roboyVR

Release 0.0.1

Jun 27, 2017

Usage and Installation

What is it?

RoboyVR is a virtual reality experience in which the user can watch, but also interact with roboy (a humanoid robot) while he performs specific tasks, e.g. walking, waving, etc. A virtual environment opens up a whole new set of perspectives for the user to enjoy and spectate roboy from all kinds of POVs. In addition to the rendering of the virtual roboy a mobile HUD shows detailed information about various roboy components, for example displaying the powerconsuption of particular motors. As the user chooses to take a more active part, roboy's pose can be influenced and altered by physical contact, e.g. shooting a projectile at the virtual model.

How does it work?

Roboy and its behavior is simulated on a virtual machine via Gazebo/ROS. Important information regarding roboy's movement are then sent through a ROSbridge(e.g. messages) towards Unity. In Unity roboy is rendered and constantly updated concerning positions, rotations, etc. On top of that detailed data (time lapsed) about components is displayed via graph rendering on different UI panels. With the help of a VR-Headset you can watch roboy move around in a virtual space.

Current status of the project and goals

Currently the project can render roboy with his pose and generate random data about his motors to visualize them. Our next tasks are as follow:

- Use real motor data and visualize that.
- Implement an interface to track the newest models and automate the process of creating the model in Unity.
- Implement an interface to record a simulation with all the data and save/ load it on runtime.
- Make the project completely Plug&Play meaning that you can send all kinds of data with a given format.

CHAPTER 1

Relevant Background Information and Pre-Requisits

For the user:

One of roboyVR design goals is to be as user friendly and intuitively as possible. Therefore the explorer in the virtual reality does not need to be familiar with explicit requirements. 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 is important for a frust-free experience, you can adjust the distance from the lenses to your eyes as well as the distance between the lenses itself, these tweaks help immensely when it comes to maintaining a sharp field of view.

Apart from that the tracking system needs to be setup correctly, too. 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.

For the developer:

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

The roboy simulation which runs on Gazebo/ROS is written in C++, for this section a basic overview is sufficient to be able to understand/construct messages which are then sent via a ROSbridge. For starting the simulation you should be familiar with Linux/Ubuntu. Further it is useful to have some understanding of python in order to transform the roboy models via the Blender-api(an early python script already exists for this purpose).

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: https://unity3d.com/de/learn/tutorials
- The UnifyWiki has a lot of example scripts for all kind of extensions: http://wiki.unity3d.com/index.php/Main_Page
- StackOverflow is a forum where you can search for answers regarding your coding problems: http://stackoverflow.com/
- 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: http://answers.unity3d.com/

- As we use ROS and our own custom messages, it is important to understand how ROS works and how ROS messages are built: http://wiki.ros.org/

If you have any further questions about the project, feel free to contact us via email: roboyvr@gmail.com

CHAPTER 2

Contents:

Installation

Roboy and its behavior is simulated on the virtual machine via ROS. Important information regarding roboy's movement are then sent through a ROSBridge(e.g. messages) towards Unity. 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.

Part 1: Setup Virtualbox with Ubuntu [OPTIONAL]

1. Download and install Virtualbox for your OS https://www.virtualbox.org/
2. Download Ubuntu 16.04 (64bit) https://www.ubuntu.com/download/desktop
3. Mount the .iso and setup Virtualbox with the following settings (if available):
1. 4 cores (Settings->System->Processor)
2. 6 GB of RAM (Settings->System->Motherboard)
3. 128 MB of VRAM (Settings->Display->Screen)
4. 30 GB HDD space (Settings->Storage)
4. Set network settings to Bridged-Adapter or Host-Only Adapter

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.

Part 3: Unity Setup

1. Download Unity

- (latest working version with roboyVR is 5.6.1: 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/sheveg/roboyVR.git
- Command: git clone -b master https://github.com/sheveg/roboyVR.git

Part 4: Blender & Python

- Install the latest version of Blender
- Install the latest version of Python
- 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/)

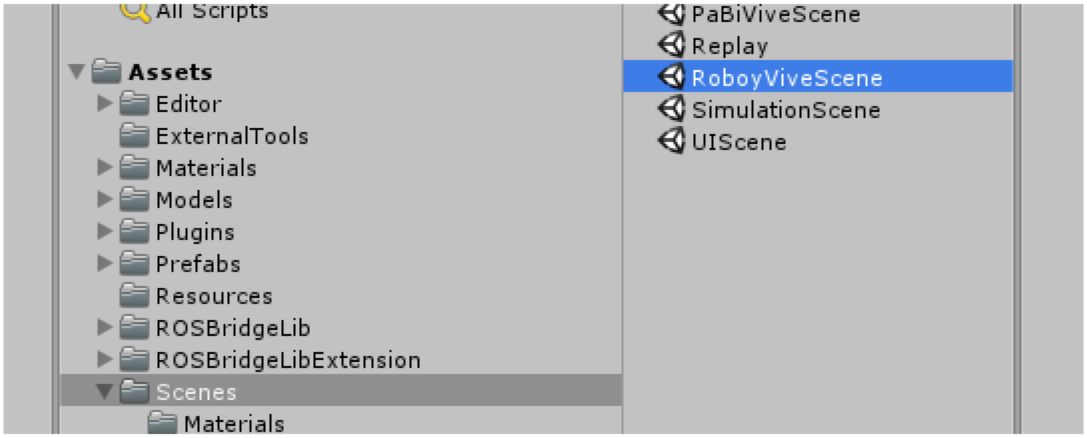
Getting started

Part 1: Run rosbridge and roboySimulation

```
source path-to-roboy-ros-control/devel.setup.bash
roslaunch rosbridge_server rosbridge_websocket.launch
rosrun roboy_simulation VRRoboy
```

Part 2: Open the project in Unity

Unity is organized in Scenes. In order to watch the simulation in Unity which is running on the VM or on another machine(in gazebo), open the RoboyViveScene.



Part 3: Setup the scene

In the Scene you can observe the simulation from the VM within Unity. To do that you need to communicate the IP adress of your VM towards the ROSBridge. The IP information is quickly found in Ubuntu by clicking on the two arrows pointing in opposite directions, right next to the system time. Afterwards a drop down menu will open, click on connection information. Remember the IP and paste it in the respective field in Unity.

You also need to drag the roboy prefab onto the RoboyManager if it is not already done. Each roboy model is tagged as a *RoboyPart*. If you import new models for roboy you need to change the tag accordingly and change the roboy prefab.

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*. Just follow the instructions on the screen to setup the controllers.

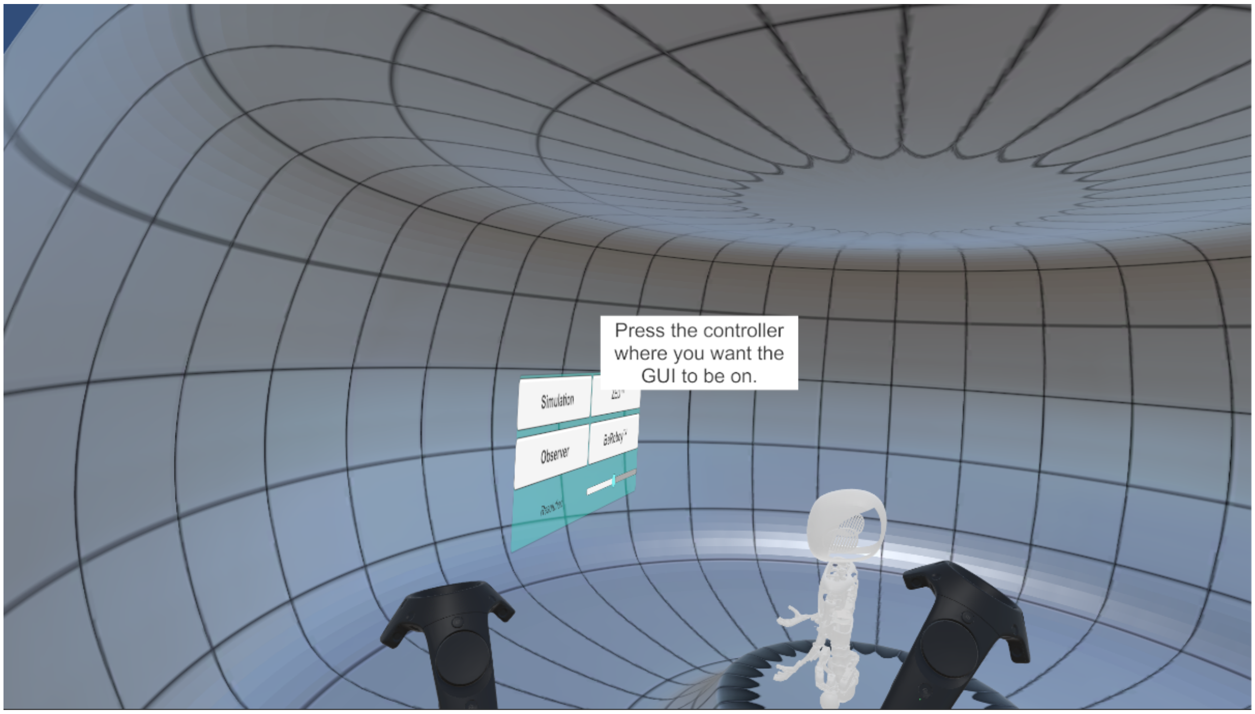
To get a better view of the simulation we recommend to set the simulation to slow motion in rviz in the VM:

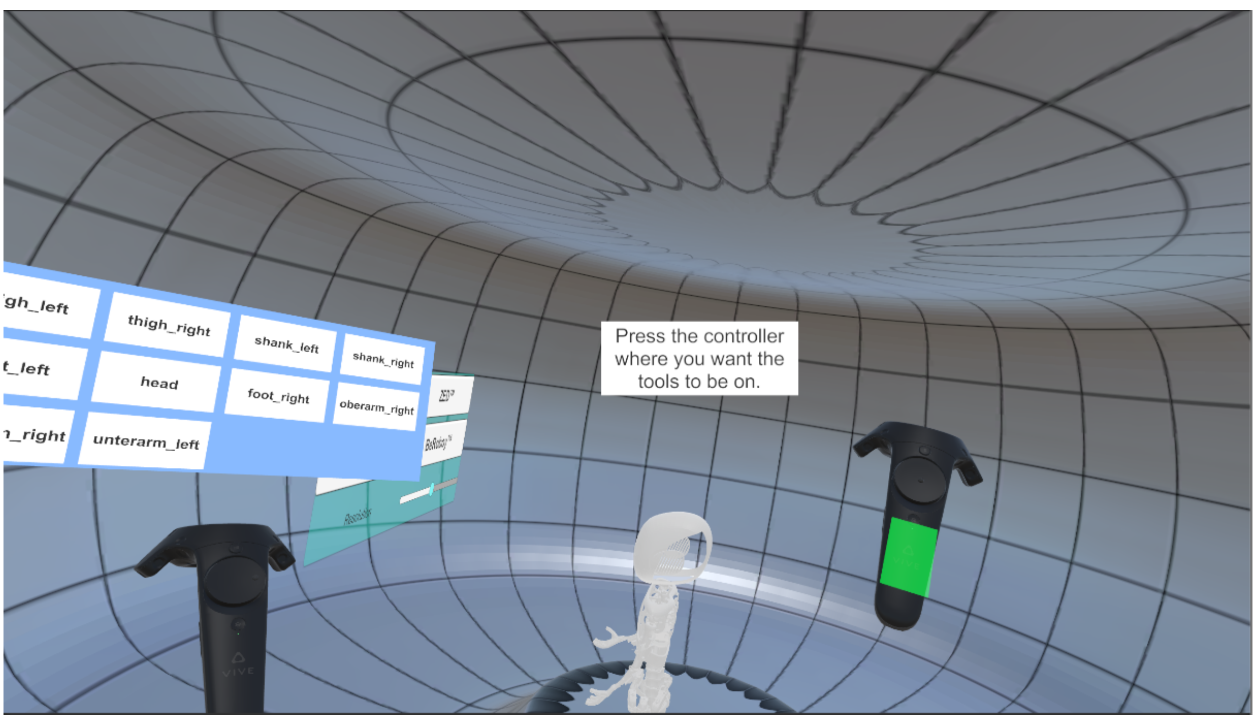
- If you want to start rviz, open a terminal (in the VM) and simply type **rviz**
- Set Fixed Frame to World (Displays->Fixed Frame)
- Add a marker (Add(Button)->marker)
- Add walking plugin (Panels->Add New Panel->WalkingPlugin)
- Turn slow motion on (within the walking plugin, it is a toggle button)

Introduction

What is it?

The Model Updater is a convinient Unity Editor extension, where you can download new models or update existing ones to use in RoboyVR.


Press the controller
where you want the
GUI to be on.
Simulation
Observer


thigh_right
shank_left
shank_right
head
foot_right
oberarm_right
unterarm_left
Press the controller
where you want the
tools to be on.

How does it work?

After a short setup, where you have to select the blender.exe and set the github_repo path, you can scan the Github Repository for models. Now a list of found models appears and you can select, which you want to download. Pressing "Download" loads the models and additionally converts them with blender to .fbx, so you can use them in Unity. 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, which you can easily just drag and drop into the VR scene.

User's Manual

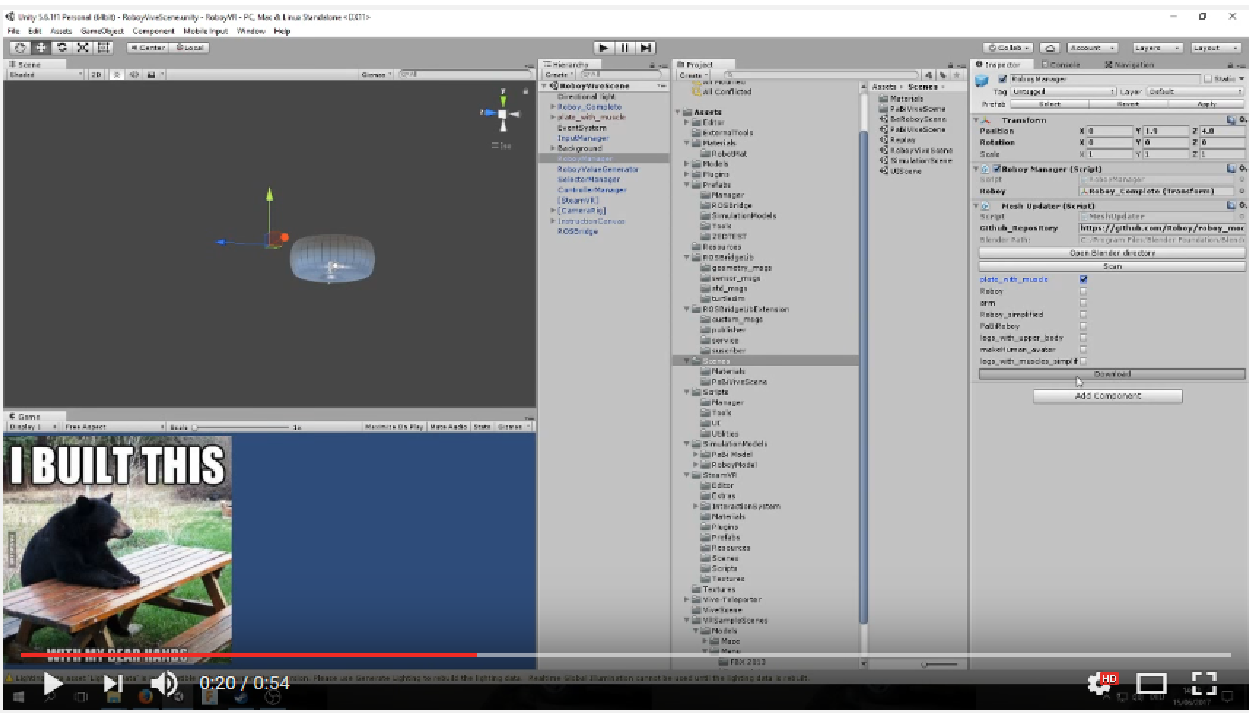


Fig. 2.1: Video showing the MeshUpdater

Part 1: Getting started

Open the Unity project RoboyVR. Open the RoboyViveScene, and select the RoboyMananger in the hierarchy tab. The RoboyManager has a script called Mesh Updater. The following instructions all are entered here.

Part 2: 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. Also you can set here, which branch you want to download the models from. As of right now you may need to change the branch to "VRTEAM", since we overhauled the folder structure and model.sdf files.

Default:

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

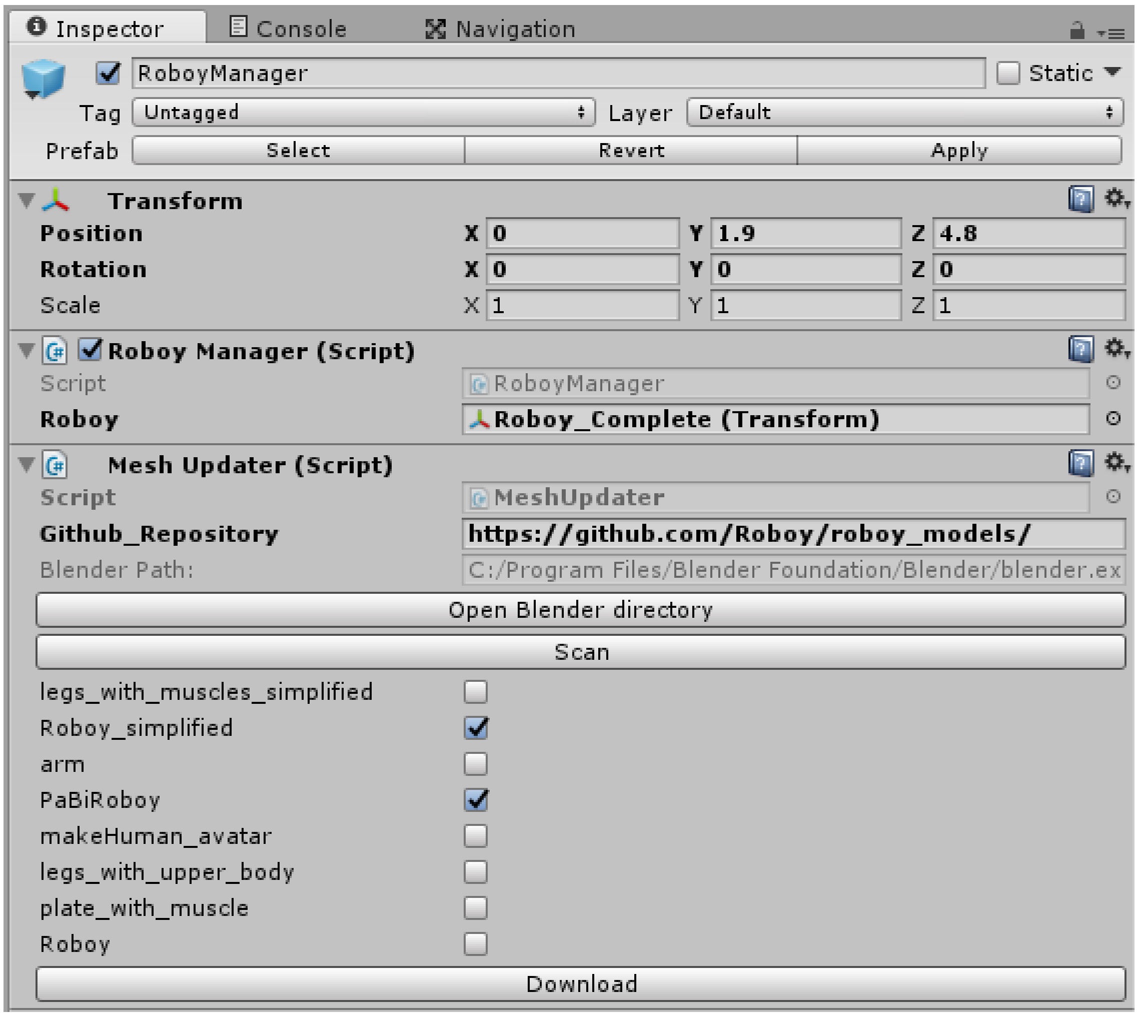


Fig. 2.2: Mesh Updater GUI

- Branch = master

Part 3: Set Blender.exe

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

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

Part 4: Scanning

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

Part 5: 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.

