BlindSpot – Dokumentation

Unser Projekt BlindSpot soll eine Person vor nahen Objekten warnen und erforschen, ob ihre Umgebung durch eine Tonkulisse wiedergeben werden kann. Um das umzusetzen, erzeugen wir ein Tiefenbild aus zwei nebeneinander angebrachten Kameras, analysieren dieses und machen nahe Objekte dreidimensional hörbar.
Das Tiefenbild wird mit Python erzeugt, die Analyse erfolgt mit C++ und die Audioausgabe läuft über Javascript. Alle Code-Bestandteile laufen dabei über denselben Rechner.

Konstruktion

Wir verwendeten zwei Speedlink Snappy Smart Webcams, die nebeneinander fest auf einer Holzunterlage angebracht wurden, um ein verrutschen der Perspektive zu vermeiden. Die Holzunterlage lackierten wir vorab in schwarz. Das Anbringen an einen Helm o.ä. Halterung wären damit leichter zu bewerkstelligen, ohne dabei die Position der Kameras zueinander verändern zu
können.

Abbildung 1: Kameraaufbau

Kamera Kalibrierung und Tiefenbild-erzeugung mit Python

Da das ursprüngliche Ziel des Projekts darin bestand, die Videodaten mit einem Raspberry Pi zu verarbeiten, entschieden wir uns, für die Kalibrierung der beiden Webcams und die Berechnung des Tiefenbildes, die Programmierung mit Python durchzuführen.

Diese Entscheidung resultierte hauptsächlich aus der Tatsache, dass wir diverse Schwierigkeiten hatten, C++ und OpenCV auf dem verfügbaren Raspberry Pi zu installieren (u.a. fehlende Binary-Verlinkungen, fehlendes C-Make, zudem einige Kompilierungsfehler) und die Python-Installation auf dem Raspberry Pi wesentlich schneller und problemloser war.
Ein weiterer ausschlaggebender Grund für Python, ist die sehr gute und ausführliche Dokumentation der OpenCV Funktionen.

Als im Verlauf des Projekts deutlich wurde, dass die Rechenleistung des Raspberry Pi nicht ausreichen würde, um eine brauchbare Bildrate für das Analysieren zu erhalten, hatten wir bereits einen funktionierenden Code geschrieben und nicht mehr genügend Zeit, diesen noch in C++ umzuwandeln. Dementsprechend werden die Kamera-Kalibrierung und Tiefenbild-Erstellung mittels
Python ausgeführt. Für die Bild-Analyse nutzten wir dann C++ (mehr dazu s.u.).

