DIE ENTWICKLUNG DES DIGITALEN „BLINDENHUNDS“

GUIDE-Walk 2.0
Autonomes Blindenführersystem mit KI

von Tamas Nemes

(Gymnasium der Regensburger Domspatzen)

Jugend Forscht 2020 (Bayern)

Projektbetreuer: René Grünbauer

Kategorie: Technik

KURZFASSUNG

Inklusion ist ein allgegenwärtiges Thema in unserer modernen Gesellschaft; wir haben erkannt, dass wir körperlich und geistig benachteiligte Menschen aktiv dabei unterstützen müssen, sich in die Gesellschaft einzugliedern. Zweifelsohne war dies bis vor gar nicht so langer Zeit nicht selbsterklärend.

Darunter sind auch Blinde und Sehbehinderte keine Ausnahme: Die unterstützende Infrastruktur für Blinde und Sehbehinderte wird mit Bodenleitsystemen und Verkehrsampeln mit Audiosignalen immer weiter ausgebaut und ist für uns heutzutage selbstverständlich. Die Technologie der künstlichen Intelligenz (KI) kann dabei helfen, spannende Technik zu kreieren, womit blinden Menschen ermöglicht wird, selbstständiger zu werden und alleine unkompliziert durch die Stadt navigieren zu können.

Ich befasse mich in meinem Projekt mit dem Bau und der Programmierung eines tragbaren, autonomen Blindenführersystems, welches mit Hilfe einer Kamera und verschiedenen Sensoren lernt, Hindernisse zu erkennen, zu identifizieren und so den Träger rechtzeitig mittels Audiosignalen auf Gefahren hinzuweisen. Basierend auf Nvidia's Single-Board-Computer Jetson Nano kann das System die nötige Leistung für den KI-Betrieb vorweisen. Dieses Projekt baut auf die erarbeiteten Kenntnisse aus meinem letztjährigen Projekt ("Entwicklung eines Prototyps für ein Blindenführersystem mit KI", 78332) auf und stellt den Entwicklungsprozess mit allen Aufgaben und Herausforderungen dar. Zusätzlich sollen die Erfahrungen und Vorschläge von Betroffenen dargestellt werden.

INFORMATIONEN ZU GEFÄHRDUNGEN

Im Gerät ist ein LiDAR-Sensor zur Entfernungsmessung integriert, welches Laserstrahlen emittiert. Dieser operiert fortwährend mit einer Wellenlänge von 905nm (nominal) und einer Leistung von max. 1,3W, wodurch er der Laserklasse 1 angehört und somit für das Auge ungefährlich ist. Weitere Informationen sind in der Bedienungsanleitung des Sensors zu finden.[1]

Das Gerät wird aus einem mobilen Akku (12V / 6A) mit Strom versorgt, welches dann im Gehäuse konvertiert wird (5V / 4A). Alle Kabel sind in gutem Zustand und hinreichend isoliert, wodurch die Gefahr eines Stromschlags ausgeschlossen werden kann.

[1] static.garmin.com/pumac/LIDAR_Lite_v3_Operation_Manual_and_Technical_Specifications.pdf

INHALTSVERZEICHNIS

Hinweise:

Das Layout und der Inhalt dieser schriftlichen Arbeit wurden in 100%-iger Eigenarbeit und ohne fremde Hilfe angefertigt.

Dieses Projekt basiert auf dem Wettbewerbsprojekt „Entwicklung eines Prototyps für ein Blindenführersystem mit KI“ (78332).

*Dieses Projekt wurde vom **Jugend Forscht Sponsorpool Bayern** finanziell unterstützt.*

gefördert vom SPONSORPOOL BAYERN
Infos unter www.sponsorpool-bayern.de

1. EINLEITUNG

Mit diesem Projekt möchte ich mein KI-basiertes Blindenführersystem im Vergleich zum Prototyp-Status vom letzten Jahr wesentlich weiterentwickeln, um ein voll funktionsfähiges und benutzerfreundliches Produkt zu erschaffen, welches die Inklusion von sehbehinderten Menschen unterstützen soll. Wie dieser Entwicklungsprozess verlaufen ist, welche Datensätze und Auswertungen dazu angefertigt und welche Hindernisse überwunden werden mussten, werde ich im Folgenden aufzeigen. Am Anfang möchte ich jedoch auf das zu diesem Projekt gehörende Repo hinweisen:

GitHub-Repository: https://github.com/Totemi1324/GUIDE-Walk-v2.0

Darin sind der gesamte Quellcode, sämtliche Zusatzdateien sowie die verwendeten Datensätze geordnet und zum Download verfügbar. Ich empfehle, den Code beim Lesen dieser schriftlichen Arbeit zur Hilfe zu nehmen.

1.1. Zusammenfassung der bisherigen Projektentwicklung

Seit Ende des Jugend Forscht Regionalwettbewerbs 2019 bestand für mich auf Anregung von Bekannten und Familienmitgliedern hin die Idee, einen KI-gestützten Helfer für Blinde zu kreieren. Seither habe ich mich in zahlreiche, mir anfangs unbekannte Aspekte der Programmierung, der künstlichen Intelligenz und der Elektrotechnik eingearbeitet, um diese Idee zu verwirklichen und dabei stets zu Experten, Unterstützern und Betroffenen (Danksagungen: → **siehe 7.**) Kontakt gesucht. Hilfreich war dabei, dass ich entsprechendes Vorwissen durch mein Jugend Forscht-Projekt „AI im Spiel: Künstliche Intelligenz mit Q-Learning und Reinforcement Learning“ (2019) sammeln konnte. Letztendlich ist es mir jedoch nicht gelungen, zum Wettbewerbstermin 2020 mehr als einen Prototyp zu entwickeln. Dieser war bereits mit akustischer Rückmeldung für Distanzen und einer KI ausgestattet, welches aber nicht selbst trainiert war und dem gesamten Programm allgemein viele wichtige Funktionen fehlten. Trotzdem erhielt das Projekt im Rahmen des Regionalwettbewerbs einen 1. Preis zusammen mit zwei Sonderpreisen, allerdings hatte ich wegen der Coronavirus-Pandemie nicht die Möglichkeit, zum landesweiten Wettbewerb anzutreten. Diese Technik habe ich für den diesjährigen Wettbewerb weiterentwickelt.

1.2. Zielsetzung

Das Ziel dieses Projekts war es, meine ursprüngliche Zielsetzung zu verwirklichen, nämlich ein mobiles Blindenführersystem zu entwickeln, welches potenzielle Gefahren im Straßenverkehr durch eine Kamera identifiziert und den blinden Träger mittels Audiowarnungen darauf hinweist, sowie darüber hinaus zusätzliche Informationen zur Optimierung der Performance sammelt und einsetzt. Ich habe mir vorgenommen, alle Aspekte des Geräts neu anzugehen, rundum zu erneuern und selber auszuarbeiten. Dabei wollte ich besonderen Fokus auf folgende Aspekte setzen:

- Alltagstauglichkeit durch einfache und intuitive Bedienung für praktische Anwendungsmöglichkeiten
- Umsetzung der Rückmeldungen von betroffenen Testpersonen
- Hohe Garantie für volle Funktionsfähigkeit, indem der Fokus auf die Kernfunktionen gesetzt und Qualität statt Quantität bevorzugt wird

2. GESAMTKONZEPT UND AUFBAU

2.1. Konzept für die Weiterentwicklung

Wie bereits erwähnt habe ich mir vorgenommen, ein Gerät zu entwickeln, welches Alltagstauglichkeit mit hoher Funktionalität und guter Performance verbindet und dadurch auch mit den Blindenführgeräten, die heutzutage auf dem Markt sind, konkurrieren kann. Mein letztjähriger Prototyp basierte gänzlich auf Python-Code, welches zwar für den Einstieg einfach zu handhaben war, jedoch im Vergleich zu kompilierten Sprachen sehr ineffizient und nicht serientauglich ist.

Aufgrund dessen, dass ich sowieso alles von Grund auf neu entwickeln wollte, war auf Anraten der Juroren schnell die Idee geboren, den Code stattdessen in *C++* umzusetzen. Dies gestaltete sich aber schwerer als zunächst angenommen, denn mit Ausnahme von *OpenCV* hatte keiner der Libraries, die ich benutzt hatte, eine C++-Einbindung, weswegen ich einen signifikanten Teil meiner diesjährigen Projektarbeit dazu aufgewendet habe, Ersatzmöglichkeiten zu suchen und zu testen (Details → **siehe einzelne Abschnitte in 3.**). Die Vorteile von C++ gegenüber Python sind vor allem eine kürzere Laufzeit und bessere Kontrolle über das Speichermanagement, weswegen ich versucht habe, diese Aspekte auch in meinem Code zu optimieren. Da ich bis zu diesem Zeitpunkt keinerlei Erfahrungen mit diesen Konzepten und C++ allgemein hatte, musste ich mich in diese Aspekte intensiv einarbeiten.

