



# Study report

of the course Informationstechnik

at the Baden-Wuerttemberg Cooperative State University Mannheim

---

## Subject

**Creating an development and deployment infrastructure for the TIGERs Mannheim On-Bot Vision Software**

---

**Felix Weinmann**

05.05.2021

---

| | |
|---|---|
| Processing Period: | 24.09.20 – 05.05.21 |
| Student id, course: | 2891832, TINF18IT1 |
| Supervisor: | Jürgen Schultheis |

# Declaration

I hereby assure you that I have written my study report on the

Subject

**Creating an development and deployment infrastructure for the TIGERs Mannheim On-Bot Vision Software**

independently and that I have not used any other sources and aids than those indicated.

I also assure you that the electronic version submitted is the same as the printed version.*

* if both versions are required.

Mannheim, 05.05.2021

# Abstract

The new RoboCup Small Size League team TIGERs Mannheim onboard camera software RobotPi requires tools and infrastructure for development and deployment. Therefore in this work a CMake configuration to compile RobotPi on multiple platforms in development and production configuration was created. A custom Raspbian image for deployment of necessary dependencies and configurations to the robots was developed with pi-gen. These steps were integrated into GitLab pipelines for continouus integration. For deployment of RobotPi to multiple robots a usb based automatic debian package installation was implemented. To support development without access to a robot and play field an image simulation tool was created with Blender and a method to load them into RobotPi provided.

# Kurzfassung

Die neue Kamerasoftware RobotPi für die integrierte Kamera auf den Robotern des RoboCup Small Size League Teams TIGERs Mannheim benötigt Werkzeuge und Infrastruktur für die Entwicklung und Verteilung. Dazu wurde im Rahmen dieser Arbeit auf der Basis von CMake eine Kompilierungskonfiguration für RobotPi geschaffen, welche auf verschiedenen Rechnerarchitekturen für die Entwicklungsumgebung sowie den Produktiveinsatz funktioniert. Eine eigene Raspbian-Distribution wurde mithilfe von pi-gen für die Auslieferung mit den benötigten Abhängigkeiten und Konfigurationen erstellt. Diese Schritte wurden zur fortlaufenden Integration in GitLab Pipelines integriert. Für die Auslieferung des RobotPi Programms an mehrere Roboter wurde eine USB-Speicher-basierte Debian-Packet Installationsroutine entwickelt. Um die Entwicklung an RobotPi ohne Zugriff auf einen Roboter und ein Spielfeld zu ermöglichen, wurde eine Bildsimulierungsmöglichkeit mit Blender geschaffen und eine Routine zum Laden dieser Bilder in RobotPi implementiert.

# Contents

# List of Figures

# Listings

# 1 Introduction

During the annual RoboCup robotics researchers from around the world compete in various categories. The target is to win in 2050 against the current world champion in soccer [1]. The RoboCup has a Small Size League which focuses on "intelligent multi-agent cooperation and control in a highly dynamic environment" [2]. In the Small Size League, information about the absolute position on the playfield is usually provided by a centralized vision system [3]. Currently there are efforts to reduce the dependency on the centralized vision. The 2019 Technical Challenge was focused around finding and capturing the ball without the central vision [4]. The 2020/2021 Technical Challenge goes even further with the requirement to determin the absolute position of the bot to solve all tasks [5].

The team Team Interacting and Game Evolving Robots (TIGERs) Mannheim is a project of the Cooperative State University Mannheim. The team is participating in the RoboCup Small Size League since 2011. Since 2019 is a front looking camera integrated on the TIGERs robots [6]. This camera was integrated for the detection of balls and successfully used in the 2019 Technical Challenge [7]. The camera is connected directly to a Raspberry Pi 3A mounted on the bot. The Raspberry Pi is connected via UART to the main microcontroller of the bot, which communicates with the central TIGERs software Sumatra running on a personal computer. The software implementation of the Technical Challenge camera software had some issues due to the focus on the 2019 challenge. Therefore, a new camera software RobotPi is developed. This work focuses on providing a development and deployment environment for this software. This includes a custom Raspbian image for all prerequisites, rapid deployment of the software package, crosscompilation for the Raspberry Pi and provision of testing images.

# 2 Definition of Tasks

1. Compilation of the TIGERs camera software RobotPi with CMake.
   - 1.1 Compilation of the TIGERs camera software RobotPi on the Raspberry Pi with CMake.
   - 1.2 Cross-Compilation of the TIGERs camera software RobotPi on Windows and Linux for the Raspberry Pi with CMake.
   - 1.3 Compilation of the TIGERs camera software RobotPi on Windows and Linux for the compilation platform with CMake.
   - 1.4 Compilation of the TIGERs camera software RobotPi for the Raspberry Pi with CMake inside a GitLab pipeline.
   - 1.5 Integration of the TIGERs camera software RobotPi compilation in the corresponding Eclipse project.
2. Creation of a custom Raspbian image.
   - 2.1 Creation of a custom Raspbian image with all necessary configurations and dependencies required for RobotPi.
   - 2.2 Integration of the creation of the custom Raspbian images in a GitLab pipeline.
3. Implementing a deployment method for the TIGERs camera software RobotPi.
4. Implementing the loading images in the camera software RobotPi for testing and development.

5. Simulating images taken by the on board camera on a RobotCup Small Size League Division A Field.

# 3 Methods and Procedures

This section contains the used architecture and software.

The machine used to develop this work:

- Kubuntu 20.04.2 LTS
- AMD Ryzen 7 3700X
- cmake 3.16.3-1ubuntu1
- pi-gen 225f69828fa05361d6028edf2d7a69db73fe2b45
- Blender 2.90.1
- arm-none-eabi-gcc 9.2.1
- gcc 9.3.0-1ubuntu2

The Raspberry Pi on the bot:

- Raspberry Pi 3A
- cmake 3.13.4-1
- g++ 4:8.3.0-1+rpi2
- gcc 4:8.3.0-1+rpi2
- make 4.2.1-1.2
- ssh 1:7.9p1-10+deb10u2
- systemd 241-7 deb10u5+rpi1
- usbmount 0.0.22

The GitLab instance used:

- GitLab Community Edition 13.9.1

The server running the pi-gen continouus integration:

- Ubuntu 20.04.2 LTS
- GitLab Runner 13.9.0
- Docker CE 20.10.5 3-0 ubuntu-focal

# 4 Implementation

## 4.1 Compilation of the TIGERs camera software RobotPi with CMake

RobotPi is the TIGERs second generation of a camera ball and position detector focused around extensibility and utilization in real games. It's therefore necessary for development, automated testing and deployment to have a flexible compilation toolchain.