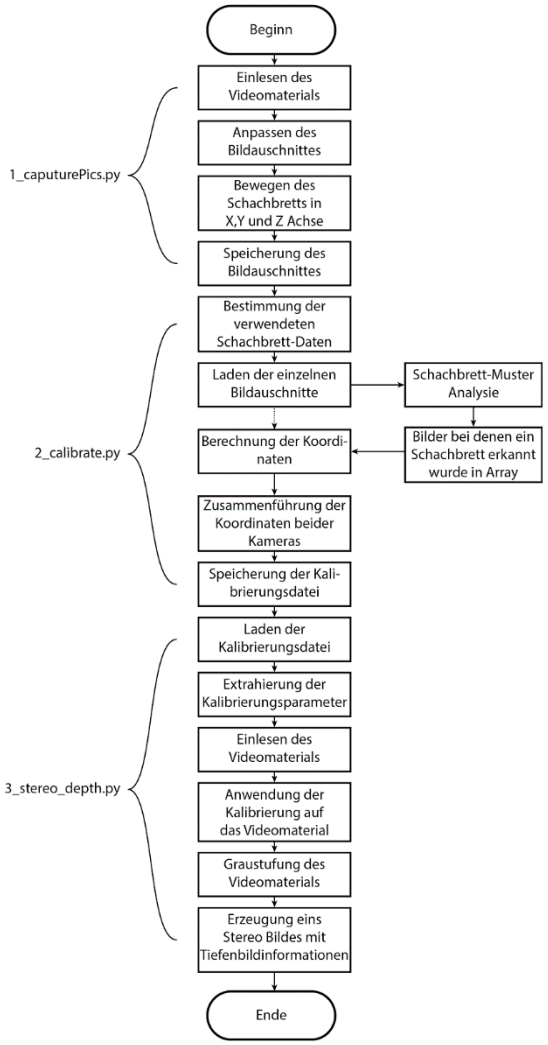

Abbildung 2: vom Capture zum Tiefenbild

Für den ersten Schritt der Kalibrierung wird das Python-Skript *1_capturePics.py* verwendet. Hier wird jeder dritte Frame der beiden Webcams als jpg-Bilddatei mit gleichen Namen abgespeichert. Dafür werden die Kameras mit der openCV-Methode *cv2.VideoCapture()* eingelesen, hochskaliert (damit die Schachbrett-Erkennung im weiteren Schritt besser erfolgt), beschnitten und schließlich in einem vorbestimmten Pfad mit der Methode *cv2.imwrite()* abgespeichert. Während man das Skript ausführt und die Bilder gespeichert werden, muss man ein Schachbrett-Muster vor die Kamera halten und mit diesem die Bildgröße und Bildtiefe abmessen. OpenCV bietet für diese Kamera-Kalibrierung bereits fertige Methoden an, die sowohl die Schachbrett-Erkennung, als auch die Erstellung einer Kalibrierungsdatei durchführen. Für diese Berechnung haben wir ein weiteres Python-Skript *2_calibrate.py* geschrieben.

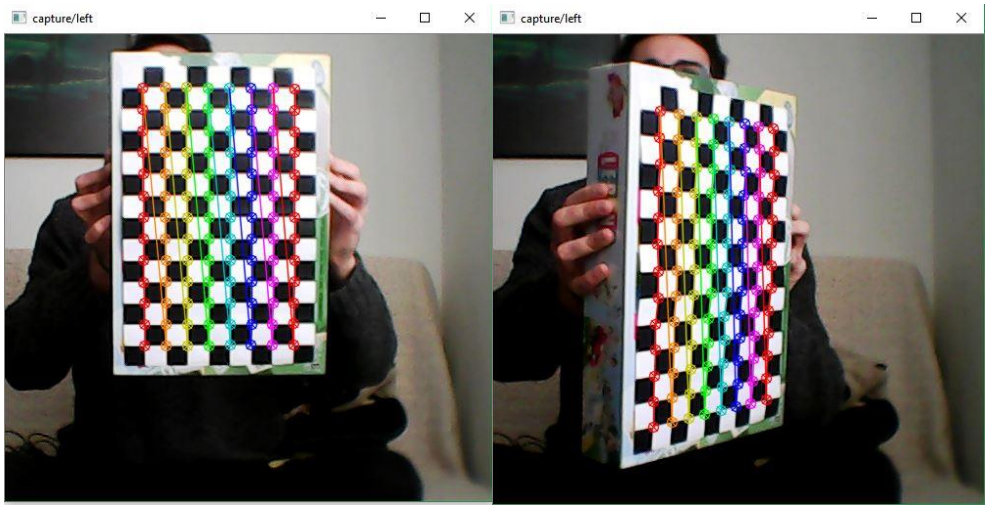

Abbildung 3: Analyse der Aufnahmen und Kalibrierung der Kameras

Hier werden zunächst die Schachbrett-Daten festgelegt und anschließend die Funktion *analyzedImagesforChessboard()* aufgerufen. Hier werden nacheinander die zuvor gespeicherten Bilder geladen und auf das Schachbrett-Muster analysiert. Wird das Schachbrett bei beiden Bildern (für die rechte und die linke Webcam) erkannt, so wird das Bild als verwendbar markiert und kann im folgenden Schritt für die Berechnung der Kalibrierungsdatei verwendet werden.
Damit nicht zu viele Bilder für die Kalibrierung verwendet werden und somit eine sehr lange Berechnung durchgeführt werden muss, haben wir hier festgelegt, dass mit maximal 100 zufällig ausgewählten Bildern Kalibriert werden soll. Im nächsten Schritt nutzen wir die Funktion *getImagePointsAndMatchingObject()* und berechnen in dieser mit OpenCV-Methoden die benötigten Koordinaten für die Bildpaare, um anschließend eine Tiefenbild-Erkennung durchzuführen.