Part 6: Create the Prefab

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

Developer's Manual

Summary

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. 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

Part 1: 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.

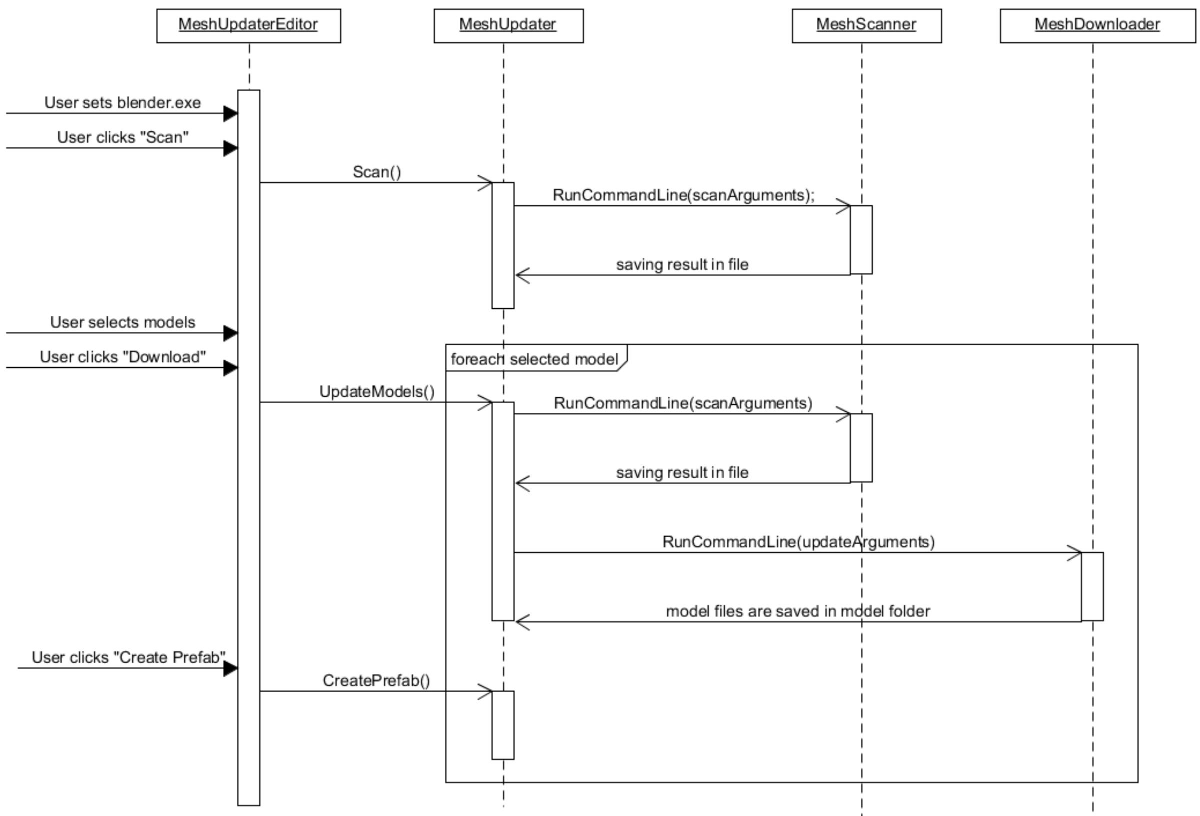


Fig. 2.3: Sequence Diagram for MeshUpdater

Also the state now shows a button called “Scan”. This button calls meshUpdater.Scan() (*Part 2: Scanning*). 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() (*Part 3: Downloading*), 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() (*Part 4: Create Prefab*).

Part 2: 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”.

Part 3: Downloading

MeshUpdater.UpdateModels(): For every entry in ModelChoiceDictionary that is true, the ModelScanner.py is used to get each subfolder. 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.

Part 4: 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. 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.

Introduction

What is it?

BeRoboy™ is taking the RoboyVR experience to the next level. With the help of seamless full body tracking the user can send commands to Roboy, take control over Roboy and become THE Roboy. BeRoboy™ puts the user in the driver seat and provides a fully immersive experience like never before. With BeRoboy™ you can control various

Fig. 2.4: BeRoboy™ is the next big thing in the world of virtual reality robots.

different versions of Roboy. This includes a Roboy in VR, in a gazebo simulation and even the real one. You want Roboy to throw a punch, shake a leg or make obscene gestures? BeRoboy™ lets you do all of that and a lot more!

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 the gazebo Roboy or the real one, 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 connected version of Roboy.

User's Manual

This manual will descripe the steps required to start BeRoboy™ and begin your journey.

Starting Gazebo

Start your Ubuntu machine and open a terminal.

1. Source the setup.bash

```
source /path-to-roboy-ros-control/devel.setup.bash
```

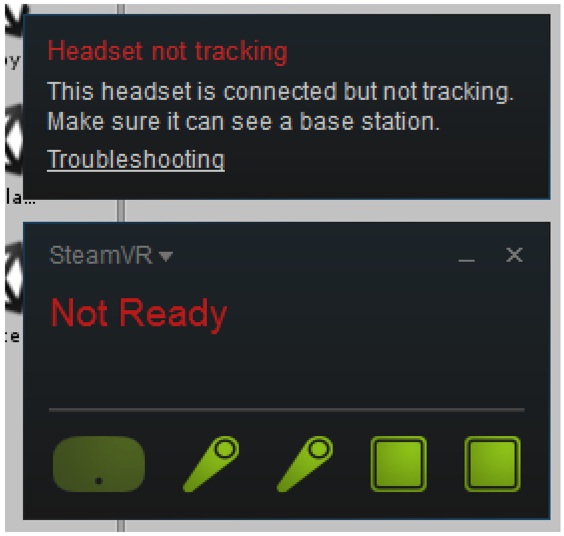
2. Start the launch file which starts Gazebo with the Roboy and a Camera ROS node

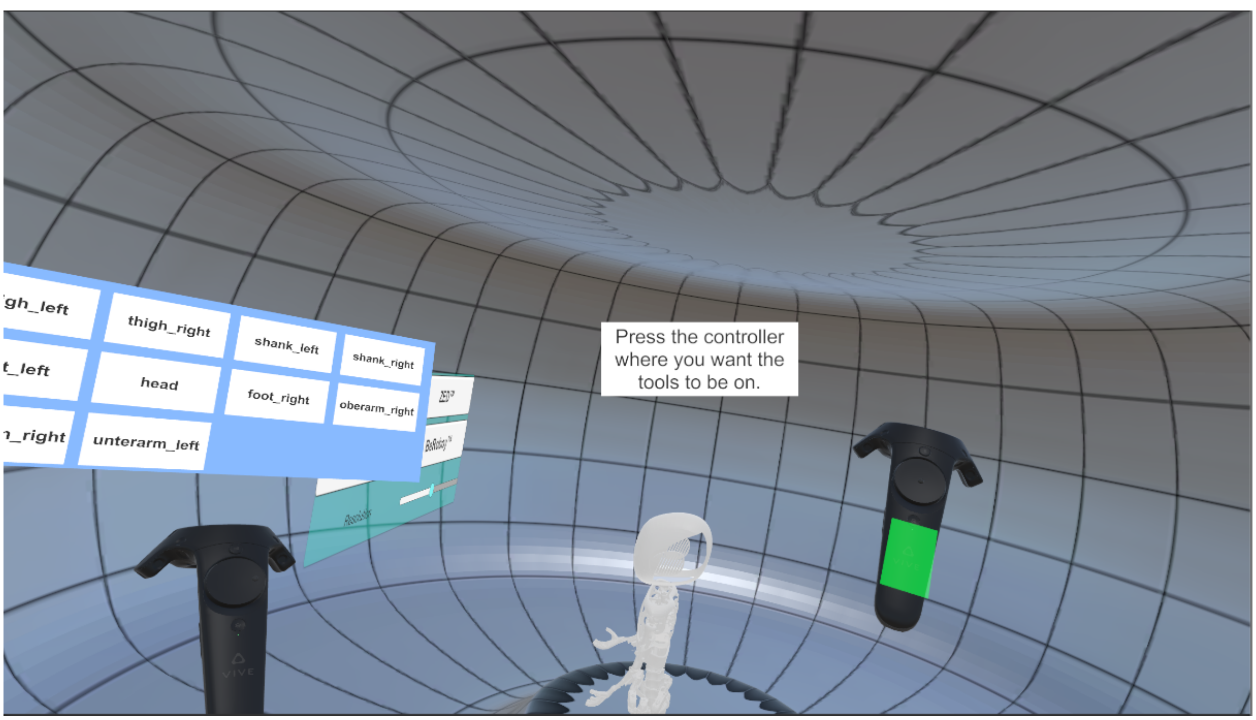
```
roslaunch roboy_simulation camera_test.launch
```

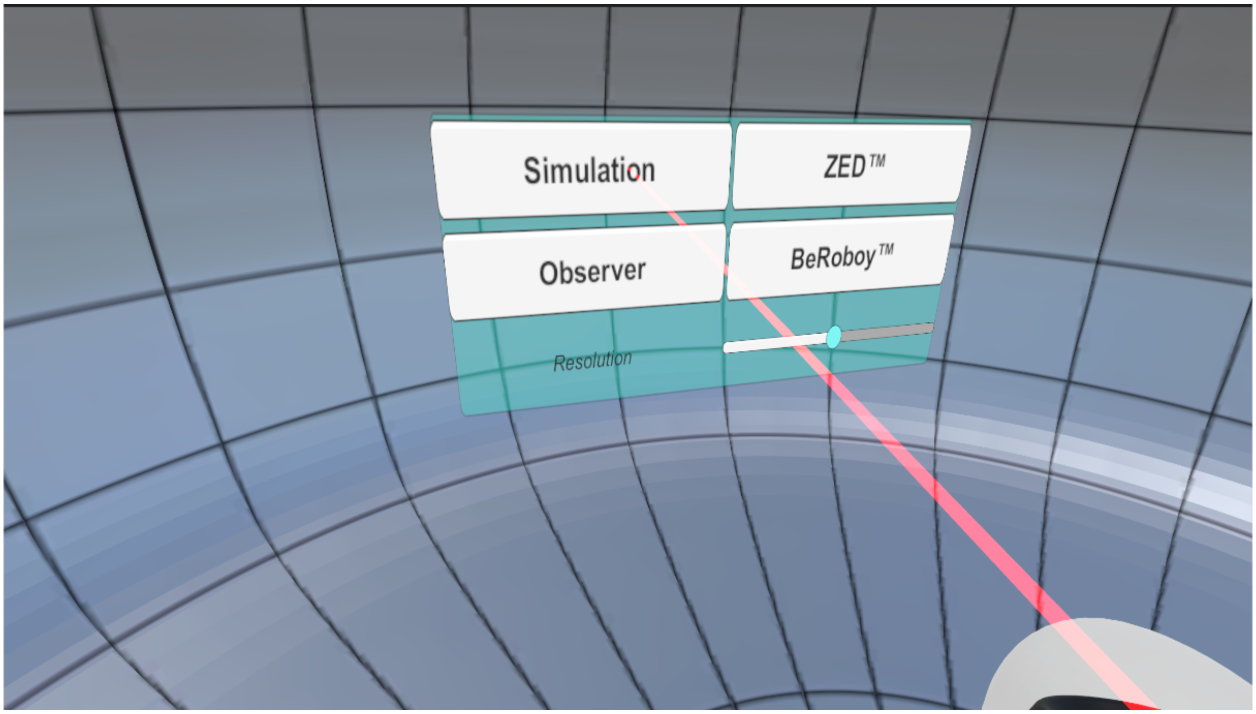
Starting Unity

1. Start the unity project inside the git repo you cloned to your hard drive.
2. Inside unity select the RoboyVR scene.
3. In the ROSBridge (located in the hierarchy) type in the correct IP of your Ubuntu machine.

4. Start the scene.
5. SteamVR should also start, if this throws errors (like "SteamVR unresponsive, not working, etc."), simply restart it.
6. When the scene starts properly, you can choose which controller should hold which tools.
7. After the controller assignment, you can switch between various view modes via a selection menu in the scene.
8. Enjoy your stay!


Headset not tracking
This headset is connected but not tracking.
Make sure it can see a base station.
Troubleshooting
SteamVR
Not Ready


thigh_right
shank_left
shank_right
head
foot_right
oberarm_right
unterarm_left
Press the controller where you want the tools to be on.



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

Troubleshooting

If gazebo encounters problems loading the Model into the world or starting the server, these commands could be useful.

1. Kill the gazebo server and restart it.

```
killall gzserver
killall gzclient
```

2. Export the gazebo paths to the model

```
source /usr/share/gazebo-7/setup.sh
export GAZEBO_MODEL_PATH=/path/to/roboy-ros-control/src/roboy_models:$GAZEBO_MODEL_
↪PATH
```

3. If you are still having trouble, please contact roboyvr@gmail.com. We will glady help you to enjoy your RoboyVR experience.

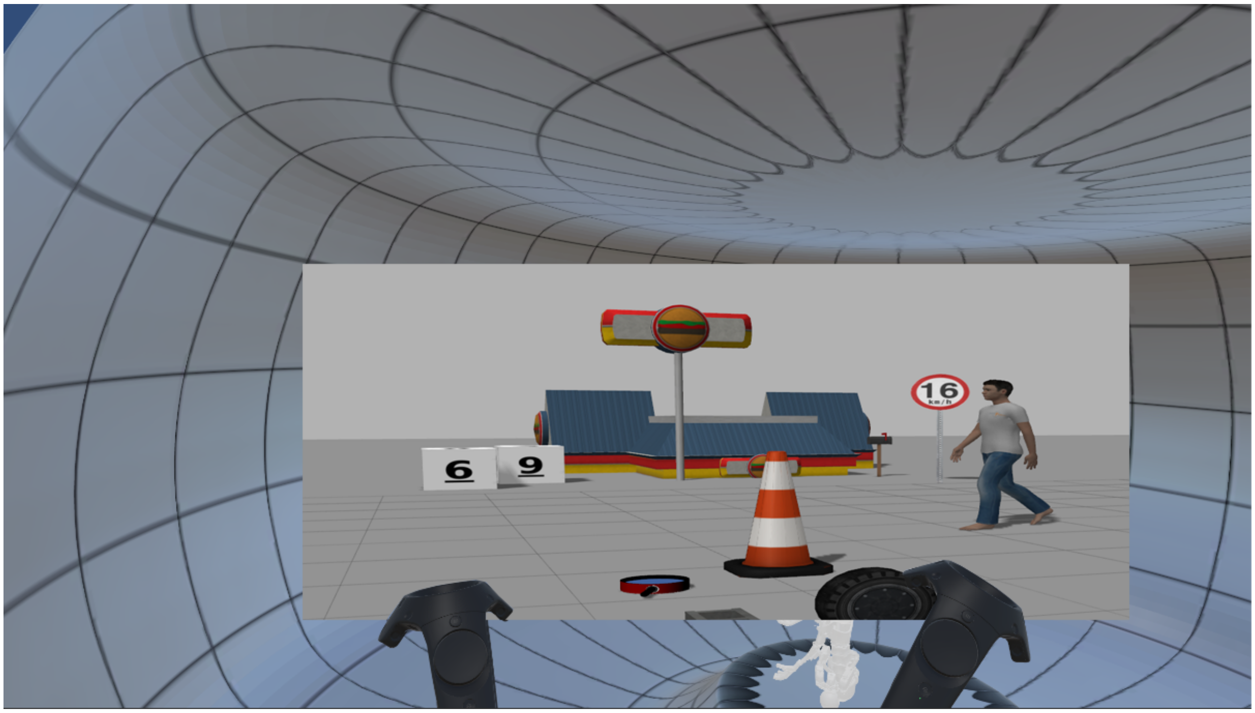


Fig. 2.5: Take control over the simulation Roboy and see what he does in gazebo.

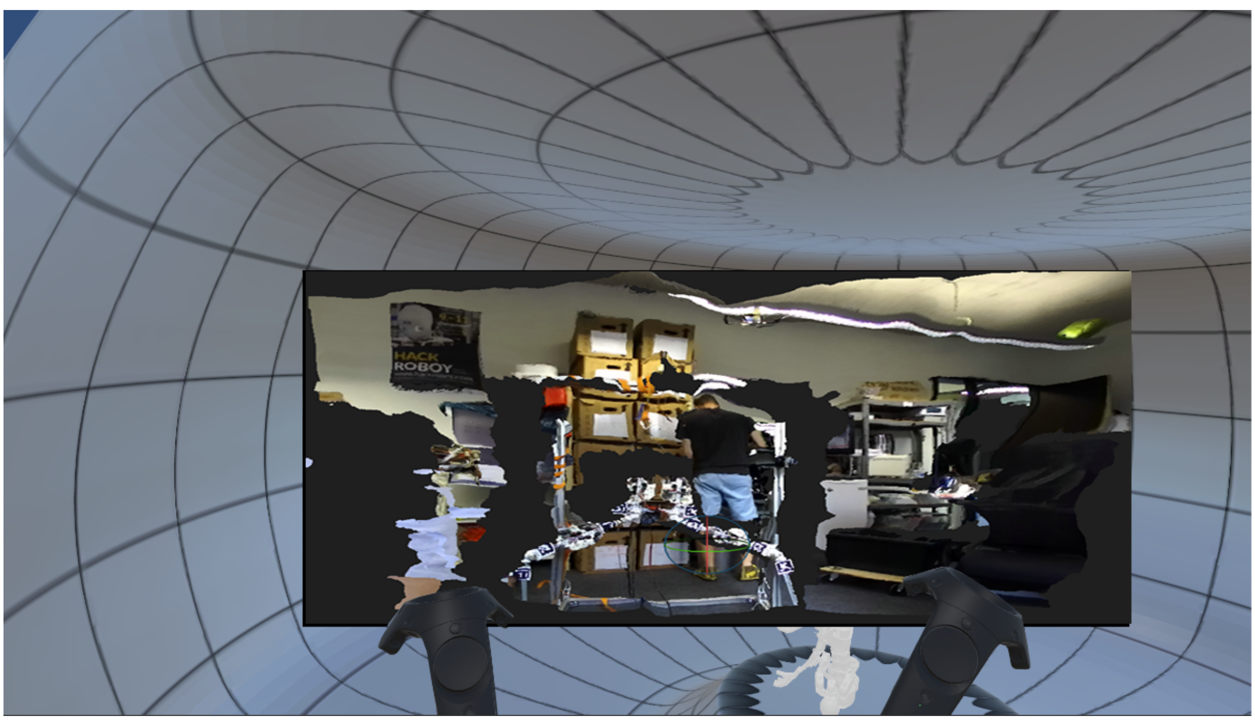


Fig. 2.6: Look through the eyes of the real Roboy and control him in real life.

