input, informs subscriber of certrain topics of changes.

Scripts: SelectionWheelScript: This script is attached to a gameobject within a canvas, which will be disabled in the beginning. Additionally, all the children of the component are realigned to fill the selection wheel according to the number of elements. The script constantly checks for input when activated. As soon as input is detected, it enables the canvas to display the wheel and all the child objects. These are rotated on a circle according to the position of the sensed input on the controller. The controller can be set in the public variable Controllerindex. The placement on the circle, where the element should be selected, can be changed in the public variable selectionIndex. This index specifies the index within the number of game objects, which shall be selected. It starts at 12 o'clock and rotates clockwise. Since the script is general in implementation and usage, it can be used multiple times under different occasions.

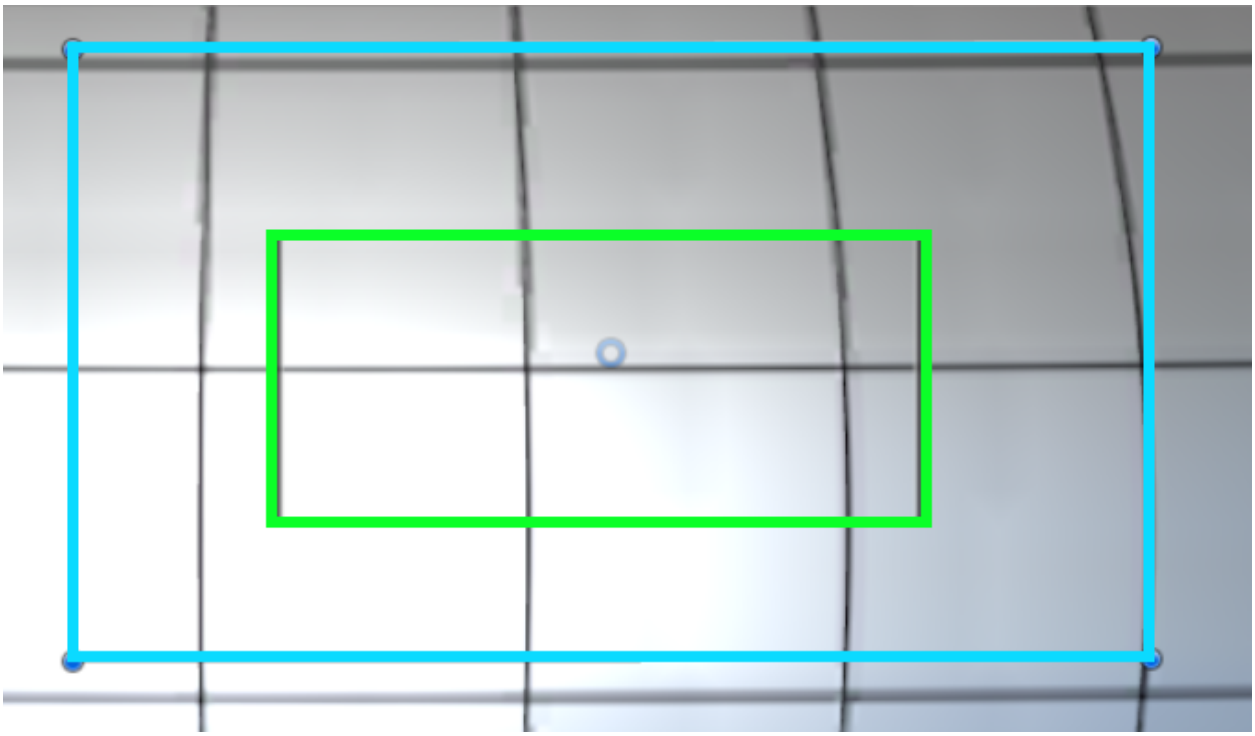


Fig. 4.27: Canvas size in blue and view frustrum in green

VRUILogic: This script operates as a database for important game values. Due to its Singleton implementation, there are no dublicates, the data is not concurrent, it is always accessible and all functions can use it as a data platform. It is designed to act passively, it does not contain an Update() function and does not actively request data. Other functions and instances can set and get the desired data. This design choice was made because it assures modularity of the respective elements, both front and back-end. This way, in case parts of the front- or back-end are no longer necessary or fail, no other components are affected. This design proved to be challenging when considering the age of the given data when later used. Functions requesting data are not presented with changes, only the current stage, which makes updates rather tedious. Therefore, the Observer-Pattern was used to inform components which need to be updated on changes. These previously added themselves to a list of subscribers. Additionally, other values such as user input and textures for notifications are provided to all other components. The textures are user-defined as can be seen in the following picture.

DummyStates: This class provides the enumerations used for notifications. This needs to be synchronized with the Control team and their error detection states and types.

Game Objects:

UILogic: This empty game object is not displayed, but contains all relevant UI components as child objects.

modes: This game object contains all mode objects, which function as containers themselves.

SelectionWheel: This object contains all components needed to display the selection wheel. This includes a canvas, a background, a wheel base image, the wheel objects and certain scripts.

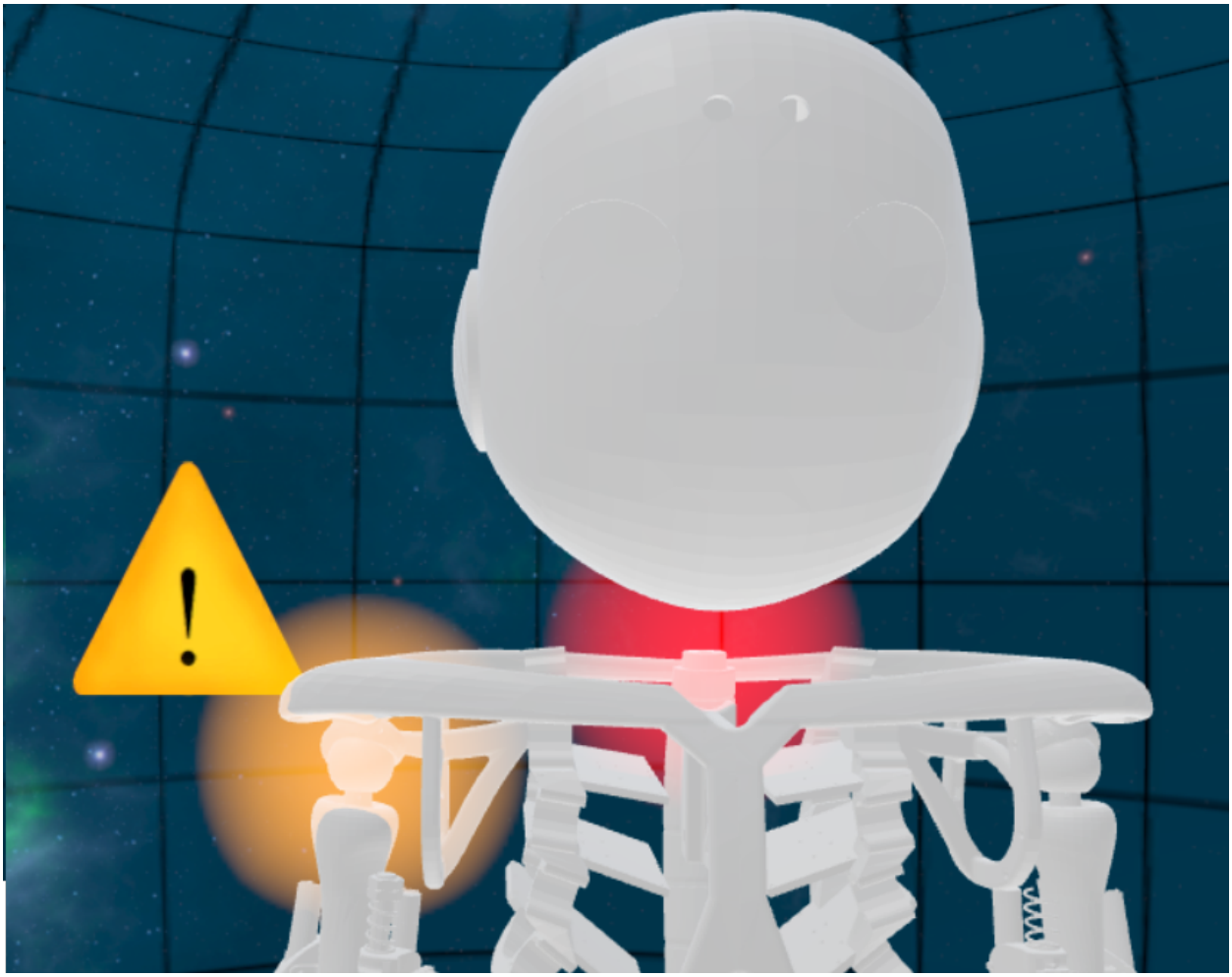


Fig. 4.28: Notification visualized using a halo and icon near concerned area

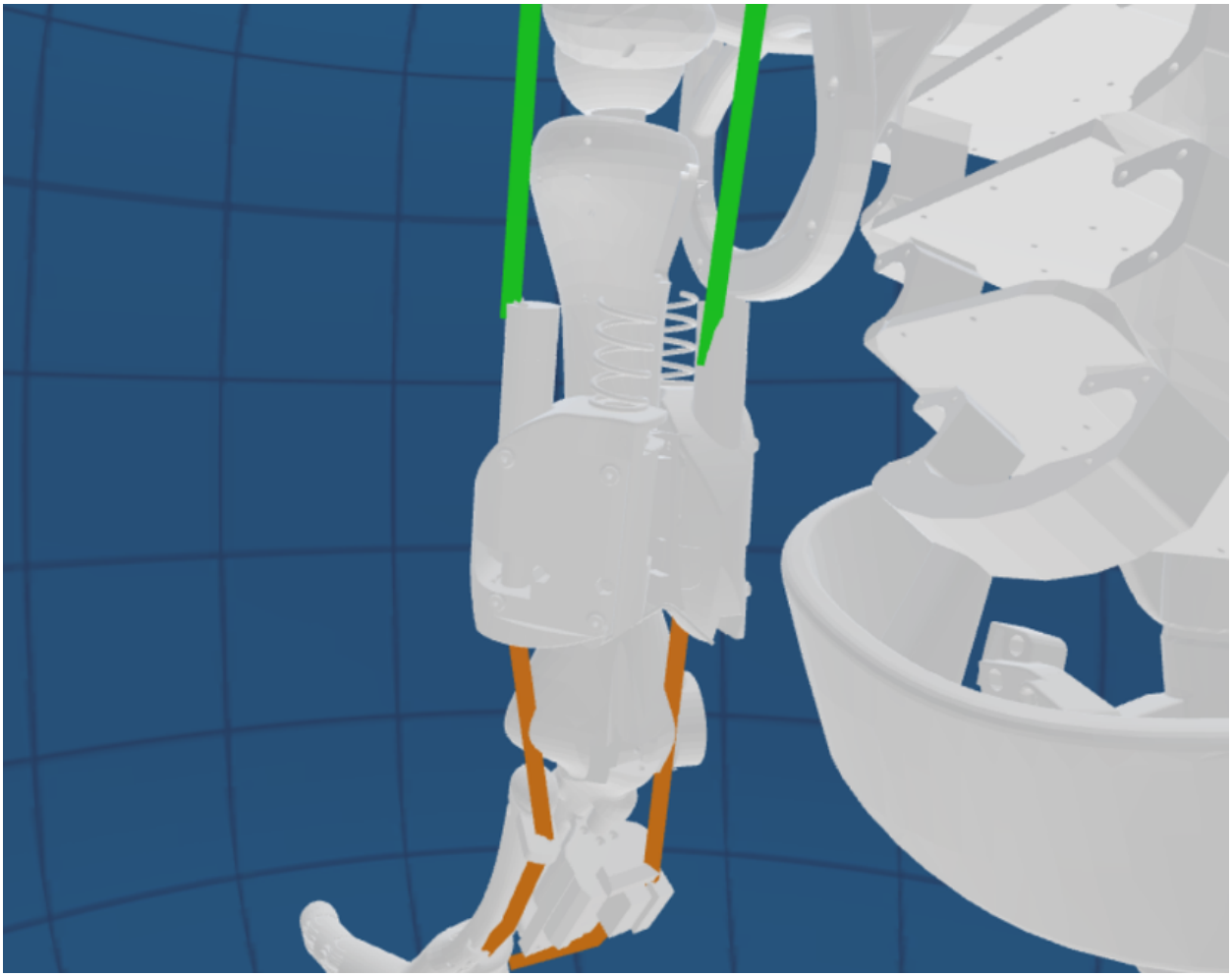


Fig. 4.29: Four exemplary tendons with different forces applied

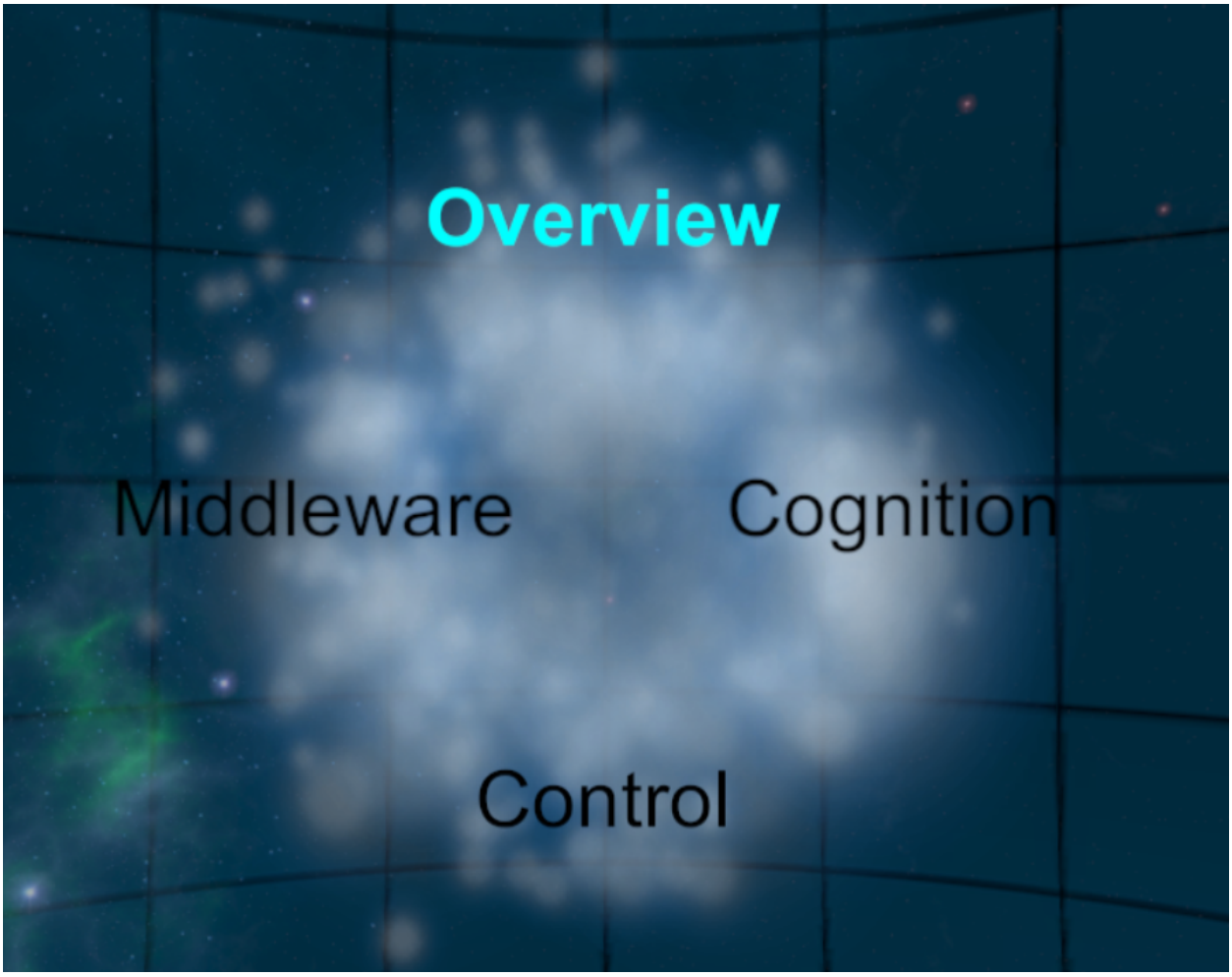


Fig. 4.30: Selection wheel with four options and Overview selected

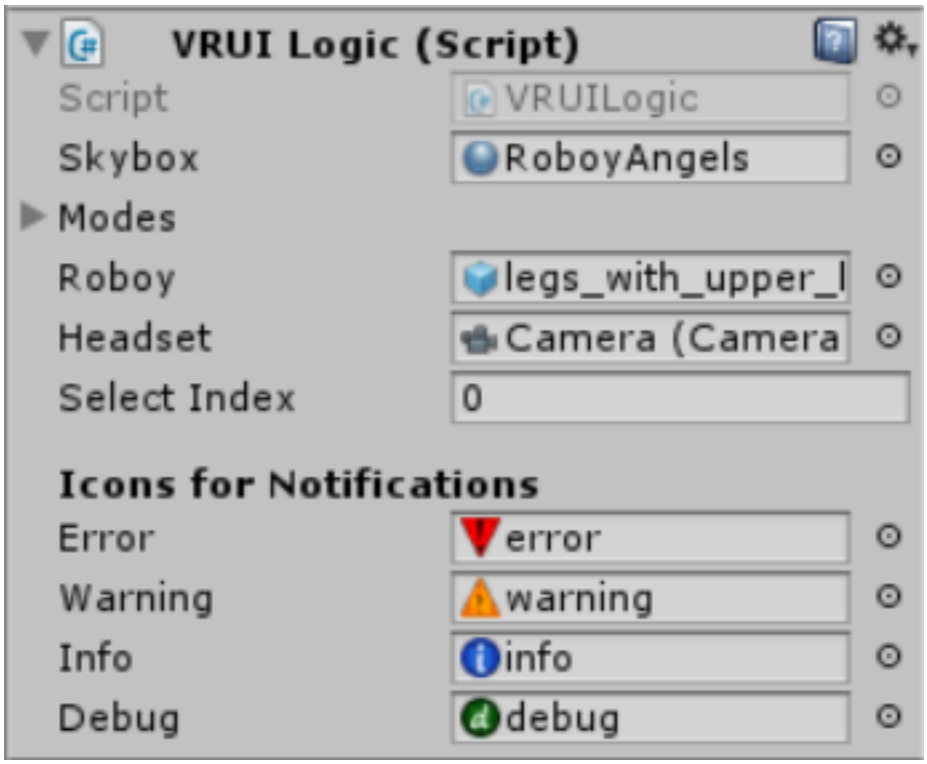


Fig. 4.31: VRUILogic script with all fields defined

Fig. 4.32: Component hierarchy in object view

Back-End

The back-end connects the Unity application to external components providing the data that is to be visualized. ROS, short for Robot Operating System, is the framework used to communicate Roboy's data clustered in topics accross different platforms and computers. An Ethernet connection is used for entities to either publish or subscribe to certain topics hosted on a server. The connection to Unity is implemented using the plug-in ROSBridge, which provides subscribing and publishing functionalities and defines standard message types.

