Detektion und Tracking von Händen in TOF Tiefeninformationen

Bachelorarbeit

im Fach Informatik

vorgelegt von

Roberto Cespi

Geboren am 15. Juli, 1987 in Wetzlar

Angefertigt am

Lehrstuhl für Computergraphik und Multimediasysteme
Fachbereich 12
Universität Siegen

Betreuer:

Dr.-Ing. M. Lindner, Lehrstuhl Computergraphik und Multimediasysteme, Universität Siegen

Prof. Dr. A. Kolb, Lehrstuhl Computergraphik und Multimediasysteme, Universität Siegen

Beginn der Arbeit: 11. Oktober 2010

Abgabe der Arbeit: 25. Februar 2011

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Wetzlar, den 25. Februar 2011

Zusammenfassung

Für diese Bachelorarbeit wurde ein einfaches System entwickelt, welches die automatische Detektion sowie das zeitliche Tracking beider Hände eines Users in einem Stream von Time-of-Flight Tiefeninformationen ermöglicht. Quelle dieser Tiefeninformation ist die Tiefenkamera PMD CamCube. Diese ermöglicht die Echtzeit-Aquisition einer Szene von 204^2 px. Zum Clustern des Bildes wurde ein bereits umgesetzter Parallel-Merging-Algorithmus benutzt, dessen lokales Merge-Kriterium auf das Hand-Tracking angepasst wurde. Aus der Anzahl an Clustern werden daraufhin beide Handcluster detektiert und anschließend durch ein auf der CPU entwickeltes *nearest neighbor* Verfahren getrackt. Das Tracking wurde so umgesetzt, dass ein Überkreuzen der Hände möglich ist.

Inhaltsverzeichnis

Verzeichnis der Bilder

Listings

Abkürzungsverzeichnis

GPU	Graphics Processing Unit
CPU	Central Processing Unit
PMD	Photonic Mixer Device
TOF	Time of Flight
FBO	Framebuffer Object
th	threshold
h1/h2	hand1 / hand2

Kapitel 1

Einleitung

Die natürlichste Arbeit für einen Menschen ist die, die er direkt mit seinen eigenen Händen verrichten kann. Je komplexer oder schwieriger eine Arbeit wird, umso mehr Maschinerie wird benötigt, diese zu verrichten. Je mehr Maschinerie benötigt wird, desto unnatürlicher wird die Arbeit für einen Menschen, da er sich erst in die Funktionsweise der Maschinerie einarbeiten muss. Sei es der Bauarbeiter bei der Bedienung des Krans, der Graphiker am PC mit der Maus, Tastatur und seinem Zeichentablett oder andere Arbeiten, die nicht ohne weitere Steuereinheiten verrichtet werden können. Würde der Mensch selbst zur Steuereinheit seiner Arbeit werden, könnte dies manche Arbeitsvorgänge erheblich erleichtern. Der Grafiker beispielsweise müsste somit nicht ein am PC erstelltes Graphikobjekt mit der Maus bearbeiten, sondern könnte, nur durch die Bewegung seiner Hände, direkt das Objekt mit den Händen greifen, drehen, vergrößern und bearbeiten, wie als würde er es in seinen Händen halten. Die Arbeitsweise wird dadurch natürlicher und einfacher zu erlernen.

Eine solche kontaktlose Mensch-Maschinen-Interaktion könnte dadurch realisiert werden, dass der Nutzer und dessen Handbewegungen via einer Kamera aufgenommen wird und die Daten aus der Kamera daraufhin ausgewertet werden. Allerdings ist die Bewegungs- und Handerkennung durch eine optische Kamera nicht optimal, da diese keine Tiefeninformationen der Bildpunkte erfassen kann und somit Abstände zwischen mehreren Objekten nicht optimal ausgewertet werden können. Durch eine Time-of-Flight-Kamera (TOF) kommt diese Fehlende 3D Komponente hinzu, was eine Objekterkennung im Raum vereinfacht.