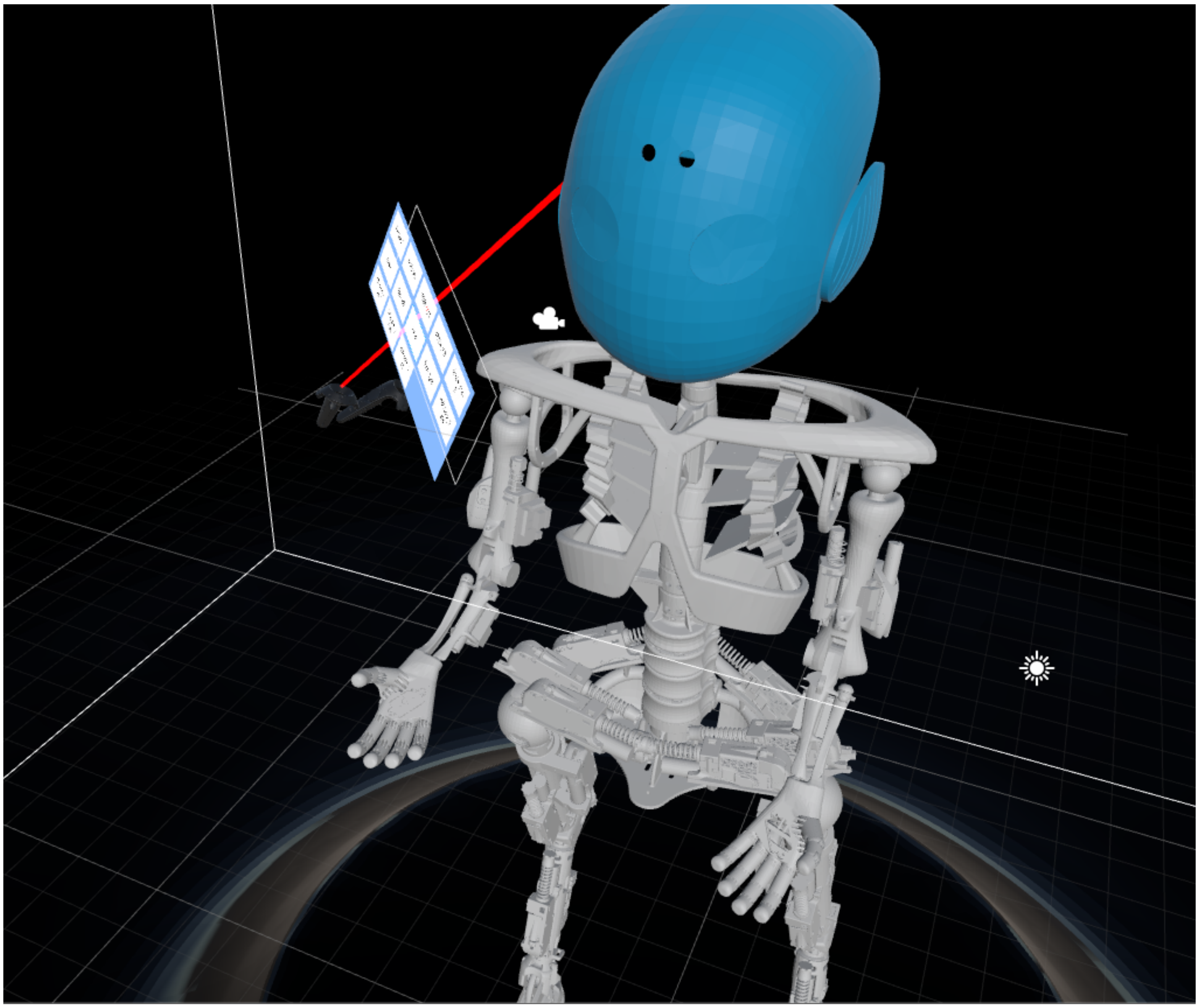


Fig. 2.7: Sit back, relax, take a look at Roboy from a safe distance and watch him do some stuff.

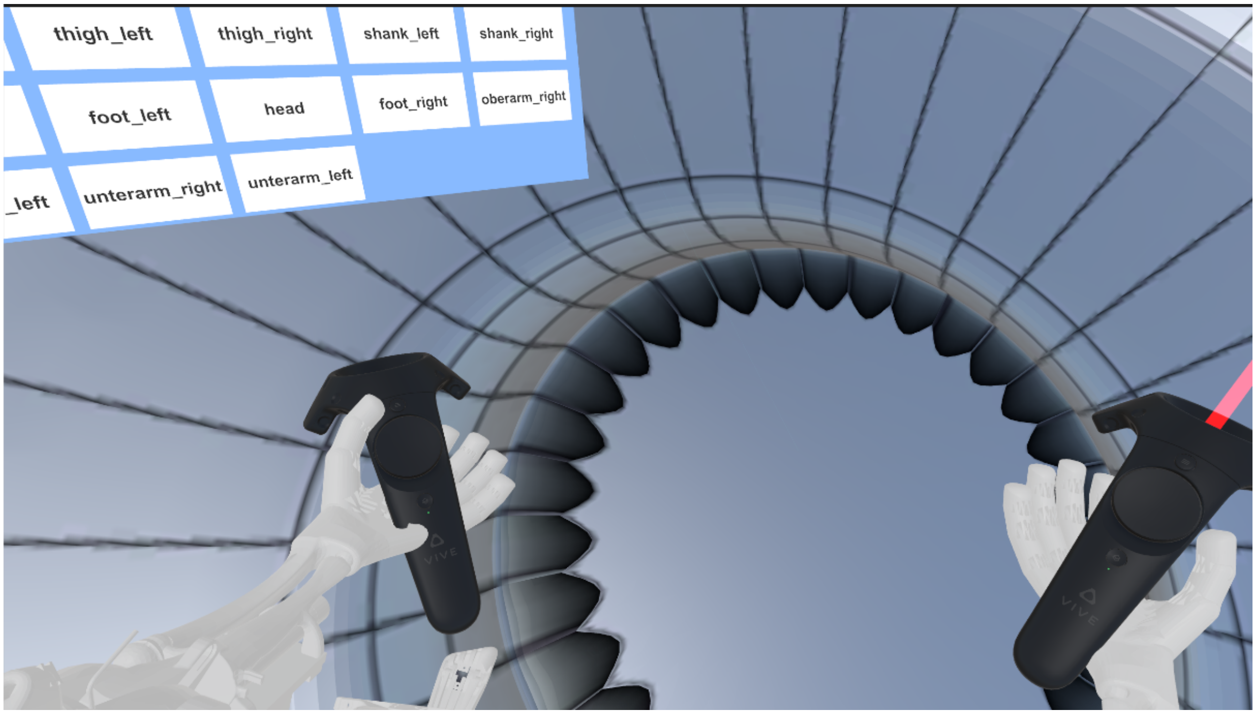


Fig. 2.8: Slip into the role of the true VR Roboy, cause mayhem or look cute, you decide.

Developer's Manual

Are you sure you want to go down that road? This will be though, are you prepared? Yeah? So follow me if you want to bring the BeRoboy™ development forth.

Note: We assume that you already have gone through the User's Manual to not repeat ourselves.

Gazebo Simulation

For the gazebo part you need to create/ edit a launch/ world file(s). When the launch file is started it automa- tically loads the world (with all the surrounding objects) that has been specified and the version of roboy you have chosen.

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"/>
        <arg name="headless" value="false"/>
                <arg name="debug" value="false"/>
        </include>
</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.

```xml
<world name="default">

<!-- A ground plane -->
<include>
    <uri>model://ground_plane</uri>
</include>
<!--PabiRoboy -->
<include>
    <uri>model://legs_with_upper_body</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>
```

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.

```xml
<sensor type="camera" name="camera">
      <update_rate>1.0</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).

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)

Use Cases

The following use cases should demonstrate how 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.

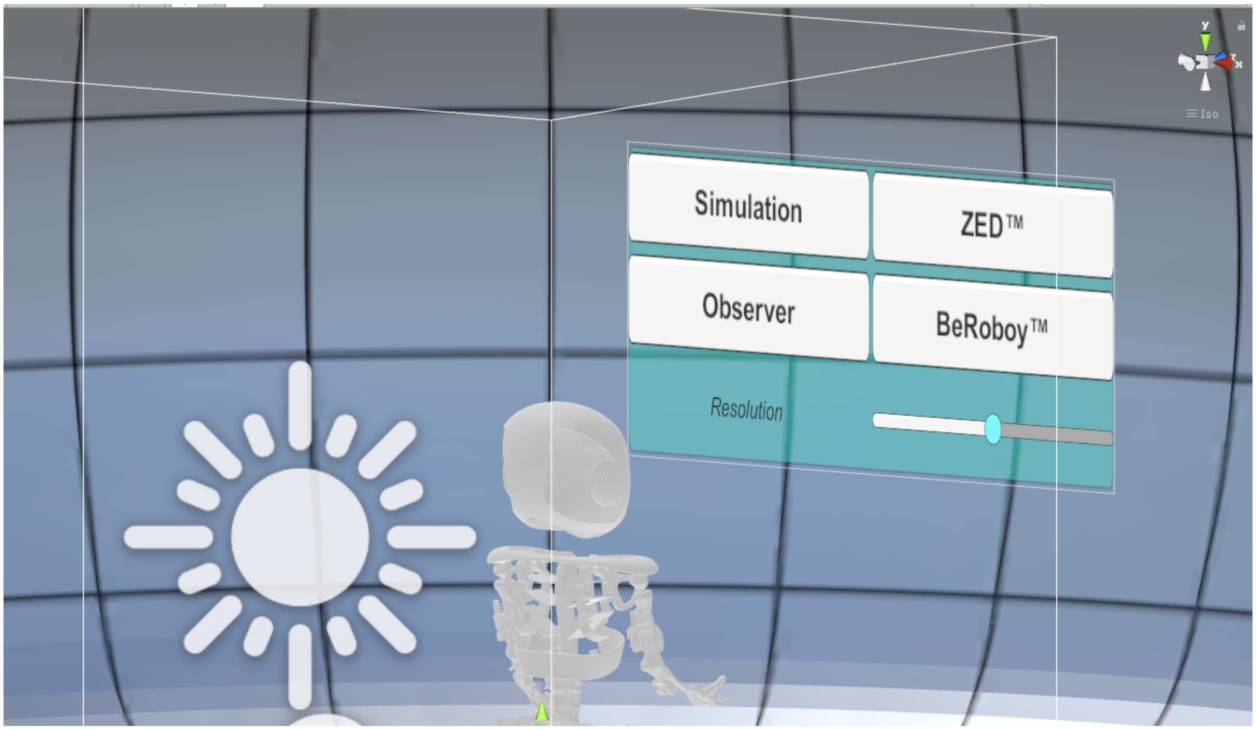


Fig. 2.9: After clicking on one of the buttons, the View Selection Manager takes the necessary steps to change to the respective view.

Switching between views

Use Case II

Sample goes here.

Introduction

What is it?

PaBi-VR shows the Roboy PaBi legs in a simulation and in a VR room simultaneously while the PaBi legs shows some furious dance moves appropriate for every music track. On top of that you can stop the *EPIC* dance and create your own dance moves with simple commands. In VR you can fully immerse yourself with the PaBi World, a world which you will not want to leave ever again. A tremendous GUI visualizes *very interesting* data.

How does it work?

The PaBi legs are loaded in Gazebo via a Plugin which makes them listen to dance commands. At the same time a ROS node starts which sends dance commands to PaBi. The whole PaBi state is send to the VR room in Unity and is processed on a GUI.

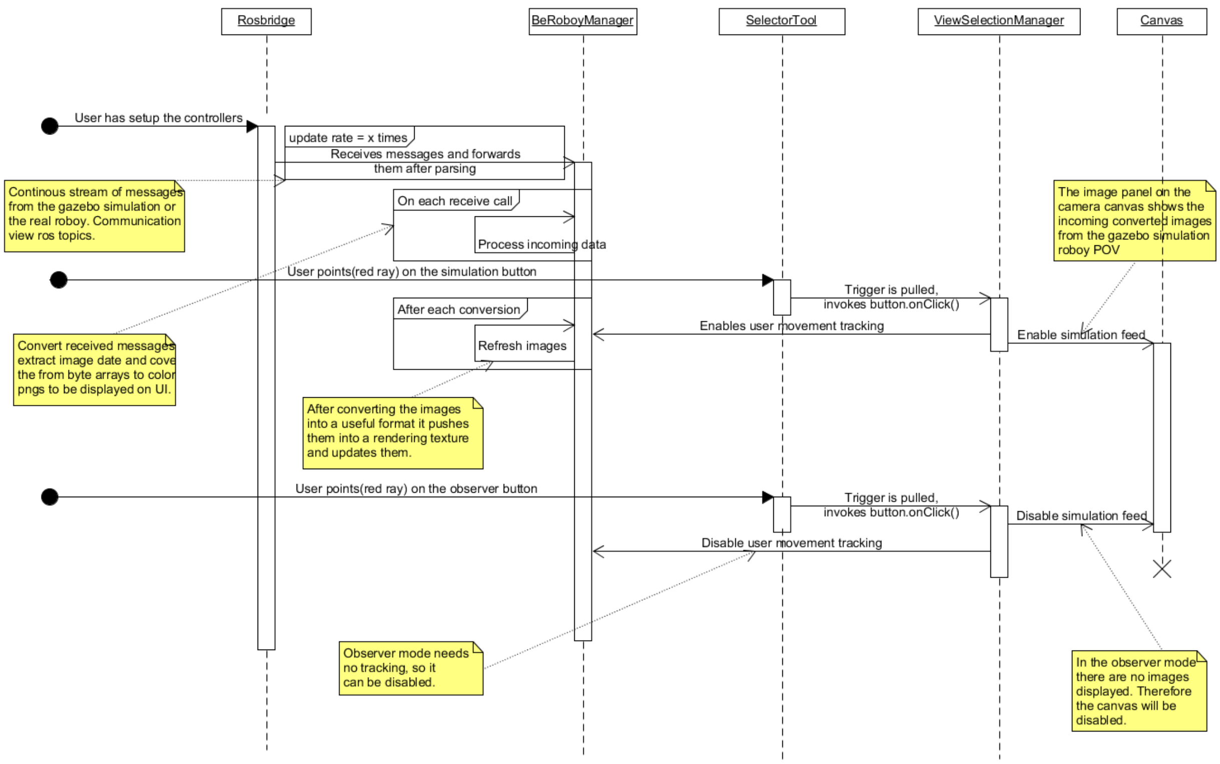


Fig. 2.10: The user first switches to the simulation view, but gets all exhausted and switches to the relaxing observer view mode.

User's Manual

Setup Ubuntu side

As you already installed gazebo and the roboy project like described in the installation part you need only to start the *.launch* file.

1. Source the setup.bash

```
source /path-to-roboy-ros-control/devel.setup.bash
```

2. Start the launch file which starts Gazebo with the PaBi legs and a PaBiDanceSimulator ROS node

```
roslaunch roboy_simulation pabi_world.launch
```

This should be the result:

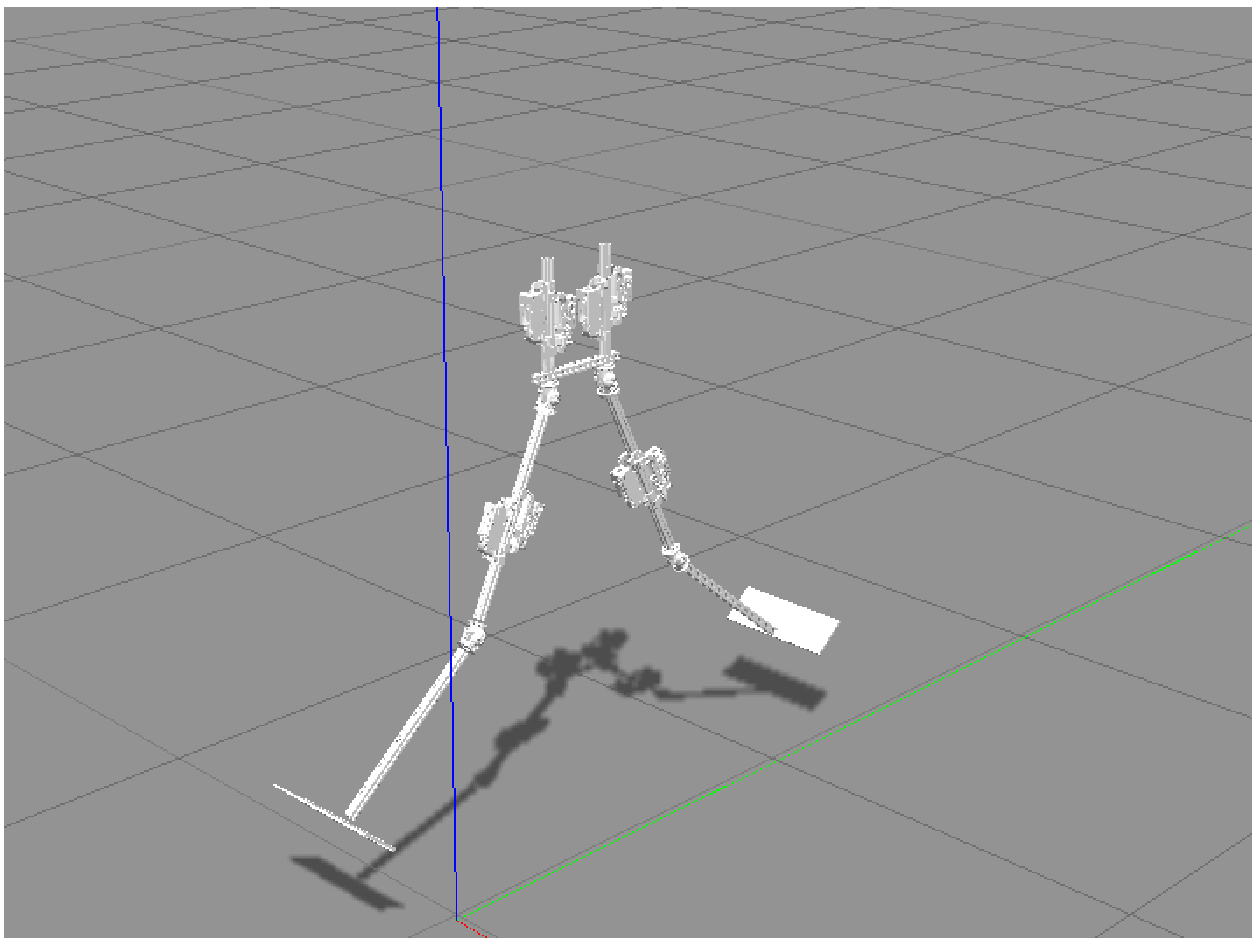


Fig. 2.11: PaBi model in Gazebo

Troubleshooting

These commands should be sufficient but it can happen that gazebo has problems loading the PaBi Model into the world or starting the gazebo server.

1. Kill the gazebo server and restart it.