Additional message types were defined to receive the needed data and parse it access the tendon and notification data. These include:

TendonInitialization.msg: This message is called once when a tendon is initialized. Therefore, all wire-points, a list of body parts each point is connected to, the maximal applicable force and the tendonID are sent.

TendonUpdate.msg: As soon as the tendon is initialized, only the force value needs to be updated. Therefore, only tendonID and the force value need to be sent.

Notification messages for different notifications types: Each notification is created using the same values and attributes as of now. This includes the concerned body part, the state that is to be communicated, a message and additional content, the latter two both in string form.

For each message type, a specific subscriber had to be defined, such as TendonUpdateSubscriber, TendonInitializationSubscriber and so on. Each of these script subscribes to a certain topic and listens for the dedicated message type. As soon as a new one is received, it is parsed and the information inserted into the database.

For all subscribers to work properly, they needed to be attached to a game object which also holds a ROS Object component. This effectively informs the plug-in, that these subscribers are to be activated when starting the application.

Game Objects:

BackendSubscriber: This component containas all subscribers as well as the ROS Object component mentioned beforehand.

ROSBridge: This component facilitates the communication with the ROS Server. To function properly, the IP-address needs to be updated manually to the current address where the server is hosted.

4.16 Implementation Documentation

The implementation can be divided in the three main categories front-end, core and back-end. Due to internal RestructuredText issues, some classes cannot be displayed here. Therefore, please refer to the Git-Repository of this project. Most scripts related to the State Visualization can be found here.

4.17 Introduction

4.17.1 What is it?

Roboy can be moved around in the virtual environment. The user can grab him, pull and push Roboy and shove him around. Roboy adapts his position based on a physical simulation of his body and the applied forces. He is glued to the ground to prevent him from crumbling to the ground and flying around in the room.

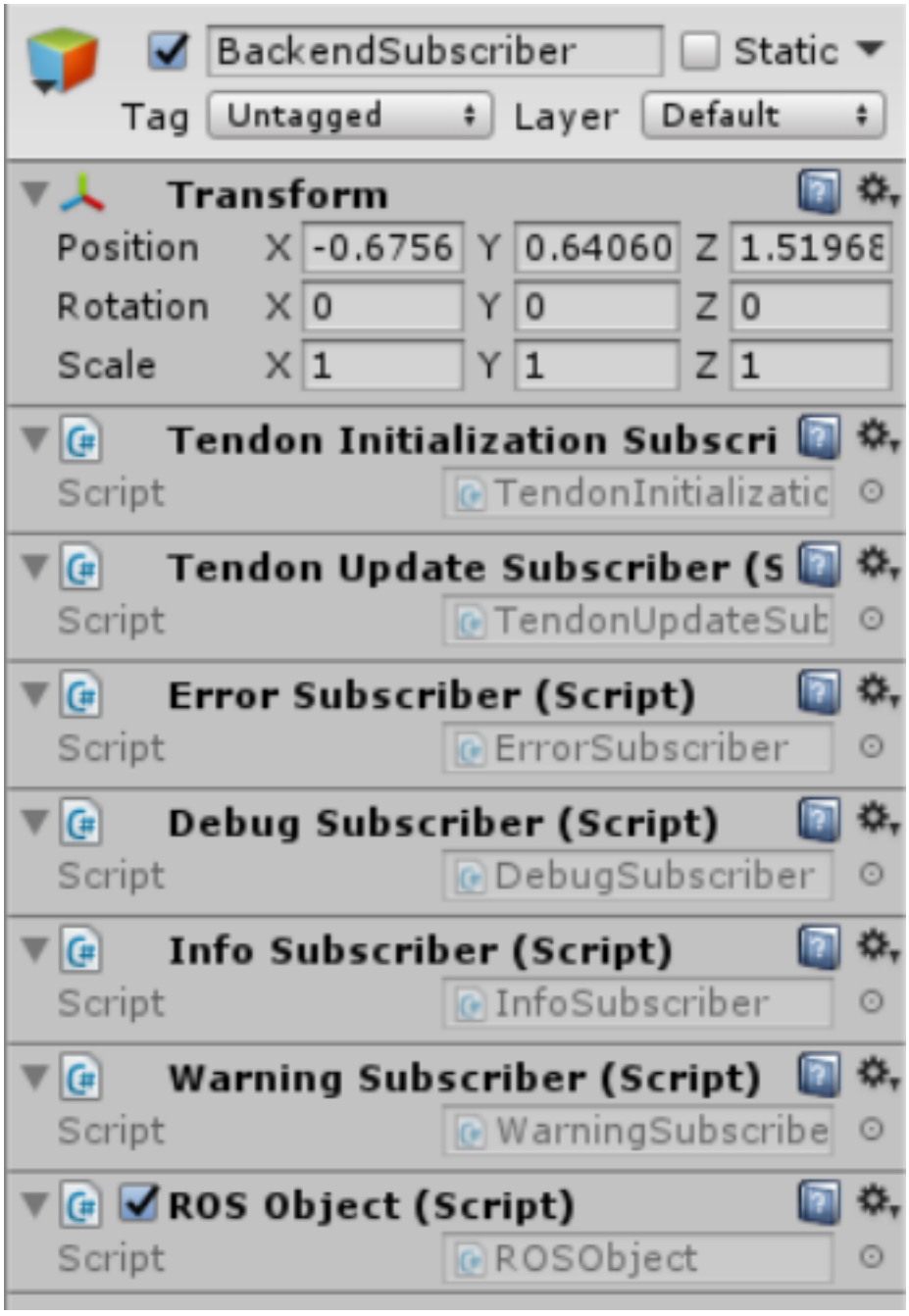


Fig. 4.33: Game Object containing all subscribers and a ROS Object component

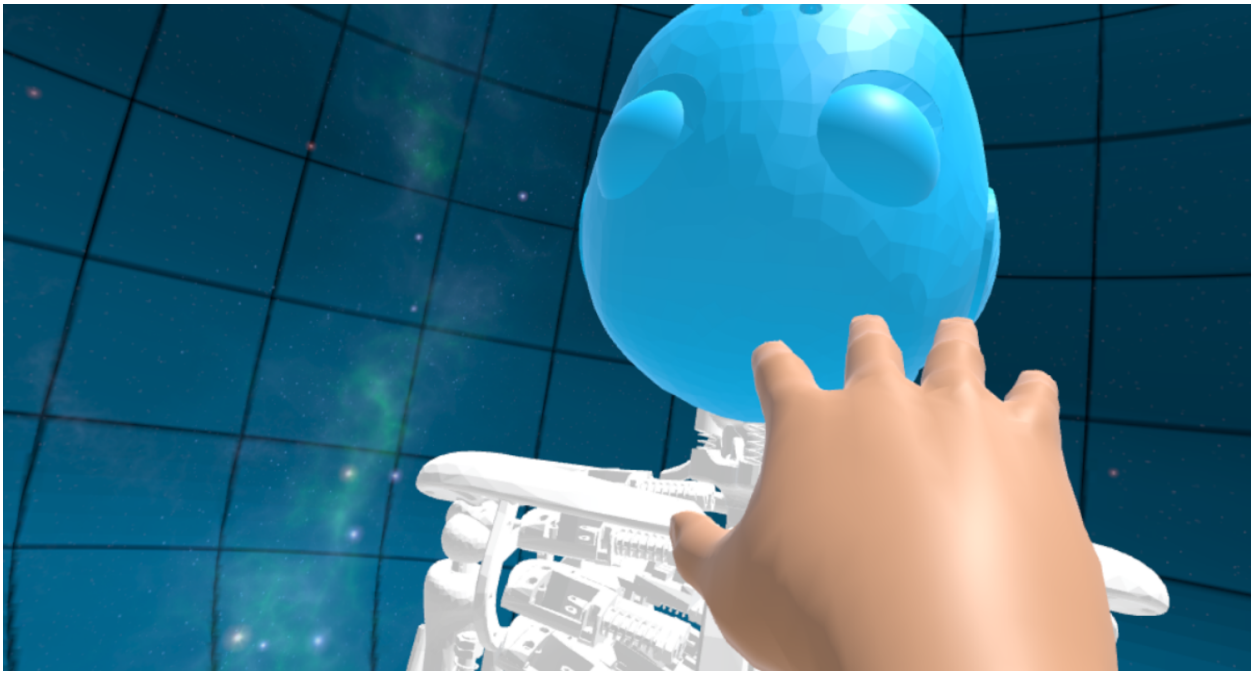


Fig. 4.34: Moving Roboy in VR using the Hand Tool

4.17.2 How does it work?

As soon as a user grabs a body part of Roboy, forces are calculated. The further the user pulls or pushes, the higher is the calculated force. This force is sent to the simulation running in Gazebo, which applies the forces and sends an updated position back to Unity, which adapts Roboy's pose.

4.18 User's Manual

4.18.1 Starting the Simulation

On the computer running the Gazebo simulation, set up all required exports and so forth as described in the *Getting started* section.

Run the launch file, which starts a ROS server and the simulation

```
roslaunch roboy_simulation roboy_moveable.launch
```

Additionally, the applied forces can be visualised in RViz to help with debugging and to gain further insight. Run the following command

```
rviz
```

Make sure to follow the additional steps described in *Getting started* to be able to see Roboy and the applied force.

4.18.2 Starting Unity

1. Start the unity project and set up the ROS server connection as described in **Getting Started**.

2. Start the scene.
3. After the controller assignment, switch to the HandTool
4. Point at Roboy and grab him using the trigger
5. Move Roboy around

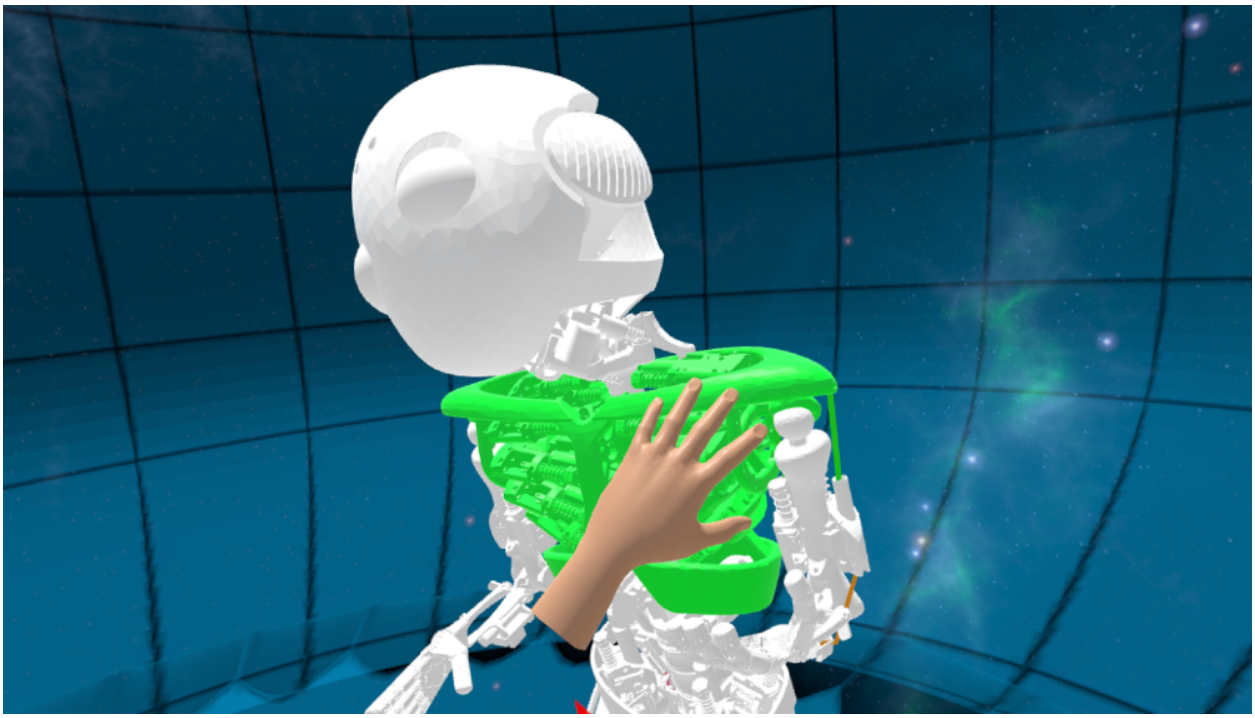


Fig. 4.35: Roboy being shoved around

4.19 Developer's Manual

4.19.1 Gazebo Simulation

For the gazebo part a launch/ world file automatically loads the world (with all the surrounding objects) that has been specified and the version of Roboy which was chosen. A plugin is in charge of applying the forces and sending the forces to be visualized by RViz.

The simulation VRRoboy2 also applies the external forces and simulates Roboy, but due to a Gazebo bug, model plugins are not loaded. Using the .launch file is recommended.

4.19.2 Model Configuration

A new model roboy_simplified_moveable was derived from an existing model and additional values / changes were applied:

- Roboy's feet are static, therefore glued to the ground
- Roboy is not affected by gravity

- The joints have a high friction: This, together with the strong forces applied, makes the inertia of Roboy's body neglectable / less prominent and he moves immediately

4.19.3 Unity Scene

The forces which are to be applied are calculated in the HandTool and published over ROS. A position subscriber receives the updated position and applies it to the model. **Issue:** Unity works with around 80 FPS, while the simulation publishes around 300 messages per second. Even though the pose subscriber drops superfluous messges, a back-log of messages cannot be prevented on Unity-side as of now and the visualised model starts to lag behind.

Solution for now: The simulation restricts the number of sent messages

4.19.4 Coordinate Systems

Coordinate systems presented a major issue in this project. Unity and gazebo have different coordinate systems, additionally, local and world coordinate systems add to the complexity. Please see the **Implementation/Coordinate Systems/** section for more information.

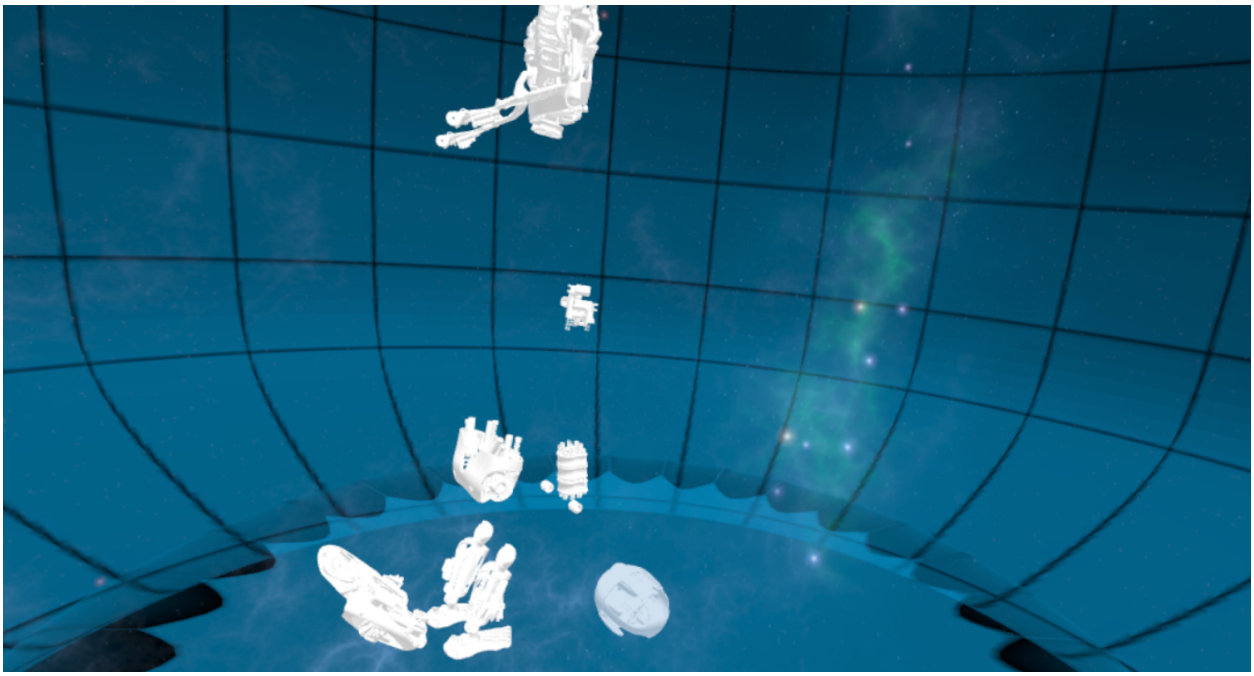


Fig. 4.36: The result of local and world coordinate system discrepancies

4.20 Introduction

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

NOTICE: NOT FULLY FUNCTIONING: DOES NOT FIND MODEL FOLDERS IN REPOSITORY

4.20.1 What is it?

The Automatic Update Pipeline is a convinient Unity Editor extension, which consists of two core features. The Model Updater is a tool to download new models or update existing ones which are used in the RoboyVR project. The World Updater downloads and updates worlds and environments created with gazebo.

