# Dokumentation zur Projektarbeit:

# Kopfsteuerung eines Androiden

Carl-Engler-Schule

Dokumentation im Rahmen des CT-Unterrichts

2021/2022

TGM13/1

[redacted]

[redacted]

# Inhaltsverzeichnis

## Anwendungsbeschreibung des Projektes aus Benutzersicht

Das Projekt besteht, hardwaretechnisch betrachtet, aus einem, gemäß dem Opensource-Projekt der Webseite inmoov.fr 3D-gedruckten und zusammengebastelten Kopf und Ständer, einem Raspberry Pi Zero W und einer aufgeschraubten Hercules Twist HD Webcam, die im rechten Auge des Kopfes verbaut ist. Anwender können sich mit ihrem mobilen Endgerät per Wi-Fi mit dem Kopf verbinden und daraufhin über eine Website den Kopf in viele verschiedene Richtungen steuern. Der Kopf besitzt 7 Servos, um letztlich 6-Achsen steuern zu können, nämlich jeweils die Augen und den gesamten Kopf in X- und Y-Richtung (vier Achsen), die Schräglage des gesamten Kopfes (eine Achse) und den Öffnungsgrad des Mundes (eine Achse). Des Weiteren besteht die Möglichkeit, auf das mobile Endgerät das Bild der Augenkamera zu streamen. Dieser Livestream besitzt eine automatisch arbeitende Bewegungserkennung. Das heißt, bei dem Registrieren einer Bewegung wird die Bewegung grün umrahmt, sobald die Bewegung stoppt, wird der grüne Kasten wieder entfernt.

Softwaretechnisch erfolgt die Steuerung über verlinkte HTML-Dokumente, der Anwender sieht zuerst eine Webseite mit einem Bild des Kopfes und Schiebereglern auf der rechten Seite. Mithilfe dieser Schiebereglern wird die Stellung des realen Kopfes eingestellt, zuerst kann man den Kopf in X- und Y- Richtung bewegen und die Schräglage setzen. Um andere Achsen anzusteuern, wie z.B. die Augen klickt man auf die Augen des Roboterkopfbildes und gelangt somit zu einer neuen Webseite. Auf dieser sind die Augen groß abgebildet, wieder mit dazugehörigen Schiebereglern auf der rechten Seite und ein Verstellen der Schieberegler bewirkt nun das Verstellen der Augen in X- und Y- Richtung. Dies funktioniert analog für den Mund. Um wieder zur Ausgangsseite zurückzugelangen benutzt man entweder die Browsernavigation, klickt auf das linkstehende Bild, oder klickt auf den Header, also die Überschrift.

## Entscheidungsfindung zur Auswahl der Werkzeuge und Software

Die Steuerung des Kopfes über einen ESP32 zu regeln, würde die Implementierung um Vieles vereinfachen, da wir bereits die notwendigen Programmiertechniken und Bibliotheken im Unterricht damit behandelt haben. Da dieses Projekt aber eine USB-Webcam nutzt, ist es einfacher, diese über einen Raspberry Pi zu betreiben, zumal der Raspberry Pi ein vollwertiger Einplatinencomputer mit Betriebssystem ist und somit über entsprechenden Treibern und Schnittstellen für die Webcam verfügt. Zur Programmierung des Raspberry Pis wird üblicherweise Python benutzt und somit auch für dieses Projekt, da es ein schnelles Testen von geschriebenen Programmen oder Skripts ermöglicht, ohne eine länger andauernde Kompilierung abwarten zu müssen. Das liegt daran, dass Python eine Interpreter-Sprache ist, bei der Skripte zur Laufzeit vom Python Interpreter übersetzt werden. Da der Raspberry Pi typischerweise mit dem Betriebssystem Raspbian, also einem Linux-

Derivat betrieben wird, könnte man für die Webserverkomponente des Projektes Webserversoftware benutzen, wie nginx oder Apache, die das Ausgeben von statischen Inhalten vereinfacht. Ich habe mich aber für eine gleichen Ansatz wie im CT-Unterricht entschieden, eben dass der Webserver softwareseitig erstellt und mithilfe bestimmter Bibliotheken gesteuert wird, da somit das Projekt zentral in einem Programm gesteuert wird und sich somit (soweit möglich und sinnvoll) in einem Skript befindet. Das Äquivalent zu der im Unterricht benutzten „ESPAsyncWebServer“-Bibliothek wird in diesem Projekt die für Python verfügbare Webserverbibliothek „Flask“ sein. Diese ist gut dokumentiert und vergleichbar in der Benutzung. Die HTML-Seiten sollen der Einfachheit halber mithilfe der Unterrichtsvorlage, basierend auf dem Template von selfhtml.org, umgestaltet und erstellt werden. Für die Interaktion mit der Webcam und der Bewegungserkennung wird die Bibliothek OpenCV benutzt. Diese ermöglicht nämlich komplexe Operationen überschaubar zu implementieren und abstrahiert essenzielle Schritte.