Die berechneten Daten, wie z.B. die X und Y-Koordinaten des Bildes werden abschließend abgespeichert und können für die Tiefenbild-Berechnung im Skript *3_stereo_depth.py* geladen werden. Für diese Zwischenspeicherung nutzen wir *NumPy*, eine Programmbibliothek für die Programmiersprache Python, die eine einfache Handhabung von Vektoren, Matrizen oder generell großen mehrdimensionalen Arrays ermöglicht und auf die wir durch Recherche aufmerksam geworden sind.

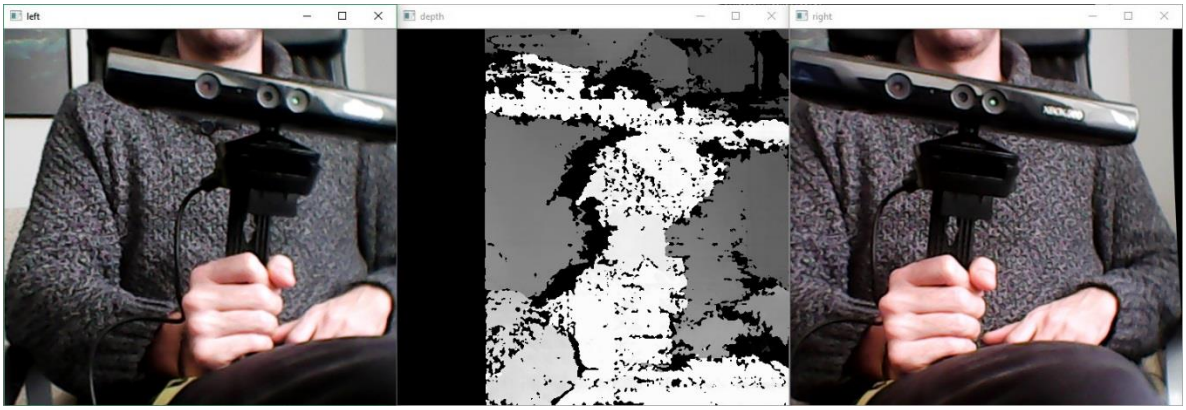

Abbildung 4: Ergebnis der Tiefenbildberechnung

Die nun vorliegende Kalibrierung wird in dem Skript *3_stereo_depth.py* geladen und die enthaltenen Parameter einzeln extrahiert. Daraufhin wird wieder über die openCV-Methode *cv2.VideoCapture()* das Videomaterial der beiden Kameras separat eingelesen welches dann aber über *cv2.remap()* mit den Parametern der Kalibrierung vereint wird. So bekommen wir zwei Kalibrierte Live-Bilder die wir über *cv2.cvtColor()* grau einfärben und folgend mit dem Stereo Matcher "StereoSGBM" ein Stereo Bild erzeugen. Mit diesem Stereo Matcher können wir Ungleichheiten herausfiltern und wir zuvor über *cv2.StereoSGBM_create()* konfiguriert anzeigen lassen. Am Ende speichern wir das so entstandene Tiefenbild zur Weiterverarbeitung in C++ als jpg-Datei ab und lassen uns die Live-Bilder (Rechts, Links) sowie das fertige Tiefenbild in 3 Separaten Frames ausgeben.

Verarbeitung des Tiefenbilds mit C++

Nachdem aus den Stereobildern ein Tiefenbild in Graustufen erzeugt wurde, werden daraus sinnvolle Werte für eine dreidimensionale Audioausgabe generiert. Nach Verarbeitung werden die ermittelten Daten per MIDI an den Webserver, der für die Audioausgabe zuständig ist, weitergeleitet. Relevant sind hierbei die zweidimensionale Position der Objekte, sowie die Entfernung zu den Kameras, also die Tiefe, die durch die Helligkeit im Tiefenbild dargestellt wird. Aus diesen Werten kann ein Objekt im Kamerabild dreidimensional geortet werden.

