Technische Universität München

Distributed Multimodal Information Processing Group

Prof. Dr. Matthias Kranz

Diplomarbeit

Implementation of a Development Toolchain for the Simulation of Mobile Devices using the ROS Middleware

Author:	Stefan Diewald
Matriculation Number:	[redacted]
Address:	[redacted]
Advisor:	Prof. Dr. Matthias Kranz
Begin:	2011-02-01
End:	2011-05-27

Kurzfassung

Implementierung einer Entwicklungswerkzeugkette für die Simulation mobiler Endgeräte unter Verwendung der ROS Middleware

Mobile Endgeräte sind mittlerweile fester Bestandteil unseres täglichen Lebens. Neben immer leistungsfähigeren Prozessoren und größeren Speichern sind die Geräte auch mit einer Vielzahl von Sensoren und Ausgabeeinheiten ausgestattet. Kontextsensitive Anwendungen nutzen zum Beispiel die Daten der Sensoren, um dem Benutzer eine einfachere Bedienungs zu erlauben. Die von moderer Grafikhardware angesteuerten hochauflösenden Displays ermöglichen viele neue Konzepte um den Nutzer mit notwendigen Informationen zu versorgen. Die Entwicklung der entsprechenden Software wird dadurch jedoch immer aufwändiger. Gleiches gilt für das Testen, denn solche Anwendungen sind von vielen Faktoren abhängig, die in einer Laborumgebung teilweise nur unvollständig nachgestellt werden können.

In dieser Arbeit wurde daher eine Entwicklungswerkzeugkette erdacht und implementiert, die die Simulation von mobilen Endgeräten in einer virtuellen Umgebung erlaubt. Das Konzept beleuchtet dabei alle gängigen Sensoren und Ausgabeeinheiten von Mobilgeräten und gibt Ideen zur Umsetzung dieser Einheiten in einer 3D Simulation. In einer virtuellen Umgebung können damit zum Beispiel viele verschiedene Szenarien erstellt werden, die umfangreiches Testen von Anwendungen erlauben. Exemplarisch wurde ein System zur Videoeingabe und -ausgabe realisiert, mit dem Kameras und Displays simuliert werden können. Für die Realisierung wurde die *Robot Operating System (ROS)* Middleware eingesetzt, für die bereits viele Werkzeuge zur 3D Simulation von Hardware existieren.

Abstract

Mobile devices have become an inherent part of our daily lives. Besides the increasingly powerful processors and larger memories, the devices are equipped with a variety of different sensors and output units. The data from the sensors are for example used by context-aware applications to provide a better user experience for the user. High-resolution displays, driven by powerful graphics hardware, enable new concepts for providing users with helpful information. However the development of software, which supports these features, is getting more and more complex. The same holds for the testing, as those applications depend on many factors, that can often only be tested incompletely in an artificial lab environment.

Thus a development toolchain for the simulation of mobile devices in virtual environments was conceived and implemented in this work. All common sensors and output units are considered and ideas for implementing those units in a 3D simulation are given. Deploying the simulated system in a virtual environment allows for instance to create several scenarios, that enable extensive testing of mobile applications. A system for video input and output was realized exemplarily, which allows the simulation of cameras and displays. The realization was done using the *Robot Operating System (ROS)* middleware, as there exists already a large set of tools for 3D simulation and for simulation of hardware components.

Contents

Chapter 1

Introduction

Mobile devices are "the swiss army knives of the 21st century" [1]. This statement from 2004 is today already the reality. Smartphones, Tablet PCs, handheld Personal Digital Assistants (PDAs) and other mobile devices are integral parts of our daily lives and a life without them can not be imagined anymore. The small pocket devices accompany us everywhere. They are our communicators, calendars, cameras, encyclopedias, toys and many more. In 2012 the sales of mobile devices will even outnumber the sales of notebook and desktop PCs together [2]. Over 1.6 billion mobile devices were sold worldwide in 2010 [3]. The first quarter of 2011 even exceeded last year's sales by 19 percent with a whole of 428 million devices [4].

1.1 Motivation and Goals

Due to the large demand in mobile devices, the manufacturers of the portable devices release new devices on a daily basis. On the one hand, this advance is helpful for the mobile device application developers, since the new devices come with bigger and better displays, larger memory and faster processors. On the other hand, they have to support an enormous number of different hardware with different operating systems and versions, to reach a large number of users. Solely the effort for testing the software on the many different platforms is tremendous.

Besides the increasingly powerful hardware, mobile devices are today equipped with a large number of different sensors, like accelerometers, GPS, ambient light sensors or short range radios. Including sensors in a user device offers more information about the user's context. Knowing what the user is doing, where he is, how he feels and what is next to him, enables to create a completely new user experience. Correctly interpreted contextual information can for example help the user focus on what is important for him at the moment or even in life.

This new complexity is great challenge for both, hardware and software creators. Hardware developers are continuously adding new units to the phone, but have to make sure that the usability of the device is kept or, ideally, even enhanced. Software developers have to handle, combine and process a lot of information in order to create context-aware applications that really add value for the user. Building prototypes for the evaluation of new hardware and software features is very time-consuming and costly. In addition to the development, the verification of the many different features is getting harder from day to day. Many things can even not be tested in an artificial lab environment.

In order to improve the development and testing process of mobile phones and application for them, a new method is proposed in this work. Shifting the primary evaluation and a major part of the complete testing into a virtual environment with a powerful simulation engine can facilitate and accelerate these processes.

1.2 Outline of the thesis

The remainder of this works is arranged as follows. Chapter 2 gives a general overview of the current situation in mobile phone development and shows a future prospect in the view of virtualization. The virtualization of a mobile phone and its sensors and output units is described in Chapter 3. The implementation of a virtual video input and output system for the Robot Operating System (ROS) middleware is presented in Chapter 4. Based on this virtual video system, a complete toolchain using the Android device emulator is described in Chapter 5. The implemented system and its components are evaluated in Chapter 6. Chapter 7 concludes the work and gives and idea, what can be done with this toolchain in the future.

Chapter 2

Mobile Phone Development – A Future Concept

Before a concept for the enhancement of mobile application development can be be created, one has to analyze the current situation and identify the deficiencies and needs. Section 2.1 gives an overview of the current state of the mobile software development and tries to identify the major problems, based on representative surveys. In Section 2.2 the new concept of simulating mobile devices in an virtual environment is introduced and it is described how it can resolve some of the major problems of today's mobile application development. To verify the theory, a sample development toolchain for the simulation of mobile devices was implemented using the Robot Operating System (ROS) middleware. The basics of the implementation are explained in Section 2.4, before it goes into the details of the system in the following chapters.

2.1 Development today

In January 2011, Appcelerator and IDC surveyed 2,235 developers on perceptions surrounding mobile operating system priorities, feature priorities and mobile development plans in 2011 [5]. The following three major trends were identified by the survey:

- "Always connected, personal, and contextual". This is the main slogan for the future. It is expected, that in addition to cloud services, the integration of social and contextual (especially location) services will define the majority of mobile experiences.
- Rapid Innovation: on average, each developer will work on 6.5 different mobile applications in 2011 (2010: 2.3). The companies try to minimize time-to-market and update cycle time for not falling behind.
- Cross-platform will be mandatory. Most applications will at least support iOS (iPhone and iPad) and Android (phones and tablets), since these systems have the largest market share.

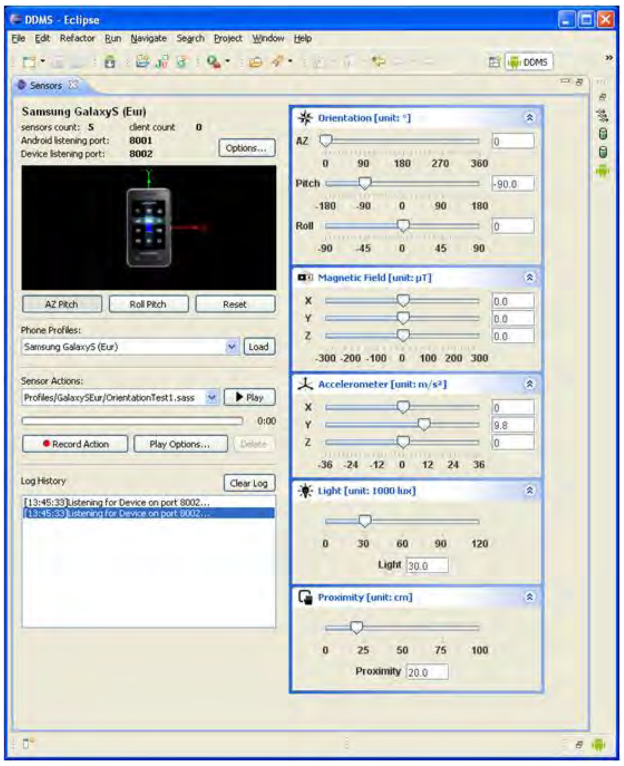
These results show, that there will be a race of software innovations, similar to the ongoing race of hardware innovations. New applications will have more features than ever before. But the time spent for development, evaluation and testing should at the same time be less then 50% then before.

Looking at another VisionMobile survey from mid 2010 [6] shows, that these goals will be very hard to reach. Although there are already good development tools, like integrated development environments (IDEs) or device emulators, for most of the platform, debugging and testing of mobile application is still a major issue. In most cases, testing of mobile applications is done by manual execution of test cases and visual verification of the results. In addition to the tests done by developers themselves, the second choice are internal beta testers. Peer reviewing, focus groups and external testers are to expensive or to time-consuming for most of the companies.

For the development and testing of user interfaces for mobile devices already some strategies and tools are available. Examples are the user centered design (UCD) concepts like introduced in [7]. With eggPlant [8] for iPhone and Robotium [9] for Android phones, there are beginnings of automated user interface test frameworks available for mobile devices. These tools offer support for almost all UI elements and thus can perform a variety of test cases, derived from different scenarios. But the available tools can only simulate touch input events. Other sensor inputs cannot be simulated with these tools.

To ease the development and testing of mobile applications, the vendors include a device emulator in their software development kits (SDKs). They emulate the basic device hardware and possess a standard system image, which includes a standard version of the mobile operating system and some tools. Some emulators offer settings for changing hardware values. For example, the Android emulator allows to set different screen resolutions and densities. Furthermore the connectivity to the networks and the battery status can be changed during runtime. The emulator even allows to use an audio and NMEA 0183 compliant GPS devices from the host.

But other sensors' data, like accelerometer or thermometer data, cannot be set during runtime. Due to this reason, tools have been created, that allow setting these sensors' values. OpenIntents has developed the Android SensorSimulator [10]. Is is an client-server application that allows to set accelerometer, compass, orientation and thermometer data. The values can even be biased with adjustable, random noise. A simple GUI allows to set the device orientation intuitively. Following the client-server principle, a live camera feed for Android is enabled by [11]. It supports several different image formats and network protocols. The big disadvantage of these tools is, that one has to add code to his application, which enables to use this simulated data. Although the tools try to mimic the behavior of the real sensor interfaces, there are some differences and two different codes have to be maintained: one for the emulator with the simulated sensors and one for the real devices.



Samsung GalaxyS (Eur)
sensors count: 5 client count 0
Android listening port: 8001
Device listening port: 8002

Phone Profiles: Samsung GalaxyS (Eur)

Sensor Actions: Profiles/GalaxySEur/OrientationTest1.sass

0:00

Log History

```
[13:45:33]Listening for Device on port 8002...
[13:45:33]Listening for Device on port 8002...
```

AZ 0
Pitch -90.0
Roll 0

X 0.0
Y 0.0
Z 0.0

X 0
Y 9.8
Z 0

Light 30.0

Proximity 20.0

Figure 2.1: The *Samsung Sensor Simulator* GUI. It enables to set several sensors' data for use with the Android emulator. Source: [12]

The *Samsung Sensor Simulator* [12] is similar to the OpenIntents SensorsSimulator, except for the fact that Samsung delivers a modified system image for the Android simulator. Exchanging the original system image with the modified image allows to access the orientation, magnetic field, accelerometer, light and proximity data exactly like it is done on the real device. When working with the *Samsung Generic Android Device* profile for the emulator, one can further use the following sensor data: gravity, linear acceleration, rotation vector, pressure, temperature and gyroscope. In addition to the live manipulation of these values, the Eclipse integrated tool also allows to record and playback sensor data scripts. A screenshot of the GUI is depicted in Figure 2.1.

One can see that there are already several accessories, that can enable faster development, evaluation and testing. But the testing paradigm using these tools stays the same: testing of mobile applications is done by manual execution of test cases and visual verification of the results.

2.2 Going virtual – a new concept

As the survey in Section 2.1 showed, faster development and testing will be needed to keep track with other competitors. For context-aware applications, it is no longer sufficient to have tools that only test the user interfaces. A complete toolchain for testing more aspects of the new generation software is necessary.

How could such a toolchain look like? A simple solution would be to take one of the available sensor simulators, and extend it with further important sensors. An image stream for the camera could for example be added, that could be fed by a webcam or a simple video file. In order to perform automated testing, parts from the automated user interface testers can monitor the device's outputs. With the help of a simple scripting engine, sensor data variation for various test cases could be created once and replayed in every test cycle.

Although this concept enables automated and repeatable testing, it is very unintuitive and artificial. The creation and the variation of test cases for this tool would be very complicated. The creator of the scripted test cases has to keep in mind the timing, causality and dependencies of the events, all the time. In this way, a simple test case can quickly become a very complex task.

In order to facilitate this process, the concept of virtual prototyping can be applied. A virtual prototype is “a computer simulation of a physical product that can be presented, analyzed, and tested” [13]. The definition is often limited to the simulation of physical properties like form or design. But *Lister et. al* also used the term for a 3D simulation of a digital mock-up which was driven by full system level hardware simulator [14]. Thus the virtual prototype could be used like a real electronic product. This concept will be used and evolved, in order to create an intuitive development and testing toolchain for mobile devices.

The basis of the toolchain is a virtual environment, in which a 3D model of the mobile device is operated. This allows the user to see, how the device looks like. In addition to that, the device is made functional by a software system that is interconnected with the virtual environment system. Figure 2.2 shows the coupling of the virtual environment to a mobile device simulator. The sensors, that are also simulated in the virtual environment, make their data available for the mobile software simulator. In turn, the different outputs of the software simulation tool are sent back to the virtual environment, which can render these outputs to the mobile device model. The virtual environment, in which the device is simulated in, does not have to be an empty 3D space. It can portray a real world scenery or a fictive, futuristic world. Rigid body simulation further adds basic physical laws to the virtual reality environment. This allows the perception of the device and its functionalities, like in the real world. In this way, the user can easily comprehend the actual state of the mobile device. Established 3D input equipment, such as the 3D mouse *SpaceNavigator* from *3Dconnexion*, can be used for navigation and manipulation in the environment.

To enable the automated testing, a external testing instance has to be added. The principle

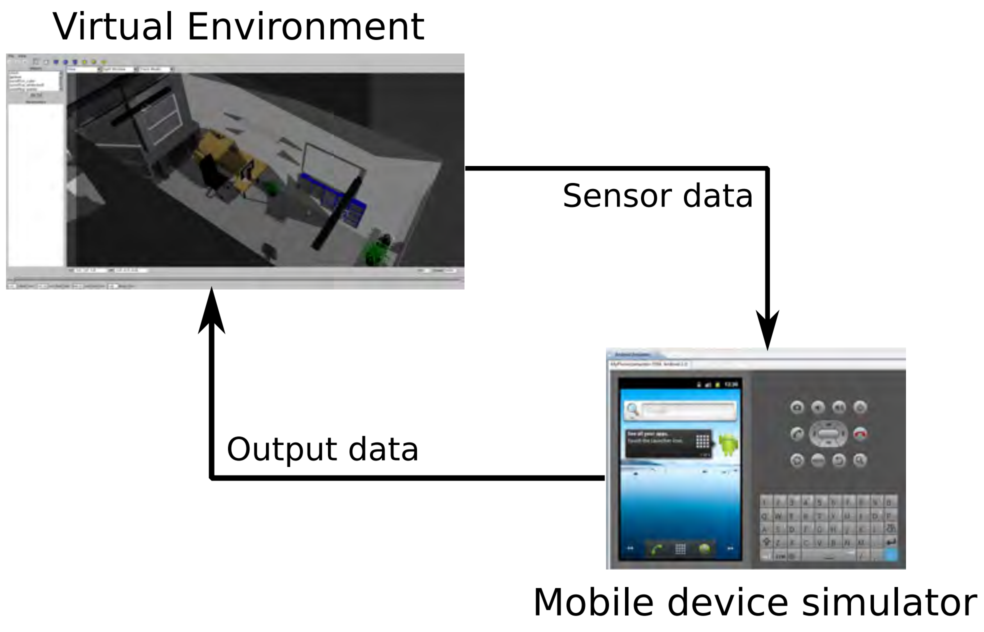


Figure 2.2: Coupling of the virtual environment to the mobile device simulator.

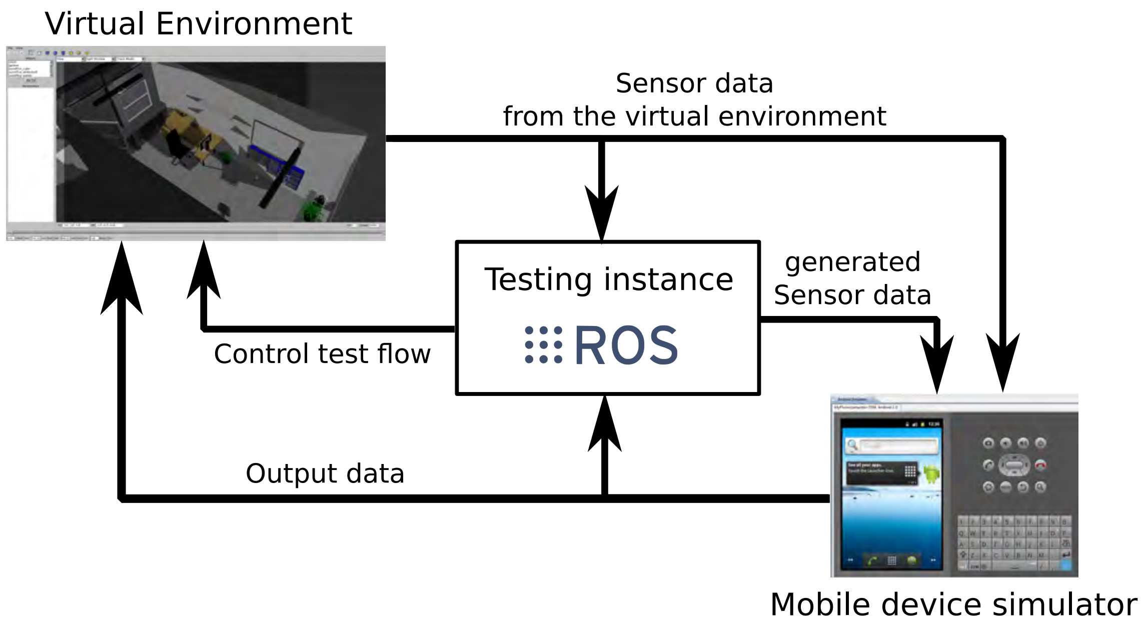


Figure 2.3: A testing instance is necessary to enable automated testing.

of the new system is shown in Figure 2.3. The testing instance is not only used for the evaluation of the device's outputs, but also controls the flow of test processes. Besides, this element also generates some sensors' data, that cannot be simulated in the virtual environment. An example for generated data is the temperature value from a thermometer, because most virtual environment do not support the simulation of thermal effects. But since this instance is connected to the environment and to the software simulator anyway and also coordinates the whole automatic testing process, this adds no additional complexity to the simulation. The state of those not directly in the simulation perceptible effects, can be displayed to the user with the help of common visual indicators in the virtual environment.

Considering the mobile software simulator, that gets the data from the simulated device (both from the virtual environment and the test instance), there are two basic concepts, how it can be implemented. If one only wants to quickly verify a new algorithm for mobile devices, that depends on one or more sensors, the software can directly be run on the host PC and use the defined interfaces to device simulation environment. In the case, that an implementation for a specified mobile device has to be tested, a fully featured mobile device emulator can be connected. In both cases, the development and testing can be significantly accelerated.

As a sample scenario a simple object detection for example for cups running on the mobile device shall be considered. In order to perform a proper test of the implemented cup detection algorithm, one needs to take several hundreds or thousands of photos, as one needs photos:

- from different perspectives and angles,
- showing various shapes of available cups,
- different graphical designs,
- at all thinkable light conditions,
- included in different object sceneries
- and seen on different backgrounds.

Whereas in a virtual environment, one can simply load different 3D models of cups, having every imaginable texture, standing in various sceneries and on distinct backgrounds. The whole scenery can be rendered from every thinkable perspective and with different light conditions on demand. Solely the data acquisition in the real world would probably take longer, then the complete testing in the virtual environment.

In addition to the fast variation and generation of test scenes, a complete virtual system further enables recording and playback of all test and output data. This can either be used for sharing test data with other developers and tester or for the reconstruction of malfunctions. This is especially useful for debugging applications, that depend on various inputs.

In summary, a development toolchain for mobile devices, that allows simulating the device in a virtual environment, has several advantages:

- Intuitive usage and rapid visual verification thanks to the 3D simulation.

- Advancing simulation time and appliance of basic physical laws ensure causality in the testing environment.
- Fast generation, variation and execution of test cases.

2.3 Demarcation of the term virtual

Before the basics of the prototypical implementation can be explained, it has to be clarified what is meant by the term *virtual* in this work. When talking about a virtual environment (VE), a computer generated 3D setting is meant. The representation of this setting can be realized in various ways, but this is not relevant for the definition of our VE. When referring to a virtual mobile device, the 3D model of the device and its simulated functionalities are meant. The virtual device is a resident of the virtual environment.

The virtual mobile device can be equipped with several virtual sensors and virtual output devices. The term virtual here only implies, that these sensors and output devices are parts of the virtual model, that is simulated in the virtual environment. This can be either be units, that emulate the characteristic behavior of real devices, or complete novel devices with a new functionality. Especially the definition of *virtual sensor* is important. In contrast to the meaning in this work, *Kabadayi et al.* defined a virtual sensor as an abstracted sensor, that encapsulates the data from several sensors [15]. And yet other used the term virtual sensor for data from a calendar [16], social-networks or other web-based services.

2.4 Prototypical implementation using the ROS middleware

In order to evaluate the proposed concept, an implementation of the suggested development toolchain was created in this work. For the simulation of mobile devices in a virtual environment, the Robot Operation System (ROS) was chosen [17]. The ROS is one of the major middleware systems in the domain of robotics, running for example on the PR2 [18] or the Fraunhofer Care-o-Bot III [19]. Recently, it is also used for immobile robots (ImmoBots) such as "intelligent environments" or ambient assisted living (AAL) environments [20]. The *Cognitive Office* [21] is for example a fully featured intelligent office environment, that uses the ROS as background system for the real implementation and for a complete 3D simulation.

The ROS is particularly suitable for the proposed toolchain as it provides sophisticated hardware and software abstraction and message passing. In addition to that, the middleware brings along several visual multi-robot simulation tools, for example the 2D simulator

Stage and the 3D simulator Gazebo. Both tools are parts from the Player/Stage middleware, which has already been used for intelligent environments, such as the AwareKitchen [22], before. In order to get a simulation that is something like a synthetic mirror of the real world, 3-dimensional simulation is performed. For this reason, Gazebo is mainly used for the simulation. As the whole ROS and its packages are open source with permissive licenses, the components can easily be customized and extended.