Im finalen Quellcode sind die einzelnen Komponenten des Blindenführers objektorientiert in Header-Dateien programmiert, die dann im Hauprogramm als Pointer einer Klasse angesteuert werden. Dies hat den großen Vorteil, dass sie, wenn sie gerade nicht benötigt werden, durch den *delete*-Befehl aus dem Speicher gelöscht und später wieder aktiviert werden können. Zudem fördert es die Übersichtlichkeit.

2.2. Aufbau des Blindenführers

Seit dem Prototyp hat sich einiges am Aufbau des Blindenführers geändert, hauptsächlich im Bereich der Sensoren und der Energieversorgung. Die folgende Abbildung veranschaulicht alle Komponenten von GUIDE-Walk mit entsprechender Erläuterung:

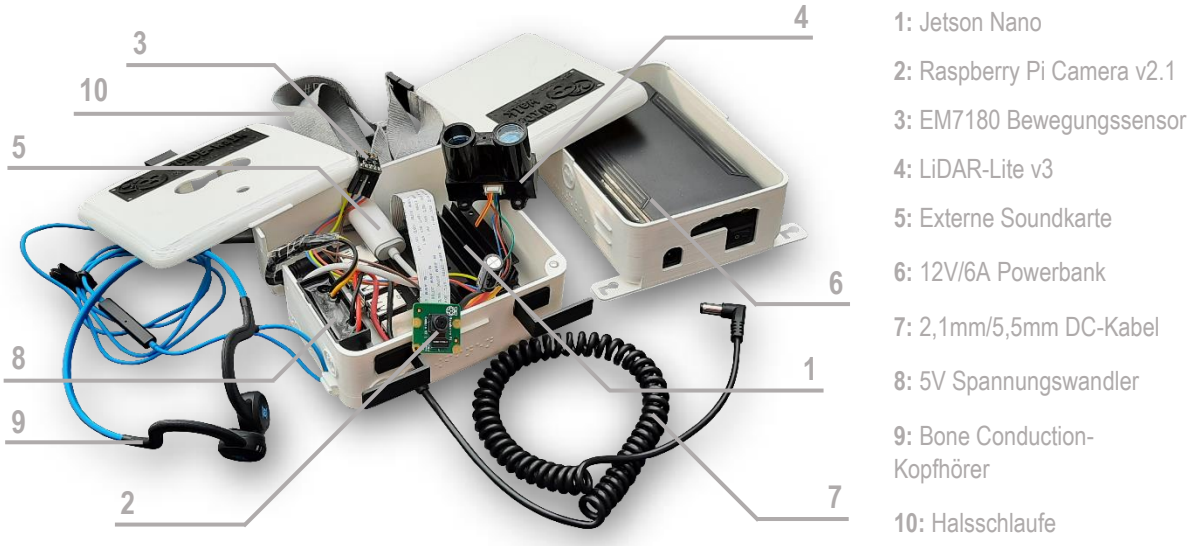


Abb. 1: Schaubild des Blindenführers mit nummerierten Komponenten

3. VORGEHENSWEISE

3.1. Training und Implementierung der KI

Als eine der wichtigsten Änderungen zum Prototyp wollte ich diesmal die künstliche Intelligenz, welche zum Erkennen von Hindernissen und Gefahrenpotenzialen in den Kamerabildern zum Einsatz kommt, in den Mittelpunkt setzen. Anders als der Projekttitel vielleicht vermuten lässt, soll die KI kein zusätzlicher, angehängter Teil, sondern das zentrale Feature sein, um welches sich alle anderen Funktionen drehen und diesen ergänzen. Da es die vielen anderen Verbesserungen, die gemacht werden mussten, zeitlich nicht zuließen, ein eigenes Netzwerk zu entwerfen, habe ich mir vorgenommen, ein bereits bestehendes Netz zu wählen, seinen Aufbau gründlich kennenzulernen und mit einem eigenen Datensatz vollständig zu trainieren.

Beim letztjährigen Programm wurde ein vortrainiertes Modell des Netzwerks *MobileNetV2* verwendet, welches durch die Nvidia-eigene Jetson Inference-Library angesteuert wurde. Für meinen Anwendungszweck eignet sich ein Object Detection-Netzwerk am besten, da es in der Lage ist, die Position von relevanten Objekten im Bild mit Bounding Boxes, also Begrenzungsboxen, zu erfassen. Die Anforderungen waren, dass das Netzwerk schnell und gleichzeitig möglichst genau ist. Zudem muss es über C++ steuerbar sein. Nach einiger Recherche zum eben erwähnten KI-Modell bin ich auf die *SSD*-Architektur von *Wei Liu* gestoßen, welches die obigen Eigenschaften vereint. Im Folgenden wird sein Aufbau und die Funktionsweise kurz erläutert.

SSD steht für *“Single Shot multibox Detector”* und bezeichnet eine bestimmte Art von CNN, welches auf Fully Connected-Layer verzichtet und daher nur einen Durchlauf braucht, um Detections zu ermitteln. Sie besteht aus zwei Teilen: Dem Stützwerk und dem SSD-Zusatz. Den zweiten Teil haben alle SSD-Modelle gemeinsam, der erste kann durch ein beliebiges Netzwerk ersetzt werden. Der Aufbau ist in der folgenden Abbildung aus dem Forschungspaper dargestellt:

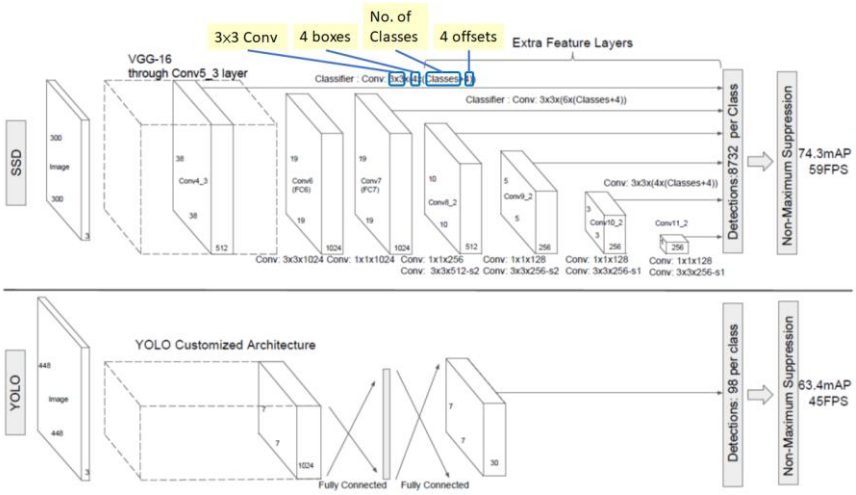


Abb. 2: Schematischer Aufbau von SSD im Vergleich zum state-of-the-art-Netzwerk YOLO

Das Stützwerk, im Bild VGG-16, ist ein leicht angepasstes CNN und bestimmt zum großen Teil die Geschwindigkeit und Genauigkeit des gesamten Netzwerks. Aus Benchmarks u.a. von Google AI wird ersichtlich, dass MobileNetV2, also das Netzwerk, welches ich ursprünglich verwendet habe, eine gute Balance zwischen

Modellgröße und Genauigkeit bietet. Daher ist es optimal für diesen Zweck geeignet. Es ist ein relativ neues und schnelles CNN bestehend aus Convolution- und Pooling-Layern. Seine Aufgabe ist, eine Feature Map aus dem Input zu generieren, eine Art Matrix, der die Ergebnisse aller Musterabgleichungen in den Convolution-Layern zusammenfasst. Dann kommt der SSD-Zusatz ins Spiel: Er unterteilt den Output des Stütznetzwerks in ein Raster, dessen Auflösung im Verlauf der Verarbeitung immer kleiner wird. Das Besondere an SSD ist, dass in jedem Raster eine bestimmte Anzahl an Bounding Boxes mit vorher festgelegten Seitenverhältnissen erzeugt wird. Der Layer *Conv4_3* beispielsweise unterteilt den Output in ein 38x38-Raster, macht in jeder Zelle 4 Vorhersagen (daher der Name "multibox") und erzeugt dadurch $\frac{Boxen}{Zelle} \cdot Höhe \cdot Breite = 4 \cdot 38 \cdot 38 = 5776$ Boxen. Am Ende gib das Netzwerk insgesamt 8732 Detections pro Klasse aus. Daher ist eine Non-Maximum-Suppression nötig, um Duplikate oder Boxen, die wahrscheinlich zum selben Objekt gehören, zu filtern. Schließlich werden die Detections nach der Confidence sortiert und die 100 Besten ausgegeben.