## Übersicht zur Umsetzung

Die Umsetzung gliedert sich in fünf abgrenzbare Teilschritte: Zuerst muss der Raspberry Pi als Router konfiguriert werden und entsprechende Bibliotheken für die Kamera oder Webserverkomponente installiert werden (1). Anschließend sollen die Webseiten nach dem Vorbild von selfhtml.org umgestaltet werden (2), der Raspberry Pi mithilfe der Webserverbibliothek Flask als Webserver agieren (3) und mithilfe von OpenCV, die Webcam Bewegung erkennen können (4). Daraufhin muss zuallerletzt das Videobild übertragen werden, ebenfalls mithilfe von Flask (5).

### *Raspberry Pi als Router konfigurieren und Vorbereitung*

Bevor der Raspberry Pi überhaupt als Router konfiguriert werden kann, muss natürlich das Betriebssystem aufgesetzt werden. Dazu wird von der Webseite raspberrypi.org die Lite Version des Betriebssystems Raspbian heruntergeladen und mithilfe des offiziellen „Raspberry Pi Imager“ die Image-Datei auf eine Micro-SD-Karte geschrieben. Die Lite Version besitzt weniger vorinstallierte Software und keine GUI und läuft dadurch letztlich effizienter. Nach dem erstmaligen Start und dem Einstellen üblicher Konfigurationen (z.B. deutsches Tastaturlayout) kann der Pi nun als Router konfiguriert werden. Der Pi als Router muss nach folgendem Schema arbeiten: Die WiFi-Schnittstelle des Raspberry Pis muss anstatt sich mit einem Netzwerk verbinden zu wollen (als Client agierend), anderen Endgeräten eine Netzwerkverbindung mit sich selbst ermöglichen (als Host agierend). Nach der Authentifizierung per WPA2 muss der Router den Client in sein eigenes lokales Subnetz einbinden, sprich ihm eine IP-Adresse zur weiteren Kommunikation vergeben. Dies geschieht mit dem **D**ynamic **H**ost **C**onfiguration **P**rotocol. Nachdem nun auch das Endgerät eine IP-Adresse besitzt, können Pi und Endgerät miteinander nach dem ISO/OSI-Referenzmodell miteinander kommunizieren. Um diese Schritte umzusetzen wird bestimmte Software auf dem Pi

installiert und Änderungen in Konfigurationsdateien vorgenommen, das Vorgehen möchte ich nun schildern:

In der Kommandozeile auf dem Raspberry Pi werden mit `sudo apt install hostapd` und `sudo apt install dnsmasq` die Programme „hostapd“ und „dnsmasq“ installiert. Ersteres Programm sorgt für das Umstellen der drahtlosen NIC des Pis als Serverkomponente und regelt die Authentifizierung und Verschlüsselung mittels WPA2. Dnsmasq sorgt für DHCP, installiert zusätzlich aber auch einen nichtbenötigten DNS-Server. Nun müssen jeweils die Dateien `/etc/dhcpcd.conf`, `/etc/dnsmasq.conf` und `/etc/hostapd/hostapd.conf` mit einem beliebigen Texteditor bearbeitet werden und jeweils bestimmte Zeilen am Dateiende hinzugefügt werden, damit die gewünschte Funktionalitäten von den Programmen umgesetzt werden. Für die Kommandozeile eignet sich das Programm „nano“ zum Bearbeiten von Dateien sehr gut dazu.

hostapd.conf

```
country_code=DE
interface=wlan0
ssid=NAME_DES_NETZWERKS
hw_mode=g
channel=7
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=PASSWORT
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
```

dnsmasq.conf

```
Interface=wlan0
dhcp-range=192.168.1.2,
192.168.4.20,255.255.255.0,24h
```

dhcpcd.conf

```
interface wlan0
static
ip_address=192.168.0.1/24
nohook wpa_supplicant
```

Kurzum wird mittels der Zeilen in dhcpcd.conf dem Pi eine statische IP-Adresse zugewiesen, damit dieser stets einfach erreichbar ist, mittels der Änderungen in dnsmasq.conf die IP-Adressvergabe spezifiziert und mit den Einstellungen in hostapd.conf das Verhalten des Routers gesteuert.
Mit einem abschließenden `sudo systemctl enable hostapd` wird festgelegt, dass bei jedem Systemstart hostapd automatisch startet, sprich der Pi auch nach einem Neustart wieder als Router agiert. Die Bibliotheken Flask und OpenCV werden mit den Befehlen `pip3 install Flask` und `pip3 install opencv-python` nachinstalliert.