CMake is a tool to generate the appropiate makefiles for a system independent from the operating system and computer architecture [8]. CMake is a good fit for compiling RobotPi due to its widespread integration into integrated development environments (IDEs) like Visual Studio [9] and due to its nature integrated cross-compilation capabilities. This section describes how the RobotPi software compilation with CMake was achived and the various pitfalls and solution to these.

### 4.1.1 Compilation of the TIGERs camera software RobotPi on the Raspberry Pi with CMake

For the generation of the makefiles CMake requires a configuration file `CMakeLists.txt` in the root directory of the project. In this file the position of the source files, the build target and build type is set [10]. With the example given in [11] a first CMake file for compiling RobotPi can be created:

**Listing 4.1:** Basic CMake-Configuration

```
1 cmake_minimum_required(VERSION 3.0)
2 project(robotpi)
3
4 include_directories(src)
5 file(GLOB SOURCES "SRC/*.cpp")
6
7 add_executable(robotpi ${SOURCES})
```

This basic CMakeLists.txt file does not account for the camera library RobotPi is based on. RobotPi requires the Multi-Media Abstraction Layer (MMAL). MMAL is a library which provides a low level interface to the Raspberry Pi camera and HDMI output [12].

For linking the library with CMake the internal file structure needs to be given to CMake. Therefore it needs the base path `/opt/vc` of the library and a module file `FindMMAL.cmake` which describes the library structure. For this work the former `FindMMAL.cmake` file of the Kodi project is used [13]. Last the library needs to be linked with the compilation target.

**Listing 4.2:** CMake-Configuration to load MMAL as library

```
1 list(APPEND CMAKE_MODULE_PATH cmake/Modules)
2 list(APPEND CMAKE_PREFIX_PATH /opt/vc)
3
4 find_package(MMAL REQUIRED)
5
6 target_include_directories(robotpi PRIVATE src ${MMAL_INCLUDE_DIRS})
7 target_link_libraries(robotpi PRIVATE ${MMAL_LIBRARIES} pthread)
```

### 4.1.2 Cross-Compilation of the TIGERs camera software RobotPi on Windows and Linux for the Raspberry Pi with CMake

The compilation on the target platform Raspberry Pi is suboptimal. It requires a Raspberry Pi for compilation and is considerably slower than compilation on a current personal computer. Furthermore it makes the deployment more complex due to retrieving the compilation result from a Raspberry Pi first. Therefore the

RobotPi software should be crosscompiled from the typical personal computer x86 architecture for the Raspberry Pi with its armhf cpu architecture.

For crosscompilation a crosscompiler is required. There are multiple crosscompilers for the Raspberry Pi availible. The official Raspberry Pi crosscompiler toolchain [14] is outdated using the Gnu Compiler Collection (GCC) version 4.8.3 while the current version on the Raspberry Pi GCC is version 8.3.0. An alternative is the crosstool-ng project which is an crosscompiler generator [15]. A working crosscompiler has been generated using crosstool-ng but its use for new contributors and the added requirement of an additional step in the continouus integration pipeline has deemed it unpreferable. The toolchain provided by Pro [16] has a reasonable current GCC version and has proven working.

CMake does support crosscompiling with the `-DCMAKE_TOOLCHAIN_FILE={file}` flag. Based on [17] a first cross compiling configuration file can be created:

**Listing 4.3:** CMake-Configuration for crosscompilation

```
1  set(CMAKE_TRY_COMPILE_TARGET_TYPE "STATIC_LIBRARY")
2
3  set(CMAKE_C_COMPILER ${TOOLCHAIN}/bin/arm-linux-gnueabihf-gcc)
4  set(CMAKE_CXX_COMPILER ${TOOLCHAIN}/bin/arm-linux-gnueabihf-g++)
5
6  set(CMAKE_SYSTEM_NAME Linux)
7  set(CMAKE_SYSTEM_PROCESSOR armv7l)
8  set(CMAKE_LIBRARY_ARCHITECTURE "arm-linux-gnueabihf")
9
10 set(CMAKE_FIND_ROOT_PATH ${ROOTFS})
11
12 set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
13 set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
14 set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
15 set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

One issue arises with the various cross compilers in combination with the MMAL library. The various cross compiling toolchains come with their own root filesystem with the basic c library. These “root” directories don’t contain nonstandard libraries such as the MMAL library. A root filesystem generated as described in Section 4.2 does contain all installed libraries of the target system. The problem is the different

position of e.g. the c library: In the compiler root filesystems it can usually be found under `/usr/lib` while in the raspbian filesystem it is found under `/usr/lib/arm-linux-gnueabihf`. This prevents sole usage of one of the both file systems. Moving the libraries inside the raspbian rootfs to the expected positions doesn't succeed in the compilation due to minor differences in the libraries. Declaring multiple `CMAKE_FIND_ROOT_PATH` doesn't help either. The GCC compiler allows for multiple `--sysroot`-flags, which determin the base path. Through directly setting these flags the issue could be resolved:

**Listing 4.4:** GCC multiple sysroot options

```
1 set(CMAKE_CXX_FLAGS "--sysroot=${TOOLCHAIN}/arm-linux-gnueabihf/sysroot ${
      CMAKE_CXX_FLAGS}")
2 set(CMAKE_CXX_FLAGS "--sysroot=${ROOTFS} ${CMAKE_CXX_FLAGS}")
```

### 4.1.3 Compilation of the TIGERs camera software RobotPi on Windows and Linux for the compilation platform with CMake

For automated testing and the development of new algorithms the software should be able to run with predefined images. The software side implementation of the image loading and separation of dependencies is discussed in Section 4.4. This section focuses solely on the compilation for the development platform.

In comparison to the raspberry pi target platform the development doesn't have a bot-mounted camera and the MMAL library isn't availible. OpenCV is used instead for loading and displaying the images. Therefore the source code only used on the raspberry pi is located in the `rpi` directory while the source code only used for testing and development is located in the `pc` directory. To separate the source files in CMake an additional variable `ADDITIONAL_SOURCES` was created for the source files used only on the raspberry pi or the development platform. For the separation of the three build targets the usage of three different cross compilation toolchain files was considered. This solution was discarded because sources and dependencies declared in the cross compile configurations are ignored by CMake. The alternative is a flag

based implementation. The flag FOR_PI declares the target platform raspberry pi if set. This resulted in the following CMakeLists modification:

**Listing 4.5:** Switch for compiling for pi or development