The Gazebo package for the ROS already offers some virtual sensors which can be deployed in the virtual environment. These are: cameras, laser scanners, an inertial measurement unit (IMU) [23], bumpers and an odometry sensor. It further provides output units, like a differential drive or a pattern projector. These elements, mainly designed for robots, can serve as models for the implementation of the sensors, that are needed for the simulation of a mobile device. Other key features of Gazebo are the rigid body dynamic simulation, which is realized by the *Open Dynamics Engine (ODE)* library [24], and a plugin system, which allows to add functionalities dynamically during runtime. The graphics renderer is based on *OGRE (Object-Oriented Graphics Rendering Engine)* 3D rendering engine [25] and the graphical user interface (GUI) is realized with wxWidgets [26].

The mobile software for the simulation is implemented in two different ways. For testing software algorithms, the software behind the model in the simulation is developed and run as a standard ROS node on the PC. For testing mobile applications on a mobile system, the Android emulator from the Android SDK [27] is used. The integration of the emulator is described in Chapter 5.

Chapter 3

Virtualization of a Mobile Device

This chapter describes how a virtual mobile device can be modeled and how its virtual sensors and output units can be implemented in a virtual environment. As already mentioned in Chapter 2, the ROS middleware was used for the prototypical implementation. The complete sample toolchain, which is presented in Chapter 5, further uses the Android emulator. For that reason, the examples for implementation ideas are focused on those tools. But the underlying principles are also valid for other systems. In most cases the given concept can easily be transfered to other solutions.

3.1 The 3D Model of the Mobile Device

The first step in the virtualization process is the creation of a 3D representation of the mobile device. This can either be an early sketch, a simplified or detailed model. But as virtual prototyping is anyway the standard for the development of those products, detailed 3D models will already be available in an early stage. By using the metric system for the model of the device and for the creation of the virtual environment, the user can see the the device in perspective.

Besides the physical dimensions, the constant rigid body properties of the model have to be set [24]. Those are the mass of the body, the position of the center of mass and an inertia tensor, which describes how the body's mass is distributed around the center of mass [28]. For a proper collision detection, a proper bounding box has to be set. This can either be the model itself or a simple box, which surrounds the original model. Gazebo offers a special view for the visualization of the collision bounding boxes, that can be used to verify the setting. In Gazebo, robots and smart objects are described in the Unified Robot Description Format (URDF) [29]. It holds the kinematics, the visual representation and the collision model of the robot. The URDF is primarily made up of two different basic elements: *links* and *joints*. *Links* specify rigid body parts which are described by the properties stated above. *Joints* are used to describe the linkage of two links.

Code snippet 3.1.1 URDF description for a cube with mass and collision settings.

```
<link name="simple_box">
  <inertial>
    <origin xyz="0 0 0.25" rpy="0 0 0" />
    <mass value="1.0" />
    <inertia ixx="0.042" ixy="0" ixz="0" iyy="0.042" iyz="0" izz="0.042" />
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="0.5 0.5 0.5" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 255 255 1.0" />
    </material>
  </visual>

  <collision>
    <origin xyz="0 0 0.25" rpy="0 0 0" />
    <geometry>
      <box size="0.5 0.5 0.5" />
    </geometry>
  </collision>
</link>
```

Code Snippet 3.1.1 shows a sample URDF definition of a 3D cube model. It has a defined mass of $m = 1.0$ kg and an edge length of $s = 0.5$ m. The collision model has the same size, orientation and position like the visual geometry. The inertia tensor has only entries for the principal axes of the cube. Its calculation is shown in Equation 3.1, which follows the example in [30].

$$\mathbb{I}_{cube} = m\frac{s^2}{6}\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix} = 1.0\text{ kg}\frac{(0.5\text{ m})^2}{6}\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix} \approx 0.042\text{ kg m}^2\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix} \tag{3.1}$$

The inertia tensor $\mathbb{I}_{cube}$ given in Equation 3.1 is only valid for a cube. But most mobile devices can not be matched with the shape of a cube, but rather with the shape of a more general cuboid. For this reason, the formula for the inertia tensor of a cuboid is given in Equation 3.2 with h describing the height, w the width and d the depth of the cuboid.

$$\mathbb{I}_{cuboid} = \frac{1}{12}m \begin{pmatrix} h^2 + d^2 & 0 & 0 \\ 0 & w^2 + d^2 & 0 \\ 0 & 0 & h^2 + w^2 \end{pmatrix} \tag{3.2}$$

For the visual geometry of the mobile device, a ready-made 3D model, available from various 3D artists or manufacturer, can be used as basis. However it should be kept in mind, that the use of high polygon count models may slow down the simulation and add unnecessary complexity. Especially for the collision model one should consider the use of simple geometric forms like boxes, cylinders or spheres, that cover the outline of the visual geometry.

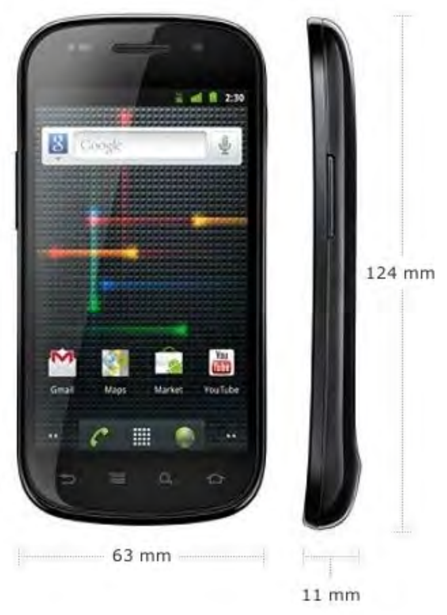


Figure 3.1: The current Android developer phone *Google Nexus S*.
Image Source: `http://androidhandys.de/samsung-nexus-s.html`

For an example implementation, a model of the current Android developer phone *Google Nexus S* was chosen. The phone is depicted in Figure 3.1 and has the following outlines: h = 123.9 mm, b = 63 mm and d = 10.88 mm. Its overall weight is 129 g. Code Snippet 3.1.2 shows the URDF definition of the mobile phone based on a 3D Collada model and the above specifications. In order to reduce simulation complexity, the collision model is described by a box which surrounds the whole visual model.

Code snippet 3.1.2 URDF description for the *Google Nexus S* model.

```
<link name="simple_box">
  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <mass value="0.129" />
    <inertia ixx="0.000166" ixy="0" ixz="0"
             iyy="0.000044" iyz="0" izz="0.000208" />
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <mesh filename="google_nexus_s.dae" />
    </geometry>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="0.063 0.1239 0.01088" />
    </geometry>
  </collision>
</link>
```

The 3D model of the *Google Nexus S* was originally distributed as a Wavefront OBJ file. Although Gazebo supports OBJ files, the model was displayed erroneously in Gazebo. It was rendered more as a wireframe model with several holes in its surface. The faulty rendering of the OBJ file can be seen in Figure 3.2.

For this reason, the model was converted to a 3D Collada file using MeshLab 1.3.0a [31]. The Collada mesh file could be loaded in Gazebo, but the model was rendered partly transparent from the back. The erroneous representation of the Collada model is depicted in Figure 3.3. The reason for the faulty rendering of the Collada mesh is probably a wrong alignment of face normals.

Although several days were spent on the wrong rendering of there models, the problem could not be isolated. The debugging of the models was very hard, since all tested 3D modeling and mesh processing tools, such as *Blender*, *Autodesk 3ds Max* or *MeshLab*, displayed the model files correctly. In addition to that, the loaded Collada model leaded to random Gazebo crashes due to shading buffer overflows. The problem is already known [32]. It is caused by models with a high triangle count and as a temporary fix, it was recommended to switch off shadows. In a future release, the issue will be fixed by switching to a more inexact but simpler shadow technique. Due to the many problems with the high

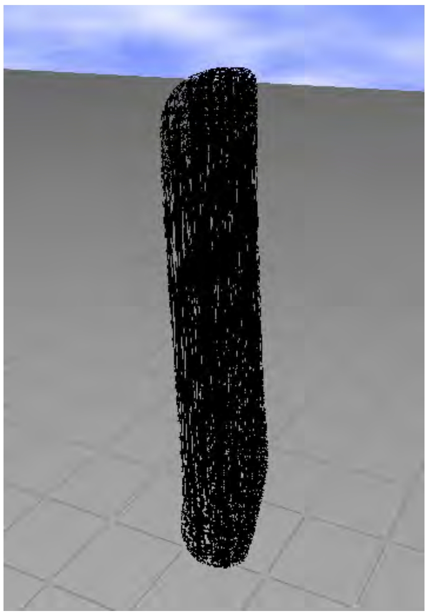


Figure 3.2: Faulty rendering of the Wavefront OBJ model in Gazebo.

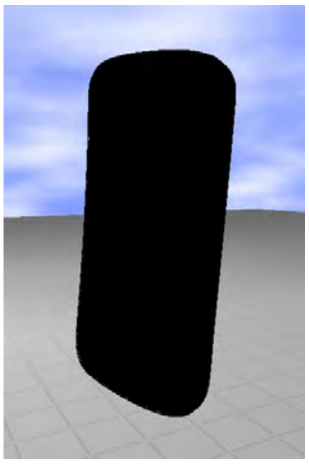
detail *Google Nexus S* model, a simplified model for a mobile device was created.

The base of the simplified model is a black box with the outlines of the *Google Nexus S*. The display is another very-thin box on the front-site of this basic box. Besides the display, the camera was modeled as a cylinder on the back-site of the model. It is depicted in Figure 3.4. In order to have a realistic model of a mobile phone, the other physical details, such as mass or inertia data, also correspond to the *Google Nexus S* model.

3.2 Level of Abstraction and Correctness

In Section 3.1 it was already mentioned, that one has to think about the desired degree of detail. Of course, a photo-realistic 3D model is visually more attractive, but for engineering new kinds of applications, a simplified version without non-functional elements is in most cases sufficient. The same consideration has to be done for simulated sensors and actuators. Choosing a high level of abstraction can significantly cut down the implementation time. However it can lead to inaccurate models, that do not reflect the identical functionality of the real device. Hence one has to define the adequate degree of detail for each element prior to the implementation.

When one wants to test a new kind of hardware for a mobile device, it could for example be necessary to use a model, that also includes physical effects. For example, when testing a magnetometer, which should be used as a digital compass, the electromagnetic radiation from other electronic devices and the distortion from ambient magnetic materials have to


(a) Front view.

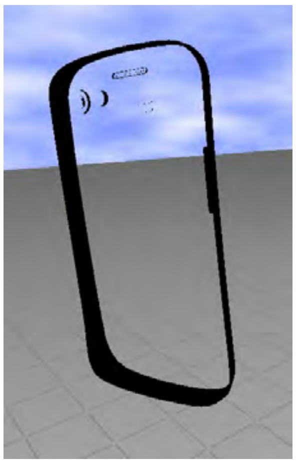
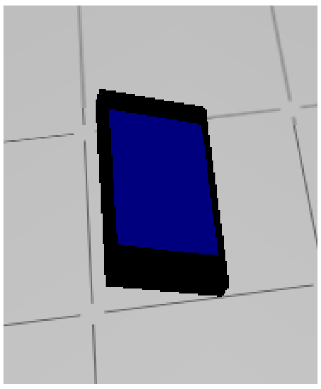

(b) Faulty back view.

Figure 3.3: The Collada model in Gazebo.


(a) Front view, the device's display is shown in blue.

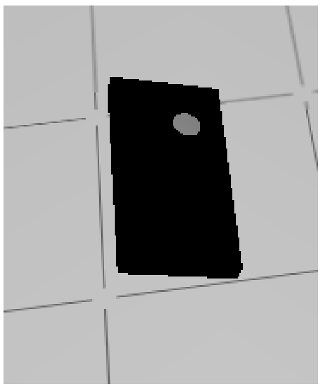

(b) Back view, the gray circle adumbrates the camera.

Figure 3.4: A simplified model of a mobile device.

be considered. But while the development of an application, that uses orientation data from the magnetometer, it is in most cases sufficient to add some random error or constant offset to the perfect data.

Since the focus of this work is to create a toolchain which allows simpler application development, the following implementation ideas are described in a high abstraction level. The goal is to describe the basic functionality considering only the most important and common errors and outside influences. But at the same time, it should be kept in mind, that there are always several effects, that are not modeled or only can simulated with great effort. Those are, for example, physical effects like:

- Electronic noise, which can distort signals or can lead to wrong sensor values (for example when using analog-to-digital-converters ADC)
- Quantization errors and distortion, since the simulation is running completely in the digital domain.
- Physical limitations and constraints, like the Nyquist-Shannon sampling theorem or, more specific, the imaging of a lens system.
- Interference and malfunctions of devices, which can lead to unpredictable errors.

In addition to those hard to model effects, the correctness of the model is another key point. Particularly for complex systems, the correctness is hard to proof. For manageable systems, model checking is a promising starting point for the verification of the required specifications. But the effort to describe the specifications, for instance in temporal logic, can grow exponentially with the complexity. As a consequence, it is again reasonable to concentrate on the substantially needed functions. In the ideal case, the model can be simplified in such a way, that it can be verified with a small number of test cases.

3.3 Virtual Sensors

A sensor is a device that produces a measurable response to a change of a physical quantity. In mobile devices sensors are mainly used to determine the context of the device. In the following subsections, the common and some not yet common mobile device sensors and values, which can be derived from other devices, are explained and analyzed. Based on the explanation of the sensors and the derived values, it is sketched how they can possibly be simulated in the virtual environment.

3.3.1 Location

The research introduced in Section 2.1 has clearly shown, that location-based services (LBS) are a running up field for mobile application. The location is a very important context which for example allows navigation, finding of nearby points of interest (POI) or location adjusted advertising. In general one can distinguish between outdoor and indoor localization methods.

GPS – The main outdoor position provider

The Global Positioning System (GPS) is today's most popular choice for positioning in open-sky environments. The system uses 24 operating satellites that transmit one-way signals at the same instant. These signals, moving at speed of light, arrive at slightly different times at the receiver, because some satellites are further away than others. Out of these delays and the Ephemeris data, which is included in the signal, the distance to the satellites can be calculated. When at least four satellites' signals are received, the GPS receiver can determine its current position in three dimensions. The position is commonly handled in geographic coordinates, which means that one will get the latitude (Lat) and longitude (Long) of the current position. Due to the low signal level of the GPS signals and thus the need for a direct view to the satellites, standard GPS receivers are often very unreliable in urban areas. Another problem is the duration for acquiring a GPS lock. The so called Time To First Fix (TTFF) can easily exceed ten minutes in some situations. In order to counteract those problems, most mobile devices are today equipped with Assisted GPS (A-GPS) [33]. A-GPS makes use of GPS *acquisition data* broadcasted by the mobile network base-stations, that have reliable GPS receivers. With the help of this initial data, the TTFF can be improved by many orders of magnitude [34]. Additional provided *sensitivity data* from the network can further facilitate the signal reception in adverse areas and in some cases even inside buildings.

Besides the *NAVSTAR GPS*, which is operated by the US military, there are several other global satellite navigation systems in development. The Russian GLONASS system is in a renewal phase and global coverage is expected to be reached within 2011. A new generation of satellites, called GLONASS-K, is currently under development. The *Galileo positioning system* from the European Union and the Chinese COMPASS navigation system are in the initial deployment phase. Galileo is expected to be compatible with the modernized GPS systems, but due to several delays, the system is scheduled to be fully operational by 2020 at the earliest. This development shows, that global navigation satellite systems will most likely remain the main system for outdoor navigation. The renewed systems will enhance the accuracy and the reliability.

A patch for adding GPS capabilities to Gazebo is available at [35]. The patch includes model and world files for the demonstration of the GPS interface. The current implementation only supports perfect GPS localization, but error injection could be added without great effort. Code snippet 3.3.1 shows a sample Gazebo model entry for a GPS sensor. The stated position, expressed as *LatLon*, represents the starting coordinates. When the object moves, to which the GPS sensor is attached to, the position is updated with respect to the starting position. By using the Universal Transverse Mercator (UTM) coordinate system for the addition of the moved distance, the resulting *LatLon* coordinates are very accurate, just depending on the accuracy of the starting point coordinates. In order to use this extension with ROS a position publisher or service to query the current position has to be implemented.

Code snippet 3.3.1 A GPS sensor sensor included in a Gazebo model description.

```
<sensor:gps name="gps_1">
  <LatLon>48.148940 11.566730</LatLon>

  <controller:generic_gps name="gps_controller_1">
    <interface:gps name="gps_iface_0"/>
  </controller:generic_gps>
</sensor:gps>
```

In order to create a realistic functionality, the GPS receiver should not work indoors. The distinction of indoors and outdoors could be added to the world file: polyhedrons can be used to define the 3D space of indoors. When the device's coordinates lay within these polyhedrons, the GPS service should not work. Another advantage of using this 3D description is, that GPS reception could also be enabled next to windows or building entrances by adding a margin for these regions.

Indoor localization

Localization is not only of interest outdoors, but also indoors, where the mobile computing devices spend most of their time. Unfortunately the satellite based navigation systems do not work reliably indoors, because of their low signal levels. Even if the satellites' signals can be received, the position error is often very large, due to multi-path propagation and other disturbing effects. For this reason, there are several other approaches available. Most of these do not use special sensors for this, but rather use data from already available sensors.

Torres-Solis et al. distinguish in their review of indoor localization technologies [36] between the technologies:

- Radio frequency waves: personal and local area networks, broadcast and wida area networks, Radio Frequency Identification (RFID) tags and radar.
- Photonic energy: visible (camera images) and invisible light (infrared (IR)/ultraviolet (UV) sensors).
- Sonic waves: commonly ultra-sonic.
- Mechanical energy: inertial (Inertial Measurement Unit (IMU)) or contact sensors.

- Atmospheric pressure.
- Magnetic fields.

The principal localization techniques can be roughly classified in these fields (according to [36]):

- **Lateration** (signal traveling time or path loss) and **Angulation** (multiple receivers/antennas).
- **Proximity**: exploit the limited range of signals.
- **Scene analysis** (monitor a wide area to detect subject of interest) and **Pattern matching** (Fingerprinting).
- **Dead reckoning**: calculate localization information based on previously-estimated data.

These distinctions are a good starting point for an ideation for (new) indoor positioning approaches. The relevant mentioned technologies and methods are explained in the following sensor descriptions. The

3.3.2 RF Signals and Network Connectivity

Following the ubiquitous connectivity paradigm, today's mobile devices are capable of numerous RF technologies. Almost every mobile device has a GSM/UMTS transceiver, a Wireless LAN transceiver, a Bluetooth transceiver and a FM radio receiver. Recently, the manufacturers began to implement Near Field Communication (NFC) chips into the devices. This development enables not only a deeper integration of the devices in the Internet of Things [37] but also the use of RF signals for the determination of other contextual information.

One popular field of application is the indoor localization. By measuring the received signal strength (RSS) or the time of arrival (TOF), the distance to a sender can be calculated. When the positions of the senders are known and enough reference points are available (at least three for the localization on a plain or four for 3D localization), the own position can be calculated by applying lateration methods. Instead of calculating the distance to the senders, fingerprinting makes use of pattern matching methods. During the initial data collection procedure, several measurements are performed in order to create a map with characteristic fingerprints, which consist of a list of all receivable transmitters with their respective hardware addresses and signal strengths [38]. The assumption, that similar fingerprints come from locations that are close, allows to compare the online fingerprints with the ones from the offline training phase and to estimate a location with the highest fingerprint similarity.

The cell ids and signal strengths from GSM can be used to determine the rough area the device is in. WLAN is today the most common used RF technology for indoor localization on mobile devices. An approach using in-house FM radio transmitters can be found in [39]. Ultra-Wide Band (UWB) is another RF technology that is often used for indoor localization [40], but is not (yet) implemented in today's mobile devices.

The signal strength can further be used to determine the actual connectivity and link speed for data services. By calculating, for example, the Signal-to-Noise Ratio (SNR) the receiver can determine the quality of the current connection and adapt the modulation scheme and thus the link speed.

Both applications are interesting for the simulation in the virtual environment. But the simulation of radiowave propagation is very complex and thus only a simplified model can be realized with reasonable expenditure. A sample implementation could look like this: the basic building blocks are clients and base stations. These functional blocks can be bound to objects in the simulation. In order to calculate the quality of the connection, the link budget is calculated in adjustable intervals for every client. The received power P_{RX} (dBm) can be calculated in the logarithmic domain with the Equation 3.3 [41], [42]. The indicated common values are valid for 2.4 GHz WLAN in Germany.

$$P_{RX} = P_{TX} + G_{TX} - L_{TX} - L_{FS} - L_{OB} - L_M + G_{RX} - L_{RX} \tag{3.3}$$

where,

- the transmitter output power P_{TX} (commonly 10 to 20 dBm), the transmitter antenna gain G_{TX} (commonly 6 to 9 dBi for omni-directional antennas) and the transmitter losses L_{TX} (commonly -3 to -10 dB, cable and connector losses) are directly set in the base station's model file.
- the free space loss L_{FS} follows Friis transmission equation [42] with the distance being the euclidean distance between the base station and the client:

$$L_{FS}\,(\text{dB}) = -27.55\text{ dB} + 20 \cdot \log\left(\text{frequency}\,(\text{MHz})\right) + 20 \cdot \log\left(\text{distance}\,(\text{m})\right) \tag{3.4}$$

- the obstacle (no line of sight) loss L_{OB} (dB), which is determined by performing a ray cast from the client to the base station summing up the distances traveled within (solid) objects. As a clue one can assume an attenuation of 110 dB/m, as it occurs for 2.4 GHz WLAN through solid walls [43].
- the miscellaneous losses L_M (in dB) through effects like fading or interference. This value should be calculated randomly, characterized by a reasonable probability density function (pdf).
- the receiver antenna gain G_{RX} (commonly 6 to 9 dBi for omni-directional antennas) and the receiver losses L_{RX} commonly -3 to -10 dB, cable and connector losses) are directly set in the client's model file.

This model considers the free space propagation and the basic loss through obstacles in the line of sight. With the random miscellaneous loss, small-scale fading effects are also covered. An omni radiation pattern and a symmetric channel is assumed in the simulation.

A hardware address can also be defined for the virtual base stations. This allows to perform fingerprinting and distance estimations to fixed targets like in real world scenarios. By defining a minimum receiver sensitivity (typically -80 to -95 dBm for IEEE 802.11b), it can be determined whether a link can be established or not. Via predefined receiver power levels, the link speed can be determined. This value can be used, for instance, to set the connection speed of the Android emulator.

3.3.3 Accelerometer and Gyroscope

An accelerometer is an electromechanical device that measures the proper acceleration of the device in terms of the force of gravity (g-force). Most modern accelerometer are micro electro-mechanical systems (MEMS), but also devices with piezoelectric, piezoresistive and capacitive components exist. Almost all mobile devices are equipped with a 3-axes accelerometer today. By measuring the the static acceleration due to gravity, the tilting of the device with respect to the earth can be found. The dynamic acceleration occurs during movements and can help be used to analyze the current movement. These acceleration values can for example be used for:

- Automatic screen alignment (switching between portrait and landscape mode) depending on the direction the device is held.
- Controlling applications and games through intuitive user input (for example as steering wheel for racing games).
- Automatic image rotation depending on the direction the device was held during taking the photo.
- Image stabilization by the included camera.
- User action determination and evaluation (for example, a pedometer for walking).
- Advanced Automatic Collision Notification (AACN). The mobile device automatically alerts emergency services when the accelerometer detects crash-strength acceleration values in a car.
- Inertial navigation systems [44].
- Supervising physical exercising [45].

