

Universität Augsburg
Fakultät für Angewandte Informatik
Institut für Software & Systems Engineering

Masterarbeit
Im Studiengang Informatik

Titel

Patrick Grüner

patrick.gmm91@gmail.com

Matrikelnummer: xxxxxxxx

Schneiderhansenweg 10

87700 Memmingen

13. Mai 2018



Betreuer: N.N.
Erstgutachter: Dr. Alwin Hoffmann

Inhaltsverzeichnis

1. Einleitung	2
1.1. Motivation	3
1.2. Ziel der Arbeit	4
1.3. Umfeld der Arbeit	5
1.3.1. CMORE Automotive GmbH	5
1.3.2. C.LABEL	5
2. Grundlagen	7
2.1. Annotation	7
2.1.1. Maschinelles Lernen	7
2.2. Virtual Reality	14
2.3. Aufnahme von Punktwolken mit LiDAR	17
3. Systembeschreibung	21
3.1. Komponente 1: Die VR-Brille	21
3.1.1. Warum VR die passende Technologie für C.LABEL-VR ist	22
3.1.2. Warum die Oculus Rift verwendet wurde	25
3.2. Komponente 2: Der Rechner	27
3.3. Komponente 3: Die Entwicklungsplattform	29
4. C.LABEL-VR	35
4.1. Import und Export von Daten	36
4.1.1. Architektur	37
4.1.2. Interne Datenstruktur	39
4.1.3. HDF5 Beispiel	40
4.2. Generierung einer Punktwolke	40
4.2.1. Optimierung	43
4.3. Navigation	45
4.3.1. VR-Krankheit	45
4.3.2. Freier Flug-Modus	46
4.3.3. Teleport-Modus	49

4.4.	Annotieren der Punktwolke	50
4.4.1.	Pointer Annotation	50
4.4.2.	Touch Annotation	50
4.4.3.	Cluster Annotation	51
4.5.	User Interface	57
4.5.1.	Applikations-Menü	57
5.	Schluss	59
5.1.	Zusammenfassung	59
5.2.	Erreichte Ziele	59
5.3.	Vorteil der VR-Applikation	59
5.4.	Ausblick	59
A.	Appendix Title	60

Abstract

Abstract goes here

Declaration

I declare that..

Acknowledgements

I want to thank...

Abbildungsverzeichnis

1.1.	Vogelperspektive auf eine LiDAR-Punktwolke, wobei die roten Punkte alle Boden- und die blauen alle Nicht-Bodenpunkte darstellen	3
1.2.	Labeling Methoden in C.LABEL	6
2.1.	Diagramm um den besten Machine Learning Algorithmus für eine Predictive Analytics-Methoden zu finden [1]	8
2.2.	Abbildung des Übertragungsprinzips von Reizen zwischen Neuronen [?]	10
2.3.	Einfaches Perzeptron nach dem Prinzip von Frank Rosenblatt	10
2.4.	Stark vereinfachte Darstellung eines KNNs zur Erkennung von Autos und Personen	12
2.5.	Vergleich zwischen den Anfängen der VR-Technologie und dem heutigen Stand. Die oberen drei Bilder zeigen das VIVED-System, das von S.S. Fisher vorgestellt wurde [2]. Die Unterer zeigen Bilder der Oculus Rift.	16
2.6.	Ablauf einer LiDAR-Messung von einem Sensor, der die Messimpulse im Uhrzeigersinn abgibt. Die Bilder stammen von [?] und 2.6b wurde um den Winkel φ erweitert.	18
2.7.	Seitliche Sicht auf eine LiDAR-Messung mit mehreren, vertikal angeordneten Lasern. Die Abbildung stammt aus der Arbeit [3] und wurde durch die Einzeichnung des Winkels θ erweitert.	19
3.1.	Die zwei potentiellen AR-Brillen	22
3.2.	Die zwei potentiellen VR-Brillen	26
3.3.	Unity Editor, der C.LABEL-VR im Play Mode ausführt.	31
4.1.	in C.LABEL-VR vom Import bis zum Export	36
4.2.	Das Import-Export-Prinzip in C.LABEL-VR	38
4.3.	Beide Bilder zeigen Komponenten aus der Unity Editor, die dem Punkt-Prefab zugeordnet sind.	41
4.4.	Oculus Touch Controller mit der offiziellen Beschriftung von Oculus [4] .	47
4.5.	Objektidentifizierung einer 3D-Punktwolke aus [5]. Die Bodenpunkte der Wolke sind blau dargestellt und die Objekte sind mit Boxen gekennzeichnet. .	51

4.6. In [5] wird gezeigt, dass die Segmentierung einer Punktwolke zu einem besseren Ergebnis der Bodenpunktanalyse führt. Die obere Abbildung ist dabei ohne Segmentierung und die untere mit Segmentierung. Bodenpunkte sind blau und Objektpunkte in grün dargestellt. Die roten Rechtecke kennzeichnen den Bereich in dem die Analyse schlechter bzw. besser ist. . .	53
4.7. Vergleich der Segmentierungsmethoden zur Bodenpunktanalyse aus [5] und C.LABEL-VR. Bodenpunkte sind rot, alle anderen schwarz. Die blauen Linien stellen die Trennung der Segmente dar. Die grünen Boxen markieren den Bereich, in dem man den Unterschied der beiden Methoden bei der Bodenpunkterkennung sieht.	54
4.8. Bei zu wenigen Bodenpunkten in einem Segment passt sich die Ebene den falschen Punkten an.	54

Einleitung

Der Einsatz künstlicher Intelligenzen ist aktuell in allen Bereichen der Informatik auf dem Vormarsch. Dies gilt vor allem für die Automobilindustrie, da das Thema des autonomen Fahrens nicht ohne intelligente Algorithmen realisierbar ist. Eine wichtige Rolle bei der Entwicklung solcher Algorithmen spielt dabei das maschinelle Lernen. Dabei wird versucht, ausgehend von vielen lehrreichen Beispieldaten, die Lösung einer Aufgabe zu lernen und auf andere unbekannte Daten zu verallgemeinern. So kann ein System auch auf vorher ungewohnte Daten reagieren, was mit einer statischen Programmierung nur schwer oder gar nicht möglich ist. Der Erfolg dieses Prinzips hängt dabei stark von der Qualität der Beispieldaten ab von denen die Algorithmen lernen. Deshalb wird in die Erstellung dieser Daten sehr viel Arbeit gesteckt.

Im Bereich der Fahrerassistenzsysteme und des autonomen Fahrens ist es wichtig, dass das Auto seine Umgebung so gut wie möglich wahrnehmen kann. Algorithmen die für die Erkennung und Identifizierung des Umfeldes verantwortlich sind werden mit Daten von verschiedenen Sensoren gefüttert. Hierbei handelt es sich meist um Kamera-, Radar- oder LiDAR-Sensoren. Die Eingangsdaten dieser Sensoren müssen nun so aufbereitet werden, damit ein lernendes Computersystem etwas damit anfangen kann. Dazu werden alle, für das Anwendungsfeld wichtigen Teile mit entsprechenden Klassifikationen versehen. Diese Klassifikationen werden auch als *Labels* oder *Annotationen* bezeichnet. Zum Beispiel werden auf einem Kamerabild alle Personen und Fahrzeuge als eben diese markiert. Das Fahrzeug kann auf diese Weise lernen wie Personen und Fahrzeuge aussehen, um sie dann später im Straßenverkehr selbstständig wiederzuerkennen. Radar- und LiDAR-Sensoren liefern stattdessen Punktwolken als Eingangsdaten (vgl. Abbildung 1.1), das Prinzip bleibt allerdings das gleiche. Auch hier müssen alle wichtigen Teile, in diesem Fall Punkte, mit entsprechenden Klassen versehen werden. Dieser Vorgang nennt sich *Labeln* bzw. *Annotieren*.

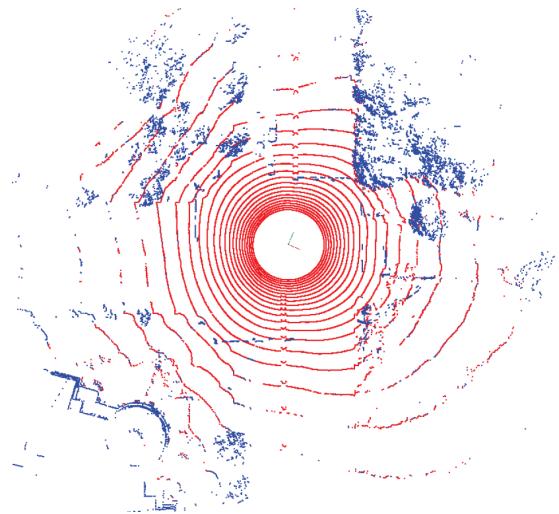


Abbildung 1.1.: Vogelperspektive auf eine LiDAR-Punktwolke, wobei die roten Punkte alle Boden- und die blauen alle Nicht-Bodenpunkte darstellen

1.1. Motivation

Der Fokus dieser Arbeit liegt auf der Annotation von Punktwolken. In einem bislang aufwändigen Verfahren (siehe Kapitel 1.3.2) müssen diese Punktwolken so aufbereitet und mit einer Bedeutung versehen werden, dass die Algorithmen an diesen Beispielen etwas lernen können. Sie müssen nach dem Lernen in der Lage sein Unterscheidungen und Erkennungen selbst durchzuführen und dabei möglichst wenige Fehler machen. Bei solchen Verfahren kommen Softwareanwendungen wie *C.LABEL* zum Einsatz. Dies ist eine Anwendung zum Annotieren verschiedener Sensordaten für den Automobilbereich und wurde von der Firma *CMORE Automotive GmbH* entwickelt. CMORE ist das betreuende Unternehmen dieser Masterarbeit und wird in Kapitel 1.3.1 vorgestellt. Auf das Programm *C.LABEL* wird in Kapitel 1.3.2 etwas genauer eingegangen. *C.LABEL* ist eine Anwendung, die zunächst die Sensordaten für den menschlichen Benutzer visualisiert. Dieser muss sie dann manuell mit unterschiedlichen Bedeutungen versehen. Wegen der großen Menge an Daten, die für das Trainieren der Algorithmen benötigt wird, muss dieses manuelle Annotieren möglichst effizient durchführbar sein.

Ein zweidimensionales Kamerabild kann sehr einfach auf einem Computermonitor dargestellt und bearbeitet werden. Eine besondere Herausforderung stellt jedoch die Verarbeitung von 3D-Daten, wie einer Punktwolke, dar. Hier stößt man mit den Möglichkeiten eines zweidimensionalen Computermonitors schnell an die Grenzen einer effizienten Darstellung und Bearbeitung. So ist es beispielsweise für die Annotierung solcher Punktwolken und ihrer Einzelmesswerte schwierig, die optimale Perspektive auf die Daten zu finden, die eine effiziente Erfassung durch den Menschen und eine entsprechende Bearbeitung zulässt. Dies erfordert von den Benutzern sehr viel Übung und Erfahrung, durch entsprechende

Drehungen der Punktwolkendarstellung sich in dieser zurechtzufinden und Messpunkte realen Objekten zuzuordnen.

Eine Alternative dazu soll die Applikation bieten, die innerhalb dieser Masterarbeit entwickelt wurde. Diese soll das Visualisieren und Annotieren von Punktwolken einfacher und effizienter gestaltet. Mittels *Virtual Reality* (VR) soll die Grenze zwischen 2-D und 3-D aufgehoben werden, sodass man sich als interaktiver Teilnehmer im dreidimensionalen Raum durch die Sensordaten navigieren, mit ihnen interagieren und sie annotieren kann. Die Applikation trägt den Namen *C.LABEL-VR*.

1.2. Ziel der Arbeit

Ziel dieser Arbeit ist es zunächst ein passendes Medium für eine solche Applikation zu finden, das nicht nur die Anforderungen der Applikation selbst erfüllt, sondern auch an einem handelsüblichen Büro-Arbeitsplatz verwendet werden kann. Anschließend soll eine Applikation erstellt werden die folgende Grundfunktionalität erfüllt:

1. Mit der Anwendung soll es möglich sein ein, bei CMORE gängiges, Datenformat einzulesen und alle nötigen Informationen daraus zu extrahieren
2. Aus den extrahierten Informationen soll eine Punktwolke generiert und innerhalb des gewählten Mediums visualisiert werden.
3. Der Benutzer muss die Möglichkeit haben durch die Punktwolke zu navigieren.
4. Die Applikation muss die Möglichkeit bieten jeden einzelnen Punkt mit einer Klassifikation versehen zu können. Dabei sollen Alleinstellungsmerkmale des gewählten Mediums benutzt werden, um eine vorzeigbare Verbesserung gegenüber der herkömmlichen Computer-Monitor-Annotation zu generieren.
5. Alle getätigten Annotationen müssen in die eingelesenen Dateien zurückgeschrieben bzw. in neue Dateien exportiert werden.

Anschließend sollen die Basisfunktionen erweitert und zusätzliche Funktionen hinzugefügt werden. Abhängig von der verbleibenden Zeit soll die Anwendung gemäß folgender Priorität erweitert werden:

1. Es sollen neue Möglichkeiten zur Annotation von Punkten implementiert werden.

2. Die Möglichkeit, eigene Klassifikationen für Punkte zu erstellen, soll hinzugefügt werden.
3. Neue Datenformate importieren.
4. Neue Navigationsmöglichkeiten implementieren.

Am wichtigsten wäre hierbei also die Applikation um effiziente Möglichkeiten zum Labeln von Punkten zu erweitern. Außerdem sollte es für den Benutzer möglich sein individuelle Klassifikationen anzulegen, um unterschiedliche Anwendungsgebiete abzudecken. Optional sind neue Navigationsmöglichkeiten und einlesbare Datenformate.

1.3. Umfeld der Arbeit

C.LABEL-VR und diese Masterarbeit wurden in Zusammenarbeit mit dem Unternehmen CMORE Automotive GmbH entwickelt. Als Grundlage für die Annotation von Punktwolken soll das CMORE-Programm C.LABEL dienen. Beides soll im Folgenden kurz vorgestellt werden.

1.3.1. CMORE Automotive GmbH

Im Mai 2011 wurde die CMORE Automotive GmbH von Richard Woller und Gregor Matenaer die CMORE Automotive GmbH in Lindau gegründet. Der Tätigkeitsbereich von CMORE liegt in der Unterstützung Entwicklung und Validierung zukünftiger Mobilitätskonzepte. Dabei bietet CMORE Lösungen und Produkte für alle Entwicklungsphasen. Dazu gehören beispielsweise komplexe Steuergerätesoftware für die Serienproduktion, Deep Learning Verfahren und Softwaretools zur hochautomatisierten Datenanalyse oder auch komplett Fahrzeugmesstechniksysteme zur Referenzdatengenerierung. Seit 2017 gibt es innerhalb des Unternehmens eine eigene Abteilung, die verantwortlich für die Distribution, Analyse und Anreicherung von Daten im Automobilbereich ist. Sie trägt den Namen C.IDS (*Integrated Data Solutions*). Ein Beispiel für solche Automobildaten sind Punktwolken von LiDAR-Sensoren, mit welchen sich auch diese Masterarbeit beschäftigt.

1.3.2. C.LABEL

C.LABEL ist ein Programm zu Annotierung von Daten im zwei- und dreidimensionalen Bereich. Im zweidimensionalen Bereich handelt es sich bei den Daten um Kamerabilder. In diesen Bildern werden wichtige Objekte mit sogenannten *Bounding Boxen* markiert, oder wichtige Bereiche werden zusammenhängend mit einer Klassifikation versehen (*Semantisches Labeln*). Im dreidimensionalen Fall werden Punktwolken eines Radar- oder LiDAR-Sensors annotiert. Hierbei müssen alle wichtigen Punkte mit einer Klassifikation

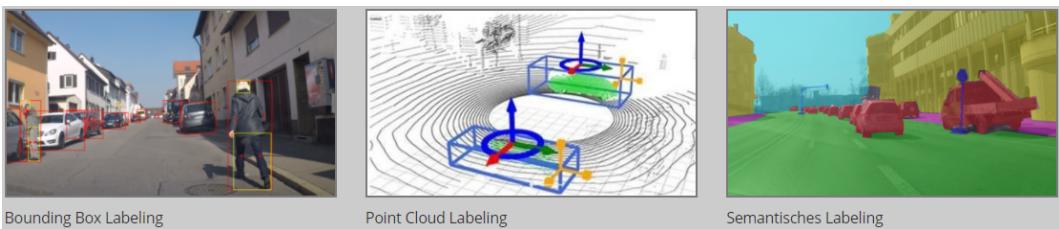


Abbildung 1.2.: Labeling Methoden in C.LABEL

versehen werden. Die Labeling Methoden sind in Abbildung 1.2 dargestellt. Das Ziel dieser Software ist es den Zeitaufwand, den die Annotation dieser Daten benötigt, zu minimieren. Dazu sind intelligente und effiziente Funktionen notwendig. Eine davon ist zum Beispiel die automatische Vorhersage von Annotationen durch stets analysierende Algorithmen. Dieses Prinzip wird *Active Learning Loop* genannt. Bei der Entwicklung von C.LABEL-VR liegt der Fokus auf der Annotierung von 3D-Punktwolken und dabei wurden Funktionen und Ansätze von C.LABEL als Basis für die Entwicklung genutzt.

Grundlagen

2.1. Annotation

TODO -Anntation im allgemeinen -Für welche zwecke wird es verwendet -Hinleitung zum maschinellen Lernen, für das Annotierte Daten notwendig sind

2.1.1. Maschinelles Lernen

Der Ausdruck Maschinelles Lernen (engl.: *Machine Learning*) ist heutzutage ein Überbegriff für Methoden, die versuchen durch Optimierung eine Funktion zu erlernen. Bei solch einer Funktion kann es sich sowohl um eine simple binäre Entscheidung auf eine Fragestellung handeln als auch um komplexere Dinge wie das Übersetzen von Texten in eine andere Sprache, das Erkennen von Personen auf Bildern, oder eben das Klassifizieren von Objekten in einer Punktwolke. Letzteres ist speziell für diese Arbeit relevant. Die Komplexität der Funktion bestimmt auch die zu verwendende Methode zur Lösung des Problems. Im Zuge des *Azure Machine Learning Studios* veröffentlichte Microsoft eine Graphik (vgl. Abbildung 2.2), mit der man den richtigen Lernalgorithmus für gewisse Vorhersage-Methoden bestimmen kann. Für die Objektklassifizierung im Automobilbereich ist es wichtig die Klasse eines Objekts mit hoher Genauigkeit aus einer großen Anzahl verschiedener Klassen zu bestimmen. Basierend auf der Graphik 2.2 ist die beste Methode dafür die Verwendung eines künstlichen neuronalen Netzes(KNN). Die Benutzung solcher Netze ist üblich, wenn es um Klassifikation von vielen verschiedenen Dingen geht und soll im Folgenden näher erläutert werden.

TODO Grafik von Bild- und Wolkenerkennung

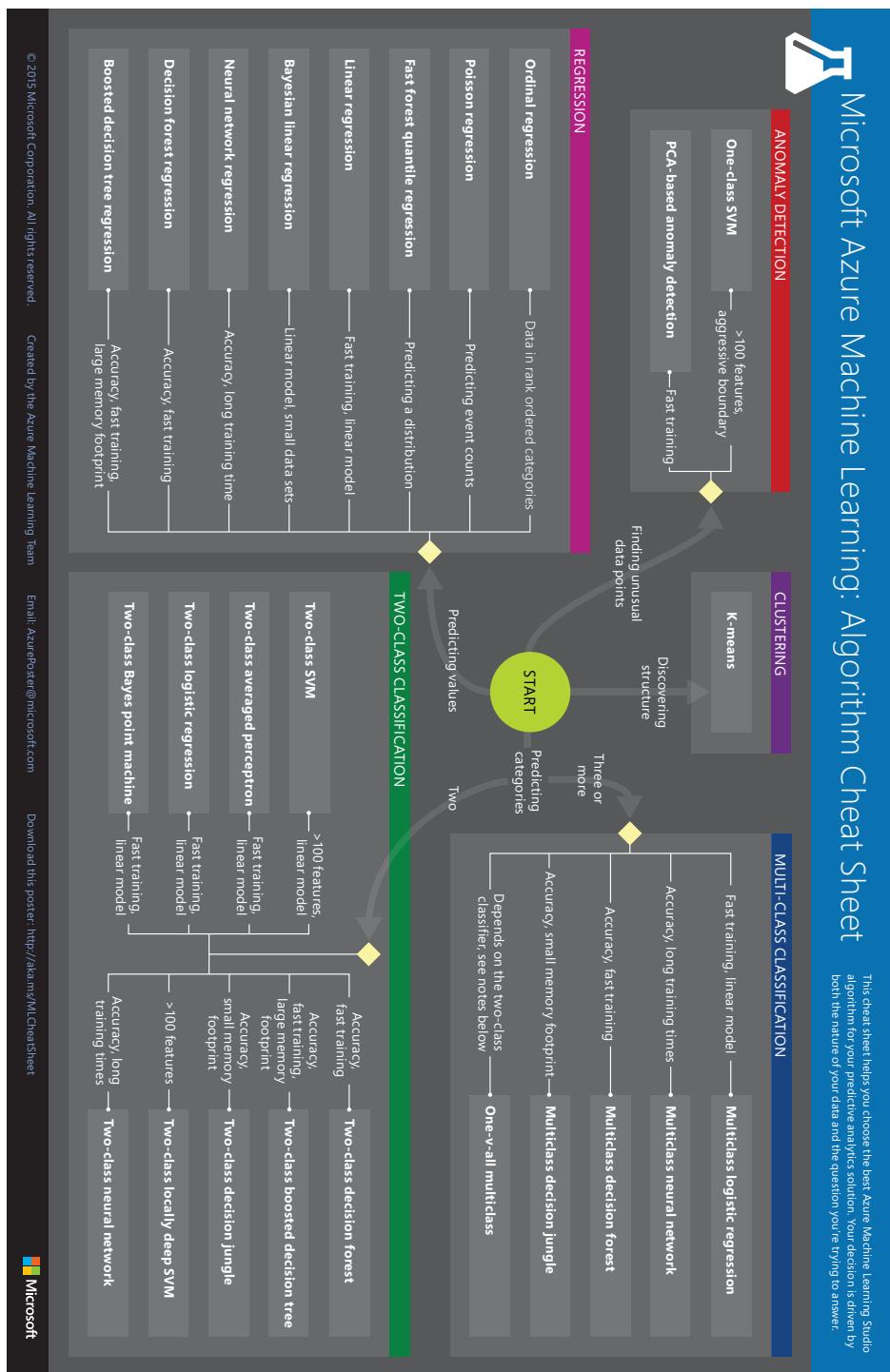


Abbildung 2.1.: Diagramm um den besten Machine Learning Algorithmus für eine Predictive Analytics-Methoden zu finden [1]

Künstliche Neuronale Netze

David Kriesel hat in [6] den Zusammenhang zwischen physischen und künstlichen neuronalen Netzen dargestellt. Seine Arbeit soll im Folgenden als Quelle dienen. Das Prinzip der künstlichen neuronalen Netze ist, wie vieles in der Informatik, einer Struktur aus der realen Welt nachempfunden. Das komplexeste neuronale Netz ist wohl das menschliche Gehirn mit etwa 86 Milliarden Nervenzellen, welche auch Neuronen genannt werden. Daher kommt auch der Begriff des neuronalen Netzes. Diese Nervenzellen sind durch Synapsen mit bis zu mehreren Tausend anderen Zellen verbunden. Diese Verbindungen bilden zusammen das Netz. Die Kommunikation zwischen den Neuronen erfolgt in der Regel über elektrische Impulse und chemische Botenstoffe. Beides wird über Synapsen zu den nächsten, verbundenen Neuronen weitergeleitet und bewirkt dort einen Reiz. Durch Übertragung von chemischen Botenstoffen kann dieser Reiz zusätzlich gewichtet werden, das heißt es kann ein stärkerer oder schwächerer Reiz ausgesendet werden. Alle ankommenden Reize werden dann am Empfangsneuron akkumuliert und ergeben den Eingangsreiz des Neurons.