## *(Um)gestaltung der Webseiten*

Insgesamt werden vier HTML-Webseiten benötigt. Für die Steuerung der Augen, für den Mund, für den Kopf selbst und eine für den Webcam-Stream. Alle vier besitzen das gleiche Grundgerüst. Ausgehend von der im Unterricht behandelten index.html Webseite, habe ich zuerst die Inhaltsverzeichnisliste und Texte entfernt und die Bilder durch Bilder von der Webseite inmoov.fr/gallery ersetzt. Zur Navigation sollte wie erläutert das Ausgangsbild des Kopfes dienen, indem man durch Klicken auf Augen oder Mund zu einer nächsten Seite gelangt, bei der nur die

Augen oder der Mund angesteuert werden können. Dazu habe ich zwei transparente Kästen erzeugt. Es werden jeweils zwei `img`-Elemente erzeugt und das Attribut `src` auf "` AABAAEAAAICTAEAOw==`" gesetzt. Dieser kryptisch aussehende Text bewirkt, dass ein transparentes GIF erzeugt wird, dessen Größe mit den üblichen Attributen `width` und `height` eines `img`-Elements festgelegt werden kann. Die Kästchen mit den ID-werten `eyeslink` und `mouthlink` werden dann mittels CSS-Selektoren relativ zum Bild des Kopfes (mit der ID `Head`) positioniert, damit sie über die Augen oder den Mund liegen. Anschließend habe ich die beiden Elemente mit einem anchor-Element umrahmt, der jeweils auf die entsprechen nächste Seite verweist. Auf jeder Webseite mussten Schieberegler mithilfe des input-Elements und dem definierten Attribut `type=range` platziert werden, die zur Steuerung der Achsen dienen und später das Senden der Werte instanziieren. Neben einem Schieberegler steht jeweils der aktuell gesetzte Wert, als veränderliches span-Element. Bei dem Verschieben eines Reglers wird die JavaScript-interne ereignisgesteuerte Funktion `oninput` aufgerufen. Mit dem Definieren des Attributes mit `oninput="change()"` wird die im Skript-Element definierte Funktion `change` aufgerufen, die den Wert des span-Elementes mit den aktuell eingestellten Wert des Reglers aktualisiert. Somit hat der Nutzer bereits beim Verschieben die Möglichkeit zu sehen, auf welchen Wert der Regler gesetzt wird und kann somit diesen genau einstellen. Beim Loslassen eines Reglers wird JavaScript-intern `onchange` und die damit im HTML verknüpfte Funktion `send()` aufgerufen. `send()` erwartet als Parameter eine Nummer, und kann somit den Aufrufer, sprich den Schieberegler identifizieren. Wie genau das Senden funktioniert soll aber später im Abschnitt „Webserver aufsetzen" erläutert werden. Des Weiteren wird bei einem Neuladen einer Webseite die JavaScript-Funktion window.onload aufgerufen. Standardmäßig ist diese leer. Verknüpft man sie mit einer Funktion, wird diese Funktion bei jedem Neuladen aufgerufen, für die Projektwebseite habe ich dieses Verhalten benutzt, um bei einem Neuladen, alle Schieberegler wieder in ihre Ausgangslage zu versetzten.