```
1  if(DEFINED FOR_PI)
2      file(GLOB_RECURSE ADDITIONAL_SOURCES "src/rpi/*.cpp" "src/RobotPi.cpp" "
           src/tests.cpp" "src/main.cpp")
3      find_package(MMAL REQUIRED)
4  else()
5      file(GLOB_RECURSE ADDITIONAL_SOURCES "src/pc/*.cpp")
6      find_package(OpenCV REQUIRED)
7  endif()
8
9  file(GLOB_RECURSE SOURCES "src/detector/*.cpp" "src/interface/*.cpp" "src/util
       /*.cpp")
10
11 add_executable(robotpi ${SOURCES} ${ADDITIONAL_SOURCES})
12
13 target_include_directories(robotpi PRIVATE src ${MMAL_INCLUDE_DIRS} ${
       OpenCV_INCLUDE_DIRS})
14 target_link_libraries(robotpi PRIVATE ${MMAL_LIBRARIES} pthread ${OpenCV_LIBS
       })
```

### 4.1.4 Compilation of the TIGERs camera software RobotPi for the Raspberry Pi with CMake inside a GitLab pipeline

The TIGERs Mannheim are using a GitLab Community Edition instance for version control, task planning and continouus integration. For the integration possibilities and other projects therefore for continouus integration of the RobotPi project a GitLab Runner instance based on docker. Apart from via common installation routines like the deb-Package manager there are two dependencies: A PiGERs (see Section 4.2) sysroot and a crosscompiling toolchain for the raspberry pi target platform.

GitLab Runner are instances that execute the steps done during the continouus integration. To configure a GitLab Runner, a `.gitlab-ci.yml` file is required for configuration [18]. The GitLab pipeline for RobotPi consists of two steps: building RobotPi for the Raspberry Pi and packaging the RobotPi project in a debian package. For the

build process the compiler toolchain is required. The toolchain [16] is precompiled and can be downloaded directly with `curl`. The newest rootfs can be downloaded as an artifact of the pipeline described in Section 4.2.2. A private token is required since the project containing PiGERs is not publicly availible [19]. Downloading the toolchain and rootfs every time slows the build process down. The cache feature can be used to cache the downloaded files and therefore speed up the build process [20].

This results in the following `.gitlab-ci.yml` script:

**Listing 4.6:** GitLab CI configuration for RobotPi

```
1  stages:
2    - build
3    - package
4
5  variables:
6    DEBIAN_FRONTEND: "noninteractive"
7
8  build:
9    stage: build
10   image: debian:buster
11   script:
12     - ./ci-build.sh
13
14   cache:
15     key: build
16     paths:
17         - "toolchain.tar.gz"
18         - "rootfs.tar"
19
20   artifacts:
21     name: "robotpi"
22     paths:
23       - bin/robotpi
24     expire_in: 1 week
25
26 package:
27   stage: package
28   image: debian:buster
29   script:
30     - apt-get update && apt-get install -y git
31     - ./package.sh
32
33   artifacts:
34     name: "robotpi.deb"
35     paths:
```

```
36        - robotpi-*.deb
37      expire_in: 1 week
```

The script `ci-build.sh` contains the routines for downloading the required dependencies and compiling the project:

**Listing 4.7:** CI build script

```
1  #!/bin/bash
2  apt-get update && apt-get install -y --no-install-recommends cmake make curl
3
4  # Get rootfs
5  if [ ! -f rootfs.tar ]; then
6      curl -k --header "PRIVATE-TOKEN: abcdefgh" https://gitlab.tigers-mannheim.
           de/api/v4/projects/94/jobs/artifacts/master/raw/deploy/TIGERs-tigers-
           rootfs.tar?job=build -o rootfs.tar
7  fi
8  mkdir rootfs
9  tar -xf rootfs.tar -C rootfs
10
11 # Get toolchain
12 if [ ! -f toolchain.tar.gz ]; then
13   curl -kL https://github.com/Pro/raspi-toolchain/releases/download/v1.0.2/
         raspi-toolchain.tar.gz -o toolchain.tar.gz
14 fi
15 tar -xf toolchain.tar.gz
16 mv cross-pi-gcc /opt/cross-pi-gcc
17
18 cmake -DROOTFS=$(readlink -f rootfs) -DTOOLCHAIN=/opt/cross-pi-gcc -
       DCMAKE_TOOLCHAIN_FILE=CrossCompile.cmake .
19 make -j
```

### 4.1.5 Integration of the TIGERs camera software RobotPi compilation in the corresponding Eclipse project

Eclipse CDT has been chosen by the TIGERs as the main integrated development environment (IDE) for the RobotPi project. Eclipse CDT has no inbuilt CMake support but provides a Unix Makefile compilation backend [21].

CMake can create an Eclipse project with `cmake -G "Eclipse CDT4 - Unix Makefiles" .` using the Unix Makefiles backend. This method has the drawback that parameters like

the -DCMAKE_TOOLCHAIN_FILE are ignored and therefore setting the compilation target has to be done in the command line manually invoking CMake.

The cmake4eclipse plugin [22] enables Eclipse CDT to directly import the Eclipse project. It's necessary to change the build system from "Ninja" to "Unix Makefiles". cmake4eclipse allows for changing the CMake parameters in the Launch Configuration window. This method was chosen as solution due to the possibility to set up compilation targets for development and the Raspberry Pi without invoking the command line.

## 4.2 Creation of a custom Raspbian image

pi-gen is a tool for the creation of Raspbian images that can be run on most debian based operating systems [23]. The Raspbian project provides the package servers used for all Raspberry Pi platforms from Raspberry Pi 1 to Raspberry Pi 3, while for the Raspberry Pi 4 the 64 bit debian package servers are used [24]. Since this paper focuses on providing an custom image for the Raspberry Pi 3A using the Raspbian packages, the old name Raspbian is used instead of the new name Raspberry Pi OS.

### 4.2.1 Creation of a custom Raspbian image with all necessary configurations and dependencies required for RobotPi

For the TIGERs its important that the distribution of the vision software to all bots works with as little efford as possible. Since during the RoboCup no Wifi connections are allowed the software on the Raspberry Pis cannot be updated with regular means. Therefore all dependencies have to be already installed with the operating system.

pi-gen generates the Raspbian in multiple stages which depend on each other and provide more functionality with each stage. Since no graphical user environment is needed on the bots the TIGERs custom image (further called PiGERs) bases on stage 2, the Raspbian Lite stage. Stage 3 to 5 are therefore deactivated. To separate the changes for the PiGERs image and the general pi-gen for future updates a new

stage `stageVision` was generated. A stage is considered a folder with the option to contain bash scripts and other configuration files. The bash script `prerun.sh` is the first script run during a stage. This script prepares the stage environment like copying the root filesystem generated in the previous stage:

**Listing 4.8:** Minimal prerun.sh script