Dieses, vereinfacht dargestellte Prinzip, wird auch bei den KNNs verwendet. Übersteigt dieser Reiz eine bestimmte Schwelle kommt es zum entscheidenden Ereignis: Das Neuron *feuert*. Dadurch gibt das feuernende Neurone einen Impuls an alle anderen Neuronen weiter mit dem es verbunden ist. Bei dafür vorgesehenen Neuronen oder verbänden davon, kann dieser Impuls auch ein Ereignis auslösen. Diese Ereignisse können physischer Natur sein, also beispielsweise Bewegungen, oder auch psychischer Natur, zum Beispiel das Erkennen von Dingen. Werden diese Nervenzellen also mit einem ausreichenden Impuls angesprochen, wird das entsprechende Ereignis ausgelöst.

Die entscheidende Komponente dieses Prinzips ist die adaptive Gewichtung der weitergegebenen Reize. Adaptiv deswegen, da die Gewichtung der Reize sich während eines Menschenlebens verändern können. So erkennen Kinder oft Gefahrensituation nicht, da ihr Gehirn nicht ausreichend geschult ist um diese zu erkennen. In diesem Kontext bedeutet das, dass an die *Gefahren-Neuronen* kein ausreichend starker Reiz gesendet wird. Durch sammeln von Erfahrungen optimiert sich dieser Wert, was in späteren Jahren zu einer anderen Reaktion des Gehirns führen kann als früher. Das Adaptieren dieser Gewichte auf einen passenden Wert bezeichnen wir als Lernen. Um Computersystemen einen derartigen Lerneffekt zu ermöglichen wurde dieses Biologische Prinzip nun mathematisch imitiert.

Künstliche Neuronale Netze basieren auf dem Prinzip des Perzeptrons, das von Frank Rosenblatt 1958 vorgestellt wurde [7]. Dieses Perzeptron repräsentiert die Funktion eines einzelnen Neurons auf eine mathematische Weise (vgl. 2.3). Die Eingangsdaten eines Perzeptrons bestehen aus einer n -großen Menge an Werten $x_1 \dots x_n$, welche eine gleich-große Menge an jeweiligen Gewichten $W_1 \dots W_n$ haben. Wenn das zu analysierende Ziel

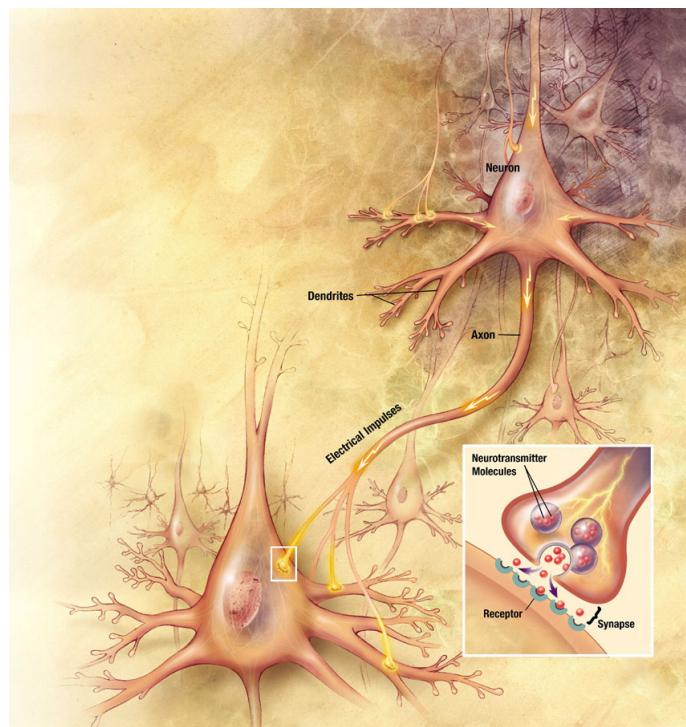


Abbildung 2.2.: Abbildung des Übertragungsprinzips von Reizen zwischen Neuronen [?]

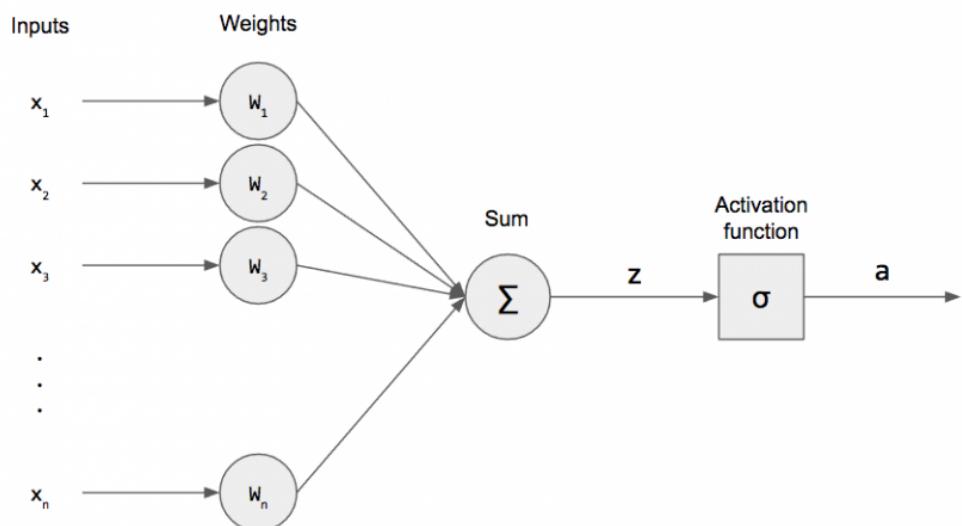


Abbildung 2.3.: Einfaches Perzeptron nach dem Prinzip von Frank Rosenblatt

beispielsweise ein Bild ist, wäre n die Menge aller Pixel des Bildes und bei Punktwolken eben die Anzahl der Punkte. Die Eingangswerte werden von einer mathematischen Funktion empfangen und zu einem skalaren Wert z zusammengefasst. Hierbei wird meist die Gewichtete Summe benutzt die folgendermaßen berechnet wird:

$$z = \sum_{i=0}^n W_i \cdot x_i \quad (2.1)$$

Im biologischen Vorbild repräsentiert diese Summe den eingehenden Reiz in eine Nervenzelle, die x -Werte sind dabei die Impulse der vorherigen Neuronen und die W -Werte repräsentieren die Gewichtung, also die Stärke der Eingangsreize. Der biologische Schwellwert wird im Perzeptron durch eine Mathematische Funktion $\sigma(z)$ dargestellt. Wie in [?] beschrieben gibt es dafür zwei beliebte Funktionen. Die *Fermifunktion* (2.2), auch Logische Funktion genannt, welche einen Ausgabe-Wertebereich von (0,1) hat und den *Tangens Hyperbolicus* ($\tanh(z)$) mit einem Wertebereich von (-1,1). Beide sind differenzierbar, was wichtig für das Lernen mit *Backpropagation* ist. Was das bedeutet wird in Abschnitt 2.1.1 näher erläutert. Wenn der Wert, den diese Funktion liefert, nicht noch durch eine weitere Ausgangsfunktion verändert wird, ist er der Ausgangswert des Perzeptrons.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (2.2)$$

In KNNs können mehrere dieser Perzeptrons eine sogenannte Schicht bilden. Je mehr Schichten ein neuronales Netz hat desto *tiefer* ist es, daher auch der Begriff *Deep Learning*, also tiefes Lernen. In der Regel bestehen KNNs aus mindestens 3 Schichten, nämlich einer Eingangsschicht, einer oder mehrerer Zwischenschichten und einer Ausgangsschicht (veranschaulicht in Abbildung 2.4). Die einzelnen Schichten werden im Folgenden kurz erläutert.

- **Eingangsschicht**

Die Eingabeschicht ist der Startpunkt des Informationsflusses in einem künstlichen neuronalen Netzes. Eingangssignale werden von den Neuronen dieser Schicht aufgenommen und gewichtet an alle Neuronen der ersten Zwischenschicht weitergegeben. Die Anzahl der Neuronen dieser Schicht hängt, wie schon erwähnt, von den Eingangsdaten ab. Werden Bilder mit einer Auflösung von 50x50 Pixeln in das Netz gespeist, hat dieses Netz in der Regel 50x50, also 2500 Neuronen in der Eingangsschicht.

- **Zwischenschicht**

Zwischen der Eingabe- und der Ausgabeschicht befindet sich in jedem künstlichen neuronalen Netz mindestens eine Zwischenschicht, die auch verborgene Schicht (engl.: *hidden layer*) genannt wird. Theoretisch ist die Anzahl der möglichen verborgenen Schichten in einem künstlichen neuronalen Netzwerk unbegrenzt. In der Praxis

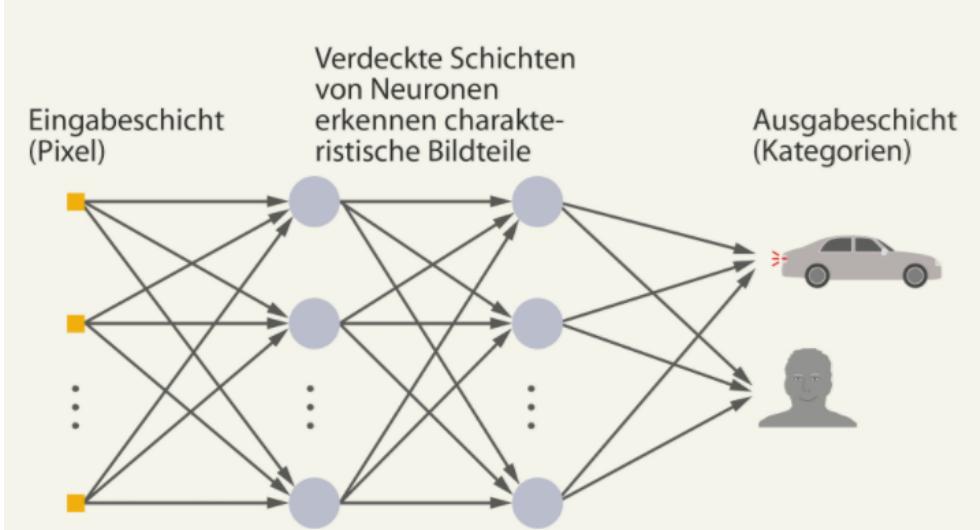


Abbildung 2.4.: Stark vereinfachte Darstellung eines KNNs zur Erkennung von Autos und Personen

bewirkt jede hinzukommende Schicht jedoch auch einen Anstieg der benötigten Rechenleistung, die für den Betrieb des Netzes notwendig ist.

- **Ausgangsschicht**

Die Ausgabeschicht liegt hinter den Zwischenschichten und bildet die letzte Schicht in einem künstlichen neuronalen Netz. In der Ausgabeschicht angeordnete Neuronen sind jeweils mit allen Neuronen der letzten Zwischenschicht verbunden. Die Ausgabeschicht stellt den Endpunkt des Informationsflusses in einem KNN dar und enthält das Ergebnis der Informationsverarbeitung durch das Netzwerk. Beim Beispiel der Objektklassifikation besteht die Ausgabeschicht nun aus n vielen Endknoten, wobei n die Menge an möglichen Klassifikationen ist. Die Eingangssumme dieser Knoten, den sie aus den vorherigen Neuronen bekommen, entsprechen der Wahrscheinlichkeit, dass es sich bei den Eingangsdaten um die jeweilige Klasse handelt. Abbildung 2.4 soll dieses Prinzip veranschaulichen.

Trainieren von künstlichen neuronalen Netzen

Damit ein Mensch eine neue Tätigkeit beherrscht muss er diese erst erlernen. In der Regel muss diese Tätigkeit dafür so oft es geht geübt werden. Bei künstlichen neuronalen Netzen ist das genauso. Das Üben bezeichnet man in diesem Kontext als *Trainieren* von Netzen. Im allgemeinen bedeutet der Vorgang des Lernens, dass ein System sich in irgendeiner Form verändert, um sich z.B. an Veränderungen in seiner Umwelt anzupassen. Die typische Veränderung von künstlichen neuronalen Netzen ist das adaptieren der Verbindungsge wichtete $W_1 \dots W_n$. Damit die Verbindungsgewichte auf einen passenden Wert verändert werden muss das Netz entsprechend oft trainiert werden. Dies geschieht nach Regeln, die

man in Algorithmen definiert – ein Lernverfahren ist also immer ein Algorithmus, den man einfach mithilfe einer Programmiersprache implementieren kann. Diese Algorithmen lassen sich grob in 3 verschiedenen Arten einteilen:

- Unüberwachtes Lernen (*Unsupervised Learning*)
- Bestärkendes Lernen (*Reinforcement Learning*)
- Überwachtes Lernen (*Supervised Learning*)

Für Objektklassifizierung wird in der Regel das überwachte Lernen verwendet, darum wird im Folgenden nur auf das dieses Verfahren eingegangen.

Der Lernprozess bei dieser Methode ist, wie der Name schon sagt, überwacht. Das kommt daher, dass der Mensch eine Menge an selbst ausgewählten Trainingsbeispielen erstellt. Das wird dann mit dieser überwachten Menge trainiert. Die Trainingsbeispiele bestehen aus einer Menge P an Eingabemustern mit jeweiliger korrekter Lösung. Das Netz kann seine eigene Ausgabe mit der richtigen Lösung des Beispiels vergleichen und somit eine Fehlerbeschreibung zurückgeben. Für das Beispiel der Punktwolkenklassifizierung bestünde dabei das Trainingsset aus vielen verschiedenen Punktwolken, die alle ein Objekt darstellen, wobei die Art des Objektes, also die Lösung der Klassifizierung, bekannt ist.

Die Werte der Gewichte $W_1 \dots W_n$ werden bei einem untrainierten Netz zu Beginn zufällig gewählt. Dadurch gibt ein solches Netz in der Regel völlig falsche Lösungen für die ersten Trainingsbeispiele aus. Das ist aber nicht weiter schlimm, da nun der Lerneffekt einsetzen soll. Durch die Bekanntheit der Lösung des Beispiels kann der Fehler zwischen dem Ergebnis des Netzes und der gewünschten Lösung genau bestimmt werden. Dieser Fehler kann anschließend durch zurückrechnen auf jedes Neuron der vorherigen Schichten projiziert werden. Auf diese Weise kann der Fehler, den jedes einzelne Neuron verursacht bestimmt werden. Dieses Zurückrechnen nennt sich *Backpropagation*. Aus dem Fehler den jedes Neuron verursacht kann schließlich der Betrag errechnet werden, um den sich das Gewicht W_i des Neurons verändern muss, damit bei zukünftigen Eingaben bessere Ergebnisse erzielt werden. Für das Verständnis des Grundprinzips ist die Formel zu Berechnung der Gewichtsänderung nicht wichtig, sie wird aber zur Vervollständigung im Folgenden angegeben:

$$\Delta w_{ij} = -\eta \delta_j o_i \quad (2.3)$$

- Δw_{ij} ist die Änderung des Gewichts w_{ij} der Verbindung von Neuron i zu Neuron j
- η ist ein fester Wert, der eine Lernrate ausdrücken soll, mit der die Stärke der Gewichtsveränderung festgelegt wird

- δ_j ist das Fehlersignal des Neurons j das sich aus der partiellen Ableitung $\frac{\partial E}{\partial \text{net}_j}$ ergibt
 - E ist dabei der quadratische Fehler der bei der Berechnung des Ergebnisses entstand, er wird durch die Formel $E = \sum_{i=1}^n (t_i - o_i)^2$ berechnet
 - * t_i steht darin für den gewünschten Zielwert
 - net_j ist die Netzeingabe $\sum_{i=1}^n x_i w_{ij}$
- o_i steht sowohl bei der Berechnung von Δw_{ij} , als auch von E für die errechnete Ausgabe des Neurons i

Zusammenfassend kann man nun sagen, dass der Lernprozess eines künstlichen Neuronalen Netzes nach folgendem Schema abläuft:

1. **Eingabe** eines Elements p aus der Trainingsmenge P in die Eingabeschicht des Netzes
2. **Vorwärtspropagierung** der Eingabe durch das Netz gesamte Netz, sodass man bei der Ausgabeschicht ein Ergebnis erhält
3. **Vergleich** des Ergebnisses mit der richtigen Lösung und Berechnung des Fehlers
4. **Zurückpropagieren** des Fehlers, um somit die Gewichte der Neuronen anzupassen

Wie unschwer zu erkennen ist basiert der Backpropagation-Algorithmus auf dem Vorhandensein eines Trainingsdatensatzes. Damit das Netz nach dem Training für neue, unbekannte Eingaben die richtige Lösung errechnet ist es elementar wichtig, dass dieser Datensatz eine hohe Qualität und Quantität hat. Qualität bedeutet in diesem Fall, dass die Trainingsdaten repräsentativ für den jeweiligen Anwendungsfall sein müssen. Ein Personenerkennungsalgorithmus der im Straßenverkehr Personen identifizieren soll liefert keine guten Ergebnisse, wenn er ausschließlich mit Bildern von Personen in Innenräumen gefüttert wird. Mit Quantität ist Anzahl der verschiedenen Elemente im Trainingssatz gemeint. KNNs verbessern ihre Ausgabe in der Regel mit größerer Menge an unterschiedlichen Trainingsdaten.

Zur Erstellung dieses Datensatzes ist es nun nötig so viele repräsentative Daten wie möglich zu sammeln und diese mit der richtigen Lösung zu annotieren. Eine Methode für eine solche Annotierung wird in dieser Arbeit vorgestellt. Dieses Kapitel sollte nun anschaulich haben warum solche Methoden so essentiell sind, um künstliche Intelligenzen zu entwickeln.

2.2. Virtual Reality

Das Konzept digitale Inhalte räumlich und interaktiv darzustellen ist nicht neu. Die Ursprünge von VR lassen bis 1965 zurückverfolgen als Ivan Sutherland „*The Ultimate Display*“ [8]

vorstellte. Diese Arbeit stellte die ersten Konzepte für Immersion in einer virtuellen Welt und die dafür nötigen Ein- und Ausgabegeräte dar. Er wollte damit einen Anreiz schaffen um die Grenzen der damaligen Mensch-Computer-Schnittstelle zu überwinden und neue Applikationen zu entwickeln, welche die eigene Präsenz in der virtuellen Welt miteinbeziehen [9].

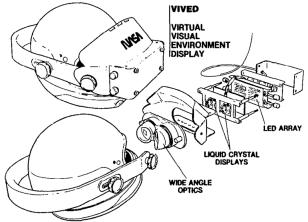
Anfang der achtziger Jahre gab es dann schon funktionierende VR-Systeme, beispielsweise das VIVED System (*Virtual Visual Environment Display*), welches von der NASA entwickelt wurde [2]. Dieses System beinhaltet eine, am Kopf befestigte Brille, mit einem integrierten 2,7" großen LCD-Display (Abbildung 2.5a). Durch Weitwinkelobjektive wird das Bild dieses Displays auf jedes Auge einzeln projiziert. Damit war es möglich Objekte, die normalerweise an einem offenen Bildschirm angezeigt werden, in einer geschlossenen und 3D-ähnlichen Sicht darzustellen (Abbildung 2.5b).

Die Projizierung auf jedes Auge vermittelt dabei einen räumlichen Eindruck, der durch das Prinzip der Stereoskopie verursacht wird [?]. Dieses Prinzip beruht darauf, dass durch zwei Augen die Umgebung aus zwei verschiedenen Blickwinkeln wahrgenommen wird. Dies kann leicht selbst festgestellt werden, indem man einen Finger zwischen die Augen hält und abwechselnd jedes Auge einzeln öffnet. Dabei ist zu erkennen, dass der Finger sich weiter rechts oder links im Blickfeld befindet, je nachdem welches Auge geöffnet ist. Durch die Zusammensetzung der zwei Blickwinkel durch das Gehirn entsteht ein für uns ein Tiefeneffekt. Dieser Effekt wird auch von VR-Brillen genutzt um 2D-Inhalte räumlich wirken zu lassen.

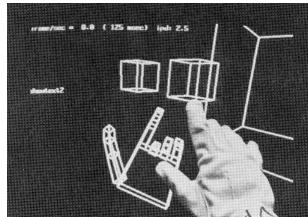
Als Eingabegerät wurde sowohl eine Spracherkennung, als auch ein Handschuh benutzt, der Bewegungssensoren an jedem Finger hatte. Somit war es dem Benutzer möglich auf eine intuitive Weise mit dem System zu Interagieren (Abbildung 2.5c), da das System die Bewegungen des Benutzers erkannte.

Am Prinzip der VR-Technologie hat sich bis heute nicht viel verändert. Die Präsentation der virtuellen Welt geschieht immer noch durch eine Brille mit integrierten Linsen und einem Display (Abbildung 2.5d). Diese beinhaltet, ebenso wie die Eingabegeräte, Bewegungssensoren, die die Position und Orientierung dieser Komponenten misst.

Die funktionelle Leistung der Komponenten ist im Laufe der Zeit jedoch deutlich besser geworden. Statt eckigen Polygonen können nun hochauflösende 3D-Modelle durch ein Display mit 2160 x 1200 Pixeln dargestellt werden (Abbildung 2.5e). Die Brille und die Eingabegeräte werden nun auch, zusätzlich zu den internen Bewegungssensoren, von externen Infrarot-Sensoren erfasst, um eine präzise Portierung der physischen Bewegungen in virtuelle zu übersetzen. Der größte Unterschied ist wohl, dass die VR-Technologie heutzutage hauptsächlich für Unterhaltungszwecke genutzt wird. Was früher mal ein Forschungsprojekt einer Raumfahrtorganisation war ist heute ein Entertainment-System, das



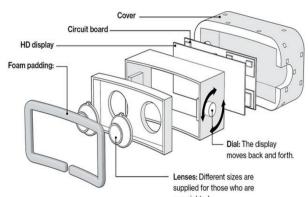
(a) VIVED Displayaufbau



(b) Sicht eines VIVED Benutzers



(c) Sicht auf einen VIVED Benutzer



(d) Rift Displayaufbau [10]



(e) Sicht eines Rift Benutzer [11]



(f) Sicht auf einen Rift Benutzer [12]

Abbildung 2.5.: Vergleich zwischen den Anfängen der VR-Technologie und dem heutigen Stand. Die oberen drei Bilder zeigen das VIVED-System, das von S.S. Fisher vorgestellt wurde [2]. Die Unteren zeigen Bilder der Oculus Rift.

in viele Privathaushalten zu finden ist.