Since an accelerometer can only determine the orientation relative to the earth's surface, modern mobile devices are additionally equipped with Gyroscopes. Gyroscopes are capable of measuring the rate of rotation around a particular axis. Most devices for consumer electronics are 3-axes micro electro-mechanical systems (MEMS).

The combination of the linear acceleration, measured by the accelerometer, with the rate of rotation, measured by the gyroscope, allows to sense the motion in all six degrees of freedom (DOF): x-, y- and z-axis, as well as, roll, pitch and yaw rotations. This is especially of interest for inertial navigation systems (INS), that continuously calculate the position, orientation and velocity via dead reckoning.

The realization of theses sensor devices can be done with already available Gazebo services. Calling the service /gazebo/get_model_state returns the current position and orientation of the model and its current linear and angular velocity.

Console output 3.3.1 Console output of a model state query.

```
# rosservice call /gazebo/get_model_state \
 '{model_name: mobile_device, relative_entity_name: world}'
pose:
  position:
    x: 3.32000003565
    y: -2.09999989332
    z: 0.000305630050682
  orientation:
    x: -4.50805085577e-05
    y: 3.18012224203e-05
    z: -2.1465383742e-07
    w: 0.999999998478
twist:
  linear:
    x: -1.73125044903e-07
    y: -9.10946262212e-08
    z: 8.16639331725e-06
  angular:
    x: 5.52560915098e-06
    y: -4.58489027467e-05
    z: -3.26748857349e-07
success: True
status_message: GetModelState: got properties
```

Console output 3.3.1 shows a sample output for a model state query. The *pose* is made up from the position and the orientation. The orientation is given as a unit quaternion [46]. *Twist* expresses the velocity in free space broken into its linear and angular parts. This service can also be used to get the relative position to another object by setting the relative_entity_name to the model name of the desired reference object. The acceleration can be calculated by dividing the velocity change through the elapsed time between two requests.

Another alternative is to use the existing RosIMU controller for Gazebo. An inertial measurement unit (IMU) is an electronic device that combines the data from accelerometers and gyroscopes to calculate the current velocity, orientation and gravitational forces. The same task is performed by RosIMU, which can be bound to any link in the simulation. An URDF usage example is given in Code snippet 3.3.2. It uses the same data like the Gazebo service introduced before. The major difference is, that Gaussian noise is added to the values and that they are published automatically with the given update rate to a specified ROS topic. The controller also performs the differentiation of the linear and angular velocities in order to display the current accelerations.

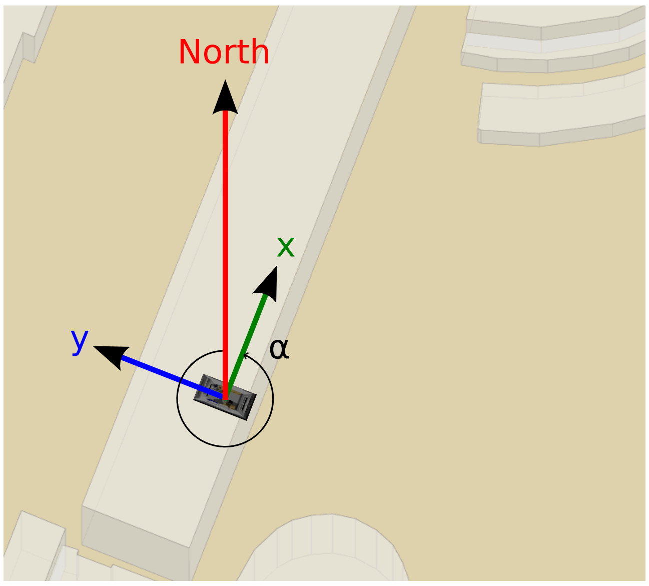
Code snippet 3.3.2 URDF description for the RosIMU controller.

```
<model:physical name="mobile_baselink">
  <controller:gazebo_ros_imu name="imu_controller"
   plugin="libgazebo_ros_imu.so">
    <alwaysOn>true</alwaysOn>
    <updateRate>1000.0</updateRate>
    <bodyName>mobile_baselink</bodyName>
    <topicName>/mobile_device/imu_sensor_data</topicName>
    <frameName>map</frameName>
    <!-- options to initialize imu for fake localization -->
    <xyzOffsets>25.65 25.65 0</xyzOffsets>
    <rpyOffsets>0 0 0</rpyOffsets>
    <interface:position name="imu_position_iface"/>
  </controller:gazebo_ros_imu>
</model:phyiscal>
```

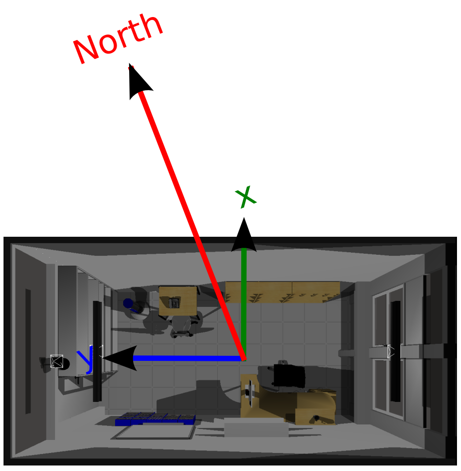
3.3.4 Magnetometer – Digital Compass

Dead reckoning is a good navigation method when no external references are available. But without initialization, this method is not applicable for most navigation use-cases. Besides an initial reference position, a reference orientation is also necessary in order to get the proper pose. A good choice for a reference orientation system is the magnetic field of the earth, which is stationary relative to the surface of the earth. In order to determine this field with the mobile device, a digital compass is necessary. The so called magnetometers, that are used for mobile devices, are made up by three solid-state magneto-resistive (MR) sensors [47]. The heading is determined by applying trigonometry rules.

The simulation of the magnetic field of the earth in the virtual environment would be very costly. For this reason, a strongly simplified model is proposed here. The basic assumptions of this model are, that the considered area is not very large (e.g. several km^2) and that it is far away from the magnetic poles (e.g. more than 1000 km). This allows to define a north direction for the whole area. A single determination off the north direction in one point of the simulated area is sufficient. Afterwards the angle α in the x-y-plane of the virtual environment between the x-axis (pointing in the positive x-direction) and the real north direction has to be defined. In order to avoid ambiguities, the angle should be given in counter-clockwise direction. This angle can be defined as a parameter for the simulation world and evaluated by the virtual compass sensor.



(a) Map view from *Google Maps* for "northing" the simulation world.



(b) View of the model in its inertial coordinate system extended with the graphical determined north direction.

Figure 3.5: North offset determination for Gazebo models.

The example in Figure 3.5 shows how the north direction can be determined by using available maps. The *Cognitive Office* is roughly positioned in a north up *Google Maps* view in Figure 3.5(a). Following the counterclockwise definition of the north offset angle, the angle $\alpha \approx 338.5°$ can be determined graphically. This angle can be used for the whole office in order to calculate the heading of the virtual compass device relative to the magnetic field of the earth. The orientation of the compass device can be determined as described in Section 3.3.3.

3.3.5 Altitude Sensor

The altitude or elevation is another important location information. In general it is defined as the vertical distance between a reference point and the point of interest. For most applications the altitude is defined as the height above sea level. This information is not only of interest for mountain climbers, but can also be used for indoor localization. While mountain climbers can rely on the altitude measured by the GPS device, it cannot be used indoors. In addition to that, the altitude accuracy for GPS is only around ± 23 m [48].

For that reason, pressure altimeters or baro-altimeters are commonly used for determining the altitude. The main component of this altimeter type is a barometer that measures the atmospheric pressure. With increasing altitude the air pressure decreases. Calibrated digital altimeters with temperature compensation have an accuracy from a fewdm to around 1 m. Due to this accurate values, these devices can be used for example for floor determination in indoor location scenarios.

The implementation for the virtual environment can again be strongly simplified. Instead of creating a virtual atmosphere, the data from the object's 3D pose can be used. For modeling the sensor variance a random step function capped by an absolute maximum variance value can be added to the sensor's measurement.

3.3.6 Touch Sensor and Common Buttons

Touch sensors for displays, also known as touchscreens, or switches have conquered the mobile device market in the last few years. For 2011 it is estimated, that mobile phones with touchscreens will have a market penetration of 38.2%. At latest in 2014 the market penetration is going to exceed 50% [49]. Especially the recently added multi-touch capabilities make touchscreens to intuitive input systems for mobile devices.

There are several different touchscreen technologies. But only the resistive and the projected capacitance technologies have a noteworthy market share today. Resistive touchscreen panels are basically made up from two layers that are coated with an electrically conductive material. One having a voltage gradient in x-direction, the other one in y-direction. Normally those two layers are separated by a narrow gap, but when pressure is applied to the panel, the two layers become connected and current can flow. The touch controller can determine the coordinates from the measured current. The projected capacitance touch (PCT) technology makes use of an electrical field, which is created by a conductive layer in front of the display. When an electrically conductive object touches the outer surface of the touchscreen, a capacitor is formed. By measuring the dynamic capacitor's capacitance on the corners of the panel, the controller can calculate the touch location.

Besides the touch input elements, there are normally also push buttons used for special actions and input, such as power on/off or the volume. Several mobile devices even offer a complete keyboard for entering text or controlling the device. These buttons are normally realized by sets of electrical contacts that are either unconnected or connected in the default state. Changes in the connection state or the connection state itself are detected by an external circuit which is connected to the different contacts.

A (multi-)touch sensor or a push button in Gazebo can be realized by evaluating the collision events of the sensor element. In order to get precise results, the collision geometry of the sensor should accurately match the visual geometry. Figure 3.6 shows the bounding box of the simplified mobile device model. The green colored boxes indicate, that the visual geometry corresponds to the collision geometry.

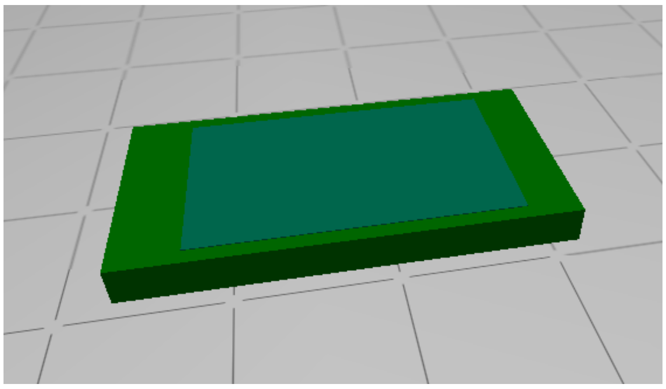


Figure 3.6: The bounding box of the simulated mobile phone is shown in green.

When two bodies touch in Gazebo, the ODE physics library creates a list of contact points, describing the points in the world where the bodies touch each other. Code snippet 3.3.3 shows the structure of a contact point. By evaluating the depth one can further create several pressure stages.

Code snippet 3.3.3 Collisions are described via contact points.

```
struct dContactGeom {
    dVector3 pos;     // contact position
    dVector3 normal; // normal vector
    dReal depth;      // penetration depth
    dGeomID g1,g2;    // the colliding geoms
}
```

3.3.7 Proximity Sensor

A proximity sensor can be used for contact-less object detection or for distance determination to objects in front of the sensor. This is reached by emitting an electro-magnetic (EM) field or radiation. A receiver, placed nearby the sender, evaluates the returned signals and can thereby provide information about the setting in front of the sensor. This sensor is, for example, built in the front side of the mobile phones in order to switch off the display when the user holds the telephone to his ear. But those sensor can also be used for user input [50].

A realization of this sensor for the simulation can reached by raytracing from a point on the surface of the mobile device in a specified direction. There are two different kinds of sensor devices that can be reached by this method. On the one hand, the sensor can simply output whether there is an object in front of the device or not. A maximum detection range could be specified in the sensor description. On the other hand, a distance measurement sensor can be realized by returning the distance to the nearest object detected during the raytracing.

3.3.8 NFC and RFID

With the recent addition of near field communication (NFC) chips to the *Google Nexus S*, the current Android developer phone, the technology has the chance to find its way into the global market. Until now, this technology has only been used extensively in Asia, where it is mainly enabling mobile payment.

The roots of NFC go back to Radio Frequency Identification (RFID). RFID uses radio frequency waves for the communication. One can mainly distinguish between active and passive RFID systems. While active systems have their own power source, the passive systems have to rely on the energy from their active interrogation partners in order to send their informations. Although this extremely limits the range, the advantage of small and cheap circuits for those passive tags is devastating. In contrast to the connectionless, mostly insecure implemented RFID technology, the main advantage of the NFC technology is the secure connection between two active devices. The pairing is reached by brining NFC enabled devices nearby to each other. The maximum distance is limited to a maximum of 10 cm, in order to avoid the abuse or the spying out of data.

Thanks to the energy-efficient and up to 424 kbit/s fast secure connections, NFC is suitable, for example, for the following applications:

- Mobile payment: The mobile device is used like a contact-less credit card. The personal identification number (PIN) can be entered directly on the device.
- Mobile ticketing: The ticket data or boarding pass information can be stored digitally signed on the device.
- Identification and authorization: ID cards and key cards features can be directly integrated into the devices.
- Data and information sharing: Exchange files or contact data by touching another mobile device or get product information by tapping passive NFC tags.

The passive NFC mode also enables to use some of these features, even when the phone is switched off or has dead batteries. Powered from the RF energy from the active part, the data written during runtime on the NFC chip can be read out.

For a realization in the virtual environment, the active and the passive mode are considered here. Since the range is limited in NFC respective RFID, it should be possible to set the maximum RF range for each device in the simulation. In addition to the limited range, the systems often exhibit a main radiation direction. This can, for example, be simulated by only allowing communication, when the partner is in front of the device. This direction could be implemented as a parameter for the virtual sensor, supporting the following directions: omni-directional, only ±x, ±x or ±z. Another reasonable parameter is the mode of operation: active or passive. This is needed to ensure, that at least one of the communication partners is in active mode. The communication can be realized by ROS topics. Each devices could have an *incoming* and an *outgoing* topic, that are automatically connected via some proxy node, when two devices are within their defined communication ranges. Another topic could be used for notification when a new communication partner was detected.

A possible NFC communication flow for exchanging contact details could look like this:

- The active NFC device **A** and the active NFC device **B** can see each other in their defined RF range and direction.
- Both devices send a message on their notification topic, e.g.
  - `/NFC_A/notification {name: B, type: active, status: connected}`
  - `/NFC_B/notification {name: A, type: active, status: connected}`
- A background node for the device **A** sends out the contact details simultaneously to its outgoing topic and to the partners incoming topic. This enables easy traceability and debugging:
  - `/NFC_A/outgoing {receiver: B, data: BEGIN:VCARD\n...\n END:VCARD}`
  - `/NFC_B/incoming {sender: A, data: BEGIN:VCARD\n...\nEND:VCARD}`
- The device **B** acknowledges the reception:
  - `/NFC_B/outgoing {receiver: A, data: OK}`
  - `/NFC_A/incoming {sender: B, data: OK}`
- When the devices are again outside their RF ranges a notification is again automatically sent:
  - `/NFC_A/notification {name: B, type: active, status: disconnected}`
  - `/NFC_B/notification {name: A, type: active, status: disconnected}`

3.3.9 Time Measurement

For many applications the time is an important factor. It is, for example, used to generate expiring user credentials, to trigger scheduled tasks or to stamp incoming messages or created files. Modern mobile operating systems offer several different solutions to synchronize the device's clock with reference clocks. The most popular clock synchronization solution for data connections is the Network Time Protocol (NTP) which is based on a client-server system. Mobile phones with CDMA/UMTS support can synchronize their clocks with the mobile network. Accurate time synchronization can further be reached via GPS.

Virtual clocks in Gazebo should run with simulation time which is for most simulations slower than the real time. The simulation time is published automatically to the `/clock` topic when the global parameter `/use_sim_time` is set to `true`. A sample output of the `/clock` topic can be seen in Console output 3.3.2. The clock controller for the different devices could use the messages from this topic as reference, but add a slight random drift value to it. The virtual clock should, in addition to publishing its current time, also allow to be set by an external instance. This could be realized via a ROS service.

Console output 3.3.2 Console output of the simulation clock topic.

```
# rostopic echo /clock
clock:
  secs: 9
  nsecs: 46000000
---
clock:
  secs: 9
  nsecs: 47000000
---
clock:
  secs: 9
  nsecs: 48000000
```

3.3.10 Battery Status

For mobile devices it is very important to keep an eye on the current battery status, since there is nothing more annoying than an empty battery when one needs to use the device. For that reason, there are some prediction algorithms that try to predict the battery life-time for the average usage profile [51]. In order to be able to develop battery-aware applications, the battery behavior should also be simulated for the virtual mobile device.

A battery can already be simulated in Gazebo using the *GazeboRosBattery* plugin. It was originally developed for the *PR2* robot, but can also be used for other scenarios. The battery plugin allows to set a full capacity, e.g. 1500 mAh for the *Google Nexus S*. A sample

URDF description can be found in Code snippet 3.3.4. The parameters are explained in Table 4.1.

Code snippet 3.3.4 URDF description for the RosPowerMonitor controller.

```
<controller:gazebo_ros_power_monitor
 name="gazebo_ros_power_monitor_controller"
 plugin="libgazebo_ros_power_monitor.so">
  <alwaysOn>true</alwaysOn>
  <updateRate>1.0</updateRate>
  <timeout>5</timeout>
  <interface:audio name="power_monitor_dummy_interface" />
  <robotNamespace>/mobile_device</robotNamespace>
  <powerStateTopic>power_state</powerStateTopic>
  <powerStateRate>10.0</powerStateRate>
  <fullChargeCapacity>1.400</fullChargeCapacity>
  <dischargeRate>0.100</dischargeRate>
  <chargeRate>5</chargeRate>
  <dischargeVoltage>4.95</dischargeVoltage>
  <chargeVoltage>5.05</chargeVoltage>
</controller:gazebo_ros_power_monitor>
```

Parameter	**Sample value**	**Description**
`robotNamespace`	`mobile_device`	The ROS namespace of the device the battery is simulated for.
`powerStateTopic`	`power_state`	The battery plugin publishes `pr2_msgs/PowerState` messages to this topic.
`powerStateRate`	10	Publishing rate for the power state messages.
`fullChargeCapacity`	1.400	Capacity of the fully charged battery in Ah.
`dischargeRate`	0.100	Power drawn from the battery during discharge mode in W.
`chargeRate`	5	Charging power in W.
`dischargeVoltage`	4.95	Voltage during discharge in V.
`chargeVoltage`	5.05	Voltage during charge in V.

Table 3.1: Important URDF parameters for the `gazebo_ros_power_monitor` plugin. The sample values should represent a common mobile phone.

This model assumes a static discharge rate which does not hold for most electronic devices. For example, when the display of a mobile device, which needs a lot of power, is switched on, the battery is drained much faster than when it is switched off. This effect could be modeled by measuring the power of the different components of a real mobile device. This can easily be done with freely available power monitors or a built-in power analyzer. A table could be used to store this values and to calculate the actual discharge rate depending on the currently used components.

3.3.11 Temperature, Air Pressure and Humidity

The temperature is one of the most frequently measured quantities in science. It influences many physical phenomena and chemical reactions. In daily life it is used, for example, to determine how warm should one dress or to control the heating and cooling. A thermometer in a mobile device can, for instance, be used to derive contextual information. The measured temperature can be a supportive indicator for location determination. During the summer and the winter periods, the outside temperature is often significantly different to the room temperature. The air pressure was already mentioned in the altimeter's description. The value, measured by a barometer, can further be used for the detection of opening and closing of doors or windows in sealed rooms. Hygrometers are used for measuring the humidity. Together with the air pressure and the temperature, it can, for example, be used for weather interpretation and forecast.

The values for these sensors cannot be derived directly in the virtual environment, since there is no atmospheric or weather model in the simulation. For this reason, an external unit is designated in the concept which was introduced in Chapter 2. This unit can be used to generate reasonable values automatically or to play back previously recorded temperature, air pressure and humidity data.

3.3.12 Ambient Light Sensor

Ambient light sensors are, for example, used to safe battery life or to enhance display visibility by adjusting the display brightness with respect to the current ambient light. These devices are typically realized by photo-transistors or photo-diodes, that detect the arriving light value. Using several such sensors together with different light sensitivity characteristics or putting filters in front of the light receivers allows to determine the current type of light the phone is exposed to. This could for example be used to determine whether the user is outside or inside. By measuring flicker frequencies, it could eventually also be used to detect when the user sits in front of a TV or another display.

The most realistic realization of a light sensor in Gazebo can be reached by modifying the camera plugin. A shader can be used to create a texture with the detected light intensity. The color can be analyzed in order to determine the color composition of the detected light. By assigning different ambient, diffuse or specular colors for the light sources, different light types can be modeled. For example, light with red parts could be used for sun light or light with blue parts could be used for the interior lightning.

3.3.13 Camera

Since several years the majority of mobile devices are equipped with at least one camera. Although the cameras are in most cases very simple components with fixed focus lenses and small image sensors, that limit the performance in poor light conditions, the low cost and the compactness have made them to indispensable parts of the mobile devices. Since the low cost is a key factor, the most image sensors are complementary metal-oxid-semiconductor (CMOS) active pixel sensors which are less expensive than the analog charged-coupled device (CCD) sensors. They consist of a matrix of photo sensors which are connected to circuitry that converts the received light energy to a voltage and afterwards to a composed digital image signal. In order to get a color image, a color separation mechanism is applied. Today's most common mechanism is the Bayer pattern. It is a color filter array that passes red, green and blue light only to selected pixel sensors. A schematic illustration is depicted in Figure 3.7. The missing color pixels are interpolated by software using a demosaicing algorithm.

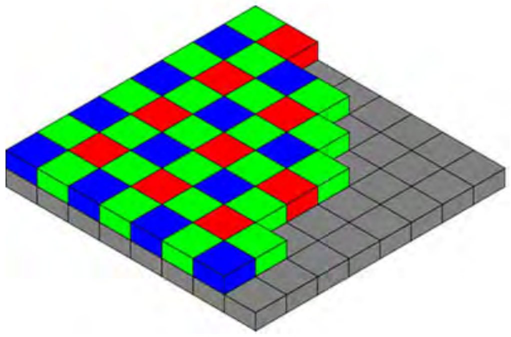


Figure 3.7: A color filter array (Bayer pattern) on a pixel sensor.

The camera in the mobile device is today not only used for capturing still images or videos. Augmented reality (AR) applications access the video stream from the camera and allow to overlay the real world images with artificial information. This can be used for task support [52], enriched navigation [53] or interactive sightseeing [54]. Front cameras enable, for example, video telephony with face-to-face communication.

Another type of camera, which is not yet built into any mobile device, is the infrared (IR) camera. It operates in wavelengths from $3500 - -14000$ nm, instead of the $450 - -750$ nm range of the visible light. This allows to detect the black body radiation of objects, which is directly related to their temperatures. IR cameras can be used for night vision, energy auditing of buildings and other insulated objects, firefighting operations or flame detectors. It can be also used for contact-less temperature measurement.

Some kind of virtual camera is already available in every 3D framework. They are primarily used to render the viewport for the user window. Most frameworks support more than one camera instance. This allows to have multiple independent virtual cameras with distinct configurations and perspectives. In the following the basic concept of a virtual camera is explained.