### *Webserver aufsetzen*

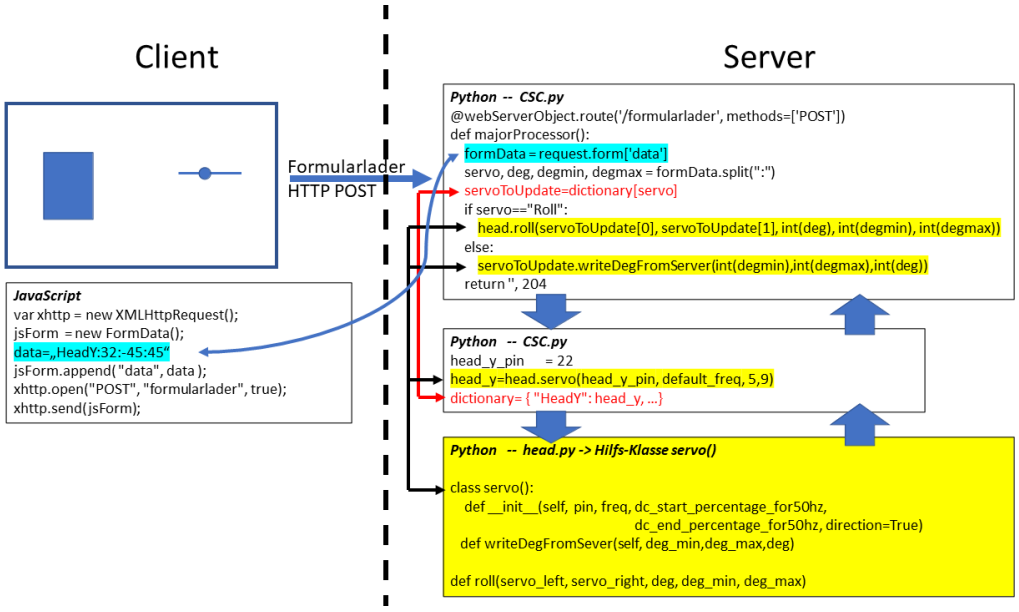
Mit nur wenigen Zeilen Code kann man mit der Bibliothek Flask einen Webserver erstellen (Beispiel 1). Es wird in Zeile 1 die Bibliothek eingebunden, in Zeile 2 ein Webserver-Objekt namens `app` instanziiert. Dieses Objekt besitzt viele Methoden und Attribute, mit denen das Verhalten des Webservers gesteuert werden kann. Die Syntax in Zeile 3 bis 5 ist etwas eigenartig, man spricht von Decorators. Kurzum, bei dem Aufrufen der URL, die innerhalb der Klammer bei `@app.route()` spezifiziert wird, (hier also http://server**/test**) wird die darunter neu definierte Funktion `testfunc()` aufgerufen. In

Beispiel 1
```
1 from flask import *
2 app = Flask(__name__)
3 @app.route('/test')
4 def testfunc():
5     return 'Hello World'
```

Beispiel 2
```
1 @app.route('/<file>')
2 def testfunc(file):
3   return 'The URL was ' + file
```

diesem Beispiel wird also beim Aufrufen der URL http://server/test, Hello World angezeigt. Alles was an den Client zurückgesendet wird, steht nach einem return, hier in Zeile 5. Das können Dateien, HTTP Statuscodes oder wie im Beispiel einfach nur Text sein. Für die in `@app.route()` angegebene URL lässt Flask aber auch eine Variable nach dem Schema `'/<var>'` (siehe Beispiel 2 oben) zu. Bei einem Aufruf jeglicher URL wird dann `testfunc` aufgerufen und die Variable `var`, die die angeforderte URL enthält, der Funktion übergeben. So kann man leicht programmiertechnisch prüfen, welche Ressource angefragt wurde und mit `return send_from_directory("Ordner-auf-Webserver", Dateiname)` entsprechende HTML, CSS oder JavaScript- Dokumente ausgeben. In diesem Projekt übernimmt die Funktion requestProcessor in den Zeilen 47/56 die Aufgabe des Ausgebens von Ressourcen. Die Funktion wird nur bei einer HTTP-Request-Methode „GET“ aufgerufen, dass bewirkt der zusätzliche Parameter `methods=['GET']` in Zeile 47. In einer for-Schleife wird die angefragte Ressource mit allen Namen der im html-Verzeichnis befindlichen Ressourcen abgeglichen und falls das Dokument gefunden wurde, ausgegeben. Falls eine angefragte Ressource nicht gefunden wird, wird der Client mit `return redirect(URL)` in Zeile 56 zur Startseite zurückgeleitet. Eine weitere Besonderheiten ist die Prüfung auf die Datei Webcam.html. Dies soll aber erst im Kapitel Livestream-Übertragung näher erläutert werden.

Ein weiterer wichtiger Aspekt des Projektes ist, wie die Daten vom Client zum Server gelangen und wie schlussendlich die Servos dadurch angesteuert werden. Dazu habe ich mich, für das Senden der Daten, für das AJAX Konzept entschieden, genauer für die XHR-API. Mit dieser ist es möglich, dass im HTML-Body kein Formular erstellt und Skript Code geschrieben werden muss, sondern Alles zentral in einem <script> Element gelagert werden kann. Die Trennung von JavaScript-Code und HTML erhöht die Übersichtlichkeit der Webseite und ermöglicht somit ein einfaches bequemes Senden der Daten. Für die genauere folgende Erklärung dient nebenstehendes Bild. Wie bereits im vorherigen Abschnitt angedeutet, wird beim Ändern und Loslassen eines Schiebereglers eine JavaScript Funktion in der Form `send(NUMMER)` aufgerufen. Jeder Schieberegler einer Webseite besitzt eine ihm eindeutig