4.20.2 How does it work?

Model Updater

After a short setup, where the user has to select the blender.exe and set the models github_repo path, the user can scan the Github Repository for models. Now a list of found models appears and the user can select, which specific models to download. Pressing "Download" loads the models and additionally converts them with blender to .fbx, so they can be used in Unity. Additionally the model.sdf file is downloaded. Because the models are downloaded into the Assets folder of the Unity project, they will automatically be imported into unity. Afterwards you just have to press "Create Prefab" and the model will be saved as a prefab, the pose and scale of given in the model.sdf will be applied here. Finally the user can easily just drag and drop the model into the VR scene.

World Updater

Just like with the Model Updater first the blender.exe has to be selected and the world github_repo path hast to be set. Then the user can scan the Github Repository for worlds. The list of found worlds appears and the user should select, which worlds to download. After the user presses "Download", the .world file is downloaded and read. The .world file gives information, which models have to be downloaded for the environment. Because the models are downloaded into the Assets folder of the Unity project, they will automatically be imported into unity. Finally pressing "Create World" will save every model in the world as one prefab, which the user can easily just drag and drop into the VR scene.

4.21 User's Manual

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

NOTICE: NOT FULLY FUNCTIONING: DOES NOT FIND MODEL FOLDERS IN REPOSITORY

Important

python needs to be installed in your path such that the command "python" is recognised and executed. This is the case for both the mesh and the world updater script. For further information, see this link .

4.21.1 MeshUpdater

Getting started

Open the MainScene in the Unity project RoboyVR, and select the Updater object in the hierarchy tab. The Updater has a script called Mesh Updater. This slightly differs from the video shown (as it is not in a separate scene), though the script functions exactly the same. The following instructions all are entered here.

Github Repository

Enter the link of the Github Repository where the models are located, which you want to download. Make sure the link ends with a slash! You chose the branch from which you want to download the models.

Default:

- Github_Repository = https://github.com/Roboy/roboy_models/

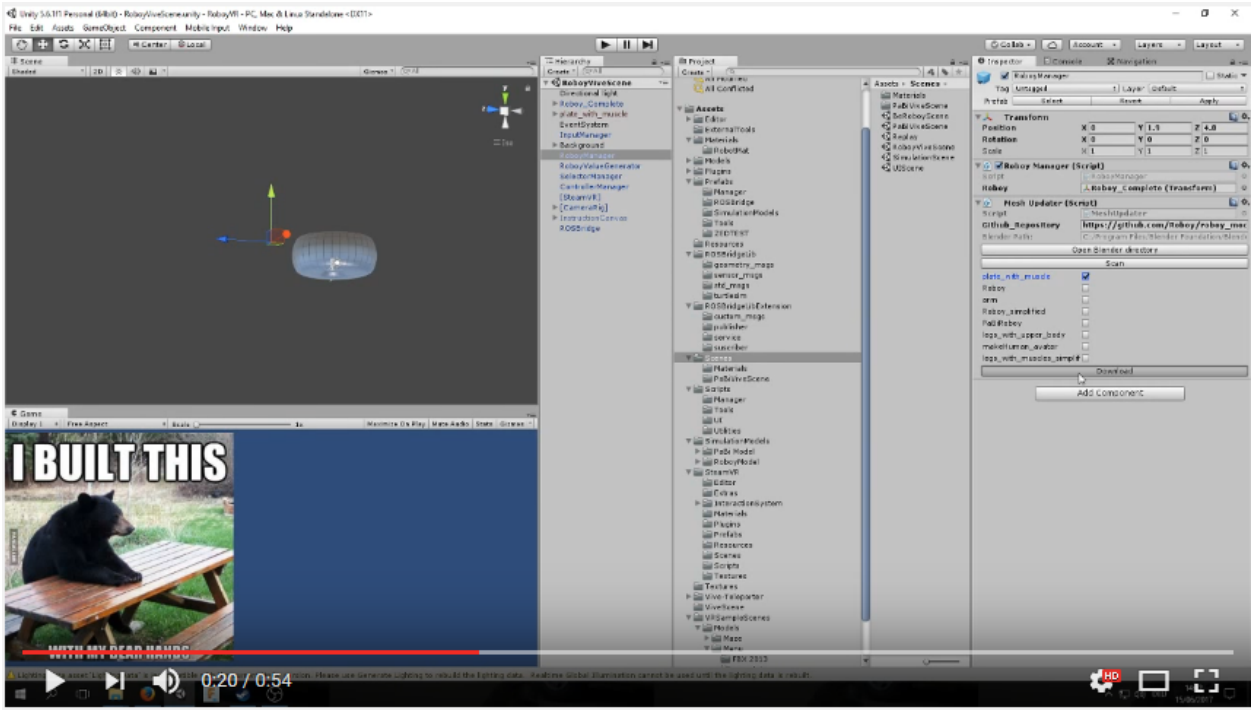


Fig. 4.37: Video showing the MeshUpdater in a custom Scene

- Branch = VRTeam

Set Blender.exe

Click on "Open Blender directory" and choose the blender.exe.

i.e.: C:\Program Files\Blender Foundation\Blender\blender.exe

Scanning

Click "Scan" and wait until UnityEditor shows you every model in the Github_Repository.

Downloading

Select the models you want save as prefab and press "Download". You can select more than one model. This may take a while, since the downloaded models will also automatically be imported into Unity.

Create the Prefab

After importing the files, press "Create Prefab". You can now find the created prefab in Assets/SimulationModels/...

4.21.2 WorldUpdater

Getting started

Open the MainScene in the Unity project RoboyVR, and select the Updater object in the hierarchy tab. The Updater has a script called World Updater. The following instructions all are entered here.

Github Repository

Enter the link of the Github Repository where the worlds and related models are located. Make sure the link ends with a slash. Also set here, which branch is used to download the models from.

Default:

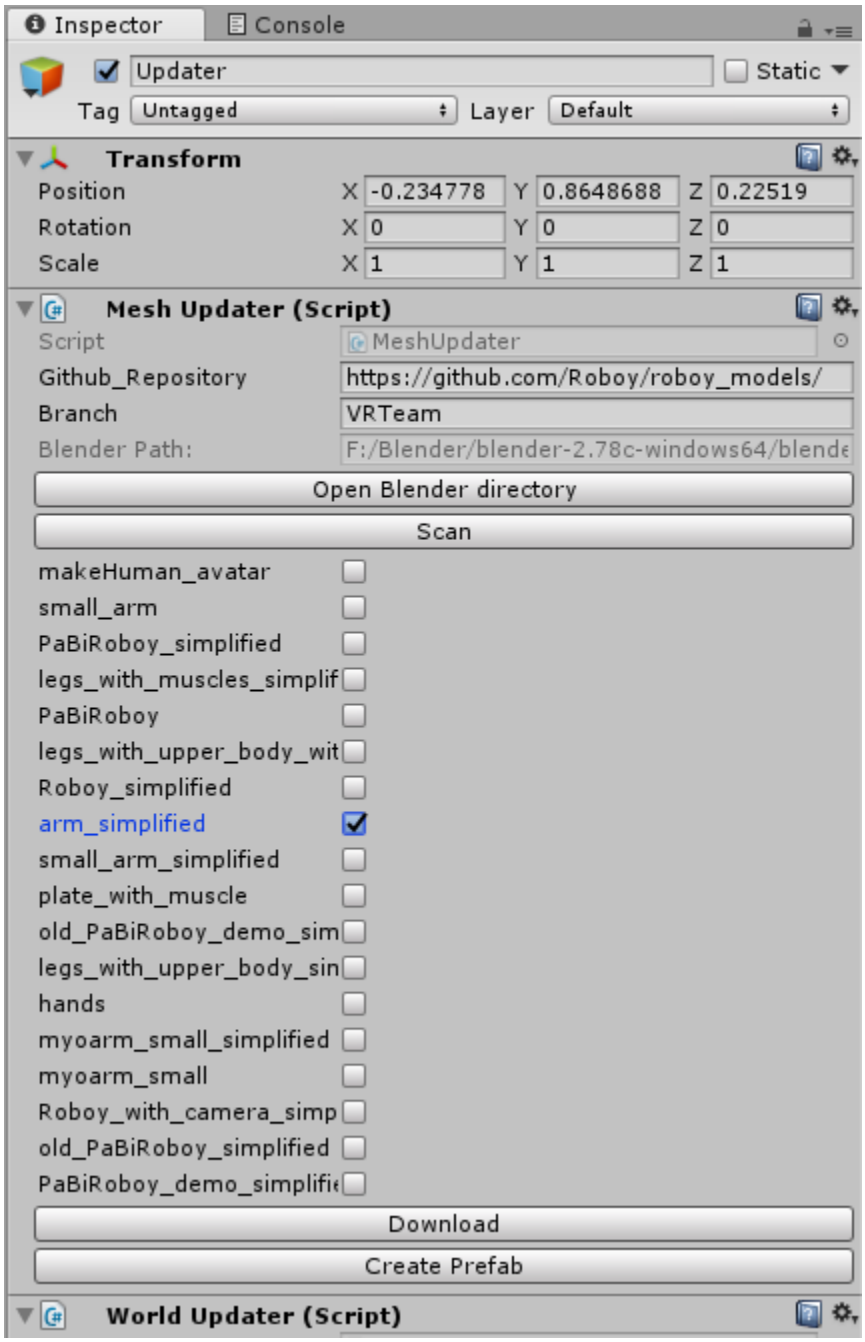


Fig. 4.38: Mesh Updater GUI

- Github_Repository = https://github.com/Roboy/roboy_worlds/
- Branch = master

Set Blender.exe

Click on "Open Blender directory" and choose the blender.exe.

i.e.: C:\Program Files\Blender Foundation\Blender\blender.exe

Scanning

Click "Scan" and wait until UnityEditor shows every world located in the Github_Repository.

Downloading

Select the worlds you want save as prefab and press "Download". You can select more than one world. This may take a while, since the downloaded worlds and related models will also automatically be imported into Unity.

Create the World

After importing the files, press "Create World". You can now find the created world in Assets/SimulationWorlds/...

4.21.3 Troubleshooting

As mentioned beforehand, make sure the "python" command is executable, adapt the system path variables if necessary.

Furthermore, please take a look at the current state chapter.

State not set correctly

If the Editor looks like this, without a blender path set:

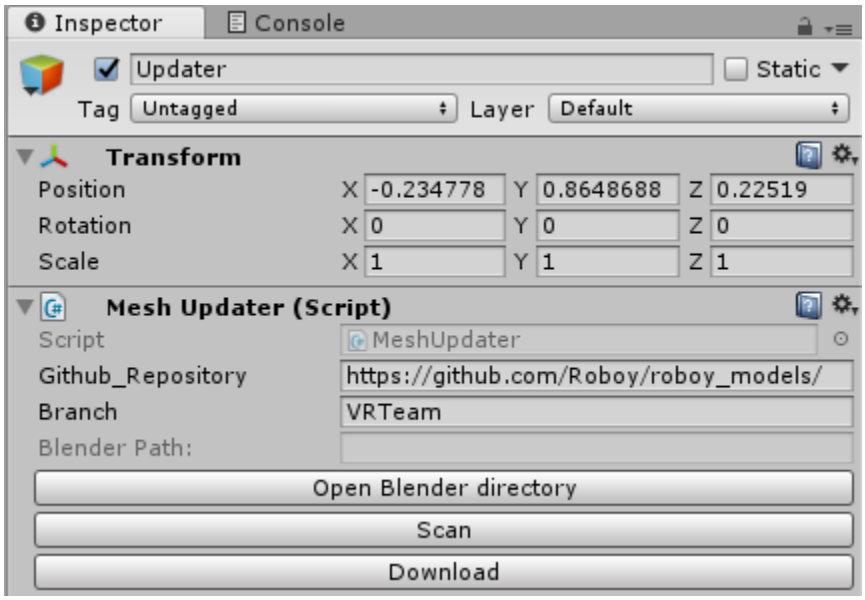


Fig. 4.39: state not correct

Try removing the script and adding it again.

