PROJEKTARBEIT

facepi - automatisierte Maskenerkennung

Ausgeführt von:

Adamicki Karolina
Behringer Michael
Brilmayer Paul
Löhr Max
Windrath Lukas

Heidenheim, am 17.02.2022

Inhaltsverzeichnis

1. Idee & Anforderungen

1.1. Idee

Für das Projekt mit dem Raspberry Pi war uns von Anfang an klar, dass wir etwas mit einer Kamera und Bilderkennung machen möchten. Da die Maskenpflicht bereits seit ca. zwei Jahren in Deutschland besteht, hat sich dieses Objekt zur Erkennung perfekt geeignet.

1.2. Anforderungen

Es sollte auf den Raspberry Pi eine Bilderkennung programmiert werden, die anzeigt, ob der Nutzer eine Maske trägt oder nicht. Außerdem wird bei fehlender Maske ein Alarm mithilfe eines Rainbow HAT ausgelöst. Wird die Maske wieder aufgesetzt, soll der Alarm automatisch ausgehen. Außerdem wird auf der 15-Segment-Anzeige die Anzahl der Personen auf dem Bild mit und ohne Maske angezeigt.

Die Bilderkennung soll auf einer Webseite angezeigt werden können. Entsprechende Buttons können dabei die Übertragung pausieren, fortführen und Screenshots erstellen sowie den Alarm des Rainbow HAT manuell ausschalten.

Für den Raspberry Pi sollte eine Halterung mit einem 3D-Drucker gedruckt werden, bei der sich die Neigung der Kamera einstellen lässt.

2. Umsetzung

2.1. Vorbereitung des Raspberry Pi

2.1.1. Raspberry aufsetzen

Zunächst wird der Installer auf https://www.raspberrypi.com/software/ herunterladen. Die Micro-SD-Karte wird in den Computer eingesteckt und die Executables werden ausgeführt. Dabei muss **OS wählen** gedrückt werden und **Raspberry Pi OS (64-bit)** sowie die SD-Karte ausgewählt werden. Nachdem die SD-Karte installiert wurde, wird diese in den Raspberry gesteckt und der Raspberry Pi wird gestartet. Die Setup Schritte werden durchgeführt und folgende Anweisungen ins Terminal geschrieben:

```
sudo apt update
sudo apt upgrade
```

2.1.2. Kamera einrichten

Die Kamera muss in den CAMERA-Slot des Raspberry Pi eingesteckt werden und aktiviert werden. Wir haben die AZ-Delivery Kamera verwendet. Dafür zunächst

```
sudo raspi-config
```

in die Konsole eingeben. Danach unter **3. Interface Options** die Kamera unter **I1 Legacy Camera** aktivieren.
Zum Testen kann ein simples Python Programm ausgeführt werden (allerdings nur auf PiOS 32):

```
from picamera import PiCamera
from time import sleep

camera = PiCamera()
camera.capture('./picture.jpg')
```

Damit wird ein Bild mit dem Namen picture.jpg in dem Ordner des Python Programms erstellt.

2.1.3. OpenCV installieren

Zuerst wurde vorsichtshalber noch einmal die aktuelle Version von PiOs überprüft → 64 Bit. Außerdem muss auch auf den RAM geachtet werden (bei einem Raspberry Pi mit 2 GB RAM müsste in der zram Konfigurationsdatei der Auslagerungsbereich erhöht werden). Da wir aber einen Raspberry Pi mit 4 GB RAM verwendet haben, hatten wir das Problem nicht.