In dieser Arbeit beschäftige ich mich daher mit der Erkennung und Verfolgung von Händen, die mittels einer TOF Kamera aufgenommen wurden. Genauer wurde für diese Arbeit eine PMD CamCube benutzt. Das PMD Prinzip wird in Kapitel 2.1 erklärt. Der für die Arbeit umgesetzte Algorithmus verarbeitet die Informationen der PMD CamCube und clustert das gegebene Bild in Regionen. Aus der Menge an Regionen werden beide Hände erkannt und in den Folgebildern verfolgt (Tracking). Das Tracking wurde so umgesetzt, dass eine Überkreuzung der Hände möglich ist. Für das Clustern wurde ein entsprechender Parallel-Merging-Algorithmus bereits umgesetzt, dessen lokales Merge-Kriterium

untersucht und auf das Hand-Tracking angepasst werden soll.

Nachdem in Kapitel 2 die theoretischen Grundlagen für die Arbeit dargelegt werden, werden in Kapitel 3 bekannte Arbeiten erläutert, die sich mit einer ähnlichen Thematik, wie diese Arbeit, beschäftigen. Kapitel 4 beinhaltet die Umsetzung des entwickelten Systems. In dem Unterkapitel 4.1 wird das genutzte Clusterverfahren und in 4.2 der Tracking-Algorithmus für die Hände vorgestellt. Die Ergebnisse meiner Arbeit werden in Kapitel 5 dargestellt. Zum Schluss gibt es in Kapitel 6 eine Zusammenfassung, sowie einen Ausblick auf zukünftige Arbeiten.

Kapitel 2

Theoretische Grundlagen

In diesem Kapitel werden relevante Grundlagen und Techniken für die vorliegende Ausarbeitung erläutert. Kapitel 2.1 zeigt einen kurzen Einblick in die Funktionsweise der PMD CamCube, mit der die Szene aufgenommen wird. Die von der CamCube gelieferten Tiefendaten können durch Clusterverfahren segmentiert werden. Drei solcher Clusterverfahren werden im Folgekapitel 2.2 vorgestellt und erläutert.

2.1 PMD

Die für diese Arbeit benutzte PMD CamCube [Rin07] ist eine TOF (*time of flight*) Kamera, die mit einem PMD Sensor ausgestattet ist. TOF Kameras haben, wie in Kapitel 1 schon erwähnt, den Vorteil, dass sie zu den Intensitätswerten jeden Pixels, auch noch die Distanzwerte der aufgenommenen Punkte im Raum speichern können. Dazu wird ein moduliertes Lichtsignal von der Beleuchtungseinheit der CamCube an das jeweilige Messobjekt gesendet. Das von dem Objekt reflektierte Licht wird daraufhin wieder von dem PMD Sensor-Chip empfangen (siehe Bild 2.1).

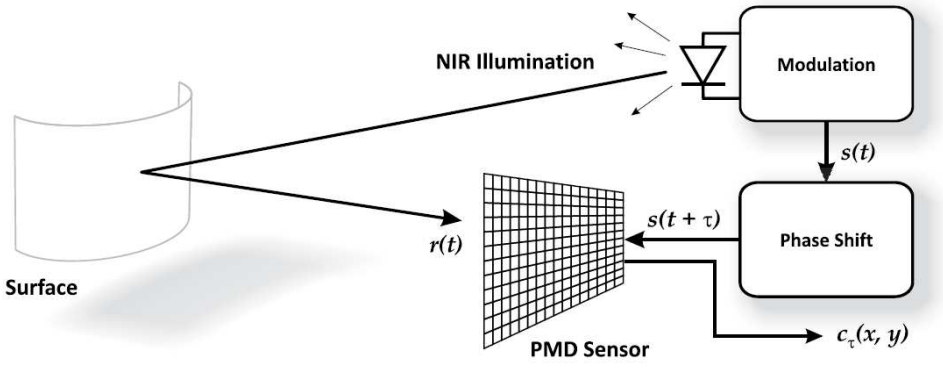


Bild 2.1: Das Prinzip von TOF Kameras mit kontinuierlich modulierter Beleuchtung [Lin10]