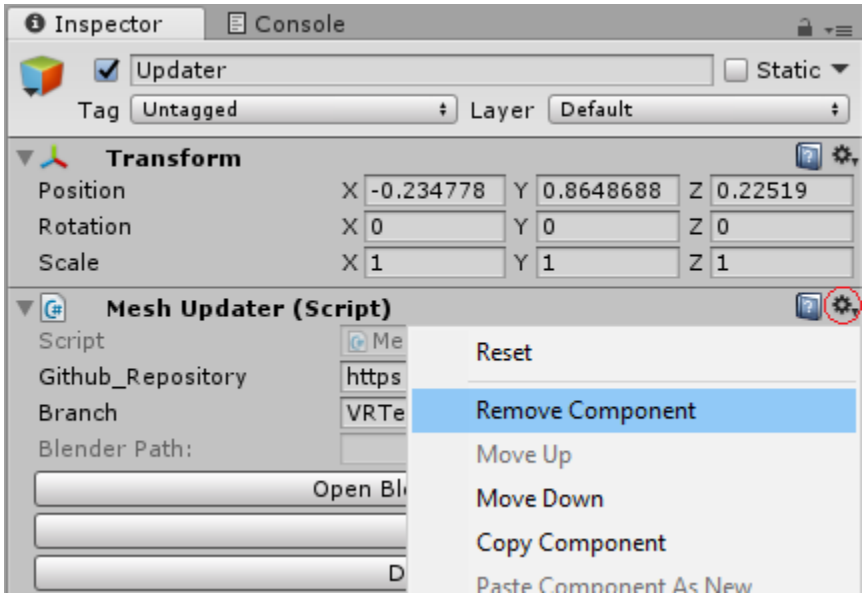


Fig. 4.40: Removing the Updater script

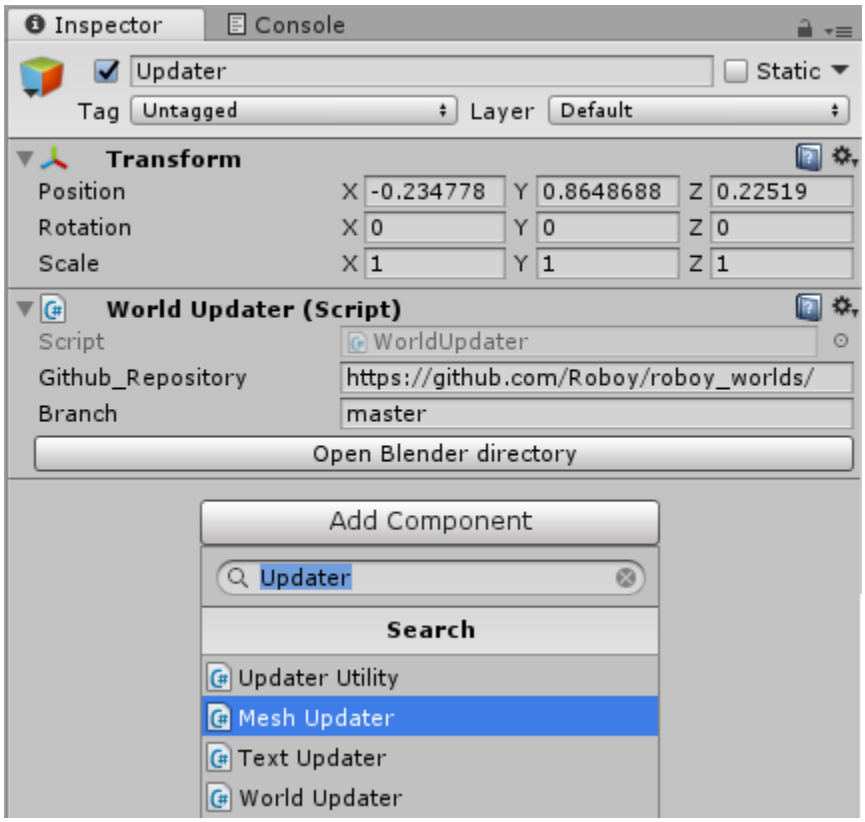


Fig. 4.41: Reloading the Updater script

4.22 Developer's Manual

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

4.22.1 MeshUpdater

Summary

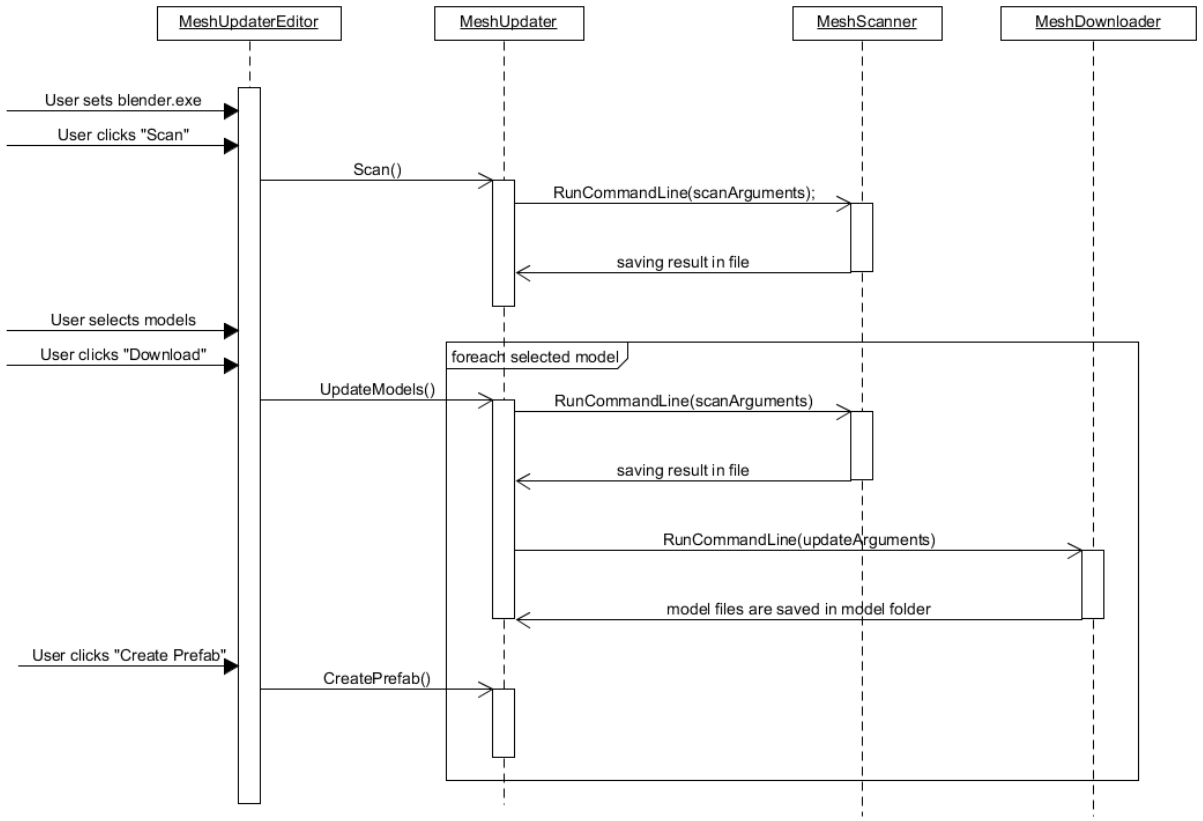


Fig. 4.42: Sequence Diagram for MeshUpdater

Prototype: a fully automated model loading script

1. Model loading is controlled by simple GUI elements
2. Models are listed from the roboy_models repo for user selection
3. Selected models are downloaded and converted to use them in Unity
4. Implement the sdf_dom.py script for the MeshUpdater
5. The selected model and world (.dae or .stl meshes) are automatically saved in a prefab, which can easily be loaded into the scene and here enable the known interaction: selection of model parts and motor state visualization

GUI elements

MeshUpdaterEditor.cs: Custom editor script to be able to call functions from meshUpdater at edit time through buttons. This is the GUI you use when updating a model. The GUI has different states, so the user can't skip necessary steps.

The first state is called "Initialized". In this state you can see the Github_Repository as a public string, used to find the models to download. You can put in any link, as long as the models are in the same folder hierarchy as in roboy_models. Make sure the link ends with a slash. Default string is "https://github.com/Roboy/roboy_models/"

If the blender directory isn't set, you can set it by clicking the button "Open Blender directory". This will open the Windows Explorer and you have to select the "blender.exe". After setting the blender directory, the state is changed to BlenderPathSet. This state shows the blender path as a string. Here the is GUI disabled so it can't be edited in UnityEditor.

Also the state now shows a button called "Scan". This button calls meshUpdater.Scan(). When meshUpdater.Scan() finishes, the state will be changed to "Scanned".

In the new state, you can see a list with the models that were found by the scanning script and can select, which of these you want to download by checking off the corresponding boxes. The "Download" button then calls meshUpdater.UpdateModels(), in which the state is set to "Downloaded".

After every downloaded model is imported in Unity, and state is set to "Downloaded", you can click the button "Create Prefab", this will call meshUpdater.CreatePrefab().

Scanning

meshUpdater.Scan(): First of all the scan function creates a local array scanArguments, filling it with {"python", m_PathToScanScript, Github_Repository} This is used to RunCommandLine(scanArguments), which starts ModelScanner.py.

ModelScanner.py scans the source code of the Github_Repository for links to subfolders by using regular expressions. The names and links of the subfolders (models) will be saved in a temporary file that we can read in later on.

Now the Scan() function creates a <string, string> dictionary. This is filled with model names and their links, which were saved in the temporary file. Then names are also written in a <string, bool> ModelChoiceDictionary, which is used for the selection in the UnityEditor. Lastly the current state is set to "Scanned".

Downloading

MeshUpdater.UpdateModels(): For every entry in ModelChoiceDictionary that is true, the ModelScanner.py is used to get each subfolder. The model.sdf is also being downloaded with the ModelDownloader.py. This is to find the mesh folder because of the way the hierarchy is currently set up in the roboy_models github repository. Now every link to the visual and collision folders inside the subfolder is given to the ModelDownloader.py, together with the m_PathToBlender and the path to where to store the downloaded models.

ModelDownloader.py is again scanning the source code, but this time not for folders, but for files with the .dae or .stf extension. Then it downloads every model to the given path, by creating new files and copying the raw content of the files stored in github. Finally all downloaded models are imported into blender, converted to a .fbx file and exported. The original files are overridden. The conversion is necessary so we can use the models in Unity.

Create Prefab

MeshUpdater.CreatePrefab(): Creates a GameObject called modelParent. With importModelCoroutine(string path, System.Action<GameObject> callback) a converted .fbx model in the model folder is loaded in as a temporary GameObject meshCopy. The sdf_dom.py is then used to get pose and scale of each mesh out of the model.sdf. Now a collider, the RoboyPart script and the SelectableObject script are attached to the meshCopy. The collider attached is the mesh downloaded in collision folder with the same name as meshCopy. The GameObject meshCopy is then attached as a child to modelParent. This happens for every model in the model folder.

Afterwards an empty prefab is created with the name modelname.prefab. The prefab's content is then replaced by modelParent. At last modelParent is deleted since we don't need it anymore.

4.22.2 WorldUpdater

Summary

Prototype: A fully automated world loading script

1. World loading is controlled by simple GUI elements
2. Worlds are downloaded from the roboy_worlds repository, and listed for user selection
3. Selected worlds and their related models are downloaded and converted to use them in Unity
4. Implements the world_dom.py script for the WorldUpdater
5. The selected worlds (.world) are automatically saved in a prefab, which can easily be loaded into the scene

GUI elements

GUI elements are almost exactly implemented like in the *MeshUpdater* , only the default Github_Repository and Github Branch are changed accordingly: "https://github.com/Roboy/roboy_worlds/"

Scanning

WorldUpdater.Scan() is doing the same as the MeshUpdater.scan() function.

Downloading

WorldUpdater.LoadWorlds(): For every entry in ModelChoiceDictionary that is true, the ModelScanner.py is used to download the .world files of the world.

WorldUpdater.Magic(): Uses the world_dom.py to get models and their pose and scale out of the .world file. Saves every single model in a struct. Now downloads every model, given by the .world files, from the roboy_worlds/models github repository, using the ModelDownloader.py.

ModelDownloader.py is again scanning the source code, but this time not for folders, but for files with the .dae or .stf extension. Then it downloads every model to the given path, by creating new files and copying the raw content of the files stored in github. Finally all downloaded models are imported into blender, converted to a .fbx file and exported. The original files are overridden. The conversion is necessary so we can use the models in Unity.

Create World

WorldUpdater.CreateWorld():

First of all, every downloaded model is saved as a Prefab (similar to MeshUpdater.CreatePrefab()), because the prefabs are later used to create the world. Then a new empty GameObject is created, which is the parent object of the entire world. Here every model prefab, contained in the .world file, will be instantiated, with the pose and scale given in the .world file. Finally the entire GameObject is saved as a prefab, which is the world, generated by the .world file.

4.23 Current State

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

NOTICE: NOT FULLY FUNCTIONING: DOES NOT FIND MODEL FOLDERS IN REPOSITORY

Right now the Updater is converting .dae and .stl files into .fbx to use them in unity. To set each model and link at the correct position, .sdf and .world files are read out and their poses are applied to the models/worlds.

Unfortunately the world_dom.py and sdf_dom.py need the .world/.sdf files in a certain format, so that most of these files have to be edited and saved in a sperate branch of the github repository. For example the sdf_dom.py script can't handle "<pose frame=' '>" or plugins. As a result of this the MeshUpdater only works with arm_simplified in the VRTeam branch and the WorldUpdater only works with testworld2 in the roboy_worlds github repository. Also as of

right now, there is no texture applied to the downloaded models, so Unity just uses some standard ones, which might look bad.

In the future, we need to include textures in the Updater process. Also we should figure out if there is an easier way to use .sdf and .world files, without having to change the actual files.

4.24 Introduction

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

4.24.1 What is it?

PaBi-VR shows the Roboy PaBi legs in a virtual environment, where the model moves according to values given by a simulation or the real-life legs. This way, simulated dance moves or the real movement of the legs can be captured and investigated further. Additionally, new dance moves can be created by shooting at the model with the toy gun.

4.24.2 How does it work?

The PaBi legs are simulated in Gazebo using a launch-file and model plugins, which receive and process the given forces. A force is created, when a body part is hit by a foam bullet. The force position is set at the point of impact, and the information is then sent to the simulation. This, in turn, moves the legs according to the force values. Another option to animate the legs is to use a plugin, which sends continuous commands to the simulation to be applied to the model. The resulting pose is then communicated to Unity and the VR model, where the new position is updated accordingly.

4.25 User's Manual

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

4.25.1 Setup Ubuntu side

Assuming Gazebo is already installed as described in the *Installation* and *Getting Started* sections, the *.launch* file needs to be run.

1. Source/ export all paths and files as initially described in the *Getting Started* section.
2. Run the launch file which starts ROS, Gazebo with the PaBi legs and a PaBiDanceSimulator ROS node

**NOTICE: THIS COMMAND IS NO LONGER WORKING AS HTHE FILES WHERE RENAMED. CHOOSE THE APPROPRIATE .LAUNCH FILE **

```
roslaunch roboy_simulation pabi_world.launch
```

This should be the result:

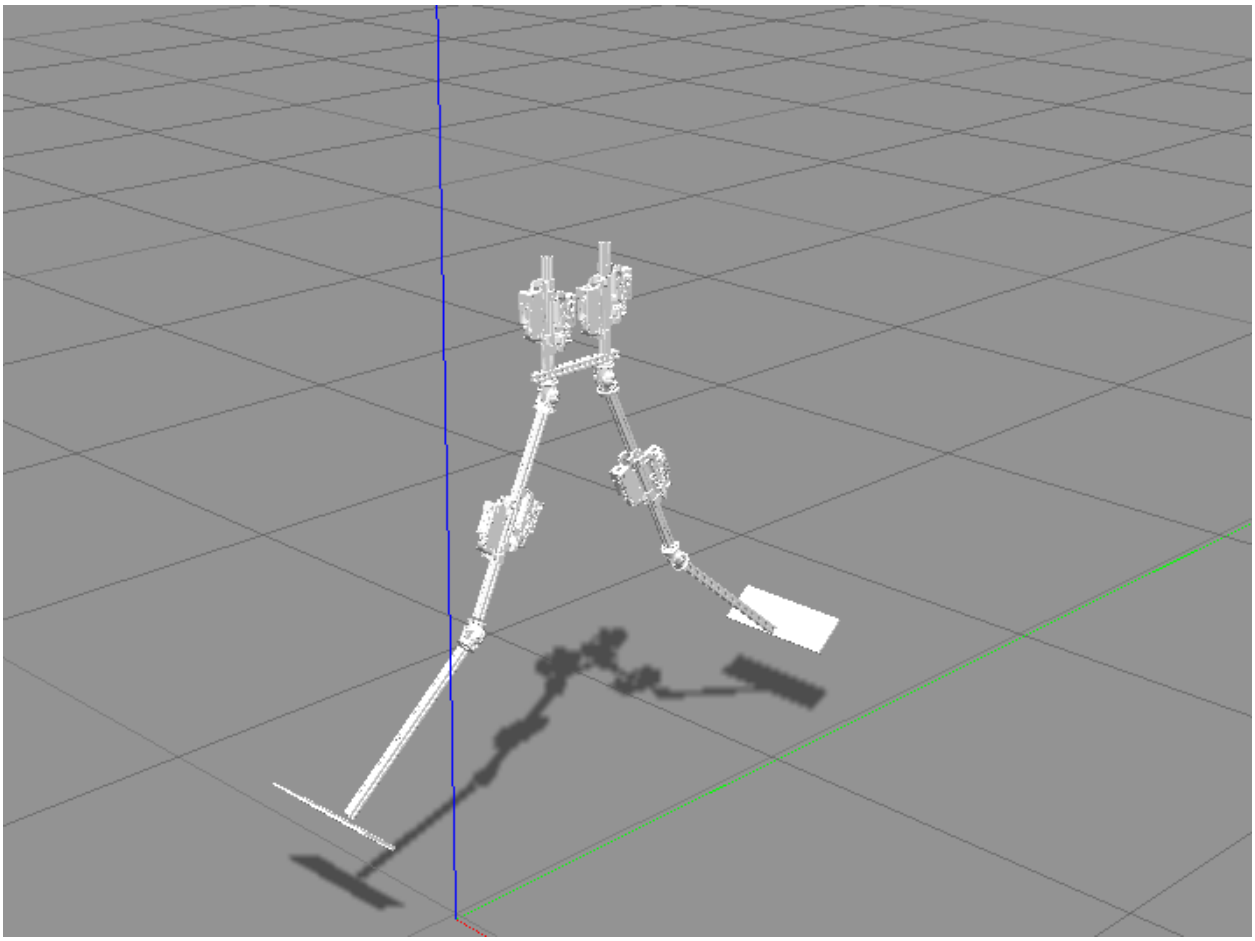
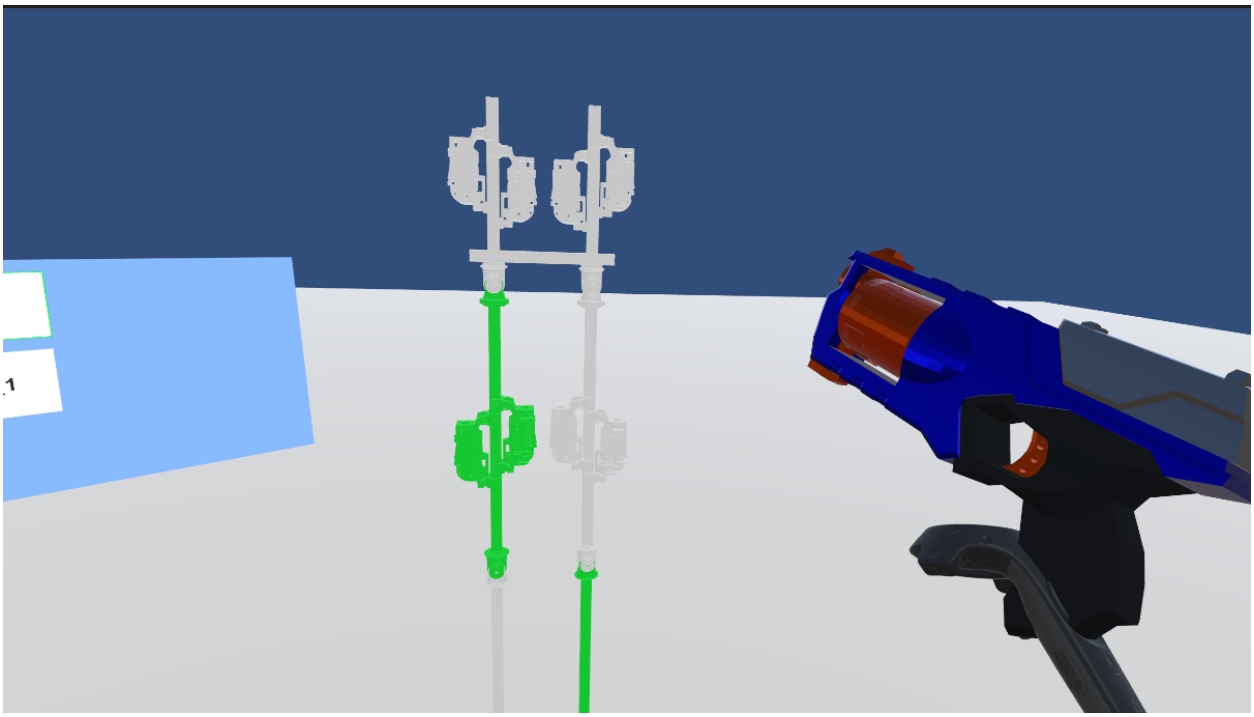


Fig. 4.43: PaBi model in Gazebo

4.25.2 Setup Unity side

Unity should be set up according to the *Installation* and *Getting Started* instructions. Therefore, only start Unity and open the PaBiViveScene. There should be a ROSBridge object in the hierarchy. Select this object and enter the IP Adress of the machine on which the simulation is running. PaBi now shows the dance moves, interaction with the leg model is possible via a GUI and different tools.