```
killall gzserver
killall gzclient
```

2. Export the gazebo paths to the model

```
source /usr/share/gazebo-7/setup.sh
export GAZEBO_MODEL_PATH=/path/to/roboy-ros-control/src/roboy_models:$GAZEBO_MODEL_
↪PATH
```

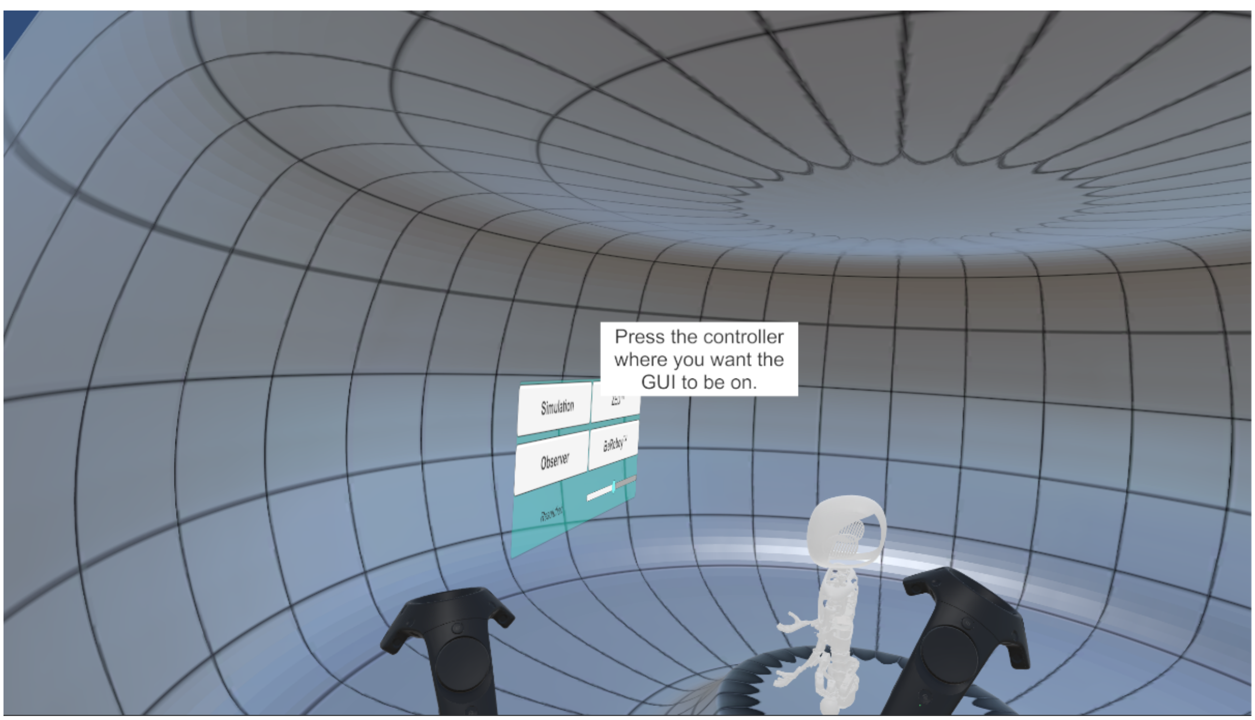
3. If nothing helps than write an email to roboyvr@gmail.com. We will glady help you to experience the RoboyVR-Experience.

Setup Unity side

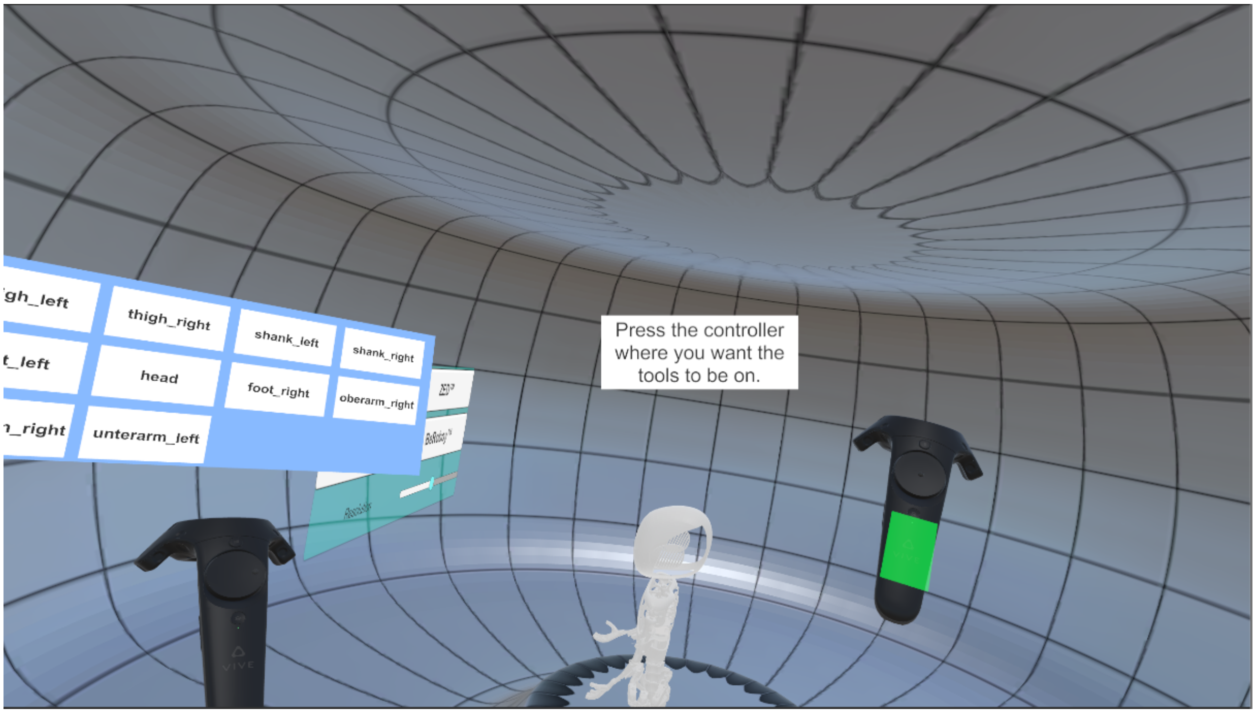
You should have the RoboyVR project already cloned on your local machine. Therefore you only need to start Unity and open the PaBiViveScene. There should be a ROSBridge object in the hierarchy. Select this object and enter the IP Adress of the machine on which the simulation is running.

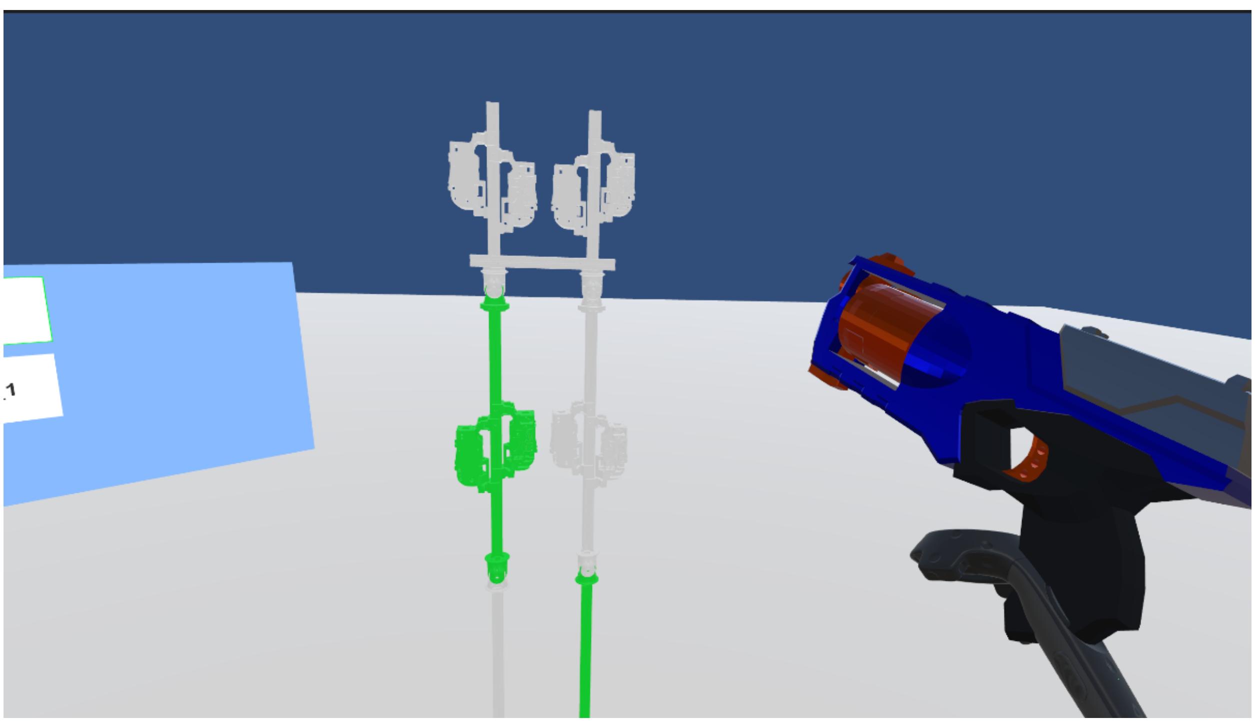
Fig. 2.12: ROSBridge in Unity

As soon as you start the scene SteamVR should open if that is not already the case. Then you have to follow the instructions on the screen to setup your Vive controllers.



Afterwards you can watch PaBi showing his best dance moves and interact with him via a GUI and different tools.

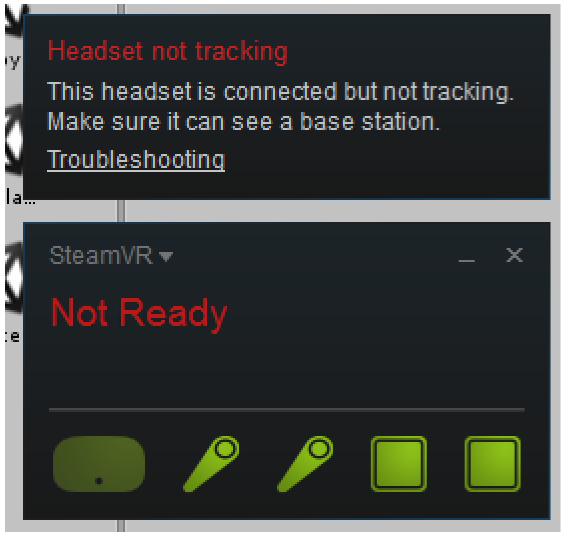

thigh_right
shank_left
shank_right
head
foot_right
oberarm_right
unterarm_left
Press the controller where you want the tools to be on.



Note: Shooting PaBi with the nerf gun does not have any consequences and serves as a alleviation of stress

Troubleshooting

If the window of SteamVR shows any errors, then simply restart it.



Developer's Manual

Note: We assume that you already have gone through the User's Manual to not repeat ourselves.

Gazebo Plugin

The main part on the simulation site is the plugin *ForceJointPlugin*. The location is:

```
path-to-roboy-ros-control/src/roboy_simulation/src/ForceJointPlugin.cpp
```

The plugin does the following:

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.

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:

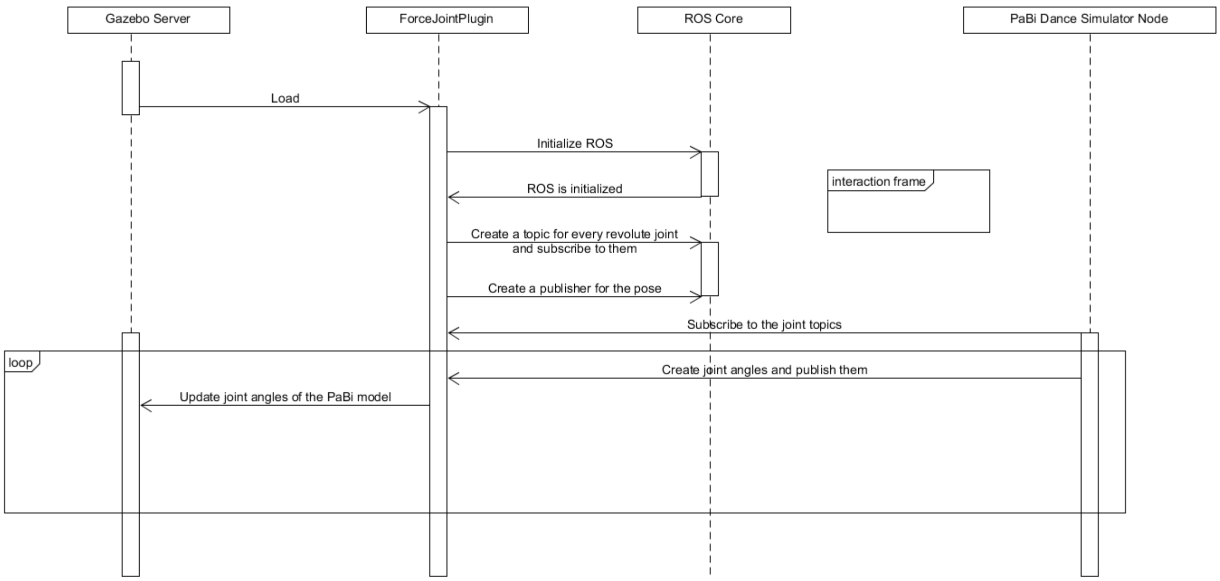


Fig. 2.13: Sequence diagramm of the ForceJointPlugin

```
/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);
```

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;
}
```

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.

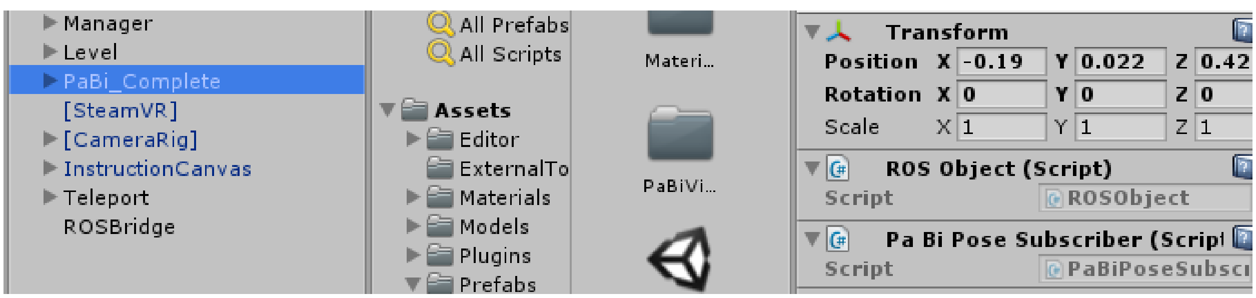


Fig. 2.14: ROSObject component

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.

Introduction

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 missleading information but provides selected information and explanations to satisfy the visitor's interest. Important aspects include structuring and grouping the given data in sets which the user can change between and activate dynamically, providing an intuitive control system which does not need much further explanation and visualizing the given data in a clear and understandable way.

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
- Core: containing logic, update methods and operations on given data set
- Back-end: Provider of data either through a connection to ROS (or method dummies sending fake data)

Current Implementation

The user interface as of now does only contain basic front-end elements such as screens, panels and camera overlay canvases, as well as parts of the basic Core area. These include basic support of the Vive Controllers, the display of all elements, both in the scene and part of the UI on the SteamVR glasses. Many UI elements are not interactive as of now but in future updates this will be changed.

User's Manual

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 connected SteamVR headset and controller, though it does not change the camera or control panels without these interaction methods, as these are the single input method. The Screen is displayed in a window in Unity and in the glasses. Since as of now there is no connection to ROS, 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.

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.

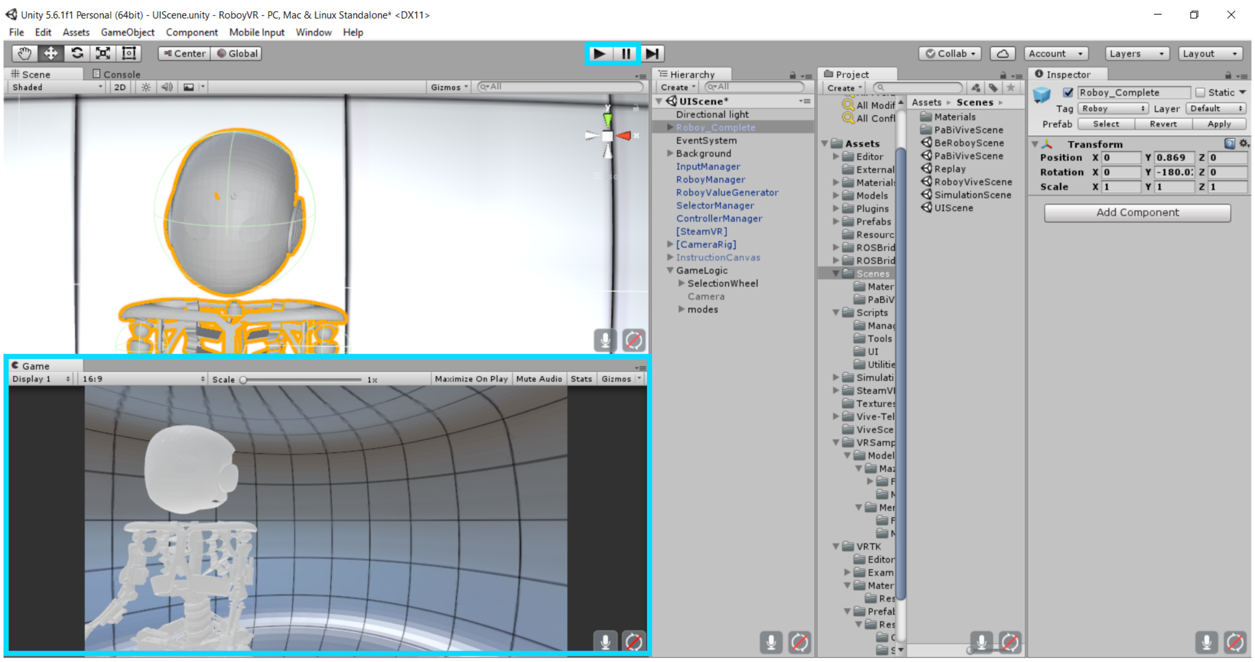


Fig. 2.15: Unity Editor

UI Features

The user can initially decide, which controller is his main controller by using the trigger at the underside of the controller. 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 surrounding.