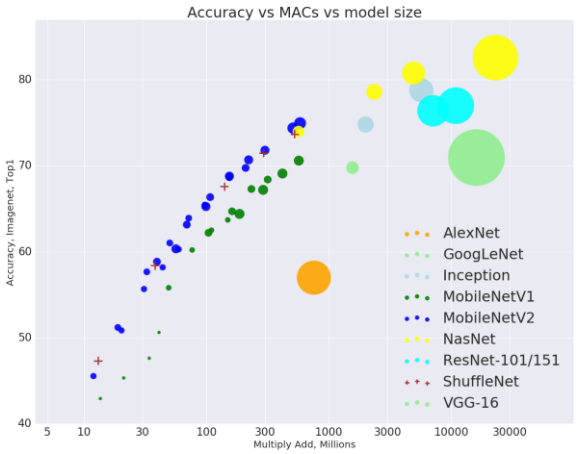


Abb. 3: Vergleich diverser CNNs hinsichtlich Modellgröße und Genauigkeit

Einer der wesentlichen Gründe für meine Entscheidung zugunsten MobileNet-SSD v2 war seine C++-Kompatibilität. Auf GitHub lässt sich eine Implementation des Netzwerks für das Deep Learning-Framework *Caffe* finden, welches ebenfalls auf C++ basiert und dessen Netzwerke mit dem DNN-Modul von OpenCV eingelesen werden können (*net.h*). Sie ist allerdings schon ein wenig veraltet und das Kompilieren auf aktueller Hard- und Software war ein unerwartet langwieriger und schwieriger Prozess, der fast ein halbes Jahr in Anspruch nahm.

3.1.1. Testen der Genauigkeit und Performance

Das Netzwerk soll in den Bildern, die von der Kamera erfasst werden, 10 verschiedene Objekte erkennen können: *Personen, Autos, Busse, Fahrräder, Motorräder, Bänke, Stühle, Mülleimer* sowie *rote und grüne Ampeln*. Um eine bestmögliche Performance zu gewährleisten, wollte ich mein Netzwerk nur auf diese Klassen trainieren. Da es aber kein fertiges Dataset nur für diese Klassen gibt, habe ich meinen eigenen Datensatz zusammengestellt. Mithilfe eigener Python-Skripte habe ich aus den Datensätzen *MS-COCO* und *Pascal-VOC* die Bilder mit den benötigten Klassen sowie die Annotations dazu extrahiert und in ein XML-Format konvertiert. Dazu kommt das *Ampelpilot*-Dataset, ein Datensatz zur Erkennung von deutschen Fußgängerampeln (dies war wichtig, da z.B. amerikanische Modelle wesentlich anders aussehen). Zum Schluss habe ich eigene Aufnahmen gemacht, sortiert und mit dem Tool *LabelImg* von Hand gelabelt. Dieser ganze Vorgang dauerte mehrere Monate, da auch die Bilder von COCO und VOC nochmal von Hand auf Fehler geprüft werden mussten. Insgesamt kam ich auf 33201 Bilder, davon 20990 aus COCO und VOC und 12211 eigene (Gesamtgröße: ca. 6GB). Sowohl der Python-Code als auch das Dataset und die KI-Konfiguration können im GitHub-Repo eingesehen und heruntergeladen werden.

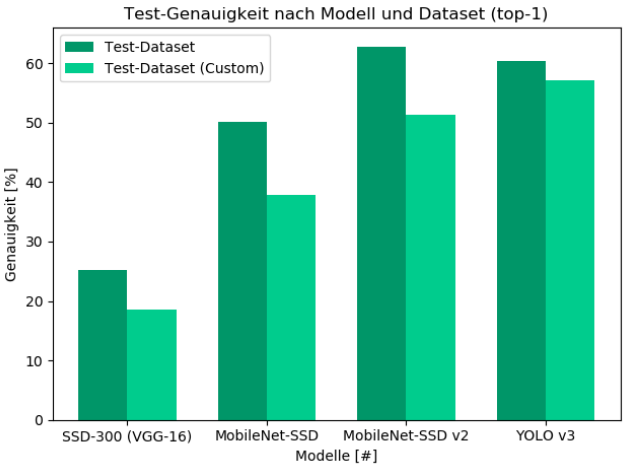


Abb. 4: Ergebnisse des Genauigkeitstests

Zum Experimentieren und für das Pretraining habe ich den Hochleistungsrechner an unserer Schule zur Hilfe gezogen (CPU: AMD Ryzen 9 3950X mit 32 Threads; GPU: Nvidia RTX 2070 Super; RAM: 32GB), mit dessen Kapazitäten ich bereits gute Ergebnisse erzielen konnte. Bevor ich allerdings meinen richtigen Datensatz anwenden konnte, wurde er aber leider gestohlen, weshalb ich auf einen Ersatzrechner angewiesen war (CPU: i7 6800K mit 12 Threads; GPU: Nvidia GeForce GTX 760; RAM: 32GB). Durch die viel geringere Rechenleistung waren die Trainingsmöglichkeiten sehr eingeschränkt, trotzdem ist es mir gelungen, einigermaßen gute Ergebnisse zu erzielen. Die Genauigkeit auf dem Testdataset beträgt 62,8% (Erwartungswert sind ca. 70%-80%). Zusätzlich habe ich zusammen mit einem Freund ein Testdataset erstellt, welches die einzelnen Objekte in einer isolierten Umgebung darstellt, um einen aussagekräftigeren Wert zu erhalten. Im obigen Diagramm ist dieses Ergebnis im Vergleich zu anderen Object Detection-Netzwerken zu sehen, die mit dem gleichen Dataset trainiert wurden. Dabei ist eindeutig zu erkennen, dass MobileNet-SSD v2 am besten abschneidet. Viele der falschen Boxen sind zudem Duplikate von größeren Boxen des gleichen Objekts, sodass mit der Filterung dieser die Genauigkeit weiter verbessert werden kann.

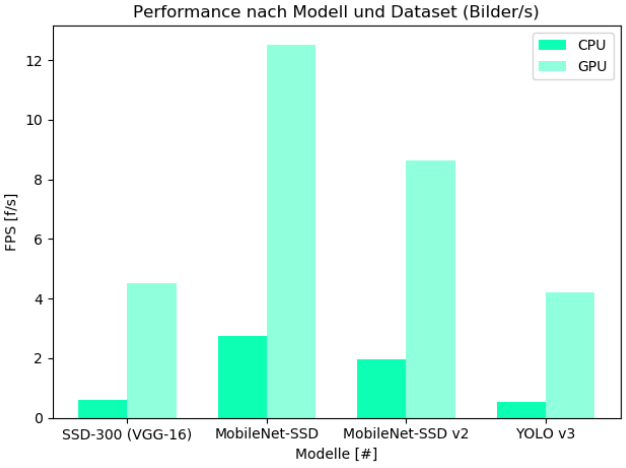


Abb. 5: Ergebnisse des Performancetests

Zudem habe ich die Performance dieser Netzwerke in der C++-Implementierung auf dem Jetson Nano getestet. Dafür wurde die Dauer der einzelnen Inference-Durchläufe pro Bild gemessen und daraus ein Durchschnittswert der FPS errechnet. Hier ist MobileNet-SSD v2 zwar nicht ganz vorne, jedoch ist die Genauigkeit vom besten Modell wesentlich geringer, daher kann das in Kauf genommen werden. Die FPS wurden sowohl ohne als auch mit GPU-Beschleunigung gemessen, was das Ergebnis im Durchschnitt vervierfacht.

3.2. Einrichtung des Bewegungssensors

Neben der KI, der den Kern des Geräts ausmacht, kommen zusätzlich mehrere Sensoren zum Einsatz, die die Objekterkennung sicherer machen und um zusätzliche Funktionen erweitern sollen. Am Jetson Nano ist ein GPIO-Erweiterungsheader mit 40 Pins integriert, an den diese angeschlossen werden können. Sie besitzt zwei I^2C-, zwei SPI- und einen UART-Bus und ist hinsichtlich der Pin-Belegung mit dem des Raspberry Pi fast identisch.

Letztes Jahr habe ich nur einen Ultraschallsensor (→ **siehe 3.3.**) benutzt, deshalb wollte ich für die zweite Version einen Bewegungssensor hinzufügen, um vielfältigere Möglichkeiten zu haben, die Bewegung auf der Straße zu erfassen. Auch dieses Mal war die Anforderung, dass der Sensor C++-kompatibel ist und dass er mit den Libraries, die die Jetson Nano für die Kommunikation über GPIO-Pins verwendet, angesteuert werden kann. Zuerst habe ich mich für den 3-Achsen-Sensor *MPU9250* von *InvenSense* mit integriertem Accelerometer, Gyroskop und Magnetometer entschieden, weil ich eine dafür geeignete C++-Library von *Simon D. Levy* entdeckt habe. Zusätzlich wollte ich den *BME280*-Sensor von *Bosch* mit der Möglichkeit für Temperaturmessungen einbauen. Jedoch aufgrund dessen, dass sich die Library für den BME280 als veraltet und damit als ungeeignet erwies, musste ich einen anderen Weg einschlagen. Schließlich bin ich auf den *EM7180 Ultimate Sensor Fusion Solution* von *Pesky Products* gestoßen, der die obigen zwei Sensoren kombiniert und ebenfalls ein C++-Interface vom gleichen Ersteller besitzt. Dieser ist im fertigen Gerät enthalten.