Die Anwendung ist in C++ geschrieben und mit Qt erstellt. Für die Bildverarbeitung wird zusätzlich die OpenCV-Bibliothek verwendet. Das Erzeugen und Versenden der MIDI-Daten geschieht mit Hilfe der Drumstick-Bibliothek. Als interner MIDI-Output wird die Software LoopBe1 genutzt.
Die im Projekt eingesetzte OpenCV-Sammlung liefert mit der Videoengine-Klasse und der Videoprocessor-Klasse bereits eine gute Grundlage für die Bildverarbeitung. Da die Tiefenbilder als Bilddateien vorliegen, musst die Videoengine-Klasse lediglich dafür umfunktioniert werden, wiederholt Einzelbilder anstatt eines einzelnen Videostreams zu öffnen. Die Videoprocessor-Klasse liefert die Basisklasse für den eigentlichen Verarbeitungsprozess, der nach jedem Öffnen eines Bildes ausgeführt wird. Der eingesetzte Processor gibt allerdings kein Output-Image zurück, sondern einen 64-Bit Integer der aus acht Datenbytes für vier MIDI-Nachrichten besteht.
Um die Verarbeitung zu starten muss zunächst im Menü unter Datei->öffnen... das gewünschte Bild geöffnet werden und mit Start->start beginnt der Prozess.

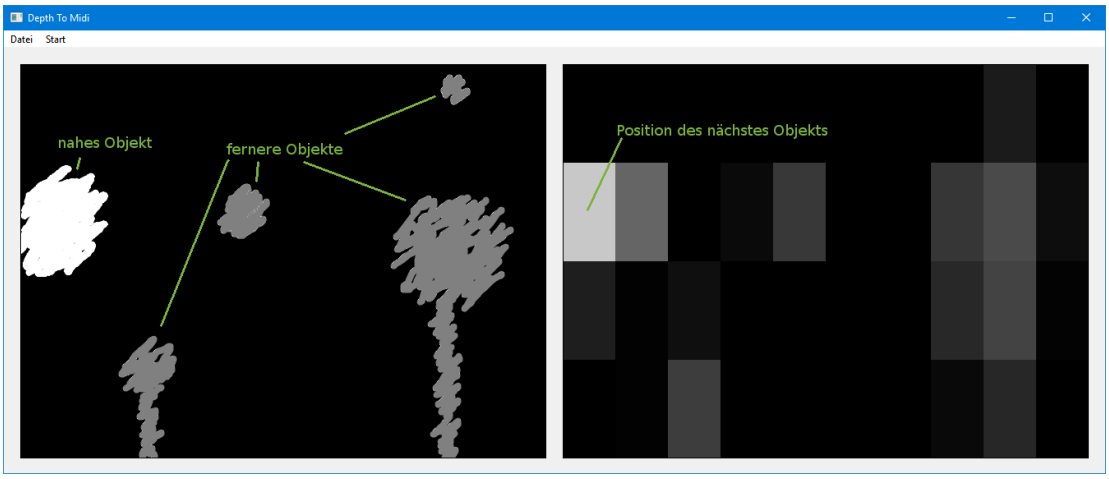

Abbildung 5: Rasterung eines (unechten) Tiefenbildes

Jedes Tiefenbild wird zunächst in ein vorher definiertes Raster eingeteilt. In der Präsentation wurde ein Raster von zehn Spalten und vier Reihen gewählt. Dieses Raster steht im späteren Verlauf für die möglichen Positionen der Audioquellen. Anschließend wird für jedes Feld im Raster der mittlere Grauwert (0-255) ermittelt. Dieser Grauwert wird in eine definierte Skala von Werten von 0 (ganz nah) bis 20 (maximal entfernt) umgerechnet und als Resultat mit dem zugehörigen Rasterindex gespeichert.