With the raycaster - seen as a red beam - different body parts of Roboy can be selected and information can be displayed. This is the current state of implementation, further changes and updates will be made in the future.

Developer Manual

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, but a modular, easy-to-adjust base which could easily be integrated in other scenes.

Scripts:

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

Use Case

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

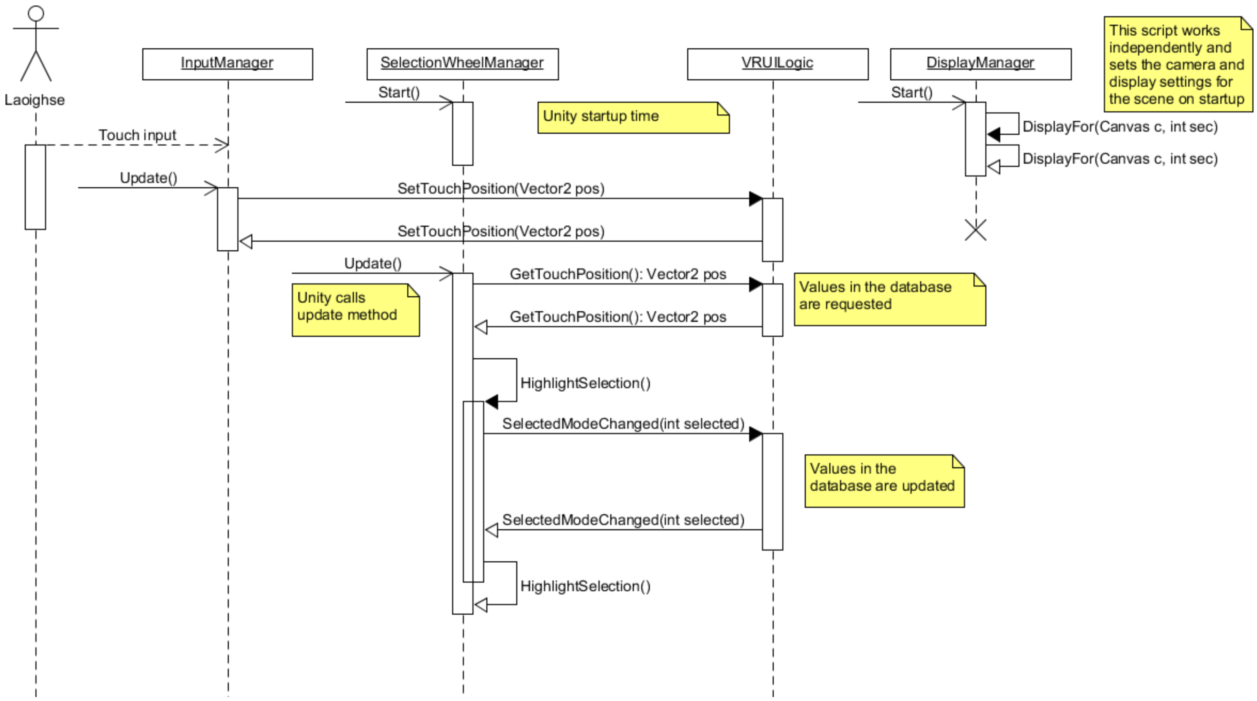


Fig. 2.16: Sequence of method calls after user touch input initialized by Unity's update function

UI Implementation and Structure

The UI can be structured in three basic layers:

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: DisplayScreens: 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.

Game Ojects: modes: this empty game object contains all modes the user can choose with the selection wheel. These are dis- and enabled dynamically by the script UILogic.

Canvas: 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.

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.

Fig. 2.17: Display message

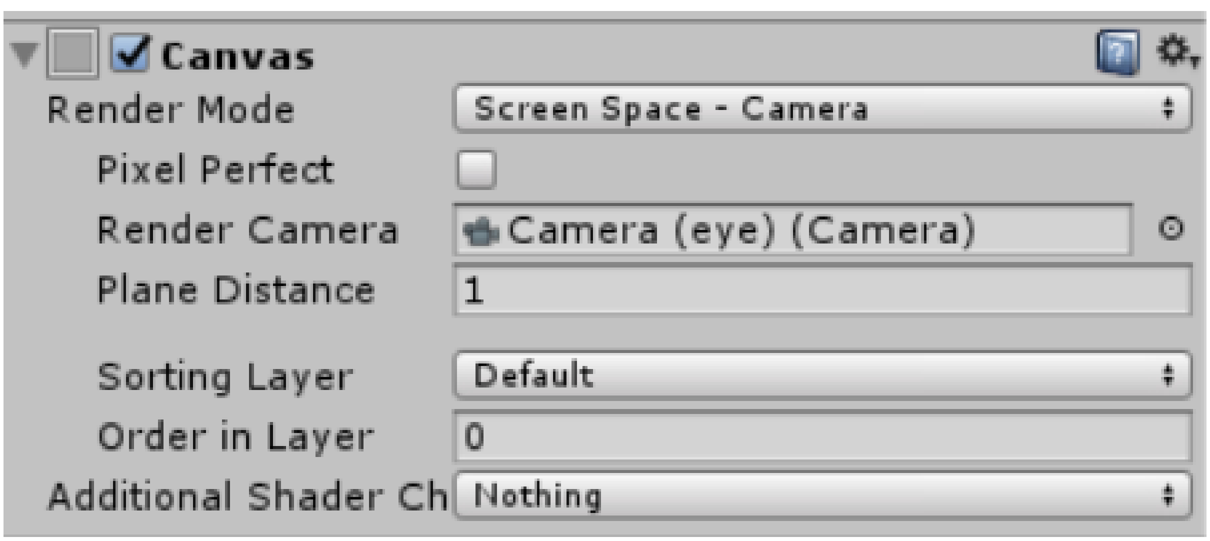


Fig. 2.18: Canvas settings for VR headsets

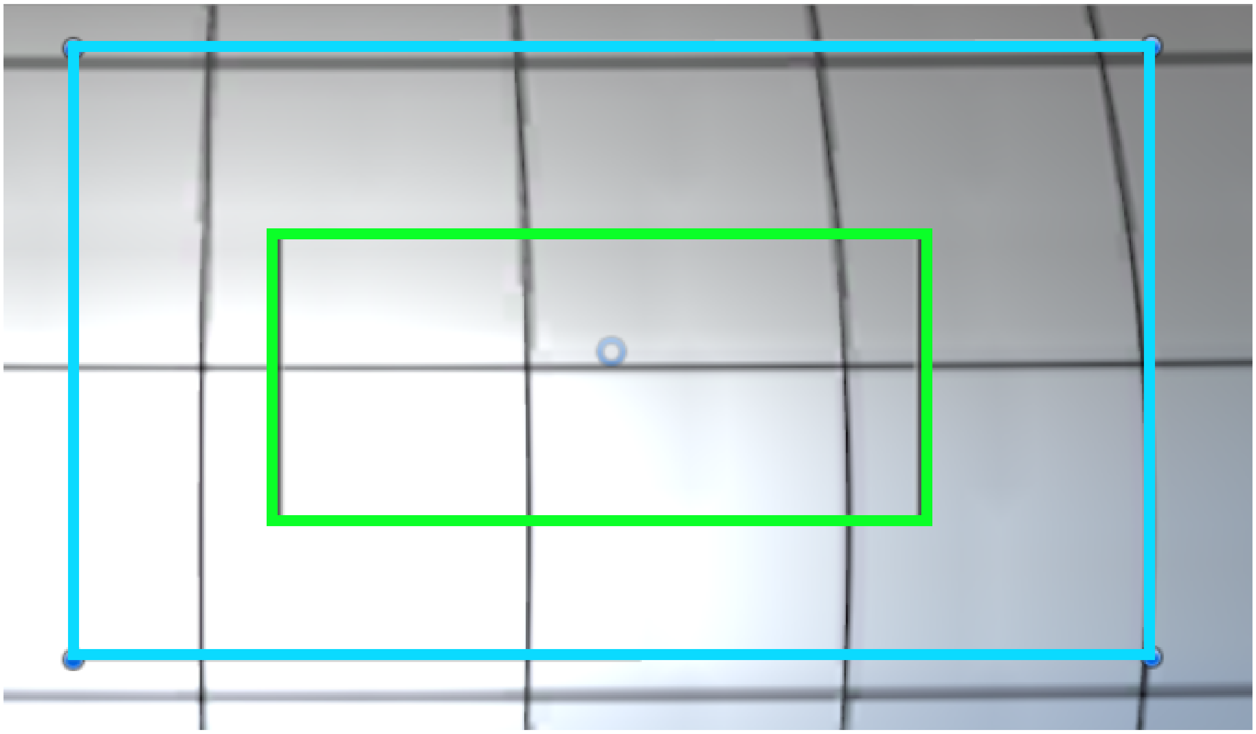


Fig. 2.19: Canvas size in blue and view frustrum in green

Core

This layer covers the UI logic. It displays the selected modes, updates the frontend based on the processed given input, performs user requests and handles user input such as pointing, clicking and scrolling. Based on demand, it creates new UI elements, alters, updates, activates and hides these.

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.

UILogic: This script operates as a database for important game values. Due to its Singleton (link: wikipedia) implementation, it is always accessible and no dublicates and therefore (versionierung /) and all functions can use it as a data platform. 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, since it assures modularity of the respective elements, both front and back-end. It provides less assurance considering the age of the given data when later used. Nevertheless, an Event-driven UI can still be implemented using the subscriber scheme. The issue there would be if the data is provided in non-continuous time intervals. One operation which is implemented using that style is the change of modes. As soon as the selected index changed, the respective game object and mode, which is linked in the public modes array, is activated, the others are deactivated.

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

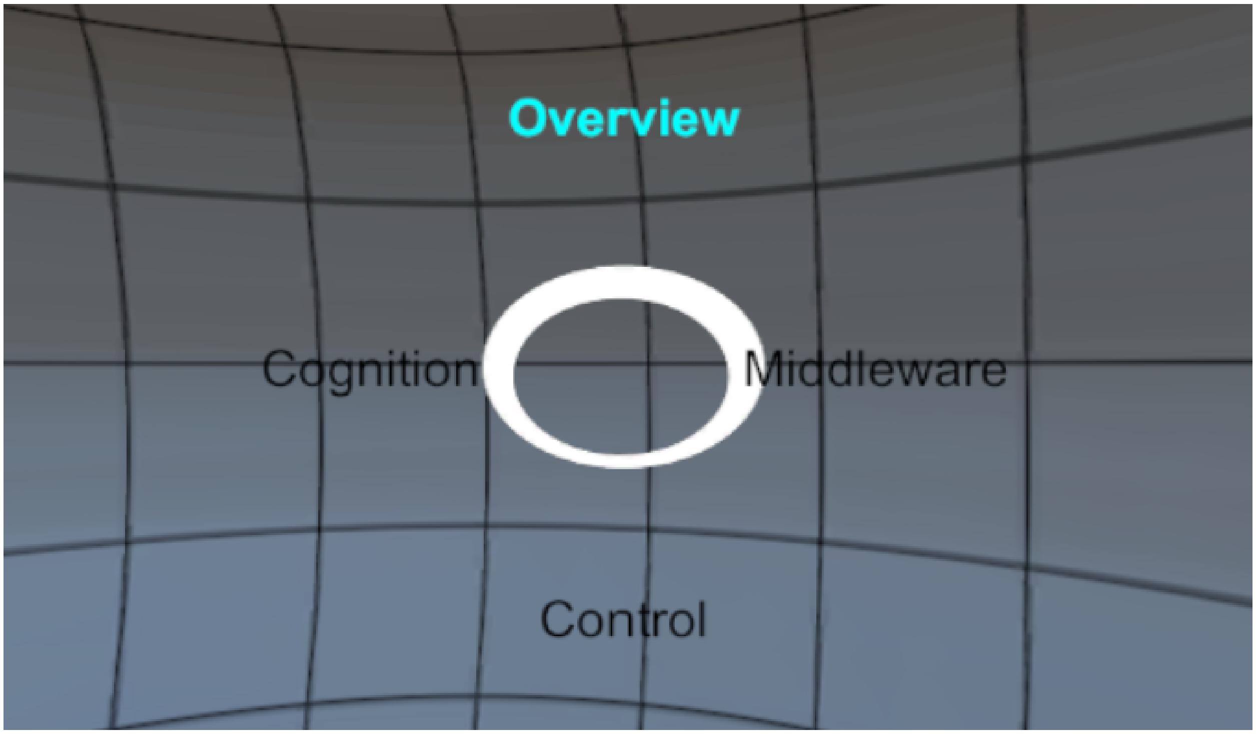


Fig. 2.20: Selection wheel with four options and Overview selected

objects.

Back-End

As of now, there is no implementation of a back-end.

Context

The core of RoboyVR renders and updates roboy's pose as its receiving data from the simulation via ROS-messages. Additional information inside messages like current powerconsumption or motorforce is displayed on an interactive GUI. Apart from that the user can actively manipulate the simulation through various tools. On top of that the system can check for the latest roboymodel with the help of github and update it if necessary.

Conventions

We follow the coding guidelines:

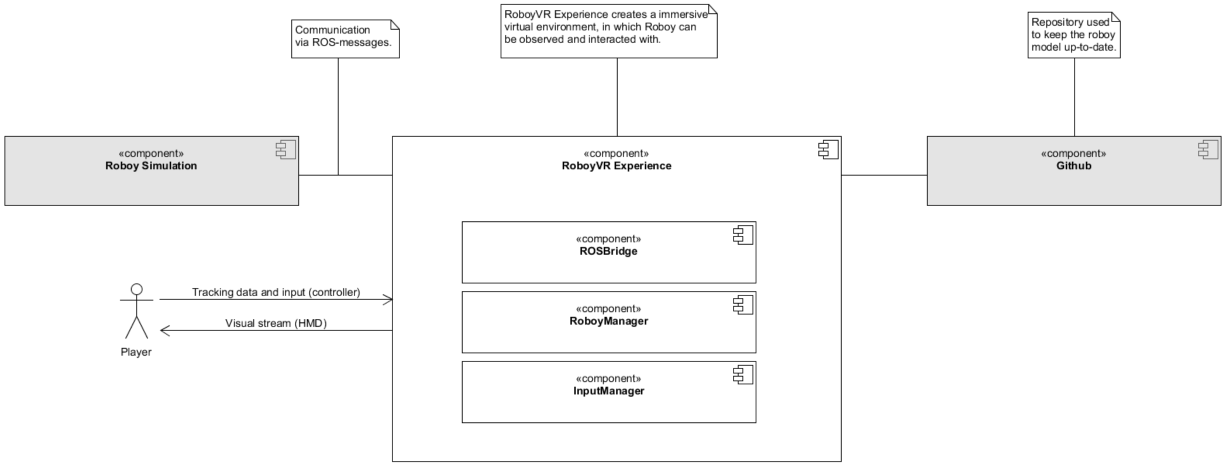


Fig. 2.21: RoboyVR Experience has two neighboring systems. Roboy simulation to receive pose data and Github for model updates.

Table 2.1: 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:

1. All scripts are structured like this:
   1. The script is ordered in regions:
      - PUBLIC_MEMBER_VARIABLES
      - PRIVATE_MEMBER_VARIABLES
      - UNTIY_MONOBEHAVIOUR_METHODS
      - PUBLIC_METHODS
      - PRIVATE_METHODS
   2. In PUBLIC_MEMBER_VARIABLES you have define at first your properties and then public variables.
   3. In PRIVATE_MEMBER_VARIABLES you have define at first your serialized private variables and then the normal ones.
   4. In UNTIY_MONOBEHAVIOUR_METHODS the order is as follows: Awake, Start, OnEnable, OnDisable, Update
2. All variables and functions where it is not instantly clear what it does, have to be commented with a summary.
3. Make variables only public if they need to be. Mark variables as Serializable when you need to edit them in the editor.
4. The capitalization follows a specific set of rules:

- public variables and properties start with an uppercase
- private variables and properties start with a lowercase
- public functions start with an uppercase
- private functions start with an lowercase

5. Coroutines which are accessed in other classes must have a public interface.
6. When you store components in a variable, which are directly on the object itself, put a [RequireComponent(typeof(ComponentType))] on top of the class.

We include a template class with all rules implemented.

Warning: doxygenclass: Cannot find class "TemplateClass" in doxygen xml output for project "roboyVR" from directory: doxyxml/

Architecture Constraints

Table 2.2: Hardware Constraints

Constraint Name	Description
HTC Vive	We need user position tracking and movement tracking.

Table 2.3: 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 2.4: 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.

Table 2.5: 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 2.6: 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.

User Interfaces

In the following figures you can see multiple tools to interact with roboy. The user can select different parts of roboy and inspect these parts further with detailed information about the motors. On top of that the user can actively interact with roboy with the Shooting Tool. It triggers an external force in the simulation and displays the result in real time in the VR environment. In the future it will be possible to control time, so to rewind the simulation and save/ load them on runtime.

At first you choose which tools go where.