Der EM7180-Sensor enthält eine Quaternion, ein Accelerometer, ein Gyroskop, ein Magnetometer und ein Barometer und wird über das Kommunikationsinterface I^2C angesteuert. Da ich diesen noch nicht kannte, habe ich mich mit einem Bekannten von mir, der Elektronik studiert und mir die Funktionsweise des Protokolls erklärt hat, zusammengetan. Um den Sensor in mein Programm zu integrieren, habe ich den Code von Levy (→ **siehe 5.**) in eine objektorientierte Header-Datei umgeschrieben (*usfs_master.h*). Nach kleineren Kompatibilitätsschwierigkeiten war es so möglich, die Messwerte der Sensoren einzeln abzufragen. Die Implementierung vom Anfang bis zum fertigen Programm nahm etwa einen Monat in Anspruch.

3.2.1. Einsatzmöglichkeiten

Die Sensoren kommen im fertigen Gerät auf unterschiedlichster Weise zum Einsatz. Bereits in der Prototyp-Phase existierte die Idee, Tap Detection zu programmieren, um es dem Nutzer zu ermöglichen, auf intuitiver Weise durch Tippgesten mit dem Gerät zu interagieren, beispielsweise um Umgebungsinformationen abzurufen. Dazu wurden mit MatPlotLib die Sensorwerte über einen bestimmten Zeitpunkt aufgezeichnet und geplottet, um Regelmäßigkeiten erkennen zu können. Leider ist es mir nicht gelungen, eine verlässliche Erkennung von Tippgesten zu erstellen, da die Messwerte durch unabsichtliche Bewegungen beeinträchtigt wurden und das Rauschen, vor allem im Gehen, viel zu groß war. Daher wurde dieser Ansatz verworfen und die Gestensteuerung über der Quaternion gelöst, welche die absolute Ausrichtung im dreidimensionalen Raum misst. Der Träger kann das Gerät in verschiedene Richtungen drehen und damit drei Gesten ausführen: Nach rechts neigen für Informationen zu Uhrzeit, Temperatur und Regenwahrscheinlichkeit (kalkuliert über das Verhältnis der Höhe über N.N. und dem Luftdruck), nach links zum Aktivieren des Standby-Modus (Details → **siehe 3.7.**) und nach oben zum Ausschalten des Geräts, was noch einmal bestätigt werden muss. Die Drehungsachsen sind links dargestellt. Zusätzlich erkennt das System bei einer zu starken Neigung nach unten einen Sturz. Wenn

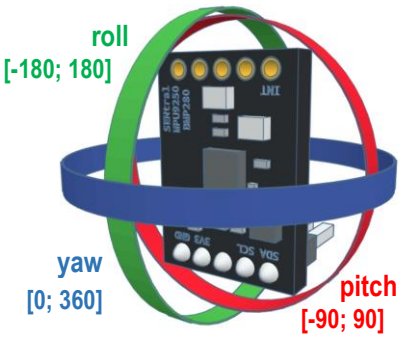


Abb. 6: Darstellung der gemessenen Drehwerte am 3D-Modell

jeweils ein bestimmter Schwellwert für die Ausrichtungen überschritten wird, wird eine Geste registriert.

Zu den wichtigsten Funktionen zählt die Bewegungserkennung, um Gehen und Stehen voneinander zu unterscheiden. Dies kommt bei der Objekterkennung zum Einsatz: Fußgängerampeln werden nur erkannt, wenn der Träger stehen bleibt, um Falscherkennungen zu minimieren und so mehr Sicherheit zu gewährleisten. Dazu werden die letzten 15 Messwerte des Accelerometers in einem Array festgehalten und die empirische Standardabweichung errechnet, die beschreibt, wie sehr die einzelnen Werte von ihrem Durchschnitt abweichen. Da es sich nur um Stichproben handelt, die alle unterschiedlich sind, wird folgende Formel verwendet:

$$s = \sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Beim Gehen ist die Streuung der Messwerte relativ groß, wenn sie hingegen unter einen Schwellwert fällt, wird dies als Stehen gewertet. Über den Mittelwert von Messungen der Höhe über N.N. zu zwei unterschiedlichen Zeitpunkten kann ein schräger Untergrund festgestellt und davor gewarnt werden. Schließlich wird über die Y-Achse des Gyroskops, der dreidimensionale Winkeländerungen misst, das Bild der Kamera zugeschnitten. Wenn der Träger sich dreht, sind die Teile des Bildes, von denen er sich wegdreht, unbedeutend und können zur Leistungsoptimierung der KI entfernt werden. Alle Auswertungen erwiesen sich nach mehreren Optimierungen als akkurat.

3.3. Einrichtung des Entfernungssensors

Wie bereits oben beschrieben war der Entfernungssensor, genauer ein Ultraschallsensor, schon fester Bestandteil des ursprünglichen Prototyps. Dieser diente dem Zweck, die Schwächen der KI, nämlich nicht antrainierte Hindernisse in unmittelbarer Nähe zu erkennen, auszugleichen. Das Programm sollte fortlaufend die Entfernung messen und durch wiederholte Signaltöne dem Träger Rückmeldung geben, ob er sich einem Hindernis nähert und wie weit dieser entfernt ist, ähnlich wie bei einer Einparkhilfe im Auto.

Doch schon letztes Jahr hatte der Ultraschallsensor mit erheblichen Problemen bei der Genauigkeit zu kämpfen, was es fast unmöglich machte, eine verlässliche Distanzrückmeldung zu geben. Am Anfang habe ich den Ansatz verfolgt, den Sensor mit C++ anzusteuern, was sich ebenfalls sehr schwierig gestaltete, da der individuelle Zugriff auf GPIO-Pins sehr umständlich ist. Am Ende erwies sich ein Skript von GitHub als funktionsfähig, die Ungenauigkeiten blieben jedoch erhalten und zum Ausführen wurden Superuser-Rechte benötigt. Auch habe ich eine wasserfeste Alternative zum Sensor mit separater Sonde getestet, welches jedoch noch ungenauer oder gar nicht gemessen hat. Aufgrund dieser Schwierigkeiten beschloss ich, einen gänzlich anderen Ansatz zu wählen und das bereits bewährte I^2C zu verwenden.

Der *LiDAR Lite v3* von *Garmin* ist mit einem Kostenpunkt von ca. 130€ eines der preisgünstigsten LiDAR-Sensoren auf dem Markt und misst Entfernungen mit Hilfe eines Lasers, indem die Zeit zwischen dem Impuls des Senders und dem Auftreffen des Lichts am Detektor ermittelt wird. Für das Modell habe ich mich vor allem wegen

der Messgenauigkeit (±2,5cm) und Reichweite (~40m) entschieden, aber auch, weil es ein C++-Interface, welches ursprünglich für den Raspberry Pi geschrieben worden war, direkt vom Hersteller besaß. Den Code von Garmin (→ **siehe 5.**) konnte ich mit wenigen kleinen Anpassungen auch auf dem Jetson Nano zum Laufen bringen und schließlich in die objektorientierte Header-Form umschreiben (*lidar.h*).