Die vier größten Werte werden für die MIDI-Übertragung als 64-Bit Integer nach dem Prozess ausgegeben, pro Wert je ein Byte für den Rasterindex und je ein Byte für den Tiefenwert (Skala 0-20). Der Rasterindex wird dann im ersten Datenbyte der MIDI-Nachricht und die Position im zweiten eingesetzt. Da pro Datenbyte sieben Bit für eigene Werte zu Verfügung stehen, könnten maximal Werte von 0-127 genutzt werden, also wäre höchstens ein Raster mit 128 Indices möglich sowie eine Tiefenwerteskala von 0-127. Da die Tiefenwerte vom Benutzer exakter unterscheidbar sein sollten, wurde die grobe Skala von 0-20 verwendet. Während der Entwicklung wurden auch andere Rasterungen ausprobiert, von 4x3 bis 16x8.

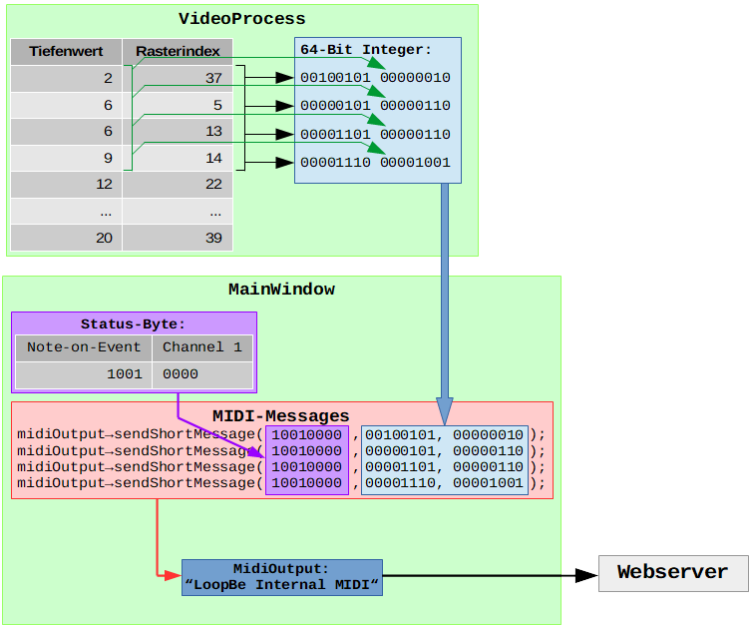

Abbildung 6: Kodierung der Bilddaten als MIDI-Nachricht

Höhere Auflösungen sind zwar genauer, allerdings verfügte die Anwendung zum Präsentationszeitpunkt nicht über eine Objekterkennung, also eine Methode zum Zusammenfassen gleicher Werte, die nebeneinander auftreten. Große Objekte werden daher einfach als mehrere Objekte dargestellt. Aus den vorher erzeugten MIDI-Datenbytes werden nun bis zu vier MIDI-Nachrichten erzeugt, die immer das Statusbyte noteOn/Kanal 0 (10010000) verwenden.

Gesendet wurde in der Präsentation allerdings immer nur der erste Datensatz, also Position und Tiefenwert des nächsten Objekts. Das Raster mit den skalierten Graustufen wird außerdem noch in der grafischen Oberfläche der Anwendung neben dem Eingangsbild dargestellt.

Audioausgabe mit JavaScript

Bei den Tests wurde herausgefunden, dass nur wenige Sounds vom Hörer lokalisiert werden können und auch nur, wenn sie leicht verzögert abgespielt werden. Desto weniger Sounds, desto genauer kann man sich auf diese konzentrieren, weshalb für unsere Präsentation nur eine Soundquelle (die am nächsten dran liegende) abgespielt wurde.