```
#!/bin/bash -e

if [ ! -d "${ROOTFS_DIR}" ]; then
  copy_previous
fi
```

If present the bash script `EXPORT_IMAGE` is run after a stage has completed to prepare the export of the stage in an `.img` image.

**Listing 4.9:** Minimal EXPORT_IMAGE script based on the stage 2

```
IMG_SUFFIX="-tigers"
if [ "${USE_QEMU}" = "1" ]; then
  export IMG_SUFFIX="${IMG_SUFFIX}-qemu"
fi
```

The crosscompilation of the RobotPi software like described in Section 4.1.2 requires a root file system containing headers and binaries of the necessary libraries. During the image export a tar archive containing the necessary files can be created by expanding the `EXPORT_IMAGE` script:

**Listing 4.10:** Exporting the rootfs

```
ROOTFS_DIR_BAK=${ROOTFS_DIR}
ROOTFS_DIR=${WORK_DIR}/$(basename "${EXPORT_DIR}")/rootfs

RFS_FILE="${DEPLOY_DIR}/${IMG_FILENAME}${IMG_SUFFIX}-rootfs.tar"
rm -f "${RFS_FILE}"

on_chroot << EOF
tar -hcf rootfs.tar etc opt usr/include usr/lib usr/local/include usr/local/
    lib
EOF
unmount ${WORK_DIR}/$(basename "${EXPORT_DIR}")
```

```
11
12 mkdir -p ${DEPLOY_DIR}
13 mv ${ROOTFS_DIR}/rootfs.tar ${RFS_FILE}
14
15 ROOTFS_DIR=${ROOTFS_DIR_BAK}
```

For the installation of additional packages for development, a folder `00-install-packages` with a file `00-packages` is created. The file contains the names used by the apt package manager. Each package named in the packages file will then be installed into the image by pi-gen. The installation and usage of OpenCV on the Raspberry Pi even for testing purposes was discarded since OpenCV requires a complete graphical environment. This resulted in an additional installation size of 1.2 GiB.

The RobotPi software requires the primary UART interface and the Raspberry Pi camera interface. Both of these interfaces have to be enabled first. Since the usual way of configuring these interfaces by running the program `raspbi-config` is not applicable here, the file `/boot/config.txt` has to be edited [25, 26]. The options that have to be set are not present in the configuration file by default. Therefore the options can be appended at the end of the file inside the `prerun.sh` script:

**Listing 4.11:** Activating camera and primary UART interface

```
1 echo "
2 start_x=1
3 gpu_mem=256
4 enable_uart=1
5 dtoverlay=disable-bt
6 " >> "${ROOTFS_DIR}/boot/config.txt"
```

The resulting image is with a size of over 1,8 GiB relative large, which a search for the largest unneccessary packages was done. With the bash command `dpkg-query -Wf '${Installed-Size}\t${Package}\n' | sort -n` [27] the largest deb packages could be found. Most of these packages are installed as a dependency of other packages. Therefore it's necessary to determin the manual packages with the command `comm -23 <(apt-mark showmanual | sort -u)<(gzip -dc /var/log/installer/initial-status.gz | sed -n 's/^Package: //p' | sort -u)` [28]. Serveral unused packages were determined. The size in mebibyte is the freed size including uninstalled dependencies.

- `libraspberrypi-doc` 33.8 MiB
- `gfortran-8` 22.2 MiB
- `iso-codes` 19.9 MiB
- `python3-picamera` 18.1 MiB
- `python2` 15.3 MiB
- `libglib2.0-data` 8.7 MiB
- `geoip-database` 8.1 MiB

For reducing the size of the image by uninstalling unused packages two procedures have been considered. The list of packages to install can be reduced. This results in a faster image generation. Alternativly the packages can be uninstalled during the `stageVision`. This doesn't require changes to the default pi-gen stages. The second option was chosen since its easier to update to upcoming pi-gen versions if the pi-gen default stages are unmodified. For uninstalling the packages the following lines were added to the `prerun.sh` script:

**Listing 4.12:** Removing unused packages

```
1 on_chroot << EOF
2 apt-get purge -y libraspberrypi-doc gfortran-8 iso-codes python3-picamera
      python-rpi.gpio python libglib2.0-data geoip-database
3 apt autoremove -y
4 EOF
```

### 4.2.2 Integration of the creation of the custom Raspbian images in a GitLab pipeline

Pi-Gen can only be run on Debian based linux distributions. For non Debian users and other purposes like crosscompiling RobotPi its therefore required to provide a root filesystem and the image ready to download. This is done with continouus integration through GitLab runners.

The TIGERs use a Docker executor for the pipelines. Standard docker executors don't allow modification of the linux kernel to provide containerisation. pi-gen requires access to the linux kernel for binformat-misc and qemu to execute ARM programs [23]. For access to the kernel the Docker executor has to be run in priviledged mode. The TIGERs don't have a priviledged runner set up, therefore a new GitLab runner instance has been set up in the context of this work. The `curl` calls of pi-gen fail silently inside the Docker environment due to outdated ssl certificates. The command calls have to be modified to use the `-k` option. Another more secure option would be to provide own docker images which was deemed out of scope for this work. The resulting `.gitlab-ci.yml` configuration file only has to install required dependencies and run the main build script:

**Listing 4.13:** CI configuration for PiGERs

```
1  stages:
2    - build
3
4  variables:
5    DEBIAN_FRONTEND: "noninteractive"
6
7  build:
8    stage: build
9    image: i386/debian:buster
10   tags:
11     - privileged
12
13   script:
14     - apt-get update && apt-get install -y --no-install-recommends quilt
           parted debootstrap zerofree zip dosfstools bsdtar rsync xz-utils curl
           xxd file git kmod bc libcap2-bin qemu-user-static binfmt-support
15     - ./build.sh
16   artifacts:
17     name: "$CI_COMMIT_REF_NAME"
18     paths:
19       - deploy/
20     expire_in: 1 week
```

## 4.3 Implementing a deployment method for the TIGERs camera software RobotPi

In the environment of the RoboCup competition wireless connections are strictly controlled. Therefore for patching the camera software RobotPi Wifi connections cannot be used. The package manager apt used by Raspbian to install packages and resolve dependencies cannot be used due to the missing internet connectivity. An alternative method is needed to be able to update the software during games.

The operating system Raspbian is based on the Debian distribution [29]. The Debian distribution uses the .deb file format for installation of software [30]. Therefore its a obvious choice to use the .deb file format for distribution of the RobotPi software. For creation of a debian package only a file under `[package-name]/DEBIAN/control` is required, all other directories except the `DEBIAN` directory are copied into the root directory on installation [31].

**Listing 4.14:** The RobotPi control file