Ein Grund dafür war die Entwicklung der zwei, derzeit geläufigsten VR-Brillen *Oculus Rift* und *HTC Vive*. Ein paar geschichtliche Eckdaten davon wurden in [13] gesammelt. 2010 wurde der erste Prototyp der Oculus Rift von Palmer Luckey entworfen. Dieser hatte eine horizontale Sichtweite von 90°, was zur damaligen Zeit im Verbrauchermarkt einzigartig war. 2013 hatte Valve einen Durchbruch bei der Entwicklung von *Low-Persistence*-Displays, den sie öffentlich geteilt haben. Dies war sehr hilfreich für alle Entwickler, denn Oculus beispielsweise benutzte seitdem bei all ihren nachfolgenden Brillen. *Low-Persistence* (zu deutsch: *geringe Ausdauer*) bedeutet in dem Falle die angezeigten Bilder nur wenige Millisekunden anzuzeigen um somit Objekte, die sich Bewegen schärfer darstellen zu können. 2014 wurde Oculus Rift für zwei Milliarden Dollar von Facebook gekauft und beschäftigen mittlerweile circa 400 Mitarbeiter allein für die Virtual Reality Entwicklung. Daran kann man das Potential sehen, das große Konzerne in der VR-Technologie sehen, denn kurz danach kündigte nämlich auch Sony eine VR-Brille für die Playstation 4 an.

Die, aus der Großproduktion der führenden VR-Konzerne resultierende, Massentauglichkeit und die Erschwinglichkeit der Technologie machen diese aber auch für die Industrie interessant. Vor allem im Zuge der *Industrie 4.0* ist VR für Digitalisierung des industriellen Umfelds von Nutzen. Es werden Leistungsfähige und qualitativ immer bessere Lösungen zur virtuellen Datenerfassung und -visualisierung entwickelt. Mittels VR lassen sich Arbeitssituationen simulieren und gefahrlos testen. Industriearbeiter können in Umgebungen

agieren, die in der Form noch gar nicht existieren. So können zum Beispiel Arbeitsumfelder und -prozesse effizient geplant werden. Schadens- und Störfälle lassen sich dank VR schnell und in Gänze erfassen und aus der Ferne warten. Es können auch Dinge räumlich begehbar gemacht werden, die es in der physischen Welt nicht gibt, wie beispielsweise Sensordaten. Im Rahmen dieser Arbeit werden Daten eines Lidarsensors als Punktwolke dargestellt, durch die man sich dann Bewegen kann.

2.3. Aufnahme von Punktwolken mit LiDAR

Punktwolken, wie sie in dieser Arbeit behandelt werden, bestehen aus vielen einzelnen Abstandsmessungen, wobei jeder Punkt einer dieser Messungen entspricht. LiDAR (*Light Detection and Ranging*) ist eine Methode zur Abstandsmessung mit Hilfe von Licht, welche in der Automobilbranche häufig verwendet wird. In den letzten Jahrzehnten wurde der LiDAR-Sensor aber auch in vielen anderen Bereichen zur herkömmlichen Methode, wenn es um die Messung von Abstandsinformationen geht. Im Bereich Archeologie benutzten Johnson und Oiumet diese Technologie um die Landschaft von New England in den USA zu analysieren [14]. Die Analyse von Landschaften ist durch LiDAR aber nicht nur auf dem Planeten Erde möglich. In [15] wurde LiDAR benutzt um Oberflächenmessungen auf dem Mars durchzuführen. In der Robotik wird es häufig verwendet um Hindernisse für mobile Roboter zu identifizieren, wie in [16].

Für die Messung der Distanz wird in der Regel ein Laser benutzt. Der LiDAR-Sensor sendet mit Hilfe eines solchen Lasers einen Lichtimpuls aus und misst die Zeit t bis das Signal wiederkehrt. Anhand dieser Zeit ist es möglich die Distanz d zu bestimmen, die vom Licht zurückgelegt wurde. Dazu nimmt man die Formel 2.4, in der c für die Geschwindigkeit des Lichts steht.

$$d = \frac{c \cdot \Delta t}{2} \quad (2.4)$$

Mit dem Impuls eines einzelnen Lasers bekommt man eine einzelne Abstandsmessung für den Zeitpunkt des Impulses. Führt man aber viele dieser Messungen innerhalb einer 360° Drehung durch bekommt man Abstandsinformationen des gesamten Umfeldes. Eine Möglichkeit eine 360° Umfeldmessung zu ermöglichen ist es den Sensor um seine eigene Achse zu drehen, wie es in Abbildung 2.6 gezeigt wird. In der oberen Reihe dieser Abbildung ist der LiDAR-Sensor zu sehen, dessen Lichtimpuls durch einen Spiegel (graues Objekt) gezielt auf die Umgebung gerichtet wird. Dieser Spiegel dreht sich um die eigene Achse um die Lichtimpulse nach und nach auf das Umfeld zu verteilen. In der mittleren Reihe ist die Vogelperspektive auf den Sensor und seine Umgebung abgebildet. Der Sensor ist, wie in der oberen Abbildung, in blau dargestellt und die Objekte des Umfeldes in grün. Die unteren Abbildungen zeigen, ebenfalls in der Vogelperspektive, die Abstandsinformationen, die

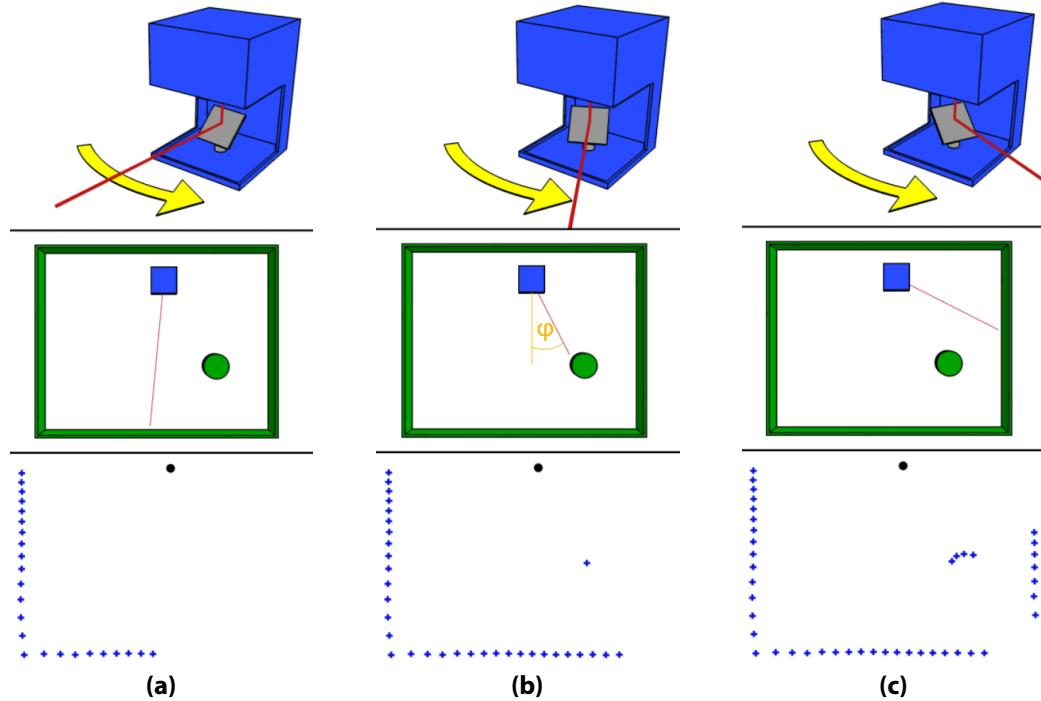


Abbildung 2.6.: Ablauf einer LiDAR-Messung von einem Sensor, der die Messimpulse im Uhrzeigersinn abgibt. Die Bilder stammen von [?] und 2.6b wurde um den Winkel φ erweitert.

der Sensor liefert. Diese werden in Form von blauen Punkten dargestellt und bilden eine zweidimensionale Punktwolke, in der man den Rahmen des Umfelds und das kreisförmige Objekt erkennen kann.

Diese Methode liefert also eine gute horizontale Rundumsicht, welche aber innerhalb der vertikalen Sichtweite keinerlei Informationen bietet. Dies genügt in der Regel für simple Abstandserkennungen und Detektion von Hindernissen auf Höhe des Sensors. Möchte man jedoch Aussagen über das detektierte treffen oder braucht Informationen über andere vertikalen Ebenen, beispielsweise den Boden, muss man diese Methode noch erweitern. Dafür werden mehrere Lichtimpulse mit verschiedenen vertikalen Winkeln ausgesendet. Diese treffen die Umgebung dann mit unterschiedlicher Höhe womit die daraus resultierende Punktwolke eine dritte Dimension erhält. In Abbildung 2.7 ist die seitliche Sicht auf einen LiDAR-Sensor mit mehreren vertikalen Lasern zu sehen.

Wie schon erwähnt werden die erhaltenen Messdaten in eine Struktur konvertiert, die sich Punktwolke nennt. Dafür werden die Abstandsinformationen innerhalb eines bestimmten Koordinatensystems dargestellt. In der Regel wird dabei das *Sphärische Koordinatensystem* verwendet. Darin wird jeder Punkt durch einen Abstand r von einem Referenzpunkt und zwei dazugehörigen Winkel repräsentiert. Einer dieser Winkel ist dabei horizontal (Winkel φ) zu interpretieren und der andere vertikal (Winkel θ).

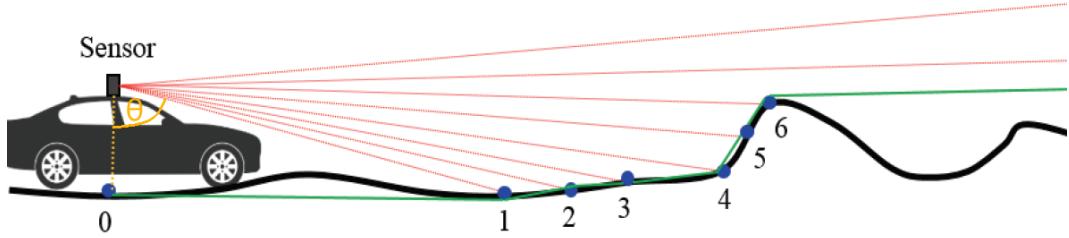


Abbildung 2.7.: Seitliche Sicht auf eine LiDAR-Messung mit mehreren, vertikal angeordneten Lasern. Die Abbildung stammt aus der Arbeit [3] und wurde durch die Einzeichnung des Winkels θ erweitert.

Nimmt man den Punkt „1“ aus der Abbildung 2.7 als Beispiel, entspräche θ dem vertikalen Winkel. Dieser gibt den Winkel zwischen der gelben, gestrichelten Linie und der roten Linie an, die den Sensor und „1“ verbindet. Diese rote Linie repräsentiert den Abstand des Punktes vom Referenzpunkt, welcher die Position des Sensors entspricht. Der horizontale Winkel ist der Rotationswinkel des Sensors um seine eigene Achse (vgl. φ in Abbildung 2.6b). Dieser wird vom Sensor bei jedem abgegebenen Impuls gemessen. Zusammenfassend kann also durch den vertikalen Winkel θ des Lichtimpulses, den horizontalen Winkel φ des Sensors und dem gemessenen Abstand r zum Sensor, der Punkt „1“ genau im Raum definiert werden.

Viele Anwendungen, die mit Punktwolken zu tun haben arbeiten jedoch mit dem kartesischen Koordinatensystem, so auch C.LABEL-VR. In diesem Koordinatensystem wird jeder dreidimensionale Punkt durch drei Koordinaten angegeben, nämlich der x -, y - und z -Koordinate. Ausgehend von den drei Werten des sphärischen Koordinatensystems können die drei kartesischen Koordinaten durch die Formeln in 2.5 berechnet werden. Auch hier steht r für den Abstand zwischen dem Referenzpunkt und dem Punkt den man definieren will. θ steht für den vertikalen Winkel und φ für den horizontalen Winkel, also den Rotationswinkel des Sensors um die eigene Achse.

$$x = r \sin \theta \cos \varphi \quad (2.5a)$$

$$y = r \sin \theta \sin \varphi \quad (2.5b)$$

$$z = r \cos \theta \quad (2.5c)$$

Durch die eben erklärte Definition der Messdaten in einem Koordinatensystem können diese Daten nun mit geeigneten Programmen (C.LABEL) als Punktwolke dargestellt werden. Darin ist dann jede einzelne Messung die der Sensor getätigt hat sichtbar. Somit

kann der Mensch die Daten visuell analysieren und beispielsweise Zusammenhänge wie Objekte entdecken. Damit auch Computersysteme solche Objekte in diesen Daten erkennen können müssen sie erst lernen wie diese aussehen. Dazu muss eine Vielzahl an Punktwolken mit Hilfe eines LiDAR-Sensors aufgenommen werden und anschließend eine, vom Anwendungsfall abhängige, Anzahl an Punkten in dieser Wolke mit entsprechenden Klassifikationen versehen werden (Prinzip der Annotation aus Kapitel 4.4). Mit den annotierten Daten können nun Algorithmen, wie beispielsweise künstliche neuronale Netze (vgl. Kapitel 2.1.1), lernen welche Zusammenhänge es in solchen Daten gibt und diese dann in nicht annotierten Daten wiedererkennen. Das Ziel von C.LABEL-VR ist es solche Daten in der virtuellen Realität zu visualisieren und Funktionen zur Verfügung zu stellen diese in der virtuelle Umgebung zu annotieren.

Systembeschreibung

Das Hauptthema mit dem sich diese Arbeit beschäftigt ist das Virtual Reality Programm C.LABEL-VR. Damit dieses Programm entwickelt und betrieben werden kann bedarf es aber einiger erwähnenswerter Komponenten, welche das System um die Applikation herum repräsentieren. Dazu gehört vor allem eine Virtual Reality Brille. Warum hierfür die Oculus Rift genommen wurde und warum Virtual Reality für solch eine Applikation geeigneter ist als vergleichbare Technologien wird in Kapitel 3.1 erläutert. Der Computer der für den Betrieb der VR-Brille benutzt wurde und was bei der Auswahl dessen Komponenten wichtig war ist in Kapitel 3.2 beschrieben. Mit den eben genannten Komponenten lässt sich eine VR-Applikation zwar ausführen, jedoch nicht entwickeln. Dafür ist zusätzlich eine Entwicklungsumgebung notwendig, genauer gesagt eine Spiel-Engine (*Game Engine*). Welche Engine in dieser Arbeit verwendet wurde und welche grundlegenden Konzepte dieser wichtig für die Entwicklung waren, wird in Kapitel 3.3 beschrieben. [17]

3.1. Komponente 1: Die VR-Brille

Wie schon erwähnt, war die praktische Hauptaufgabe dieser Arbeit die Realisierung einer Applikation zur Annotation von Punktwolken in einer dreidimensionalen Umgebung. Hierfür wurde die Virtual Reality Technologie verwendet, welche im Kapitel 2.2 näher erläutert wird. Diese ist jedoch nicht die einzige Technologie die für diesen Anwendungsfall in Frage kam. Auch die *Augmented Reality* (AR) bietet einige interessante Konzepte um solch eine Applikation zu entwickeln. Warum hierbei aber VR zu bevorzugen ist, wird im folgenden Abschnitt 3.1.1 erläutert. Im anschließenden Abschnitt 3.1.2 wird gezeigt welche VR-Brillen für die Entwicklung in Frage kamen und warum die Oculus Rift aus diesen ausgewählt wurde.



(a) Microsoft Hololens AR-Brille

(b) Meta 2 AR-Brille

Abbildung 3.1.: Die zwei potentiellen AR-Brillen

3.1.1. Warum VR die passende Technologie für C.LABEL-VR ist

Um das Labeling von Daten, insbesondere Punktwolken, im dreidimensionalem Raum zu verwirklichen, gibt es zwei wesentliche Technologien die dafür relevant erscheinen. Neben VR, das schon vorgestellt wurde (Kapitel 2.2), gibt es noch die Augmented Reality Technologie. Unter AR versteht man die direkte oder indirekte Sicht auf die reale, physische Welt, welche durch digitale Inhalte erweitert wird. Dies wird vor allem durch Smartphones oder AR-Brillen realisiert. Bei einem Smartphone hat man beispielsweise einen indirekten Blick auf die reale Umgebung durch das Display, welches ein Live-Bild oder ein aufgenommenes der Kamera anzeigt. Diesen Bildern können nun digitale Inhalte hinzugefügt werden. Bei einer Brille (vgl. Abbildung 3.1) sieht man durch die Gläser direkt auf die physische Welt. Hierbei fungieren die Gläser als Display, welche in der Lage sind holographische Objekte anzuzeigen. Dies führt zu einer, für den Benutzer, sehr immersiven Vermischung der realen und der digitalen Welt. Üblicherweise ist es bei diesen Brillen sogar möglich durch Gesten, die mittels Händen durchgeführt werden, mit den Hologrammen zu interagieren.

Eignung von Augmented Reality für 3D-Annotation

Der, im Rahmen dieser Arbeit, entwickelte Prototyp zur 3D-Datenannotation ist nicht nur als Nutzungssoftware geplant, sondern dient auch als Anschauungsmaterial, beispielsweise für Messen. Zieht man diese Tatsache in Betracht, eignet sich Augmented Reality sehr gut für diesen Zweck. Es wirkt durch die Vermischung von realen und digitalen Inhalten nämlich sehr futuristisch. Zudem ist das direkte einblenden holographischer Inhalte, durch eine AR-Brille, zum Zeitpunkt der Erstellung dieser Arbeit, noch wenig verbreitet (Microsoft Hololens wurde nur wenige Tausend mal verkauft [18]), was bei unwissenden Benutzern zu einem beeindruckenden Effekt führt. Des Weiteren bietet die Augmented Reality Technologie viele Möglichkeiten, die für zukünftige Labeling-Methoden relevant sein könnten. Den heutigen AR-Brillen ist es durch Tiefenkameras möglich Objekte und deren Distanzen zur Brille wahrzunehmen. Folglich könnte ein zukünftiger Ansatz sein, eine Punktwolke der Umgebung, wie sie in Abbildung ?? zu sehen ist, mit einer AR-Brille zu erstellen. Diese kann anschließend vor Ort, in der Umgebung in der die Punktwolke

erstellt wurde, klassifiziert werden.

Ein wichtiger Punkt der ebenfalls betrachtet werden muss ist die Bedienung der Brille bzw. die Interaktion mit den digital dargestellten Objekten und Informationen. Die Steuerung von AR-Brillen erfolgt handelsüblich über Gesten, welche mit der Hand bzw. den Händen getätigten werden. Ob diese Art der Interaktion für den hier gewünschten Anwendungsfall passend wäre, lässt sich im Voraus schwer feststellen. Denkbar wäre, dass sich der Benutzer durch intuitive Gesten, die man auch bei realen Objekten nutzt, schnell an die Bedienung des Tools gewöhnen würde. Andererseits könnte die Selektion der Elemente einer Punktwolke zu ungenau sein, da die Brille die Position der Hand nicht richtig interpretiert bzw. diese nicht richtig zur Position der digitalen Inhalte interpretiert. Eine genaue Einschätzung der Bedienung für solch einen Anwendungsfall kann allerdings nur gegeben werden indem man einen Labeling-Prototypen erstellt und ihn anschließend über längere Zeit testet. Dies ist im Zeitrahmen einer Masterarbeit jedoch nicht möglich. Falls sich nämlich die Steuerung als ungeeignet herausstellt bleibt nicht genug Zeit für eine Neuentwicklung auf einer anderen Plattform.

Darüber hinaus gibt es natürlich auch Aspekte die gegen die AR-Technologie sprechen. Die Klassifizierung einer Punktwolke mit dieser Technik ist an einem normalen Büro-Arbeitsplatz nicht möglich, zumindest nicht in einem Maßstab in dem es sinnvoll wäre. Die holographischen Punkte kollidieren, wenn sie darauf programmiert sind, mit der Umgebung und so würde Darstellung des Umfeldmodells verfälscht werden. Auch wenn sie das nicht tun, dann wäre die Umgebung selbst immer noch ein Hindernis für den Benutzer, denn man möchte sich ja durch die Punktwolke bewegen.

Des Weiteren sind bei AR-Brillen die vorherrschenden Lichtbedingungen ein großer Faktor, welche die Funktionalität gewisser Anwendungsfälle beeinflussen können. Wird beispielsweise eine Lichtquelle zu sehr von einem Objekt reflektiert, kann dieses von der Tiefenkamera der Brille nicht mehr richtig erfasst werden. Somit stimmt das Umfeldmodell nicht mehr mit der Realität überein. Zudem wird nicht nur die Funktionalität von Lichteinflüssen gestört, sondern auch die Darstellung der Hologramme. Bei zu viel Licht sind diese deutlich schwerer zu erkennen, vergleichbar mit der Nutzung eines Laptops im Freien, bei dem der Bildschirminhalt wegen zu heller Umgebung ebenfalls schlecht erkennbar ist.

Für die Entwicklung selbst ist es ebenfalls hinderlich das es, zum Zeitpunkt der Erstellung dieser Arbeit, wenig Dokumentationen und Hilfestellungen, zum Beispiel in Entwicklerforen, gibt. Dies ist jedoch üblich für Technologien, die noch nicht lange auf dem Markt sind. Zu guter Letzt ist noch der finanzielle Faktor zu berücksichtigen. Durch die Aktualität der Technologie ist diese auch sehr teuer. Der Preis für eine AR-Brille kann dabei den Betrag von 5.000 Euro überschreiten [19].

Eignung von Virtual Reality für 3D-Annotation

Die Virtual Reality Technologie bietet den großen Vorteil, beliebig große Räume virtuell begehbar zu machen, ohne sich in der realen Welt selbst bewegen zu müssen. Dies ermöglicht die Bewegung durch Punktewolken, die im dreidimensionalen Raum dargestellt sind, an jedem üblichen Arbeitsplatz eines Büros. Des weiteren wirkt die Steuerung der VR-Brillen mittels den zugehörigen Controllern zum Teil sehr ausgereift. Die Positions- und Bewegungserkennung des Benutzers wird als „*äußerst präzise*“ beschrieben [20]. Dies ist sehr wichtig um die Handhabung der Anwendung für den späteren Endnutzer so angenehm wie möglich zu machen.

Für die Entwicklung von VR-Applikationen selbst ist zu sagen, dass es mittlerweile sehr umfassende Dokumentationen, Anleitungen und Beiträge zu den jeweiligen Plattformen und Software Development Kits (SDKs) gibt. Dies hängt damit zusammen, dass sich die Virtual Reality Technologie schon etwas länger auf dem Markt befindet und die Zahl der Entwickler für VR-Applikationen stetig steigt. Dies ist vermutlich nicht zuletzt der Tatsache zu verdanken, dass die Preise für VR-Brillen gesunken sind. Der Preis einer Oculus Rift beträgt zum Zeitpunkt dieser Arbeit 449 Euro. Sowohl die zahlreichen Informationen für Entwickler als auch der niedrige Preis der Hardware sprechen, neben den zuvor genannten Aspekten, für die Wahl von Virtual Reality als Plattform für die Entwicklung von 3D-Labeling.