zugewiesene Nummer, die beim Aufruf der Funktion `send()` in einem if-else Konstrukt mit verschiedenen Nummernmöglichkeiten abgeglichen wird. Findet `send()` die zugehörige Nummer wieder, so fügt sie einen Namen des Schiebereglers, den neuen, aktuell eingestellten Wert, den minimal möglichen Wert, den ein Regler annehmen kann und den maximal möglichen Wert, jeweils separiert mit einem Doppelpunkt, einer Variable namens `data` hinzu. Mit `jsForm = new FormData()` und `jsForm.append(„data", data)` wird skriptseitig ein Formular erstellt und unter dem Schlüssel `„data"` die Werte der Variable `data`, im obigen Beispiel also `HeadY:32:-45:45,` hinzugefügt. Mit dem zum Anfang instanziierten Objekt `xhttp`, wird mit der Methode `xhttp.open()` eine Verbindung zum Webserver, genauer zur URL `/formloader,` initialisiert und mit `xhttp.send()` das zuvor erstellte Formular an diese URL mittels HTTP-POST gesendet. Auf dem Webserver wurde mithilfe von Flask ein Listener erstellt, der bei dem Aufruf dieser URL und bei einem HTTP Request Method = POST, die Funktion `majorProcessor` aufruft. In den ersten zwei Zeilen dieser Funktion werden die Daten des Formulars unter dem Schlüssel `,data'` mit `request.form[,data']` abgerufen und die darunter befindlichen Formulardaten, die mit einem Doppelpunkt voneinander getrennt waren, nun separat in eigene Variablen gespeichert. Im Beispielbild oben ist nun `servo = „HeadY"`, `deg = 32, degmin = -45` und `degmax = 45.` Im Python-Server Skript wurde zu Beginn, vor dem Implementieren des Servers, mehrere Servo-Objekte erstellt, die auf der Vorlage einer selbstgeschriebenen Hilfsklasse servo() basieren. Die Hilfsklasse befindet sich in einer seperaten Datei namens head.py und wird in der neunten Zeile von CSC.py im Namensraum des Skriptes importiert. Die relevanten Methoden dieser Klasse, deren Interface bzw. Aufrufschema jeweils im obigen Beispielbild im gelben Kasten zu sehen ist, sind der Konstruktor `__init__()`, `writeDegFromServer()` und die freistehende Funktion `roll()`. Beim Erstellen eines Objektes wird der Konstruktor aufgerufen und dem Servo-Objekt z.B. mitgeteilt welcher Tastgradbereich erlaubt ist. Alle Servo-Objekte werden in einem Python-Dictionary unter speziellen Schlüsseln abgespeichert (siehe rote Schrift Beispielbild). Diese String-Schlüssel entsprechen genau den vom Client im Formular übertragenen Namen der Schieberegler. Mit `servoToUpdate = dictionary[servo]` wird mit dem gesendeten Namen des Servos als Schlüssel, aus dem Dictionary das entsprechende Servo-Objekt entnommen und dieses der Variable `servoToUpdate` übergeben. Nun das richtige Servo-Objekt darin gespeichert ist, kann darüber die Methode `writeDegFromServer()` aufgerufen werden, welche die vom Formular gesendeten Daten als Parameter benötigt. Die Methode setzt den Servo per PWM auf den richtigen Wert. Zuletzt wird mit `return ' ', 204` der HTTP Statuscode 204 „No-Content" zurückgesendet.

```
head_y=head.servo(head_y_pin, default_freq, 5,9)
→ head_y_pin: GPIO Pin
→ default_freq: einzustellende Frequenz für
                PWM-Signal hier 50 Hertz
→ 5: Minimaler Tastgrad
→ 9: Maximaler Tastgrad
```

Da beim setzen der Schräglage des Kopfes zwei Servos angesteuert werden müssen, wird auf diesen Sonderfall geprüft. Das Python-Dictionary gibt für den Schlüssel „Roll“ zwei Servo-Objekte zurück und anschließend werden diese der Funktion roll() aus head.py übergeben. Diese setzt beide Servos, mithilfe entsprechender Berechnungen, auf die richtigen Tastgrad-Werte.

### *Bewegungserkennung*

```
1 import cv2
2 cameraObject = cv2.VideoCapture(0)
3 while True:
4     ret, frame = cameraObject.read()
5     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
6     cv2.imshow('Picture', gray)
```