3.3.1. Experimente zur Genauigkeit

Um die Genauigkeit zu testen, wurde in einem Versuch der Sensor in einem Abstand von 30cm vor eine Wand befestigt und 10 Sekunden lang 10-mal pro Sekunde die Entfernung gemessen. Dasselbe wurde mit einem Abstand von 2m sowie 5m wiederholt. In den nachfolgenden Diagrammen sieht man die Veränderung der Messwerte im Laufe der Messungen im Verhältnis zur tatsächlichen Distanz:

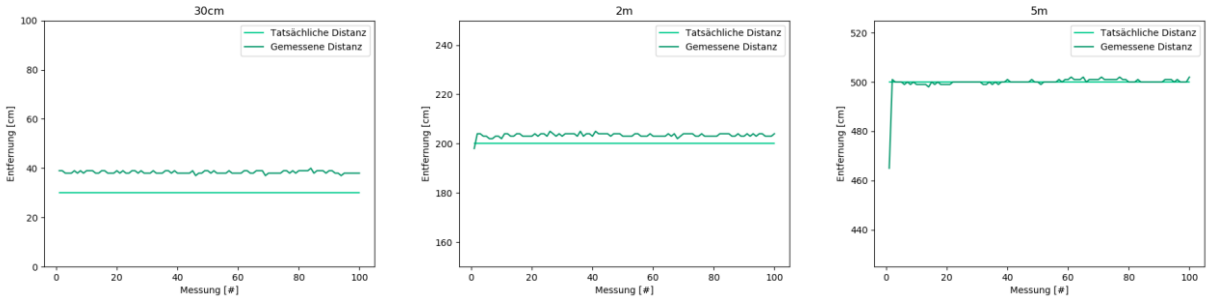


Abb. 7: Ergebnisse der Entfernungsmessungen

Hierbei ist interessanterweise zu erkennen, dass die Distanz tendenziell als zu weit gemessen wird, wobei diese Abweichung vom tatsächlichen Wert mit zunehmender Distanz abnimmt. Bei sehr kurzen Entfernungen wie 30cm kann der Fehler bis zu 10cm betragen, bei 5m hingegen ist die Messung relativ genau. Alle drei Versuche haben gemeinsam, dass die Messwerte sehr wenig schwanken. Daher kann man davon ausgehen, dass der Sensor eine hohe Genauigkeit besitzt, da man im Gehen auf der Straße wohl kaum eine so kleine Distanz wie 30cm erreichen wird.

3.3.2. Abwägung der Vor- und Nachteile

Auch wenn der LiDAR-Sensor am Ende erfolgreich ins Programm integriert und steuerbar wurde, könnten doch Zweifel aufkommen, ob der Wechsel der Art der Distanzmessung vom Ultraschall zum Laser eine sinnvolle Entscheidung war und ob es seinen Zweck erfüllt hat. Im Folgenden möchte ich die Vor- und Nachteile von LiDAR gegeneinander abwägen, um diese Entscheidung zu begründen.

Vorteile:

- LiDAR ermöglicht gleichbleibend genaue Messungen über Zeit.
- Die Reichweite ist durch die Verwendung eines Lasers hoch.
- Genauso wie der Ultraschallsensor ist LiDAR (zumindest in meinem Fall) sicher für andere in der Umgebung, da es keinen Schaden bei direktem Augenkontakt verursacht.

Nachteile:

- Da es sich um ein sehr preiswertes Modell handelt, sendet der LiDAR Lite v3 im Gegensatz zu teureren Sensoren nur einen, gerichteten Laserstrahl aus.

Eine 3D-Erfassung der Umgebung ist somit nicht möglich und es können nur Hindernisse erfasst werden, die direkt vor einem stehen.

- LiDAR hat allgemein eine Schwäche gegen lichtdurchlässige und spiegelnde Oberflächen, da der Laser so nicht in den Detektor reflektiert werden kann.

Die Verwendung eines LiDAR-Sensors in einem Blindenführgerät bringt also deutliche Nachteile mit sich, da beispielsweise Glaswände schwer erkannt werden können. Jedoch zeigen meine Testerfahrungen, dass der gerichtete Laserstrahl dadurch, dass das Gerät und damit der Laser beim Gehen durchgehend leicht hin- und herschwingt, teilweise kompensiert wird. Auch ist zu bedenken, dass GUIDE-Walk nur eine Ergänzung ist und Sehbehinderte immer noch mit Blindenstock ausgestattet sind. Zusammen mit der leichten Bedienbarkeit und der erhöhten Genauigkeit gegenüber dem Ultraschallsensor ich also der Meinung, dass die Nachteile vertretbar und der Einsatz von LiDAR gerechtfertigt ist.

3.4. Programmierung der Audioausgabe

Neben der KI und den Sensoren ist ein wesentlicher Grundpfeiler meines Systems die Ausgabe von Audiosignalen, um dem blinden Träger die Informationen, die von den einzelnen Komponenten gesammelt und verarbeitet wurde, weiterzugeben. Der letztjährige Prototyp war nur in der Lage, die Entfernung durch Pieptöne zu rückmelden, daher habe ich für das fertige Gerät alle Warnungen und Mitteilungen von einem TTS-System einsprechen lassen, es aufgenommen und bearbeitet, damit sie später vom Programm abgespielt werden konnten. Insgesamt sind so 41 Voicelines entstanden, die im GitHub-Repo eingesehen werden können.

Für die C++-Integration habe ich eine Möglichkeit gesucht, Audiodateien asynchron, also parallel zum normalen Programmverlauf abgespielt werden können. Schließlich habe ich eine Audioplayer-Klasse (*audio.h*) mit dem Mixer-Modul der Library *Simple DirectMedia Layer* (SDL) geschrieben, die die Voicelines in den Speicher lädt und in 2 Kanälen simultan abspielen kann. Da die Jetson Nano keine Klinkenbuchse besitzt, habe ich hierfür eine externe Soundkarte mit USB-Anschluss eingebaut.

Schon bald zeichnete sich jedoch ab, dass die vom Programm erzeugten Warnungen auf irgendeine Weise gefiltert werden müssen, da sonst die Warnungen konstant und durcheinander ausgegeben werden. Um dieses Problem zu lösen, werden Warnungen zuerst in ein dynamisches Array, in C++ Vektoren genannt, versetzt. Jede Warnung ist selbst ein Array und besitzt das folgende Format:

$$\{\,[Warnung\_ID]\;[Status]\;[Cooldown]\;[Priorität]\,\}$$

Jede Audiomitteilung besitzt eine eigene ID (festgelegt in der Header-Datei) und einen Prioritätswert von 0-1000. In einer Funktion, der am Ende jedes Programmdurchlaufs aufgerufen wird, werden die Warnungen nach Priorität sortiert. Da alle Warnungen unterschiedliche Prioritätswerte besitzen, kann immer, wenn mehrere Mitteilungen auf einmal eintreffen, entschieden werden, welcher der wichtigste ist. Beispielsweise warnt das System immer eher vor Personen oder Ampeln, als dass es die Anwesenheit von Bänken oder Mülleimern mitteilt und Informationen über Uhrzeit und Wetter werden bei der Erkennung von nahen Objekten sofort abgebrochen. Wenn ein Element des Vektors abgespielt wurde, wird sein Status entsprechend markiert. Ab dann trifft ein Cooldown (Zeitwert) ein, nach dem erst der Eintrag entfernt wird. Dies ist wichtig, da

Warnungen nur gespeichert werden, wenn sich kein anderer des gleichen Typs in der Liste befindet, wodurch eine Häufung von gleichartigen Warnungen vermieden wird. Insgesamt trägt dieses System dazu bei, dass Audiomitteilungen sortiert werden und der Träger nicht übermäßig mit Informationen überladen wird.

3.5. Verworfene Funktionen

In der Einleitung wurde bereits erwähnt, dass ich mir für das fertige Gerät zum Ziel gesetzt habe, den Fokus auf bessere statt mehr Features zu legen. Damit wollte ich sicherstellen, dass das Gerät am Ende eine hohe Funktionalität und Sicherheit gewährleistet. Nicht oder wenig zuverlässige Komponenten musste ich daher entfernen. Im Folgenden möchte ich dafür ein paar Beispiele liefern.

- Ursprünglich war die Implementation eines Gesichtserkennungssystems vorgesehen, die es möglich machen sollte, Gesichter von Freunden oder Bekannten einzuspeichern und sie bei einer Wiedererkennung anzusagen. Aufgrund mangelhafter Möglichkeiten für die C++-Integration habe ich einen Ansatz mit Haar-Cascades ausprobiert, welches jedoch auch nach mehreren Anläufen bestenfalls mittelmäßig funktionierte. Da ich zudem erkennen musste, dass für den Nutzer am Ende gar keine Möglichkeit bestehen würde, mit dem Gerät zu interagieren, um Namen einzuspeichern, wurde die Funktion verworfen.
- Ein wichtiges Feature sollte außerdem die Bordsteinkantenerkennung werden, die den Träger warnen würde, wenn er einer Bordsteinkante gefährlich nahekommen würde. Ich habe mich mit diesem Thema intensiv beschäftigt, jedoch verfolgen die meisten Paper, die ich dazu fand, den gleichen Ansatz. Mittels eines Canny Edge-Detectors sollen zuerst Kanten und dann mit Hilfe des Hough Lines-Algorithmus gerade Linien erkannt werden. Im Gegensatz zu den Forschungspapern wurde dieser Vorgang in meinem Fall aufgrund der Rillen in den Bordsteinen und dem allgemeinen Verkehr erheblich erschwert, weswegen es mir nicht gelungen ist, verlässliche Parameter zu finden.
- Auch war es vorgesehen, zu erkennen, wenn man sich auf einer Grünfläche befindet, damit die KI um dann nicht relevante Klassen reduziert werden kann. Mit einer HSV-Konversion wolle ich die Grünwerte des Bodens analysieren, was jedoch auch an Ungenauigkeit und hohem Rauschen scheiterte.