Was mit der VR-Technologie nicht ohne Weiteres funktioniert, ist die physische Begehung einer Punktewolke. Zwar bietet die Technik die Möglichkeit durch zusätzliche Sensoren die Bewegungen des Benutzers in die virtuelle Welt zu übersetzen, jedoch funktioniert dies nur auf begrenztem Raum und ist unmöglich an einem üblichen Büro-Arbeitsplatz durchführbar. Der reale begehbarer Raum muss nämlich durch diese Sensoren abgesteckt werden und ist somit durch deren Reichweite und Genauigkeit begrenzt. Ein weiter Negativaspekt ist, dass viele Benutzer von VR-Brillen über Übelkeit klagen. In manchen Tests sind es mehr als 50 Prozent aller Teilnehmer, vor allem wenn es sich um Frauen handelt [21].

Entscheidung

Für die Wahl des passenden Mediums, auf dem das dreidimensionale Labeling entwickelt werden soll, sind mehrere Faktoren entscheidend. Diese Faktoren ergeben sich danach, wie sehr sich etwas für die genannte Anwendung eignet. Zunächst ist es wichtig eine Punktewolke deutlich und nach Möglichkeit im richtigen Maßstab darzustellen. Hier kommt die Augmented Reality-Technologie sicherlich an ihre Grenzen. Grund dafür ist die ungenaue Interaktion mit der Umgebung und die lichtempfindlichen Darstellung digitaler Inhalte. In der virtuellen Realität dagegen kann eine solche Wolke problemlos bei jeder Bedingung angezeigt werden, da die reale Umgebung nicht berücksichtigt wird. Sogar der Maßstab

kann gut eingehalten werden, da dem virtuellen Raum keine Grenzen gesetzt sind, sodass beliebig große Umfeldmodelle an jedem Arbeitsplatz dargestellt werden können.

Auch die Steuerung spricht für den Einsatz von Virtual Reality. Zwar ist das Konzept der Greif-Steuerung der Meta 2 ein guter Ansatz, da diese aber noch nicht zuverlässig und genau funktioniert, ist sie für die Selektion vieler Objekte nicht gut geeignet. Dagegen bietet die Steuerung durch Controller bei VR-Brillen deutlich mehr Möglichkeiten eine gute Bedienung für die gewünschte Anwendung zu entwickeln. Die Tasten eines solchen Controllers können vom Entwickler nämlich nach Wunsch programmiert werden wogegen die Gesten für AR-Brillen nicht verändert bzw. nicht neu erstellt werden sollten [22]. Zudem gibt es, durch die deutlich frühere Markteinführung, viel mehr Dokumentationen und Hilfestellungen im Bereich Virtual Reality, was die Entwicklung deutlich erleichtert. Aufgrund der eben genannten Vorteile von VR gegenüber AR wird für die Entwicklung, im Rahmen dieser Arbeit, **Virtual Reality** verwendet.

3.1.2. Warum die Oculus Rift verwendet wurde

Die, in dieser Arbeit, entwickelte Methode zur 3D-Datennotation ist vorwiegend für einen normalen Arbeitsplatz im Büro konzipiert, also einen Platz an dem man einen Desktop-Computer zur Verfügung hat. Deswegen macht es keinen Sinn VR-Brillen in Betracht zu ziehen die von einem mobilen Gerät betrieben werden (*Samsung Gear VR*), da diese deutlich weniger Leistung und eine schlechtere Bedienung haben als Geräte für einen Desktop-PC. Auch die Nutzung der *Playstation VR* ist nicht sinnvoll, da zum Betrieb eine Playstation 4 nötig ist, welche in den wenigsten Unternehmen vorhanden sein dürfte.

Für das Klassifizieren im dreidimensionalen Raum ist eine gute Darstellung und Bedienung notwendig. Bei VR-Brillen, die einen Desktop-PC als Recheneinheit haben wird beides geboten. Durch die hohe Leistung eines performanten Computers kann der dreidimensionale Raum hochauflösend dargestellt werden. Auch die genaue, für Spiele konzipierte, Steuerung über Controller kommt einem Labeling-Tool zugute. Zum Zeitpunkt der Erstellung dieser Arbeit, gibt es zwei relevante VR-Brillen aus dieser Sparte, die *Oculus Rift* und die *HTC Vive*.

Oculus Rift

Mit 470 Gramm ist die Oculus Rift (vgl. Abbildung 3.2b) deutlich leichter als andere Modelle, wie beispielsweise die HTC Vive mit 600 Gramm oder die Playstation VR mit 610 Gramm [25]. Dies ist für längeres Arbeiten mit ihr sehr vorteilhaft, da schwere Brillen oft schon nach kurzer Zeit störend beim Tragen sind. Des weiteren lässt sich die Brille problemlos an jedem handelsüblichen Büro-Arbeitsplatz verwenden. „*Oculus empfiehlt, die*



(a) HTC Vive VR-Brille mit Controllern [23]



(b) Oculus Rift VR-Brille mit Controllern [24]

Abbildung 3.2.: Die zwei potentiellen VR-Brillen

*Kameras im Abstand von zwei Metern nebeneinander zu platzieren – bei einem klassischen PC-Arbeitsplatz also links und rechts neben dem Monitor. Das funktioniert in der Praxis gut, man kann sich mit diesem Aufbau ungefähr jeweils einen Schritt in alle Richtungen bewegen“ [26]. Auch in Sachen Steuerung ist die Oculus Rift anderen Modellen überlegen. Durch die sogenannten *Oculus Touch Controller*, mit denen man virtuell nicht nur Greifen, sondern auch Daumen und Zeigefinger bewegen kann, gelingen ihr „deutlich feinere Bewegungen“ als beispielsweise der HTC Vive [26]. Dies bietet für die Entwicklung einer Lösung zur Markierung vieler Objekte zahlreiche Möglichkeiten.*

Was bei der Oculus Rift nicht so gut funktioniert, wie bei anderen Brillen ist das Roomscaling, also das Erfassen der Bewegungen des Benutzers durch einen größeren Raum. Dieser Aspekt erschwert somit das räumliche Begehen einer Punktwolke. Für dieses Verfahren gibt es bei der Rift zwei Methoden. Einmal mit zwei Kameras (kleine Überwachungsfläche), die sich im Abstand von drei Metern in zwei Raumecken gegenüberstehen und eine mit drei Kameras (größere Überwachungsfläche). Beide Methoden funktionieren nicht einwandfrei und auch nicht so gut wie bei der HTC Vive [26].

HTC Vive

Die HTC Vive (vgl. Abbildung 3.2a) punktet mit dem ausgeklügelten Lighthouse Tracking System, welches in Zusammenarbeit mit VALVE entwickelt wurde [27]. Das System kann die Kopfbewegungen des Benutzers durch mehr als 70 Sensoren auf bis zu ein Zehntel eines Grades messen [28]. Die Sensoren sind dabei sowohl in die Brille integriert als auch extra positioniert. Auch die Controller werden durch dieses System erfasst, was zu einer genauen Übertragung der physischen Handbewegungen in die virtuelle Welt führt. Dies kommt der Auswahl zahlreicher Objekte, wie es bei der Klassifizierung von Punkten notwendig ist, sehr entgegen. Auch eine gute Darstellung ist aufgrund zweier Displays gewährleistet, die jeweils eine Auflösung von 1080 auf 1200 Pixel haben. Bei diesen Displays kommt es weder zu Verzerrungen des Bildes („lens distortion“ [28]) noch zu Pixelfehlern. Ebenfalls nützlich, vor allem am Arbeitsplatz, ist die Bluetooth-Funktion der Vive. Damit ist es möglich sein

Smartphone mit der Brille zu verbinden und somit eingehende Emails oder Ähnliches zu lesen und zu beantworten, ohne die Brille absetzen zu müssen.

Der große Unterschied zwischen Vive und Rift liegt bei der Bedienung durch die unterschiedlichen Controller. Hier hat die HTC-Brille das Nachsehen, weil sie durch Anatomie der Controller weniger Möglichkeiten zur Interaktion in der virtuellen Umgebung bietet. Die sogenannten *Wands* bieten nämlich keinerlei Funktionen die es ermöglichen eine Hand nachzuahmen, um beispielsweise einzelne Finger zu bewegen und somit virtuelle Objekte greifen zu können. Im Hinblick auf eine durchdachte Steuerung für die Auswahl und Markierung vieler Punkte in einer Punktewolke gibt es somit weniger Möglichkeiten diese zu entwickeln.

Entscheidung

Bleibt noch die Frage zu klären welche Brille verwendet werden soll. Diese Frage ist schwerer zu beantworten als die vorherige. Die zwei potentiellen VR-Brillen, also Oculus Rift und HTC Vive, sind beide geeignete Kandidaten. Die Vive punktet durch gutes Roomscaling und gutes Erfassen der Brille und der Controller, wogegen die Oculus Rift guten Tragekomfort und gut durchdachte Controller-Bedienung bietet.

Wichtig für die Entwicklung einer 3D-Datennotation ist aber nicht, ob man sich physisch, also mittels Roomscaling durch die Punktewolke bewegen kann, sondern wie man die klassifizierbaren Punkte markieren bzw. auswählen kann. Da hierfür die Oculus Touch Controller mehr Möglichkeiten bieten wird für die Entwicklung eines VR-Labeling-Tools die **Oculus Rift** verwendet.

3.2. Komponente 2: Der Rechner

Für die Berechnungen, die notwendig sind um Inhalte in der Oculus Rift anzuzeigen, ist nicht die Brille selbst verantwortlich. Diese Aufgabe übernimmt ein separater Computer. Die Brille fungiert dementsprechend nur als Anzeigegerät. Auch die Sensoren sind an den PC angeschlossen um das Erfassen der Brille und der Controller zu gewährleisten. Vor allem die Grafikberechnungen sind sehr aufwendig sind, da die Darstellung von VR-Inhalten zwei mal berechnet werden (für jedes Auge extra) und nicht nur einmal, wie es an einem Bildschirm der Fall ist. Deshalb kann nicht jeder handelsübliche PC oder Laptop dazu verwendet werden eine solche VR-Brille zu betreiben. Was ein passender Computer für die Nutzung von Virtual Reality bieten muss wird im Folgenden erläutert.

Wichtig sind zunächst die Basiskomponenten wie ein schneller Prozessor und eine leistungsstarke Grafikkarte. Aber auch ein schneller Arbeitsspeicher mit ausreichender Kapazität ist wichtig damit notwendige Daten so schnell wie möglich zur Verfügung stehen.

Für die Wahl, welche Komponenten eingesetzt werden sollen, kann sich an den empfohlenen Spezifikationen des Herstellers orientiert werden (Oculus Rift Recommended System Specs [29]). Diese sind jedoch vorwiegend für den Endnutzer gedacht, bei dem es wichtig ist ein einzelnes Programm für die Brille auszuführen. Für die Entwicklung ist dagegen mehr Leistung notwendig. Soll zum Beispiel eine erstellte Applikation getestet werden, muss nicht nur die Applikation selber ausgeführt werden, sondern auch die Entwicklungsumgebung und zusätzliche Fehlerbehebungs- und Analyse-Anwendungen. An dieser Stelle darf also nicht gespart werden.

Sind die Basiskomponenten ausgewählt, sollte noch geprüft werden ob noch weiteres Zubehör wichtig ist um diese zu betreiben. Da gerade Prozessoren mit hoher Taktrate viel Wärme produzieren, muss stets für ausreichende Kühlung gesorgt werden. Bei der Rechner-Konfiguration für diese Masterarbeit wurden beispielsweise, neben einem hochwertigen Prozessorkühler, noch zwei weitere Gehäuselüfter verbaut, um die Abwärme aus dem inneren des Computers heraus zu leiten.

Zu guter Letzt ist es wichtig ein passendes Mainboard auszuwählen, auf dem alle Komponenten verbaut werden. Für die Oculus Rift werden vier USB-Anschlüsse empfohlen, welche das Mainboard haben muss [29]. Ebenfalls muss das Board genug Platz bieten um die gewählten Komponenten zusammen anbringen zu können. Leistungsstarke Grafikkarten und Prozessorlüfter können zum Teil sehr groß ausfallen. Auch für weitere Aufrüstungen sollte das Board genügend Platz bieten, falls es, zu einem späteren Zeitpunkt, notwendig wäre zusätzliche Komponenten (zweite Grafikkarte) hinzuzufügen. Unter Berücksichtigung der eben genannten Aspekte wurde folgende Rechner-Konfiguration für die Entwicklung verwendet:

Bezeichnung der Komponente	Verwendete Komponente
<i>Prozessor:</i>	Intel® Core™ i7-7700K
<i>Grafikkarte:</i>	Gainward GeForce GTX 1070 Phoenix
<i>Mainboard:</i>	ASUS PRIME Z270-A
<i>Arbeitsspeicher(RAM):</i>	HyperX DIMM 16 GB DDR4-2400 Kit
<i>Festplatte:</i>	Samsung 850 Pro 2,5" 512 GB
<i>Netzteil:</i>	Cooler Master G550M 550W
<i>Prozessorlüfter:</i>	Noctua NH-D9L
<i>Gehäuselüfter:</i>	2x Coolink SWiF2-1200 120x120x25
<i>Gehäuse:</i>	Cooler Master N300

Tabelle 3.1.: Rechner-Konfiguration für die Entwicklungen dieser Arbeit

3.3. Komponente 3: Die Entwicklungsplattform

Für die Entwicklung von Virtual Reality Anwendungen werden in der Regel sogenannte Spiel-Engines (*Game Engines*) verwendet. Dies sind komplexe Mehrzweckwerkzeuge für die Erstellung von Multimedia-Inhalten und Videospielen. Eine Spiel-Engine bietet Funktionen, welche die wichtigsten Bereiche der Spielentwicklung abdecken. Eine davon ist die Bildsynthese aus Rohdaten, welche meist geometrische Beschreibungen im zwei- oder dreidimensionalen Raum sind. Die Bildsynthese ist in der Fachsprache besser bekannt als *Rendering*. Der Prozess des Renderings berechnet für jedes angezeigte Bild die Objekte, welche vom Benutzer aus sichtbar sind, deren Oberflächen anhand von ihren Materialeigenschaften und die Lichtverhältnisse des Bildes, die sich unter anderem durch die indirekte Beleuchtung zwischen Körpern äußert. Weitere Funktionen, welche diese Plattformen zu Verfügung stellen, sind beispielsweise Physikalisch Berechnungen (z.B. Schwerkraft), Animationen von Objekten und künstliche Intelligenzfunktionen.

Für die Entwicklung von VR-Inhalten ändert sich an den Funktionen, welche die Engine bieten muss nicht all zu viel, da die meisten Anwendungen schon dreidimensional sind. Die meisten Elemente der Bildsynthese müssen allerdings, wie schon im vorherigen Abschnitt 3.2 erwähnt, zweimal berechnet werden, nämlich für jedes Auge einzeln. Im Folgenden soll erläutert werden warum für diese Arbeit die Unity Engine als Entwicklungsplattform ausgewählt wurde. Anschließend wird diese kurz vorgestellt.

Vergleich zwischen Unreal und Unity

Für die Erstellung von Virtual Reality Software gibt es zwei relevante Entwicklungsumgebungen, die gut für das Entwickeln von Virtual Reality geeignet sind. Der Grund dafür ist, dass sie schon seit längerem die Erstellung von VR-Anwendungen unterstützen und dies auf umfassende Weise. Die zwei Entwicklungsumgebungen sind *Unreal Engine 4* und *Unity Engine*. Beide sind für nicht kommerzielle Zwecke zunächst kostenlos benutzbar. Beide Plattformen bieten alles nötige um VR-Applikationen zu entwickeln, weshalb die Wahl zwischen ihnen eher subjektiv, aufgrund der eigenen Bedürfnisse und Erfahrungen, zu fällen ist.

In [30] wurden diese beiden Entwicklungsplattformen gegenübergestellt. Im Allgemeinen kann man sagen, dass die Unreal Engine mehr Möglichkeiten für professionelle Spieleentwicklung bietet, da viel Wert auf visuelle Qualität gelegt wird. Beispielsweise wird durch integrierte Post Processing-Techniken und gute Shader das erzeugte Bild besser dargestellt als in Unity. Sie bietet neben dem Scripting auch die Gelegenheit, durch C++ Programmierung Module der Engine zu verändern bzw. neue hinzuzufügen. Das gestalten von User Interfaces ist mit dem *UMG UI Designer* ebenfalls gut gelöst .

Die Unity Engine bietet vor allem Anfängern einen leichten Einstieg in die Welt der Grafik- und Spieleprogrammierung. Im Editor können ganz einfach sogenannte *Game Objects* erstellt werden, denen dann Funktionalitäten und Eigenschaften angehängt werden können. Dies geschieht meist in Form von Skripten, welche oft schon vorgefertigt zur Verfügung stehen. Wenn dies nicht der Fall ist können diese angepasst oder neu erstellt werden. Die große Community der Unity Engine ist hierbei eine große Hilfe.

Entscheidung

Ein Vorteil den die Unity Engine, im Falle dieser Arbeit, bietet ist Nutzung von C# als Skript-Sprache. Da C.LABEL (vgl. 1.3.2) ebenfalls in C# programmiert wurde können Komponenten, wie etwa das Einlesen von Daten, leicht in die VR-Anwendung übernommen werden. Des Weiteren habe ich selbst, im Rahmen eines Universitätsprojekts, mit dieser Plattform gearbeitet. Ein längeres einarbeiten in die Entwicklungsumgebung wäre demnach nicht nötig. Die Vorteile der Unreal Engine sind, wie schon erwähnt, die visuelle Präsentation und die dabei gebotene Performance. Diese sind vor allem für das Erstellen aufwändiger Landschaften und deren Inhalten von Vorteil.

Für die Entwicklung der angestrebten Anwendung spielen jedoch Dinge wie Objekt- und Landschaftsdarstellung keine große Rolle. Viel wichtiger ist sowohl die Möglichkeit der Kompatibilität zur C.LABEL-Anwendung, als auch die vorhandene Entwicklungserfahrung mit einer Plattform. Gerade letzteres ist auf Grund des begrenzten Zeitrahmens einer Masterarbeit von Vorteil, da ohne große Einarbeitung mehr Zeit zum Entwickeln von Funktionalität und Optimierungen bleibt. Aus diesen Gründen wird im weiteren Verlauf dieser Arbeit die **Unity Engine** als Entwicklungsumgebung verwendet.

Entwickeln mit Unity

Die Unity Engine ist eine der am häufigst benutzten Spiel-Engines auf dem Markt. Gerade im Bereich der mobilen Geräte wurden 34 Prozent der 1000 beliebtesten Spiele mit Unity erstellt [31]. Sie ist ebenfalls bei Neueinsteigern und Hobbyprogrammierern beliebt, da sie über umfangreiche Lernmaterialien und eine große Entwicklergemeinschaft verfügt. Die Unity Engine unterstützt sowohl alle gängigen großen Plattformen wie PC, Playstation und XBOX, als auch die kleinen mobilen Geräte wie Android, iOS, Nintendo 3DS und PS Vita. Darüber hinaus kann man mit ihr auch Anwendungen für, in der Applikationsentwicklung, weniger gebräuchliche Plattformen wie Samsung SMART TV, oder tvOS erstellen. Wichtig für diese Arbeit ist die umfangreiche Unterstützung von Virtual Reality Brillen, speziell der Oculus Rift. Um die Vorgänge von C.LABEL-VR, die in Kapitel 4 beschrieben werden, besser zu verstehen, werden im Folgenden die Grundlagen für die Entwicklung mit Unity etwas näher erläutert.

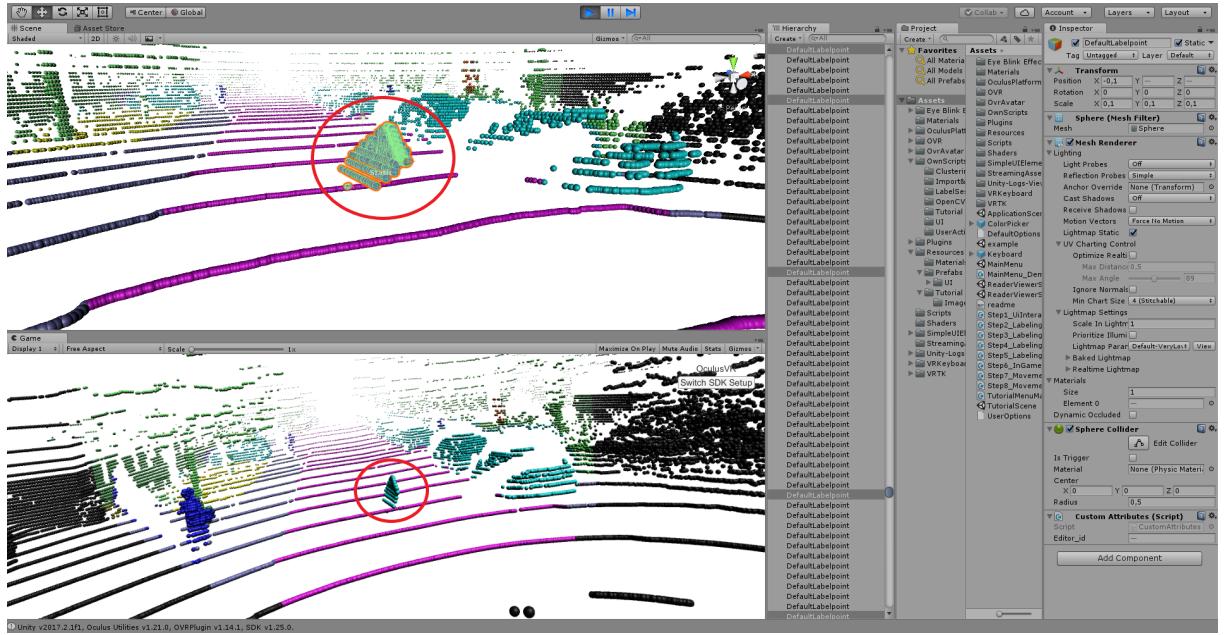


Abbildung 3.3.: Unity Editor, der C.LABEL-VR im Play Mode ausführt.

Bei der Entwicklung mit Unity werden Anwendungen mittels Szenen konzipiert. Eine Anwendung besteht aus mindestens einer Szene, in der Regel sind es aber mehrere. Jede Szene kann dabei eigene Objekte und Logiken beinhalten und es kann je nach Programmierung des Entwicklers zwischen den Szenen gewechselt werden. Ein klassischer Fall dafür wäre eine Applikation mit einer Hauptmenü- und einer Anwendungsszene. Als erstes wird die Hauptmenü-Szene gestartet und von dieser aus dann, beispielsweise durch die Betätigung eines Knopfes, die Anwendungsszene. In größeren Spielen ist es üblich jeden abgetrennten Bereich im Spiel mit einer extra Szene zu erstellen. Dies dient der Übersicht und der Wartbarkeit, die in Szenen mit zu vielen Objekten nicht mehr gegeben wäre.

Die Konzeption der einzelnen Szenen geschieht, wie in der Regel in bei jeder Spiel-Engine, über einen Editor (siehe Abbildung 3.3). Die Konzeption beinhaltet beispielsweise die Erstellung von Landschaften und deren Objekten, Anordnung der Objekte und das Testen der Szenen. Für die Erstellung von komplexen 3D-Objekten werden externe Programme wie *Blender* verwendet, deren Erzeugnisse dann in den Unity Editor importiert werden können [32]. Die Erstellung von simplen primitiven Objekten wie Kugeln oder Würfeln ist aber auch direkt im Unity Editor möglich.