Bevor die Bewegungserkennung näher beschrieben wird, soll nebenstehendes Beispiel zeigen, wie das Arbeiten mit der Webcam und der Bibliothek OpenCV generell funktioniert. Nach dem importieren der Bibliothek OpenCV die intern den Namen cv2 trägt, wird in Zeile 2 die Webcam initialisiert. Mit dem neu erstellten Kamera-Objekt kann mithilfe der Methode read() ein Schnappschuss erstellt werden. Die Methode liefert zwei Werte zurück, erstens ob das Auslesen der Kamera geklappt hat mit True oder False, zweitens ein großes Array mit den RGB-Werten jedes einzelnes Pixels. Die großen Datenmengen können mithilfe eingebauter Funktionen von OpenCV einfach verarbeitet werden. In Zeile 5 wird das Bild beispielhaft in Graustufen konvertiert, dass erkennt man gut an den vielen aussagekräftigen Konstanten oder Namen der Methoden, hier zum Beispiel am zweiten Parameter „cv2.COLOR_**BGR2GRAY**“, sprich BGR (bzw. RGB) *zu* Grau. Die Methode `cv2.cvtColour` konvertiert also das aufgenommene Bild, welches in der Variable frame gespeichert wurde und der Rückgabewert der Funktion ist ein neues Array aus allen Pixeln, die nun nur noch Graustufenwerte besitzen. Das zurückgegebene Bild wird in der Variable `gray` gespeichert und in Zeile 6 mit `imshow()` ausgegeben.

Für die Bewegungserkennung auf Bildern gibt es viele verschiedene komplexe Algorithmen, es gibt aber auch eine durchaus verständliche und nachvollziehbare Methode, die diese Aufgabe bewältigen kann: In einer Endlosschleife wird stets ein aktuelles Bild in einer Variable gespeichert und später einem Referenzbild mit 25% Gewichtung hinzugefügt. Über die Zeit hinweg gibt das Referenzbild Aufschluss darüber, wie die Umgebung aussieht, während das aktuelle Bild, rasche Änderungen wahrnehmen kann, die das Referenzbild noch nicht wahrgenommen hat. Beide Bilder werden während eines Schleifendurchlaufes in Graustufen umgewandelt und computertechnisch unscharf gemacht, damit das Rauschen der Webcam nicht einen Fehlalarm auslösen kann. Anschließend werden alle Pixelwerte beider Bilder Pixel für Pixel voneinander abgezogen. In diesem neu entstandenen Bild werden nun Pixel, deren Differenz ziemlich groß ist, auf weiß gesetzt und Pixel, deren Differenz sehr klein ist, auf Schwarz. Dadurch entstehen schmale Konturen, die verstärkt werden müssen, indem umliegende Pixel ebenfalls auf weiß gesetzt werden. Das Zwischenergebnis ist nun ein Bild, auf dem Bereiche, in denen rasche Bewegung stattgefunden hat,

mit weißen Konturen umrandet sind, oft auch als weiße Flächen sichtbar, und der Rest des Bildes schwarz geblieben ist. Normalerweise gibt es dabei kleinere und größere Bewegungsbereiche, nur die größeren Bereiche, ab einem bestimmten Schwellenwert, sollen als Bewegung erkannt werden und eine Umrahmung bekommen. Über das Ursprungsbild, also das am Anfang eines Durchlaufes aufgenommene Farbbild, wird dann ein Rahmen an den entsprechenden Stellen gezeichnet. Falls keine Bewegung wahrgenommen wurde bzw. so langsam stattgefunden hat, dass das Referenzbild auf diese langsame Änderung schon aktualisiert worden ist, und folglich keine große weißen Konturen entstehen, wird kein Kasten gezeichnet. In dem Hauptskript des Projektes befindet sich dieser Teil, der genau diese Schritte implementiert, in den Zeilen 74/101, in den Funktionen `getFrame()` und `createDefaultPicture()`. Die Funktion `createDefaultPicture()` sorgt dafür, dass vor einem ersten Durchlauf bereits ein Referenzbild besteht, mit dem `getFrame()` arbeiten und dieses über die Zeit hinweg aktualisieren kann. `getFrame()` kann auf die Variable des Referenzbildes zugreifen, da das Referenzbild in einer globalen Variable gespeichert ist. Auch das OpenCV-Kamera-Objekt ist global, damit beide Funktionen Bilder von der Webcam auslesen können. Wie nun in einer Schleife das ganze durchgeführt wird und wann die Funktionen aufgerufen werden, um später dem Webbrowser des Clients einen Stream zuzusenden, soll später im Abschnitt Livestream-Übertragung erläutert werden.
Generell gibt es bei den einzelnen Methoden oft Besonderheiten in der Übergabe bestimmter Parameter, die bestimmte Datentypen benötigen. So müssen zum Beispiel die Pixelwerte des Referenzbildes stets Gleitkommazahlen sein, da beim gewichteten Aktualisieren des Referenzbildes, welches mit `cv2.accumulateWeighted(AKTUELLES_BILD, REFERENZBILD, GEWICHTUNG)` in Zeile 90 von csc.py erledigt wird, Kommazahlen entstehen können. Um aber die Differenz des Referenzbildes und des aktuellen Bildes bilden zu können, wie z.B. in der nächsten Zeile 91 mit `cv2.absdiff(AKTUELLES_BILD, REFERENZBILD)` müssen die Pixel beider Bilder wieder Ganzahlen sein. Falls nicht, wird eine Fehlermeldung erzeugt. Um dies zu Lösen habe ich in der Funktion `createDefaultPicture()` alle Pixelwerte des unscharfen, grauen, verkleinerten Gesamtbildes mit dem Aufruf `astype(„float“)` zum Datentyp `float` konvertiert, damit `accumulateWeighted()` zufrieden ist und später für `absdiff()` mit der Methode `cv2.convertScaleAbs(REFERENZBILD)` die Pixelwerte wieder zurück zu Integers konvertiert. Die Funktion `astype()` entstammt der Hilfsklasse „numpy“, die hinter den Kulissen viele weitere Dinge erledigt und auch von OpenCV unumgänglich selbst benutzt wird. Um nicht jeden einzelnen Befehl in der Dokumentation erläutern zu müssen, habe ich die beiden Funktionen `getFrame()` und `createDefaultPicture()` ausführlich kommentiert, ich hoffe die genauen Einzelheiten werden dadurch weitgehend ersichtlich.