In der Theorie wird die Distanz d zu einem Messobjekt, durch die Signallaufzeit des Lichtes t, folgendermaßen berechnet:

$$d = \frac{c \cdot t}{2}$$

c entspricht dabei der Lichtgeschwindigkeit.
Bei der PMD CamCube, wird die Szene mit moduliertem Infrarotlicht, mit einer Frequenz von 20 MHz, beleuchtet. Intern wird, durch Messen der Phasenverschiebung des kontinuierlich modulierten Lichts pro Pixel, die Szene auf ein Pixelarray abgebildet. Da dieser Vorgang bei PMD CamCubes individuell pro Pixel stattfindet, werden diese PMD Elemente auch *smart pixel* genannt.
Die Phasenverschiebung φ wird durch die Autokorrelationsfunktion (engl.: *autocorrelation function* (ACF), siehe Bild 2.2) beider Signale bestimmt. Hierfür werden 4 Phasenwerte genommen, die jeweils um 90 Grad des internen Referenzsignals verschoben wurden. φ wird dadurch folgendermaßen berechnet:

$$\varphi = \arctan\left(\frac{A_1 - A_3}{A_2 - A_4}\right)$$

Entsprechend der Lichtgeschwindigkeit, ergibt sich die Distanz aus dem Phasenversatz wie folgt:

$$d = \frac{c \cdot \varphi}{4\pi \cdot f_{mod}}$$

wobei f_{mod} der Frequenz der Modulation entspricht.
Zu der Phasenverschiebung kann man, mit Hilfe der vier Phasenwerte $\{A_1, \ldots, A_3\}$, noch zwei weitere Werte berechnen. Zum einen die Signalstärke a, des vom Messobjekt wieder zurückkehrenden Signals:

$$a = \frac{\sqrt{(A_1 - A_3)^2 + (A_2 - A_4)^2}}{2}$$

Zum anderen kann man das Offset b berechnen, welches den Grauwerten des jeweiligen Pixels repräsentiert:

$$b = \frac{A_1 + A_3 + A_2 + A_4}{4}$$

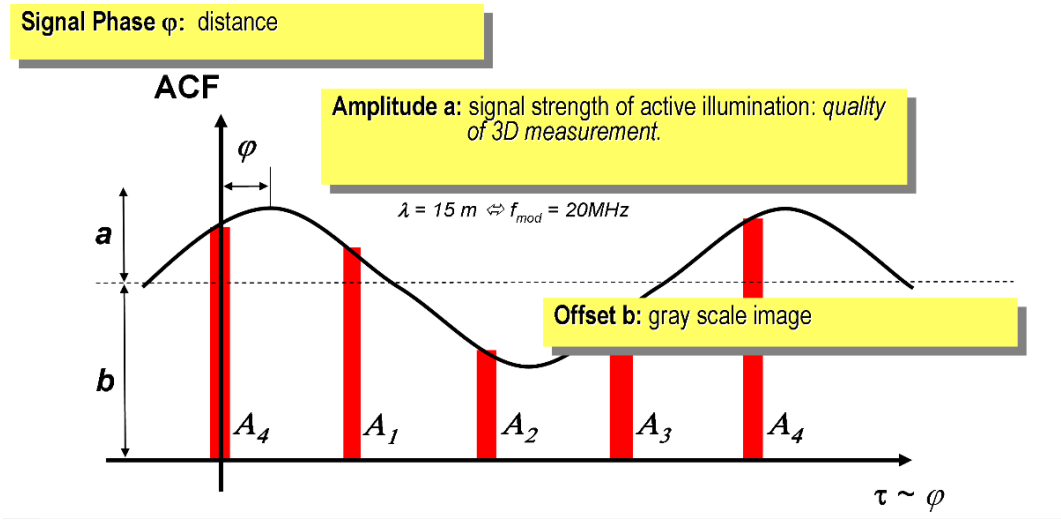


Bild 2.2: ACF (*autocorrelation function*), Signalphase, Amplitude und Offset [Rin07]

Bei einer Modulationsfrequenz von 20 MHz beträgt die Wellenlänge des Lichtes λ_{mod} = 15 Meter, berechnet durch:

$$\lambda_{mod} = \frac{c}{f_{mod}}$$

Da die Phasen proportional zur Distanz sind, kann somit eine Maximaldistanz von 7,5 Metern erfasst werden. Die Maximaldistanz ist $d_{max} = \lambda_{mod}/2$, weil λ_{mod} den Weg von der Lichtquelle, zum Messobjekt und wieder zurück, repräsentiert.