3.6. Design der Hülle und Stromversorgung

Eine der größten Herausforderungen abgesehen von der Software war es, die Tragweise des Geräts und die Hülle der Komponenten so zu gestalten, dass es gleichzeitig bequem und alltagstauglich sowie praktisch und sicher ist. Für den Prototyp des letzten Jahres habe ich mich nach langem Ausprobieren letztlich für das Tragen um den Hals in einer 3D-gedruckten Box entschieden, der alle Komponenten einschließlich des Akkus enthielt. Zum damaligen Zeitpunkt war es aber ein großer, provisorischer, von Gummibändern zusammengehaltener Klotz, der eher unhandlich und schwer zu tragen war, weswegen mir die Juroren den Vorschlag machten, auf eine Art Bodycam-System, wie es beispielsweise die Polizei benutzt, umzusteigen.

Durch schulische Kontakte habe ich zu diesem Zweck Kontakt zur Bayerischen Polizei und zu Bodycam-Herstellern gesucht, worauf ich entweder keine Antwort oder einen

freundlichen Verweis erhielt, mich anderswo zu erkundigen. Am Ende habe ich beschlossen, beim Tragen um den Hals zu bleiben, vor allem aus zwei Gründen:

- Das Anziehen und Ablegen des Geräts können dadurch viel schneller und unkomplizierter erfolgen und der Träger ist in der Wahl seiner Kleidung nicht eingeschränkt.
- Für das I^2C-Protokoll der Sensoren kann kein Kabel verwendet werden, welches länger ist als 20-30cm, ohne dass das Signal beschädigt wird.

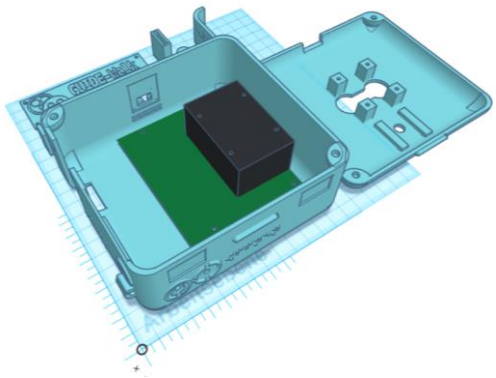


Abb. 8: Abbild des fertigen CAD-Modells der Hülle

Seitdem sind aber viele Iterationen der Hülle entstanden, welche ich insgesamt 5-mal neu entworfen habe, um das System kompakter und komfortabler zu machen. Bilder über diese Entwicklung sowie über das Tragen des Geräts sind im Anhang zu finden (→ **siehe 6.**). Zudem wurde im Gegensatz zum Vorjahr der Akku nun in eine separate Box ausgelagert, die man sich in eine Tasche legen oder auf den Gürtel schnallen kann. Der Akku ist mittels eines Schiebemechanismus aus der Box entnehmbar und wird über ein Spiralkabel mit dem Gerät verbunden. Die Haupt-Box, welche das Jetson Nano und die anderen Komponenten beinhaltet, ist mit den Abmessungen 12,9x12x6,4cm relativ kompakt. Sie besitzt einen leicht abnehmbaren, aber dennoch fest sitzenden Deckel mit Magnetverschluss mit Aussparungen für die Kamera und den LiDAR-Sensor. An der unteren Hälfte befindet sich eine Kopfhörerbuchse, Ringe für die Sicherungsnadeln sowie Stäbe zum Aufwickeln des Kabels. Um Blinden die Handhabung zu erleichtern, sind alle wichtigen Stellen mit Blindenschrift oder, für den Fall, dass der Träger keine Blindenschrift beherrscht, mit simplen haptischen Zeichen markiert. Dies trifft auch auf den Behälter des Akkus zu. Die Jetson Nano ist mit Schrauben im Gehäuse fixiert, wodurch das Gerät für Reparaturzwecke leicht demontiert werden kann. Die Hüllen wurden mit *TinkerCAD* designt und mit den 3D-Druckern unserer Schule gedruckt.

Die Frage nach der Energieversorgung des Geräts hat mich ebenfalls lange beschäftigt, schließlich muss die Jetson Nano bei einem mobilen Design mit ausreichend Strom versorgt werden. Die Leistung des Mini-Computers liegt zwischen 5-10W, wodurch es theoretisch problemlos über den eingebauten Micro-USB-Port durch eine Powerbank betrieben werden könnte. Diese Lösung habe ich beim letztjährigen Prototyp angewendet, wie sich jedoch beim Ausführen leistungsintensiverer Programme herausstellte, liegt der Verbrauch tatsächlich höher, weswegen die von USB bereitgestellten 5V⎓2,1A nicht ausreichen. Dies wurde sowohl durch meine Experimente als auch durch Nvidia selbst auf ihren Foren bestätigt. Höchstwahrscheinlich sind die Sensoren und Peripherie die Ursache, welche zusätzlichen Strom verbrauchen. Daher war ich gezwungen, auf den DC-Barrel-Jack umzusteigen.

Die DC-Buchse (5,5mm/2,1mm) ermöglicht es, der Jetson Nano bis zu 5V⎓4A, also die doppelte Stromstärke, zu liefern. Dazu müssen die J48-Pins durch einen Jumper kurzgeschlossen werden. Ich hatte Schwierigkeiten damit, eine Powerbank zu finden,

die diese Menge an Strom liefern konnte und zusätzlich vergleichsweise handlich und mobil war. Daher habe ich versucht, ein zweites Exemplar der im letzten Jahr von mir verwendeten USB-Powerbank zu besorgen und die beiden Ausgänge parallel zu schalten. Dieses Vorhaben scheiterte jedoch, da der Strom nicht mal ausreichend war, um die Jetson Nano einzuschalten, vermutlich, weil die Powerbanks ihren Output drosselten. Auch der Betrieb durch USB-C war wegen des fehlenden Ports am Board nicht möglich. Schließlich habe ich dank Hilfe von Foren doch einen Akku (Kapazität: 8300 mAh) finden können, der für Blinde einfach zu bedienen ist und eine DC-Buchse mit einem Output von 12V⎓6A besitzt. Um diesen Strom herunterzustufen, habe ich einen Transformator in das Gehäuse integriert und die Kabel miteinander verlötet.

3.7. Optimierung des Programms

Nach erfolgreicher Implementierung der einzelnen Bausteine galt es, das Programm zu vervollständigen und soweit wie möglich zu optimieren. Mein Ziel war dabei, eine möglichst kurze Reaktionszeit (sprich viele Bilder pro Sekunde) und gleichzeitig einen möglichst geringen Energiekonsum zu erzielen. Abgesehen von ein paar Ausnahmen laufen diese Optimierungsvorgänge vollständig im Hintergrund und regeln sich selber. Ich habe mehrere neue Lösungen entwickelt, einiges aber auch vom letztjährigen Prototyp übernommen und in C++ umgeschrieben.

Dazu gehören unter anderem die KI-Optimierungen. Zusätzlich zum bereits beschriebenen eigenen Training und damit Reduzierung der Modellgröße werden im Voraus Bilder, die zum Auswerten ungeeignet sind, aussortiert. Dies geschieht beispielsweise mit der Errechnung der Laplace'schen Varianz der Kamerabilder, dessen Wert den Grad der Unschärfe abbildet. Dazu wird ein kantenempfindliches Kernel am Bild angewendet und die quadratische Standardabweichung der Ergebnisse berechnet. Wenn es kleiner ist als ein bestimmter Schwellwert, wird ein neues Bild aufgenommen. Dies war nötig, weil ich bei meinen Tests festgestellt habe, dass die Bilder der Kamera bei zu schnellen Bewegungen oder wenig Licht schnell verwaschen werden. Allgemein leidet die Bildqualität des verwendeten Kameramoduls bei Lichtmangel enorm, weswegen ich mich entschieden habe, die Objekterkennung bei Nacht zu deaktivieren. Dazu werden die Bilder auf eine Größe von 64x35px verkleinert und die Helligkeitswerte im oberen Drittel im HSV-Farbraum analysiert. Ist die Helligkeit zu lange zu gering, wird der KI-Pointer auf einen Nullpointer gesetzt und die KI so ausgeschaltet. Dies wird dem Träger durch einen Hinweis mitgeteilt. Auf neuer Weise hingegen konnte ich das Gyroskop des Bewegungssensors einsetzen, um die Seiten der Kamerabilder zu croppen, wenn man sich dreht. Auch habe ich es geschafft, die KI-Berechnungen auf der GPU durchführen zu lassen, was den Vorgang um einiges beschleunigt (Details: → **siehe 3.7.1.**). Dafür musste ich einen neues Energieprofil für die Jetson Nano definieren und Grenzen für die Taktfrequenzen der CPU (918 Mhz) und GPU (640 Mhz) begrenzen, weil der Computer sonst wegen zu hohem Energiekonsum und Unterspannung abstürzte. Die Leistung wurde dadurch zwar etwas gedrosselt, diese Maßnahme war aber zwingend notwendig.