```
1  Package: robotpi
2  Version: 1.0
3  Section: custom
4  Priority: optional
5  Architecture: armhf
6  Essential: no
7  Installed-Size: 1024
8  Maintainer: tigers-mannheim.de
9  Description: Use the pi camera as sensor for the TIGERs Mannheim bots.
```

For the copying of the binary into the package and versioning of the package a short bash script was created:

**Listing 4.15:** The script creating the RobotPi .deb package

```
1  #!/bin/bash
2  mkdir robotpi/usr/bin -p
3  cp bin/robotpi robotpi/usr/bin
4
5  chmod -R =775 robotpi/DEBIAN
```

```
6  dpkg-deb --build robotpi robotpi-$(grep "Version" robotpi/DEBIAN/control | cut
       -d " " -f 2 -)-armhf-$(git rev-parse --short HEAD).deb
7
8  rm -r robotpi/usr
```

The RobotPi software should start to run after installation and on system startup. Raspbian uses systemd for initialisation of services and deamons [32]. systemd uses service files to describe the service file to execute. A requirement for enabling a systemd service is an `Install` section with a unique alias defined [33]. This results in the following service file for RobotPi:

**Listing 4.16:** The RobotPi systemd service file

```
1  [Unit]
2  AssertPathExists=/usr/bin/robotpi
3
4  [Service]
5  ExecStart=/usr/bin/robotpi
6  Restart=always
7
8  [Install]
9  Alias=robotpi
10 WantedBy=multi-user.target
```

For inclusion into the .deb package, the service file has to be located in `lib/systemd/system` inside the debian package folder. The Debian package format allows the definition of scripts that are run during the installation [34, pp. 86–87]. For enabling and starting the RobotPi service the `DEBIAN/postinst` script is utilized.

**Listing 4.17:** The RobotPi postinst script

```
1  #!/bin/bash
2
3  systemctl enable robotpi
4  systemctl start robotpi
```

For cleanup prior to uninstallation or an upgrade the `DEBIAN/prerm` script is used.

**Listing 4.18:** The RobotPi prerm script

```
1 #!/bin/bash
2
3 systemctl stop robotpi
4 systemctl disable robotpi
```

For distribution of the RobotPi software the USB port of the Raspberry Pi 3A is used. Since a network connection is not availible to the onboard Raspberry Pi this seemed to be the most accessible way to install updates. The intended workflow is to plug a USB stick into the USB port which leads to an automated installation of all .deb packages on the USB stick.

To mount the usb stick (making its contents as child nodes of the root directory accessible) on insertion various methods have been proposed [35, 36]. The package usbmount has been chosen as solution since it seemes to be a ready out of the box implementation. One modification to the system is necessary for usbmount to work. The option `PrivateMounts` in systemd has to be disabled [37].

The last step is the installation of the .deb packages on the usb stick after insertion. Monitoring the mounting directory with inotify[38] did not work since through the mounting process no files are changed or created. systemd has the capability to execute services once a directory is mounted [39]. The following service file triggers once a usb stick is mounted at `/media/usb0`:

**Listing 4.19:** systemd service file to run post usb drive mounting

```
1  [Unit]
2  Description=Autoinstaller of .deb-Packages
3  Requires=media-usb0.mount
4  After=media-usb0.mount
5
6  [Service]
7  ExecStart=/usr/bin/autoinstall
8
9  [Install]
10 WantedBy=media-usb0.mount
```

The autoinstall script contains a line to install all .deb files:

**Listing 4.20:** The script installing all .deb packages on the usb stick

```
1 #!/bin/bash
2
3 find /media/usb0 -name *.deb -exec dpkg -i {} \;
```

To provide the installation capability with a fresh install the installation method has to be included in the TIGERs Raspbian image. This is done on top of the changes described in Section 4.2. Inside stageVision, a new folder `01-autoinstall` was created, containing the above described systemd configuration and run script in the subfolder `files`. The script `00-run.sh` installs the script in the image:

**Listing 4.21:** The script installing the usb installer in the Raspbian image

```
1 #!/bin/bash -e
2
3 sed -i 's/PrivateMounts=yes/PrivateMounts=no/' "${ROOTFS_DIR}/lib/systemd/
      system/systemd-udevd.service"
4 install -m 755 files/autoinstall "${ROOTFS_DIR}/usr/bin/autoinstall"
5 install files/autoinstall.service "${ROOTFS_DIR}/lib/systemd/system/
      autoinstall.service"
6 mkdir -p "${ROOTFS_DIR}/etc/systemd/system/media-usb0.mount.wants/"
7 ln -s /lib/systemd/system/autoinstall.service "${ROOTFS_DIR}/etc/systemd/
      system/media-usb0.mount.wants/autoinstall.service"
```

## 4.4 Implementing the loading images in the camera software RobotPi for testing and development

The RobotPi software is intended for image processing on the Raspberry Pi with live captured images. For testing and development the possibility to run the software on the development platform without camera is required. In Section 4.1.3 is the process of separating the compilation targets for Raspberry Pi and development platform. This section focuses on the code side for development.

First for loading the images into a C readable format the reference implementation for the png file format libpng [40] was considered. Due to the complexity in comparison to other solutions (e.g. it's required to provide a malloc function for allocating memory)

the usage of libpng was dropped. The usage of OpenCV was deemed acceptible since the loading of images needs only to work on the development platform. OpenCV further can be used to show debug images during runtime.

OpenCV allows for reading various file formats into the OpenCV data structure `cv::Mat` with the call `cv::imread(char* path)` [41]. Most image formats provide images in the RGB color space. It's assumed that the input data is in RGB. RobotPi internally uses the color space of the pi camera, the YUV color space. Therefore the data has to be transformed. OpenCV provides the function for transforming the color space `cv::cvtColor`. The `YUVFrame` internally used by RobotPi is in the YUV420 format. Only for every fourth pixel exists a new UV color value. Due to the noise which comes from the pi camera only every fourth color value is used for the transformation between the `cv::Mat` and the `YUVFrame`. Therefore the frame can be constructed:

**Listing 4.22:** The function reading a file into a YUVFrame