### *Livestream-Übertragung*

Wie wird nun aber mithilfe dieser beiden Funktionen ein Livestream erzeugt, der dann auch noch per Webserver übertragen werden kann? Bis jetzt sind es nämlich nur zwei Funktionen, die bei ihrem Aufruf die Bewegungserkennungsschritte einmal durchführen können, und sich danach selbst nicht wiederholen. Grob zusammengefasst, wird bei dem Aufruf der Webcam-Webseite eine Endlosschleife gestartet, in der automatisch, ohne dass der Client erneut die Webcam-webseite aufrufen muss, wiederkehrend ein neues Bild gesendet und das alte Bild im Client-Browser ersetzt. Wie genau dies erfolgt, soll nun geschildert werden:

Der Client gelangt zur Webseite mit der Bewegungserkennung, indem er auf die Webseite, mit der man die Augen steuern kann, auf ein nebenstehendes Piktogramm einer Kamera klickt. Daraufhin wird die Webseite `Webcam.html` aufgerufen. Diese Webseite besitzt ein img-Element, dessen src Attribut auf die URL `/VS` referiert, der Browser versucht nun diese Ressource vom Webserver zu holen. Auf dem Server wird auf diese URL geprüft und bei einem Treffer zuerst mit `createDefaultPicture()` ein Referenzbild erstellt. Dann wird `return Response(stream(), mimetype='multipart/x-mixed-replace; boundary=grenze')` in Zeile 51 vom Hauptskript ausgeführt. Um Daten zum Client zurückzusenden benutzt Flask selbst hinter den Kulissen <u>immer</u> die Methode `Response()`. Im Beispiel 1 im Abschnitt „Webserver aufsetzen“ wurde erläutert, dass man mit `return'TEXT'` willkürlichen Text zum Client zurücksenden kann. Eigentlich benutzt Flask auch hier im Hintergrund die Methode `Response()`, übergibt ihr den Text und fügt weitere Parameter hinzu, wie z.B. den für das HTT(P)-Protokoll benötigten HTTP-Header, den MIME-Type (für Texte also `text/plain`) und Vieles mehr. Für viele Standardfälle reicht es einfach aus, sich über dieses Verhalten keine Gedanken zu machen, da diese HTTP-spezifische Parameter von Flask automatisch gesetzt und generiert werden. Andererseits ermöglicht die Benutzung der Response-Methode mehr Optionen und sie ist hilfreich, wenn Flask nicht erkennen kann, was man genau vorhat. Um einen kontinuierlichen Videostream zu erzeugen, bedarf es nämlich bestimmter Änderungen an MIME-Types oder HTTP-Header Parameter und deshalb wird in Zeile 51 die Response-Methode benutzt. Dieser übergibt man eine Funktion (im Projektskript also `stream()`) die Response ausführt, und einen MIME-Type mit dem Wert „`multipart/x-mixed-replace; boundary=grenze`“. Dies ist ein spezieller Ausdruck, der besagt, dass das gesendete Element Teil einer langen Kette von übertragenen Daten sein wird. Nach einem einmaligem senden eines HTTP-Response-Headers werden alle darauffolgenden Elemente in Form eines TCP-Datenstreams, abgegrenzt mithilfe des in der Variable `boundary` festgelegten Strings (hier also „grenze“), gesendet. Die linkstehende Grafik veranschaulicht dieses Verhalten.