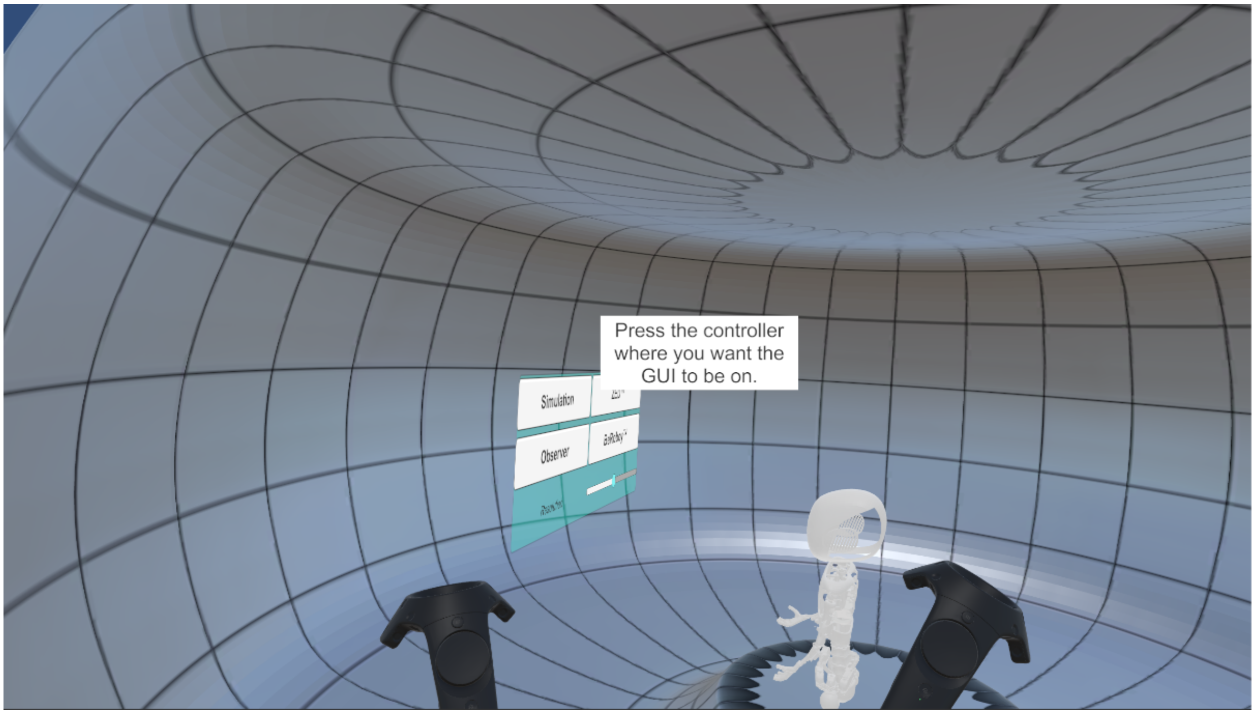


Fig. 2.22: Press any of the controller to set it respectively.

Too tired to walk? Just teleport!

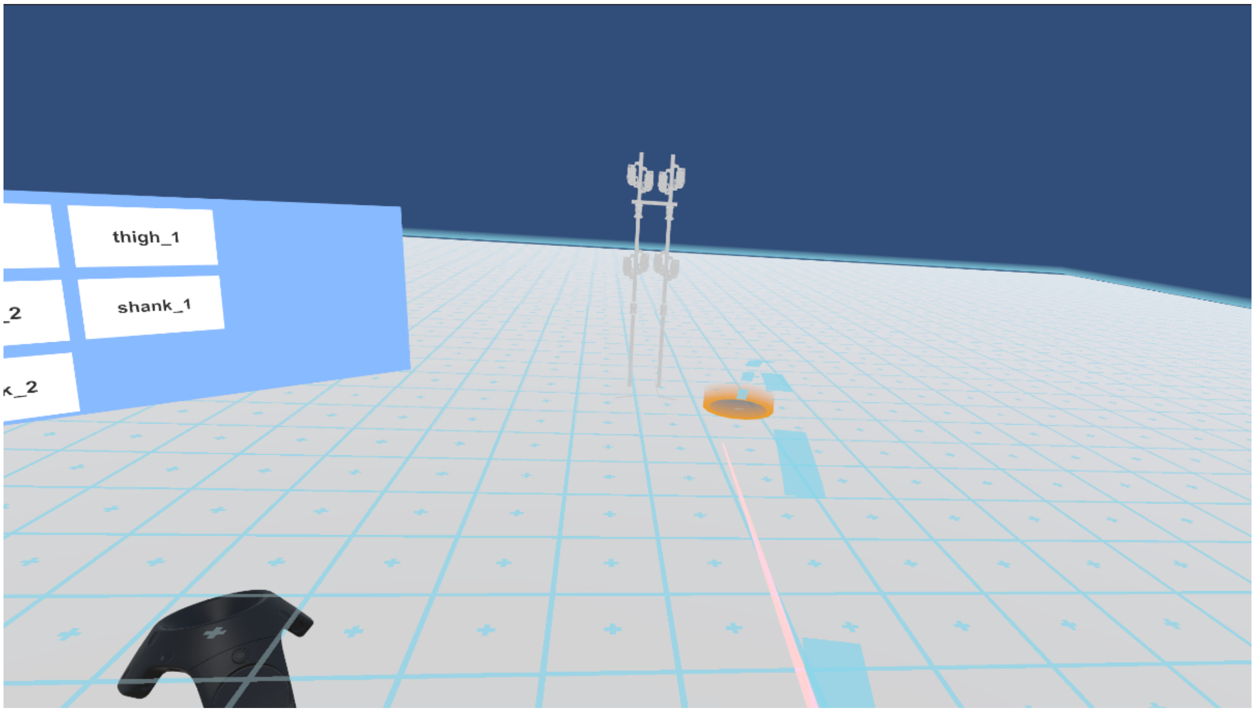


Fig. 2.23: Press down on the touchpad to teleport to a specific position.

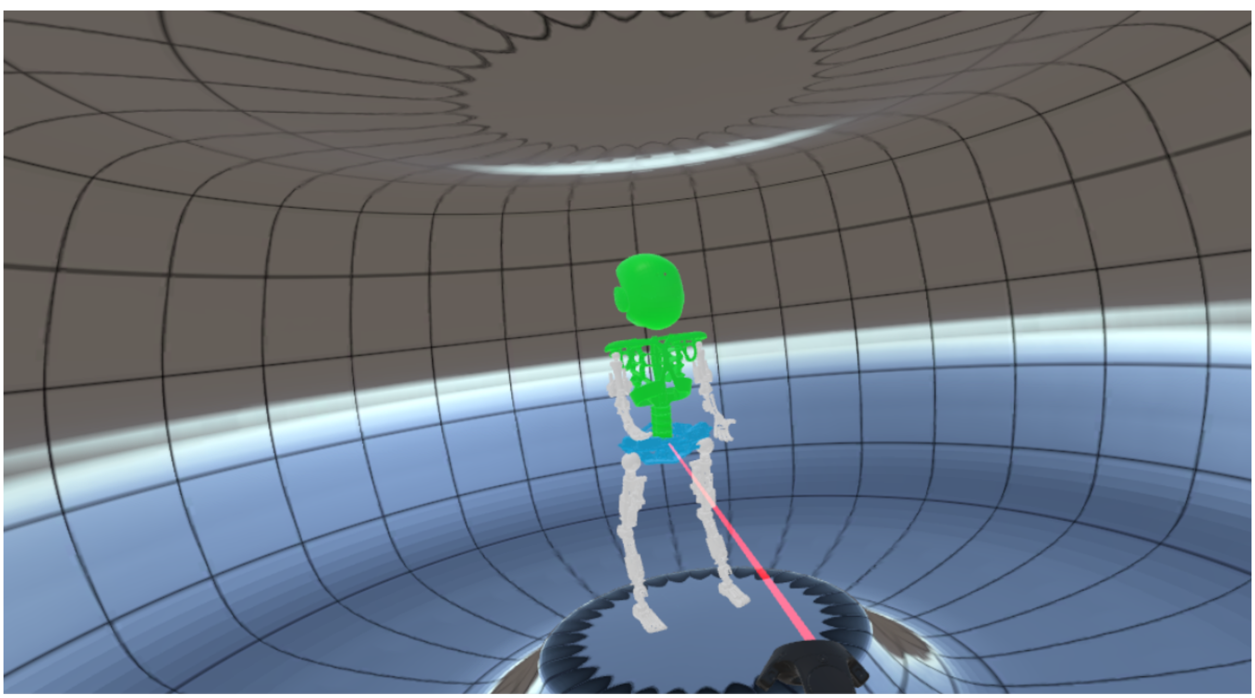


Fig. 2.24: Tool for selecting roboy parts.

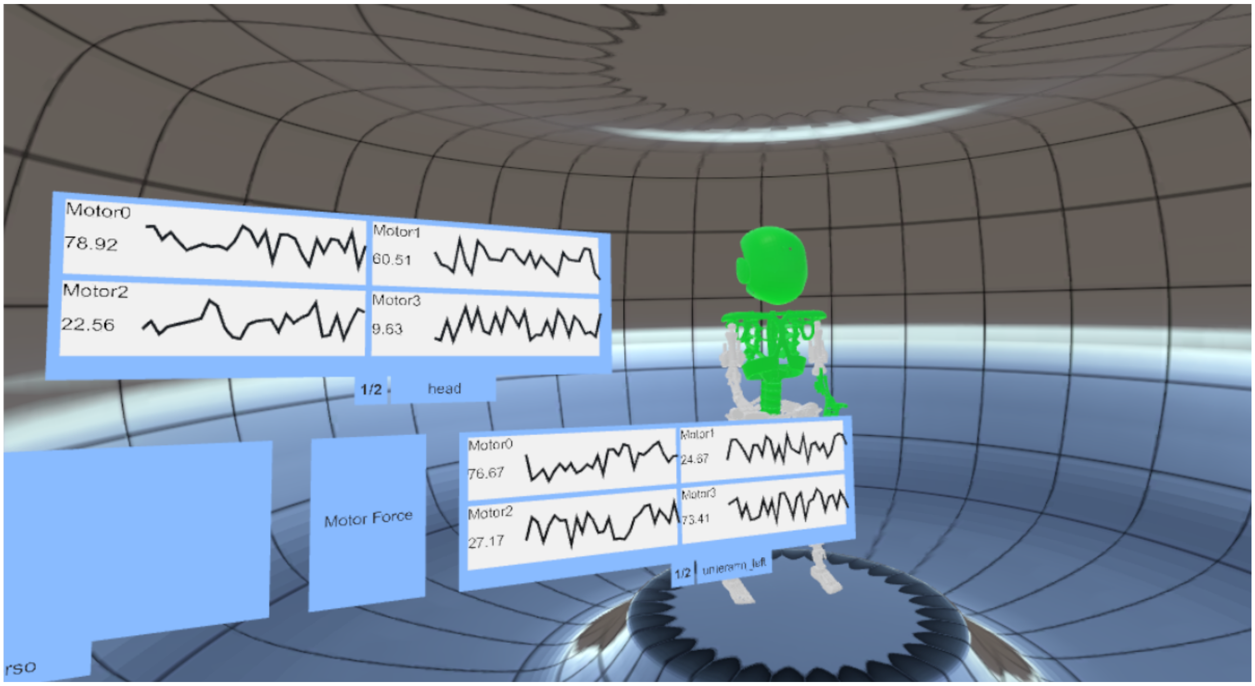


Fig. 2.25: UI Panels displaying motor force of several roboy parts.

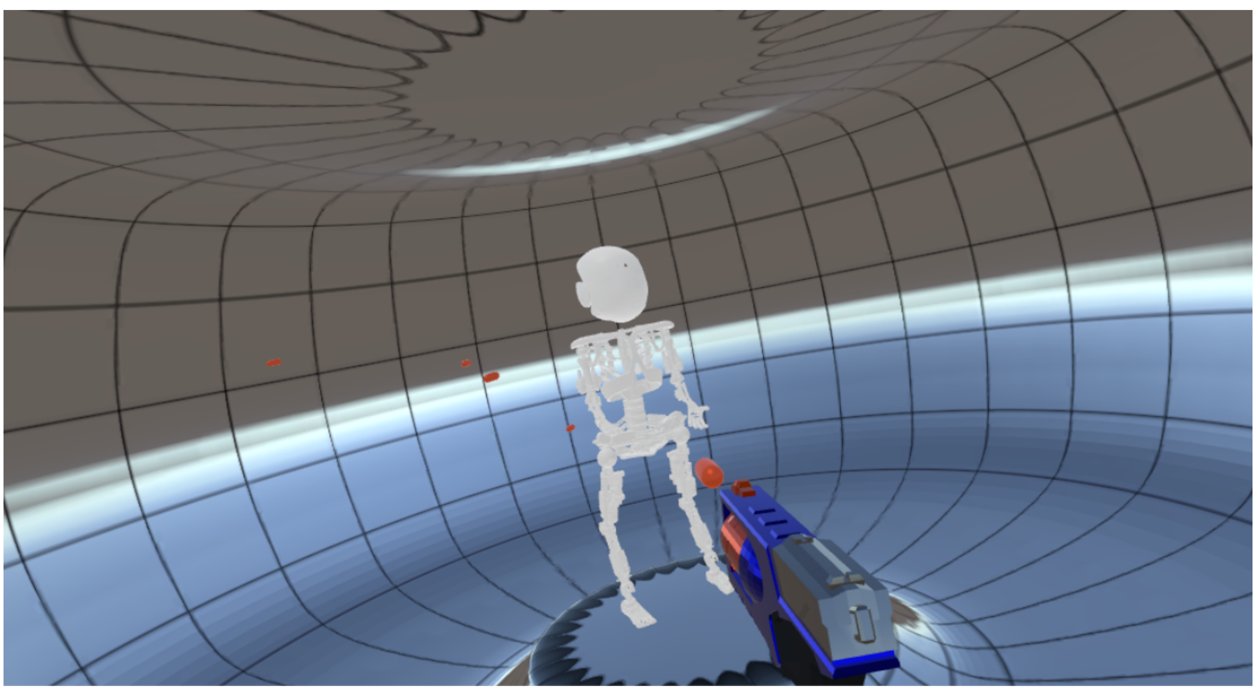


Fig. 2.26: Tool to shoot roboy and trigger an external force.

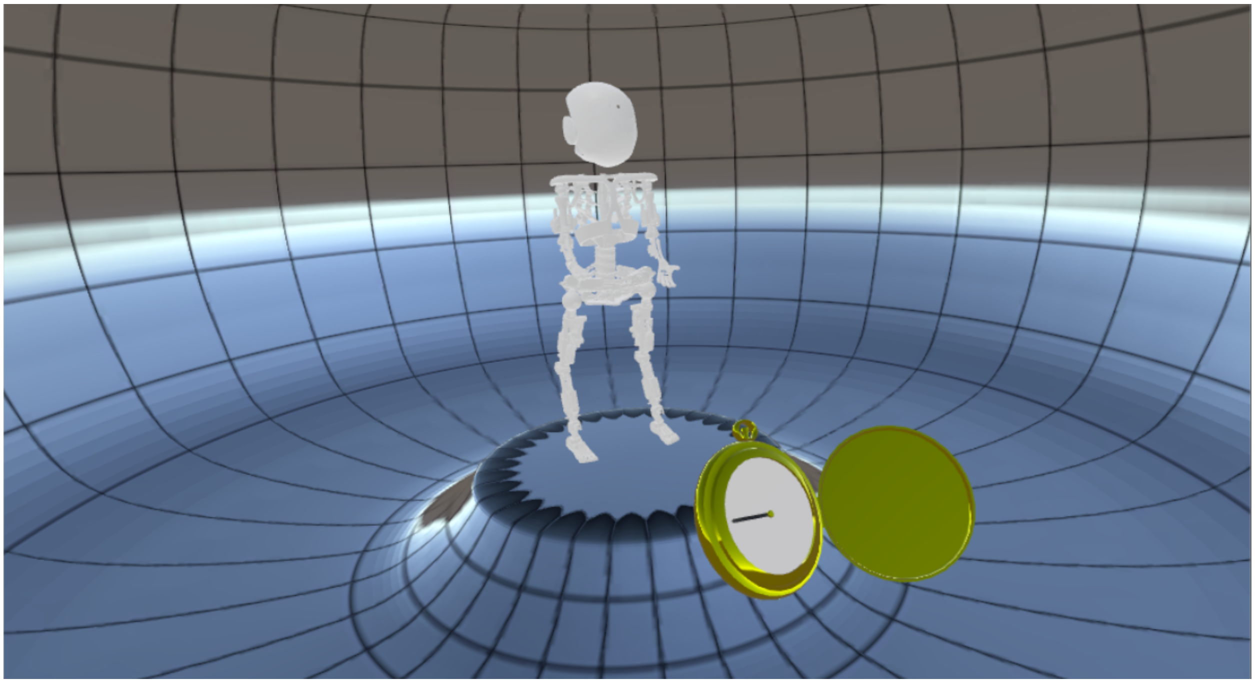


Fig. 2.27: Tool to alter flow of time.

Public Interfaces

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, in

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.AddSubscriber(Type subscriber)

Add a subscriber callback to this connection.

There can be many subscribers.

void ROSBridgeLib.ROSBridgeWebSocketConnection.AddPublisher(Type publisher)

Add a publisher to this connection.

There can be many publishers.

void ROSBridgeLib.ROSBridgeWebSocketConnection.Connect()

Connect to the remote ros environment.

void ROSBridgeLib.ROSBridgeWebSocketConnection.Disconnect()

Disconnect from the remote ros environment.

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.DurationMsg, ROSBridgeLib.custom_msgs.ExternalForceMsg, ROSBridgeLib.custom_msgs.ForceMsg, ROSBridgeLib.custom_msgs.LinkMsg, ROSBridgeLib.custom_msgs.PositionCustomMsg, ROSBridgeLib.custom_msgs.RoboyPoseMsg, 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, ROS-

BridgeLib.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, ROSBridgeLib.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_velo`

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

ROSBridgeActors

class `ROSBridgeLib::`**`ROSBridgePublisher`**

This defines a publisher.

There had better be a corresponding subscriber somewhere. This is really just a holder for the message topic and message type.

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

Inherits from MonoBehaviour

Subclassed by ROSBridgeLib.RoboyForcePublisher

class `ROSBridgeLib::`**`ROSBridgeSubscriber`**

This defines a subscriber.

Subscribers listen to publishers in ROS. Now if we could have inheritance on static classes then we could do this differently. But basically, you have to make up one of these for every subscriber you need.

Subscribers require a ROSBridgePacket to subscribe to (its type). They need the name of the message, and they need something to draw it.

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

Inherits from MonoBehaviour

Subclassed by ROSBridgeLib.PaBiPoseSubscriber, ROSBridgeLib.RoboyCameraSimSubscriber, ROSBridgeLib.RoboyCameraZedSubscriber, ROSBridgeLib.RoboyPoseSubscriber

class `ROSBridgeLib::`**`ROSBridgeService`**

This defines a ROS service.

Basically a service serves as function call. Therefore you need the service aka the function and arguments when you call a service. As soon as you send a service call the service waits for a response.

Inherits from MonoBehaviour

Subclassed by ROSBridgeLib.RoboyServiceResponse

ROSBridgeLibExtension

We extended the library in the form that we implemented a singleton class to handle all ROSActors in the scene.

class **`ROSBridge`**

Handles the *ROSBridge* connection.

Adds all ROS components of each *ROSObject* in the scene. You need one object of this in each scene where you have ROS actors.

Inherits from Singleton< ROSBridge >

Public Members

string ROSBridge.ROSCoreIP = ""
The IP address of the roscore running on the other side of the *ROSBridge*.

int ROSBridge.Port = 9090
Port of the *ROSBridge*.