```
1  FrameYUV420 imageByPath(const char *path)
2  {
3      cv::Mat image = cv::imread(path);
4
5      FrameMetadata meta;
6      meta.width = image.size[1];
7      meta.height = image.size[0];
8
9      cv::Mat yuvImg (meta.height, meta.width, CV_8UC3);
10     cv::cvtColor(image, yuvImg, cv::COLOR_BGR2YUV);
11
12     int size = image.size[0] * image.size[1];
13     uint8_t* yData = new uint8_t[size];
14     uint8_t* uData = new uint8_t[size/4];
15     uint8_t* vData = new uint8_t[size/4];
16
17     int halfWidth = meta.width/2;
18     for(int y = 0; y < meta.height; y++)
19     {
20         for(int x = 0; x < meta.width; x++)
21         {
22             cv::Vec3b pixel = yuvImg.at<cv::Vec3b>(y,x);
23             yData[y*meta.width+x] = pixel[0];
24             uData[y/2*halfWidth+x/2] = pixel[1];
25             vData[y/2*halfWidth+x/2] = pixel[2];
26         }
27     }
```

```
28
29      return FrameYUV420(yData, uData, vData, meta);
30  }
```

The function for the other way round works accordingly in inverted order.

## 4.5 Simulating images taken by the on board camera on a RobotCup Small Size League Division A Field

The TIGERs don't have access to a full sized RoboCup Small Size League Division A field. This limits the possibilities to take images for development and testing. Therefore images have to be simulated. Blender is an open source 3d creation suite an as such capable of simulating the images from the perspective of the bot [42]. The simulated images generated in the scope of this work are not a physically accurate simulation.

The 3d mesh, the geometry of the playing field, has to be created according to the rules [43]. The walls, goals, lines, field and bot have to be shaded according to a correct color. To give the field the optic of a carpet a normal map with random noise as input was applied. For simulating the properties of the pi camera, a small gaussian filter is applied and the color spectrum is reduced to a "filmic very low contrast" setting.

It's important for the usage of the 3d model as source of simulated images to be able to choose the position of the camera freely. Blender has the option to execute a Python script at startup. This can be used to render the image from a predefined position [44]. To render a image in blender with the headless mode a large range of command line options are required: The headless mode has to be set (`-b`), the blend file loaded (`blendfile.blend`), the script file loaded (`-P render.py`), the output file specified (`-o render.png`), the frame to render given (`-f 0`) and the camera settings provided for the script (`-- x y rotation`). Some of the options can be set with the Python api such as loading the blend file, specifing the output file and rendering the image. This

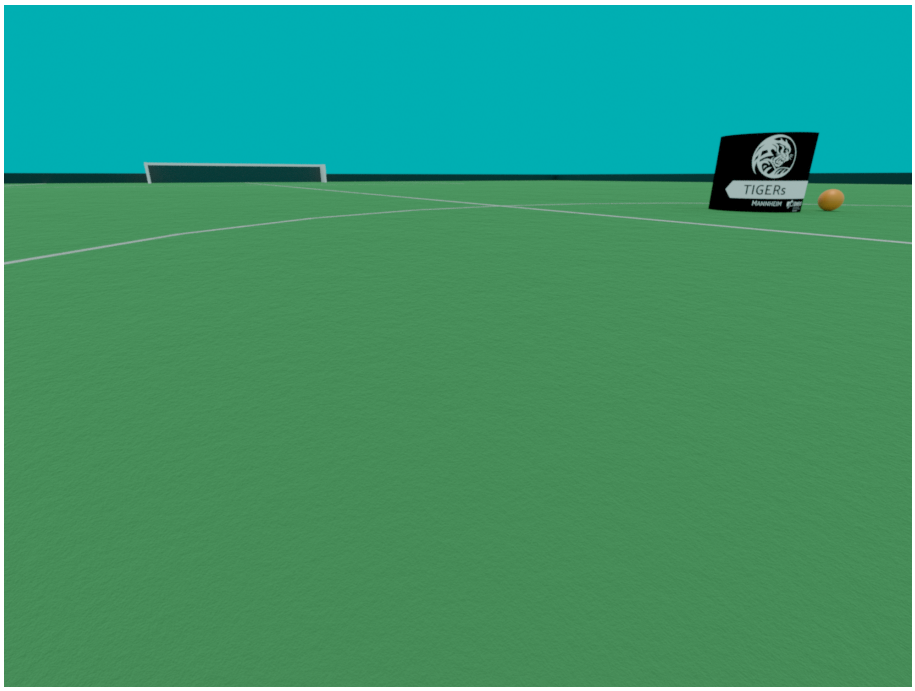


**Figure 4.1:** Example simulated image

shortens the necessary command line call to `blender -b -P render.py -- x y rotation` and results in the following python script:

**Listing 4.23:** Blender python script for rendering the images

```
1  import sys, math, argparse, bpy
2
3  scene_loaded = any([not arg.startswith('-') for arg in sys.argv[1:sys.argv.
       index('-P')]])
4
5  if not scene_loaded:
6      try:
7          bpy.ops.wm.open_mainfile(filepath="SSLAField.blend")
8      except RuntimeError:
9          print("ERROR: No scene loaded prior to script execution and could not
               find SSLAField.blend!")
10         sys.exit(0)
11
12 parser = argparse.ArgumentParser()
13 parser.add_argument("x", type=float, help="x position of the bot camera")
14 parser.add_argument("y", type=float, help="y position of the bot camera")
15 parser.add_argument("r", type=float, help="rotation of the bot in degrees")
```