Die für diese Arbeit benutzte PMD CamCube, hat eine Auflösung von 204^2px.

2.2 Clusterverfahren

Clusterverfahren sind Verfahren um ein Bild in einzelne Cluster (Pixelgruppen / Regionen) einzuteilen. Ziel dabei ist es, alle Pixel die zu einem bestimmten Objekt gehören, zu einer Region zusammenzufassen. Dies kann einerseits dadurch erreicht werden, dass man jedes Pixel als einzelnes Element betrachtet und stätig mit anderen Pixeln vereint (*merge*), um Regionen zu bilden, oder man betrachtet das Gesamtbild als Ganzes und zerteilt es in die einzelnen Regionen (*split*). Eine dritte Möglichkeit der Regionsbildung ist, das Bild zu Beginn in eine feste Anzahl an Regionen aufzuteilen und diese daraufhin, durch Umverteilung der Pixel, zu optimieren. In unserem Fall gehören die einzelnen Hände zu jeweils einer Region.
Formal ist eine Region R eine Gruppe von Pixeln mit folgenden Eigenschaften [WLR89]:

- Ein Pixel p_i in R ist mit p_j in R verbunden, wenn es eine Sequenz $\{p_i, \ldots, p_j\}$ gibt in der p_k und p_{k+1} direkte Pixelnachbarn sind und alle Pixel in R sind.

- R beinhaltet nur Pixel die gegenseitig miteinander verbunden sind
- Das Gesamtbild $I = \bigcup_{k=1}^{m} R_k$
- $R_i \cap R_j = \emptyset, i \neq j$

Im Folgenden werden 3 Clusterverfahren erläutert.
Das hierarchische Verfahren (*merge*) in 2.2.1, das K-Means Verfahren (Optimierung) in 2.2.2 und zum Schluss das Graph Cuts Verfahren (*split*) in 2.2.3

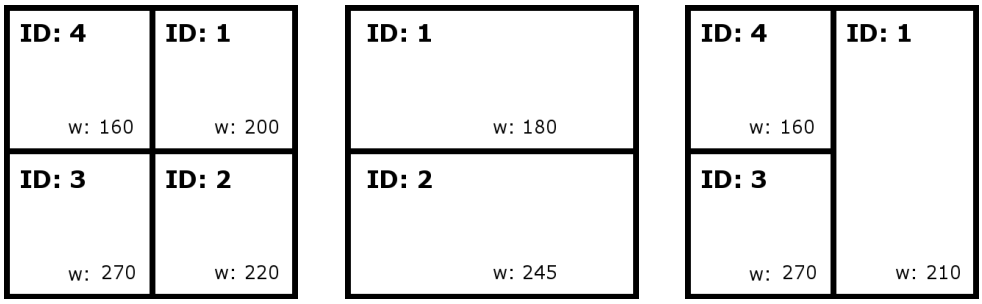
2.2.1 Hierarchisches Verfahren

Beim hierarchischen Clustern [WLR89] wird initial jedes Pixel in einem Bild als eine eigene Region angesehen und besitzt eine eigene Regions-ID. Damit nun mehrere kleine Regionen zu einer großen zusammengefasst werden können, werden diese nach einem gewissen Merge-Kriterium (bzw. Homogeniätskriterium) der Reihe nach verglichen. Die Regionen können nach verschiedenen Eigenschaften, wie z.B. der Intensität, der Distanz zur Kamera, der Normalen am Punkt, usw. verglichen werden.