Mit der Anordnung der Objekte ist zum einen die Platzierung der Objekte innerhalb der Szene gemeint, welche beim Beginn der Szene die Startposition der Objekte ist. Diese kann aber während des Ablaufs der Szene durch Programmierung geändert werden. Zum anderen ist die Anordnung der Objekte innerhalb von Hierarchien gemeint. Dabei werden *Eltern-Kind*-Beziehungen zwischen den Objekten festgelegt, was zur Folge hat, dass Änderungen bei den Eltern sich auch auf die Kinder auswirken. Erstellt man zum Beispiel ein leeres Objekt „Mensch“ und ordnet diesem Kopf, Arme, Beine und Oberkörper zu, so kann man

das Objekt „Mensch“ bewegen und alle Körperteile bewegen sich mit.

Das Testen einer oder mehrerer Szenen ist mit dem *Unity Play Mode* möglich. Startet man diesen Modus wird die aktuelle Szene, die im Editor geöffnet ist, gestartet ohne die gesamte Applikation vorher kompilieren zu müssen. Das spart dem Entwickler Zeit da zum einen die Kompilierzeit wegfällt und zum anderen die Applikation nicht immer von der Startszene aus gestartet werden muss. Der Editor in der Abbildung 3.3 befindet sich im Play Modus. In diesem Editor sind zwei große Fenster abgebildet, in denen die laufende Szene zu sehen sind. Das obere Fenster zeigt die Editor-Sicht. Innerhalb dieser können Objekte während des Ablaufs der Szene ausgewählt und geändert werden. Im roten Kreis sind markierte Objekte zu sehen, die nach links verschoben wurden. Im unteren Fenster ist die Benutzer-Sicht zu sehen, welche die Szene aus der Sicht des Benutzers zeigt. Dadurch kann überprüft werden wie sich die getätigten Änderungen aus der Editor-Sicht auf die Tatsächliche Sicht des Benutzers auswirken. Auch hier sieht man im roten Kreis die Objekte die im Editor verschoben wurden.

Die erwähnten Objekte, welche sich in den einzelnen Szenen befinden, werden in Unity *Gameobjects* genannt. Ein Gameobject ist die übergeordnete Klasse für alle Arten von Objekten in Unity, beispielsweise 2D- und 3D-Objekte, aber auch Elemente von Benutzeroberflächen. Jedes Objekt in Unity ist also ein Gameobject und es ist möglich diesen Objekten bestimmte Komponenten zuzuordnen. Mit diesen Komponenten lassen sich Eigenschaften und Verhaltensweisen der Objekte festlegen. Jedes Gameobject in Unity hat standardmäßig eine, nicht entfernbare, Komponente die sich *Transform* nennt. Diese legt die Position, Orientierung und Größe des Objekts in der Szene fest. Sichtbare Objekte müssen beispielsweise eine *Mesh Renderer*-Komponente haben, welche die Materialeigenschaften des Objektes festlegt und somit der Unity Engine signalisiert, dass das Objekt bei der Bildsynthese berücksichtigt werden muss. Es kann aber auch unsichtbare Objekte geben, die lediglich eine Logik-Komponente haben, welche beispielsweise Aktionen anderer Objekte überwacht.

Die Komponenten legen die Eigenschaften und Verhaltensweisen von Objekten mit Hilfe eines Skriptes fest, welches mit einer bestimmten Programmiersprache erstellt wird. In Unity stehen die Sprachen C# und JavaScript zur Verfügung. Das Implementieren dieser Skripte wird in der Fachsprache als *Scripting* bezeichnet. Dadurch kann der Programmierer, wie schon erwähnt, die Eigenschaften und das Verhalten von Objekten festlegen und beeinflussen. Zur Veranschaulichung dieses Vorgangs dient die Auflistung 3.1. Darin ist ein minimalistischer Grundaufbau eines Skriptes zu sehen, dass ein Objekt unter einer Bedingung größer werden lässt.

Auflistung 3.1: Grundaufbei eines Gameobjects in Unity

```
using UnityEngine;

public class TestObject : MonoBehaviour
{
    public static TestObject Instance { get; set; }
    public bool EnableGrowing { get; set; }

    private void Awake()
    {
        Instance = this;
    }
    private void Start()
    {
        Instance.EnableGrowing = true;
    }
    private void Update()
    {
        if(Instance.EnableGrowing)
            Instance.transform.localScale += new Vector3(1, 1, 1);
    }
}
```

Ein Skript besteht in der Regel aus einer Klasse, wie beispielsweise *TestObject* in Auflistung 3.1, die von der Unity-Basisklasse *MonoBehaviour* erbt. Diese Vererbung führt dazu, dass der Klasse *TestObject* diverse Funktionen zugeordnet werden, die von der Unity Engine zu gewissen Zeitpunkten aufgerufen werden. Dadurch kann das Verhalten des Objekt gezielt gesteuert werden. Diese Funktionen werden jedoch nur aufgerufen, wenn sie innerhalb der erbenden Klasse auch implementiert werden. All diese Funktionen sind in der Dokumentation der Unity Engine aufgelistet und beschrieben [33]. Die Klasse *TestObject* implementiert drei von diesen Funktionen, nämlich *Awake()*, *Start()* und *Update()*.

Awake wird als erstes aufgerufen, sobald alle Objekte in Unity geladen sind und kann somit zur Initialisierung von Referenzen verwendet werden bevor die Szene startet. Im Beispiel wird der statischen Instanz von *TestObject* die Referenz des geladenen Skripts *TestObject* zugewiesen. Dieses Vorgehen nennt sich *Singleton-Pattern* und ist eine übliche Methode zum verwalten von Referenzen in Unity [34]. Dadurch ist es für andere Skripte möglich, ohne Instanziierung der Klasse, auf diese zuzugreifen.

Die Methode *Start* wird vor dem ersten Aufruf von *Update* ausgeführt, also kurz vor dem Start der Szene. Hier können somit noch Werte zugewiesen werden. *EnableGrowing* wird im Skript *TestObject* auf den boolschen Wert „wahr“ gesetzt. Dieser kann dann von anderen Skripten durch das eben beschriebene Singleton-Pattern geändert werden

(*TestObject.Instance.EnableGrowing = false*).

Die Update-Methode ist eine Art Endlosschleife die von der Unity Engine immer wieder aufgerufen wird solange das Skript in einem aktiven Zustand ist. Sie repräsentiert die, in der Fachsprache oft genannte, *Game-Loop*. Alle Vorgänge die während der Laufzeit der Applikation ablaufen sollen müssen also in dieser Funktion implementiert werden. Im Beispiel wird die Größe des Objektes erhöht, solange die Variable EnableGrowing den boolschen Wert „wahr“ hat. Es gibt darüber hinaus noch einige weitere dieser Methoden innerhalb der MonoBehaviour-Klasse, beispielsweise die Methode *OnEnable()*, die aufgerufen wird wenn das Skript vom inaktiven in den aktiven Zustand wechselt. Mit Hilfe all dieser MonoBehavior-Funktionen lassen sich also Objekte und deren Verhalten manipulieren. Auf diese Weise werden gewünschte Vorgänge in Unity implementiert.

In diesem Abschnitt wurden nun grundlegende Vorgänge beschrieben, die wichtig sind um Anwendungen mit Unity zu erstellen. Darunter waren zum Beispiel das Konzipieren von Szenen im Editor und das Scripting, also das Programmieren von Logik für die Anwendung. Damit soll ein Einblick in den Prozess der Entwicklung von C.LABEL-VR gegeben werden. Die Beschreibung der Funktionen von C.LABEL-VR selbst und deren Konzepte sind im folgenden Kapitel 4 beschrieben.

4

C.LABEL-VR

Dieses Kapitel beschreibt die Virtual Reality Applikation *C.LABEL-VR* und ist somit der Hauptteil dieser Arbeit. Das Ziel dieser Applikation ist es die Annotierung von Punktwolken, wie sie beispielsweise in C.LABEL möglich ist, innerhalb einer dreidimensionalen Umgebung zu realisieren. Dazu müssen zunächst Datenformate eingelesen werden, die Informationen über Punktwolken enthalten. Anschließend müssen aus diesen Daten Punktwolken erzeugt werden. Der Vorgang des Imports wird in Kapitel 4.1 näher beschrieben, die Erzeugung der Punktwolken in 4.2. Um sich in der virtuellen Umgebung durch diese Wolken bewegen zu können, wurden diverse Möglichkeiten zur Navigation entwickelt. Auf die Funktion dieser Möglichkeiten und deren Auswirkungen auf das Befinden des Menschen (*Virtual Motion Sickness*) wird in Abschnitt 4.3 eingegangen.

Anschließend geht es um die Annotierung der generierten Punktwolken. Annotierung bedeutet in diesem Kontext, dass die einzelnen Punkte der Wolken mit bestimmten Klassifikationen versehen werden, welche der Art des Objektes entsprechen, dem die Punkte zugehören. Ist der Punkt beispielsweise Bestandteil eines Autos, so wird er mit der Klasse *Auto* versehen. Um diese Aufgabe erfüllen zu können wurden mehrere Arten der Annotierung entwickelt. Welche dies sind und wie sie realisiert wurden, wird in Abschnitt 4.4 gezeigt. Das letzte Kapitel 4.5 beschäftigt sich mit der Benutzerschnittstelle der Applikation. Dieses wird auch *User Interface* oder kurz *UI* genannt. Dabei wird hauptsächlich auf die Funktionen des Menüs eingegangen welches man während des Labelns aufrufen kann. Ebenfalls wird erläutert welche Herausforderungen es bei der Erstellung von UIs in der virtuellen Realität gibt und wie diese bewältigt wurden. Zur Veranschaulichung aller Funktionen von C.LABEL-VR ist in der Abbildung 4.1 der grobe Workflow in Form eines Diagramms abgebildet.

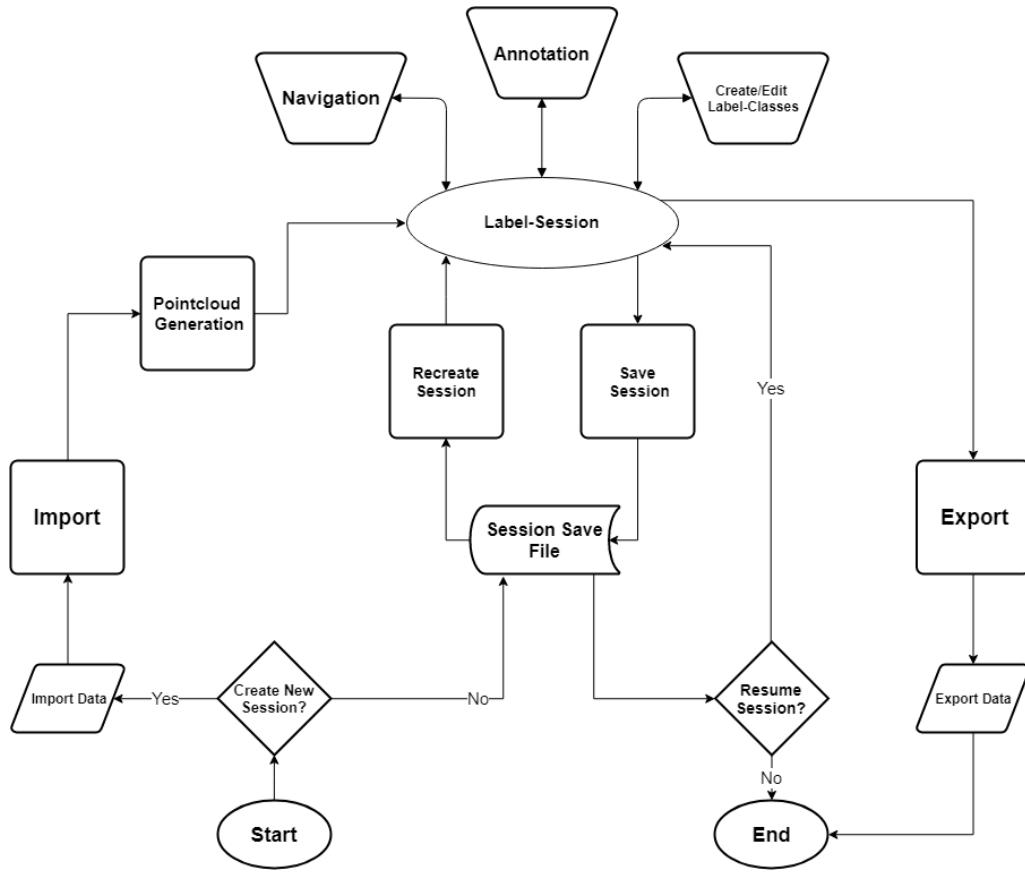


Abbildung 4.1.: in C.LABEL-VR vom Import bis zum Export

4.1. Import und Export von Daten

Umfeldmodelle, welche in Form einer Punktwolke vorliegen, werden meist von einem Radar-, oder, wie in Kapitel ?? gezeigt, von einem LiDAR-Sensor aufgenommen. Die Informationen über diese Modelle, zum Beispiel die Positionen der einzelnen Punkte, werden in einem passenden Datenformat abgespeichert. Um welches Format es sich dabei handelt kann sehr unterschiedlich sein. In der Regel hängt es davon ab zu welchem Zweck die aufgenommenen Daten gebraucht werden und wie umfangreich diese Daten sind. Am häufigsten werden sie dafür verwendet um Lernalgorithmen und neuronale Netze (siehe Abschnitt 2.1.1) zu trainieren. Automobilhersteller haben für solche Vorhaben natürlich unterschiedliche Konzepte und somit können auch die Formate der Daten unterschiedlich sein. Um die Applikation C.LABEL-VR für mehrere dieser Hersteller benutzbar zu machen, muss das Programm also mit verschiedenen Datenformaten umgehen können.

Innerhalb dieser Arbeit wurden mit zwei verschiedenen Arten von Daten gearbeitet. Das erste Format ist das PCD-Format (*Point Cloud Data*). Die Bibliothek PCL(*Point Cloud Library*) benutzt das Datenformat **PCD** um Informationen über Punktwolken zu verwalten. PCL ist eine sehr mächtige C++-Bibliothek, die in der Regel von jedem benutzt wird, der performante Algorithmen für Punktwolken entwickeln möchte. Auch C.LABEL generiert

aus eingelesenen Daten PCD-Files um Funktionen der Point Cloud Library für die Wolken zu benutzen.

Das zweite Format ist das, von der HDF-Group bereitgestellte, HDF5-Format (*Heterogeneous Data Format*). Es ist ausgelegt zum schnellen Erstellen und Auslesen von komplexen und großen Datenstrukturen. Welche Elemente diese Strukturen enthalten können wird später in Kapitel 4.1.3 erklärt. Zur schnellen Bearbeitung großer Datenmengen ist dies ein gängiges Format und wird von den meisten Kunden von CMORE benutzt um Sensordaten zu verwalten. Deshalb wird es auch in C.LABEL-VR verwendet.

4.1.1. Architektur

Wichtig ist es also sowohl den Import, als auch den Export so modular wie möglich zu gestalten, dass C.LABEL-VR ohne große Änderungen an der Hauptapplikation um neue Datenformate erweitert werden kann. Das Prinzip, das dafür entwickelt wurde, ist in Abbildung 4.2 dargestellt und wird im Folgenden näher erläutert. Die angesprochenen Erweiterungen um neue Datenformate werden als *Addons* bezeichnet. Jedoch handelt es sich dabei nicht immer nur um eine Erweiterung eines, von Grund auf, neuen Datenformats. Bei Komplexen Datenformaten wie HDF5 kann der Entwickler die interne Struktur der Datei selbst bestimmen. Jede unterschiedliche Struktur muss auf unterschiedliche Weise eingelesen werden und braucht somit unterschiedliche Addons. Um die Erweiterbarkeit der Applikation sicherzustellen wurde der Import und der Export von der Hauptapplikation abgekapselt.

Dafür wurde eine interne Datenstruktur (kurz IDS) eingeführt, welche alle nötigen Informationen enthält, die für das Labeling notwendig sind. Sie wird im späteren Kapitel 4.1.2 genauer vorgestellt. Der Vorteil dabei ist, dass die Hauptapplikation dadurch stets auf die gleiche Datenstruktur zurückgreifen kann. Sie bleibt somit unabhängig vom importierten Datenformat. Wird ein neues Import-Format eingeführt muss an der Hauptapplikation also nichts verändert werden. In der Abbildung 4.2 erkennt man, wie die Hauptapplikation (*C.LABEL-VR Session*) nur mit internen Daten arbeitet und somit von den Import- und Exportdaten getrennt ist.

Jedes Import-Addon hat also zunächst die Aufgabe, seine jeweiligen Daten einzulesen und alle Informationen, die für die IDS notwendig sind, daraus zu extrahieren. Aus diesen Informationen können anschließend die entsprechenden Punktwolken generiert werden. Die Generierung wird in Kapitel ?? erklärt. Beim Export ist es notwendig, dass die Struktur der Daten gleich bleibt, die vom Import-Addon eingelesenen wurden. Es sollen lediglich die Labeling-Informationen aus der IDS hinzugefügt werden. Aus der IDS kann das Export-Addon diese Informationen extrahieren und anschließend in die eingelesenen Daten exportieren. Die importierten Daten werden so allerdings überschrieben. Möchte

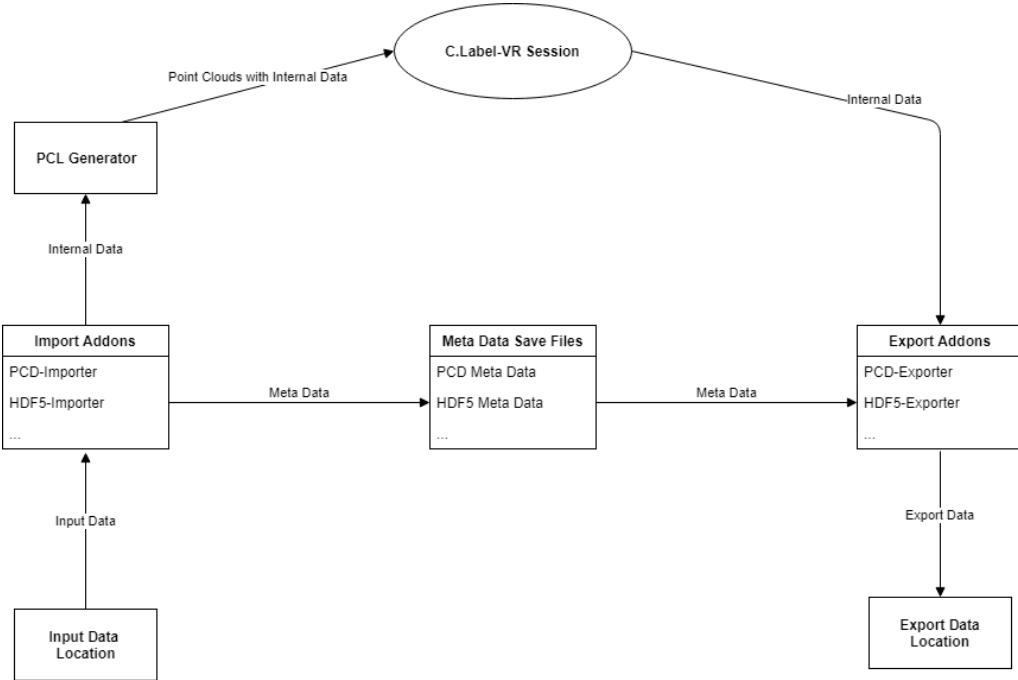


Abbildung 4.2.: Das Import-Export-Prinzip in C.LABEL-VR

man die Daten nicht überschreiben, müssen die Daten neu erstellt und mit den Labeling-Informationen versehen werden.

Die Importierten Daten, beispielsweise HDF5, enthalten allerdings oft deutlich mehr Informationen, als für die interne Struktur notwendig. Diese Zusatzinformationen werden *Metadaten* genannt. Für den Export bedeutet dies, dass man aus der internen Datenstruktur von C.LABEL-VR, die Struktur der anfangs eingelesenen Daten nicht wieder rekonstruieren kann, da die *Metadaten* fehlen würden. Um dieses Problem zu lösen müssen die zusätzlichen Daten separat gespeichert werden. Dazu wurde eine erweiterbare Datenstruktur angelegt. In dieser kann man für jede Import-Struktur eine Metadatenstruktur anlegen. Das entsprechende Export-Addon kann die Metadaten dann verwenden um, zusammen mit den internen Daten, das gewünschte Exportformat zu rekonstruieren.

Zusammenfassend ist also folgendes zu tun, um die Benutzung ein neues Datenformates oder einer neuen Datenstruktur zu gewährleisten: Zunächst muss eine Importfunktion programmiert werden, die alle Informationen aus den gewünschten Daten ausliest. Anschließend muss diese Funktion das genormte, interne Datenformat (Kapitel 4.1.2) erstellen und alle nötigen Informationen aus den eingelesenen Daten darin speichern. Daraus können die entsprechenden Punktwolken dann generiert werden können. Falls es Daten gibt, die über diejenigen in der IDS hinausgehen, muss eine Metadatenstruktur angelegt werden, in welche die zusätzlichen Daten abgelegt werden können. Zuletzt ist eine Exportfunktion zu implementieren, welche die Labeling-Informationen aus der internen Datenstruktur in die eingelesenen Daten exportieren kann. Außerdem muss sie in der Lage sein aus den

internen- und den Metadaten das Ausgangsdatenformat wiederherzustellen, um die Daten als neue Files exportieren zu können. Im folgenden Kapitel soll dieses Prinzip konkret an einem HDF5-Beispiel gezeigt werden.

4.1.2. Interne Datenstruktur

Wie im Kapitel 4.1 schon erwähnt, ist die interne Datenstruktur (*IDS*) dafür da, um den Prozess des Imports und Exports von der Hautapplikation zu trennen. Dazu müssen aus den Daten die eingelesen werden sollen, alle Informationen extrahiert werden die für die interne Datenstruktur notwendig sind. Die *IDS* wird repräsentiert durch ein Skript, dass einem Objekt in Unity angehängt werden kann, also einer Komponente (vgl. Abschnitt 3.3). Jedem Punkt einer Wolke in C.LABEL-VR ist solch ein Skript zugeordnet. Das heißt aus jeder Punktinformation aus den Input-Daten wird ein *IDS*-Komponente erstellt. Eine solche Komponente besteht aus folgenden Eigenschaften:

- **ID:** Identifikationsnummer zur eindeutigen Erkennung des Punktes
- **Label:** Information, zu welcher Klassifikation der Punkt gehört
- **Position:** Dreidimensionale Position des Punktes
- **Ground Point:** Information, ob der Punkt ein Bodenpunkt ist oder nicht

Dadurch kann die Möglichkeit genutzt werden, dass auf jede aktive Komponente eines Objekts in Unity zugegriffen werden kann. Wenn man nun beispielsweise einen Punkt identifiziert, den man neu klassifizieren möchte, kann auf die *IDS*-Komponente dieses Objekts zugegriffen werden und das darin enthaltene Label geändert werden.