Note: Shooting PaBi with the nerf gun does not have any consequences

4.26 Developer's Manual

NOTICE: THIS CODE IS NOT BEING MAINTAINED AND MAY THEREFORE NOT WORK CORRECTLY ANYMORE OR THE INFORMATION MAY BE OUTDATED

4.26.1 Gazebo Plugin

The main functionalities are implemented in the plugin *ForceJointPlugin* (path: path-to-roboy-ros-control/src/roboy_simulation/src/ForceJointPlugin.cpp)

The plugin works as follows:

1. It loads the model into Gazebo.
2. It starts one topic for all revolute joints of the PaBi model. That means you have only one topic for all joints at once.
3. It subscribes to the created topic.
4. It creates a publisher which publishes the pose of PaBi so we can subscribe to the topic on the Unity side.
5. It makes PaBi stationary so he does not fall down when the legs are not touching the ground.

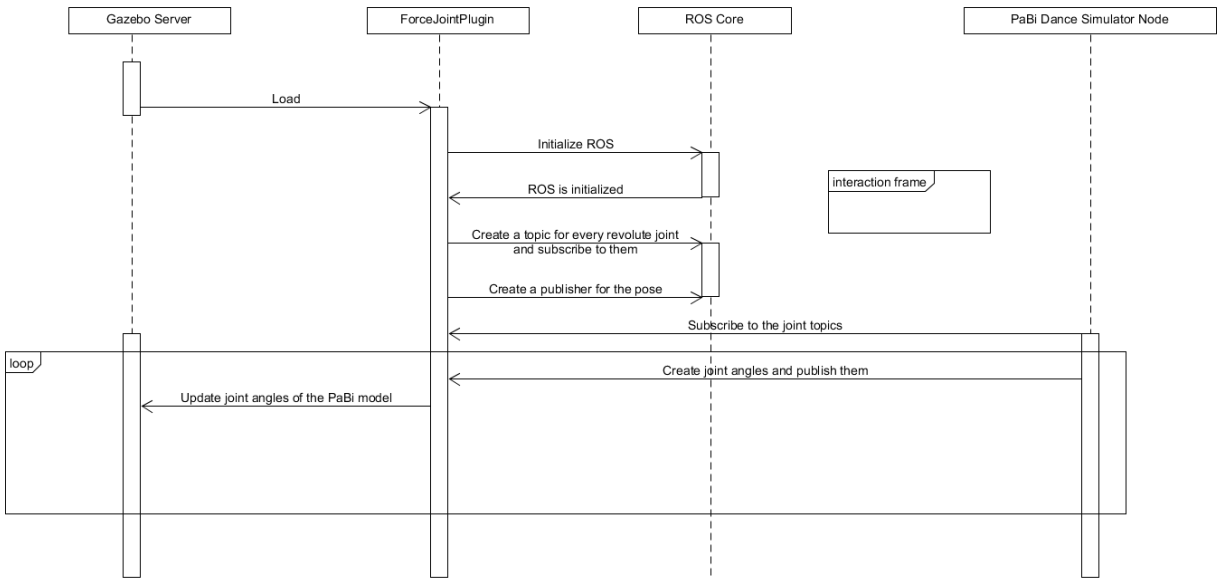


Fig. 4.44: Sequence diagramm of the ForceJointPlugin

The topic name for the joint commands with type **roboy_communication_middleware::JointCommand** is:

```
/roboy/middleware/JointCommand
```

The JointCommand expects an array of the link names and one value for each given link, meaning in the case of PaBi you need four values in both arrays.

The pose is published with message type **roboy_communication_simulation::Pose** on the topic:

```
/roboy/pabi_pose
```

The main functions of the plugin are:

1. Load: It loads the model into gazebo and creates the joint subscribers and the pose publisher.
2. JointCommand: Is called every time the plugin receives a joint command. It updates the joint angles value list and publishes the new state.
3. publishPose: Publishes the pose of PaBi.

4) OnUpdate: Is called every gazebo update frame. Therefore we have to zero out the forces of PaBi and update the joint angles of the actual model.

Change the following line in OnUpdate if you want PaBi to be able to fall down:

```
model->SetWorldPose(initPose);
```

4.26.2 PaBiDanceSimulatorNode

This ROS node creates four publishers for the joints of PaBi. In the Main loop it publishes new joint angles. To make the movement smooth the published joint angles are changed gradually in small steps from -90° to 0° and back. Therefore we have two functions. One to start the animation:

```
void PaBiDanceSimulator::startDanceAnimation()
{
    while(ros::ok())
    {
        if(adjustPoseGradually(true))
            adjustPoseGradually(false);
    }
}
```

And another to adjust the pose:

```
bool PaBiDanceSimulator::adjustPoseGradually(bool goUp)
{
    float stepSize = 1;
    int sleeptime = 10000;
    // adjusts the joint angles to -90° in 90 * stepSize * 0.01 seconds
    if(goUp)
    {
        float currentAngle = 0;
        while(currentAngle > -90)
        {
            publishAngles(currentAngle);
            usleep(sleeptime);
            currentAngle -= stepSize;
        }
    }
    else
    {
        float currentAngle = -90;
        while(currentAngle < 0)
        {
            publishAngles(currentAngle);
            usleep(sleeptime);
            currentAngle += stepSize;
        }
    }
    return true;
}
```

4.26.3 Unity Scene

In Unity we have the ROSBridge which connects to the ROSBridge on the simulation side. On the PaBi legs we have a **ROSObject** script attached to the legs.

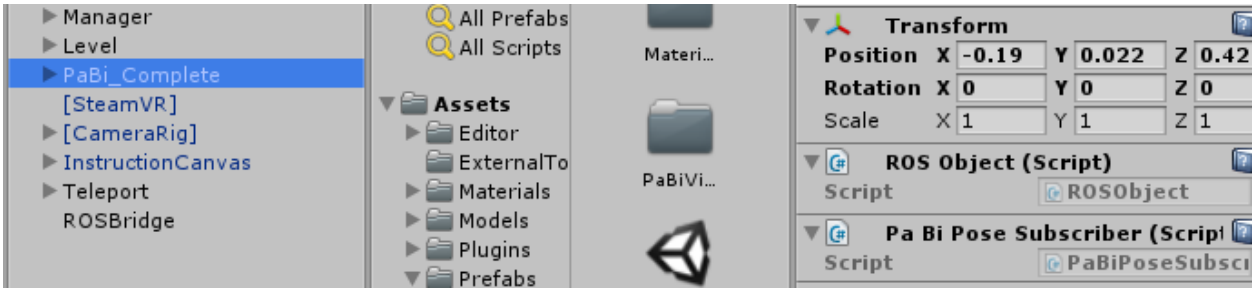


Fig. 4.45: ROSObject component

This script is needed because the **ROSBridge** searches for every **ROSObject** in the scene and adds every **ROS Actor** (Subscriber, Publisher, Service) on this object. So f.e. if you want to add your own subscriber you have to write the subscriber such that it derives from *ROSBridgeSubscriber* and define on which topic you subscribe, which message type the topic has and what happens at a callback meaning when you receive a message.

4.27 General Project Architecture

4.27.1 Building Blocks View

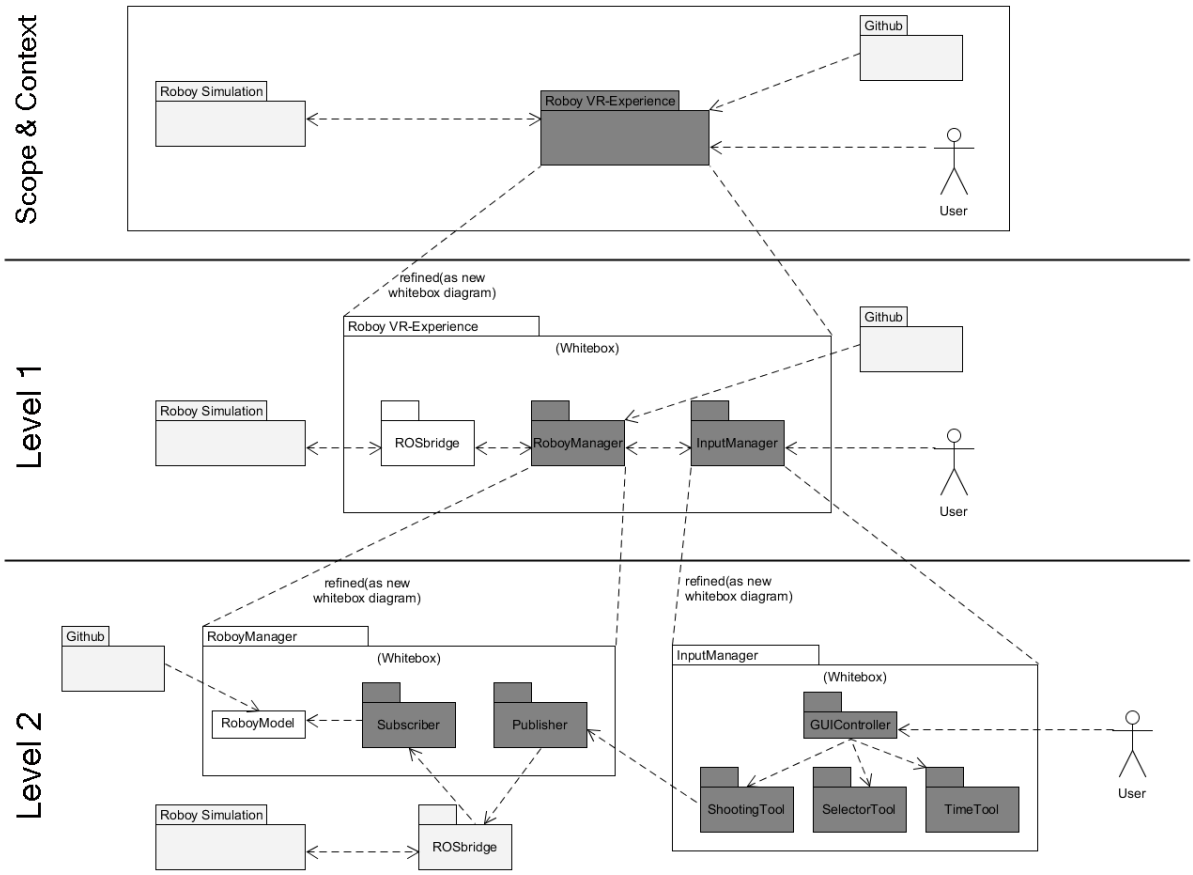


Fig. 4.46: RoboyVR Experience has several neighbouring systems like the simulation and github, it consists of various components like RoboyManager/Inputmanager and can be manipulated by the user through the HMD system and the controllers.

4.27.2 Deployment View

Note: Further information about certain implementation strategies can also be found in the *Developer's* Manual subsections of the individual projects *BeRoboy*, *State Visualisation*, *Model World Loader* and *PaBi*.

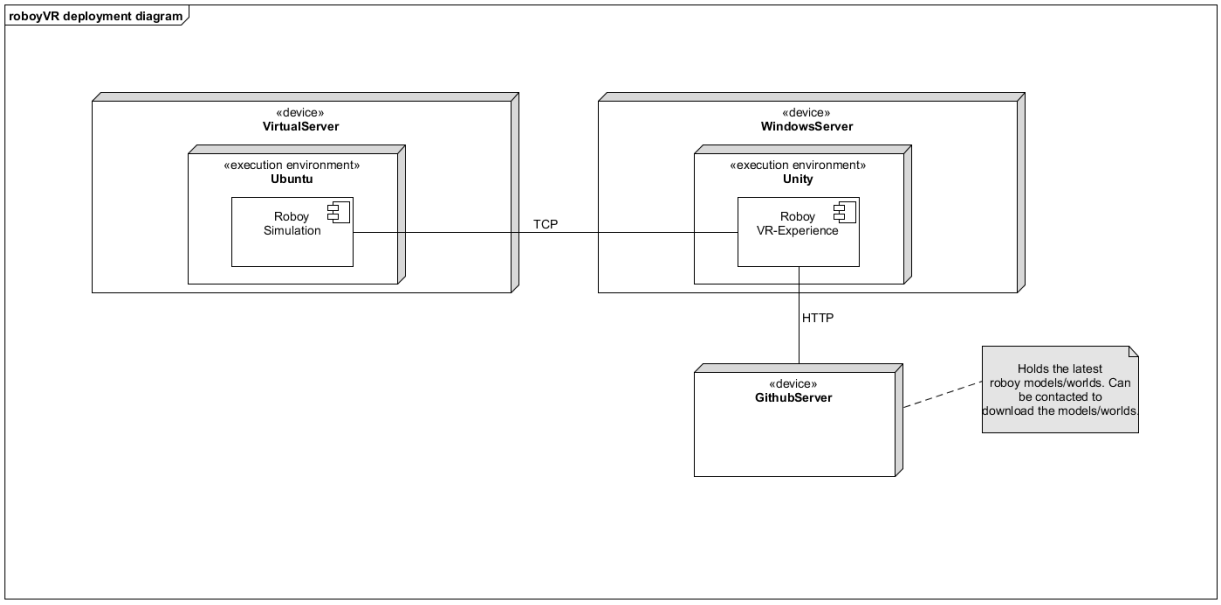


Fig. 4.47: Roboy simulation runs on Ubuntu on a seperate(virtual) machine, RoboyVR Experience runs on Unity.

4.28 Use Cases

4.28.1 Runtime Display Information regarding Roboyparts

4.28.2 Runtime Physical impact on roboy (shooting)

...

4.29 Conventions

We follow the coding guidelines:

Table 4.1: Coding Guidelines

Language	Guideline	Tools
Python	https://www.python.org/dev/peps/pep-0008/	
C++	http://wiki.ros.org/CppStyleGuide	clang-format: https://github.com/davetcoleman/roscpp_code_format

The project follows custom guidelines.

4.29.1 Regions

In general all scripts are ordered in regions:

- PUBLIC_MEMBER_VARIABLES

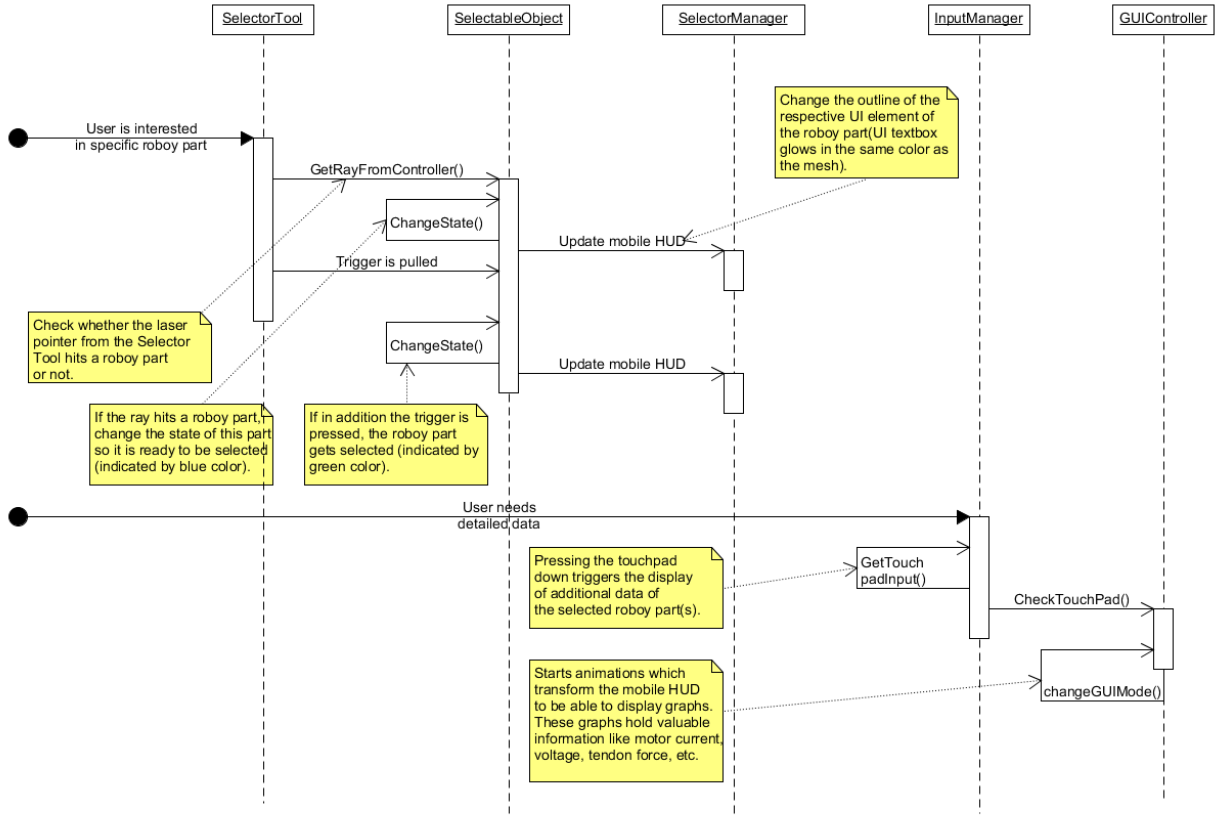


Fig. 4.48: User needs detailed information regarding specific roboy parts, e.g. power-consumption in motor24 upper_left_arm.