Property

property ROSBridge::ROS
Public property for other classes to the ros websocket.

property ROSBridge::ROSObjects
Public property of all active ROSObjects in the scene.

Private Functions

void ROSBridge.Awake()
Initializes the ROS websocket connection and searches for all ROSObjects in the scene.

void ROSBridge.Update()
Run *ROSBridge*.

void ROSBridge.OnApplicationQuit()
Disconnect from the simulation when Unity is not running.

Private Members

ROSBridgeWebSocketConnection ROSBridge.m_ROS = null
ROS websocket connection.

bool ROSBridge.m_ROSInitialized = false
Is ROS initialized?

List<GameObject> ROSBridge.m_ROSObjects = new List<GameObject>()
List of all active ROSObjects.

class ROSObject
Empty class to mark this object as an *ROSObject* so that *ROSBridge* finds this object and adds all ROS components attached to this object.

Inherits from MonoBehaviour

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.
-# subscribe to the external force event and forward the message to the␣
↪simulation.
-# 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 and the motors values (future).

Parameters

- `msg`: JSON msg containing roboy pose.

`void RoboyManager.ReceiveExternalForce(RoboyPart roboyPart, Vector3 position, Vector3`
: Sends a message to the simulation to apply an external force at a certain position.

Parameters

- `roboyPart`: The roboypart where the force should be applied.
- `position`: The relative position of the force to the roboypart.
- `force`: The direction and the amount of force relative to roboypart.
- `duration`: The duration for which the force should be applied.

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*

`void RoboyManager.drawTendons()`
: Test function to draw tendons.

For now draws only random lines. TEMPLATE!

void RoboyManager.adjustPose(RoboyPoseMsg msg)
Adjusts roboy pose for all parts with the values from the simulation.

Parameters

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

void RoboyManager.getRoboy()
Searches for roboy via the "Roboy" tag.

void RoboyManager.getRoboyParts()
Searches for roboy and all roboy parts.

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

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

Parameters

- `sender`
- `e`

void InputManager.ToolControllerSideButtons(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`

Property

property InputManager::GUI_Controller
Public *GUIController* reference.

property InputManager::Selector_Tool
Public *SelectorTool* reference.

property InputManager::ShootingTool
Public *ShootingTool* reference.

property InputManager::TimeTool
Public TimeTool reference.

property InputManager::SelectorTool_TouchpadStatus
Touchpad status of the controller where selector tool is attached to.

property InputManager::GUIController_TouchpadStatus
Touchpad status of the controller where gui controller tool is attached to.

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.

GUIController InputManager.m_GUIController
Private *GUIController* reference.

Is serialized so it can be dragged in the editor.

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 GUIMode
Enum for current GUI mode.

Values:

Selection

GUIPanels

enum ToolMode
SelectorTool: Select roboy meshes.

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

Values:

SelectorTool

ShooterTool

TimeTool

Public Functions

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

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

void ModeManager.ChangeToolMode()
Switches between all tools.

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::CurrentGUIMode
Current GUI mode, READ ONLY.

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

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.

GUIMode ModeManager.m_CurrentGUIMode = GUIMode.Selection
Private variable for current GUI mode.

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

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. FUTURE: 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.

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_CurrentAngle` = 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.

ViewSelectionManager

class **ViewSelectionManager**
ViewSelectionManager handles the transition between various view scenarios.

Inherits from MonoBehaviour

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.

Tools

ControllerTool

class **ControllerTool**
ControllerTool is a base class for all tools which are attached to a controller.

It provides access to steamVR functions to track the input of the controllers. On top of that it provides a function to vibrate the controller for a defined time.

Inherits from MonoBehaviour

Subclassed by *GUIController*, *SelectorTool*, *ShootingTool*, TimeTool

Public Functions

`void ControllerTool.Vibrate()`
Starts a coroutine to vibrate the controller for a fixed time.

`void ControllerTool.Initialize()`
Initiliazes the controller in a coroutine.

Intermediate function for outside classes.

Property

property `ControllerTool::Controller`
Returns the controller identity for verification purposes for outside classes.

property `ControllerTool::ControllerEventListener`
Returns a component which listens to controller events like OnTouchpad.

Private Functions

`void ControllerTool.Awake()`
Calls initialize for all controller members.

`IEnumerator ControllerTool.vibrateController()`
Coroutine to vibrate the controller for a fixed time.

Return

`IEnumerator ControllerTool.initializeCoroutine()`
Coroutine to initialize all controller members.

Return

SelectorTool

class `SelectorTool`
SelectorTool provides a functionality to select parts of roboy on the mesh itself or through the GUI.

Inherits from *ControllerTool*

Public Functions

`void SelectorTool.GetRayFromController()`
Starts a ray from the controller.

If the ray hits a roboy part, it changes its selection status. Otherwise it resets the last selected/targeted roboy part.

Private Functions

void SelectorTool.Start()
Initializes the lineRenderer component.

Private Members

LineRenderer SelectorTool.m_LineRenderer
LineRenderer to draw the laser for selection.

SelectableObject SelectorTool.m_LastSelectedObject
Variable to track the last selected object for comparison.

float SelectorTool.m_RayDistance = 3f
Maximum ray length for selection.

ShootingTool

class ShootingTool
ShootingTool is used to shoot a projectile on roboy.

The projectile then triggers a ROS message to send an external force to the simulation.

Inherits from *ControllerTool*

Public Members

Projectile ShootingTool.ProjectilePrefab
Projectile prefab which is responsible to send the ROS message.

Transform ShootingTool.SpawnPoint
Spawn transform to retrieve the spawn position and direction.

Transform ShootingTool.Trigger
Trigger transform for trigger animation.

Transform ShootingTool.TriggerBack
Transform of the position when trigger is fully pressed.

float ShootingTool.ShootDelay = 0.5f
Reload time between shots.

Private Functions

void ShootingTool.Start()
Initializes trigger position.

void ShootingTool.Update()
Shoots when the user presses the trigger to maximum value if shooting is not on cooldown.

void ShootingTool.Shoot()
Instantiates a projectile prefab on the SpawnPoint.

void ShootingTool.animateTrigger()
Animates trigger based on current trigger value.

Private Members

Vector3 ShootingTool.m_InitTriggerPosition
The standard trigger position.

float ShootingTool.m_CurrentShootCooldown = 0f
Variable for tracking current shooting cooldown.

GUIController

class GUIController
GUIController is attached on another controller as the Tools like *ShootingTool* or *SelectorTool*.

It is mainly responsible for animating so the following tasks refer always to animation:

- manage the switch between selection mode and panel mode
- manage switch between different panel modes
- manage page switch inside a panel mode
- **NOTICE: Right now *GUIController* is not inheriting from *ControllerTool* as we implemented this script at the beginning of the project. This will be changed soon, so be aware that this documentation could be out of date!**

Inherits from *ControllerTool*

Public Types

enum UIPanelAlignment
Enum for possible panel alignments.

Values:

Left

Top

Right

Public Functions

void GUIController.CheckTouchPad(InputManager.TouchpadStatus touchpadStatus)
Checks the touchpad input of the controller and acts accordingly:

1. Left: changes to previous panel if in panel mode
2. Right: changes to next panel if in panel mode
3. Top: changes between GUI modes
4. Bottom: changes the page of the current panel if in panel mode

Parameters

- `touchpadStatus`

Public Members

UIPanelRoboyPart GUIController.UIPanelRoboyPartPrefab
Prefab variable for a roboy UI panel.

Property

property GUIController::UIFadePanels
Property which holds a dictionary to store a reference to the standard position of panels in panel mode.

Private Functions

void GUIController.Start()
Initializes the controller variables.

Intializes the UI Panels and creates them for every roboy part for every panel mode.

void GUIController.initializeFadePanels()
Initializes all fade panels which are used for the animation of the different modes.

void GUIController.initializePanels()
Initialize the position of all panels and set their corresponding roboy part reference.

void GUIController.changePageOfPanel()
Changes the page of the current panel if the current GUI mode is set to panel mode.

void GUIController.changepanelsToNextMode()
Changes to the next panel if the current GUI mode if set to panel mode.

void GUIController.changeToPreviousMode()
Changes to the previous panel if the current GUI mode if set to panel mode.

IEnumerator GUIController.changeGUIMode()
Changes GUI mode between selection and panel mode.

Return

void GUIController.positionPanels()
Positions the panels according to the template panel positions in the editor.

Private Members

Dictionary<RoboyPart, UIPanelRoboyPart> GUIController.m_RoboyPartPanelsDic = new Dictionary
Dictionary to store a reference to all UI Panels which are created at the start of the scene.

Dictionary<UIPanelAlignment, FadePanelStruct> GUIController.m_UIFadePanels = new Dictionary
Dictionary to store a reference to the standard position of panels in panel mode.

SelectionPanel GUIController.m_SelectionPanel
Reference to the *SelectionPanel*.

struct FadePanelStruct
Struct to store the position where a panel should fade in and out.

Additional classes

SelectableObject

class SelectableObject

SelectableObject is attached on every roboy part.

Is used to switch between selection states, which then again changes the material and manages GUI highlighting.

Inherits from MonoBehaviour

Public Types

enum State

Enum for possible selection states.

Values:

DEFAULT

TARGETED

SELECTED

Public Functions

void SelectableObject.SetStateSelected()

Changes the state depending on the current state and updates the result in *SelectorManager*.

void SelectableObject.SetStateTargeted()

Sets the state to targeted if the last state was default.

void SelectableObject.SetStateDefault(bool forceMode = false)

Resets the state to default if the last state was targeted (without force mode).

Parameters

- `forceMode`: Boolean to force the state switch.

Public Members

Material SelectableObject.TargetedMaterial

Material of meshes which are targeted.

Material SelectableObject.SelectedMaterial

Material of meshes which are selected.

Property

property SelectableObject::CurrentState

Public property to track the selection state for outside classes.

Private Functions

void SelectableObject.Awake()
Initializes the renderer array and default material.

void SelectableObject.changeState(State s)
Switches the state based on the parameter and manages GUI highlighting.

Parameters

- s: State to which the object should switch to.

Private Members

State SelectableObject.m_CurrentState = State.DEFAULT
Variable to track the current selection state.

Renderer [] SelectableObject.m_Renderers
Array of all renderer to change the material.

Material SelectableObject.m_DefaultMaterial
Default material of all meshes.

SelectionPanel

class SelectionPanel
SelectionPanel is the panel where you can select roboy parts with the *SelectorTool* on a GUI interface.

Whereas the components inside the panel provide functions to switch between selection states, this class is responsible to animate the switch between Selection Mode and GUI Panel mode.

Inherits from MonoBehaviour

Public Functions

void SelectionPanel.Shrink()
Starts a coroutine to shrink the selection panel.

void SelectionPanel.Enlarge()
Starts a coroutine to enlarge the selection panel.

IEnumerator SelectionPanel.shrinkCoroutine()
Coroutine to shrink the selection panel.

Fades out the UI elements, turns off the colliders and shrinks the selection panel.

Return

Public Members

Text SelectionPanel.CurrentPanelModeText
Reference to the text component to display the current panel mode like MotorForce etc.

Private Functions

void SelectionPanel.Awake()
Initializes all variables like the RectTransform and the lists.

IEnumerator SelectionPanel.enlargeCoroutine()
Coroutine to enlarge the selection panel.

Fades in the UI elements, turns on the colliders and enlarges the selection panel.

Return

Private Members

RectTransform SelectionPanel.m_RectTransform
Private RectTransform component for animation purposes.

List<CanvasGroup> SelectionPanel.m_ChildCanvasGroups = new List<CanvasGroup>()
List of all canvas groups to change the alpha value.

List<BoxCollider> SelectionPanel.m_ChildBoxColliders = new List<BoxCollider>()
List of all colliders on the UI elements to switch them off and on.

Projectile

class **Projectile**
Inherits from MonoBehaviour

Public Members

float Projectile.projectileSpeed
The speed of the projectile.

Private Functions

void Projectile.Update()
Move forward and destroy yourself if you are not in the roboy cave.

void Projectile.OnCollisionEnter(Collision collision)
Triggers a ROS external force message.

Transforms the hit point from world space to roboy local space.

Parameters

- `collision`

MeshUpdater

class **MeshUpdater**
Inherits from MonoBehaviour

Public Types

enum State
State enum to track the current state of the mesh updater

Values:

None = 0

Initialized = 1

BlenderPathSet = 2

Scanned = 3

Downloaded = 4

Public Functions

void MeshUpdater.Initialize()
Initializes the paths of the python scripts.

void MeshUpdater.Scan()
Scans the repository for the roboy models and stores them in a dictionary.

void MeshUpdater.UpdateModels()
Downloads the models from the scan dictionary which were selected by the user.

void MeshUpdater.CreatePrefab()
Creates prefabs for every model which were downloaded.

Public Members

string MeshUpdater.Github_Repository = @"https://github.com/Roboy/roboy_models/"
Github repository of the roboy models.

Dictionary<string, bool> MeshUpdater.ModelChoiceDictionary = new Dictionary<string, bool>()
Dictionary to store the users choice whether he wants to update the model or not

Property

property MeshUpdater::PathToBlender
Path to blender.exe.

Is set via the user via a file selection through the file explorer.

property MeshUpdater::URLDictionary
Public property of the URL Dic for the editor script

property MeshUpdater::CurrentState
Public property for the editor script

Private Functions

List<string> MeshUpdater.getFilePathsFBX(string sDir)
Returns fbx file paths in the given directory.

Return List of all fbx file paths.

Parameters

- `sDir`: The directory you want to search.

void MeshUpdater.attachCollider(GameObject meshGO, string path, string modelName)
Attaches a collider to the given gameObject.

Parameters

- `meshGO`: The gameObject you want to attach the colliders on.
- `path`: The path of the parent object in the Origin folder.
- `modelName`: The actual name of the visual model.

void MeshUpdater.showWarnings()
Shows warnings for each python script.

Private Members

State MeshUpdater.m_CurrentState = State.None
Current state of the meshupdater

string MeshUpdater.m_PathToBlender
Private variable for the blender path to encapsulate the get and set in a property instead of a function.

string MeshUpdater.m_PathToDownloadScript
This should be the path to the "MeshDownloader".

It is located in the ExternalTools directory.

string MeshUpdater.m_PathToScanScript
This should be the path to the "MeshScanner".

It is located in the ExternalTools directory.

string MeshUpdater.m_ProjectFolder
Cached variable of the projects assets directory.

Dictionary<string, string> MeshUpdater.m_URLDictionary = new Dictionary<string, string>()
Stores all model "Titles + URLs"

List<string> MeshUpdater.m_ModelNames = new List<string>()
Temp ModelName

MeshUpdaterEditor

class MeshUpdaterEditor
Custom editor script to be able to call functions from *MeshUpdater* at edit time through buttons.

Inherits from Editor

Solution Strategy

RoboyVR consists of different components which work together. One big part deals with the transition between the different coordinate frames of Gazebo and Unity. At first the rotations were represented via Euler Angles, this lead to gimbal locks. To avoid this we switched to quaternions. Roboy's pose needs to be converted to Unity's coordinate frame. In addition we convert the model of roboy to a unity friendly format. The other part deals with user interaction. RoboyVR uses user input to manipulate the simulation and renders the result on a GUI.

Building Block View

Runtime View

Runtime Display Information regarding Roboyparts

Runtime Physical impact on roboy (shooting)

...

Deployment View

Libraries and external Software

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

Todo

List libraries you are using

Table 2.7: 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

Presentations

Midterm WS16/17: https://drive.google.com/open?id=0BxLtAtPNIIYQOHFIRjdrajR0UVk

Endterm WS16/17: https://drive.google.com/open?id=0BxLtAtPNIIYQUVhzNHY5NlVHbVE

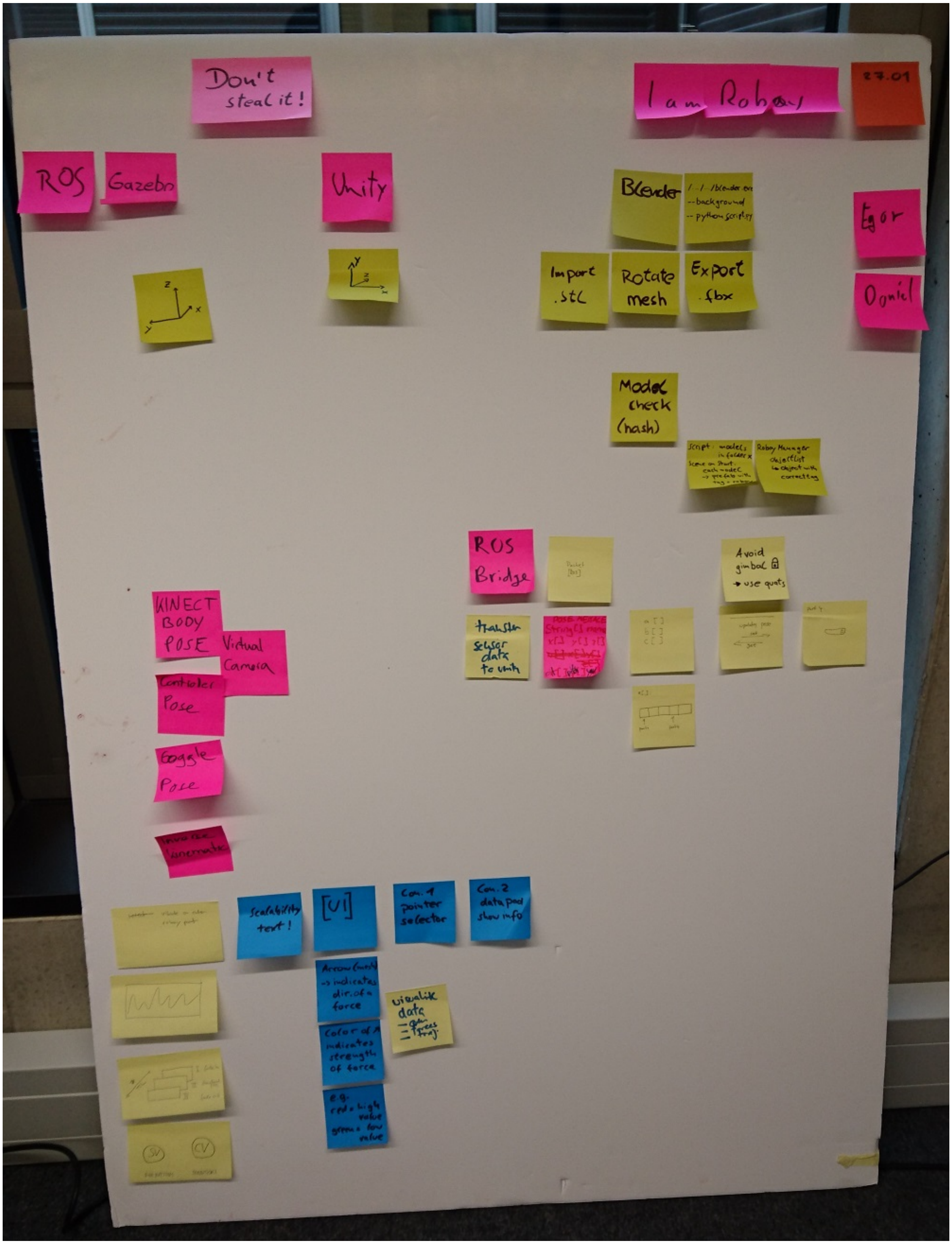


Fig. 2.28: Whiteboard showing problems and solutions that occured during development of roboyVR.