Um die Sounds dreidimensional hörbar zu machen verwenden wir das Resonance Audio Web SDK. In der *audio3d.js* werden die Sounds in der Audio Resonance Umgebung erstellt. Mit der *soundObjectNumber* Variable wird festgelegt, wie viele Soundquellen erstellt werden. Sie muss mit der Anzahl der Objekte bzw. MIDI Messages übereinstimmen, die pro Frame von C++ zur Webanwendung übertragen werden. Die Klasse *audioGridBox* dient als Zwischenspeicher für die Koordinaten der Resonance Audio Sources. Im Gegensatz zur richtigen Source können so die Koordinaten nicht nur gesetzt sondern auch wieder abgefragt werden.

Die Methode *createSoundElements()* erstellt die Audiosounds und die Resonance Audio Sources, mit Berücksichtigung der festgelegten Sound (WAV) Datei. Dabei wird erst eine neue Audiodatei erzeugt, diese in eine *MediaElementSource* umgewandelt und anschließend zur Audio Resonance Source verbunden. Die Funktion *playSound()* spielt die Audiofiles verzögert ab, indem *playDelayed()* aufgerufen wird.
Die Variable time gibt die Delayzeit an, bis der nächste Ton zu hören ist.

Damit nun mit Hilfe der *audio3d.js* Sound abgespielt wird, muss erst eine MIDI Message eingehen. *midi.js* ruft bei Empfang einer Message die Funktion *onMIDIMessage()* auf. Das MIDI Signal wird hier zerlegt in Byte1 und Byte2, das erste Byte (an Stelle 0) ist für uns nicht relevant, da wir nur “note on” Messages von der C++ Anwendung bekommen. In Byte1 ist der Index, um die X und X-Koordinaten der Quelle zu errechnen. In Byte2 ist der Entfernungsindex (z-Koordinate).
Byte1 und 2 werden an die Methode *onMIDIiPlay-ResonanceAudio()* übergeben, diese befindet sich in der *midiCalculator.js*.
Hier wird *calculatePosition()* aufgerufen, um die Meterangaben für die Positionen der Soundquellen zu berechnen, die Methode befindet sich im gleichen Script.
Anschließend wird *setSoundPositions()* und *playSounds()* aus *audio3d.js* aufgerufen, um zuerst die berechneten Meterangaben für die Resonance Audio Sources zu setzten und anschließend abzuspielen.

Die Funktion *calculatePosition()* benötigt für die Berechnung der MIDI Daten in Meterangaben die Zeilen- und Spaltenanzahl, diese werden vor Anwendungsaufruf unter den Variablen *allRows* und *allColums* gespeichert. So kann aus dem Index bestimmt werden, in welcher Zeile und Spalte sich die Soundquelle befindet. Damit die Positionen der Sounds besser differenziert werden können, werden die Meterangaben noch einmal gestreckt, sowohl in der Breite als auch in der Tiefe. Die Ergebnisse werden dann in der passenden *audioGridBox* gespeichert.

Fazit

Wir konnten feststellen, dass eine bessere Kameraqualität die Kalibrierung deutlich vereinfacht hätte und somit das Tiefenbild noch detaillierter würde. Darüber hinaus lässt sich auch durch mehr Rechenleistung und eine optimierte Programmierung das Ergebnis positiv beeinflussen. Der ursprünglich geplante Ansatz, die Anwendung auf dem Raspberry Pi zu programmieren ist dementsprechend gescheitert und wir mussten zur Präsentation unsere Konzeptidee, dass man die Kameras am Kopf befestigt und sich die Augen verbindet, um dann den Raum nur über den Ton wahrzunehmen, verwerfen.

Wir sind aber dennoch zufrieden, dass die Erstellung des Tiefenbilds und dessen Analyse gut geklappt haben. Im Bereich der Tiefenbildverarbeitung wäre der nächste logische Schritt, eine Objekterkennung zu programmieren, die gleichartige Pixelflächen zusammenfasst und einen genauen Mittelpunkt bestimmt. Dadurch werden große Objekte nicht mehr als mehrere Objekte erkannt und damit erst ein Verfeinern der Rasterung sinnvoll.

Eike Lewandowski	2203694
Nina Tietje	2238710
Christopher von Bargen	2239427
Luca Hillen	2221029