Ein sehr wichtiges Tool für die Verbesserung der Performance war Multithreading. Die Verwendung mehrerer Threads hatte ich bereits für den Prototyp geplant, jedoch aufgrund von Zeitmangel nicht umgesetzt. Zum Glück gestaltet sich Multithreading mit C++ um einiges unkomplizierter als mit Python, wo dieser Vorgang höchstens simuliert

werden wann. Um ein grundsätzliches Problem zu lösen, habe ich den Kamera-Stream in einen separaten Thread ausgelagert. Wenn die GStreamer-Pipeline nämlich mehr Bilder pro Sekunde liefert, als die KI in der gleichen Zeit verarbeiten kann, stauen sich die Aufnahmen an und es entsteht eine zeitliche Verschiebung zwischen den gerade aufgenommenen und den ausgewerteten Bildern. Daher produziert der Thread einen konstanten Stream von 30 Bildern pro Sekunde, aus dem sich das Hauptprogramm jederzeit bei Bedarf das nächste Bild nehmen kann. Die Aufnahmerate der Kamera ist so unabhängig vom Hauptprogramm, wie das nächste Schaubild zeigt:

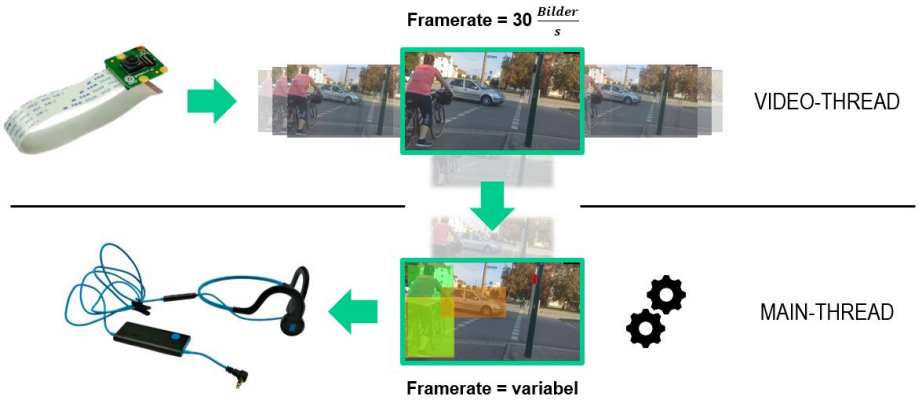


Abb. 9: Schematische Darstellung des Kamera-Streams im separaten Thread

Auch die Distanzmessung und -rückmeldung läuft in einem anderen Thread, damit der zeitliche Abstand zwischen den Signaltönen nicht von der Geschwindigkeit des restlichen Programms beeinflusst wird und parallel dazu laufen kann. Um Abstürze zu verhindern, habe ich Funktionen implementiert, die Fehler bei den Sensoren oder anderen Komponenten erkennen, eine Benachrichtigung senden und das Programm kontrolliert herunterfahren. Dabei werden die Instanzen der einzelnen Klassen aus dem Speicher geleert und die Threads beendet. Falls es doch zu einem Absturz oder unerwartetem Fehler kommen sollte, kann der Programmverlauf mit einem eingebauten Logger, der alle relevanten Vorgänge in einer Log-Datei festhält, getraced werden. Bestimmte Optionen kann der Nutzer auch selber steuern. Durch eine Drehung des Geräts nach links wird das Programm in den Energiesparmodus versetzt, wodurch alle Komponenten, einschließlich der Kamera und des LiDARs deaktiviert und später wieder aktiviert werden können. Dies ist sinnvoll, wenn man sich drinnen befindet (zum Beispiel in einem Geschäft oder im Bus) oder das Gerät zeitweise nicht braucht. Auch das Ein- und Ausschalten ist so benutzerfreundlich wie möglich gestaltet: Durch eine Integration in die Autostart-Applikationen von Linux startet das Programm nach dem Anstecken des Akkus automatisch. Mit zwei Neigungen nach oben kann das System wieder heruntergefahren werden.

3.7.1. Messung der Laufzeit und Performance

Als das Hauptprogramm fertiggestellt und funktionsfähig war, wollte ich das System testen, um eine zusammenfassende Statistik über die Performance zu erstellen und zu überprüfen, wie effektiv die oben aufgeführten Optimierungen letztendlich waren. Mit der Implementierung eines einfachen FPS-Counters habe ich gemessen, wie viele Programmdurchläufe in einer Sekunde gemacht werden können. Die Ergebnisse sind im Graphen auf der nächsten Seite abgebildet.

- Nur CPU: Ø **0,6 FPS**
- Mit GPU-Beschleunigung: Ø **6,5 FPS**

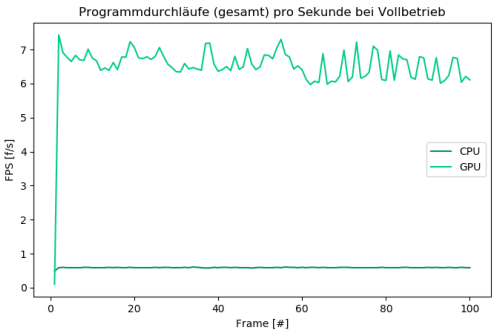


Abb. 10: Ergebnisse der Geschwindigkeitsmessung

Am Verlauf des FPS-Wertes ist deutlich zu erkennen, dass dieser im GPU-Modus mehr, aber nicht übermäßig schwankt und das Programm dementsprechend größtenteils flüssig und ohne Lags läuft. Bei Verwendung der GPU liegt die Reaktionszeit bei $\frac{1}{6{,}5}s = 0{,}15s$. Dies mag vielleicht gar nicht so kurz klingen, doch für den alltäglichen Verkehr auf der Straße ist diese Zeit nach den Erfahrungen von mir und den Testpersonen (Details: → **siehe 3.8.**) völlig ausreichend, besonders wenn man beachtet, dass das Gerät nur als eine Ergänzung für die Wahrnehmung des Blinden gilt.

Weiterhin habe ich die Akkulaufzeit des Geräts bei Vollauslastung und bei aktiviertem Standby-Modus zu bestimmen versucht. Auf rechnerische Weise kann die Laufzeit eines Akkus folgendermaßen berechnet werden:

$$t_{Akku} = \frac{U \cdot Q}{P} = \frac{5V \cdot 8{,}3Ah}{10W} = 4{,}15h$$

Dieser Wert ist jedoch aus mehreren Gründen ungenau: Zum einen ist der Verbrauch der Jetson Nano nur ein Schätzwert, zum anderen werden hier physikalische Größen wie Verluste bei der Transformation usw. nicht miteinbezogen. Daher war eine experimentelle Bestimmung nötig. Über die LED-Leuchten der Powerbank habe ich beobachtet, wie schnell dieser sich entleert, die Zeit gemessen und hochgerechnet. Im folgenden Diagramm sind die Ergebnisse dargestellt:

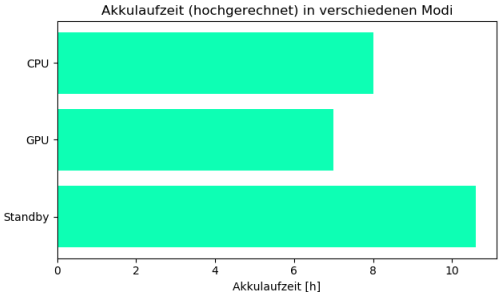


Abb. 11: Messungen der Akkulaufzeit

Bei einer normalen Verwendung des Geräts ohne Unterbrechungen kommt man also auf eine gute Akkulaufzeit von ungefähr 7-8 Stunden am Stück. Dies ist ein solider Wert, der den größten Anteil der Zeiten, in denen man zu Fuß unterwegs ist, ausreichend abdeckt.

3.8. Erfahrungen von Testpersonen

Für das Erreichen meines diesjährigen Ziels, also der Entwicklung eines KI-gesteuerten Alltagshelfers für Blinde, war es mir besonders wichtig, Feedback aus erster Hand einzuholen, um Aspekte zu verbessern, die mir bislang noch unbekannt waren. Dazu habe ich Kontakt zu verschiedenen Blindeninstitutionen und anderen Einrichtungen gesucht und mein Gerät von zwei sehbehinderten Menschen im Einsatz auf der Straße testen lassen.