```
16
17 try:
18     argv = sys.argv[sys.argv.index('--')+1:]
19     args = parser.parse_args(argv)
20 except ValueError:
21     print("ERROR: You need to add \" -- \" prior to the script arguments to
           separate blender and script arguments!")
22     sys.exit(0)
23
24 cam = bpy.data.scenes["Scene"].camera
25
26 cam.location.x = args.x
27 cam.location.y = args.y
28 cam.rotation_euler[2] = args.r * math.pi/180
29
30 if '-o' not in sys.argv:
31     bpy.context.scene.render.filepath = 'render.png'
32
33 if '-a' not in sys.argv and '-f' not in sys.argv:
34     bpy.ops.render.render(write_still=True)
```

# 5 Result

1. Compilation of the TIGERs camera software RobotPi with CMake: Successful completed
   - 1.1 Compilation of the TIGERs camera software RobotPi on the Raspberry Pi with CMake: Successful completed
   - 1.2 Cross-Compilation of the TIGERs camera software RobotPi on Windows and Linux for the Raspberry Pi with CMake: Successful completed
   - 1.3 Compilation of the TIGERs camera software RobotPi on Windows and Linux for the compilation platform with CMake: Successful completed
   - 1.4 Compilation of the TIGERs camera software RobotPi for the Raspberry Pi with CMake inside a GitLab pipeline: Successful completed
   - 1.5 Integration of the TIGERs camera software RobotPi compilation in the corresponding Eclipse project: Successful completed
2. Creation of a custom Raspbian image: Successful completed
   - 2.1 Creation of a custom Raspbian image with all necessary configurations and dependencies required for RobotPi: Successful completed
   - 2.2 Integration of the creation of the custom Raspbian images in a GitLab pipeline: Successful completed
3. Implementing a deployment method for the TIGERs camera software RobotPi: Successful completed

4. Implementing the loading images in the camera software RobotPi for testing and development: Successful completed
5. Simulating images taken by the on board camera on a RobotCup Small Size League Division A Field: Successful completed

# 6 Discussion

CMake is an overly complicated tool for compilation tasks. The advances in dependency management and compilation tooling in the recent years make it unbelievable how difficult the compilation of a non-trivial C or C++ program like RobotPi is. As comparison I would like to exhibit the relativly new Rust programming language, whose default compiler rustup has inbuilt cross-compilation support [45]. The CMake compilation chain for RobotPi created in this work has the slight drawback that for compiling for the production target the command line option `-DFOR_PI=1` has to be passed. Alternative approches like three different cross-compilation files didn't work out due to restrictions what values can be set in the toolchain files.

The GitLab pipelines work generally as planned, but there might be caching issues in the RobotPi pipeline since the RobotPi pipeline caches the root filesystem from the PiGERs pipeline. This cache can be cleared manually but is never cleared automatically. Sadly the pi-gen tool used to create the PiGERs images works only on Debian based linux distribution which might hinder contributors. This drawback is slightly compensated through the possibility to push the changes to the TIGERs GitLab which triggers an image build with the GitLab runner.

The simulation in Blender works reasonably well. There are some slight issues with the circle in the center of the play field due to the overlap with the carpet. Currently the simulation uses a perspective projection in comparison to the fish eye projection of the pi camera lens. Blender has an option for fish eye rendering but we could not measure the realism of the deformation model against the pi camera model. Therefore we cannot test the projection inversion of RobotPi with simulated images.

# 7 Prospects

This work should be a good basis for the development of global position detection and other image detection algorithms. It might be useful to change away from Eclipse CDT as default IDE since Eclipse hasn't an inbuilt CMake support and issues detecting the availible header files. Future work on the development and deployment infrastructure might go into the deployment process of the RobotPi software to the Raspberry Pi in a development environment where the availibilty of wireless network connections can be assumed. A speedup on the GitLab pipelines is imaginable investigating into the possibilities of applying more caching e.g. saving the working directory of pi-gen. The tradeoff between speedup through caching and potential conflicts through outdated files or results has to be considered. Currently the pipelines between the PiGERs project and the RobotPi project are running independently. This requires manual clearing of the RobotPi cache and starting of a new build process for RobotPi to use the new PiGERs root filesystem. It might be useful to consider triggering the RobotPi pipeline on completion of the PiGERs pipeline on the master branch.

In the image simulation the bot model could be overhauled to allow differently styled bot styles and a more realistic front side. The python script might be expanded to allow other freely positioned bots and balls. A change from taking the camera position to taking the center bot position might be useful. Currently the simulation misses the environment around the play field, which might contain artifacts that can disturb image recognition software and should therefore be considered during image simulation. The lighting of the play field is usually not so uniform like currently in the simulation, it might be better to create a more challenging lighting situation to be perpared for all circumstances. Many properties of the simulation are currently

estimated or approximated, it might be useful measure the correct physical properties and adust the simulation to provide more realistic images.

# Bibliography


[1] RoboCup Federation. *Objective*. 2016. URL: https://www.robocup.org/objective (visited on 02/23/2021).

[2] RoboCup Federation. *About RoboCup Small Size League*. URL: https://ssl.robocup.org/about/ (visited on 03/03/2021).

[3] Stefan Zickler et al. *SSL-Vision: The Shared Vision System for the RoboCup Small Size League*. 2009. URL: http://www.informatik.uni-bremen.de/agebv2/downloads/published/zickler_rs_09.pdf (visited on 03/03/2021).

[4] RoboCup Small Size League Technical Committee. *SSL-Vision Blackout Technical Challenge*. 2018. URL: https://github.com/RoboCup-SSL/technical-challenge-rules/releases/download/2019-v1.0/ssl-vision-blackout-technical-challenge.pdf (visited on 03/03/2021).