Als nächstes wird die EEPROM-Version überprüft:

```
sudo rpi-eeprom-update
```

und falls nötig, mit folgendem Befehl aktualisiert:

```
sudo rpi-eeprom-update -a
sudo reboot
```

OpenCV sollte nicht mit `pip3` oder `sudo apt-get install` installiert werden, da hier die Versionen nicht die aktuellen sind. Somit wird OpenCV 4.5 mit dem Quellcode installiert.
Unter “Raspberry Pi Configuration” → “Performance” → GPU Memory muss mindestens 128 MByte eingestellt sein.
Zuerst wird mit dem Befehl

```
free -m
```

überprüft, ob zumindest 6.5 GB Speicherplatz verfügbar ist.
Dann wird OpenCV mit folgenden Befehlen installiert:

```
$ wget
https://github.com/Qengineering/Install-OpenCV-Raspberry-Pi-64-
bits/raw/main/OpenCV-4-5-5.sh
$ sudo chmod 755 ./OpenCV-4-5-5.sh
$ ./OpenCV-4-5-5.sh
```

2.1.4. TensorFlow 2.7 installieren

Tensorflow ist eine Bibliothek, die für machine learning erstellt wurde und somit für die Maskenerkennung geeignet ist.

Zunächst muss das tensorflow-io-gcs file system installiert werden:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install git python3-pip
$ git clone -b v0.23.1 --depth=1 --recursive
https://github.com/tensorflow/io.git
$ cd io
$ python3 setup.py -q bdist_wheel --project tensorflow_io_gcs_filesystem
$ cd dist
$ sudo -H pip3 install tensorflow_io_gcs_filesystem-0.23.1-cp39-cp39-
linux_aarch64.whl
$ cd ~
```

Anschließend wird folgendes installiert:

```
$ pip3 list | grep numpy
$ sudo -H pip3 install numpy==1.19.5
$ python3 -m pip install termcolor
$ sudo -H pip3 install gdown
$ gdown https://drive.google.com/uc?id=1FdVZ1kX5QZgWk2SSgq31C2-CF95QhT58
$ sudo -H pip3 install tensorflow-2.7.0-cp39-cp39-linux_aarch64.whl
```

2.2. Maskenerkennung nach Modell

Zunächst haben wir ein im Tutorial erstelltes Maskenerkennungs-Modell auf den Raspberry Pi zum Laufen gebracht und als Grundlage für unsere angepasste Maskenerkennung verwendet.

Als Tutorial haben wir folgende Webseite verwendet: https://www.tomshardware.com/how-to/raspberry-pi-face-mask-detector

Unser gesamter Code zur Maskenerkennung mit beschreibenden Kommentaren ist im Git zu finden (siehe **4**), die Beschreibung des Rainbow HAT ist unter **2.2.1** und die der Webseite unter **2.4** aufgeführt.

2.2.1. Rainbow HAT

Hier wird auf die Programmierung des Rainbow HAT eingegangen. Der Rainbow HAT wird im **main.py** programmiert und besitzt die Funktion, die Anzahl der Personen mit und ohne Maske im Bild anzuzeigen. Außerdem löst der Rainbow HAT einen Alarm aus, wenn jemand im Bild keine Maske trägt.

Für den Zähler werden die Variablen withCounter und withoutCounter angelegt und auf 0 gesetzt. Wird nun eine Person mit oder ohne Maske erkennt, wird der Wert dem entsprechend hochgezählt. Der Alarm (Buzzer) löst in der else-Bedingung aus, also dann, wenn jemand keine Maske trägt und der Button zum Alarm ausschalten auf der Webseite nicht betätigt wurde.

```
withCounter = 0
withoutCounter = 0
for (box, prediction) in zip(boxes, predictions):
    (startX, startY, endX, endY) = box
    (mask, withoutMask) = prediction

     if mask > withoutMask:
         label = "Thank You. Mask On."
         color = (0, 255, 0)
         withCounter+=1