Beispiel:
Nehmen wir an, die Regionen werden anhand ihrer Intensitätswerte verglichen. Dann ist das Merge-Kriterium $M(R_i, R_j)$ folgendermaßen definiert:
Sei $f(R_i)$ der mittlere Intensitätswert von R_i und $f(R_j)$ der mittlere Intensitätswert der Nachbarregion R_j und $\Delta f = f(R_i) - f(R_j)$.
Dann gilt:

$$M(R_i, R_j) = \begin{cases} true, & \Delta f < t \\ false, & \text{sonst .} \end{cases}$$

wobei t das gesetzte Threshold ist, das eingehalten werden muss.
Aus der Menge an Nachbarregionen, die dieses Merge-Kriterium erfüllen, wird die Nachbarregion als optimaler Merge-Partner gewält, die das Homogenitätskriterium am besten erfüllt, d.h. mit der die Differenz Δf minimal ist. Die nun neu entstandene Region bekommt, je nachdem wie es zu Beginn festgelegt wurde, jeweils die größere bzw. kleinere ID, der beiden fusionierten Regionen. Die Durchschnittsintensität wird daraufhin neu berechnet. Dieser Vorgang wird so oft wiederholt, bis keine benachbarten Regionen mehr das Merge-Kriterium erfüllen.



(a) (b) (c)

Bild 2.3: (a) ist ein in 4 Regionen aufgeteiltes Bild mit je einer Regions-ID. Durch ein Gewicht w, werden die Regionen auf Homogenität geprüft. Threshold $t = 45$. (b) Regionsbildung nach sequentiellen Mergen von links oben nach rechts unten. (c) Regionsbildung nach sequentielles Mergen von rechts oben nach links unten.

Wie in Bild 2.3 zu sehen ist, hängt die finale Regionsbildung von der Reihenfolge ab, in der die Regionen zusammengefügt werden. Beginnt die optimale Partnersuche bei Region 4, resultiert das Ergebnis aus Bild 2.3(b). Würde die optimale Partnersuche bei Region 1 beginnen, wird deutlich, dass Region 4 nicht der optimale Merge-Partner für Region 1 ist (siehe Ergebnis in Bild 2.3(b)). Dieses Beispiel zeigt also, dass durch das sequentielle Verfahren nicht garantiert werden kann, dass sich nur die Regionen zu einer Region zusammenfügen, die auch optimal (mit $min(\Delta f)$) zueinander passen.
Eine günstigere Regionsbildung wäre dann gegeben, wenn sich die Regionen R_i und R_j gegenseitig als optimalen Merge-Partner ansehen würden. Es müsste also bei jedem Durchgang parallel geprüft werden, ob die optimale Nachbarregion von R_i, auch R_i als den für sich optimalen Merge-Partner ansieht. Hierfür wird ein paralleles Merging-Verfahren benutzt, welches in Kapitel 4.1.1 näher erläutert wird.

2.2.2 K-Means Verfahren

Beim K-Means Verfahren [GLHL07, Chi10] wird initial eine feste Zahl K an Regionen gewählt, die aus dem Gesamtbild $\{p_1, p_2, \ldots, p_N\}$ gebildet werden soll. Dazu werden nach Zufallsprinzip K-viele Pixel gewählt, die das Clusterzentrum μ_k der jeweiligen Region repräsentiert.
Um nun die restlichen Pixel, den jeweiligen Regionen zuordnen zu können, wird anhand folgender Funktion:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|p_n - \mu_k\|^2$$

das globale Minimum von J gesucht.
$\|p_n - \mu_k\|^2$ entspricht dabei der Distanz zwischen dem Pixel p_n und dem Clusterzentrum μ_k und r_{nk}

ist eine binäre Membership-Funktion, die für jedes Pixel p_n folgendermaßen definiert ist:

$$r_{nk} = \begin{cases} 1, & \text{Wenn } p_n \text{ zu Region } R_k \text{ Anhand von Homogänitätskriteríen zuordenbar} \\ 0, & \text{sonst .} \end{cases}$$