The basis of a virtual camera is the so called view frustum. It basically describes the volume of space in the virtual 3D world that is visible to the camera. This is the so called field of view of the virtual camera. A common frustum shape can be seen in Figure 3.8. It has the shape of a rectangular pyramid frustum. The camera position **C** lies at the apex of the notional completed pyramid. The near and the far frustum planes define the minimum and the maximum distances at which objects are visible to the camera. The far plane is often placed "infinitely" far away from the camera in order to get a realistic image in which all objects are drawn that are in the frustum regardless of their distance from the camera. The shown right-handed coordinate system defines the so called *camera space* or *eye space*. The camera lies at the origin of the camera space, the x-axis point to the right and the y-axis points upward. The pointing of the z-axis is defined differently depending on the 3D library. For OpenGL and thus OGRE 3D, it points to the opposite that in which the camera points.

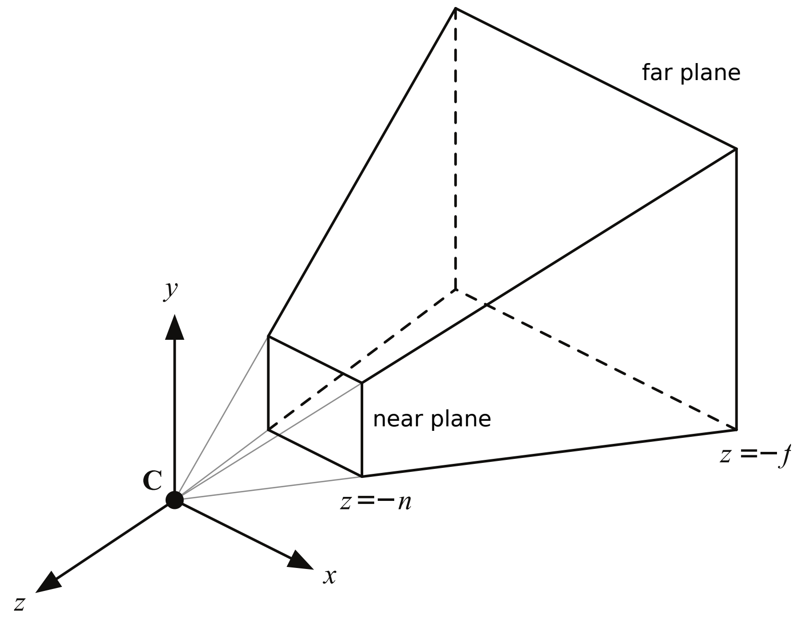


Figure 3.8: The view frustum is bounded by the near plane at distance n from the camera **C** and the far plane at distance f from the camera **C**. The top, bottom, left and right planes all pass through the camera position **C**. Source: [55]

Figure 3.9 shows the projection plane of the camera. It is perpendicular to the camera's viewing direction and intersects the left and right frustum planes at $x = -1$ and $x = 1$. The resulting distance e between the camera and the projection plane is called the focal length. The distance e depends on the so called horizontal field of view angle α formed between the left and the right frustum plane. The distance e can be calculated for a given

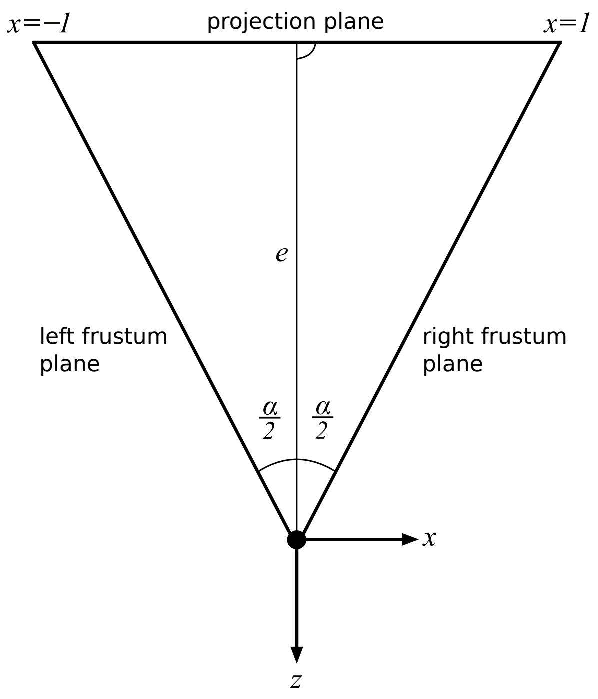


Figure 3.9: The horizontal field of view angle α formed by the angle between the left and the right frustum plane. Template: [55]

horizontal field of view angle α:

$$e = \frac{1}{\tan\left(\frac{\alpha}{2}\right)} \tag{3.5}$$

The calculation of the vertical field of view angle is often performed automatically by the 3D framework in order to get an undistorted image. In general it is not equal to the horizontal angle, since most displays are not square, but rectangular. For example, an image with a pixel resolution of 640×480 has an aspect ratio a of $a = \frac{\text{width}}{\text{height}} = \frac{640}{480} = 0.75$. Thus the projection plane intersects the bottom and the top frustum planes at $y = -a$ and $y = a$. The resulting triangle for the vertical field of view angle β is shown in Figure 3.11. It can be calculated in the following way:

$$\beta = 2\tan^{-1}\left(\frac{a}{e}\right) \tag{3.6}$$

A small field of view angle makes the camera “zoom in”, resulting in a telescopic view. A panorama view or “zoom out” can be reached by increasing the field of view angle.

The perspective projection onto the projection plane is reached by tracing the path from the camera through every pixel of the projection plane to the next intersection with an object in the view frustum. Basically the color and texture attributes of this point are evaluated and copied into the projection plane. A part of a sample scanline is shown in

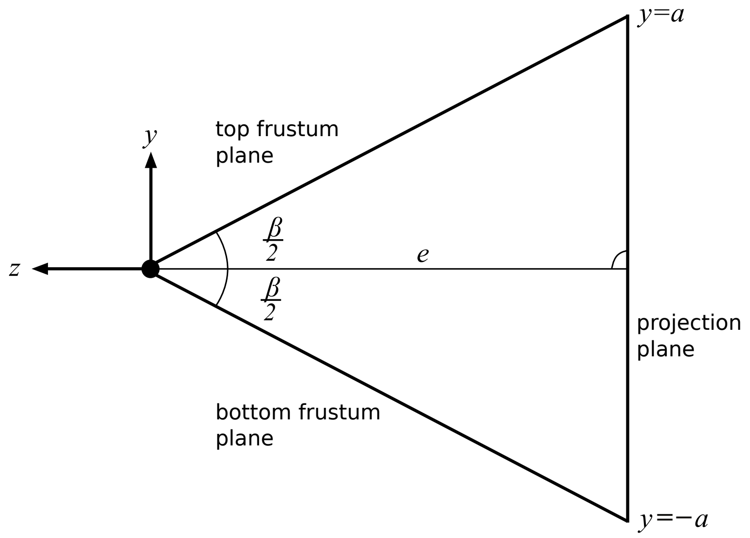


Figure 3.10: The vertical field of view angle α formed by the angle between the bottom and the top frustum plane. Template: [55]

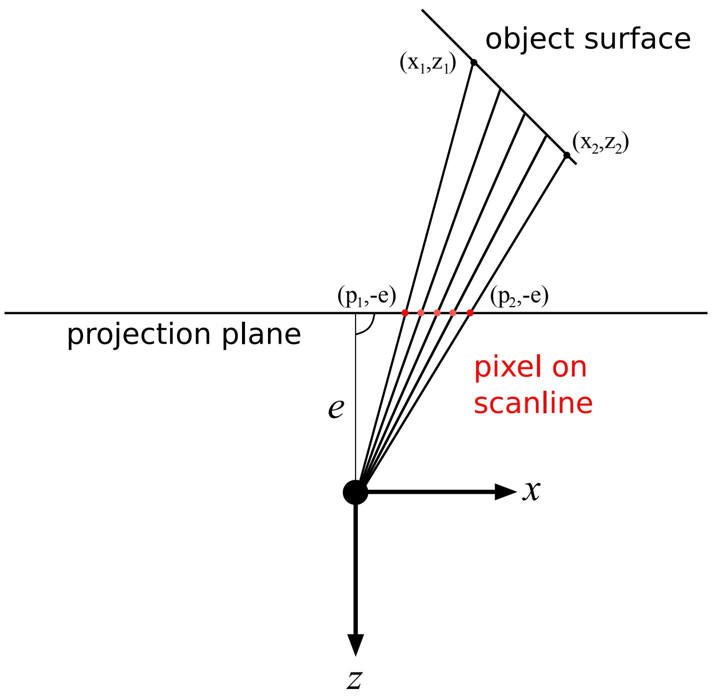


Figure 3.11: A sample scanline showing the ray casting for the projection. Template: [55]

Figure 3.11. The x-coordinate p on the projection plane corresponds to one point (x, y) in the view frustum. The relationship of the projection plane coordinates and the camera

coordinates of the eye space is expressed with the following equation:

$$\frac{p}{x} = \frac{-e}{z} \tag{3.7}$$

For the projection of the 3D scene to the 2D screen one gets the following equations, where x and y represent the pixel coordinates and **P** is the point which gets projected onto the screen:

$$x = -\frac{e}{P_z} P_x \qquad y = -\frac{e}{P_z} P_y \tag{3.8}$$

To obtain a correct perspective projection the view frustum is transformed to the so called homogeneous clip space. For calculating the respective lightning, shadows and reflections several interpolation methods are applied. A comprehensive description of the view frustum, projections and 3D computer graphic basics can be found in [55].

For getting a realistic image, the virtual lens system should exhibit a behavior similar to a real lens system. Some approaches for modeling effects like focus and distortion are described in [56] and [57].

3.3.14 Microphone

A microphone is a sensor that converts sound into an electrical signal. This is reached by detecting the pressure waves caused by the sound. In order to detect the vibrations, most microphones use either a moving coil which is suspended in a magnetic field (dynamic microphone), or a capacitor where one of the capacitor elements is a lightweight, conductive foil (condenser microphones).

Almost every mobile device has at least one microphone today. Besides their original intended use for audio communication, the microphone of a mobile device is today used for audio recording, for spectrum analyzing or for (very controversial) eavesdropping. Audio is also an important context provider. It can, for example, be used to detect the count of present people or to identify talking people by voice identification. By analyzing the ambient sounds, information about the current location and situation can be derived. For example, the ambient audio can be used to recognize ongoing human actions [58]. Equipping the device with more than one microphone further enables to locate the audio sources. There are even sound locating approaches that work with only one microphone [59].

Due to the large importance of audio, it should also be simulated in the virtual environment. Gazebo has a built-in audio interface. The free *Open Audio Library (OpenAL)* [60] is used for generating the three dimensional positional audio. The main elements of the library are *buffers*, *sources* and *listeners*. The *buffers* provide the audio signal for the *sources* which represent audio emitting points in the 3D space. The *listener* represents the 3D position where the audio is heard. The audio rendering is done independently for every *listener* in the scene. A listener corresponds to the microphone in the virtual scene.

The main *OpenAL* features are:

- 3D positioning with fully customizable distance attenuation.
- Different sound gains for every *source*.
- Multi-channel audio.
- Directional sound.
- Doppler effect: final output is dependent of distance, velocity and direction of the *listener*.

A sample Gazebo XML description for the audio controller is given in Code snippet 3.3.5. It shows an audio controller that plays the sound from the *test.mp3* file one time. In the current implementation, there is only a single listener in the simulation which is bound to the spectator's position. This has to be extended before several independent listeners can be modeled in the simulation. Additionally, there is not yet a ROS interface to the Gazebo audio controller, but by using the ROS interface for the camera as template, it should be implementable without excessive effort.

Code snippet 3.3.5 Gazebo XML description for the audio controller.

```
<controller:audio name="audio_controller_1">
  <pitch>1.0</pitch>
  <gain>1.0</gain>
  <loop>false</loop>
  <mp3>test.mp3</mp3>
  <interface:audio name="audio_iface_1"/>
</controller:audio>
```

3.3.15 Further internal and external sensors

There are, of course, uncountable other sensors available on the market that could also be interesting for mobile device. For instance, radiation or gas sensors could be implemented in mobile device and warn the user from hazardous substances or places.

Mood sensor

Another important context is the user's mood. There are several medical ways to measure the arousal of a human. A well understood technical method is to measure the galvanic

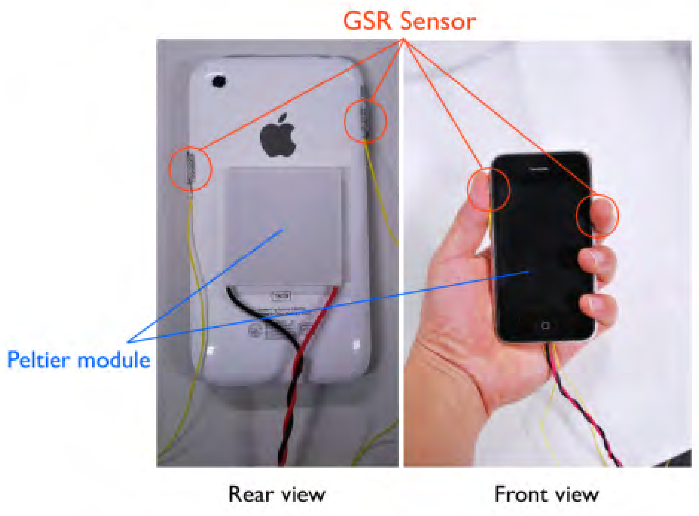


Figure 3.12: The *AffectPhone* measures the galvanic skin response (GSR) of the user for determining his emotional state. Source: [61]

skin response (GSR) or, to use the medically preferred term, the electrodermal response (EDR) [62]. The response can be obtained by attaching two electrons to the person's skin and metering its' resistance. This technique has proven to be very reliable [63] and is for example also used for lie detector. The theoretical background is the following: the more relaxed a person is, the dryer his skin will be. This will lead to a high resistance. Low resistance is metered when a person is under stress and thus his hand begins to sweat. The so called AffectPhone [61] uses this technique to detect the user's emotional state. Figure 3.12 shows the the AffectPhone with the attached electrons, that allow measuring the GSR.

This kind of sensor can not yet be directly implemented in the virtual device. But as soon as there is a human model available, that can handle the device, an interface can be determined that allows to read out the mood of the person and translates this to a corresponding resistance. Possible translatable states could be: nervous, happy, sad, caring, angry, relaxed, clearheaded, fatigued or activated [63].

Linking of External Sensors

There are a number of external sensors that can be connected with mobile phones. Common standards for their connection to the mobile devices are, for example, Bluetooth or ANT+. Examples for those sensors are headsets, heart rate or pulse monitors or fitness equipment data sensors.

Since the virtual device does not distinguish between internal and external sensors, these external sensors can also be connected to the device simulator via ROS topics. For modeling the external character it should be possible to define an out of reach distance for the simulation. Further the external sensors should be unavailable when the respective connection is switched off at the mobile phone. For example, a Bluetooth devices should only be able to send data to the mobile phone, when the device's Bluetooth adapter is active.

3.4 Virtual Output Devices

The output devices of a mobile device are mainly used to inform and entertain the user. In the following Subsections the most common output devices are specified.

3.4.1 LEDs, displays and projectors

Mobile devices are equipped with several visual output devices. Those are, for example, light emitting diodes (LEDs), displays and – in the future – also projectors.

The display is the main output and interaction element. The most common kind of screen used on mobile devices is the *thin film transistor liquid crystal display (TFT-LCD)*. In simplified terms, the LCD is working like a window blind for each pixel. The light from the external back-light can only pass the LCD when the liquid crystals in each pixel cell are arranged in a specific direction. The arrangement is achieved through an electric field which is controlled by the thin film transistor circuitry. The *active matrix organic light emitting diode (AMOLED)* is another screen technology which is used for mobile devices. Unlike LCD screens, AMOLED displays do not need a back-light, since the used organic light emitting diodes (OLDEs) can emit light by themselves. This allows to create thinner and lighter screens that offer a deep black level. For dark images with high gray levels an AMOLED display has an up to 60% lower power consumption than LCD [64]. Not necessary depending on the technology, the screens differ in many aspects like size, resolution, brightness, color brilliance and visibility angle.

In many cases *light emitting diodes (LEDs)* are embedded in the mobile device's body. Due to the large power consumption of the screen, they are often used for visual status indication and visual event notification. For example, notifying the user of an arrived e-mail by turning on the screen would massively drain the battery. Using a flashing LED instead saves much energy, but has the same effect, when the user knows the meaning of it. In order to offer several distinguishable notifications, multicolor LEDs are used. Embedded infrared LEDs allow to use a mobile device as a remote control unit for many commercial products like common multi-media equipment.

Another visual output device for mobile devices is the video projector. First cellphones with embedded pico projectors are already available on the market. Due to the big advances of the projector technology, the pico projectors reach images sizes and resolutions that are not only sufficient for sharing images and videos with friends, but also for giving on-the-go business presentations. The study in [?] revealed that a mobile phone with projector can enhance the working efficiency. The results in [?] shows that people want to use this kind of display possibility.

The simulation of screens in the virtual environment can be reached by manipulating the textures of objects. This allows to create a screen out of almost every object in the virtual environment. By converting the image, that should be displayed, to the specified resolution before it is passed to the texture buffer, the resolution of the display can be modeled. The self-illumination of a screen can be modeled by using lightning techniques. Further information about the implementation of a virtual display can be found in Section 4.3.

A projector can be realized by creating some kind of overlay texture which is applied onto the objects that are hit by the projection rays. An example of a projector for ROS/Gazebo is already available. The Gazebo plugin gazebo_ros_projector is able to project a filtered texture onto objects into the simulation. The calculation of the projected areas on the objects is done by applying a frustum model. Instead of the camera's view, the frustum describes in that case the projection volume. This allows to define the field of view (FOV) and the near and the far clip planes. A sample URDF description for the gazebo_ros_projector controller is depicted in Code snippet 3.4.1. This implementation can be used as basis for a virtual video projector. The main thing, that has to be added, is the possibility to feed the projector with images via a ROS topic.

Code snippet 3.4.1 Gazebo URDF description for the projector controller.

```
<controller:gazebo_ros_projector name="projector_controller"
 plugin="libgazebo_ros_projector.so">
  <alwaysOn>true</alwaysOn>
  <updateRate>15.0</updateRate>
  <textureName>projection_pattern_high_res_red.png</textureName>
  <bodyName>projector_base_link</bodyName>
  <filterTextureName>projection_pattern_filter.png</filterTextureName>
  <textureTopicName>projector_controller/image</textureTopicName>
  <projectorTopicName>projector_controller/projector</projectorTopicName>
  <fov>${M_PI/4}</fov>
  <nearClipDist>0.4</nearClipDist>
  <farClipDist>10</farClipDist>
</controller:gazebo_ros_projector>
```

3.4.2 Speaker

The speaker is another important output device. It is the counterpart of the microphone which is described in Section 3.3.14. Besides enabling to hear each other during audio communication, speakers are also used for informing the user of incoming calls, messages and other events and for giving back multimedia material.

The modeling of the audio system in Gazebo was already addressed by the audio sensor description in Section 3.3.14. A speaker corresponds to an audio source in this implementation. The user in front of the simulation window can hear the emitted sound from the current position of the spectator view in Gazebo.

3.4.3 Micro-sized Vibration Motors

Micro-sized vibration motors are integrated into many mobile devices. They are mainly used to inform the user silently or to give haptic feedback during the interaction with the device. For example, most mobile phones offer a vibration alarm for incoming calls and messages or calendar events. The tactile feedback, for instance, is used to inform the when a key on a touch keyboard was successfully pressed, since there is no tangible pressure point like on real keyboards.

For simulating the effect of the vibration motor, a little force can be applied on the simulated devices. In Gazebo this can be reached by using the available gazebo_ros_force plugin. This plugin allows to bind a force controller to a body. By publishing a geometry_msgs/Wrench to the defined wrench control topic, a force can be applied to body. For not accelerating the body in a single direction, one could choose a diagonal force direction that alternates every few milliseconds. A sample URDF description is depicted in Code snippet 3.4.2.

Code snippet 3.4.2 Gazebo URDF description for the gazebo_ros_force controller.

```
<controller:ros_force name="plug_force_controller" plugin="libros_force.so">
  <alwaysOn>true</alwaysOn>
  <updateRate>100.0</updateRate>
  <bodyName>mobile_base_link</bodyName>
  <topicName>/mobile_device/force</topicName>
  <interface:position name="mobile_device_force_iface"/>
</controller:ros_force>
```

3.4.4 Thermal displays

Another possibility to give feedback to the user are thermal displays [65], that can, for example, change the temperature of the device or at least some parts of it. This is especially of interest during calls, when a user holds the device in his hand. By cooling or heating the back of the device, the user can be informed of at least two different events. The sensitive palm can easily recognize these thermal changes. The AffectPhone, which was introduced in Section 3.3.15, is for example equipped with a Peltier module which can be heated or cooled. In that implementation, a user can feel changes in the emotional state of the person he is conversing. When the arousal of the other person increases, the device gets warmer. A decreasing arousal of the other user results in a cooler temperature. This technique allows to extend the verbal communication with a non-verbal component, which is communicated without interfering the user's sight or hearing.

Implementing this feature in a simulated device is not that simple. Assuming a simulated person is holding the device in his hand, a interface could be defined, that makes the person feel a temperature change on his palm. A method for visualizing the thermal unit's state is the use of different colors. When the unit would be heated, the back of the device could get red. Cooling could lead to a blue coloring.

3.4.5 Other output devices

There are, of course, many other output devices, that could be implemented in the virtual mobile device. An example for a somewhat unusual output device is the a Braille display which enables the use of the mobile device for visually impaired and blind people. Samsung's "Touch Messenger" and the conceptual cellphone B-Touch [66] are examples of mobile devices that are equipped with a Braille display.

Besides the described internal output devices, there are also external output devices. These can be realized similar to the external sensors described in Section 3.3.15. Example of external output devices are headsets

Chapter 4

Implementation of a Virtual Video System in Gazebo

In this Chapter, a virtual video system for the 3D simulator Gazebo is prototypically realized. The system supports the acquisition of an image stream from a virtual camera and the rendering of images to a virtual display. In Section 4.4 the implementation of two applications is described, that allow to feed images from external sources to the virtual displays.

4.1 Camera in Gazebo

Virtual cameras are already available in ROS/Gazebo through the gazebo_ros_camera plugin. It supports various parameters that can directly be set in the URDF description. Code snippet 4.1.1 shows an example URDF description. The size of the requested image is defined by editing the `<imageSize>width height</imageSize>` in the sensor section. An overview of the ROS specific parameters of the controller is given in Table 4.1.

Besides the raw image, which is published as sensor_msgs/Image, the virtual camera node also publishes the camera info (sensor_msgs/CameraInfo) as it is done by real camera devices. The camera info holds the image size and data about the distortion model with the corresponding distortion parameters. The given distortion parameters do not have any effect yet on the image itself, but an automatic distortion based on the parameters is planed to be added. In the most recent SVN version, the virtual camera sensor can also be used as a depth sensor. In this mode, the camera delivers an additional sensor_msgs/PointCloud message. The update rate (frames per second) and the horizontal field of view can be set via ROS topics that are explained at the end of this Section.