Aus den Input-Daten wird meist nur die Position des Punktes ausgelesen und manchmal auch die Klassifikation des Punktes, falls diese schon vorhanden ist. Die ID wird innerhalb vom Import vergeben und dient dazu den Punkt eindeutig zu identifizieren. Werden beispielsweise Informationen eines Punktes als Metadaten abgespeichert, müssen diese später dem richtigen Punkt zugeordnet werden können. Dies geschieht über die eindeutige ID. Genauer erklärt wird dieser Vorgang im Abschnitt 4.1.3. Die Information, ob es sich um einen Bodenpunkt handelt oder nicht, wird nach dem Import von einem Algorithmus analysiert. Sie ist wichtig für die Cluster-Annotation, die später in Kapitel 4.4.3 vorgestellt wird. Wie die Bodenpunkt-Analyse funktioniert wird in Abschnitt 4.4.3 erklärt.

Da die interne Datenstruktur nicht viele Informationen enthält ist es eine simple Methode die Punktwolken-Daten in C.LABEL-VR zu verwalten. Alles was darüber hinaus geht kann in einer separaten Meta-Datei gespeichert werden, um die Zusatzinformationen beim Export zur Verfügung zu haben. Darüber hinaus ist sie leicht erweiterbar. Im Skript der

IDS-Komponente müsste lediglich eine neue Eigenschaft angelegt werden, um jedem Punkt einer eingelesenen Punktwolke die neue Eigenschaft zuzuordnen. Das genaue Vorgehen bei der Bearbeitung der internen Datenstruktur und der Metadaten ist im folgenden Kapitel 4.1.3 näher erklärt.

4.1.3. HDF5 Beispiel

HDF5 Daten

TODO Evtl. den Import und Exportvorgang anhand eines minimalistischen HDF5-Beispiel erklären

4.2. Generierung einer Punktwolke

TODO Kurze einleitung

Nach dem man die Informationen über die internen Datenstrukturen durch den Import gesammelt hat, kann man mit dem Beginn der Generierung der Punktwolken anfangen. Ziel davon ist es die Punktwolkendaten zu visualisieren, damit der Benutzer sie sehen und mit ihnen interagieren kann. In der Regel wird dafür, in einer passenden Visualisierungs-umgebung, für jede Position eines Punktes ein Objekt eingezeichnet. Meistens sind dies zweidimensionale Punkte, die lediglich eine quadratische Einfärbung von Pixeln sind.

-----Bild Punktwolke normal und VR-----

In C.Label-VR wird jeder Punkt durch eine dreidimensionale Kugel repräsentiert. Dadurch wird für den Benutzer der Eindruck erzeugt, dass die Sensordaten durch echte Objekte im Raum repräsentiert werden, die gut sichtbar sind und greifbar für den Benutzer wirken. Dies ist ein großer Unterschied zu bisherigen Visualisierungen und bietet einen erheblichen Mehrwert. Die Unity Engine als Visualisierungsumgebung bietet nämlich die richtige Darstellung der räumlichen Proportionen an jeder Position. Diese Proportionen, also auch die Darstellung der Sensordaten im richtigen Maßstab, werden durch das Rendering der Unity Engine für jedes Bild berechnet (erklärt in Abschnitt 3.3) und somit immer richtig für den Benutzer dargestellt.

Die dreidimensionalen Punkte sind dabei vorgefertigte Objekte. Dieses Vorfertigen von Objekten bietet sich bei der Entwicklung mit Unity immer dann an, wenn mehrere gleiche Objekte erstellt werden sollen, wie es beispielsweise bei einer Punktwolke der Fall ist. Solche Objekte, die in Unity *Prefabs* genannt werden, zu erstellen ist eine herkömmliche Methode bei der Entwicklung mit Unity [35].

Das Punkt-Prefab für die Generierung der Punktwolke besteht dabei aus mehreren Komponenten. Eine Komponente ist natürlich die Transform-Komponente (siehe Abschnitt

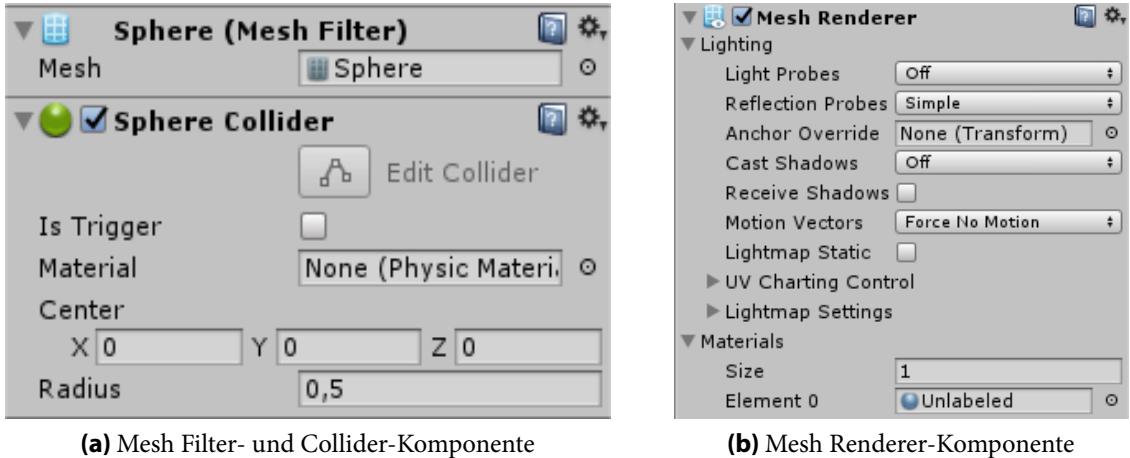


Abbildung 4.3.: Beide Bilder zeigen Komponenten aus der Unity Editor, die dem Punkt-Prefab zugeordnet sind.

3.3), die die Position des Punktes festlegt. Eine weitere ist die *Mesh Renderer*-Komponente. Ein Mesh (zu deutsch: Gewebe/Netz) ist in der Spielentwicklung ein gängiger Begriff für das dreidimensionale Modell eines Objektes. Für das Punkt-Prefab wird das Modell einer Kugel verwendet (*Sphere* als Mesh in Abbildung 4.3a). Der Mesh-Renderer legt dabei die Materialeigenschaften des Objektes fest [36]. Diese Eigenschaften enthalten beispielsweise die Textur, also das Aussehen der Oberfläche des Objektes, die Reflexionseigenschaften oder wie die Art wie das Objekt Lichteinwirkungen auf die Umgebung verteilt. Das Standard-Material das beim Punkt-Prefab verwendet wird ist ein vorgefertigtes Material *Unlabeled* (siehe Abbildung 4.3b), welches eine einfarbige, schwarze Oberfläche hat. Schwarze Punkte repräsentieren also diejenigen Punkte, die noch nicht annotiert wurden. Andere Eigenschaften wie Licht, Reflexionen oder Schatten wurden mit Parametern belegt, die so wenig wie möglich Berechnungen erfordern, damit die Bildwiederholungsrate der Applikation so hoch wie möglich gehalten werden kann. Genauer ist dies im folgenden Abschnitt 4.2.1 erklärt.

Eine weitere wichtige Komponente ist der *Sphere Collider* (siehe Abbildung 4.3a). Sie ist notwendig um mit dem Punkt zu interagieren, da der Sphere Collider die Kollisionseigenschaften des Objektes festlegt. Möchte man mit dem Punkt beispielsweise mit einem virtuellen Gegenstand, wie einer virtuellen Hand interagieren, muss die Kollision zwischen Punkt und Hand detektiert werden. Das geschieht durch die Collider-Komponenten beider Objekte. Für die Leistungsfähigkeit von C.LABEL-VR ist es ebenfalls wichtig, dass die *Is Trigger*-Eigenschaft des Punkt-Colliders deaktiviert ist. Im aktiven Zustand besagt diese Eigenschaft, dass das Objekt welches den Collider hat, der Auslöser für Kollisionen ist. Das heißt die Collider-Komponente überprüft durchgehend ob eine Kollision vorliegt. Wenn alle Punkte der Wolke diese Überprüfung durchgehen machen würden, würde sich das negativ auf die Leistung der Applikation auswirken.

Zuletzt enthält das Punkt-Prefab doch die Komponente der internen Datenstruktur (siehe Abschnitt 4.1.2). Diese enthält die Informationen die man bei der Interaktion mit dem Punkt wissen muss, wie das Label und die Position. Die Label-Eigenschaft der IDS-Komponente wurde so implementiert, dass sie mit der Mesh-Renderer Komponente verknüpft ist. Dies dient dem Zweck, dass sich bei Änderung des Labels automatisch die Material-Farbe des Punktes ändert. Eine ähnliche Verknüpfung gibt es bei der Positionseigenschaft der internen Datenstruktur. Diese ist mit der Transform-Komponente verknüpft, sodass bei Änderung der Positionseigenschaft auch wirklich die Position geändert wird, da dies nur mit der Transform-Komponente möglich ist. Beide Implementationen sind in der Auflistung ?? zu sehen.

Auflistung 4.1: Vereinfachte Implementierung des Beschleunigungs- und Verzögerungsprinzip in eine einzelne Richtung

```
private uint _Label;
public uint Label
{
    get
    {
        return _Label;
    }
    set
    {
        _Label = value;
        GetComponent<MeshRenderer>().material =
            Labeling.GetLabelClassMaterial(_Label);
    }
}

private Vector3 _PointPosition;
public Vector3 PointPosition
{
    get
    {
        return transform.position;
    }
    set
    {
        transform.position = value;
    }
}
```

—Evtl. Koordinatensystem anpassung—————

Das Punkt-Prefab, mit all seinen Komponenten und deren richtig festgelegten Eigenschaften, kann nun für jeden Punkt aus den Input-Daten instantiiert werden. Dafür gibt es in Unity die spezielle Funktion *Instantiate* [37]. Die Unity Engine klonnt dadurch für jede Instanz das vorgefertigte Ausgangsobjekt, wobei jedes der Klone nach der Instantiierung individuell veränderbar ist. Nachdem die Instanz eines Punktes erstellt wurde kann also seine Position festgelegt werden und, wenn beim Import schon vorhanden, das Label des Punktes gesetzt werden. Durch das Vorgehen, das in diesem Abschnitt beschrieben wurde entsteht eine Punktwolke, welche in einer virtuellen Umgebung betrachtet werden kann.

————— Abbildung Punktwolke mit pre labels ————

4.2.1. Optimierung

Die Bildwiederholungsrate einer Applikation ist wesentlich entscheidend für die Benutzererfahrung mit ihr. Geringe Wiederholungsraten führen zu einer nicht flüssigen Darstellung des Applikationsinhaltes, was sehr störend für den Benutzer sein kann. Gerade bei VR-Applikationen sind konstant hohe Bildwiederholungsraten wichtig um dem Benutzer eine angenehme visuelle Präsentation zu bieten und keine Symptome der VR-Krankheit hervorzurufen. Nähere Informationen zur VR-Krankheit werden im folgenden Abschnitt 4.3.1 gegeben. In C.LABEL-VR werden keine komplexen 3D-Objekte oder hochauflösende Texturen dargestellt, deswegen könnte hier der Eindruck entstehen dass die Applikation, bei einem Leistungsstarken Rechner wie in Abschnitt 3.2, keine Performance-Probleme haben sollte. Da aber Punktwolkendaten oft über 30.000 Punkte enthalten können, ist die große Anzahl an Objekten selbst, die dargestellt werden müssen, auch für Leistungsstarke Rechenmaschinen eine Herausforderung. Deswegen muss bei der Erstellung der Punktwolken auf einige Dinge geachtet werden, um keine Einbrüche bei der Bildwiederholungsrate zu bekommen.

Im Abschnitt 3.3 wurde die Update-Funktion erwähnt, die bei jedem Bildwiederholungszyklus in Unity ausgeführt wird. Durch diese hohe Ausführungsrate ist es generell ratsam, so wenig wie möglich Leistungsintensiven Programm-Code in diesen Methoden zu verwenden, da dieser ständig ausgeführt wird. Bei der Erstellung von Punktwolken in C.LABEL-VR ist es außerdem wichtig, dass das Punkt-Prefab über keine dieser Update Methoden verfügt. Diese würde nämlich von allen Punkten durchgehen ausgeführt werden, was bei einer hohen Anzahl an Punkten zu einem großen Leistungsverlust führen würde.

Wichtig bei der Erstellung des vorgefertigten Punktes ist es auch, die Materialeigenschaften richtig zu wählen. Das Material selbst, das dem Punkt-Prefab zugeordnet ist, ist ebenfalls vorgefertigt (Material *Unlabeled* in Abbildung 4.3b). Der Unity Engine ist somit bekannt, dass jeder Punkt das gleiche Material hat, das dargestellt werden muss. Die Engine muss somit nicht, für jeden Punkt, eine neue Information über das Material des nächsten Punktes

anfordern. Vergleicht man das Anzeigen einer Punktes mit dem Bohren eines Loches, so kann der Bohrer für jedes neue Loch den gleichen Bohrkopf verwenden und müsste diesen nie Wechseln. Das spart Zeit und schlägt sich in C.LABEL-VR durch weniger Berechnungszeit und damit eine höhere Bildwiederholungsrate nieder.

Bei den Eigenschaften der Mesh Renderer-Komponente (siehe Abbildung 4.3b) kann ebenfalls darauf geachtet werden, keine unnötigen Berechnungen zuzulassen. Die Optionen *Light Probes* und *Lightmap Static* sind, grob gesagt, für die Lichtverteilung des Objektes in der Szene zuständig [38]. Da eine Lichtverteilung des Objektes keinen optischen Mehrwert bringt, da der Hintergrund weiß ist, können diese Optionen deaktiviert werden, um Rechenzeit zu sparen. Auch die Option *Cast Shadows* wird deaktiviert, da Schatten bei der Berechnung aufwendig und bei der Darstellung der Punktfolge nicht erwünscht sind. Lediglich die Eigenschaft *Reflection Probes* wird mit der Option *Simple* belegt. Diese Eigenschaft legt die Reflexion, die vom Objekt zurückgegeben wird fest. Solch eine Reflexion ist in C.LABEL-VR gewünscht, da dies die Punkte für den Benutzer räumlicher darstellt, da er Reflexionen wie ein echtes Objekt hat. Damit die Berechnungen dafür aber nicht zu aufwendig werden, wird hier nur die Option Simple gewählt.

Auch bei der Sphere Collider-Komponente ist auf eine Eigenschaft zu achten. Wie schon in Abschnitt 4.2 erwähnt, ist diese Komponente für die Interaktion mit dem Objekt zuständig. Durch diese Komponente kann nämlich detektiert werden, ob ein anderes Objekt mit einer Collider-Komponente mit diesem kollidiert [39]. Eine der beiden Collider-Komponenten muss dabei der Auslöser der Kollision sein. Um dies zu gewährleisten kann die Eigenschaft *Is Trigger* aktiviert werden. Ist diese im aktiven Zustand, so überprüft die Collider-Komponente durchgehend, ob eine Kollision mit einem anderen Objekt vorliegt. Ist dies bei jedem Punkt in der Punktfolge der Fall, gibt es durchgehend von allen Punkten diese Überprüfung, was zu einem erheblichen Performance-Verlust führt. Möchte man also durch ein Objekt mit einem Punkt interagieren, muss das Objekt der Auslöser der Kollision sein, da somit nur ein Objekt in der Szene eine Kollisionsprüfung vollzieht.

Trotz all dieser Einstellungen zur Verbesserung der Anwendungs-Performance kann es bei zu vielen Punkten im Sichtfeld des Benutzers zu einer niedrigen Bildwiederholungsrate kommen. Der Grund dafür ist, dass für jedes Objekt, das auf den Bildschirm eingezeichnet werden muss, ein Zeichnungs-Befehl an die Grafikkomponente des Rechners geschickt werden muss. Dies wird in der Fachsprache *Draw Call* genannt [40]. Bei einer zu hohen Anzahl an Punkten in einer Punktfolge, kann es somit zu einer zu hohen Anzahl an Zeichnungs-Befehlen kommen, was zu einer niedrigen Bildwiederholungsrate führt. In diesem Fall hilft die Verbesserung der Leistungskomponenten des VR-Rechners (vgl. Abschnitt 3.2). Für C.LABEL-VR wurde als zusätzliche Leistungsoptimierung eine Funktion implementiert,

die dieses Problem umgeht. Diese Funktion verringert die Anzahl der angezeigten Punkte während der Bewegung des Benutzers, um die Anzahl an Draw-Calls zu verringern und somit die Bildwiederholungsrate zu erhöhen. Diese Funktion wird näher im Abschnitt 4.5.1 erklärt.

4.3. Navigation

Die Navigation in Virtual Reality-Anwendungen ist ein entscheidendes Thema. Wird dieses Thema während der Entwicklung einer VR-Applikation nicht ausreichend berücksichtigt, so kann dies große Auswirkung auf die Benutzererfahrung haben [41]. Ein Grund dafür ist die *VR-Krankheit*, die häufig auftritt wenn sich Benutzer durch eine virtuelle Umgebung bewegen [42]. Diese Krankheit verursacht unter anderem Übelkeit und tritt bei manchen Nutzern mehr und bei manchen weniger auf. Details dazu sind im Abschnitt 4.3.1 gegeben.

Um sich in C.LABEL durch Punktewolken zu bewegen wurden zwei Navigationsmöglichkeiten entwickelt. Zum einen gibt es den Freien Flug Modus (Abschnitt 4.3.2). In diesem Modus kann sich der Benutzer, mit Hilfe der Oculus Eingabegeräte intuitiv und frei durch die Punktewolke bewegen. Die Bewegung ist dabei durchgängig und wird durch Beschleunigung und Bremsen gestartet und beendet. Diese Art der Bewegung fördert jedoch bei manchen Nutzen den Effekt der VR-Krankheit.

Deshalb gibt es mit dem Teleportationsmodus (*Teleport Mode*) noch eine alternative Navigationsart (4.3.3). Das Grundprinzip der Steuerung ist ähnlich, allerdings gibt es keine durchgängige Bewegung. Stattdessen wird der Benutzer von einer Stelle zur anderen teleportiert. Dies wirkt dem Übelkeitseffekt entgegen.

4.3.1. VR-Krankheit

Die VR-Krankheit ist ein Effekt der auftritt während oder nach dem Aufenthalt einer Person in einer virtuellen Umgebung. Sie äußert sich meistens durch Übelkeit, aber auch durch andere Symptome wie Kopfschmerzen oder Überanstrengung der Augen [42]. Es wird geschätzt, dass bei dreißig bis achtzig Prozent der Menschen Symptome der VR-Krankheit auftreten wenn sie sich in einer virtuellen Umgebung befinden [43]. Die Erforschung der Ursache für diese Krankheit ist noch nicht gänzlich abgeschlossen. Somit gibt es mehrere Theorien was der Auslöser für sie ist.

Die gängigste Theorie dafür ist das Prinzip der Sinnestäuschung. Dieses Prinzip besagt, dass das menschliche Gehirn einen Konflikt feststellt, wenn sich die erwartete Wahrnehmung zweier Sinnesorgane voneinander unterscheiden. Im Falle der VR-Krankheit handelt

es sich bei den Sinnesorganen um die visuelle Wahrnehmung und den Gleichgewichtssinn. Bewegt sich ein VR-Nutzer nun durch die virtuelle Welt, so registriert dies die visuelle Wahrnehmung als Bewegung in eine bestimmte Richtung. Solche Bewegungen in der virtuellen Welt geschehen oft ohne Bewegung des Benutzers in der physischen Welt. Der Gleichgewichtssinn des Menschen registriert also keinerlei Bewegung im Gegensatz zur visuellen Wahrnehmung und somit kommt es zum Konflikt. [44].

Dieser Konflikt kann jedoch nicht nur ausgelöst werden, wenn sich die Bewegung des Nutzers in der virtuellen und der realen Welt grundlegend unterscheiden. Symptome der VR-Krankheit können auch dann auftreten, wenn die Bewegungen gleich sind, jedoch leicht verzögert. Dies kann beispielsweise der Fall sein, wenn die Leistungsgrenze des betreibenden Rechners erreicht ist und somit keine flüssige Wiederholungsrate des Bildes mehr gewährleistet werden kann. Ist die Wiederholungsrate zu niedrig entsteht ein zu großer Zeitunterschied zwischen der Tätigkeit der Bewegung und dem Sehen der Bewegung. Lilienthal hat in einer frühen Forschung empfohlen, einen maximalen Zeitunterschied von 35 Millisekunden zwischen den Wahrnehmungen der verschiedenen Sinne einzuhalten[45]. Das bedeutet das System muss mindestens eine Bildwiederholungsrate von ungefähr 30 Bildern pro Sekunde gewährleisten. Aktuell empfehlen Hersteller von VR-Brillen sogar eine Wiederholungsrate von 90 Bildern pro Sekunde, um einen komfortablen Gebrauch des VR-Gerätes sicherzustellen [46]. Um der VR-Krankheit vorzubeugen ist also auch die Optimierung der Applikation (wie in Abschnitt 4.2.1) für eine hohe Bildwiederholungsrate und Leistungsfähigkeit vom betreibenden Rechner des VR-Gerätes zu beachten.

Der häufigste Grund für das Auslösen der VR-Krankheit sind aber, wie Anfangs erwähnt, zu starke Bewegungen in der virtuellen Welt bei fehlenden Bewegungen in der realen Welt. Deshalb wurde in C.LABEL-VR dieser Aspekt bei den Navigationsmodi beachtet und versucht durch geeignete Methoden der VR-Krankheit vorzubeugen.

4.3.2. Freier Flug-Modus

Der freie Flug-Modus ist die erste Navigationsart, die für C.LABEL-VR entwickelt wurde. Hierbei wurde vor allem auf den praktischen Nutzen für die Bewegung wert gelegt. Das Ziel war es also einen Bewegungsmodus zu implementieren, der für den Benutzer leicht verständlich und schnell zu erlernen ist. Außerdem sollte es möglich sein sich präzise zu Bewegen um leicht an gewünschte Positionen in der Punktwolke zu kommen.

Die Benutzereingabe für diesen Navigationsmodus erfolgt über die Analog-Sticks der Oculus Touch Controller (siehe *Thumbstick* in Abbildung 4.4). Bewegt der Benutzer einen dieser Sticks, bewegt er sich auch in der virtuellen Welt. Wird der Stick wieder losgelassen, endet auch die Bewegung innerhalb der virtuellen Umgebung. Dies soll die gewünschte

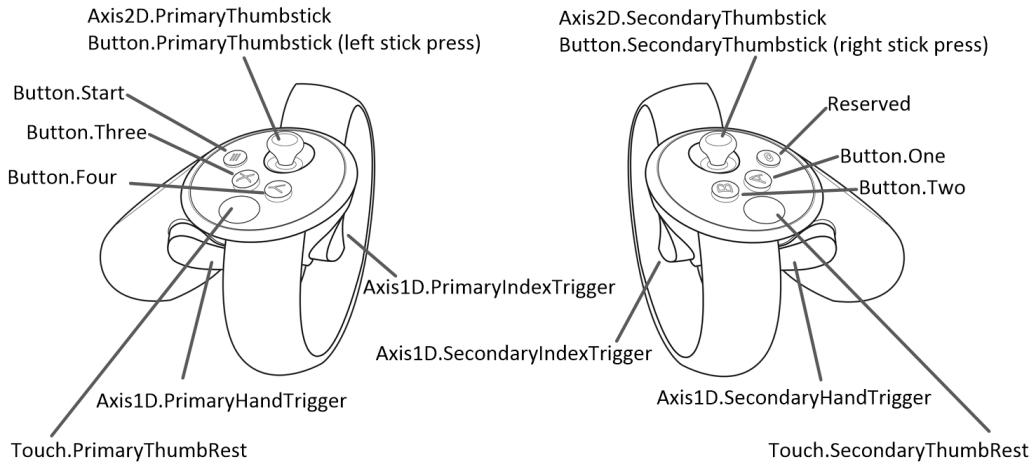


Abbildung 4.4.: Oculus Touch Controller mit der offiziellen Beschriftung von Oculus [4]

Intuitivität liefern. Die Längs- und die Lateral-Bewegung werden dabei mit dem linken Stick gesteuert. Durch Betätigung des rechten Sticks kann eine Vertikal-Bewegung oder eine Rotation um die eigene Achse ausgeführt werden. Die genauen Steueranweisungen sind in der Bedienungsanleitung zur Applikation zu finden (APPENDIX).