[5] RoboCup Small Size League Technical Committee. *RoboCup 2020 SSL Vision Blackout Technical Challenge Rules*. 2020. URL: https://ssl.robocup.org/wp-content/uploads/2020/07/2020-ssl-vision-blackout-rules.pdf (visited on 03/03/2021).

[6] Andre Ryll and Sabolc Jut. *TIGERs Mannheim Extended Team Description for RoboCup 2020*. 2020. URL: https://ssl.robocup.org/wp-content/uploads/2020/03/2020_ETDP_TIGERS.pdf (visited on 03/03/2021).

[7] Sabolc Jut and Fabio Seel. *On-Board Computer Vision for Autonomous Ball Interception*. 2019. URL: https://tigers-mannheim.de/download/papers/2019-BallIntercept_TC-Seel_Jut.pdf (visited on 03/03/2021).

[8] Kitware Inc. *Overview*. URL: https://cmake.org/overview/ (visited on 03/05/2021).

[9] corob-msft et al. *CMake projects in Visual Studio*. 2020. URL: https://docs.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio?view=msvc-160 (visited on 03/05/2021).

[10] Kitware Inc. and Contributors. *CMake Tutorial*. URL: https://cmake.org/cmake/help/v3.20/guide/tutorial/index.html (visited on 03/09/2021).

[11] Dr. Derek Molloy. *Introduction to CMake by Example*. 2015. URL: http://derekmolloy.ie/hello-world-introductions-to-cmake (visited on 03/09/2021).

[12] *Multi-Media Abstraction Layer (MMAL). Draft Version 0.1.* 2015. URL: http://www.jvcref.com/files/PI/documentation/mmal_10_2015/html/ (visited on 03/10/2021).

[13] Team Kodi. *FindMMAL.cmake*. URL: https://searchcode.com/file/115640357/project/cmake/modules/FindMMAL.cmake/ (visited on 05/01/2021).

[14] Raspberry Pi Foundation. *raspberrypi/tools*. 2020. URL: https://github.com/raspberrypi/tools (visited on 03/10/2021).

[15] Alexey Neyman et al. *Crosstool-NG*. 2021. URL: https://github.com/crosstool-ng/crosstool-ng (visited on 03/10/2021).

[16] Stefan Profanter. *Raspberry PI Toolchains*. 2021. URL: https://github.com/Pro/raspi-toolchain (visited on 03/10/2021).

[17] Kitware Inc. and Contributors. *cmake-toolchains(7)*. URL: https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html (visited on 03/15/2021).

[18] Marcel Amirault and Suzanne Selhorn. *The .gitlab-ci.yml file*. 2021. URL: https://docs.gitlab.com/ce/ci/yaml/gitlab_ci_yaml.html (visited on 05/01/2021).

[19] thekucays and Xavier D. *The .gitlab-ci.yml file*. 2019. URL: https://stackoverflow.com/questions/56233243/gitlab-ci-get-last-artifact (visited on 05/01/2021).

[20] Marcel Amirault and Suzanne Selhorn. *cache*. 2021. URL: https://docs.gitlab.com/ce/ci/yaml/README.html#cache (visited on 05/01/2021).

[21] gvd and Martin Gerhardy. *How to configure Eclipse CDT for cmake?* 2012. URL: https://stackoverflow.com/questions/9453851/how-to-configure-eclipse-cdt-for-cmake (visited on 05/01/2021).

[22] Martin Weber. *cmake4eclipse*. 2021. URL: https://marketplace.eclipse.org/content/cmake4eclipse (visited on 05/01/2021).

[23] RPi-Distro. *pi-gen*. 2021. URL: https://github.com/RPi-Distro/Pi-gen (visited on 04/01/2021).

[24] Avram Piltch. *Raspberry Pi OS: Why It's No Longer Called 'Raspbian'*. 2020. URL: https://www.tomshardware.com/news/raspberry-pi-os-no-longer-raspbian (visited on 04/01/2021).

[25] *Boot options in config.txt*. 2021. URL: https://www.raspberrypi.org/documentation/configuration/config-txt/boot.md (visited on 04/01/2021).

[26] Raspberry Pi Foundation. *Device Trees, overlays, and parameters*. 2021. URL: https://www.raspberrypi.org/documentation/configuration/device-tree.md#part4.6 (visited on 04/01/2021).

[27] raspi. *List installed deb packages by size*. 2009. URL: https://www.commandlinefu.com/commands/view/3842/list-your-largest-installed-packages-on-debianubuntu (visited on 04/08/2021).

[28] jmiserez. *Generating list of manually installed packages and querying individual packages*. 2014. URL: https://askubuntu.com/questions/2389/generating-list-of-manually-installed-packages-and-querying-individual-packages (visited on 04/08/2021).

[29] Raspbian.org. *Welcome to Raspbian*. URL: https://www.raspbian.org/ (visited on 04/21/2021).

[30] Debian Wiki team. *Debian package*. 2019. URL: https://wiki.debian.org/deb (visited on 04/21/2021).

[31] Mithil Poojary. *How To Make A .deb For Your Program*. 2020. URL: https://dev.to/mithil467/how-to-make-a-deb-for-your-program-3n0d (visited on 04/21/2021).

[32] Raspberry Pi Foundation. *systemd*. URL: https://www.raspberrypi.org/documentation/linux/usage/systemd.md (visited on 04/21/2021).

[33] Gabriel. *Automatically enable systemd services installed using deb*. 2018. URL: https://unix.stackexchange.com/questions/274624/automatically-enable-systemd-services-installed-using-deb (visited on 04/21/2021).

[34] Raphaël Hertzog and Roland Mas. *The Debian Administrator's Handbook*. 2020. URL: https://www.debian.org/doc/manuals/debian-handbook/index.en.html (visited on 04/21/2021).

[35] pauliucxz. *Auto mount USB stick on plug-in without UUID*. 2017. URL: https://raspberrypi.stackexchange.com/questions/66169/auto-mount-usb-stick-on-plug-in-without-uuid/66324#66324 (visited on 04/21/2021).

[36] Foo Bar. *How to automatically mount an USB device on plugin-time on an already running system?* 2014. URL: https://unix.stackexchange.com/questions/134797/how-to-automatically-mount-an-usb-device-on-plugin-time-on-an-already-running-sy (visited on 04/22/2021).

[37] Greg. *Raspberry 4 usbmount not working*. 2019. URL: https://raspberrypi.stackexchange.com/questions/100312/raspberry-4-usbmount-not-working (visited on 04/22/2021).

[38] DJKUhpisse. *inotify*. 2020. URL: https://wiki.ubuntuusers.de/inotify/ (visited on 04/22/2021).

[39] koichirose. *Systemd service to run a script when a USB HDD is plugged in*. 2017. URL: https://unix.stackexchange.com/questions/396519/systemd-service-to-run-a-script-when-a-usb-hdd-is-plugged-in (visited on 04/22/2021).

[40] Cosmin Truta et al. *libpng*. 2019. URL: http://www.libpng.org/pub/png/libpng.html (visited on 05/01/2021).

[41] OpenCV team. *Getting Started with Images*. 2021. URL: https://docs.opencv.org/3.4/db/deb/tutorial_display_image.html (visited on 05/01/2021).

[42] Blender Foundation. *About*. URL: https://www.blender.org/ (visited on 04/29/2021).

[43] RoboCup Small Size League Technical Committee. *Rules of the RoboCup Small Size League*. URL: https://robocup-ssl.github.io/ssl-rules/sslrules.pdf (visited on 04/29/2021).

[44] roho. *How to move a camera in Blender 2.61 with Python*. 2012. URL: https://stackoverflow.com/questions/8865672/how-to-move-a-camera-in-blender-2-61-with-python (visited on 05/01/2021).

[45] Eric Huss and Daniel Silverstone. *Cross-Compilation*. 2020. URL: https://rust-lang.github.io/rustup/cross-compilation.html (visited on 05/01/2021).