The gazebo_ros_camera plugin is realized using an `Ogre::Camera` instance which is created and basically configured using the Gazebo built-in wrapper `Gazebo::MonoCameraSensor`. The OGRE camera represents a viewport from which the scene is rendered. It is configured to use the perspective projection mode with full texture support. A frustum model, which is explained in Section 3.3.13, is used to calculate the visible area and the perspective projection. The horizontal field of view can be set via the `/camera/set_hfov` topic during runtime. The vertical field of view is calculated automatically based on the aspect ratio of the requested image size.

Code snippet 4.1.1 Sample URDF snippet for a virtual camera.

```
<gazebo reference="mobile_camera_unit">
  <sensor:camera name="camera_sensor">
    <imageSize>480 800</imageSize>
    <controller:gazebo_ros_camera name="controller-name"
     plugin="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <imageTopicName>/mobile_phone/image_raw</imageTopicName>
      <cameraInfoTopicName>/mobile_phone/camera_info</cameraInfoTopicName>
      <frameName>mobile_camera_unit</frameName>
    </controller:gazebo_ros_camera>
  </sensor:camera>
</gazebo>
```

The virtual camera sensor can be bound to any link of a model. It points by default from the origin of the link in the $+x$ direction. For the mobile device model, the camera link `mobile_camera_unit` is rotated by $-90°$ in pitch to get the camera pointing in $-z$ direction. The complete URDF description of the mobile device can be found in Appendix A. The model is part of the sim_camera package.

`/camera/image_raw` **ROS Publisher**

The raw camera image from the virtual device.
Message: `sensor_msgs/Image`

`/camera/camera_info` **ROS Publisher**

The camera's calibration data and information about the image (size, encoding, ...). This is published synchronously with every image frame.
Message: `sensor_msgs/CameraInfo`

`/camera/points` **ROS Publisher**

Depth data seen by the camera as point cloud. Only available in the most recent SVN version.
Message: `sensor_msgs/PointCloud`

Parameter	**Sample value**	**Description**
`robotNamespace`	`mobile_device`	The ROS namespace of the device the camera is simulated for.
`imageTopicName`	`image_raw`	The ROS topic where the image messages are published.
`cameraInfoTopicName`	`camera_info`	The ROS topic where the camera info messages are published.
`pointCloudTopicName`	`points`	The ROS topic where a point cloud (depth data) is published. Only available in newest SVN version.
`frameName`	`mobile_camera_unit`	The name of the link to which the camera is bound.
`Cx`	`0`	Principal point x-coordinate. Set to 0 for automatic calculation.
`Cy`	`0`	Principal point y-coordinate. Set to 0 for automatic calculation.
`focal_length`	`0`	Focal length. Set to 0 for automatic calculation. focal_length = image_width (px) / (2 tan hvof (radian)).
`distortion_k1`	`0`	First coefficient of radial distortion. No effect yet.
`distortion_k2`	`0`	Second coefficient of radial distortion. No effect yet.
`distortion_k3`	`0`	Third coefficient of radial distortion. No effect yet.
`distortion_t1`	`0`	First coefficient of tangential distortion. No effect yet.
`distortion_t2`	`0`	Second coefficient of tangential distortion. No effect yet.

Table 4.1: URDF parameters for the `gazebo_ros_camera` plugin. The sample values represent a common mobile phone.

`/camera/set_hfov` **ROS Subscriber**

Set the horizontal field of view in radian. The vertical field of view is calculated automatically based on the aspect ratio of the requested image size. Values of 90° *and more result in a wide-angle view. Low values (e.g.* 30°*) result in a telescopic kind of view. Default values are* 45° *to* 60°*.*
Message: `std_msgs/Float64`

`/camera/set_update_rate` **ROS Subscriber**

Can be used to set the frame rate of the camera.
Message: `std_msgs/Float64`

4.2 Usage Example for the Virtual Camera

Another realized implementation example for virtual camera is the `flycam` which is also part of the `sim_camera` ROS package in the `simvid` stack. It can be used to navigate freely through a Gazebo world using a 3D joystick. For controlling the camera, the so called camera_move node was created. It allows intuitive control of Gazebo objects. Table 4.2 shows the available parameters.

Parameter	Default value	Description
`joy_topic`	`/spacenav/joy`	The ROS topic where the joystick messages (joy/Joy) are read from.
`model_name`	`flycam`	The name of the Gazebo model which should be controlled.
`angular_scaler`	`100.0`	The angular scaler allows to scale the speed for rotations relative to the incoming joystick values.
`linear_scaler`	`100.0`	The linear scaler allows to scale the speed for translations relative to the incoming joystick values.

Table 4.2: URDF parameters for the `camera_move` node.

When the node is started, it subscribes to the specified joystick topic `joy_topic`. These messages are automatically generated by the respective joystick nodes. Whenever a joystick message with values unequal to zero arrive, the current model pose of the specified model `model_name` is requested from Gazebo via the /gazebo/get_model_state service. For an intuitive, first person like view, the camera object should be moved relative to its current position and orientation and not in world coordinates. This is reached by performing a coordinate transformation using *Bullet Physics Library* [67] functions in the following way:

1. The received joystick translation values are scaled by the linear scaler. The angular values are given in Euler coordinates and are also directly scaled by the angular scaler.
2. The translation and the rotation are combined to the *btTransform* `movement`.
3. The current world pose from the /gazebo/get_model_state service is transformed to the *btTransform* `pose_in`. The rotation of the pose is given as a quaternion and can directly be used.
4. The new world pose *btTransform* `pose_out` after the movement is calculated: `pose_out = pose_in * movement`.

The result of the calculation is used to set the new pose of the object. It can be set via the Gazebo service /gazebo/get_model_state. Due to the fast publishing rate of the joystick and the small movement values it is not visible to the eye, that the model pose is jumping. The camera_move node can be used with every Gazebo object. Since the pose is updated every few milliseconds, the controlled objects can fly in the air or move through wall even if they have turned on gravity or collision boxes.

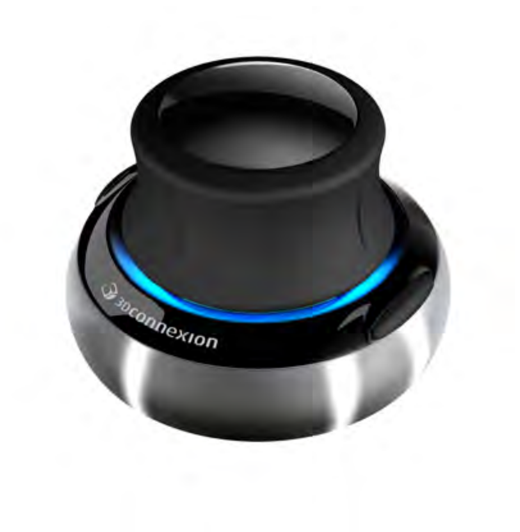


Figure 4.1: The *3Dconnexion SpaceNavigator* offers six degrees of freedom. Source: [68]

The `flycam` was mainly tested with the 3D mouse *3Dconnexion SpaceNavigator* [68] which is depicted in Figure 4.1. It allows the control of all six degrees of freedom. The available ROS package `spacenav_node` allows to use it like a normal joystick in ROS. Before it can be used the `spacenavd`, which is part of the `spacenav` ROS package, has to be started. This can, for example, be done in the following way:

```
sudo `rospack find spacenav`/spacenav_svn/spacenavd/spacenavd
```

4.3 Video Texture in Gazebo

As mentioned in Section 3.4.1, virtual displays can be realized by dynamic textures. The implementation of such a dynamic texture for Gazebo is described in this Section. The realization was done by creating the Gazebo plugin `gazebo_video_plugin`. It is part of the `textvid` stack.

Before the functionality of this plugin is described in detail, a short examination of the texture handling in ROS/Gazebo respective OGRE is reasonable. For coloring or texturing a mesh in Gazebo, so called materials are used. These materials are defined in material scripts which can reside inside the `Media/materials/scripts` folder in any ROS package. Gazebo automatically loads the defined materials at startup, when the path is exported in the `manifest.xml` of the package. A sample export entry is shown in Code snippet 4.3.1.

Code snippet 4.3.1 This export entry in the `manifest.xml` makes Gazebo load all material scripts that reside in the `Media/materials/scripts` subfolder.

```
<export>
  <gazebo gazebo_media_path="${prefix}" />
</export>
```

The default file extension of a material script is `.material`. In a material script several *materials* can be defined. Each *material* is made up of at least one *technique* which describes the visual effects for the rendering of the material. A *technique* is usually made up of one or more *passes*. A *pass* describes one complete render procedure of the material. By defining several *passes*, composite effects can be reached. In a *pass* the color, lightning effects and texture units can be defined. The image files specified for the textures are searched automatically in the `Media/materials/textures` folder. A sample script file is given in Code snippet 4.3.2. An overview of the parameters can be found in the OGRE Manual [69].

Code snippet 4.3.2 A sample OGRE material script. Source: [69]

```
// This is a comment
material walls/funkywall1
{
  // first, preferred technique
  technique
  {
    // first pass
    pass
    {
      ambient 0.5 0.5 0.5
      diffuse 1.0 1.0 1.0

      // Texture unit 0
      texture_unit
      {
        texture wibbly.jpg
        scroll_anim 0.1 0.0
        wave_xform scale sine 0.0 0.7 0.0 1.0
      }
      // Texture unit 1 (this is a multitexture pass)
      texture_unit
      {
        texture wobbly.png
        rotate_anim 0.25
        colour_op add
      }
    }
  }

  // Second technique, can be used as a fallback or LOD level
  technique
  {
    // .. and so on
  }
}
```

A defined *material* can be assigned to a Gazebo object by adding the *¡material¿* tag.

For that reason, the realization of the virtual display is based on material manipulations. Whenever a `gazebo_ros_video` controller is loaded, a new material with a unique random name is created and assigned to the body that will be the virtual display. In the current implementation, the dynamic material is assigned to all visual faces of the body. That means, that the complete surface of the object is covered by the same virtual display. Since most displays are only visible from one side, this is no problem. As soon as the material is created a single technique with a single pass is added to it. After this step, the dynamic texture has to be created. Similar to the material, it gets a random unique name. It is configured to use the BGRA pixel format and the TU_DYNAMIC_WRITE_ONLY_DISCARDABLE option is set in order to reserve a buffer that is optimized for dynamic, frequently updated textures. After the initialization of the texture it is added to the previously created pass of the material. To enhance the visibility of the material, the lightning is disabled in the pass. This leads to brighter images on the virtual screen in dark environments.

Code snippet 4.3.3 Sample URDF snippet for a virtual display.

```
<gazebo>
  <controller:gazebo_ros_video name="gazebo_ros_video_controller"
   plugin="libgazebo_ros_video.so">
    <bodyName>mobile_display</bodyName>
    <imageTopic>/mobile_device/image_in</imageTopic>
    <imageLoad>/mobile_device/load_file</imageLoad>
  </controller:gazebo_ros_video>
</gazebo>
```

Code snippet 4.3.3 is a sample URDF description for the implemented virtual display. It can be configured by the following three parameters:

bodyName is used to define the name of the link to which the dynamic texture will be bound to. Only links of the model, for which the virtual display is defined, are considered. This allows to have the same link name in different models. It is a mandatory parameter and the plugin will thrown an error when the specified link name cannot be found for the model.

imageTopic can be used to define the topic name of the video texture's "video in". The video texture will subscribe to this image topic and display the images that are published on this topic. The default value for the `imageTopic` parameter is `/monitor/image_in`.

imageLoad specifies the topic of the video texture's image file loading interface. When a `std_msgs/String` message with a path to an existing image file is published to this topic, the video texture tries to load and display the local image file. The default value of this parameter is `/monitor/load_file`.

The details of the mentioned topics are explained at the end of this Section.

The video texture can either display an image which is received on the subscribed ROS image topic `imageTopic`, or it can load and display a local image file which is triggered by publishing the path to the image file as a std_msgs/String ROS message to the specified `imageLoad` topic. Both capabilities can be used at the same time. The last received image or image loaded from the given path is displayed until a new image or file path is received. In order to load the image to the texture buffer, its data is converted to an `Ogre::Image` by using an appropriate `Ogre::PixelFormat`. The valid pixel formats for received ROS image messages are given in Table 4.3. It also shows the appropriate Ogre pixel format for loading the data to an Ogre image. For copying the image data from the incoming ROS image message to the Ogre image that is written to the texture buffer, it is required to set a global lock. The lock prevents that the buffer of the Ogre image is accessed by multiple threads. It further ensures that only one output image can be set at once. When a path is received on the `imageLoad` topic, it is checked whether the file exists on the local system or not. When the file exists and the file type is supported by Ogre, it can directly be loaded to an Ogre::Image. The image handling of Ogre 3D is realized with the *FreeImage* library [70]. It supports all common file formats like for example PNG, BMP, GIF, JPEG, TIFF and many other.

ROS image topic pixel format	**Ogre pixel format**
RGBA8	Ogre::PF_BYTE_RGB
BGRA8 (OpenCV types: 8UC4, 8SC4)	Ogre::PF_BYTE_RGBA
BGR8 (OpenCV types: 8UC3, 8SC3)	Ogre::PF_BYTE_BGRA
MONO8, Bayer (OpenCV types: 8UC1, 8SC1)	Ogre::PF_BYTE_L

Table 4.3: Supported pixel formats for ROS image messages.

The created Ogre image can not be written directly to the texture buffer of the dynamic texture. This is due to the different contexts in which the threads are running. The processing of incoming ROS messages is done in a thread which is running parallel to the Ogre rendering thread. For this reason, a frame listener is attached to the Ogre rendering pipeline during the plugin's initialization. The frame listener is implemented in the class `GazeboRosVideo::DynamicTextureHandler` by overriding the methods of the `Ogre::FrameListener` class. Table 4.4 shows the frame listener functions that are called during the rendering of a frame.

ROS image topic pixel format	**Ogre pixel format**
`bool frameStarted(const FrameEvent &evt)`	Called when a frame is about to begin rendering.
`bool frameRenderingQueued(const FrameEvent &evt)`	Called after all render targets have had their rendering commands issued, but before render windows have been asked to flip their buffers over.
`bool frameEnded(const FrameEvent &evt)`	Called just after a frame has been rendered.

Table 4.4: Methods of an `Ogre::FrameListener`. Source: [69]

For loading the image data from the Ogre image into the texture buffer, the `frameStarted` function was chosen. This allows to display a newly received image already in the next rendered frame. For copying the image data to the texture, the global lock has to be set again so that the Ogre image cannot be altered during the copying. The image can be loaded into the texture buffer by calling the `void loadImage(const Image &img)` function of the dynamic texture object. The new image is automatically scaled to fit the texture's size. The pixel format of the texture is also automatically adjusted to fit the Ogre image's pixel format. This procedure only takes place when new image data was set since the last replacement. During the initialization a blue color is set for the dynamic texture which is kept until the first image data is set.

`/mobile_device/load_image_file` **ROS Subscriber**

Path to an image file on the node PC that should be loaded to the video texture. The image is automatically scaled to fit the texture's size. All major image formats are supported. For a complete list, see `http://freeimage.sourceforge.net/features.html`
Message: `std_msgs/string`

`/mobile_device/image_in` **ROS Subscriber**

This is the "video in" of the video texture. Every image, that is published to this topic is displayed by the video texture. The image is automatically scaled to fit the texture's size. It supports the formats RGB8, RGBA8, BGRA8, BGR8 and MONO8.
Message: `sensor_msgs/Image`

4.4 Feeds for Video Texture

To feed the video texture from different source two image format converters have been created:

- Remote Frame Buffer (RFB) node for converting VNC images to a ROS image topic.
- PointCloud2 to ROS image topic converter – `kinect4skype`.

4.4.1 Remote Frame Buffer node – `vnc_image_client`

The `vnc_image_client` package allows to connect ROS to a VNC server. It currently supports publishing the image data of the remote frame buffer to a ROS image topic. This can, for instance, be used to display the content of a real computer monitor to a virtual monitor in the simulation. In Chapter 5 it is used to display the content of the Android emulator's display on the simulated mobile device. The node was realized in Python and is mainly based on the *rospy* package. Although there is also a pure Python implementation for a VNC viewer available [71], the ROS node was built using *ctypes* Python bindings to the original *LibVNCClient* [72].

The *ctypes* wrapper for *LibVNCClient*, which was automatically generated using the tool RPythonic, allows to call the functions of the C library directly from Python. The library itself had also to be patched, in order to support a callback function for newly arrived RFB data and for passing the VNC server's password directly as a parameter to the library. By using different threads for the RFB data acquisition and the ROS image publishing, the node can work very efficient. A global lock ensures, that the RFB data can be copied safely in the ROS image publishing thread. The image is published in the RGBA8 format.

The configuration of the `vnc_image_client` node is realized through private ROS parameters which are described in Table 4.5

Parameter	**Default value**	**Description**
`~image_topic`	`/vnc_image`	The name of the topic to which the VNC image is published.
`~host`	`localhost`	IP address or hostname of the VNC server.
`~port`	`5901`	Port on which the VNC server is running on.
`~password`		Password for the VNC server.
`~publish_rate`	15	Maximum frames per second (fps) limit for the publisher.

Table 4.5: ROS parameters for the `vnc_image_client` node.

When the VNC server is password-protected and no password was given as parameter, the password can be entered during the connection attempt. This allows to keep the password private.

`/vnc_image` ROS **Publisher**

The image acquired from the Remote Frame Buffer (RFB). It has the same dimensions like the RFB image from the VNC server and is published in the RGBA8 format.
Message: `sensor_msgs/Image`

4.4.2 `kinect4skype` – a PointCloud2 to Image Topic Converter

The Point Cloud Library (PCL) [73], which is part of the `perception_pcl` stack, is an important ROS component – at least since the arrival of the Microsoft Kinect sensor. The `pcl_tutorials` package, which comes with the `perception_pcl_addons` stack, contains several visualization tools, tutorials and demos. One of these demos is the pointcloud_online_viewer node. This node is an online rendering tool for PointCloud2 messages. It is based on the Visualization ToolKit VTK [74].

For being able to share the rendered point cloud image via *Skype* and other video conferencing tools, the rendered view has to be exported to a ROS image topic which can be published to the video loopback device by the `image_sink`. In order to achieve this, the C++ class `PCLVisualizer`, which is used for visualizing the point cloud, was inherited by the self created `PCLVisualizerImage`. By overloading the render function `spinOnce()`, the exporting to a ROS image topic could be realized without altering the original library. The conversion of the render window to a ROS image message is straight forward:

1. A `vtkWindowToImageFilter` instance is used to convert the render window content to a VTK image. The input of the filter is set to the render window.
2. A `vtkImageExport` instance takes over the conversion of the VTK image to a standard RGB8 image format. The output of the `vtkWindowToImageFilter` has to be set as the input of the image exporter.
3. In order to publish the image, an OpenCV image is created from the exported image data.
4. The OpenCV image is converted to the ROS image message data by using the `cv_bridge`.
5. And finally the image is published to the specified topic with the help of an `image_transport` publisher.

The pcl2image node is part of the newly created kinect4skype stack. Figure 4.2 is showing the render window. The image_view in Figure 4.3 displays the ROS image topic published by the pcl2image node. In Figure 4.7 the output of the `pcl2image` node is rendered to the monitor screen in the virtual *Cognitive Office*.

For a future version of this package, a Skype plugin shall be created that allows to directly send the point cloud information to another client. A possible realization could be based on the recently released PCL compression algorithms [75].

4.5 Video Loopback Device

Another aspect of the video toolchain was to create a method for using ROS image messages outside the ROS. For feeding the video data to other video processing applications, the best method is to create a local video loopback device that can be accessed and controlled via a virtual device node /dev/video*. This can be reached by creating a Linux kernel module which uses the Video4Linux2 (V4L2) API. Video4Linux was created in order to provide unified access to various video capturing devices, such as TV tuners or USB web cameras [76]. The standard Linux kernel module `vivi` (Virtual Video driver) shows a sample implementation of a virtual video loopback device that is based on V4L2. It is intended to be a working example code for current V4L2 drivers to facilitate the implementation of new

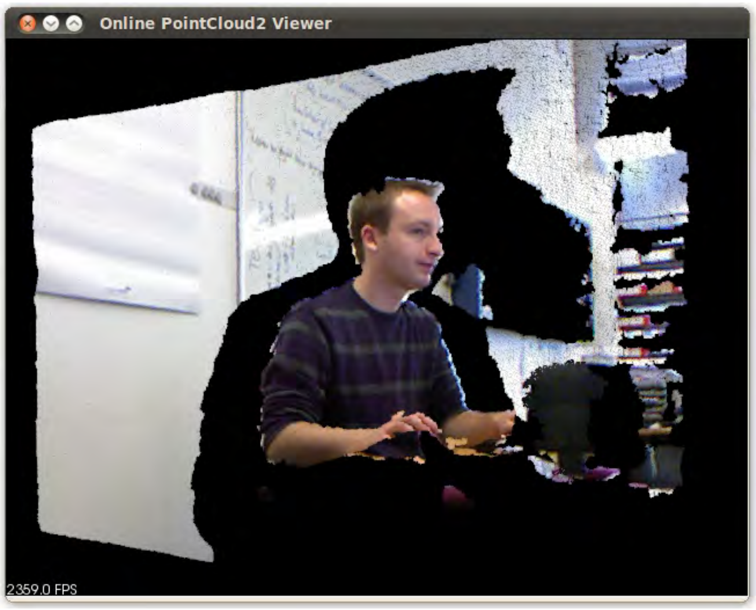


Figure 4.2: The render and control window of the pcl2image. It is derived from the pointcloud_online_viewer node which comes with the `perception_pcl_addons` stack.

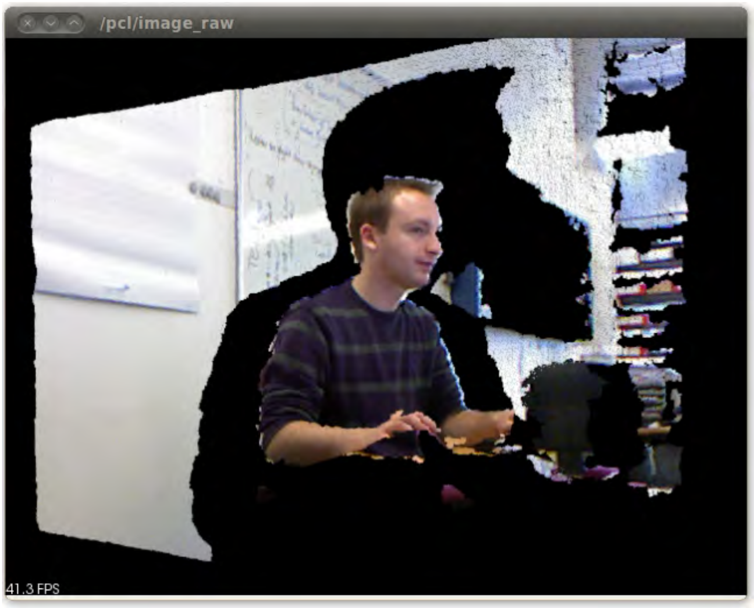


Figure 4.3: An image_view instance is showing the output of the `pcl2image` node.

video drivers. Its current implementation acts like a real device but without interfacing any physical hardware and only displays some color bars with a time counter.