Um die Bewegungen aus dem Stand oder Richtungswechsel angenehmer für den VR-Benutzer zu machen wird im freien Flugmodus Beschleunigung und Verzögerung simuliert. Wird ein Steuerbefehl vom Benutzer gegeben, bewegt sich dieser nicht sofort mit voller Geschwindigkeit. Dies wäre ein, für den Menschen unnatürliches Bewegungsverhalten und könnte, zusätzlich zu den Auslösern aus Abschnitt 4.3.1, zu Symptomen der VR-Krankheit führen. Stattdessen kommt die Bewegung, durch die Beschleunigung, langsam ins Rollen, sodass der Benutzer eine bestimmte Zeit hat um sich an die Geschwindigkeitsveränderung zu gewöhnen. Endet der Bewegungsbefehl, also lässt der Benutzer den entsprechenden Stick los, endet die Bewegung nicht schlagartig, sondern die Geschwindigkeit wird wie beim Bremsen mit einem Auto immer weniger, sodass auch hier der Benutzer Zeit hat sich an den Bewegungsstopp zu gewöhnen. Besonders wichtig wird dieses Prinzip bei Richtungswechseln. Würde der Benutzer bei voller Geschwindigkeit einen Richtungswechsel vollziehen, bei dem er sich in die neue Richtung ebenfalls sofort mit Höchstgeschwindigkeit bewegt, würde dies in den meisten Fällen zu Übelkeit führen, da der Richtungswechsel zu schlagartig vollzogen werden würde. Durch die Verzögerung kommt, bei einem Richtungswechsel, der Benutzer erst langsam zum Stehen und wird danach in die neue Richtung beschleunigt. Der Wechsel der Bewegungsrichtung wird also übergangsweise ausgeführt, was der VR-Krankheit entgegenwirkt.

Die Geschwindigkeit der Bewegung des Nutzers wird in jedem Update-Zyklus (erklärt in Abschnitt 3.3) der Applikation berechnet. Das heißt die Funktion *ComputeSpeed* aus der Auflistung 4.2 wird in jedem dieser Zyklen aufgerufen. Die Beschleunigung wird nun so si-

muliert, dass ein bestimmter Beschleunigungswert auf den aktuellen Geschwindigkeitswert addiert wird, bis die Maximalgeschwindigkeit erreicht ist. Gibt es keinen Bewegungsbefehl mehr, wird vom aktuellen Geschwindigkeitswert solange ein Verzögerungswert abgezogen, bis die Geschwindigkeit null beträgt. Der Verzögerungswert ist in der Regel höher als der Beschleunigungswert, damit der Benutzer schneller zum Stehen kommt, als er Beschleunigt. Dies hat den Zweck dass der Anhaltevorgang präzise bleibt und der Benutzer so nicht zu weit verzögert bis er zum stehen kommt. Beide Werte können allerdings vom Benutzer selbst eingestellt werden (siehe Kapitel 4.5.1). Der Richtungswechsel ist vom Prinzip her ähnlich implementiert. Hierbei wird allerdings ein noch höherer Subtrahend als der Verzögerungswert von der aktuellen Geschwindigkeit abgezogen, damit der Richtungswechsel schnell von statten geht, jedoch immer noch verzögert.

Auflistung 4.2: Vereinfachte Implementierung des Beschleunigungs- und Verzögerungsprinzip in eine einzelne Richtung

```

private float ComputeSpeed(bool move, float currentSpeed, float
    maxSpeed, float accelerationValue, float decelerationValue)
{
    float newSpeed;

    if(move == true)
    {
        if(currentSpeed < maxSpeed)
            newSpeed = currentSpeed + accelerationValue;
        else
            newSpeed = maxSpeed;
    }
    else
    {
        if(currentSpeed > 0)
            newSpeed = currentSpeed - decelerationValue;
        else
            newSpeed = 0;
    }

    return newSpeed;
}

```

Durch den freien Flugmodus soll dem Benutzer eine Bewegungsart zur Verfügung gestellt werden, die den gewohnten Bewegungsabläufen des Menschen ähnelt. Dies ist mit der Simulation von Beschleunigung und Verzögerung gegeben. Durch einen höheren Verzögerungswert kann darüber hinaus sichergestellt werden, dass der Benutzer präzise an eine gewünschte Position navigieren kann. Nichts desto trotz kann es bei diesem Bewegungsmo-

dus dazu kommen, dass Symptome der VR-Krankheit auftreten. Dies kommt daher, dass der Mensch eine konstante Bewegung visuell wahrnimmt, sie aber nicht körperlich fühlt (vgl. Abschnitt 4.3.1). Um diesen Nachteil auszugleichen wurde zusätzlich noch ein zweiter Navigationsmodus entwickelt.

4.3.3. Teleport-Modus

Semaphor mit GetDown austauschen

Der Teleport-Modus ist der zweite Navigationsmodus der für C.LABEL-VR entwickelt wurde. Der Zweck dieses Modus ist es, die Anfälligkeit des freien Flug-Modus (Abschnitt 4.3.2) für die VR-Krankheit auszugleichen. Symptome der VR-Krankheit treten beim freien Flug auf, da der Benutzer eine konstante Bewegung auf visuellem Weg wahrnimmt, jedoch nicht mit dem Gleichgewichtssinn spürt. Durch Teleportation, also den unverzögerten Sprung von einer Position zu einer anderen, gibt es keine konstante Bewegung. So soll nun der VR-Krankheit entgegen gewirkt werden. Diese Art der Bewegung ist aktuell die am weitesten verbreitet unter VR-Anwendungen [47], da sie nachweislich weniger anfällig für die VR-Krankheit ist [41].

Das Prinzip der Steuerung ist ähnlich wie das des freien Flug-Modus. Steuerungsbefehle kann der Benutzer zunächst mit den Sticks tätigen (siehe *Thumbstick* in Abbildung 4.4). Auch hier wird die Längs- und die Lateral-Bewegung mit dem linken und die Vertikal- und Rotations-Bewegung mit dem Rechten Stick gesteuert. Die genauen Steueranweisungen sind in der Bedienungsanleitung zur Applikation zu finden (APPENDIX).

Wie schon erwähnt gibt es, im Gegensatz zum freien Flug-Modus, keine konstante Bewegung durch die Benutzereingaben. Stattdessen wird die Position des Benutzers in der virtuellen Welt durch die Steuerbefehle unverzögert verschoben, sodass es einem Teleport gleicht. Die Implementation dieses Vorgangs ist dank der Transform-Komponente eines Gameobjects (siehe Abschnitt 3.3) nicht aufwändig. Wie in der Auflistung 4.3 zu sehen ist, kann die Verschiebung des Objekts *userView* um einen Vektor mit einem einzigen Befehl vollzogen werden. Dieses Objekt repräsentiert dabei die Augen des Benutzers in der virtuellen Welt. Wird es also verschoben, ist es für den Benutzer so, als würde er von einer zur anderen Position teleportiert werden. Der Vektor um den der Benutzer verschoben wird, kann von ihm selber eingestellt werden (siehe Kapitel 4.5.1).

Auflistung 4.3: Implementierung des Teleportationsbefehls

```
GameObject userView;
```

```
userView.transform.Translate(new Vector3(xDirection, yDirection,  
zDirection));
```

-----Bild mit pointer teleport einfügen-----

Zusätzlich zu den Benutzereingaben durch den Stick, gibt es eine weitere Möglichkeit für den Benutzer sich an eine bestimmte Stelle in der virtuellen Welt zu teleportieren. Dazu kann er eine Art Laserstrahl aktivieren, der vom linken virtuellen Controller ausgeht (Abbildung ??). Dieser Strahl fungiert wie eine Stange die der Benutzer in der Hand hält, sie bewegt sich also mit der Handbewegung des Nutzers mit. Der Strahl kann durch die Hand-Auslöser-Taste des linken Controllers aktiviert werden (siehe *PrimaryHandTrigger* in Abbildung 4.4). Dabei wird auch die Länge des Strahls angezeigt, was ebenfalls in der Abbildung ?? zu sehen ist. Die Länge des Strahls kann anschließend vom Benutzer angepasst werden, indem er den linken Stick nach vorne (Strahl wird länger) oder nach hinten (Strahl wird kürzer) drückt. Betätigt der Benutzer die Index-Auslöser-Taste (*PrimaryIndexTrigger* in Abbildung 4.4) wird er an die Spitze des Laserstrahls teleportiert. Mit Hilfe dieser Teleportationsart kann der Benutzer auch weite Distanzen in der virtuellen Welt schnell zurücklegen. Für die Implementation dieser Funktion kann ebenfalls der Befehl aus der Auflistung 4.3 verwendet werden, wobei statt dem neu erzeugten Vektor die Position des Strahlendes verwendet wird.

Durch die Bewegung mittels Teleport wird also der VR-Krankheit entgegengewirkt, da es keine konstante Bewegung für den Benutzer gibt. Durch den Teleport mittels Laserstrahl können hierbei auch große Distanzen vom Benutzer zurückgelegt werden. Der Nachteil dieses Bewegungsmodus ist es, dass durch feste Teleportationsdistanzen keine präzise Steuerung mehr garantiert werden kann. Da aber genau dafür der freie Flug-Modus Abhilfe schafft, ergänzen sich die beiden Navigationsmodi sehr gut und bieten dem Benutzer so passende Möglichkeiten sich durch die virtuelle Welt zu Bewegen.

4.4. Annotieren der Punktfolke

TODO kurze Einleitung

4.4.1. Pointer Annotation

TODO Erklärung der Pointer Annotation und des Funktionsprinzips dahinter

4.4.2. Touch Annotation

TODO Erklärung der Pointer Annotation und des Funktionsprinzips dahinter

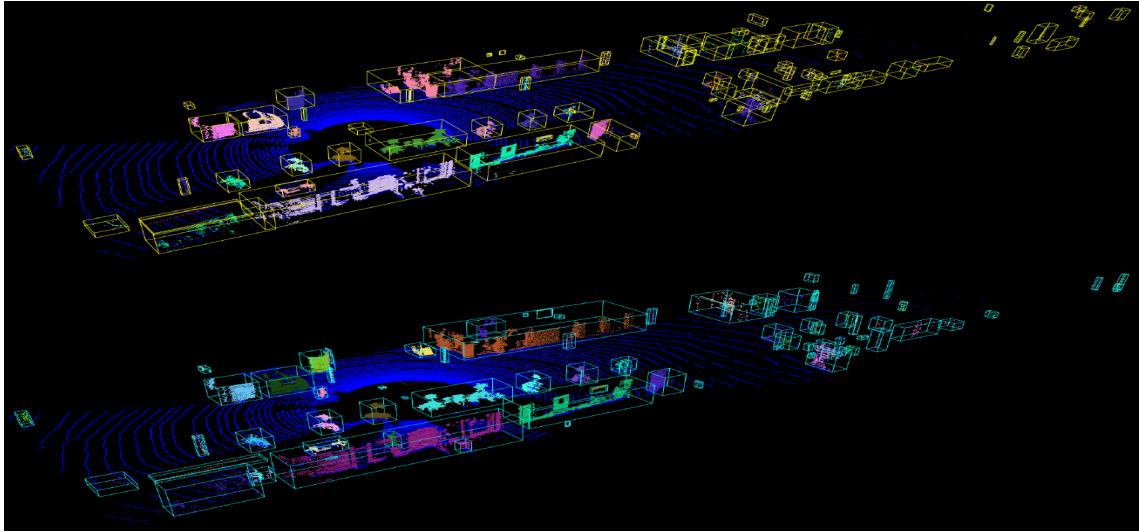


Abbildung 4.5.: Objektidentifizierung einer 3D-Punktwolke aus [5]. Die Bodenpunkte der Wolke sind blau dargestellt und die Objekte sind mit Boxen gekennzeichnet.

4.4.3. Cluster Annotation

In einer Punktwolke mit mehreren tausend Punkten wird es für den Benutzer auf Dauer mühsam jeden Punkt einzeln zu Annotieren. Bei diesem Problem soll die *Cluster Annotation* Abhilfe schaffen. Diese Art der Annotation soll selbstständig diejenigen Punkte in einen Cluster aufnehmen und sie mit der gleichen Klasse labeln, die zum gleichen Objekt gehören. Der Benutzer wählt hierfür, wie im vorherigen Kapitel 4.4.1 einen Punkt mit Hilfe des Pointers aus. Ausgehend davon werden umliegende Punkte untersucht, ob sie zum gleichen Objekt gehören wie der ausgewählte Startpunkt. Ist dies der Fall werden sie in den Cluster aufgenommen. Werden keine Punkte mehr aufgenommen bleibt der Endcluster übrig, welcher dann einem Objekt entspricht, zum Beispiel einem Auto.

Wissenschaftler beschäftigen sich intensiv mit solchen Verfahren zur Objektidentifizierung in dreidimensionalen Punktwolken. Ansätze, wie sie in [5], [48] oder [49] vorgestellt werden sind nur wenige von vielen. Die meisten haben aber ein ähnliches Vorgehen. Zuerst werden all diejenigen Punkte in der Wolke entfernt, die zum Boden gehören. So bleiben anschließend nur noch Punkte übrig, welche zu Objekten wie beispielsweise Autos, Häuser oder Straßenschilder gehören. Die übrigen Punkte können nun mit diversen Algorithmen zu Clustern zusammengefasst werden. Ein Beispiel für das Ergebnis eines solchen Verfahrens bietet die Abbildung 4.5. Im Folgenden wird vorgestellt wie die Cluster-Analyse in C.LABEL-VR funktioniert.

Bodenpunkt-Analyse

Wie schon erwähnt werden bei vielen Verfahren der Cluster-Analyse von Punktwolken zuerst die Bodenpunkte identifiziert. Dieses Verfahren wird auch in dieser Arbeit angewendet.

Der Grund dafür ist, dass die Bodenpunkte nicht in das Suchverfahren nach Objektpunkten aufgenommen werden dürfen. Sie würden nicht nur die Suche nach echten Objektpunkten erschweren sondern auch das Ergebnis der Objektcluster verfälschen. Als Erklärung soll folgender, simpler Cluster-Algorithmus dienen:

1. Definiere einen Startpunkt p .
2. Ausgehend von p , suche alle Punkte innerhalb von Radius x .
3. Jeder gefundene Punkt ist ein neuer Startpunkt p .
4. Führe 2. aus bis es keine neuen Punkte p mehr gibt.

Alle Objekte die auf dem Boden stehen, also deren Punkte nahe an Bodenpunkten sind, würden mit dem obigen Algorithmus die Bodenpunkte mit in den Cluster aufnehmen. Dies kann sogar dazu führen, dass 2 Objekte, die mittels Bodenpunkten miteinander verbunden sind zu einem Cluster zusammengefügt werden. Darum müssen die Bodenpunkte von den Objektpunkten getrennt werden um eine exakte Objekterkennung zu garantieren.

Die Segmentierung des Bodens wird in C.LABEL-VR nach dem Einlesen des internen Datenformates ausgeführt, also zwischen Import Addons und PCL Generator (siehe Abbildung 4.2). Beim Erstellen der Punktwolken ist also schon bekannt welcher Punkt ein Bodenpunkt ist und welcher nicht. Dazu wird im internen Datenformat das Attribut für die Bodenpunkt kennzeichnung mit dem entsprechenden Wert versehen (vgl. Abschnitt 4.1.2). Wie diese Werte ermittelt werden ist im Folgenden erklärt.

Als Basis für die Bodenpunktanalyse in C.LABEL-VR wurde das *Ground Plane Fitting*-Verfahren (GPF) aus [5] verwendet. Ist im weiteren Verlauf dieser Arbeit die Rede von GPF, ist immer der Ansatz der eben zitierten Arbeit gemeint. Ziel dieses Verfahrens ist es ein mathematisches Modell einer Ebene zu errechnen, welches den Boden repräsentieren soll. Anschließend kann von jedem Punkt der Abstand zu dieser Ebene ermittelt werden. Ist der Abstand eines Punktes kleiner als ein definierter Wert, so handelt es sich bei ihm um einen Bodenpunkt.

Die Punktwolke, die analysiert werden soll, wird dabei zunächst in eine Anzahl an N_{segs} Segmenten aufgeteilt. Der Algorithmus zur Identifizierung der Bodenpunkte wird für jedes dieser Segmente ausgeführt. Der Grund für diese Segmentierung ist die mögliche Unebenheit der Bodenfläche. Soll eine wellige Oberfläche von einer Ebene repräsentiert werden kommt es zu ungenauen Ergebnissen. Wird aber für jedes Segment der Punktwolke eine eigene Ebene ermittelt so verbessert sich das Ergebnis der Bodenpunktanalyse, wie in Abbildung 4.6 zu sehen ist.

Beim GPF-Verfahren wurden Punktwolken in 3 große Segmente entlang der x -Achse eingeteilt, da diese Achse der Fahrtrichtung des Autos entspricht. Damit wurden für die Testdaten, die CMORE für diese Arbeit zur Verfügung stellte, keine zufriedenstellenden

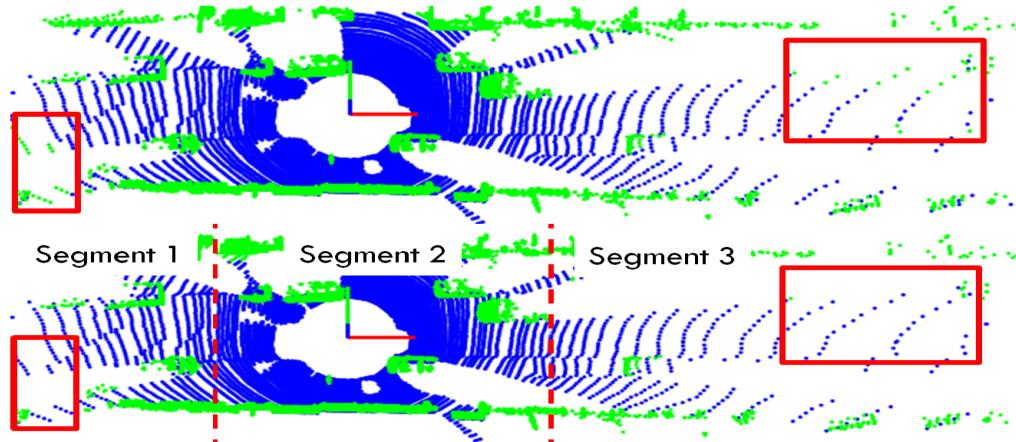
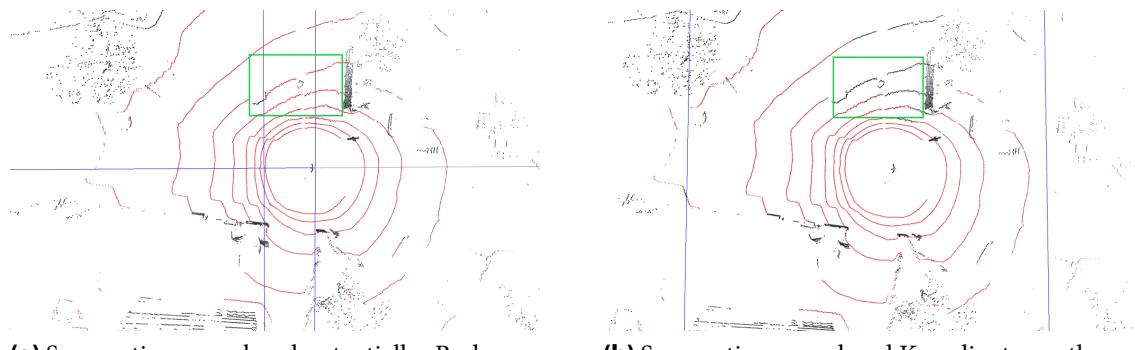


Abbildung 4.6.: In [5] wird gezeigt, dass die Segmentierung einer Punktwolke zu einem besseren Ergebnis der Bodenpunktanalyse führt. Die obere Abbildung ist dabei ohne Segmentierung und die untere mit Segmentierung. Bodenpunkte sind blau und Objektpunkte in grün dargestellt. Die roten Rechtecke kennzeichnen den Bereich in dem die Analyse schlechter bzw. besser ist.

Ergebnisse erzielt (vgl. Abbildung 4.7b). Das Problem ist, dass sich durch eine zu simple Segmentierung entweder zu viele oder zu wenige Bodenpunkte in einem Sektor befinden können. Bei zu vielen Bodenpunkten kann es sein, dass der Boden nicht linear verläuft. Auf der linken Seite des Autos könnte sich beispielsweise eine Anhöhe befinden und auf der rechten Seite ein flaches Gelände. In diese Unebenheit kann keine richtige Ebene eingesetzt werden. Bei zu wenigen Bodenpunkten kann es sein, dass der Algorithmus die Ebene an Objekte annähert. Handelt es sich bei solchen Objekten beispielsweise um eine senkrechte Wand, können die vielen hohen Punkte die Ebene senkrecht werden lassen (siehe Abbildung 4.8).

Darum wurde in dieser Arbeit ein Verfahren entwickelt, dass die Segmente nicht anhand ihrer Größe einteilt sondern nach der Anzahl der potentiellen Bodenpunkte, welche die Segmente enthalten (siehe Algorithmus 1). Potentielle Bodenpunkte sind diejenigen Punkte, welche sich in einem bestimmten Höhenbereich befinden. Der Höhenbereich wurde auf -0.5m bis 0.5m festgelegt. Das bedeutet alle Punkte, deren Höhenwert sich innerhalb des Wertebereichs befindet, sind potentielle Bodenpunkte. Das Segmentierungsverfahren erstellt nun die Segmente so, dass sich in jedem Segment gleich viele dieser potentiellen Bodenpunkte befinden.

Dazu werden alle Punkte des gesamten Segments nach ihrem x -Wert sortiert und anschließend wird durch diese Menge an sortierten Punkten iteriert. Bei jeder Iteration wird geprüft ob der aktuell betrachtete Punkt ein potentieller Bodenpunkt ist oder nicht. Ist das der Fall wird eine Zählvariable erhöht. Übersteigt der Wert dieser Variable die Anzahl an potentiellen Bodenpunkten, die in einem Segment sein sollen, werden zukünftig betrachtete Punkte in ein neues Segment aufgenommen. Die Zählvariable wird anschließend wieder



(a) Segmentierung anhand potentieller Bodenpunkten

(b) Segmentierung anhand Koordinaten entlang der x -Achse

Abbildung 4.7.: Vergleich der Segmentierungsmethoden zur Bodenpunktanalyse aus [5] und C.LABEL-VR. Bodenpunkte sind rot, alle anderen schwarz. Die blauen Linien stellen die Trennung der Segmente dar. Die grünen Boxen markieren den Bereich, in dem man den Unterschied der beiden Methoden bei der Bodenpunkterkennung sieht.