1.Paket
```
HTTP/1.1 200 OK
Content-Type: multipart/x-mixed-replace;
boundary=grenze
```

2.Paket
```
--grenze
Content-Type: image/jpeg
JPEG-BYTES
```

3.Paket
```
--grenze
Content-Type: image/jpeg
JPEG-BYTES
```

Dadurch, dass die von Response aufgerufene Funktion stream() eine Endlosschleife ist (siehe Zeile 106 in CSC.py), welche endlos ein Bild mithilfe der Bewegungserkennungsfunktion `getFrame()` generiert, wird auch Endlos ein Bild aufgenommen und dem Client zugesendet. Dies vermittelt letztlich den Anschein eines sich bewegendes Bildes im Browser. Eine wichtige Rolle spielt dabei das `yield (b'--grenze\r\n' b'Content-Type: image/jpeg\r\n\r\n' + frame + b'\r\n\r\n')` in Zeile 108. Die Funktion Response() muss letztlich das zu streamende Bild nach dem in der Grafik gezeigten Schema senden. `Response()` ruft `stream()` auf, `stream()` muss aber der Funktion `Response()` das Bild, in Form eines multipart MIME-Type Strings zurückgeben. Würde die Funktion in einer Endlosschleife laufen, so würde die Funktion `stream()` nie beenden und niemals `Response()` etwas zurückgeben. Mit einem return-Statement würde man zwar für einen ersten Durchlauf den Datensatz an `Response()` zurückgeben, da mit einem return aber auch, abgesehen von einer laufenden Schleife, die überliegende Funktion beendet wird, würde dies nicht funktionieren und dem Client letztlich nur ein einzelnes Bild zurückgesendet. Hier kommt `yield` ins Spiel, `yield` verhält sich ähnlich wie `return`, indem es dem Aufrufer Werte zurückgeben kann, beendet jedoch nicht die Funktion. Dadurch kann `stream()` endlos weiterlaufen, und einen Videostream generieren, bis der Client die Seite verlässt.

## Ausblick

Nach vielem Testen, umschreiben und experimentieren funktioniert im Moment das meiste relativ gut und vor allem das Setzen der Servos geschieht sehr schnell. Die Livestream-Übertragung hingegen läuft relativ langsam, circa mit 1,5 Sekunden Verzögerung und außerdem ist der Raspberry Pi in den Momenten des Sendens zu 100% (Prozessor) ausgelastet. Mit dem Kommandozeilenprogramm `htop`, welches für den Pi frei verfügbar und vergleichbar mit dem Task-Manager in Windows ist, kann man dieses Verhalten gut betrachten. Interessant ist dabei auch, dass der Arbeitsspeicher hingegen nicht völlig ausgeschöpft ist, obwohl dieser bei dem Raspberry Pi Zero W nur 512MB groß ist.
Nach einem Untersuchen des Netzwerkverkehrs zwischen Pi und Client mithilfe des Programmes Wireshark konnte ich feststellen, dass alle Netzwerkpakete des Streams mittels TCP übertragen werden. TCP ist ein verbindungsorientiertes Protokoll, für das Senden eines Datenpaketes werden insgesamt drei Netzwerkpakete zwischen Client und Server gesendet, um unter anderem zu garantieren, dass alle Daten fehlerfrei angekommen sind. Dies ist bei Livestreams aber nicht notwendig, da bei einer solch, computertechnisch betrachtet, enormen Datenmenge es nicht auffallen würde, falls für einen Pixel eines Bildes nicht der korrekte Wert übertragen wurde. Deshalb werden normalerweise Livestreams mittels UDP, dem Gegenstück zu TCP, gesendet, welches aufgrund fehlender Überprüfungen, bei Medientransfers viel schneller arbeiten kann. Zur Optimierung des Skriptes könnte man versuchen, dies umzustellen. Die Performance würde

ebenfalls verbessert werden, wenn das Programm in C umgeschrieben und dann kompiliert wird und/oder auf einen leistungsstärkeren Pi, als den Raspberry Pi Zero W ausgeführt wird. Auf meinem Laptop konnte ich, bei dem Ausführen des Skriptes und dem Prüfen der Webcamseite, keine Livestream-Verzögerungen feststellen. Auch in Punkten Sicherheit müssten einige Programmteile neugeschrieben werden. Obwohl nur Clients mit dem richtigen WiFi-Passwort sich einloggen und somit keine willkürlichen Endgeräte mit dem Pi verbinden können, validiert der Server nicht die Eingaben des Clients. Würde ein Angreifer es schaffen das WiFi-Passwort zu knacken und sich einzuloggen, so könnte er mit einem manipuliertem HTTP-POST Request willkürlich Daten an den Webserver senden, und wohlmöglich dass Programm zum Abstürzen bringen oder im schlimmsten Falle sogar eine Remote Code Execution durchführen.