An extended version of the `vivi` module is the `v4l2loopback` module. The most recent development version can be obtained from [77]. It allows to create one or more virtual

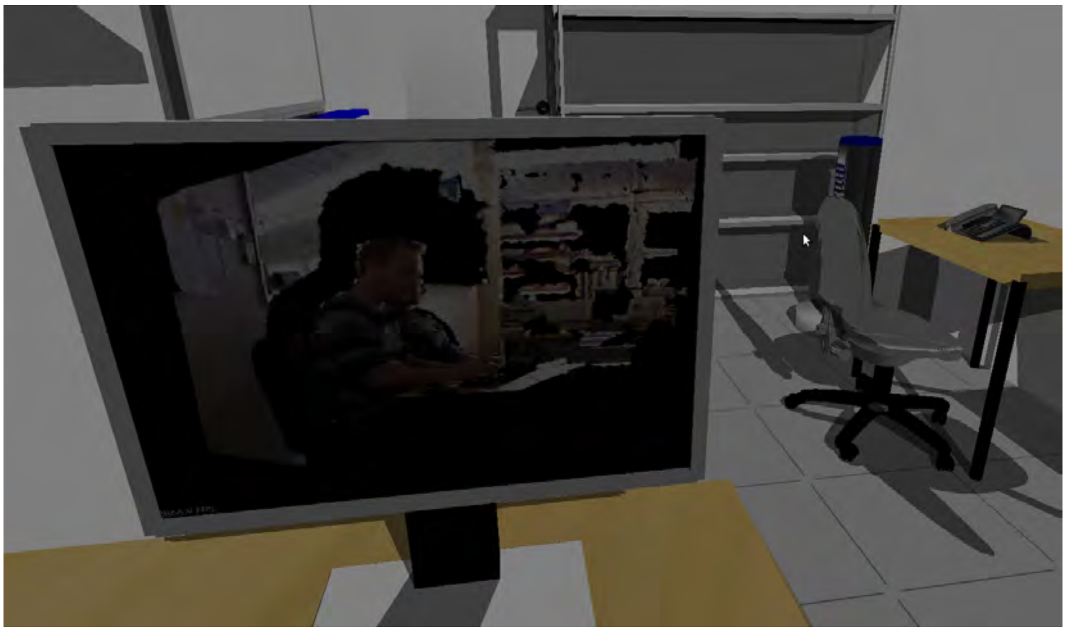
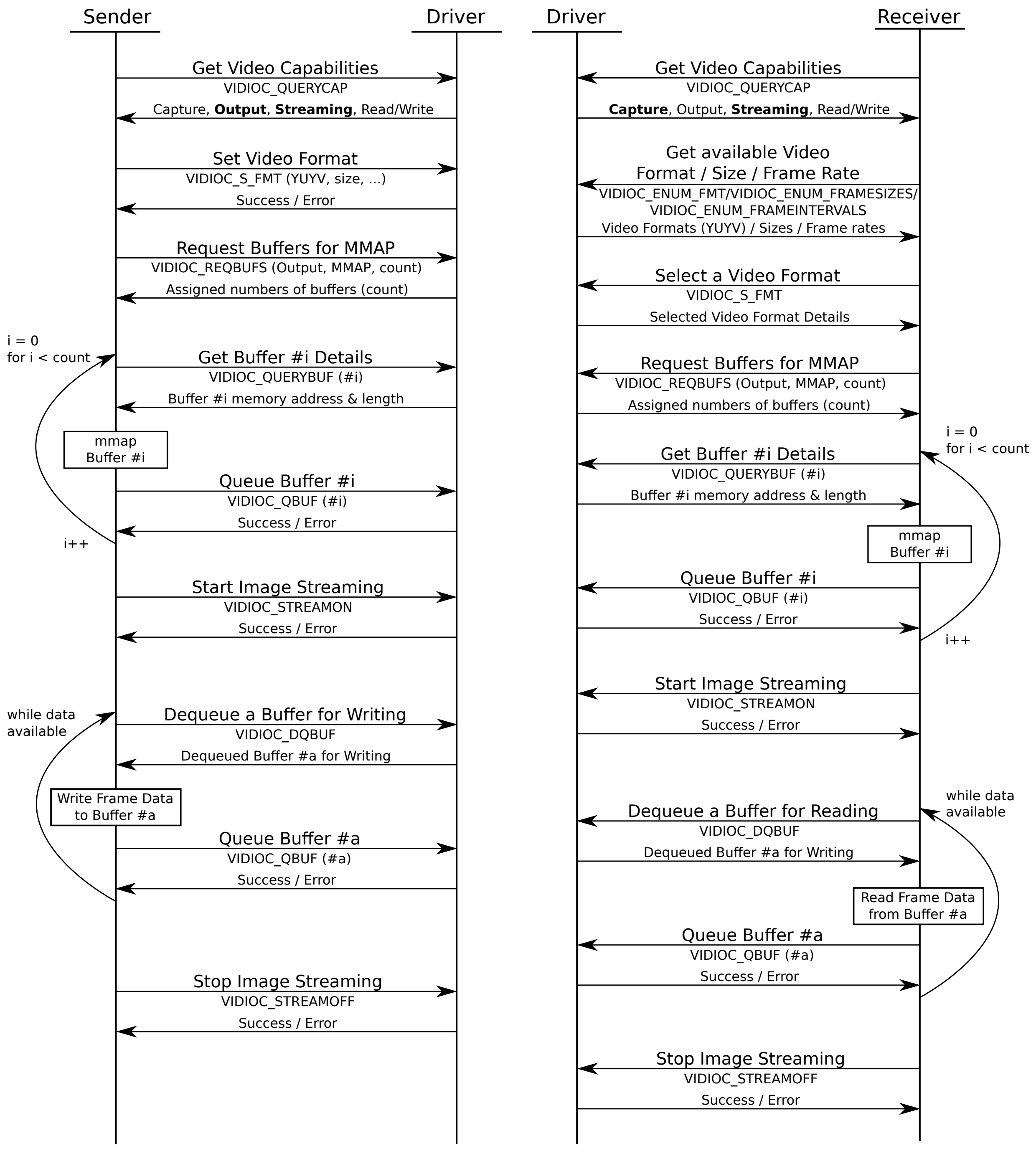


Figure 4.4: The image topic output of the `pcl2image` is rendered to the video texture on the virtual monitor of the *Cognitive Office*.

video character devices that can be written to and read from. It is mainly intended for the use with GStreamer's `v4l2sink` element as writer. It is working with many different V4L2 compatible readers, but it is for example not working with *Skype*. During several tests it revealed that a logical error in the memory-mapped (mmap) implementation leads to a memory access in an undefined memory region. An error in the implementation of the `vidioc_enum_framesizes` function further caused the reader application to get the wrong framesize. In addition the `vidioc_enum_frameintervals` was missing. This function is called by the reader for determining the valid frame rate for the available framesizes. In Figure 4.5 the message flows between the driver and the writer and between the driver and the reader are given. A general overview of *V4L2* driver programming can be found in [76].

For the aimed use case, it is sufficient that the driver just outputs the image data that was written to the video loopback device. For that reason, the driver only support one framesize and one framerate, that correspond to the data from the writer. For meeting the requirements, the `vidioc_enum_framesizes` was changed to output the image size of the incoming image data. Additionally the `vidioc_enum_frameintervals` function was created, in order to output the configured frame rate. The frame rate can be set by the image writer during the initialization.

An usage example for this function is for example a tele operation system via *Skype*.



(a) Image sender and driver.

(b) Driver and image receiver.

Figure 4.5: Simplified V4L2 message diagram.

4.6 Video Loopback Device Writer

In order to publish a ROS image stream to the adapted video loopback device, a *v4l2* compatible writer had to be created. For this reason, the image_sink node was implemented. It is part of the simvid stack. It was written from scratch and uses the memory mapping (mmap). In general, it reads images from a given image topic, converts them to the specified image size and to the YUYV pixel format and writes them to specified video device. A parameter overview is given in Table 4.6. The memory mapping is explained graphically in Figure 4.6.

Parameter	**Default value**	**Description**
`device`	`/dev/video0`	The video loopback device to which the data is written.
`topic`	`/camera/image_raw`	The image topic from which the images are read from.
`width`	`640`	The image width to which the input images are converted before output.
`height`	`480`	The image height to which the input images are converted before output.
`fps`	`15`	The expected and defined frame rate.

Table 4.6: ROS parameters for the `image_sink` node.

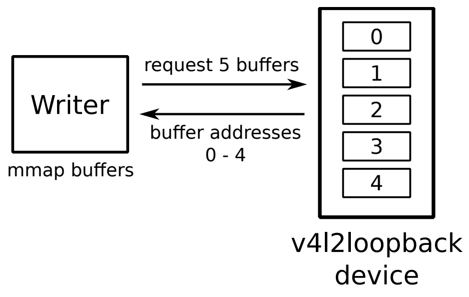
For publishing the image data the YUV2 (YUYV) format was chosen, because it is supported by almost all video readers. For getting the YUV2 data out of the RGB images, the following scheme is used:

```
y0 u0 y1 v0 from r0 g0 b0 r1 g1 b1
y2 u1 y3 v1 from r2 g2 b2 r3 g3 b3
```

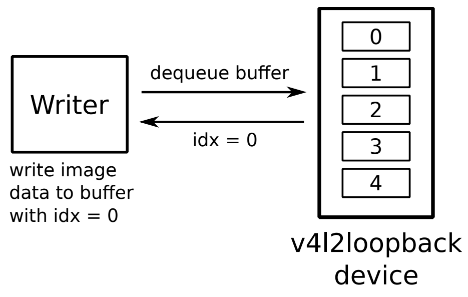
Code snippet 4.6.1 shows the formulas for the converting of RGB to YUV2. For the final implementation the conversion was changed from a float based solution to an integer solution. The new conversion code is depicted in Code snippet 4.6.2. This optimization leads to an about 35% faster color conversion. The values are clipped to the range 0 to 255 after the conversion.

Code snippet 4.6.1 RGB to YUYV conversion with float values.

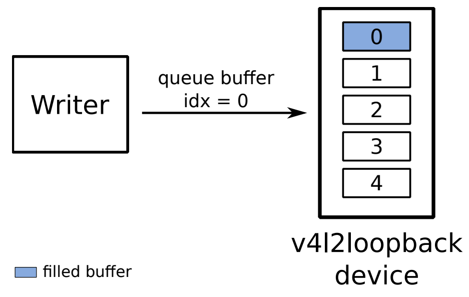
```
y0 =  (0.257 * r0) + (0.504 * g0) + (0.098 * b0) + 16;
u  = -(0.148 * r0) - (0.291 * g0) + (0.439 * b0) + 128;
v  =  (0.439 * r0) - (0.368 * g0) - (0.071 * b0) + 128;
y1 =  (0.257 * r1) + (0.504 * g1) + (0.098 * b1) + 16;
```



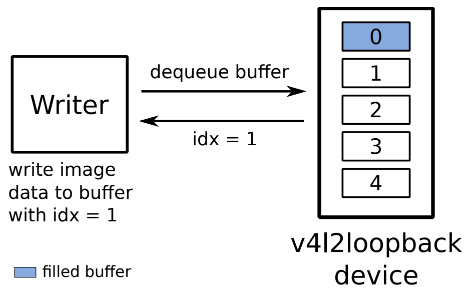
(a) Image writer requests buffers and mmap*s* them.



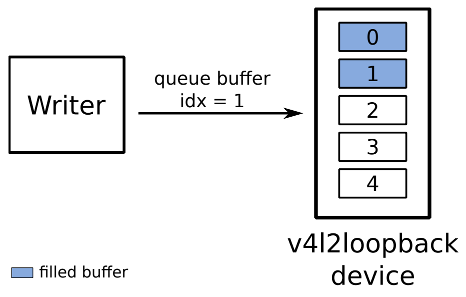
(b) Before writing the image data, a buffer has to be dequeued. This corresponds to locking the buffer.



(c) When the image data is written, the buffer is queued again and can be accessed by the readers.



(d) v4l2loopback returns the next buffer's index. When the last requested buffer index is reached it begins from the top again.



(e) The image writer always writes to the returned index, as this is locked by the v4l2loopback device.

Figure 4.6: Buffer handling of the v4l2loopback device, seen from the writer's perspective.

Code snippet 4.6.2 RGB to YUYV conversion with integer values.

```
y0 = (( 263 * r0 + 516 * g0 + 100 * b0) >> 10) + 16;
u  = ((-151 * r0 - 298 * g0 + 450 * b0) >> 10) + 128;
v  = (( 450 * r0 - 377 * g0 -  73 * b0) >> 10) + 128;
y1 = (( 263 * r0 + 516 * g0 + 100 * b0) >> 10) + 16;
```

`/camera/image_raw` ROS **Subscriber**

The images from this topic are written to the configured video device.
Message: `sensor_msgs/Image`

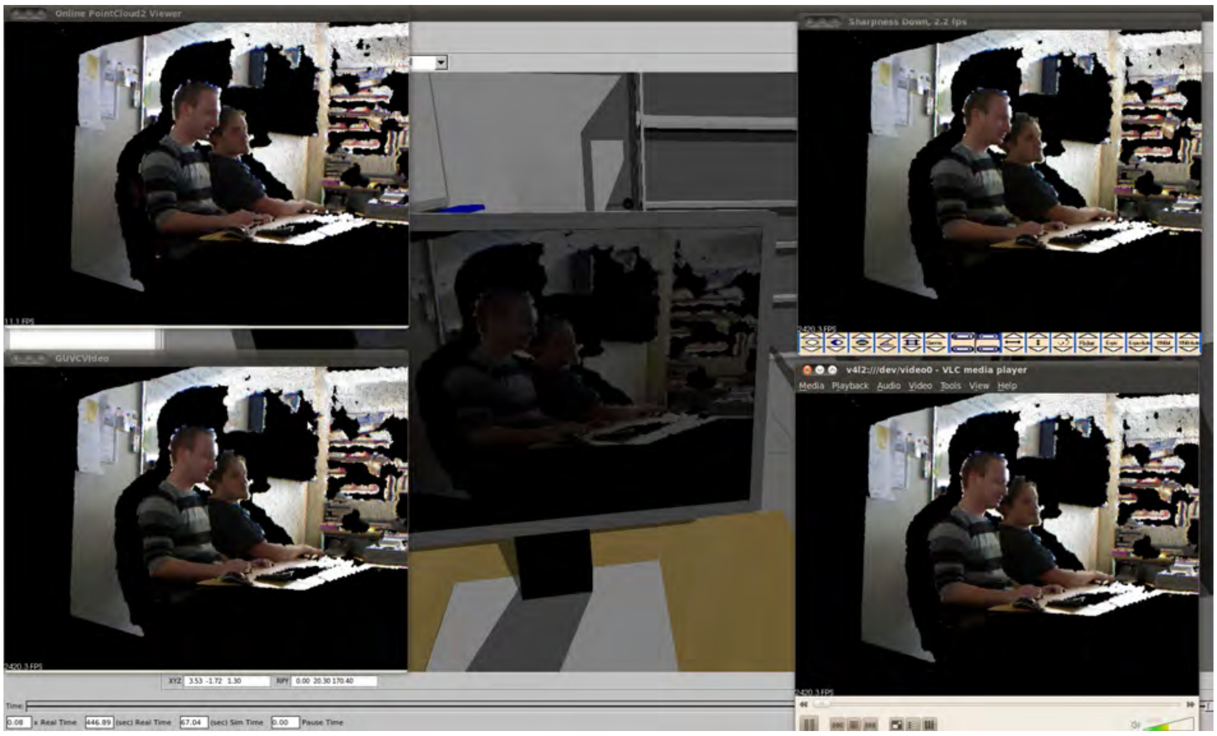

Online PointCloud2 Viewer
Sharpness Down, 2.2 fps
GUVCVideo
v4l2:///dev/video0 - VLC media player
Media Playback Audio Video Tools View Help

Figure 4.7: The image topic output of the `kinect4skype` rendered to video texture and simultaneously exported to a video loopback device, which is opened by various media players.

4.7 A Sample Video Processing Toolchain

On the basis of the implemented video toolchain one can simulate almost every kind of application, that needs data from a camera and/or produces graphical output for the user. For the evaluation, a simple barcode detection was implemented. The detection was realized with the existing ROS package zbar_barcode_reader_node [78]. The barcode detection node is based on the open-source software suite *zbar* [79] which can read barcodes from different sources, such as video streams or image files. It supports the most common barcode types including EAN-13, EAN-8, Code 128 and QR Code.

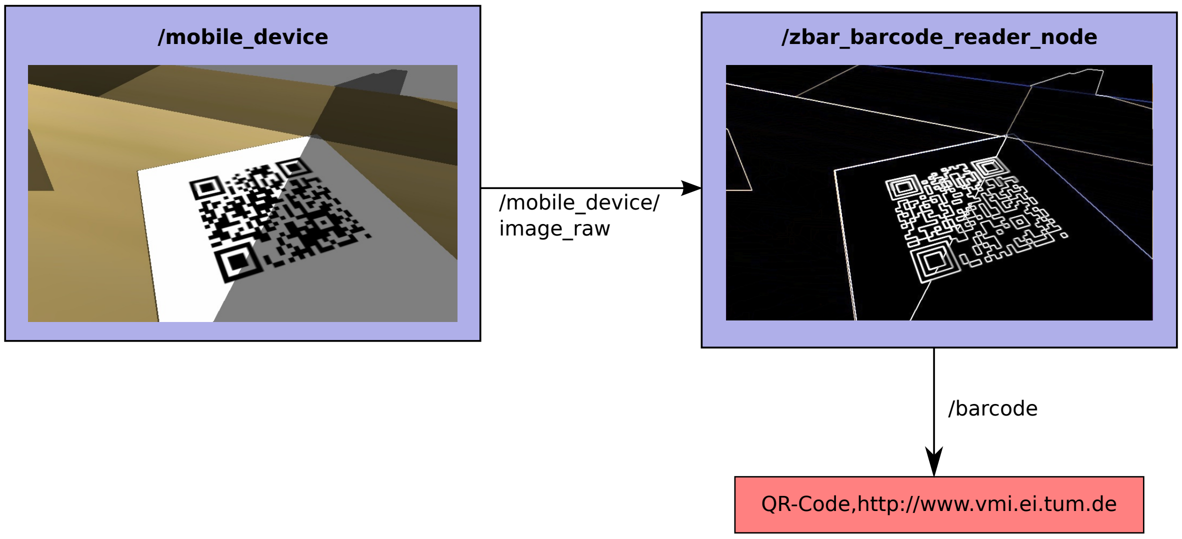

/mobile_device
/zbar_barcode_reader_node
/mobile_device/ image_raw
/barcode
QR-Code,http://www.vmi.ei.tum.de

Figure 4.8: The structure of the barcode detection sample.

Figure 4.8 shows the structure of the barcode detection sample. The image from the virtual camera is processed by the `zbar_barcode_reader_node`. When the barcode reader node detects a barcode in the image, it publishes a message with the barcode type and the barcode data to the /barcode topic.

The Gazebo model of the *Cognitive Office* was chosen as test environment for the implemented sample. A white “piece of paper” showing a QR Code, which holds the URL `http://vmi.lmt.ei.tum.de`, is lying on the writing desk. Figure 4.9 shows the mobile device filming the QR code on the desk. Simultaneously the scan result is published to the `/barcode` topic. The output is depicted in Console output 4.7.1.

Console output 4.7.1 Sample URDF snippet for a virtual camera.

```
# rostopic echo /barcode
data: QR-Code,http://www.vmi.ei.tum.de
---
```

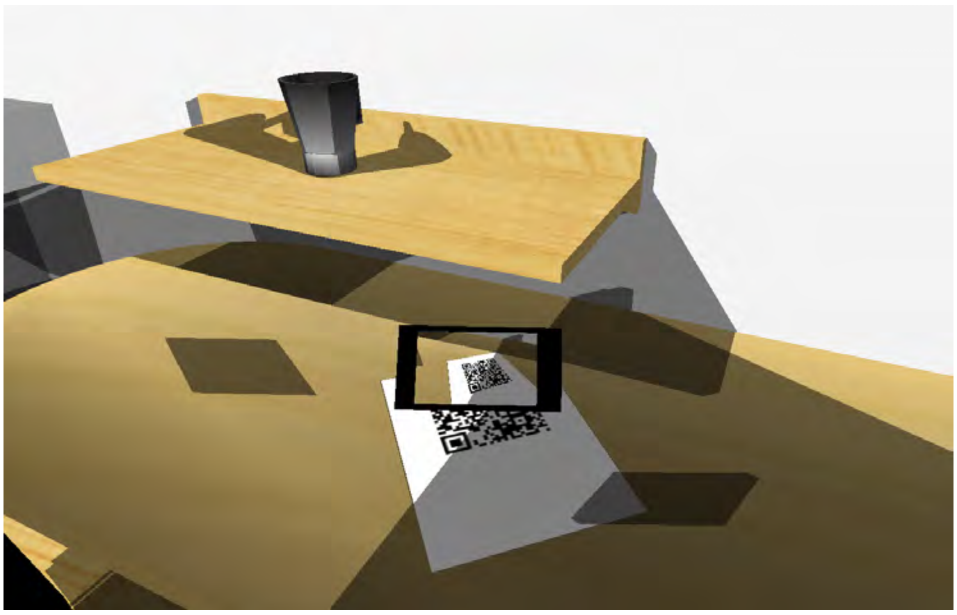


Figure 4.9: The mobile device scans the QR code.

Chapter 5

Integration of Android Emulator in the Video Toolchain

In order to simulate mobile applications, that can also run on real devices without modifications, the Android emulator was integrated in the video toolchain. Several modification had to be done for "seamless" integration of the emulator system. These modifications are described in the following sections.

5.1 Adding a ROS Client to Android Camera Service

The major modification had to be done to the camera service of the Android emulator. At the moment, only a *fake camera* is supported by the emulator which generates a simple checkerboard image with a floating square on it for the camera preview and a static jpeg image for the capture function.

For that reason, a closer look to the Android source was necessary. Already during the familiarization with the code, a bug was identified that prevented the Camera preview from being shown in the emulator for Android 2.3 and newer. The bug was already known and reported first on 12 December 2010 [80].

Created `tiny_roscpp`. A small cpp class, that supports the basic ROS API in order to subscribe to an image topic.
Depends only in cURL and Berkley sockets.
Features: Creates a "webserver", performs XML-RPCs, negotiates ROS message format and handles tcp stream for image data reception.