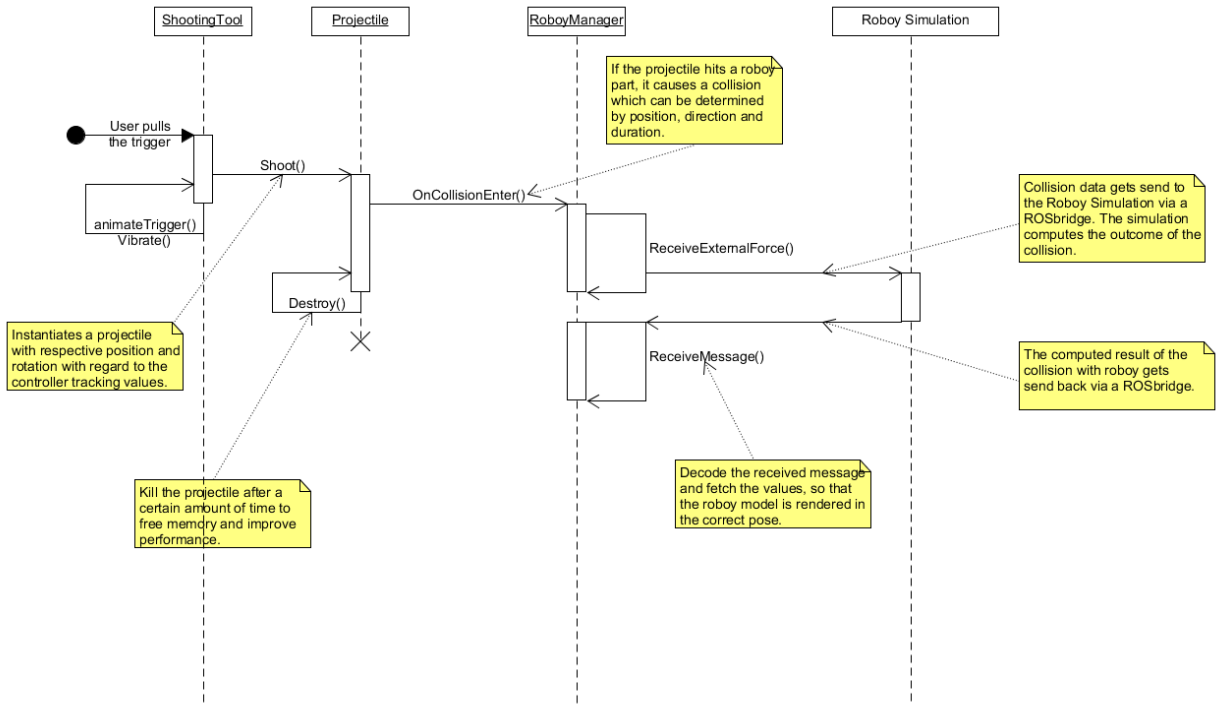


Fig. 4.49: User wants to physically harm the poor roboy and shoots a nerf dart towards him.

- PRIVATE_MEMBER_VARIABLES
- UNTIY_MONOBEHAVIOUR_METHODS
- PUBLIC_METHODS
- PRIVATE_METHODS

However, in some cases it does make sense do deviate from this structure.

1. Delete empty regions!

It is okay to have empty regions if you plan on filling this region in the near future (READ: in the next seven days). But to avoid whitespace code remove empty regions.

2. Add extra regions!

If you have a lot of variables of functions which are bundled together through some kind of logic then it may be beneficial for the readability to structure them in a seperate region, e.g. variables which are private but are marked with [SerializeField] so you can adjust them in the editor and are only used once then you can create a region **EDITOR_VARIABLES**.

3. Delete regions in small classes!

If you have a class with only two or three small functions/ variables then just write the code without the regions at all.

4. Use your brain!

As we are not the masterminds of the universe there are a ton of other cases where it does make more sense to use your **own** region logic. Feel free to extend/ adjust this list if you have a better approach.

Region Content

What goes where should be obvious. In the UNTIY_MONOBEHAVIOUR_METHODS region write your code in the same order as the unity functions are called, meaning:

```
void Awake(){}

void Start() {}

void OnEnable() {}

void OnDisable() {}

void FixedUpdate() {}

void Update() {}

void LateUpdate() {}
```

These are of course not all Unity functions but the most common ones. Note that **OnEnable()** is actually called before **Start()** but more often than not the logic in **OnEnable()** is coupled with **OnDisable()**. If this is not the case for you then change the order as you need it. For further reading of the execution order click here.

Also put **static** variables/ methods on top of the respective region.

```
#region PUBLIC_MEMBER_VARIABLES

public float static MagicNumber = 42f;

public float Speed;

#endregion // PUBLIC_MEMBER_VARIABLES

#region PRIVATE_MEMBER_VARIABLES

private float const m_PI = 3.14;

private int m_ElementCount;

#endregion // PRIVATE_MEMBER_VARIABLES
```

4.29.2 Naming

All code of this project which was implemented by ourselves follows also some specific patterns. In general, names should not contain underscores or numbers, exceptions see below. Addionally there is one important rule:

Write stuff out!

As we live in the glory days of code completion from the IDE you actually can give long names to your classes, variables etc. Therefore avoid using shortened names or abbrivations. (Exception: Widely-known ones like Xml, Html, Fbx etc.)

```
// Correct
public float EnemyHealth;
public float DamageOverTime;

// Avoid
public float elemCount;
public float dmgInAMinByTypeAInLvlKek
```

Classes

Classes should use PascalCasing.

```
// Correct
public class PlayerController
{
  //...
}
// Correct
private class PlayerHelper
{
  //...
}

// Avoid
public class Enemy_Controller
{
  //...
}
public class bullet01
{
  //...
}
```

On top of that scripts which implement the Singleton approach, the name should suffix "Manager". Note that we have already a singleton class. At the same time, classes which are **not** a singleton should avoid using "Manager" in the name. Most of the time what you want is then a **controller**.

```
// Correct
public class EnemyManager : Singleton<EnemyManager>
{
  //...
}
// Correct
public class UIController : Monobehaviour
{
  //...
}

// Avoid: a singleton without "Manager" suffix!
public class Robot : Singleton<Robot>
{
      //...
}
// Avoid: not a singleton!
public class LevelManager : Monobehaviour
{
      //...
}
```

Variables

Public variables

Public variables should also use **PascalCasing**.

```
// Correct
public float Width;
public float Height;

// Avoid
public float scale;
public float _size;
```

Private and protected variables

Private and protected variables should also use **PascalCasing** with a prefix "m_" for *member variables*.

```
// Correct
protected string m_Name = "Simon";
private string m_NickName = "The coding god";

// Avoid
private string heh = "heh?";
```

The reasoning behind this is so that you can differentiate between local and private variables at first glance.

Local variables

Local variables inside a method should use camelCasing.

```
// Correct
public void Heh()
{
  string heh = "heh??????";
      //...
}

// Avoid
public void BadHeh()
{
  string m_Wut = "wut?";
  //...
}
```

Methods

Public methods

Public methods should use PascalCasing.

```
// Correct
public void DoStuff()
{
  //...
}

// Avoid
public void doBadStuff()
{
  //...
}
```

Private and protected methods

Private and protected methods should use camelCasing.

```
// Correct
private void doPrivateStuff()
{
  //...
}

// Avoid
private void DoPrivateBadStuff()
{
  //...
}
```

Coroutines

Coroutines are special methods in Unity. They cannot be started by a simple method call. You must start them via **StartCoroutine()**. Therefore to make sure that future developers see at first glance whether your method is a coroutine or not, name them with a "coroutine" suffix.

```
// Correct
private IEnumerator someCoroutine()
{
  //...
}

// Avoid
private IEnumerator badCode()
{
  //...
}
```

4.29.3 General rules

If you follow the rules above you should be now a naming god. But as syntax is only one half of the equation here comes the second one: **semantics.**

1. Public variables/ methods should only be public if other scripts **really** need or may need access to this functionality.

It makes it easier to understand what your class actually does from the standpoint of other classes.

Variables which you need to be editable in the editor but are not actually accessed by other classes should be non public and marked with a [SerializeField] attribute. Note that not all types can be marked as serializable, e.g. properties of any kind are not. However, we have just the right custom attribute for this case: [ExposeProperty]. For this to work mark the variable which is wrapped in the property as serializable but also hidden from the inspector by [HideInInspector] so you do not see both, the property and the variable in the editor.

```
// Read only property exposed in the editor so we always see the current status for␣
↪debug reasons
[ExposeProperty]
public int Status { get { return m_Status;} }

[HideInInspector]
private int m_Status = 0;
```

2. Hide public variables when you do not want to change them in the editor.

This removes clutter from the inspector and makes it somewhat "safer". You can achieve this behaviour with the attribute [HideInInspector]. But be aware that if you first change the value of a variable in the inspector to something else than the default value given by code and **then** add the attribute it **still** holds the new value although it is not visible in the inspector.

```
// This will leave the public field serializable so it saves the state from the␣
↪editor to playing but hide it in the inspector.
[HideInInspector]
public int RandomNumber = 42;
```

3. Use properties if changes in your variables trigger other logical changes.

Properties are a nice way to wrap a variable and add more functionality to this variable. For example if you have a player who has different states depending on his current health, then you can write the logic or trigger the logic inside the property.

```
public int Health
{
  get
      {
        return m_Health;
      }
      set
      {
        m_Health = Mathf.Max(0f, value);
        // different callbacks depending on the health
        if(m_Health < 80)
          decreaseSpeed();
        if(m_Health < 40)
          setCanJump(false);
        if(m_Health < 0)
          dieMiserably();
      }
}

private int m_Health = 100;
```

4. Use the [RequireComponent(typeof(ComponentType))] to signal a strict dependency on another component on the same gameObject.

If your script depends on another script to work properly on the same gameObject add this attribute on top of the class declaration. This way every time your script is attached to another object the dependency is added automatically. It is useful e.g. if you have a script which adds a listener to a button in **Start()**.

```
[RequireComponent(typeof(Button))]
public class CustomButton
{
  void Start()
      {
        GetComponent<Button>().onClick.AddListener(pickleRick);
      }

      void pickleRick()
      {
        Debug.Log("I am a pickle Morty!");
      }
}
```

5. Coroutines should have an intermediate function if accessed from other classes.

To avoid the situation that another developer tries to call your awesome Coroutine method without a **StartCoroutine** don't expose coroutines but rather write an intermediate method which calls the desired coroutine.

```
// Class MrMeeseeks
public void AwesomeStuff()
{
      StartCoroutine(awesomeStuffCoroutine());
}

private IEnumerator awesomeStuffCoroutine()
{
  // magic
}

// Class Jesus
void Start()
{
  GameObject maria = new GameObject("Maria");
      MrMeeseeks meeseeks = maria.AddComponent<MrMeeseeks>();
      meeseeks.AwesomeStuff();
}
```

6. With this knowledge take over the world.

Basic stuff actually. Ask simon how to do it.

4.30 Architecture Constraints

Table 4.2: Hardware Constraints

Constraint Name	Description
HTC Vive	We need user position tracking and movement tracking.
ZED	For spatial mapping and a live stream of the environment.

Table 4.3: Software Constraints

Constraint Name	Description
Unity3D	Unity provides an interface for the HTC Vive with the steamVR plugin. On top of that it renders the simulation.
Gazebo&ROS	The simulation uses both systems.
OracleVM	We use the VM for running Ubuntu on the same machine. You can also just use Ubuntu on a separate machine.
Blender	We used blender to convert the roboy models so that Unity can import them.

Table 4.4: Additional Plugins

Constraint Name	Description
ROSBridge	It connects the simulation on Ubuntu with Unity on Windows.
steamVR	We use this interface to use the API of the HTC Vive.
ZED	This interface connects the ZED (Roboy's Eyes) with Unity.
PyXB	This is used for reading XML files in the Model/World Updater.

Table 4.5: Operating System Constraints

Constraint Name	Description
Windows 10	We did not test it yet on other Windows versions. It may also work on older machines.
Ubuntu 16.04	The simulation runs on Ubuntu.

Table 4.6: Programming Constraints

Constraint Name	Description
C++	The simulation is written in C++.
C#	Unity uses C# as the standard programming language.
Python	We use Python with the Blender API to automate the process of converting the roboy models.

4.31 Libraries and external Software

Contains a list of the libraries and external software used by this system.

Table 4.7: Libraries and external Software

Name	URL/Author	License	Description
Unity	https://unity3d.com/	Creative Commens Attribution license.	Game engine for developing interactive software.
SteamVR Plugin for Unity	https://www.assetstore.unity3d.com/en/#!/content/32647	Creative Commens Attribution license.	Unity-Plugin for HTC Vive Headset support.
ZED Plugin for Unity	https://github.com/stereolabs/zed-unity	Creative Commens Attribution license.	Unity-Plugin for the ZED camera.
Blender	https://www.blender.org/	Creative Commens Attribution license.	Tool for modeling and animating.
Oracle Virtual Machine	https://www.oracle.com	Creative Commens Attribution license.	Tool to run a virtual machine.
arc42	http://www.arc42.de/template/	Creative Commens Attribution license.	Template for documenting and developing software
PyXB	https://sourceforge.net/projects/pyxb/	Apache License V2.0	Python package that generates Python source code for classes that correspond to data structures defined by XMLSchema.

4.32 ROSBridge

4.32.1 ROS overview

The ROSBridge is a main part of this project as it connects the Unity project on Windows and the simulation on Ubuntu. We recommend to have a look at the tutorials here to get a basic understanding of ROS. The short version follows below.

The ROS core establishes connections between clients and creates topics. Topics are similar to a public blackboard, where entities can decide to publish - write on the blackboard, or subscribe - read from it.

Publisher

The publisher **sends** message over **one** topic. When you create the publisher you have to announce to ROS which type of message is to be sent as well as the name of the topic over which it is sent. Therefore at first you have to advertise the new publisher with providing this information. Afterwards you can publish data with the defined message type.

Subscriber

A subscriber *receives* data from one topic. The topic name as well as the message type need to be defined and announced to the ROS server as was previously the case for teh publisher.

Every time a message is published, the subscribing entities receive these. A CallBack function is triggered which handles the received data **Note: this is handled using threads**. Here, it can parse the message and process it.

Service

A service works differently compared to the publisher and subscriber pattern. It serves like function call in which the the service is invoked via a topic. This means that you as the *service client* call the *service server* with the required data depending on the certain service via a *service request*. Then the server responds to you via a *service response*. This is useful when you need to do certain tasks only on rare occasions e.g. reset a world.

Messages

To communicate the necessary information, different standard or already defined messages can be chosen, addiitionally, new messages can be defined. As the communication is between a Ubuntu side and a Windows computer using a ROSBridge as a mediator, messages must to be defined on both sides and need to match exactly.

Messages for the simulation Examples can be found in the previously mentioned tutorials. The conversion from the message type to the YAML format, which is used by ROS, is automatically applied when implementing simulations and plugins using the designated libraries. The message needs to be defined in an extra file and its path must be sourced, so that ROS is able to find and use the message type.

Messages in Unity In Unity, messages need to be converted manually. Looking at all custom messages defined in Unity gives a good insight into the structure and construction of messages. A ToYAMLString() method for each message type is defined, to parse the given message into a format expected and accepted by ROS. On the other side, messages which arrive at the Unity side need to be parsed so that these can be further processed.

Important

Gazebo and Unity have different coordinate systems, therefore, to avoid confusion and missing or superfluous conversions, **outgoing coordinates and rotations must be translated into the Gazebo coordinate space**, while **incoming messages must be parsed to Unity coodrinate space**. It is sufficient to only implement constructors and ToYamlString() parser, in case messages of this type are only designed to be published.

```
public override string ToYAMLString()
{
        // format the message ...
    return string_in_yaml_format;
}
```

Respectively, messages which will only ever be received only need to define a function which parses the messages into the correct format, as can be seen below.

```
public CustomMessage(JSONNode msg)
{
    // parse message here
}
```

Wrap-Up

Messages To define a custom message:

1. Define and source the message on Ubuntu side
2. Create a class for the message on Windows side: Derive the message class from *ROSBridgeMsg* so the ROSBridge knows that this is a message. Implement the constructor and/or the ToYAMLString method depending on the purpose.

Subscribers/Publishers/Services With a newly created message, create a subscriber, publisher or or a service for this message type and a given topic name. To implement this on the Ubuntu side, just follow the tutorial.

In Unity, a new subscriber/ publisher / service needs to be defined. Each of this ros types have different requirements to be valid but each of these have to derive from the corresponding base class, e.g. CustomPublisher : ROSBridgePublisher

The publisher needs to have:

1. GetMessageType()
2. GetMessageTopic()

The subscriber needs to have:

1. CallBack()
2. GetMessageType()
3. GetMessageTopic()
4. ParseMessage()

The service needs to have:

1. ServiceCallBack

All methods must be public and static. If you forget to implement one of these the ROSBridge will throw an expcetion which tells you which method is missing.

4.32.2 Wrapper

Originally we got the template from this git project. To make it more usable and developer-friendly we wrote a wrapper. The Wrapper consists of two main classes.

ROSObject

A ROSObject retrieves all ros subscrivers, publishers and services which are components of the same gameObject. These are communicated to the *ROSBridge* to announce all entities on start-up and remove all entities when disabling this ROSObject.