Prof. Dr. Rainer Schliermann (Professor für Erziehungswissenschaften und sozial-wissenschaftliche Forschungsmethoden, Ostbayerische Technische Hochschule (OTH) Regensburg) sieht bei der Erkennung von unerwartet oder plötzlich eintretenden Ereignissen wie Hindernissen oder roten Ampeln in der Objekterkennung mit künstlicher Intelligenz einen sinnvollen Ansatz. Auch befürwortet er die möglichst

geringe Einschränkung des Hörvermögens des Trägers. Genau da sieht er aber auch einen Kritikpunkt, denn das System gab ihm zu viele und teils nicht verlässliche Informationen. Auf diese Anregung hin wurde die KI mit zusätzlichen Aufnahmen und neuem Training verbessert und die Bordsteinkantenerkennung mangels funktionaleren Alternativen entfernt. Insgesamt, so versichert er, kann das Blindenführgerät seiner Einschätzung nach definitiv einen Teil zum Inklusionsprozess beitragen. *Rudolf Pichlmeier* (Gruppenleiter der Bezirksgruppe Oberpfalz des Bayerischen Blinden- und Sehbehindertenbundes) betont, dass ein Assistenzgerät wie meines Betroffene bei Einschnitten durch Menschen in der Umgebung wie Hektik oder (oftmals unbeabsichtigt) fehlender Rücksichtnahme unterstützen könne. Er lobt daher vor allem den Einsatz der KI für Objekt- und Ampelerkennung, kritisiert aber die unzureichende Filterung und die „Flut“ an Informationen, die dem Träger weitergegeben werden. Gemäß seinem Vorschlag wurde deshalb das System zur Priorisierung von Warnungen überarbeitet. Auch bemängelte er, dass eine räumliche Vorstellung zu erkannten Gefahren fehlte, weswegen im fertigen Programm die Distanzmessung mit der Objekterkennung verknüpft wurde, um nur dann zu warnen, wenn sich ein Hindernis in unmittelbarer Nähe befindet.

Zusammenfassend waren die Einsichten und Erkenntnisse aus diesen Gesprächen und Tests für mich überaus hilfreich und haben wesentlich zur Verbesserung des Geräts in vielerlei Aspekten beigetragen.

4. ZUSAMMENFASSUNG UND ZUKUNFSAUSSICHTEN

Nachdem nun die Zusammensetzung und Funktionsweise des Blindenführersystems eingehend erläutert wurden, kann man meiner Einschätzung nach festhalten, dass mein Ziel, einen sicheren Alltagshelfer für Blinde zu kreieren, dieses Jahr letzten Endes erreicht wurde. Besonders die KI hat sich im Vergleich erheblich weiterentwickelt und arbeitet viel genauer. Dieses Projekt war eine außerordentlich lange Reise mit vielen Hindernissen und Rückschlägen, in die sehr viel Arbeit geflossen ist. Ich bin äußerst froh und dankbar, dass ich mit der Unterstützung zahlreicher Personen dieses Gerät fertigstellen konnte, um Blinden und Sehbehinderten den Alltag zu erleichtern. Zudem konnte ich in diesen zwei Jahren so viel neues Wissen ansammeln wie nie zuvor.

Zugleich bin ich mir bewusst, dass kein System perfekt ist und es auch an meinem Gerät noch zahlreiche Aspekte zu verbessern gibt. Als Ersteller einer derartigen Software hat man meiner Meinung nach eine große Verantwortung dessen Nutzern gegenüber. Jedoch blicke ich der Zukunft des GUIDE-Walk aufgeschlossen und erwartungsvoll gegenüber. Um daraus ein marktfähiges Produkt zu schaffen, welches für einen günstigen Preis allen Betroffenen angeboten werden kann, könnte man das Gerät z.B. mit einem individuellen Chipset auf einem Custom-PCB in eine Brille integrieren. Was die Zukunft auch immer bringen mag, so möchte ich mich übereinstimmend mit den betroffenen Testern weiterhin dafür einsetzen, dass KI eingesetzt wird, um Menschen zu helfen und ihnen durch das „Auge des Computers“ vielleicht eine neue Perspektive geben zu können.

5. QUELLEN- UND LITERATURVERZEICHNIS

Internetquellen:

- **GitHub-Repositories:**
  ML-Framework Caffe: https://github.com/weiliu89/caffe/tree/ssd
  MobileNet-SSD for Caffe: https://github.com/chuanqi305/MobileNet-SSD
  MobileNet-SSD v2 for Caffe: https://github.com/zhanghanbin3159/MobileNetV2-SSD
  LabelImg: https://github.com/tzutalin/labelImg
  Cross-Platform Data Bus: https://github.com/simondlevy/CrossPlatformDataBus
  USFS-Library: https://github.com/simondlevy/USFS
  LiDARLite v3-Library: https://github.com/garmin/LIDARLite_RaspberryPi_Library
- **Verwendete Datasets:**
  MS-COCO: https://cocodataset.org/#home
  Pascal-VOC: http://host.robots.ox.ac.uk/pascal/VOC/
  Ampelpilot: https://github.com/patVlnta/Ampel-Pilot
- **Foren:**
  Jetson Nano-Forum: https://forums.developer.nvidia.com/
  Stack Overflow: https://stackoverflow.com/
  AskUbuntu: https://askubuntu.com/
  Unix & Linux Stack Exchange: https://unix.stackexchange.com/
  Raspberry Pi-Forum: https://www.raspberrypi.org/forums/
- **Sonstige Guides und Quellen:**
  Caffe-Ubuntu: https://qengineering.eu/install-caffe-on-ubuntu-20.04-with-opencv-4.4.html
  SSD on C++: https://gist.github.com/yiling-chen/7d36389192d54f89a5fe0b810ac7bdf3
  OpenCV Documentation: https://docs.opencv.org/4.4.0/

Literaturverzeichnis:

José, J., Farrajota, M., Rodrigues, J., & du Buf, J. M. (Jan 2010). *A vision system for detecting paths and moving obstacles for the blind.* Von ResearchGate: https://www.researchgate.net/publication/216435222_A_vision_system_for_detecting_paths_and_moving_obstacles_for_the_blind abgerufen

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (29. Dez. 2016). *SSD: Single Shot MultiBox Detector.* Von arXiv: https://arxiv.org/abs/1512.02325 abgerufen

Rodrigues, J., du Buf, J. M., & Castells, D. E. (Jan. 2010). *Obstacle detection and avoidance on sidewalks.* Von ResearchGate: https://www.researchgate.net/publication/216435190_Obstacle_detection_and_avoidance_on_sidewalks abgerufen

Bildnachweis:

[S. 6] Liu et al.: *A comparison between two single shot detection models* https://arxiv.org/pdf/1512.02325.pdf, S. 4

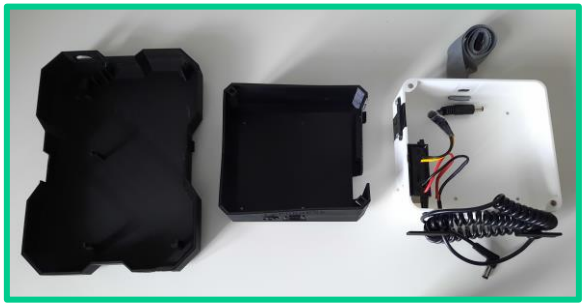
[S. 7] Sandler et al.: *Comparison of general-purpose computer vision neural networks* https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html

Alle anderen Abbildungen und Fotografien sind Eigentum des Urhebers dieser schriftlichen Arbeit.

6. ANHANG

Anh. 1: Tragweise des GUIDE-Walk in voller Ausstattung

Anh. 2: Abbildung der Haupt- und Akku-Box von allen Seiten



Anh. 3: Entwicklung des Box-Designs vom Prototyp zum fertigen Gerät

7. UNTERSTÜTZUNGSLEISTUNGEN UND DANKSAGUNGEN

Vielen Dank an all diejenigen, die mich bei der Planung und Verwirklichung des Projekts unterstützten und weiterhalfen! Insbesondere möchte ich danken:

- Meinen Eltern für die guten Ideen, ihre Hilfsbereitschaft und der finanziellen Unterstützung meines Vorhabens,
- Dem Gymnasium der Regensburger Domspatzen, Schulleiterin Christine Lohse und meinem Projektbetreuer René Grünbauer, ohne deren Hilfe und Unterstützung mein Projekt nicht möglich gewesen wäre,
- Dem Jugend Forscht Sponsorpool Bayern und dem EDIsys Kft., die mir ein Budget zur Verwirklichung des Projekts bereitgestellt haben,
- Den Testern Rainer Schliermann (OTH Regensburg) und Rudolf Pichlmeier (BBSB e.V. Oberpfalz) für ihre Kooperation, Mühen und konstruktive Kritik,
- Ulrike Weimer (Blindeninstitut Regensburg) für ihre Vermittlungstätigkeit,
- Moritz Walker und Christoph Högl für ihre Beratung in fachlichen Fragen,
- Yannick Rittner, Paul Kutzer und Patrick Soller für ihre spontane Hilfsbereitschaft.