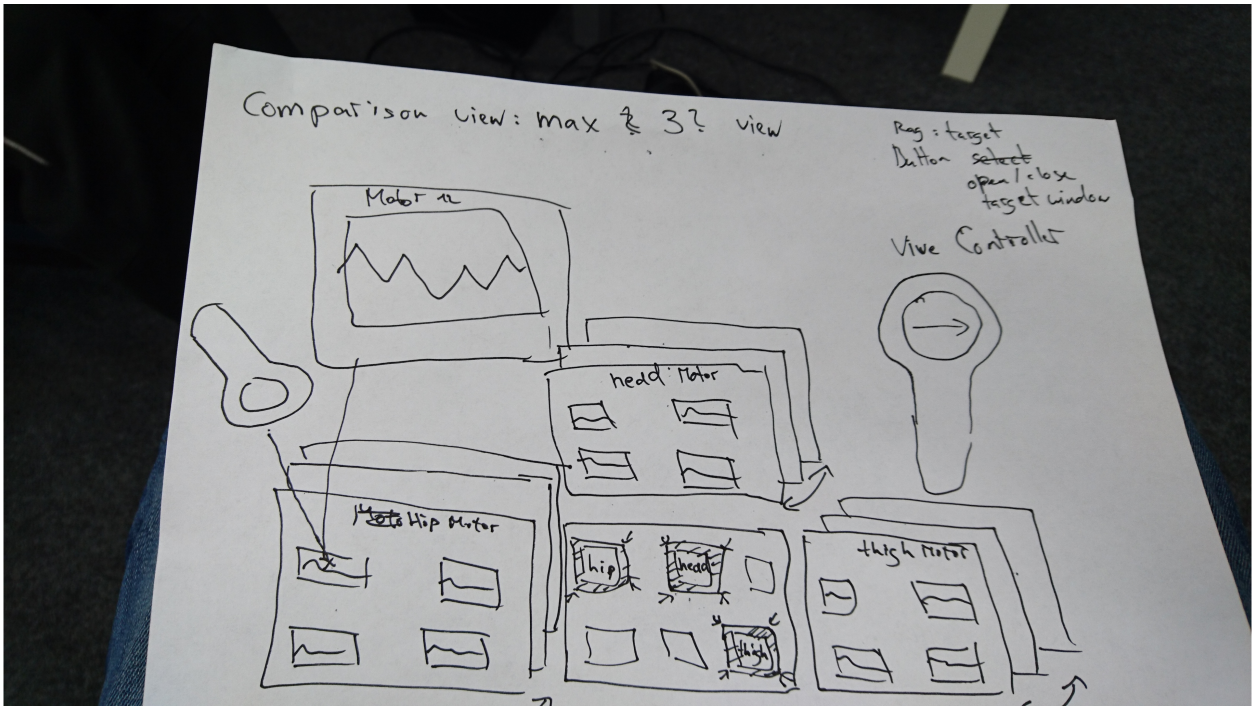


Fig. 2.29: Handdrawn sketch showcasing the design of a specific UI Panelmode (comparison).

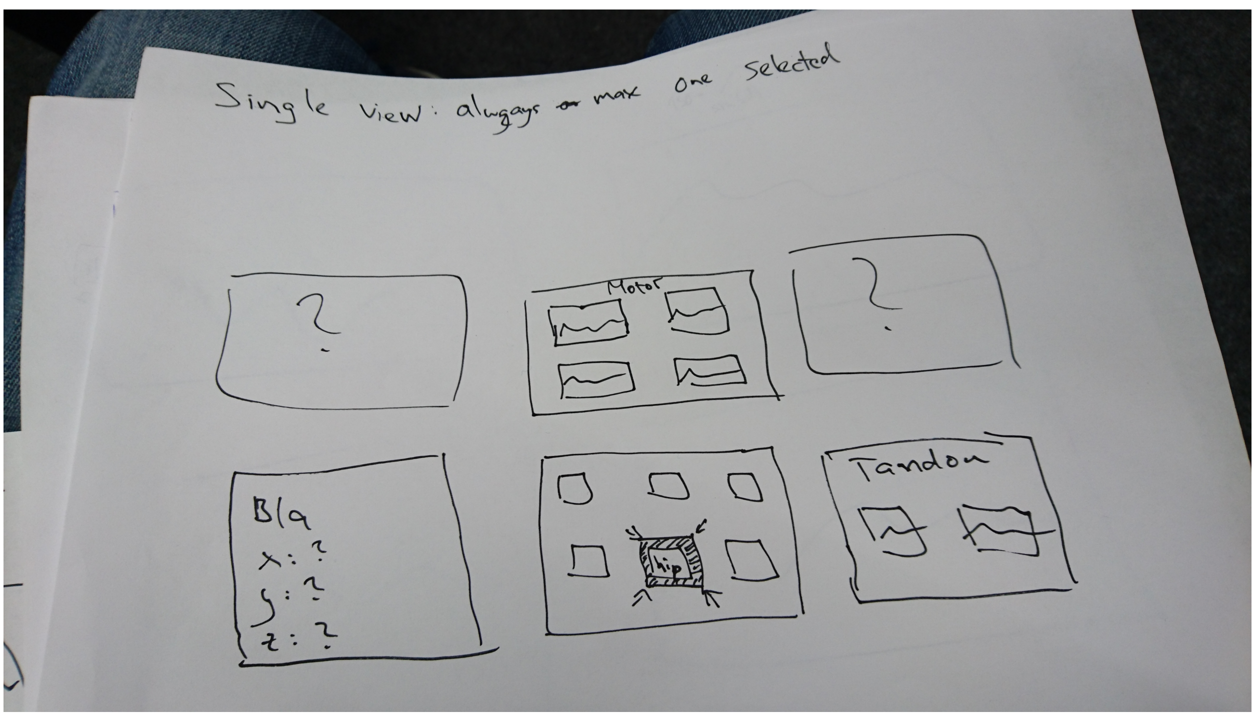


Fig. 2.30: Handdrawn sketch showcasing the design of a specific UI Panelmode (single).

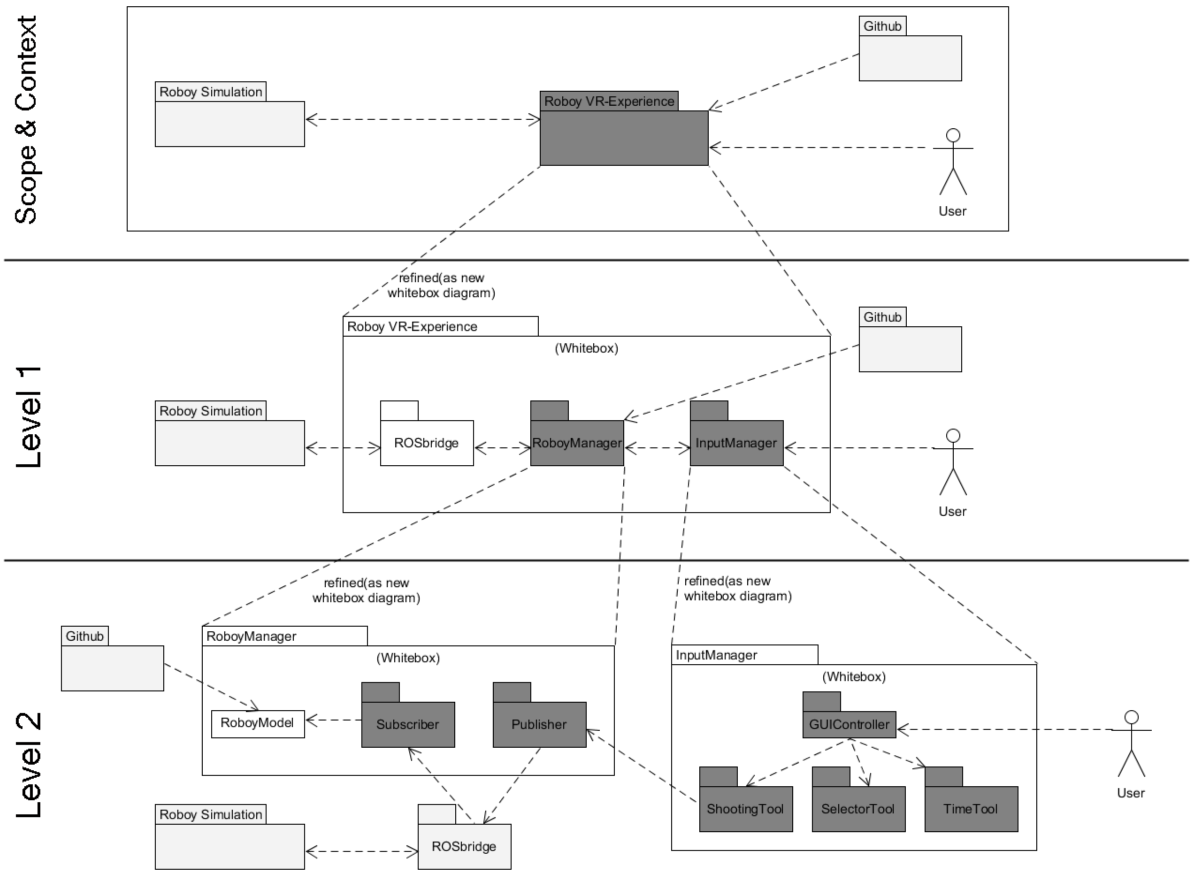


Fig. 2.31: RoboyVR Experience has several neighbouring systems like the simulation and github, it consists of various components like RoboyManager/Inputmanager and can be manipulated by the user through the HMD system.

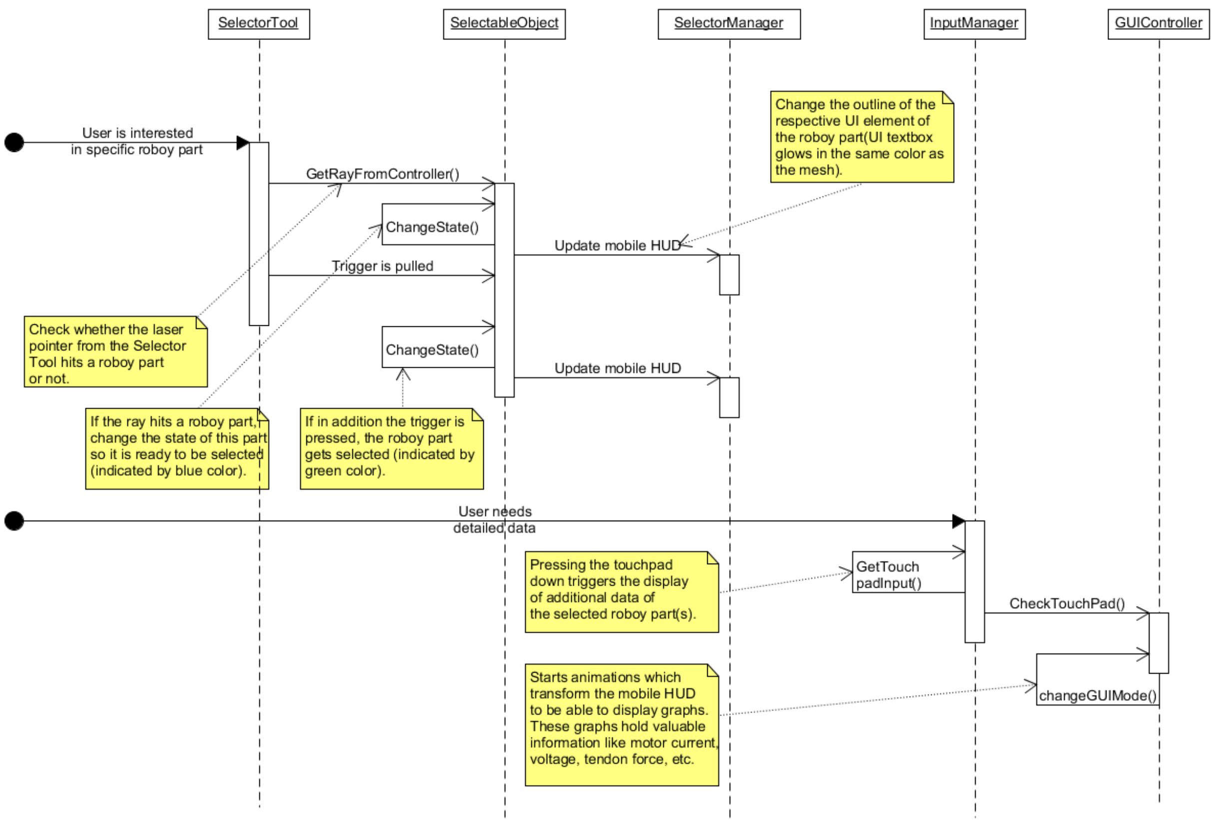


Fig. 2.32: User needs detailed information regarding specific roboy parts, e.g. power-consumption in motor24 upper_left_arm.

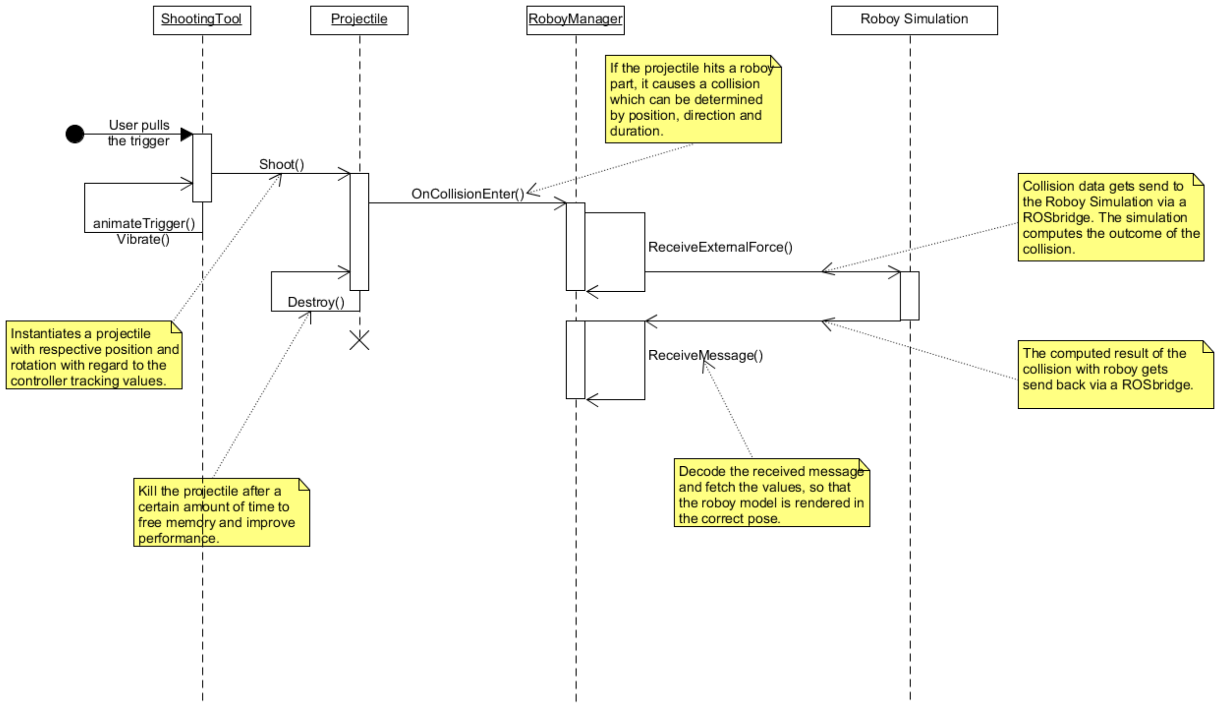


Fig. 2.33: User wants to physically harm the poor roboy and shoots a nerf dart towards him.

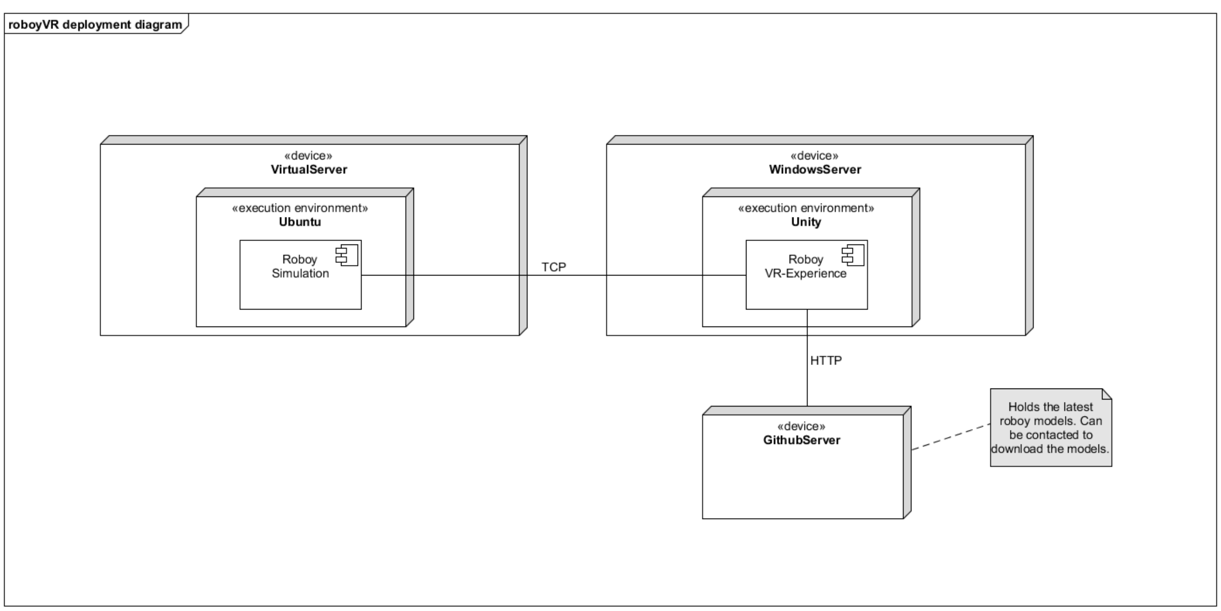


Fig. 2.34: Roboy simulation runs on a virtual machine, RoboyVR Experience runs on Unity.

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

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.

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

Examples

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

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*!

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