     else:
         label = "No Face Mask Detected"
         color = (0, 0, 255)
         withoutCounter+=1
         if(isAlarmActive and isVideoFeedActive):
             rainbowhat.buzzer.midi_note(60, 1)
```

2.3. Selbst programmierte KI

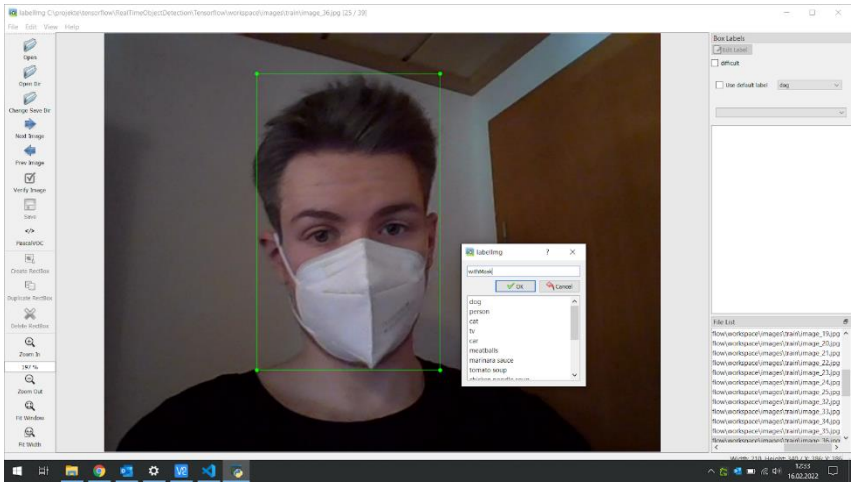
Das Wichtigste bei einer guten KI sind die Trainingsbilder. Dabei ist zu beachten:

- je mehr gute Bilder desto besser
- die Bilder sollen verschiedene Menschen zeigen, unterschiedliche Lichtverhältnisse und Posen → so allgemein wie möglich

Nachdem die Bilder gesammelt wurden, muss nun zu jedem Bild, auf dem das Gesicht mit oder ohne Maske zu sehen ist, eine Box gezeichnet werden.
Dazu kann z.B. folgendes Python-Script von tzutalin benutzt werden:

```
git clone https://github.com/tzutalin/labelImg
pyrcc5 -o libs/resources.py resources.qrc
python labelImg.py
```

Mit openDir wird das Verzeichnis mit den Bildern ausgewählt.



Hierbei ist genaue Arbeit sehr wichtig, da dies direkte Auswirkungen auf die Qualität des resultierenden Modells hat.

Das Vorgehen: die Person mit/ohne Maske wird ausgewählt und das Tag withMask oder withoutMask hinzugefügt.
Anschließend werden 80 % der Bilder in ein Train-Verzeichnis kopiert und der Rest in ein Test-Verzeichnis.

Im nächsten Schritt wird eine labelMap-Datei erstellt, welche alle Tags beinhaltet die wir beim Zeichnen der Boxen verwendet haben. Wichtig dabei ist, auf gleiche Schreibweise zu achten.

Bei uns gibt es jetzt die Datei “**label_map.pbtxt**” mit folgendem Inhalt:

```
item{
	name: ‚withoutMask‘
	id: 1
}
item{
	name: ‚withMask‘
	id: 2
}
```

Als nächstes wird eine sogenannte record Datei generiert. Dazu nutzen wir folgendes Skript: https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/_downloads/da4babe668a8afb093cc7776d7e630f3/generate_tfrecord.py
Das Skript bekommt entweder das Train oder das Test Verzeichnis mit dem Bildern und andererseits die labelMap-Datei. Einer der Aufrufe sah folgendermaßen aus:

```
python generate_tfrecord.py -x images/train -l label_map.pbtxt -o
output/train.record
```

Wir generieren nicht ein komplett neues Modell, sondern wenden transfair-learning an, um ein Standardmodell an unsere Maskenerkennung anzupassen. Dazu besuchen wir den Tensorflow Model Zoo. Jedes Standardmodell besitzt einen bestimmten Geschwindigkeits- und Genauigkeitswert. Wir benutzen das SSD MobileNet V2 FPNLite 320x320:
https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

In der Verzeichnisstruktur des eben heruntergeladenen Modells befindet sich eine Datei namens “pipeline.config”. Diese Datei wird zum Konfigurieren benutzt, deshalb muss diese bearbeitet werden. Unsere Konfigurationen sind folgende:

```
pipeline_config.model.ssd.num_classes = 2
pipeline_config.train_config.batch_size = 10
pipeline_config.train_config.fine_tune_checkpoint =
'/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8/checkpoint/ckpt-0'
pipeline_config.train_config.fine_tune_checkpoint_type = "detection"
pipeline_config.train_input_reader.label_map_path='label_map.pbtxt'
pipeline_config.train_input_reader.tf_record_input_reader.input_path[:] =
['train.record']
pipeline_config.eval_input_reader[0].label_map_path ='label_map.pbtxt'
pipeline_config.eval_input_reader[0].tf_record_input_reader.input_path[:] =
['test.record']
```

Die Haupteinstellungen hier sind:

- ssd.num_classes: Anzahl der verschiedenen Szenarien, hier mit Maske und ohne Maske
- train_config.batch_size: Anzahl der verarbeiteten Bilder pro Trainings-Durchlauf
- viele Pfade (Label-Map oder Record Dateien)

Jetzt fehlt nur noch das Skript zum Trainieren, dieses kann einfach aus dem Git von Tensorflow heruntergeladen werden: https://github.com/tensorflow/models.
Nun sind alle Vorbereitungen getroffen und das Training kann gestartet werden. Für den ersten Versuch reicht es, wenn wir 2.000 Durchläufe festlegen. Das Kommando zum Starten lautet bei uns:

```
python models/research/object_detection/model_main_tf2.py --
model_dir=Tensorflow/workspace/models/my_ssd_mobnet --
pipeline_config_path=pipeline.config --num_train_steps=2000
```

Nach rund 3 Stunden wird das Programm beendet und es sind verschiedene Checkpoint-Dateien entstanden. Diese können entweder direkt genutzt werden oder in andere Dateiformate exportiert werden.

2.4. Webseite

Die Webseite wurde zunächst nur lokal programmiert, wobei uns am wichtigsten war, dass sie übersichtlich und leicht zu bedienen ist.

2.4.1. Code

Der Code für die Webseite ist in einen HTML-Code und einen CSS-Code unterteilt:

```
<!DOCTYPE html>
<html>
  <head>
    <title>FacePi</title>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static',
      filename='facePi.css') }}">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/jpg" href="{{ url_for('static',
      filename='maske.jpg') }}">
  </head>
  <body>
    <div class="cameraWindow">
      <img class="facePiCam" src="{{ url_for('video_feed') }}">
    </div>

    <div class="logo">
      <a>facePi - Stream </a>
    </div>

    <div class="navbar">
      <button onclick="callFlask('disableVideoFeed')" type="button"
      class="stopButton">&#x2715;</button>
      <button onclick="callFlask('activateVideoFeed')" type="button"
      class="playButton">&#10148;</button>
      <button onclick="callFlask('picture')" type="button"
      class="screenShotButton">&#128247;</button>
      <button onclick="callFlask('alarm')" type="button"
      class="alarmButton">&#9888;</button>
    </div>

    <script>
      const callFlask = async (path) => {
      await fetch(path);
      }
    </script>
  </body>
</html>
```

1: Webseite HTML-Code

```css
.facePiCam{
    width: 100%;
    background-attachment: fixed;
}

body {
    margin: 0;
    min-height: 100%;
    min-width: 100%;
    overflow: hidden;
    font-family: Arial, Helvetica, sans-serif;
}

.stopButton{
    width: 60px;
    height: 60px;
    background: red;
    color: white;
    border: none;
    border-radius: 100%;
    font-size: 2vw;
    margin-left: 1%;
    margin-top: 5px;
    opacity: 0.6;
}

.stopButton:hover{
    opacity: 1;
    cursor: pointer;
}

.playButton{
    width: 60px;
    height: 60px;
    background: green;
    color: white;
    border: none;
    border-radius: 100%;
    font-size: 2vw;
    margin-left: 1%;
    margin-top: 5px;
    opacity: 0.6;
}