ROSBridge

The ROSBridge is the main part of the wrapper. It forwards the needed calls to *ROSBridgeWebsocketConnection* which establishes a websocket and is the actual backend.

4.32.3 Scene Setup

When you created your custom publishers etc. and messages you need to edit your scene such as the ROSBridge is informed of these. First a ROSBridge prefab must be present in the scene (either drag and drop or create a custom version). Enter the IP adress and port of the ROS server to which you want to connect.

Fig. 4.50: ROSBridge

Attach the ROSObject script to the game object which also holds the subscriber, publisher and / or service components.

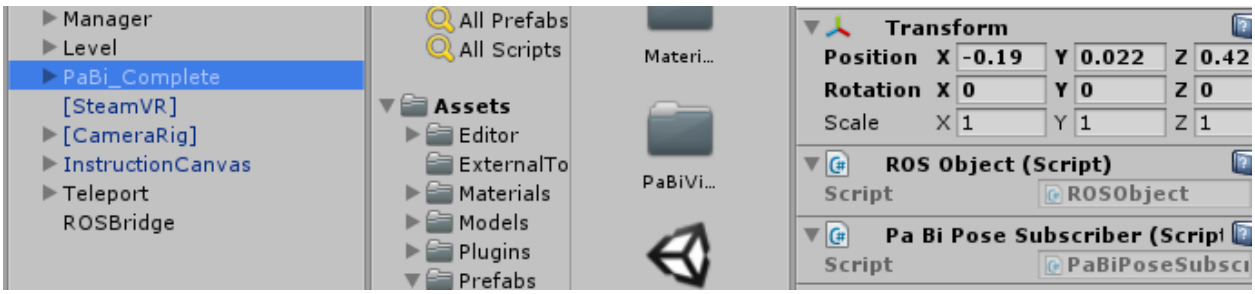


Fig. 4.51: ROSObject

Now, all publishers and subscribers should automatically be registered and announced as well as removed according to the ROSObject state.

4.33 Public Interfaces

4.33.1 ROSBridgeLib

We use the following template from github for the ROSBridge: https://github.com/michaeljenkin/unityros.

Basically the ROSBridge consists of three different parts:

1. ROSBridgeWebSocketConnection
2. ROSBridgeMsg
3. ROSBridge Actor aka Subscriber, Publisher and Service

ROSBridgeWebSocketConnection

class ROSBridgeLib::**ROSBridgeWebSocketConnection**

This class handles the connection with the external ROS world, deserializing json messages into appropriate instances of packets and messages.

This class also provides a mechanism for having the callback's exectued on the rendering thread. (Remember, Unity has a single rendering thread, so we want to do all of the communications stuff away from that.

The one other clever thing that is done here is that we only keep 1 (the most recent!) copy of each message type that comes along.

Version History 3.1 - changed methods to start with an upper case letter to be more consistent with c# style. 3.0 - modification from hand crafted version 2.0

Author Michael Jenkin, Robert Codd-Downey and Andrew Speers

Version 3.1

Public Functions

ROSBridgeLib.ROSBridgeWebSocketConnection.ROSBridgeWebSocketConnection(string host, int
Make a connection to a host/port.

This does not actually start the connection, use Connect to do that.

void ROSBridgeLib.ROSBridgeWebSocketConnection.AddServiceResponse(Type serviceResponse
Add a service response callback to this connection.

void ROSBridgeLib.ROSBridgeWebSocketConnection.RemoveServiceResponse(Type serviceRespo
Not Implemented!

void ROSBridgeLib.ROSBridgeWebSocketConnection.AddSubscriber(Type subscriber)
Add a subscriber callback to this connection.

There can be many subscribers.

void ROSBridgeLib.ROSBridgeWebSocketConnection.RemoveSubscriber(Type subscriber)
Removes a subscriber and disconnects him if the connection is already running.

void ROSBridgeLib.ROSBridgeWebSocketConnection.AddPublisher(Type publisher)
Add a publisher to this connection.

There can be many publishers.

void ROSBridgeLib.ROSBridgeWebSocketConnection.RemovePublisher(Type publisher)
Removes a publisher from the connection.

Disconnects the publisher if connection is already running.

void ROSBridgeLib.ROSBridgeWebSocketConnection.Connect()
Connect to the remote ros environment.

void ROSBridgeLib.ROSBridgeWebSocketConnection.Disconnect()
Disconnect from the remote ros environment.

Private Functions

void ROSBridgeLib.ROSBridgeWebSocketConnection.AnnouncePublishersAndSubscribers()
Announces all publishers and subscribers which were previously added to the lists _publishers and _subscriers

void ROSBridgeLib.ROSBridgeWebSocketConnection.CheckConnection()
Checks if the connection is open and prints a message in that case.

Does not do anything in the other case as *Connect()* has an exception in this case and it wont be printed anyway.

void ROSBridgeLib.ROSBridgeWebSocketConnection.OnError(object sender, WebSocketSharp.E
Error handler method which simply prints the error message.

Private Members

List<string> ROSBridgeLib.ROSBridgeWebSocketConnection.m_AnnouncedTopics = new List<string>()
list containing all publishing topics (only one message type possible -> not necessary) that have been announced so far

List<string> ROSBridgeLib.ROSBridgeWebSocketConnection.m_SubscribedTopics = new List<string>
list containing all subscription topics (only one message type possible -> not necessary to specify) that have been announced so far

ROSBridgeMsg

class ROSBridgeLib::**ROSBridgeMsg**
This (mostly empty) class is the parent class for all RosBridgeMsg's (the actual message) from ROS.

As the message can be empty. . . .

This could be omitted I suppose, but it is retained here as (i) it nicely parallels the ROSBRidgePacket class which encapsulates the top of the ROSBridge messages which are not empty, and (ii) someday ROS may actually define a minimal message.

Version History 3.1 - changed methods to start with an upper case letter to be more consistent with c# style. 3.0 - modification from hand crafted version 2.0

Author Michael Jenkin, Robert Codd-Downey and Andrew Speers

Version 3.1

Subclassed by ROSBridgeLib.custom_msgs.DebugMsg, ROSBridgeLib.custom_msgs.DurationMsg, ROSBridgeLib.custom_msgs.ErrorMsg, ROSBridgeLib.custom_msgs.ExternalForceMsg, ROSBridgeLib.custom_msgs.ExternalJointMsg, ROSBridgeLib.custom_msgs.FloatArrayMsg, ROSBridgeLib.custom_msgs.InfoMsg, ROSBridgeLib.custom_msgs.ModelMsg, ROSBridgeLib.custom_msgs.RoboyPoseMsg, ROSBridgeLib.custom_msgs.RoboyPositionMsg, ROSBridgeLib.custom_msgs.StringArrayMsg, ROSBridgeLib.custom_msgs.TendonInitializationMsg, ROSBridgeLib.custom_msgs.TendonUpdateMsg, ROSBridgeLib.custom_msgs.WarningMsg, ROSBridgeLib.geometry_msgs.PointMsg, ROSBridgeLib.geometry_msgs.PoseMsg, ROSBridgeLib.geometry_msgs.QuaternionMsg, ROSBridgeLib.geometry_msgs.TwistMsg, ROSBridgeLib.geometry_msgs.Vector3Msg, ROSBridgeLib.sensor_msgs.CompressedImageMsg, ROSBridgeLib.sensor_msgs.ImageMsg, ROSBridgeLib.std_msgs.BoolMsg, ROSBridgeLib.std_msgs.ColorRGBAMsg, ROSBridgeLib.std_msgs.HeaderMsg, ROSBridgeLib.std_msgs.Int32Msg, ROSBridgeLib.std_msgs.Int32MultiArrayMsg, ROSBridgeLib.std_msgs.Int64Msg, ROSBridgeLib.std_msgs.Int64MultiArrayMsg, ROSBridgeLib.std_msgs.Int8Msg, ROSBridgeLib.std_msgs.Int8MultiArrayMsg, ROSBridgeLib.std_msgs.MultiArrayDimensionMsg, ROSBridgeLib.std_msgs.MultiArrayLayoutMsg, ROSBridgeLib.std_msgs.StringMsg, ROSBridgeLib.std_msgs.TimeMsg, ROSBridgeLib.std_msgs.UInt16Msg, ROSBridgeLib.std_msgs.UInt16MultiArrayMsg, ROSBridgeLib.std_msgs.UInt32Msg, ROSBridgeLib.std_msgs.UInt32MultiArrayMsg, ROSBridgeLib.std_msgs.Uint64Msg, ROSBridgeLib.std_msgs.UInt64MultiArrayMsg, ROSBridgeLib.std_msgs.UInt8Msg, ROS-

BridgeLib.std_msgs.UInt8MultiArrayMsg, ROSBridgeLib.turtlesim.ColorMsg, *ROSBridgeLib.turtlesim.PoseMsg*, ROSBridgeLib.turtlesim.VelocityMsg

As every type of ROSBridgeMsg should derive from this class, here is an example how an actual implementation looks like.

class ROSBridgeLib::turtlesim::**PoseMsg**

Define a turtle pose message.

This has been hand-crafted from the corresponding turtle message file.

Version History 3.1 - changed methods to start with an upper case letter to be more consistent with c# style. 3.0 - modification from hand crafted version 2.0

Inherits from *ROSBridgeLib.ROSBridgeMsg*

Public Functions

ROSBridgeLib.turtlesim.PoseMsg.PoseMsg(JSONNode msg)

This constructor is called when you receive a message from the ROSBridge.

Parameters

- msg

ROSBridgeLib.turtlesim.PoseMsg.PoseMsg(float x, float y, float theta, float linear_vel

This constuctor can be used to construct a message in Unity and send it over the ROSBridge.

Parameters

- x
- y
- theta
- linear_velocity
- angular_velocity

override string ROSBridgeLib.turtlesim.PoseMsg.ToYAMLString()

You need this function to send a message over the ROSBridge to the desired ROS node as YAML is the standard format for this.

Return

Public Static Functions

static string ROSBridgeLib.turtlesim.PoseMsg.GetMessageType()

This is called when you send the message over the ROSBridge.

It must be equal to the type of the input of the receiving node.

Return

4.33.2 Managers

RoboyManager

class RoboyManager

Roboymanager has the task to adjust roboys state depending on the ROS messages.

In summary it does the following:

```
-# receive pose messages to adjust roboy pose.
-# send a service call for a world reset.
-# FUTURE: receive motor msg and forward it to the according motors.
```

Inherits from Singleton< RoboyManager >

Public Functions

void RoboyManager.InitializeRoboyParts()

Initializes the roboy parts with a random count of motors => WILL BE CHANGED IN THE FUTURE, for now just a template

void RoboyManager.ReceiveMessage(RoboyPoseMsg msg)

Main function to receive messages from ROSBridge.

Adjusts the roboy pose

Parameters

- `msg`: JSON msg containing roboy pose.

void RoboyManager.SendExternalForce(RoboyPart roboyPart, Vector3 position, Vector3 for

Sends a message to the simulation to apply an external force at a certain position.

The position is transformed from Unity to Gazebo space here.

Parameters

- `roboyPart`: The roboypart where the force should be applied.
- `position`: The RELATIVE position of the force to the roboypart.
- `force`: The RELATIVE direction and the amount of force with respect to the roboypart.
- `duration`: The duration for which the force should be applied (in millisecs).

Property

property RoboyManager::Roboy

Public variable so that all classes can access the roboy object.

property RoboyManager::RoboyParts

Public variable for the dictionary with all roboyparts, used to adjust pose and motor values

Private Functions

void RoboyManager.Awake()
Initialize ROSBridge and roboy parts

void RoboyManager.Update()
Run ROSBridge, Reset ROS simulation if Key R is pressed

void RoboyManager.AdjustPose(string name, RoboyPoseMsg msg)
Adjusts roboy pose for all parts with the values from the simulation.

Parameters

- `msg`: JSON msg containing the roboy pose.

void RoboyManager.LocateRoboy()
Searches for all roboys via the "Roboy" tag.

void RoboyManager.LocateRoboyParts(string name)
Searches for all roboy parts for the specified roboy(s).

Private Members

Transform RoboyManager.m_Roboy
Transform of roboy with all roboy parts as child objects

RoboyPoseMsg RoboyManager.m_RoboyPoseMessage
Pose message of roboy in our build in class

Dictionary<string, RoboyPart> RoboyManager.m_RoboyParts = new Dictionary<string, RoboyPart>()
Dictionary with all roboyparts, used to adjust pose and motor values

Dictionary<string, Dictionary<string, RoboyPart> > RoboyManager.m_RoboyPartsList = new D
list of roboy_parts for each roboy (name maps to his parts in a dictionary)

InputManager

class InputManager
InputManager holds a reference of every tool.

On top of that it listens to button events from these tools and forwards touchpad input to the respective classes.

Inherits from Singleton< InputManager >

Public Types

enum TouchpadStatus
Possible touchpad positions.

Values:

Right

Left

Top

Bottom

None

Public Functions

void InputManager.Initialize(List< ControllerTool > toolList)
Initialize all tools.

void InputManager.OnChangeGUITool(object sender, ClickedEventArgs e)
Changes view mode when the user presses the side button on the controller.

Parameters

- sender
- e

void InputManager.OnChangeTool(object sender, ClickedEventArgs e)
Changes the tool when the user presses the side button on the controller.

Parameters

- sender
- e

void InputManager.GetTouchpadInput(object sender, ClickedEventArgs e)
Retrives the touchpad input of the tool controller and updates the values.

Parameters

- sender
- e

Public Members

GameObject InputManager.m_ModeWritingPrefab
Reference to prefab which writes "mode" on one side of the controller

GameObject InputManager.m_ToolWritingPrefab
Reference to prefab which writes "tool" on one side of the controller

Property

property InputManager::GUI_Controller
Public GUIController reference.

property InputManager::View_Controller
Public ViewController reference.

property InputManager::ModelSpawn_Controller
Public ModelSpawnController reference.

property InputManager::Selector_Tool
Public SelectorTool reference.

property InputManager::ShootingTool
Public ShootingTool reference.

property InputManager::TimeTool
Public TimeTool reference.

Private Functions

void InputManager.Update()
Calls the ray cast from the selector tool if it is active.

void InputManager.SetTools(List< ControllerTool > toolList)
Set all tools depending on their type to the respective variable.

Parameters

- toolList

IEnumerator InputManager.initControllersCoroutine()
Initializes all controllers and tools.

Return

Private Members

SelectorTool InputManager.m_SelectorTool
Private SelectorTool reference.

Is serialized so it can be dragged in the editor.

ShootingTool InputManager.m_ShootingTool
Private ShootingTool reference.

Is serialized so it can be dragged in the editor.

TimeTool InputManager.m_TimeTool
Private TimeTool reference.

Is serialized so it can be dragged in the editor.

HandTool InputManager.m_HandTool
Private HandTool reference.

Is serialized so it can be dragged in the editor.

GUIController InputManager.m_GUIController
Private GUIController reference.

Is serialized so it can be dragged in the editor.

ViewController InputManager.m_ViewController
Private GUIController reference.

Is serialized so it can be dragged in the editor.

ModelSpawnController InputManager.m_ModelSpawnController
Private ModelSpawn reference.

Is serialized so it can be dragged in the editor.

SelectionWheel InputManager.m_ToolWheel
Selection wheel to select tools.

SelectionWheel InputManager.m_GUIWheel
Selection wheel to select different GUI modes.

bool InputManager.m_Initialized = false
Controllers initialized or not.

ModeManager

class ModeManager
ModeManager holds a reference of every active mode and provides function to switch between them.

This includes:

- Current tool: ShootingTool, SelectionTool etc.
- Current view mode: singe vs. comparison
- Current GUI mode: selection vs. GUI panels
- Current panel mode: motorforce, motorvoltage etc.

Inherits from Singleton< ModeManager >

Public Types

enum Viewmode
We change between Single view where we can choose only one objet at a time and comparison view with three maximum objects at a time.

Values:

Single

Comparison

enum Panelmode
Describes the different modes for panel visualization.

Values:

Motor_Force

Motor_Voltage

Motor_Current

Energy_Consumption

Tendon_Forces

enum GUIViewerMode
Enum for current GUI mode.

Values:

Selection
Panel of all selectable mesh parts of the current roboy model

MotorValues
Graphs for each motor for each type of information

enum ToolMode
SelectorTool: Select roboy meshes.

ShooterTool: Shoot projectiles at roboy. TimeTool: Reverse/stop time.

Values:

SelectorTool

ShootingTool

TimeTool

HandTool

Undefined

Public Functions

void ModeManager.ChangeViewMode()
Changes between single and comparison view.

void ModeManager.ChangeGUIViewerMode()
Switches between selection and panels GUI mode.

void ModeManager.ChangeToolMode()
Switches between all tools.

Deactivates previous tool and activates new one

void ModeManager.ChangeGUIToolMode(ControllerTool tool)
Changes the current mode, disables previous one and informs VRUILogic singleton

Parameters

- `tool`

void ModeManager.ChangePanelModeNext()
Changes the panel mode to the next one based on the order in the enum defintion.

void ModeManager.ChangePanelModePrevious()
Changes the panel mode to the previous one based on the order in the enum defintion.

void ModeManager.ResetPanelMode()
Resets current panel mode to MotorForce.

Property

property ModeManager::CurrentViewmode
Current view mode, READ ONLY.

property ModeManager::CurrentPanelmode
Current panel mode, READ ONLY.

property ModeManager::CurrentGUIViewerMode
Current GUIViewer mode, READ ONLY.

property ModeManager::CurrentToolMode
Current Tool mode, READ ONLY.

property ModeManager::CurrentGUIMode
Current GUI Mode.