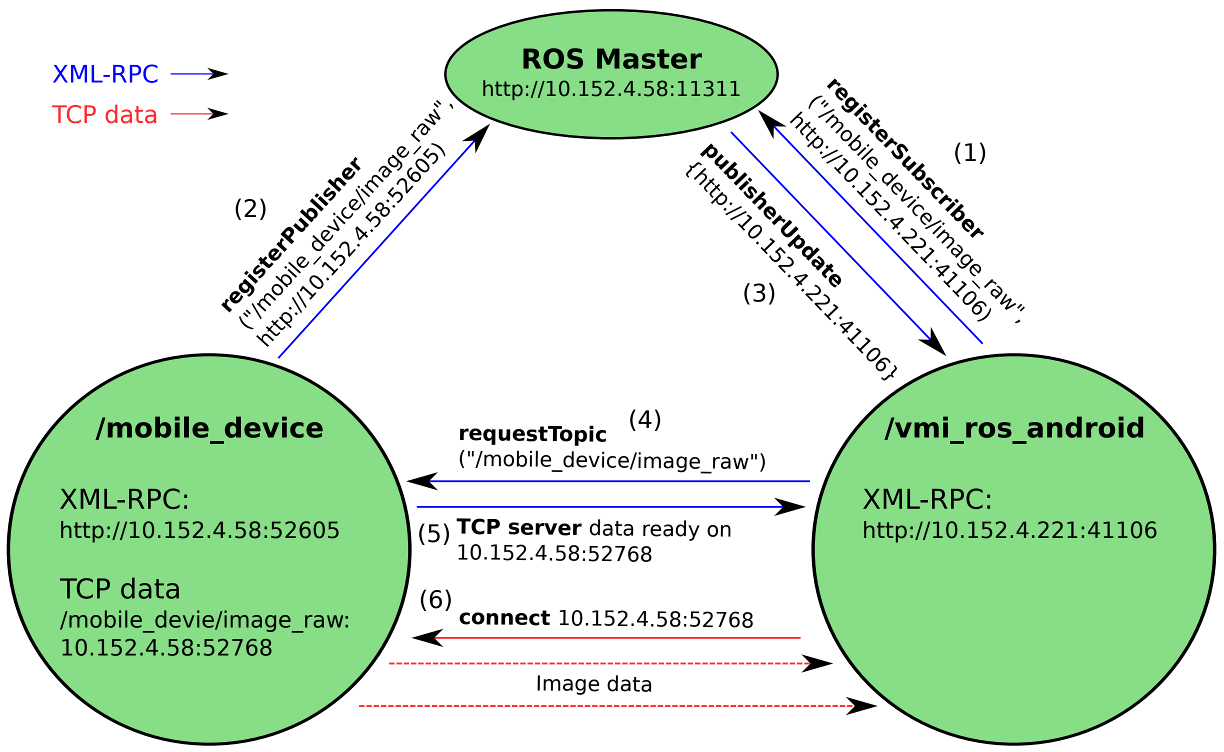


Figure 5.1: ROS API messages exchange for subscribing to a topic.

Code snippet 5.1.1 Register subscriber with Master.

```
<?xml version="1.0"?>
<methodCall>
  <methodName>registerSubscriber</methodName>
  <params>
    <param><value>/vmi_ros_android</value></param>
    <param><value>/mobile_device/image_raw</value></param>
    <param><value>sensor_msgs/Image</value></param>
    <param><value>http://10.152.4.221:41106/</value></param>
  </params>
</methodCall>
```

Code snippet 5.1.2 Response from Master. Already one publisher available at *http://10.152.4.58:52605/*.

```
<?xml version='1.0'?>
<methodResponse>
  <params>
    <param>
      <value><array><data>
        <value><int>1</int></value>
        <value><string>
          Subscribed to [/mobile_device/image_raw]
        </string></value>
        <value><array><data>
          <value><string>http://10.152.4.58:52605/</string></value>
        </data></array></value>
      </data></array></value>
    </param>
  </params>
</methodResponse>
```

Code snippet 5.1.3 Master announces a publisher update to subscriber node. Array contains all active publishers. If last publisher stops sending an empty list is sent.

```
<?xml version='1.0'?>
<methodCall>
  <methodName>publisherUpdate</methodName>
  <params>
    <param>
      <value><string>/master</string></value>
    </param>
    <param>
      <value><string>/webcam/image_raw</string></value>
    </param>
    <param>
      <value><array><data>
        <value><string>http://10.152.4.58:52605/</string></value>
      </data></array></value>
    </param>
  </params>
</methodCall>
```

Code snippet 5.1.4 Request the topic from publisher node *http://10.152.4.58:52605/*.

```
<?xml version="1.0"?>
<methodCall>
  <methodName>requestTopic</methodName>
  <params>
    <param><value><string>/vmi_ros_android</string></value></param>
    <param><value><string>/mobile_device/image_raw</string></value></param>
    <param>
      <value><array><data>
        <value><array><data>
          <value><string>TCPROS</string></value>
        </data></array></value>
      </data></array></value>
    </param>
  </params>
</methodCall>
```

Code snippet 5.1.5 Answer from publisher node *http://10.152.4.58:52605/*. A TCP data channel is open on *10.152.4.58:52768*.

```
<?xml version='1.0'?>
<methodResponse>
  <params>
    <param>
      <value><array><data>
        <value><int>1</int></value>
        <value><string>ready on 10.152.4.58:52768</string></value>
        <value><array><data>
          <value><string>TCPROS</string></value>
          <value><string>10.152.4.58</string></value>
          <value><int>52768</int></value>
        </data></array></value>
      </data></array></value>
    </param>
  </params>
</methodResponse>
```

Before the data transmission can start, the message type has to be matched. This is realized by exchanging data about the message format file. The obligatory fields are the topic name, the message type, the MD5 sum of the message file and the caller id, which corresponds to the ROS node name. The other options can be found on `http://www.ros.org/wiki/ROS/Connection%20Header`.

Code snippet 5.1.6 Exchange message format and routing information with *10.152.4.58:52768*.

```
topic=/mobile_device/image_raw
type=sensor_msgs/Image
md5sum=060021388200f6f0f447d0fcd9c64743
callerid=/vmi_ros_android
```

The publisher node will send his message format file afterwards. When both formats match, which is ensured through the MD5 sum, the publisher starts sending TCP packets with the data for the defined format. The data is encoded in the following format:

```
4-byte length + [4-byte field length + field data]*
```

The length fields are little-endian integers. The values are encoded similar to the C++ representation of `http://www.ros.org/wiki/msg#Fields`.

The decoding of a `uint16` and a `string` field can be realized like this (with intermediate step, to make it more understandable):

Code snippet 5.1.7 Decoding example in C++.

```
unsigned char * dataptr = message;
uint32_t temp;
uint16_t image_height;

/* Decode uint16 */
memcpy((void *)&temp, dataptr, 4);
dataptr += 4;
// decoded uint16 field
image_height = (uint16_t)temp;
printf("Image Height: %d\n", image_height);

/* Decode string */
// temp is here the length of the string
memcpy((void *)&temp, dataptr, 4);
dataptr += 4;
// allocate memory for the string
unsigned char * encoding = (unsigned char *) malloc((temp + 1)
                           * sizeof(char));
// add string terminator
encoding[size] = 0x00;
// copy the string to encoding
memcpy(encoding, ptr, temp);
dataptr += temp;
printf("Image Encoding: %s\n", encoding);
```

Implemented in the Android mediaserver binary.

5.2 Further Modifications to the Android Emulator

Before the ROS binding could be used, another limitation of the Android emulator had to be bypassed. Due to a the virtual network, in which the Android emulator is running, no direct network connection can be established by default. Although it is possible to set up port forwarding for single ports, complicated network address translation (NAT) rules would be necessary to get the ROS communication working. For this reason, it was decided to establish a virtual private network (VPN) connection from the Android system directly to the institute's network.

In order to get the OpenVPN Android client running, the kernel of the Android emulator system has to be modified.

Compiled a new kernel for the emulator, that has module support. Compiled `tun` module to enable VPN connections. Compiled, installed and configured OpenVPN in the Android system.img. Modified init script automatically loads the module and creates VPN connection.

The backward video channel to the virtual environment was realized by installing a VNC server (fast-droidvnc).

5.3 Overview of the Complete Toolchain

An overview of the video toolchain is depicted in Figure 5.3. The virtual camera and the virtual display are Gazebo plugins.

5.4 A Sample Test Scenario

For testing the incorporation of the modified Android emulator, a test scenario similar to the one in Section 4.7 was performed. The virtual camera is directly connected with the camera of the Android emulator. The *Barcode scanner* application from the Android market is used for detecting the QR code. Figure 5.4 shows the screen of the mobile device. It correctly shows the decoded content of the QR code.

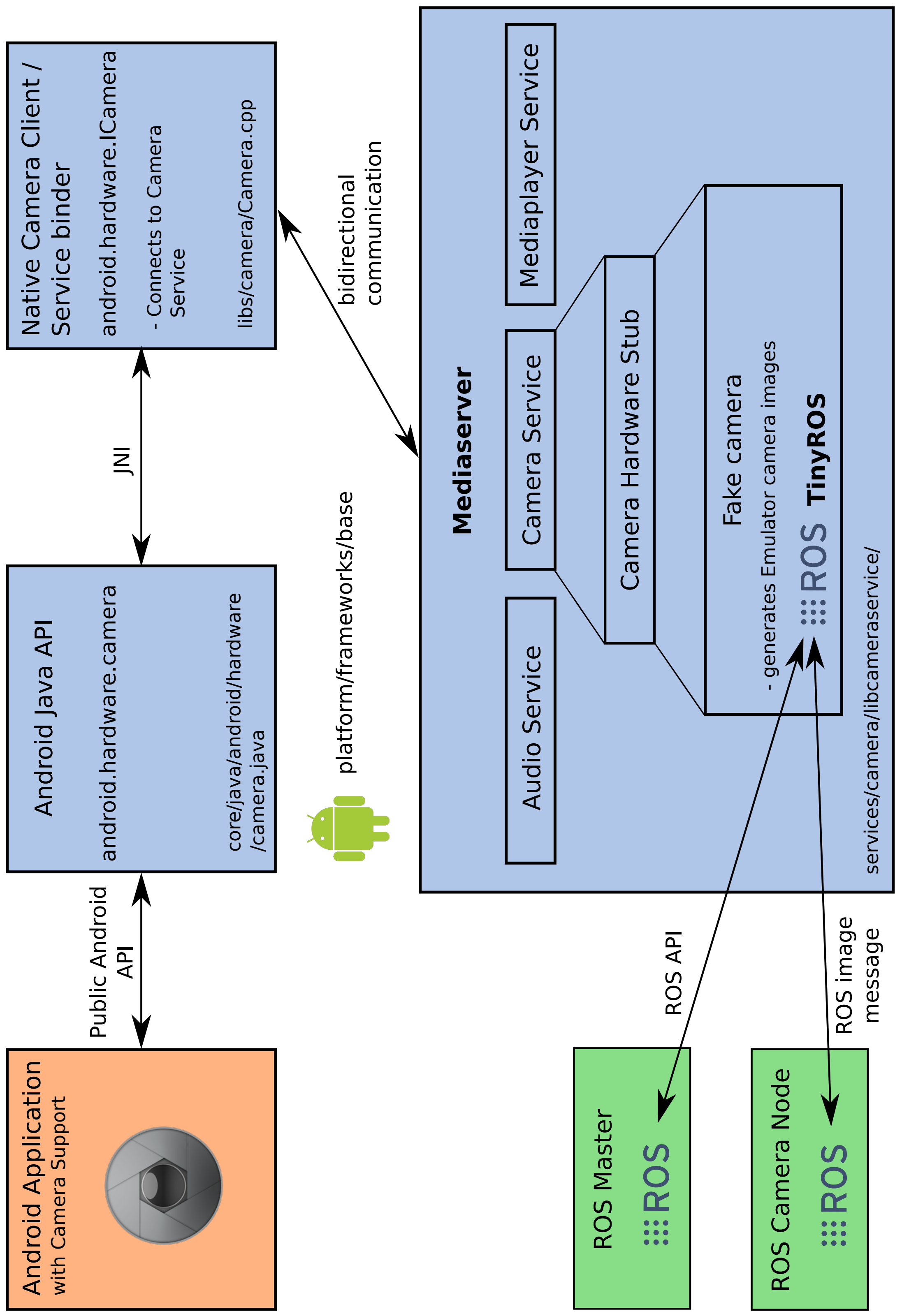


Figure 5.2: The new Android emulator camera processing chain with TinyROS extension.

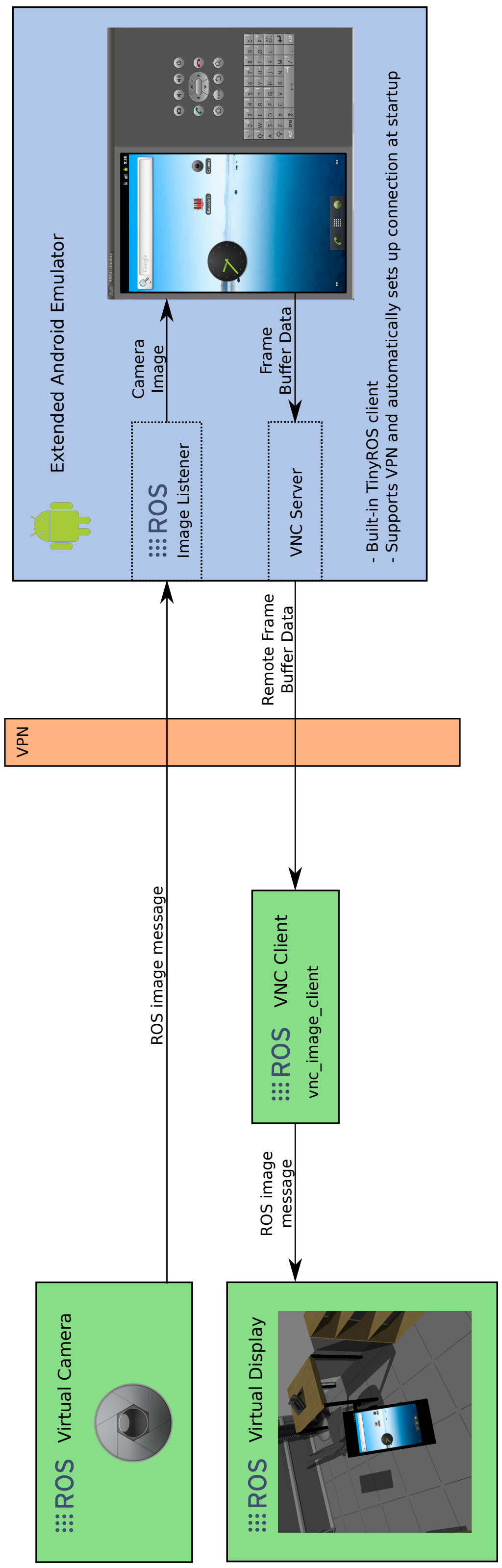
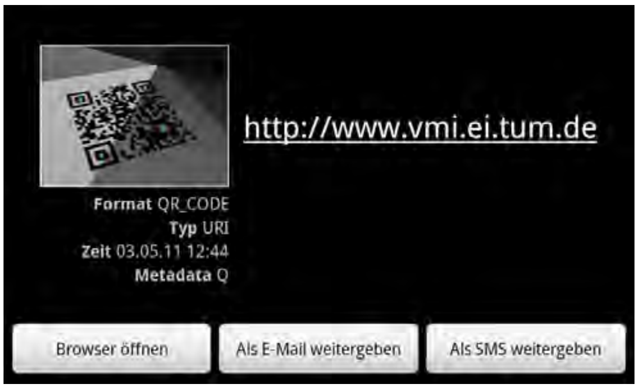


Figure 5.3: An overview of the complete toolchain with the Android emulator.


http://www.vmi.ei.tum.de

Format QR_CODE
Typ URI
Zeit 03.05.11 12:44
Metadata Q

Figure 5.4: The *Barcode scanner* application shows the information of the detected QR Code.

Chapter 6

Evaluation of the Implemented System and Improvement Ideas

In this Chapter the implemented toolchain is evaluated based on first test results. Based on the results of the review, improvement potentials are identified and concrete enhancement ideas are given.

6.1 The Virtual Environment – ROS and Gazebo

Before the actual video toolchain is evaluated, a quick view at the simulation framework is reasonable and necessary. The ROS and the 3D simulator Gazebo have been chosen for the realization. Especially the already available sensors and source codes for sensor simulation in Gazebo enabled a fast and solid implementation of the video toolchain. Predefined ROS messages and packages were used to create reusable modules which are fully compatible with existing image processing tools. The package system of ROS allowed to create packages that can be used independently to provide certain functionality without loading the whole system. For example, the `kinect4skype` package can be used with any other image topic subscriber node and the video texture can display images from every other image topic publishing node. An enormous advantage is the availability of fully featured virtual environments. The example scenarios given in Section 4.7 and Section 5.4 are based on the *Cognitive Office* simulation model. This allows a quick realization of a simulation in a realistic virtual environment. The sample implementation in Section 4.7 further shows the advantage of the ROS package system. By using the predefined sensor_msgs/Image message for image transport, the already available barcode detection node zbar_barcode_reader_node can be used without any modification.

During the creation of the 3D model of a mobile device, minor drawbacks of the 3D simulator *Gazebo* have been identified. The crashing due to an overflow of the shadow buffer when using high polygon models was already mentioned in Section 3.1. The suggested solution for this problem was to disable shadowing. But this is no acceptable solution for creating a realistic video toolchain that should also evaluate the behavior of algorithms for varying lighting. For that reason, the detailed *Google Nexus S* model could not be used and a simple boxed based model had to be created. Another problem was the weak support of 3D mesh files. Although Wavefront OBJ and Collada mesh files are supported by Gazebo and could be loaded successfully, the models did not look like in the 3D modeling tools. Some error patterns are described in Section 3.1. Debugging model issues is extremely difficult or even impossible in Gazebo, since no debug output or error messages are available. Only deep changes in the source code and a recompilation of the whole Gazebo system could help identifying this issues. Another weak point of the tool is the low fidelity of the 3D simulation. Although today's graphic cards allow fast and realistic rendering, The solid body simulation with *ODE* did not reveal any problems, but its persistent movement and collision calculation slows down the simulation a lot.

But major enhancements for the Gazebo simulator are already planned [81]. These could eliminate the above mentioned problems. The most interesting technical enhancements for the next two releases are:

- Visualization enhancements: improvement of shadows and materials.
- Better third-party model support.
- GUI for importing and manipulating models.
- Enhancement of physics engine: replacement of ODE with simpler and faster system.
- Simplification of simulation loading, rewinding, saving and resuming.

An important long term goal is the enhancement of the rendering fidelity to allow more accurate simulation of vision based systems. But not only technical enhancements are planned. The following things on the development roadmap could simplify the realization of the introduced concept:

- Incorporation of a human model.
- Addition of sensors: GPS, Wi-Fi, microphones/speakers, switches and buttons.
- Model creation from real-world sensor data.
- More tutorials and a more detailed documentation.

Especially the addition of the new sensors and the incorporation of a human model would be very useful for the realization. Since the incorporation of the new sensor device is planned for the next but one release of ROS, the ideas of the simulation concept, introduced in Chapter 3, could be used as basis for this work.

Looking at the planed enhancements and the recent work in the code repositories shows, that Gazebo is on the move to get even more powerful. With over 100 public, open source repositories, the ROS is enhanced and growing every day. This allows the conclusion that the decision to use ROS and Gazebo for the realization of the mobile device simulation toolchain was the right choice.

6.2 Virtual Video System for Gazebo

The virtual video system has been tested in several different scenarios. Some examples of those scenarios are:

1. The flycam, controlled by the *SpaceNavigator*, was used for presenting the *Cognitive Office* over several *Skype* video chats.
2. The flycam and the *camera_move* node were used for flying through a model of the TUM building 0509.
3. The virtual mobile device, equipped with a virtual camera and a virtual display, was used for performing the QR scenarios that are described in Section 4.7 and Section 5.4.
4. The Linux video loopback module was used for streaming the kinect4skype output to several media players at the same time.
5. The `vnc_image_client` was used to display the content of a real PC monitor on the virtual monitor of the *Cognitive Office*.

Example scenario 1 used almost all parts of the implemented video toolchain. The virtual camera was bound to the flycam model and the monitor in the *Cognitive Office* displayed the image stream of the camera using the dynamic texture based virtual display. The image_sink was used to write the camera images to a video loopback device from which Skype was reading the V4L2 stream. In Example scenario 3, the virtual camera and the virtual display were both bound to the mobile device model. Scenario 4 was performed for testing the compatibility with the most common multimedia player and other video clients, such as

- guvcview, a GTK+ UVC viewer.
- luvcview, a small video capture program for webcam testing and problem debugging.
- VLC media player, a common media player and multimedia framework.
- Skype, a messenger with video chat support.
- Google voice and video chat, adds voice and video chat support to Google products like *Google Mail* or *Google Talk*.

The scenarios have been used to test the modules and their interaction. Regular iterations and changes of constellations have been performed in order to locate problems and to verify, that problems have been solved. In addition to the given scenarios, variations and combinations of them have been tested. The system was further tested on different hardware and under different *Ubuntu* versions. At the current stage, their are no known problems in the video toolchain and all of the tested V4L2 clients are working flawlessly with the video loopback device.

The source code of the implemented modules is well commented. For C++ implementations, a Doxygen [82] compatible notation was used. Python *docstrings* were used for most of the written Python code. This means, that the source code is prepared for automated source code documentation systems like Doxygen [82] for C++ or Epydoc [83] for Python that are commonly used by the ROS project.

The frame rate of the virtual camera in Gazebo is depended on the rendering frame rate of the scene itself, since it is hooked to this rendering. That means, when the rendering in Gazebo is slow, the frame rate of the camera will also be slow. Another drawback of the virtual camera is, that it does not support distortion at the moment. This is especially a disadvantage when computer vision algorithms should be tested, since compensating for the distortion is an important task for real camera systems.

When a virtual display is bound to an object, the dynamic texture is currently shown on every vertex face of it. This can easily be changed in the source code. But defining the the correct face in the URDF file would be very hard for generic meshes. This could for example be solved by assigning a special texture to the mesh in the 3D modeling tool and replacing this afterwards with the dynamic texture for the virtual display. The image file loading algorithm supports all common image formats and all kinds of BGR8 and RGB8 images from ROS image topics can be loaded into the texture buffer. This can should be sufficient for most use cases. In a next version, the displaying of video files could be added to the virtual display.

The image_sink video stream writer currently only supports the YUYV pixel format and memory mapped operation. Since the YUYV format is also the default pixel format for most clients, there is currently no need for adding other formats. The memory mapped operation is only relevant for the communication with the video loopback module. For this reason, it is not necessary to add other writing techniques to the writer.

6.3 Android Emulator Integration

The sample scenario in Section 5.4 gives a first impression what is enabled by the linking of the Android emulator with video toolchain. The realization of the integration allows the usage of the virtual camera images in every available Android application which access the camera via the common Android API functions. The content of the Android emulators display can also be shown on the virtual display of the simulated mobile display. The modified Android system allows to use the network in the emulator similar as on a real device. The automatically set up VPN connection enables direct connections to all hosts in the *Münchner Wissenschaftsnetz (MWN)* and vice versa.

In contrast to the video toolchain, the Android emulator integration is not yet working flawlessly. The `tiny_roscpp` client, which was integrated at the lowest possible level in the Android emulator, is very slow. On average it can only handle a frame rate of 2 frames per second. Sometimes the applications, that access the camera, even crash due to the slow image processing of the mediaserver. As the potential cause of the problem, the incomplete support of *POSIX Threads (Pthreads)* and *std::String* could be identified. Although the approach to integrate the ROS binding at the lowest possible level was chosen in order to maintain all API functions unchanged, it seems to be reasonable to integrate the ROS binding directly into the Android Java API. This should be not to hard to implement by using the recently released rosjava ROS client library [84]. It is a pure Java implementation of ROS with Android support. Using this library for implementing the ROS binding directly in the Android Java API could have several advantages:

- The integration is easier to debug since stack traces and error are automatically published by the logging feature of Android. The `tiny_roscpp` client could only be debugged using the *GNU Debugger (GDB)*.
- More complete and tested implementation of the ROS API which can be used for the integration of other sensor data from the virtual environment, too.
- No need to update and maintain the `tiny_roscpp` library on Android or ROS updates.

The automatically started VNC server on the Android emulator is used to get the screen content of the Android emulator back into the simulation. The chosen VNC server fastdroid-vnc is working very reliably, but causes some significant processor load for the virtual Android instance. A big drawback of the VNC solution is, that camera content, that is displayed on the screen, cannot be read out by VNC. For this reason, the image of the barcode scanner application in Figure 5.4 does only show the overlay generated by the application itself. This is due to the fact, that the camera image is rendered to another buffer like the generated screen content which is read out by the VNC server. A possible solution for this problem could be to use the built-in VNC server of *QEMU* on which the Android emulator is based. But enabling the VNC server by adding the parameter `-vnc :1` to the Android emulator startup command, leads to a stuck loading screen of the Android emulator.