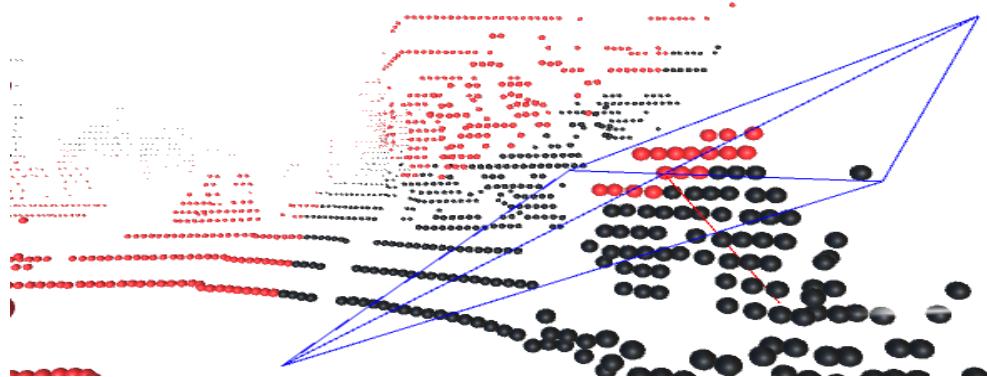


Abbildung 4.8.: Bei zu wenigen Bodenpunkten in einem Segment passt sich die Ebene den falschen Punkten an.

auf Null gesetzt um im neuen Segment die potentiellen Bodenpunkte erneut zu zählen.

Algorithm 1 Algorithmus zur Einteilung einer Punktwolke in N_{segs} Segmente

1: **Inputs:**

P : Menge aller Punkte

N_{segs} : Anzahl der Segmente

2: **Initialize:**

S : Menge aller Segmente mit ihren Punkten

P_{pot} : Menge aller potentiellen Bodenpunkte

C_{segs} : Zähler des aktuellen Segments

C_{points} : Zähler der Bodenpunkte des aktuellen Segments

N_{gpps} : Anzahl der potentiellen Bodenpunkte pro Segment

3: **MainLoop:**

4: $P_{sorted} = \text{SORTASCENDINGONXVALUE}(P);$

5: $P_{pot} = \text{GETPOINTSBETWEENHEIGHTVALUES}(-0.5, 0.5, P_{sorted});$

6: $N_{gpps} = \left\lceil \frac{|P_{pot}|}{N_{segs}} \right\rceil;$

7: $C_{segs} = C_{points} = 0;$

8: **for** $i = 1 : |P_{sorted}|$ **do**

9: **if** $p_i.yValue \geq 0$ **then**

10: $S[C_{segs}] \leftarrow p_i;$

11: **else**

12: $S[C_{segs} + 1] \leftarrow p_i;$

13: **end if**

14: **if** $p_i \in P_{pot}$ **then**

15: $C_{points} ++;$

16: **end if**

17: **if** $C_{points} \geq N_{gpps}$ **then**

18: $C_{points} = 0;$

19: $C_{segs} += 2;$

20: **end if**

21: **end for**

Zusätzlich zu der Segmentierung aus [5] wurden die Segmente noch durch die y -Achse geteilt, um den vorher angesprochenen Problemfall der unterschiedlichen Geländebeschaffenheiten links bzw. rechts vom Auto auszugleichen. Dazu wird bei jeder Iteration der y -Wert des betrachteten Punktes geprüft. Punkte mit einem größeren Wert als Null werden in ein anderes Segment aufgenommen als solche mit kleinerem Wert. Das Ergebnis und der direkte Vergleich mit der normalen GPF-Methode ist in Abbildung 4.7a zu sehen.

Nach dem die Segmentierung der Punktwolke vorgenommen wurde, muss nun die Bodenebene für jedes Segment berechnet werden. Die Definition dieses Vorgehens ist in Algorithmus 2 gegeben. Auch dafür wurde die GPF-Methode aus [5] als Basis hergenommen. Zuerst werden gewisse Startpunkte ermittelt, welche auch *Seedpoints* bzw. *Seed-Punkte* genannt werden. Laut GPF werden diese Punkte durch den *LPR*-Wert (*lowest point representative*) ermittelt. Um diesen Wert zu berechnen wird zunächst eine Anzahl an N_{LPR} Punkten definiert. Danach werden die N_{LPR} tiefsten Punkte des Segments genommen und

der Durchschnitt der Höhenwerte dieser Punkte berechnet. Alle Punkte des Sektors die eine maximale Höhendifferenz T_{init} zu diesem Wert haben sind Seed-Punkte.

Dieses Vorgehen ist allerdings anfällig für Rauschen in Form von sehr tiefen Punkten. Befinden sich im Segment also fälschlicherweise Punkte, die tief unter dem Boden sind, verfälschen diese die richtige Berechnung des Durchschnittshöhenwerts. Deshalb wurden in C.LABEL-VR für die Berechnung Durchschnittshöhenwerts LPR nicht die N_{LPR} tiefsten Punkte hergenommen, sondern alle potentiellen Bodenpunkte die sich in dem Segment befinden. Aus diesem Durchschnittshöhenwert und der maximal zulässigen Differenz T_{init} werden, wie schon beschrieben, die Startpunkte ermittelt. Die Formale Beschreibung dieses Vorgehens ist ebenfalls in Algorithmus 2 zu sehen (GetInitialSeedPoints).

Mit diesen Startpunkten kann nun die erste von N_{iter} Schätzungen für die Bodenebene berechnet werden. Eine Ebene E kann definiert werden durch einen Vektor \vec{n} der Senkrecht auf der Ebene steht (Normalenvektor) und einem Punkt p der auf der Ebene liegt. Ziel ist es nun diese beiden Komponenten so zu berechnen, dass die dadurch definierte Ebene bestmöglich in Seed-Punkte passt. Für den Punkt p kann man den einfach Durchschnitt \hat{s} aus allen Elementen der Menge aller Seed-Punkte S nehmen. Dieser Punkt repräsentiert den Ursprung der Ebene und bietet sich deshalb an weil man ihn bei späteren Berechnungen wiederverwenden kann (siehe Gleichung 4.1).

Die Berechnung des Normalenvektors \vec{n} ist dagegen deutlich aufwändiger. Zunächst wird die Streuung aller Punkte $s \in S$ untersucht. Dazu wird eine Kovarianzmatrix C des Ranges R^{3x3} berechnet, welche die Streuung der Seedpoints repräsentiert. Diese Matrix kann nun auf geometrische Eigenschaften untersucht werden, beispielsweise welcher Vektor die kleinste Streuung repräsentiert und somit als \vec{n} verwendet werden kann.

$$C = \sum_{i=1:|S|} (s_i - \hat{s})(s_i - \hat{s})^T \quad (4.1)$$

Im Allgemeinen gilt für eine Matrix Folgendes: „Eine Matrix definiert durch $y = A * x$ eine lineare Abbildung $R^m \rightarrow R^n$. Der Hauptberuf einer Matrix ist, aus einem Vektor einen anderen zu machen. Im Allgemeinen ändert die Matrix dadurch Richtung, Betrag (und sogar Dimension) eines Vektors. Matrixzerlegungen spalten die Aktion der Matrix in leichter zu durchblickende Einzelschritte auf“ [50]. Aus diesen Einzelschritten können geometrische Informationen extrahiert werden. Bei der GPF-Methode wird C durch die Singulärwertzerlegung gespalten. Dieses Verfahren wird auch *Singular Value Decomposition* (SVD) genannt. Wie in [51] gezeigt, wird bei SVD eine Matrix $A \in R^{m \times n}$ in 2 orthogonale Matrizen $U \in R^{m \times m}$ und $V \in R^{n \times n}$ und eine diagonal Matrix Σ zerlegt, sodass die Gleichung 4.2 erfüllt wird. Der Normalenvektor \vec{n} der gesuchten Ebene ist gegeben durch

die dritte Spalte der Matrix U [52].

$$A = U\Sigma V^T \quad (4.2)$$

Die gesuchte Ebene ist nun durch den Punkt p und den Vektor \vec{n} nun eindeutig definiert. Nun kann der Abstand jedes Punktes des Segments zu seiner orthogonalen Projektion auf der definierten Ebene berechnet werden. Dieser Abstand wird dann mit einem Schwellwert T_{plane} verglichen, welcher zu Beginn der Berechnung definiert werden muss. Ist der Abstand eines Punktes kleiner als der Schwellwert so handelt es sich bei ihm um einen Bodenpunkt. In C.LABEL-VR wird solch ein Punkt vorläufig als Bodenpunkt markiert und in die Menge an neuen Seed-Punkten aufgenommen. Mit diesen neuen Seed-Punkten wird anschließend die nächste von N_{iter} Berechnungen der Bodenebene getätigt(vgl. Algorithmus 2).

Cluster-Analyse

TODO Erklärung des Cluster-Algorithmus (Recursive Radius Search)

4.5. User Interface

TODO -kurze Einleitung -Erklärung der Architektur hinter dem UI-System

4.5.1. Applikations-Menü

TODO Erklärung der Einstellbaren Parameter im Ingame Menü

Algorithm 2 Algorithmus zur Identifizierung von Bodenpunkten eines Punktwolken-Segments

- 1: **Inputs:**
 \mathbf{P} : Menge aller Punkte der Wolke
- 2: **Initialize:**
 \mathbf{P}_{seeds} : Menge aller Seed-Punkte
 N_{iter} : Anzahl der Iterationen
 T_{init} : Distanzgrenzwert der initialen Seed-Punkte
 T_{plane} : Distanzgrenzwert zur geschätzten Ebene
- 3: **MainLoop:**
 - 4: $\mathbf{P}_{seeds} = \text{GETINITIALSEEDPOINTS}(\mathbf{P}, T_{init});$
 - 5: **for** $i = 1 : N_{iter}$ **do**
 - 6: $plane = \text{FITPLANEINTOPOLY}(P_{seeds});$
 - 7: $\text{CLEAR}(\mathbf{P}_{seeds});$
 - 8: **for** $j = 1 : |\mathbf{P}|$ **do**
 - 9: $distance = \text{GETDISTANCETOPLANE}(plane, p_k);$
 - 10: **if** $distance < T_{plane}$ **then**
 - 11: $p_k.groundpoint = true;$
 - 12: $\mathbf{P}_{seeds} \leftarrow p_k$
 - 13: **else**
 - 14: $p_k.groundpoint = false;$
 - 15: **end if**
 - 16: **end for**
 - 17: **end for**
 - 18:
- 19: **GetInitialSeedPoints:**
- 20: **Initialize:**
 \mathbf{P}_{low} : Alle tiefen Punkte aus \mathbf{P}
 H_{avg} : Durchschnittshöhe aller Punkte aus \mathbf{P}_{low}
- 21: **for** $i = 1 : |\mathbf{P}|$ **do**
- 22: **if** $p_i.height < 0.5 \ \& \ p_i.height > -0.5$ **then**
- 23: $\mathbf{P}_{low} \leftarrow p_i$
- 24: **end if**
- 25: **end for**
- 26: $H_{avg} = \text{GETAVERAGEHEIGHT}(\mathbf{P}_{low});$
- 27: **for** $i = 1 : |\mathbf{P}|$ **do**
- 28: **if** $p_i.height < T_{init}$ **then**
- 29: $\mathbf{P}_{seeds} \leftarrow p : i;$
- 30: **end if**
- 31: **end for**

Schluss

5.1. Zusammenfassung

5.2. Erreichte Ziele

5.3. Vorteil der VR-Applikation

5.4. Ausblick

A

Appendix Title

Literaturverzeichnis

- [1] Microsoft, “Machine learning – cheat sheet für algorithmen für microsoft azure machine learning studio,” Internetquelle(abgerufen am 21.03.2018), Dec. 2017.
- [2] S. S. Fisher, M. McGreevy, J. Humphries, and W. Robinett, “Virtual environment display system,” in *Proceedings of the 1986 workshop on Interactive 3D graphics - SI3D '86*. ACM Press, 1987.
- [3] P. Chu, S. Cho, S. Sim, K. Kwak, and K. Cho, “A fast ground segmentation method for 3d point cloud,” *Journal of information processing systems*, vol. 13, no. 3, pp. 491–499, 2017.
- [4] L. Oculus VR, “Ovrinput,” Internetquelle (abgerufen am 12.05.2018), May 2018. [Online]. Available: <https://developer.oculus.com/documentation/unity/latest/concepts/unity-ovrinput/>
- [5] D. Zermas, I. Izzat, and N. Papanikolopoulos, “Fast segmentation of 3d point clouds: A paradigm on LiDAR data for autonomous vehicle applications,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2017.
- [6] D. Kriesel, *Ein kleiner Überblick über Neuronale Netze*, 2007. [Online]. Available: erhältlich auf <http://www.dkriesel.com>
- [7] C. V. D. Malsburg, “Frank rosenblatt: Principles of neurodynamics: Perceptrons and the theory of brain mechanisms,” in *Brain Theory*. Springer Berlin Heidelberg, 1986, pp. 245–248.
- [8] I. E. Sutherland, “The ultimate display,” in *Proceedings of the IFIP Congress*, 1965, pp. 506–508.
- [9] G. Enrico, “Virtual reality: Past, present and future,” *Studies in Health Technology and Informatics*, vol. 58, no. Virtual Environments in Clinical Psychology and Neuroscience, pp. 3–20, 1998.
- [10] P. R. Desai, P. N. Desai, K. D. Ajmera, and K. Mehta, “A review paper on oculus rift-a virtual reality headset,” *CoRR*, vol. abs/1408.1173, 2014.

- [11] TheMorenar, "Oculus rift cv1 - vrtk avatar sdk gamedev 3," Internetquelle (abgerufen am 07.05.2018), May 2018. [Online]. Available: <https://www.youtube.com/watch?v=Czc4V-Mxb5k>
- [12] MSPoweruser, "All oculus rift users left in the dark following "cant reach oculus runtime services" error: Update – patch now available," Internetquelle (abgerufen am 07.05.2018), May 2018. [Online]. Available: <https://mspoweruser.com/all-oculus-rift-users-left-in-the-dark-following-cant-reach-oculus-runtime-services-error/>
- [13] C. Gammage, "Exploring oculus rift: A historical analysis of the 'virtual reality' paradigm," *ART 108: Introduction to Games Studies*, 2017.
- [14] K. M. Johnson and W. B. Ouimet, "Rediscovering the lost archaeological landscape of southern new england using airborne light detection and ranging (LiDAR)," *Journal of Archaeological Science*, vol. 43, pp. 9–20, mar 2014.
- [15] D. E. Smith, "The global topography of mars and implications for surface evolution," *Science*, vol. 284, no. 5419, pp. 1495–1503, may 1999.
- [16] S. K. Reddy and P. K. Pal, "Segmentation of point cloud from a 3d LIDAR using range difference between neighbouring beams," in *Proceedings of the 2015 Conference on Advances In Robotics - AIR '15*. ACM Press, 2015.
- [17] S. W. Greg Vellante, "Ikea joins forces with htc vive for virtual reality kitchen," Internetquelle (abgerufen am 07.05.2018), Apr. 2016. [Online]. Available: <http://techhomebuilder.com/emagazine-articles-1/ikea-joins-forces-with-htc-vive-for-virtual-reality-kitchen>
- [18] P. Gothard, "Microsoft hololens: Firm admits sales figures are in the 'thousands,'" Internetquelle (abgerufen am 24.11.2017), Jan. 2017. [Online]. Available: <https://www.theinquirer.net/inquirer/news/3003380/microsoft-hololens-firm-admits-sales-figures-are-in-the-thousands>
- [19] Microsoft, "Microsoft hololens choose the option that best suits your purpose." Internetquelle (abgerufen am 27.11.2017), Nov. 2017. [Online]. Available: <https://www.microsoft.com/de-de/hololens/buy>
- [20] R. Berg, "Touch-controller machen oculus rift zur besten vr-brille," Internetquelle (abgerufen am 24.11.2017), Dec. 2016. [Online]. Available: <https://www.welt.de/wirtschaft/webwelt/article160455301/Touch-Controller-machen-Oculus-Rift-zur-besten-VR-Brille.html>
- [21] J. Munafo, M. Diedrick, and T. A. Stoffregen, "The virtual reality head-mounted display oculus rift induces motion sickness and is sexist in its effects," *Experimental Brain Research*, vol. 235, no. 3, pp. 889–901, dec 2016.

- [22] W. M. R. D. Forum, “Creating a new gesture,” Internetquelle (abgerufen am 28.11.2017), Apr. 2016. [Online]. Available: <https://forums.hololens.com/discussion/549/creating-a-new-gesture>
- [23] Walmart, “Htc vive virtual reality system,” Internetquelle (abgerufen am 07.05.2018), May 2018. [Online]. Available: <https://www.walmart.com/ip/HTC-VIVE-Virtual-Reality-System-Black-99HALN00200/754371281>
- [24] Amazon, “Oculus rift + touch bundle,” Internetquelle (abgerufen am 07.05.2018), May 2018. [Online]. Available: <https://www.amazon.de/OCULUS-Germany-301-00095-01-Oculus-Bundle/dp/B073X8N1YW>
- [25] J.-K. Janssen, “Oculus rift im test: Virtual reality für die massen,” Internetquelle (abgerufen am 27.11.2017), Mar. 2016. [Online]. Available: <https://www.heise.de/ct/artikel/Oculus-Rift-im-Test-Virtual-Reality-fuer-die-Massen-3151909.html>
- [26] ——, “Oculus touch im test: So echt können sich händе in vr anfühlen,” Internetquelle (abgerufen am 27.11.2017), Dec. 2017. [Online]. Available: <https://www.heise.de/ct/artikel/Oculus-Touch-im-Test-So-echt-koennen-sich-Haende-in-VR-anfuehlen-3552295.html>
- [27] R. Artus, “Lighthouse von valve erklärt,” Internetquelle (abgerufen am 28.11.2017), Nov. 2017. [Online]. Available: <https://vrjump.de/lighthouse-erklaert>
- [28] L. Painter, “Htc vive review,” Internetquelle (abgerufen am 28.11.2017), Aug. 2017. [Online]. Available: <http://www.techadvisor.co.uk/review/wearable-tech/htc-vive-review-2017-3635648/>
- [29] O. R. Support, “Was sind die empfohlenen mindestsystemvoraussetzungen, die für den betrieb der oculus rift erforderlich sind?” Internetquelle (abgerufen am 29.11.2017), Nov. 2017. [Online]. Available: <https://support.oculus.com/170128916778795/>
- [30] P. Gora and L. Leibetseder, “Unreal vs unity: Ein vergleich zwischen zwei modernen spiele-engines,” Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, Oct. 2016, 1. [Online]. Available: https://www.cg.tuwien.ac.at/research/publications/2016/Przemyslaw_Gora_2016_UVU/
- [31] U. Technologies, “Die weltweit führende software der game-industrie,” Internetquelle (abgerufen am 24.04.2018), Apr. 2018. [Online]. Available: <https://unity3d.com/de/public-relations>
- [32] Unity, “Importing objects from blender,” Internetquelle (aufgerufen am 04.05.2018), May 2018. [Online]. Available: <https://docs.unity3d.com/560/Documentation/Manual/HOWTO-ImportObjectBlender.html>

- [33] ——, “Monobehaviour,” Internetquelle (abgerufen am 01.05.2018), May 2018. [Online]. Available: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>
- [34] C. Games, “Unity3d tips 2 – the singleton pattern,” Internetquelle (abgerufen am 01.05.2018). [Online]. Available: <http://clearcutgames.net/home/?p=437>
- [35] Unity, “Prefabs,” Internetquelle (abgerufen am 13.05.2018), May 2018. [Online]. Available: <https://docs.unity3d.com/Manual/Prefabs.html>
- [36] ——, “Mesh renderer,” Internetquelle (abgerufen am 13.05.2018), May 2018. [Online]. Available: <https://docs.unity3d.com/Manual/class-MeshRenderer.html>
- [37] ——, “Object.instantiate,” Internetquelle (abgerufen am 13.05.2018), May 2018. [Online]. Available: <https://docs.unity3d.com/ScriptReference/Object.Instantiate.html>
- [38] ——, “Light probes,” Internetquelle (abgerufen am 13.05.2018), May 2018. [Online]. Available: <https://docs.unity3d.com/Manual/LightProbes.html>
- [39] ——, “Colliders,” Internetquelle (abgerufen am 13.05.2018), May 2018. [Online]. Available: <https://docs.unity3d.com/560/Documentation/Manual/CollidersOverview.html>
- [40] ——, “Draw call batching,” Internetquelle (abgerufen am 13.05.2018), May 2018. [Online]. Available: <https://docs.unity3d.com/Manual/DrawCallBatching.html>
- [41] E. Bozgeyikli, A. Raij, S. Katkoori, and R. Dubey, “Point & teleport locomotion technique for virtual reality,” pp. 205–216, 10 2016.
- [42] J. J. LaViola, Jr., “A discussion of cybersickness in virtual environments,” *SIGCHI Bull.*, vol. 32, no. 1, pp. 47–56, Jan. 2000. [Online]. Available: <http://doi.acm.org/10.1145/333329.333344>
- [43] L. Rebenitsch and C. Owen, “Review on cybersickness in applications and visual displays,” *Virtual Reality*, vol. 20, no. 2, pp. 101–125, apr 2016.
- [44] E. M. Kolasinski, “Simulator sickness in virtual environments,” p. 68, 05 1995.
- [45] R. Pausch, T. Crea, and M. Conway, “A literature survey for virtual environments: Military flight simulator visual systems and simulator sickness,” *Presence: Teleoper. Virtual Environ.*, vol. 1, no. 3, pp. 344–363, Jul. 1992. [Online]. Available: <http://dx.doi.org/10.1162/pres.1992.1.3.344>
- [46] IrisVR, “The importance of frame rates,” Internetquelle (abgerufen am 12.05.2018), May 2018. [Online]. Available: <https://help.irisvr.com/hc/en-us/articles/215884547-The-Importance-of-Frame-Rates>

- [47] C. Boletsis, “The new era of virtual reality locomotion: A systematic literature review of techniques and a proposed typology,” *Multimodal Technologies and Interaction*, vol. 1, no. 4, p. 24, sep 2017.
- [48] M. Hoy, J. Dauwels, and J. Yuan, “Efficient ground object segmentation in 3d lidar based on cascaded mode seeking,” in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, Oct 2017, pp. 1–6.
- [49] S. Goga and S. Nedevschi, “An approach for segmenting 3d lidar data using multi-volume grid structures,” in *2017 13th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, Sept 2017, pp. 309–315.
- [50] E. H. Clemens Brand, “Matrixzerlegungen. Überbestimmte systeme,” Vorlesungsskript, Montanuniversität Leoben, Mar. 2014.
- [51] D. Kalman, “A singularly valuable decomposition: The svd of a matrix,” *College Math Journal*, vol. 27, pp. 2–23, 1996.
- [52] I. Söderkvist, “Using svd for some fitting problems,” Lulea University of Technology.