READ ONLY.

Private Functions

void ModeManager.SetNewActiveTool(ControllerTool tool)
deactivates currently active tool and activates new tool

Parameters

- `tool`

Private Members

Viewmode ModeManager.m_CurrentViewmode = Viewmode.Comparison
Private variable for current view mode.

Panelmode ModeManager.m_CurrentPanelmode = Panelmode.Motor_Force
Private variable for current panel mode.

GUIViewerMode ModeManager.m_CurrentGUIViewerMode = *GUIViewerMode.Selection*
Private variable for current GUIViewer mode.

SpawnViewerMode ModeManager.m_CurrentSpawnViewerMode = SpawnViewerMode.Insert
Private variable for the current mode of the model spawn controller.

ToolMode ModeManager.m_CurrentToolMode = ToolMode.SelectorTool
Private variable for current Tool mode.

GUIMode ModeManager.m_CurrentGUIMode = GUIMode.GUIViewer
Current view mode of the GUI tools

ControllerTool ModeManager.m_CurrentlyActiveTool
Holds reference to currently active controller tool from inputmanager

SelectorManager

class SelectorManager
SelectorManager is responsible to hold references of all selected roboy parts and the corresponding UI elements.

Inherits from Singleton< SelectorManager >

Public Functions

void SelectorManager.AddSelectedObject(SelectableObject obj)
Adds the roboy part to selected objects.

Parameters

- `obj`: SelectableObject component of the roboy part.

void SelectorManager.RemoveSelectedObject(SelectableObject obj)
Removes the roboy part from the selected objects.

Parameters

- `obj`: SelectableObject component of the roboy part.

void SelectorManager.ResetSelectedObjects()
Resets all roboy parts to default state and empties the selected objects list.

Public Members

int SelectorManager.RoboyUIElementsCount = 13
TEMPORARY VARIABLE TO CHECK HOW MANY UI ELEMENTS ARE INITIALIZED

Property

property SelectorManager::UI_Elements
Property which returns a dictionary of all UI elements in the SelectionPanel.

property SelectorManager::SelectedParts
Reference of all currently selected roboy parts.

property SelectorManager::MaximumSelectableObjects
Integer to switch between single mode selection and normal mode collection.

Private Functions

IEnumerator SelectorManager.Start()
Initializes all variables.

Return

Private Members

Transform SelectorManager.m_Roboy
Transform of roboy model.

List<SelectableObject> SelectorManager.m_RoboyParts = new List<SelectableObject>()
List of SelectableObject components of all roboy parts.

List<SelectableObject> SelectorManager.m_SelectedParts = new List<SelectableObject>()
List of SelectableObject components of all selected parts.

int SelectorManager.m_MaximumSelectableObjects = 3
Maximum cound of selectable objects in multiple selection mode.

int SelectorManager.m_CurrentMaximumSelectedObjects = 3
Current count of maximum selectable objects.

Dictionary<string, GameObject> SelectorManager.m_UI_Elements = new Dictionary<string, GameObject
Private reference to all UI elements.

BeRoboyManager

class BeRoboyManager
BeRoboymanager has different tasks to do:

1. Keep track of user movement and translate roboy when in specific view modes
2. Convert received images into textures which can then be rendered on screen
3. Send tracking messages over the rosbridge to gazebo/ real roboy

Inherits from Singleton< BeRoboyManager >

Public Functions

void BeRoboyManager.ReceiveZedMessage(ImageMsg image)
Primary function to receive image (zed) messages from ROSBridge.

Renders the received images.

Parameters

- `msg`: JSON msg containing roboy pose.

void BeRoboyManager.ReceiveSimMessage(ImageMsg image)
Primary function to receive image (simulation) messages from ROSBridge.

Renders the received images.

Parameters

- `msg`: JSON msg containing roboy pose.

void BeRoboyManager.PublishExternalJoint(List< string > jointNames, List< float > angl
Function to publish ExternalJoint messages via ROS.

Parameters

- `jointNames`
- `angles`

void BeRoboyManager.PublishRoboyPosition(Vector3 pos)
Function to publish Position messages via ROS.

Parameters

- `pos`

Public Members

bool BeRoboyManager.TrackingEnabled = false
Set whether head movement should be tracked or not.

RenderTexture BeRoboyManager.RT_Zed
Reference to the render texture in which the Zed feed gets pushed into.

RenderTexture BeRoboyManager.RT_Simulation
Reference to the render texture in which the Simulation feed gets pushed into.

Private Functions

void BeRoboyManager.Awake()
Initialize textures.

void BeRoboyManager.RefreshZedImage(ImageMsg image)
Renders the received images from the zed camera

Parameters

- msg: JSON msg containing the roboy pose.

void BeRoboyManager.RefreshSimImage(ImageMsg image)
Renders the received images from the simulation.

Parameters

- msg: JSON msg containing the roboy pose.

void BeRoboyManager.translateRoboy()
Turn Roboy with the movement of the HMD.

void BeRoboyManager.tryInitializeCamera()
Looking for the main camera in the scene, which can be attached to Roboy.

Private Members

GameObject BeRoboyManager.m_Cam
The HMD main camera.

Texture2D BeRoboyManager.m_TexSim
Texture in which the received simulation images get drawn.

Texture2D BeRoboyManager.m_TexZed
Texture in which the received zed images get drawn.

bool BeRoboyManager.m_CamInitialized = false
Is the main camera initialized or not.

float BeRoboyManager.m_CurrentAngleX = 0.0f
Variable to determine if headset was rotated.

float BeRoboyManager.m_CurrentAngleY = 0.0f
Variable to determine if headset was rotated.

Color [] BeRoboyManager.m_ColorArraySim = new Color[640 * 480]
Color array for the simulation image conversion.

Color [] BeRoboyManager.m_ColorArrayZed = new Color[1280 * 720]
Color array for the zed image conversion.

SteamVR_TrackedObject BeRoboyManager.m_Controller
Reference to the tools controller (right).

ViewSelectionManager

class ViewSelectionManager
ViewSelectionManager handles the transition between various view scenarios.

Inherits from Singleton< ViewSelectionManager >

Public Functions

void ViewSelectionManager.TurnTrackingOn()
Turn head tracking for BeRoboy on.

void ViewSelectionManager.TurnTrackingOff()
Turn head tracking for BeRoboy off.

void ViewSelectionManager.SwitchToSimulationView()
Switches the view to the simulation view.

void ViewSelectionManager.SwitchToZEDView()
Switches the view to the ZED(real roboy camera in the head) view.

void ViewSelectionManager.SwitchToObserverView()
Switches the view to the observer view.

void ViewSelectionManager.SwitchToBeRoboyView()
Switches the view to the beroboy view.

Public Members

Canvas ViewSelectionManager.InstructionCanvas
Reference to the Canvas that is placed on the Camera plane(HMD).

Image ViewSelectionManager.BackgroundImage
Reference to the image where intructive text can be displayed.

RawImage ViewSelectionManager.GazeboImage
Reference to the image where the simulation feed can be displayed.

Image ViewSelectionManager.HtcImage
Reference to the image where the htc feed can be displayed.

RawImage ViewSelectionManager.ZedImage
Reference to the image where the zed feed can be displayed.

GameObject ViewSelectionManager.Cave
Reference to Roboy's surrounding.

GameObject ViewSelectionManager.Roboy
Reference to Roboy itself.

GameObject ViewSelectionManager.Pointer
Reference to the teleporter ray.

4.34 ZED Wrapper

NOTICE : LEGACY CODE. NOT MAINTAINED

Introduction The ZED camera works together with the API to create a spatial representation of the environment. It can also be used to track the position of the user but for that we already use the HTC Vice. Unfortunately the the Unity plugin for the ZED API does not expose the spatial mapping API. But we wanted to do a case study to test the possibilities and performance of the camera. Therefore we wrote our own wrapper which maps the c++ API to Unity. You can find the c++ code here and the c# code here.

Implementation The difficult part was to figure out how to successfully transfer the results from the c++ dll to Unity. After a lot of trial and error we settled with the approach to send data to Unity in sperate arrays.

The spatial scan API saves the result in an internal *Mesh* class. This class has various arrays for the vertices, triangles, normals and uvs. On top of that it structures the vertices in *Chunks to make things faster. This means that every time the camera sees a new part it saves it in a chunk instead of going through the whole vertices array. So to send the retrieved data the plugin offers functions to retrieve the mesh arrays. So the wanted usage of the plugin is as follows:

1. Start the spatial mapping process.
2. Run the main spatial mapping loop each frame/time step.
3. Retrieve the mesh each time you get a new update to see the live progress.
4. Stop the spatial mapping process.
5. Get the final mesh and the texture for color information.

To retrieve the mesh you just have to call the functions to get the mesh arrays and save them in the mesh representation in Unity. This means you have to import the needed functions and call them. In our example we call the mapping loop in a seperate thread for performance reasons.

Results There is a good and a bad message. The good message first. It works, kinda. So the result of the scan itself is somewhat right. However, there is a problem with the scaling and orientation. It is way too big, of factor 1000, and it is turned upside-down. We tried different approaches but did not find out the reasons for these problems.

The bad message is that the live progress has a lot of artifacts. This may be because of the multithreading approach. We tried to make it threadsafe via locking and only reading when the mappingLoop is not writing but for some reason it did not solve the artifacts completely but rather just reduced them. This problem makes the usage of this appraoch in VR impossible. The user would fill sick and the vision of him would be completely clustered by seemingly random triangles.

Another problem is that, as the standard plugin and the extension hold both a reference to the zed camera they cannot be used together. Therefore you cannot start the spatial mapping process while getting a live stream of the camera. This is unfortunate as the wanted result was exactly this behaviour.

Summerized, in the current state it is more of a presentation that it can work in Unity with a decent performance but lacks stability and robustness to use it in a real-case scenerio.

4.35 Coordinate Systems

4.35.1 General

Coordinate systems present an immense hazard when implementing messages from Unity to Gazebo or back. In case these messages contain positions, rotations and directions, it is of **utmost improtance** to transform coordinates into the **expected coordinate spaces**:

- Unity's coordinate system:
- The x coordinate points to the right (right edge of the screen)
- The y coordinate points upwards (top of the screen)
- The z coordinate points into the scene, away from the user
- Gazebo's coordinate system:
- The x coordinate points to the right (right edge of the screen)
- The y coordinate points into the scene, away from the user

- The z coordinate points upwards (top of the screen)

NOTICE: y and z coordinate are swapped when converting coordinates into the other coordinate system

Implementation

A utility class called *GazeboUtility* provides conversions for both angles and rotations from and to Gazebo coordinates. All defined messages automatically transform the values, depending on if they were parsed from a received message or created using the Unity constructors.

4.35.2 World and local space

Another important aspect to consider: Positions and forces can be defined in world space, giving the absolute position / direction, or they can be defined in local space. When values are defined in local space, the local coordinate system always needs to be provided as well.

Message examples The *ExternalForceMessage* is an exellent example: A force is **defined locally**, with its relative position and relative direction specified **along with the Roboy part** . The part provides the local coordinate system. Other messages may specify positions locally or in worls space, when in doubt: The message definition *should* provide further information, otherwise check the receiving entity how it handles the values.

Simulations in Gazebo In gazebo, every position can be queried in local or world coordinate space, and forces can be applied locally (*AddRelativeForce(. . .)*) or in world space (*AddWorldForce(. . .)*).

Issues When applying relative forces in Gazebo, the results never matched the given forces and directions. One option, how to solve this issue, would be to convert the local values to world coordinate space. Unfortunately, during the given time, no function was found which transforms the values correctly (The scale of the coordinate systems might be one of the reasons it failed).

For now, external forces are defined **in World Space** and the respective gazebo simulation applies forces in world space as well.

4.36 Troubleshooting Hacks

4.36.1 VIVE Headset

If the HTC Vive headset isn't working at all:

- USB: Unplug the usb cable connecting the Headset with the computer you are using. Simply plug back in - in case another port is used, he drivers are newly chosen by Windows.
- Restart SteamVR: Click the SteamVR drop-down menu, select Settings. Choose the Developer tab, scroll to the bottom and click **Reboot Vive Headset**.
- Restart Steam: This works in case Steam tried to update SteamVR and failed.

If there are problems with tracking:

- Make sure the lighthouses can both see each other and at least one of them can see the headset at any point in time.
- If controllers aren't tracked make sure their batteries are charged. You can also sync them manually by right clicking on one of them in SteamVR.
- If the tracking is off or doesn't work at all, try doing the SteamVR room setup.

If the position is not optimal:

- Open the SteamVR drop-down menu, click **Run Room Setup**.

4.36.2 Unity

Unity sometimes displays unexpected behaviour contrary to what is coded and/or expected. First advise: Restart Unity - this may sound banal, but it would have saved our team at least an hour of debugging.

4.36.3 Simulation

1. If gazebo fails to start server or they die unexpectedly:

   Kill the gazebo server and restart it.

   ```
   killall gzserver
   killall gzclient
   ```

   Another possible fix is to kill Roscore

   ```
   killall roscore
   ```

2. In case Models could not be found - which is often the reason when simulations take forever to load:

   Export the gazebo paths to the model: General Usage/Getting started/

3. If you are still having trouble, please contact the Roboy Team. We will glady help you to set up your RoboyVR experience.

4.37 Presentations

Midterm WS16/17

Endterm WS16/17

Midterm SS17

Endterm SS17

Midterm WS2017/18

Endterm WS17/18

Youtube

4.38 About arc42

This information should stay in every repository as per their license: http://www.arc42.de/template/licence.html

arc42, the Template for documentation of software and system architecture.

By Dr. Gernot Starke, Dr. Peter Hruschka and contributors.

Template Revision: 6.5 EN (based on asciidoc), Juni 2014

© We acknowledge that this document uses material from the arc 42 architecture template, http://www.arc42.de. Created by Dr. Peter Hruschka & Dr. Gernot Starke. For additional contributors see http://arc42.de/sonstiges/contributors.html

Note

This version of the template contains some help and explanations. It is used for familiarization with arc42 and the understanding of the concepts. For documentation of your own system you use better the *plain* version.

4.38.1 Literature and references

Starke-2014 Gernot Starke: Effektive Softwarearchitekturen - Ein praktischer Leitfaden. Carl Hanser Verlag, 6, Auflage 2014.

Starke-Hruschka-2011 Gernot Starke und Peter Hruschka: Softwarearchitektur kompakt. Springer Akademischer Verlag, 2. Auflage 2011.

Zörner-2013 Softwarearchitekturen dokumentieren und kommunizieren, Carl Hanser Verlag, 2012

4.38.2 Examples

- HTML Sanity Checker
- DocChess (german)
- Gradle (german)
- MaMa CRM (german)
- Financial Data Migration (german)

4.38.3 Acknowledgements and collaborations

arc42 originally envisioned by Dr. Peter Hruschka and Dr. Gernot Starke.

Sources We maintain arc42 in *asciidoc* format at the moment, hosted in GitHub under the aim42-Organisation.

Issues We maintain a list of open topics and bugs.

We are looking forward to your corrections and clarifications! Please fork the repository mentioned over this lines and send us a *pull request*!

4.38.4 Collaborators

We are very thankful and acknowledge the support and help provided by all active and former collaborators, uncountable (anonymous) advisors, bug finders and users of this method.

Currently active

- Gernot Starke
- Stefan Zörner
- Markus Schärtel
- Ralf D. Müller
- Peter Hruschka
- Jürgen Krey

Former collaborators

(in alphabetical order)

- Anne Aloysius
- Matthias Bohlen
- Karl Eilebrecht
- Manfred Ferken
- Phillip Ghadir
- Carsten Klein
- Prof. Arne Koschel
- Axel Scheithauer

Index

B

BeRoboyManager (C++ class), 89

I

InputManager (C++ class), 82
InputManager::Bottom (C++ enumerator), 82
InputManager::Left (C++ enumerator), 82
InputManager::None (C++ enumerator), 82
InputManager::Right (C++ enumerator), 82
InputManager::Top (C++ enumerator), 82
InputManager::TouchpadStatus (C++ type), 82

M

ModeManager (C++ class), 85
ModeManager::Comparison (C++ enumerator), 85
ModeManager::Energy_Consumption (C++ enumerator), 85
ModeManager::GUIViewerMode (C++ type), 85
ModeManager::HandTool (C++ enumerator), 86
ModeManager::Motor_Current (C++ enumerator), 85
ModeManager::Motor_Force (C++ enumerator), 85
ModeManager::Motor_Voltage (C++ enumerator), 85
ModeManager::MotorValues (C++ enumerator), 85
ModeManager::Panelmode (C++ type), 85
ModeManager::Selection (C++ enumerator), 85
ModeManager::SelectorTool (C++ enumerator), 86
ModeManager::ShootingTool (C++ enumerator), 86
ModeManager::Single (C++ enumerator), 85
ModeManager::Tendon_Forces (C++ enumerator), 85
ModeManager::TimeTool (C++ enumerator), 86
ModeManager::ToolMode (C++ type), 86
ModeManager::Undefined (C++ enumerator), 86
ModeManager::Viewmode (C++ type), 85

R

RoboyManager (C++ class), 81
ROSBridgeLib::ROSBridgeMsg (C++ class), 79
ROSBridgeLib::ROSBridgeWebSocketConnection (C++ class), 77
ROSBridgeLib::turtlesim::PoseMsg (C++ class), 80

S

SelectorManager (C++ class), 87

V

ViewSelectionManager (C++ class), 90