.playButton:hover{
    opacity: 1;
    cursor: pointer;
}
```

2: Webseite CSS-Code 1

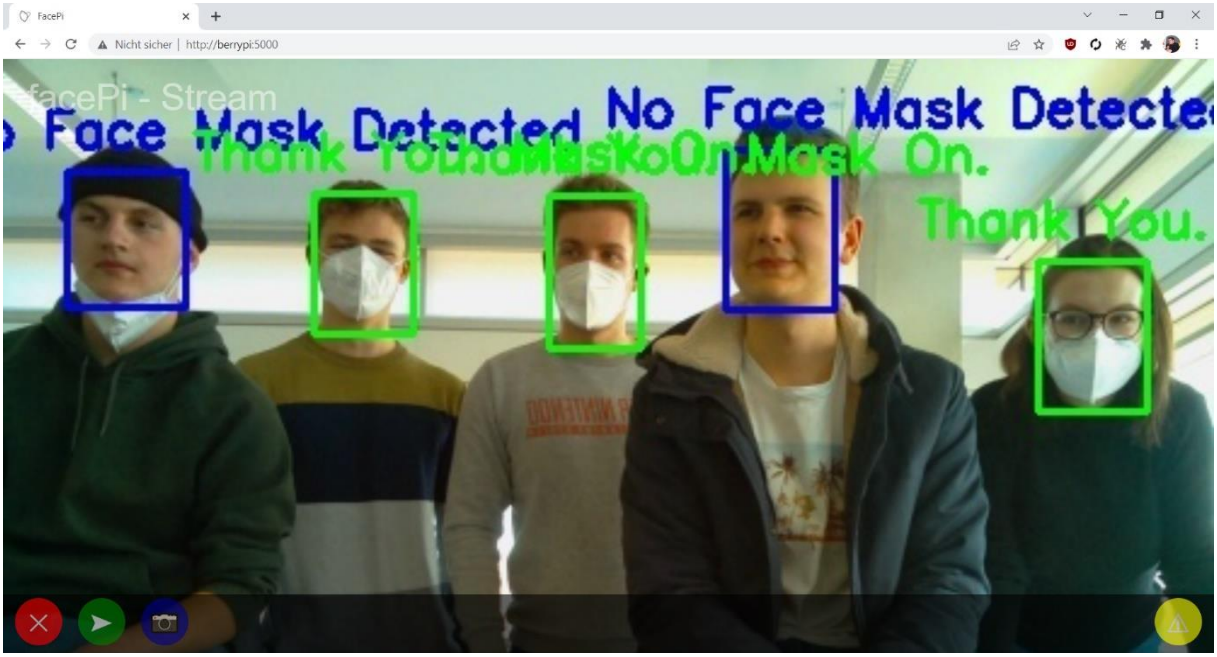
```css
.screenShotButton{
    width: 60px;
    height: 60px;
    background-color: darkblue;
    color: white;
    border: none;
    border-radius: 100%;
    font-size: 2vw;
    margin-left: 1%;
    margin-top: 5px;
    opacity: 0.6;
}

.screenShotButton:hover{
    opacity: 1;
    cursor: pointer;
}

.alarmButton{
    width: 60px;
    height: 60px;
    float: right;
    background: yellow;
    color: white;
    border: none;
    border-radius: 100%;
    font-size: 2vw;
    margin-right: 1%;
    margin-top: 5px;
    opacity: 0.6;
}

.alarmButton:hover{
    opacity: 1;
    cursor: pointer;
}

.logo {
    position: absolute;
    top: 3%;
    left: 2%;
    font-size: 3vw;
    color: white;
    opacity: 0.5;
}
```

3: Webseite CSS-Code 2

```
.navbar {
    overflow: hidden;
    position: fixed;
    bottom: 0;
    width: 100%;
    height: 10%;
    margin: auto;
    background-color: black;
    background-color: rgba(0,0,0,0.6)
}
```

4: Webseite CSS-Code 3

Die Webseite kann, wenn die Maskenerkennung auf dem Raspberry Pi läuft, unter der IP-Adresse des Raspberry Pi aufgerufen werden und sieht so aus:



2.4.2. Verknüpfung der Buttons mit Raspberry Pi

Die Buttons werden im Hauptskript (main.py) mit der Maskenerkennung verknüpft. Im HTML der Webseite wird, wenn auf einen Button gedrückt wird, die Funktion callFlask aufgerufen:

```
<button onclick="callFlask('disableVideoFeed')" type="button"
class="stopButton">&#x2715;</button>
```

Mit callFlask wird hier z. B. http://berrypi:5000/disableVideoFeed aufgerufen:

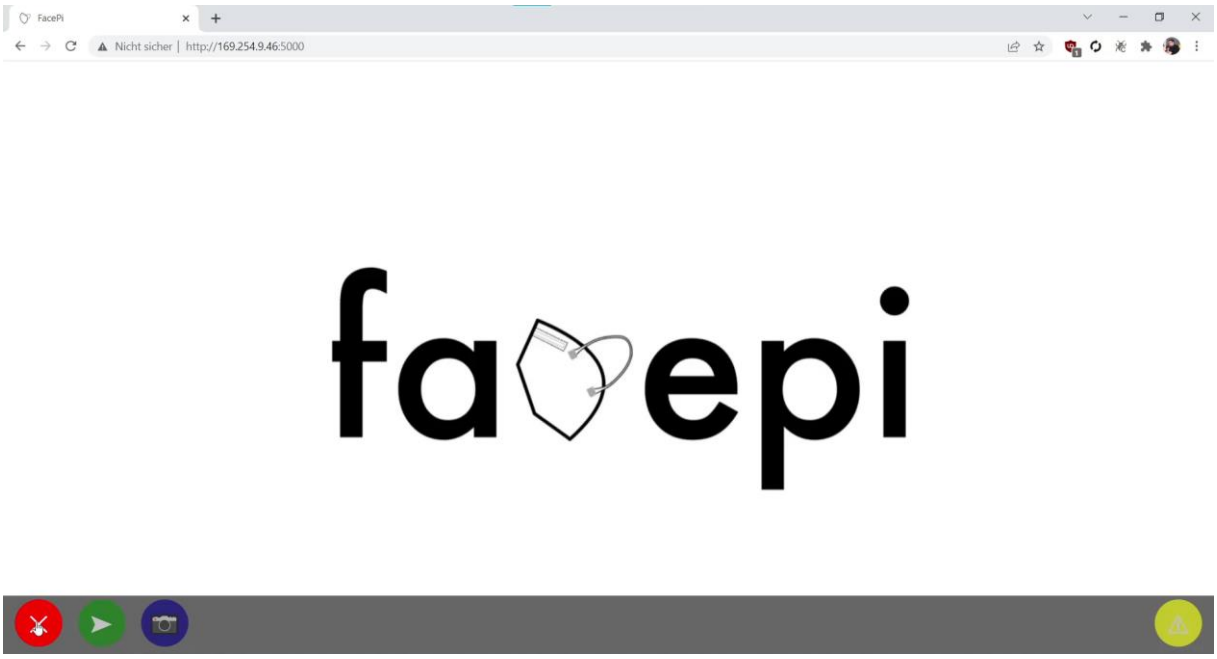
```
<script>
      const callFlask = async (path) => {
        await fetch(path);
      }
</script>
```

Im **main.py** wird dann definiert, was passieren soll, wenn diese Route aufgerufen wird. Hier also zum Beispiel das Stoppen der Übertragung, die Variable isVideoFeedActive wird auf falsch gesetzt:

```
@app.route('/disableVideoFeed')
def disableVideoFeed():
    global isVideoFeedActive
    isVideoFeedActive = False
    return 'disableVideoFeed'
```

Wird die disableVideoFeed-Variable zurückgegeben, erscheint das FacePi-Logo:

```
if(not isVideoFeedActive):
     RGB_img = cv2.imread('./static/logo.jpeg')
     rainbowhat.display.print_str('----')
  else:
     rainbowhat.display.print_str('0'+str(withCounter)+'0'+str(withoutCounter))
```

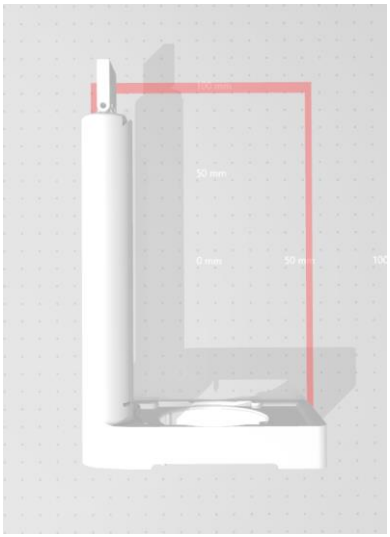


Mit dem grünen Button kann die Übertragung fortgesetzt werden. Mit dem blauen Button wird ein Screenshot erstellt, der dann auf dem Raspberry Pi gespeichert wird. Der gelbe Button schaltet den Alarm bei fehlender Maske aus.

2.5. 3D-Modell

Für die Maskenerkennung wurde eine mit einem 3D-Drucker gedruckte Halterung benötigt. Dabei war uns wichtig, dass die Neigung der Kamera einstellbar ist sowie ein Lüfter unter den Raspberry Pi eingebaut werden kann.

Das Modell wurde mit Fusion 360 erstellt:



Anschließend wurde das Modell mit dem Ender 3 Pro gedruckt:
https://www.youtube.com/watch?v=ITCdewf0aqU

Zusammengebaut sieht das Konstrukt folgendermaßen aus:

3. Lessons learned

Sehr positiv ist bei uns das Teambuilding gelaufen. Obwohl wir uns anfangs garnicht kannten, haben wir uns auf Anhieb sehr gut verstanden. So haben wir zu Beginn erst die Kompetenzen von jedem einzelnen abgefragt, um die Arbeitsteilung sinnvoll zu gestalten. Außerdem haben wir aufgrund von weiten Fahrstrecken unsere Arbeit oft online erledigt, was sehr gut funktioniert hat. Als VPN, damit jeder auf den Raspberry Pi zugreifen kann, haben wir ZeroTier verwendet.

Grundsätzlich hat jeder von uns sehr viel neues über den Raspberry Pi gelernt, vor allem durch Recherchen bei Problemen, die aufgetreten sind.

Ein Problem, was uns sehr früh aufgefallen ist war, dass der Raspberry Pi 3 bei der Bilderkennung eine schlechte Performance geliefert hat. Deshalb haben wir einen privaten Raspberry Pi 4 fürs Projekt verwendet, wodurch sich die FPS verdoppelt haben.

Des Weiteren wurde der Raspberry bei laufendem Programm sehr warm, weshalb wir einen Lüfter in die Halterung an die Unterseite des Raspberry integriert haben.

Ein großes Problem war, dass wir TensorFlow nicht auf dem PiOS 32 Bit installieren konnten, sodass wir die ganze Speicherkarte zurücksetzen mussten und zunächst PiOS 64 Bit installiert haben. Dann konnte TensorFlow installiert werden.

Außerdem ging beim Transport der HDMI-Port des Raspberry kaputt, sodass wir diesen nicht mehr an einen Bildschirm anschließen konnten und nur über einen VNC-Viewer erreichen konnten.

Für das nächste Projekt werden wir zu Beginn mehr Zeit damit verbringen, intensive Recherche zum Thema zu betreiben und nicht einfach drauf los zu legen. Damit hätten wir wahrscheinlich einige Probleme vermeiden können.

4. Quellen

Installation OpenCV:
https://qengineering.eu/install-opencv-4.5-on-raspberry-64-os.html

Installation TensorFlow:
https://qengineering.eu/install-tensorflow-2.7-on-raspberry-64-os.html

Tutorial Maskenerkennung:
https://www.tomshardware.com/how-to/raspberry-pi-face-mask-detector

Recherche Selbstprogrammierung von Maskenerkennung:
https://www.youtube.com/watch?v=yqkISICHH-U
https://www.youtube.com/watch?v=IOI0o3Cxv9Q

Gesamter Code:
https://github.com/MichaelBehringer/facePi