Another problem is the Android emulator itself. In some cases, the Android emulator is very slow. But during this year's Google I/O Xavier Ducrohet and Tor Norbye announced some upcoming enhancements for the Android emulator [85]. Besides a new GL render strategy and the emulator should also get some other performance improvements.

Chapter 7

Conclusion and Future Work

A new concept for modeling a mobile device and testing mobile software in a virtual environment was introduced in this work. The overview of possible realization ideas for various existing sensors and actuators is a good starting point for other implementations. The prototypical implementation of the video toolchain gives an insight, how to approach a realization of a complete toolchain in ROS/Gazebo. The integration of the Android emulator enables testing of real-deployable Android applications in the created toolchain. The QR code sample verifies that the implementation is working as expected and that mobile software testing can be performed easily in the virtual environment.

The implemented video toolchain is not limited to the mobile device scenario. The *kinect4skype* idea shows the capability of the implemented parts. The modular structure of the implementation allows reuse and fast adaptations for other projects, such as for example tele-operation or surveillance scenarios. Especially the tele-operation of robots and the remote control of intelligent things will become more and more important in relation with mobile devices in the next few years. Already today mobile devices are most popular user terminals for this applications. The implemented toolchain together with the already available packages for the ROS are able to simulate the whole scenarios with just a little development effort: many robots and intelligent objects are already available and the mobile devices can be simulated with the implementation described in this work.

For a future work, the implemented toolchain could be refined and other concepts for virtual sensors in Gazebo could be realized. Combining all the results in one application could create a comprising mobile device simulator, that allows to create the mobile device via an intuitive GUI, to prepare a test scenario and finally to perform and evaluate the testing results. This would be a powerful application for mobile software developers. It could be a decisive advantage in development and testing time and quality, and thus for the success of the created application.

Other scenarios for the simulation toolchain are acceptance and usability tests for all types of electronic devices in an early stage of their development. On the one hand, these tests could be carried out by test candidates in front of computers. On the other hand, one could use human models in the simulation environment. Representatives with different physical abilities could be created for each demographic group. This would allow to test the usability of the devices, no matter what age, size or physical constraints the people have.

Appendix A

Mobile Device URDF xacro-File

```
<?xml version="1.0"?>
<robot xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#sensor"
 xmlns:controller="http://playerstage.sourceforge.net/gazebo/xmlschema/#controller"
 xmlns:interface="http://playerstage.sourceforge.net/gazebo/xmlschema/#interface"
 xmlns:xacro="http://playerstage.sourceforge.net/gazebo/xmlschema/#interface"
 name="mobile_phone">

  <xacro:property name="pi" value="3.1415" />

  <!-- Mobile device body dimensions and mass -->
  <xacro:property name="height" value="0.1239" />
  <xacro:property name="width" value="0.063" />
  <xacro:property name="depth" value="0.01088" />
  <xacro:property name="mass" value="0.129" />

  <!-- Display dimensions -->
  <xacro:property name="display_height" value="0.0909" />
  <xacro:property name="display_width" value="0.0545" />
  <xacro:property name="display_depth" value="0.0005" />

  <!-- Mobile device body -->
  <link name="mobile_base_link">
    <inertial>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <mass value="${mass}" />
      <!-- Calculate inertia tensor for cuboid -->
      <inertia ixx="${1/12 * mass * (height*height + depth*depth)}" ixy="0" ixz="0"
       iyy="${1/12 * mass * (width*width + depth*depth)}" iyz="0"
       izz="${1/12 * mass * (height*height + width*width)}" />
    </inertial>
    <visual>
```

```
      <geometry>
        <box size="${height} ${width} ${depth}" />
      </geometry>
    </visual>
    <collision>
      <geometry>
        <box size="${height} ${width} ${depth}" />
      </geometry>
    </collision>
  </link>
  <gazebo reference="mobile_base_link">
    <turnGravityOff>true</turnGravityOff>
    <material>Gazebo/Black</material>
  </gazebo>

  <joint name="display_joint" type="fixed">
    <parent link="mobile_base_link"/>
    <child link="mobile_display"/>
    <origin xyz="0.002 0 ${depth/2}"/>
  </joint>

  <!-- The display -->
  <link name="mobile_display">
    <inertial>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <mass value="0.0001" />
      <inertia ixx="0.00000001" ixy="0" ixz="0"
       iyy="0.00000001" iyz="0" izz="0.00000001" />
    </inertial>
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0 " />
      <geometry name="mobile_display_visual_geom">
        <box size="${display_height} ${display_width} ${display_depth}" />
      </geometry>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0 " />
      <geometry name="mobile_display_collision_geom">
        <box size="${display_height} ${display_width} ${display_depth}" />
      </geometry>
    </collision>
  </link>
  <gazebo reference="mobile_display">
```

```
    <turnGravityOff>true</turnGravityOff>
    <material>Gazebo/Grey</material>
  </gazebo>
  <!-- Video texture controller binding -->
  <gazebo>
    <controller:gazebo_ros_video name="gazebo_ros_video_controller"
     plugin="libgazebo_ros_video.so">
      <!-- Link name, where the texture will be bound to -->
      <bodyName>mobile_display</bodyName>
      <!-- Topic, where image is read from -->
      <imageTopic>/mobile_device/image_raw</imageTopic>
      <!-- Topic for loading a local image file -->
      <imageLoad>/mobile_device/load_image_file</imageLoad>
    </controller:gazebo_ros_video>
  </gazebo>

  <joint name="camera_joint" type="fixed">
    <parent link="mobile_base_link"/>
    <child link="mobile_camera"/>
    <origin xyz="0.032 0.01 ${-depth/2}" rpy="0 0 0"/>
  </joint>

  <!-- Visual representation of the camera -->
  <link name="mobile_camera">
    <inertial>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <mass value="0.0001" />
      <inertia ixx="0.00000001" ixy="0" ixz="0"
        iyy="0.00000001" iyz="0" izz="0.00000001" />
    </inertial>
    <visual>
      <geometry>
        <cylinder radius="0.008" length="${display_depth}" />
      </geometry>
    </visual>
    <collision>
      <geometry>
        <cylinder radius="0.008" length="${display_depth}" />
      </geometry>
    </collision>
  </link>
  <gazebo reference="mobile_camera">
    <turnGravityOff>true</turnGravityOff>
    <material>Gazebo/Grey</material>
  </gazebo>

  <joint name="camera_unit_joint" type="fixed">
    <parent link="mobile_camera"/>
    <child link="mobile_camera_unit"/>
    <!-- Rotate by pi/2 in pitch to get camera pointing in
      -z direction -->
    <origin xyz="0 0 ${-display_depth/2}" rpy="0 ${pi/2} 0"/>
  </joint>

  <!-- Camera unit (functional) -->
  <link name="mobile_camera_unit">
    <inertial>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <mass value="0.0001" />
      <inertia ixx="0.00000001" ixy="0" ixz="0"
       iyy="0.00000001" iyz="0" izz="0.00000001" />
    </inertial>
    <visual>
      <geometry>
        <sphere radius="0.0001" />
      </geometry>
    </visual>
    <collision>
      <geometry>
        <sphere radius="0.008" />
      </geometry>
    </collision>
  </link>
  <gazebo reference="mobile_camera_unit">
    <turnGravityOff>true</turnGravityOff>
    <material>Gazebo/Grey</material>
  </gazebo>
  <!-- Camera controller -->
  <gazebo reference="mobile_camera_unit">
    <sensor:camera name="camera_sensor">
      <imageSize>480 800</imageSize>
      <updateRate>30.0</updateRate>
      <controller:gazebo_ros_camera name="controller-name"
       plugin="libgazebo_ros_camera.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>100.0</updateRate>
        <!-- Topic, where image is published to -->
        <imageTopicName>/mobile_device/image_raw</imageTopicName>
        <cameraInfoTopicName>/mobile_device/camera_info</cameraInfoTopicName>
        <frameName>mobile_camera_unit</frameName>
      </controller:gazebo_ros_camera>
    </sensor:camera>
  </gazebo>
</robot>
```

Listing 1: URDF description of the mobile device. The macro language *xacro* is used to simplify maintaining and editing of the file.

List of Figures

List of Tables

Bibliography

[1] Alan Livingston. Smartphones and Other Mobile Devices: The Swiss Army Knives of the 21st Century. *EDUCAUSE Quarterly*, 27(2):46–52, 2004.

[2] Morgan Stanley. Internet Trends. http://www.morganstanley.com/institutional/techresearch/pdfs/MS_Internet_Trends_060710.pdf, June 2010.

[3] Gartner, Inc. Worldwide Mobile Device Sales 2010. http://www.gartner.com/it/page.jsp?id=1543014, February 2011.

[4] Gartner, Inc. Worldwide Mobile Device Sales Q1 2011. http://www.gartner.com/it/page.jsp?id=1689814, May 2011.

[5] Appcelerator Inc. and IDC International Data Group, Inc. Appcelerator IDC Mobile Developer Report, January 2011. http://www.appcelerator.com/company/survey-results/mobile-developer-report-january-2011/, January 2011.

[6] VisionMobile. Mobile Developer Economics 2010 and Beyond. http://www.visionmobile.com/research.php#devecon, July 2010.

[7] Haruka Kikuchi, Shinji Kimura, Shinzo Ohkubo, Hiroshi Inamura, and Atsushi Takeshita. User Interface Development from Conceptualization to Prototype Evaluation through UCD Processes. *NTT DoCoMo Technical Journal*, 12(3):33–41, December 2010.

[8] TestPlant Ltd. Testing iPhone Applications with eggPlant. http://www.testplant.com/solutions/iphone_app_testing, August 2010.

[9] Jayway. robotium – User scenario testing for Android. http://code.google.com/p/robotium/, April 2011.

[10] OpenIntents . Sensor simulator. http://code.google.com/p/openintents/wiki/SensorSimulator, January 2011.

[11] Tom Gibara. Obtaining a Live Camera Preview in Android. http://www.tomgibara.com/android/camera-source, May 2010.

[12] Samsung Electronics Co. Ltd. Samsung Sensor Simulator. http://innovator.samsungmobile.com/down/cnts/toolSDK.detail.view.do?platformId=1&cntsId=9460, May 2011.

[13] G. Gary Wang. Definition and Review of Virtual Prototyping. *Journal of Computing and Information Science in Engineering*, 2(3):232–236, 2002.

[14] P.F. Lister, P.L. Watten, M.R. Lewis, P.F. Newbury, M. White, M.C. Bassett, B.J.C. Jackson, and V. Trignano. Electronic simulation for virtual reality: virtual prototyping. In *Theory and Practice of Computer Graphics, 2004. Proceedings*, pages 71–76, june 2004.

[15] S. Kabadayi, A. Pridgen, and C. Julien. Virtual sensors: abstracting data from physical sensors. In *World of Wireless, Mobile and Multimedia Networks, 2006. WoWMoM 2006. International Symposium on a*, pages 6 pp.–592, 0-0 2006.

[16] Tom Lovett, Eamonn O'Neill, James Irwin, and David Pollington. The calendar as a sensor: analysis and improvement using data fusion with social networks and location. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*, Ubicomp '10, pages 3–12, New York, NY, USA, 2010. ACM.

[17] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully B. Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[18] S. Cousins. ROS on the PR2 [ROS Topics]. *Robotics Automation Magazine, IEEE*, 17(3):23–25, sept. 2010.

[19] Florian Weisshardt, Ulrich Reiser, Alexander Verl, and Christopher Parlitz. Making high-tech service robot platforms available. In *ISR/ROBOTIK 2010*, ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics), 2010.

[20] Matthias Kranz, Thomas Linner, Bernhard Ellmann, Andreas Bittner, and Luis Roalter. Robotic service cores for ambient assisted living. In *4th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth2010)*, pages 1–8, 22–25 March 2010.

[21] Luis Roalter, Matthias Kranz, and Andreas Möller. A Middleware for Intelligent Environments and the Internet of Things. In *Ubiquitous Intelligence and Computing*, volume 6406 of *Lecture Notes in Computer Science*, pages 267–281. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16355-5_23.

[22] Matthias Kranz, Radu Bogdan Rusu, Alexis Maldonado, Michael Beetz, and Albrecht Schmidt. A player/stage system for context-aware intelligent environments. In *Proceedings of UbiSys'06, System Support for Ubiquitous Computing Workshop, at the 8th Annual Conference on Ubiquitous Computing (Ubicomp 2006), Orange County California, September 17-21, 2006*, 2006.

[23] O. Wongwirat and C. Chaiyarat. A position tracking experiment of mobile robot with inertial measurement unit (imu). In *Control Automation and Systems (ICCAS), 2010 International Conference on*, pages 304–308, oct. 2010.

[24] Russell Smith. Open Dynamics Engine – User Guide. `http://me.seu.edu.cn/RCBaoMing/2007/download/ode-latest-userguide.pdf`, February 2006.

[25] Torus Knot Software Ltd. OGRE – Open Source 3D Graphics Engine. `http://www.ogre3d.org/`, May 2011.

[26] The wxWidgets Team. wxWidgets – Cross-Platform GUI Library.

[27] Open Handset Alliance and Google Inc. Android SDK. `http://developer.android.com/sdk/index.html`, May 2011.

[28] Millard F. Beatty. *Principles of Engineering Mechanics*, volume 33 of *Mathematical Concepts and Methods in Science and Engineering*, chapter 9, pages 355–404. Springer, 2006.

[29] ROS.org community. XML Robot Description Format (URDF). `http://www.ros.org/wiki/urdf/XML`, June 2010.

[30] Jim Branson. Example: The Inertia Tensor for a Cube. `http://hepweb.ucsd.edu/ph110b/110b_notes/node26.html`, March 2011.

[31] Visual Computing Lab, ISTI - CNR. MeshLab. `http://meshlab.sourceforge.net/`, February 2011.

[32] ROS.org bug tracker. Shadow rendering crashes gazebo with too many triangles. `https://code.ros.org/trac/ros-pkg/ticket/4241`, December 2010.

[33] G.M. Djuknic and R.E. Richton. Geolocation and assisted gps. *Computer*, 34(2):123–125, feb 2001.

[34] Ian Poole. Assisted GPS, A-GPS. `http://www.radio-electronics.com/info/cellulartelecomms/location_services/assisted_gps.php`, January 2010.

[35] Raul Correal. The Player Project – ID: 3194889 – GPS Interface. `http://sourceforge.net/tracker/?func=detail&aid=3194889&group_id=42445&atid=433166`, February 2011.

[36] Jorge Torres-Solis, Tiago H. Falk, and Tom Chau. *Ambient Intelligence*, chapter A review of indoor localization technologies: towards navigational assistance for topographical disorientation, pages 51–83. InTech, March 2010.

[37] Matthias Kranz, Albrecht Schmidt, and Paul Holleis. Embedded interaction: Interacting with the internet of things. *IEEE Internet Computing*, 14(2):46 – 53, March-April 2010.

[38] Matthias Kranz, Carl Fischer, and Albrecht Schmidt. A comparative study of DECT and WLAN signals for indoor localization. In *8th Annual IEEE Int'l Conference on Pervasive Computing and Communications (PerCom 2010)*, pages 235–243, Mannheim, Germany, March 2010. IEEE.

[39] Andrei Popletee. *Indoor positioning using FM radio signals.* PhD thesis, DISI – University of Trento, April 2011.

[40] T. Gigl, G.J.M. Janssen, V. Dizdarevic, K. Witrisal, and Z. Irahhauten. Analysis of a uwb indoor positioning system based on received signal strength. In *Positioning, Navigation and Communication, 2007. WPNC '07. 4th Workshop on*, pages 97–101, march 2007.

[41] Jim Zyren and Al Petrick. Tutorial on basic link budget analysis, 1998.

[42] Viering Ingo. Lecture Notes for System Aspects in Communications. Lectures notes available from `http://www.lnt.ei.tum.de/mainframe/DBdata/lehrveranstaltungen_einzeln.php?ID_p_h=15&language=en`, October 2010.

[43] 3Com Corporation. 3Com Wireless Antennas Product Guide. `http://www.scribd.com/doc/32613170/3Com%C2%AE-Wireless-Antennas`, 2005.

[44] C.W. Tan and S. Park. Design of accelerometer-based inertial navigation systems. *Instrumentation and Measurement, IEEE Transactions on*, 54(6):2520–2530, 2005.

[45] Andreas Möller, Johannes Scherr, Luis Roalter, Stefan Diewald, Nils Hammerla, Thomas Plötz, Patrick Olivier, and Matthias Kranz. Gymskill: Mobile exercise skill assessment to support personal health and fitness. In *Adjunct Proceedings of 9th International Conference on Pervasive Computing (Pervasive 2011)*, San Francisco, CA, USA, June 2011. To appear.

[46] James Diebel. Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors. Technical report, Stanford University, 2006.

[47] M.J. Caruso. Applications of magnetic sensors for low cost compass systems. In *Position Location and Navigation Symposium, IEEE 2000*, pages 177–184, 2000.

[48] Joe Mehaffey. GPS Altitude Readout – Altitude Accuracy. `http://gpsinformation.net/main/altitude.htm`, October 2001.

[49] Paul Semenza. Touch Screen Market Update. `http://www.displaysearch.com/cps/rde/xbcr/displaysearch/FlexTech_Sensor_Workship.pdf`, November 2010.

[50] M. Kranz, P. Holleis, and A. Schmidt. Distscroll - a new one-handed interaction device. In *Proc. 25th IEEE International Conference on Distributed Computing Systems Workshops*, pages 499–505, 2005.

[51] Nishkam Ravi, James Scott, and Liviu Iftode. Context-aware battery management for mobile phones: A feasibility study. Technical report, 2006.

[52] Makoto Yamauchi and Kazuyo Iwamoto. Combination of optical shape measurement and augmented reality for task support: I. Accuracy of position and pose detection by ARToolKit. *Optical Review*, 17:263–268, 2010. 10.1007/s10043-010-0046-z.

[53] B. Thomas, V. Demczuk, W. Piekarski, D. Hepworth, and B. Gunther. A wearable computer system with augmented reality to support terrestrial navigation. In *Wearable Computers, 1998. Digest of Papers. Second International Symposium on*, pages 168–171, oct 1998.

[54] Patrick Dähne and John N. Karigiannis. Archeoguide: System architecture of a mobile outdoor augmented reality system. In *Proceedings of the 1st International Symposium on Mixed and Augmented Reality*, ISMAR '02, pages 263–, Washington, DC, USA, 2002. IEEE Computer Society.

[55] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Second Edition*. Charles River Media, Inc., Rockland, MA, USA, 2003.

[56] Marc Christie and Patrick Olivier. Camera control in computer graphics: models, techniques and applications. In *ACM SIGGRAPH ASIA 2009 Courses*, SIGGRAPH ASIA '09, pages 3:1–3:197, New York, NY, USA, 2009. ACM.

[57] Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. Real-time lens blur effects and focus control. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 65:1–65:7, New York, NY, USA, 2010. ACM.

[58] Qiuxia Wu, Zhiyong Wang, Feiqi Deng, and D.D. Feng. Realistic human action recognition with audio context. In *Digital Image Computing: Techniques and Applications (DICTA), 2010 International Conference on*, pages 288–293, dec. 2010.

[59] A. Saxena and A.Y. Ng. Learning sound location from a single microphone. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 1737–1742, may 2009.

[60] Creative Labs: Connect. OpenAL. `http://connect.creativelabs.com/openal/default.aspx`, September 2010.

[61] Ken Iwasaki, Takashi Miyaki, and Jun Rekimoto. AffectPhone: A Handset Device to Present User's Emotional State with Warmth/Coolness. In *B-Interface*, pages 83–88, 2010. DBLP:conf/biostec/2010in.

[62] Michael Gasperi. Galvanic Skin Response Sensor. `http://www.extremenxt.com/gsr.htm`.

[63] Elena A. Fedorovskaya, Paige Miller, Girish Prabhu, Cecelia Horwitz, Tomasz Matraszek, Richard Blazey, and Serguei Endrikhovski. Affective imaging: psychological and physiological reactions to individually chosen images. In *SPIE*, volume 4299, 2001. doi:10.1117/12.429524.

[64] Jon Stokes. This September, OLED no longer "three to five years away". `http://arstechnica.com/gadgets/news/2009/08/this-september-oled-no-longer-three-to-five-years-away.ars`, August 2009.

[65] L.A. Jones and H.-N. Ho. Warm or cool, large or small? the challenge of thermal displays. *Haptics, IEEE Transactions on*, 1(1):53–70, jan.-june 2008.

[66] Zhenwei You. B-Touch Innovative Mobile Touchphone For The Blind. `http://www.yankodesign.com/2009/06/15/touchphone-for-the-blind/`.

[67] Game Physics Simulation. Bullet – Game Physics Simulation. `http://bulletphysics.org/`, March 2011.

[68] 3Dconnexion. 3Dconnexion : SpaceNavigator. `http://www.3dconnexion.com/products/spacenavigator.html`.

[69] Steve Streeting. OGRE Manual v1.7. `http://www.ogre3d.org/docs/manual/`, December 2009.

[70] Hervé Drolon. The FreeImage Project. `http://freeimage.sourceforge.net/`, January 2011.

[71] techtonik. python-vnc-viewer. `http://code.google.com/p/python-vnc-viewer/`, September 2010.

[72] Johannes Schindelin et al. LibVNCServer/LibVNCClient. `http://libvncserver.sourceforge.net/`, March 2011.

[73] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[74] Kitware Inc. VTK – The Visualization Toolkit. `http://www.vtk.org/`, May 2011.

[75] Julius Kammerl. Point Cloud Compression Tutorial. `http://www.pointclouds.org/documentation/tutorials/compression.php#octree-compression`, May 2011.

[76] Linux For Devices. Intro to V4L2. `http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Intro-to-V4L2/`, March 2009.

[77] IOhannes m zmölnig. v4l2loopback - a kernel module to create V4L2 loopback devices. `https://github.com/umlaeute/v4l2loopback`.

[78] Dejan Pangercic. zbar_barcode_reader_node. `http://www.ros.org/browse/details.php?name=zbar_barcode_reader_node`, August 2010.

[79] Jeff Brown. ZBar bar code reader. `http://zbar.sourceforge.net/`, May 2011.

[80] seeingwithsound seeingwithsound seeingwithsound seeingwithsound seeingwithsound seeingwithsound seeingwithsound seeingwithsound. Issue 13133: Camera preview blank in SDK 2.3 emulator. `http://code.google.com/p/android/issues/detail?id=13133`, December 2010.

[81] ROS community. simulator_gazebo/Roadmap. `http://www.ros.org/wiki/simulator_gazebo/Roadmap`, March 2011.

[82] Dimitri van Heesch. Doxygen. `http://www.stack.nl/~dimitri/doxygen/`, March 2011.

[83] Edward Loper. Epydoc. `http://epydoc.sourceforge.net/`, January 2008.

[84] Damon Kohler and Ken Conley. rosjava – An implementation of ROS in pure Java with Android support. `http://code.google.com/p/rosjava/`, May 2011.

[85] Xavier Ducrohet and Tor Norbye. Google I/O 2011 – Android Development Tools. `http://www.google.com/events/io/2011/sessions/android-development-tools.html`, May 2011.