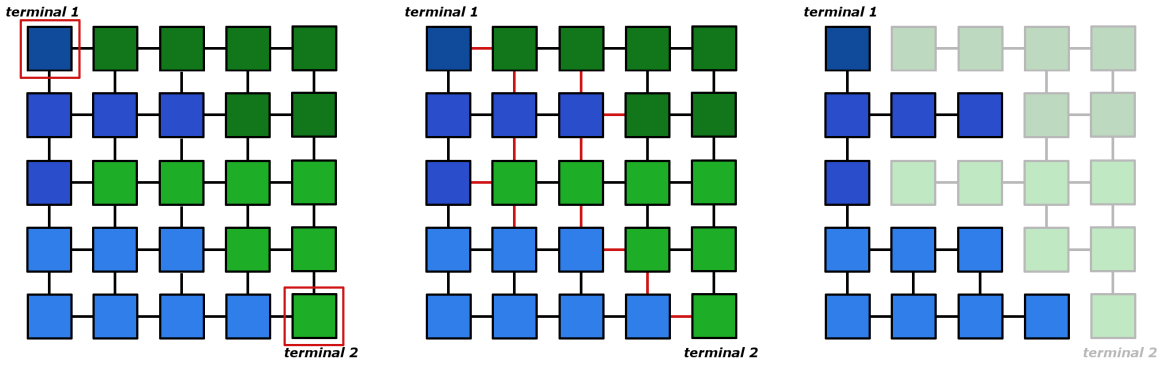
Nach diesem ersten Durchlauf sind nun alle Pixel jeweils einer aus den k Regionen zugeordnet.
Im nächsten Durchlauf werden die Regionsschwerpunkte μ_k neu berechnet und die Pixel der Gesamtszene wieder neu zugeordnet, indem das globale Minimum von J gesucht wird.
Dieser Durchlauf wird so oft wiederholt bis:

- eine festgelegte maximale Iterationstiefe erreicht wurde oder
- sich die Regionsschwerpunkte nicht mehr bewegen, d. h. bei der Neuverteilung kein Pixel einer anderen Region zugeordnet wurde

2.2.3 Graph Cuts Verfahren

Im Gegensatz zu den Verfahren, in denen jedes Pixel als einzelne Regionen angesehen werden und diese stetig, durch Verschmelzung mit Nachbarregionen, zu größeren Regionen heranwachsen, wird beim Graph Cuts Verfahren [BVZ01] die zu clusternde Szene zu Beginn als Ganzes gesehen.
Da das Graph Cuts Verfahren ein Graphen-basiertes Verfahren ist, repräsentieren alle Pixel im Bild eine Menge von Knoten V eines Graphen G, bei dem jedes Pixel mit seinen Nachbarpixeln durch eine Menge von Kanten E verbunden ist.

In diesem gewichteten Graph $G = \langle V, E \rangle$, werden zwei sich voneinander unterscheidende Pixel gewählt, die *terminals* genannt werden. Diese beiden *terminals* werden anschließend durch einen Cut $C \subset E$ getrennt. D.h. C ist die Menge von Kanten, die dem Graphen $G(C) = \langle V, E - C \rangle$ entnommen werden, sodass es keine Kantenverbindung (sei es über mehrere Pixel) gibt, die die beiden *terminals* verbindet. Entscheidungskriterium, ob eine Kante gelöscht wird, hängt von den Kosten der Kante ab. Die Kostenfunktion einer Kante zwischen zwei benachbarten Pixeln $C(p_i, p_j)$ gibt dabei an, wie sehr sich p_i und p_j ähneln. Die Gesamtkosten eines Cuts sind also die Summe der Kosten aller Kanten in C.



(a) (b) (c)

Bild 2.4: Ein 5 x 5 großes Pixelfeld. Jedes Pixel repräsentiert einen Knoten, der mit jedem Nachbarknoten durch eine Kante verbunden ist. Nachdem in (a) zwei Pixel als *terminals* bestimmt wurden, werden diese durch Cuts voneinander getrennt (rot markierte Kanten in (b)), wodurch zwei neue Teilgraphen entstehen (c)

Danach kann jeder Teil-Graph erneut mit dem oben beschriebenen Verfahren in zwei Teile aufgeteilt werden. Die optimale Lösung bei diesem Verfahren wäre der *minimum cut*. Hierbei wird der günstigste Cut gesucht, der die beiden *terminals* voneinander trennt.

Kapitel 3

Bekannte Arbeiten

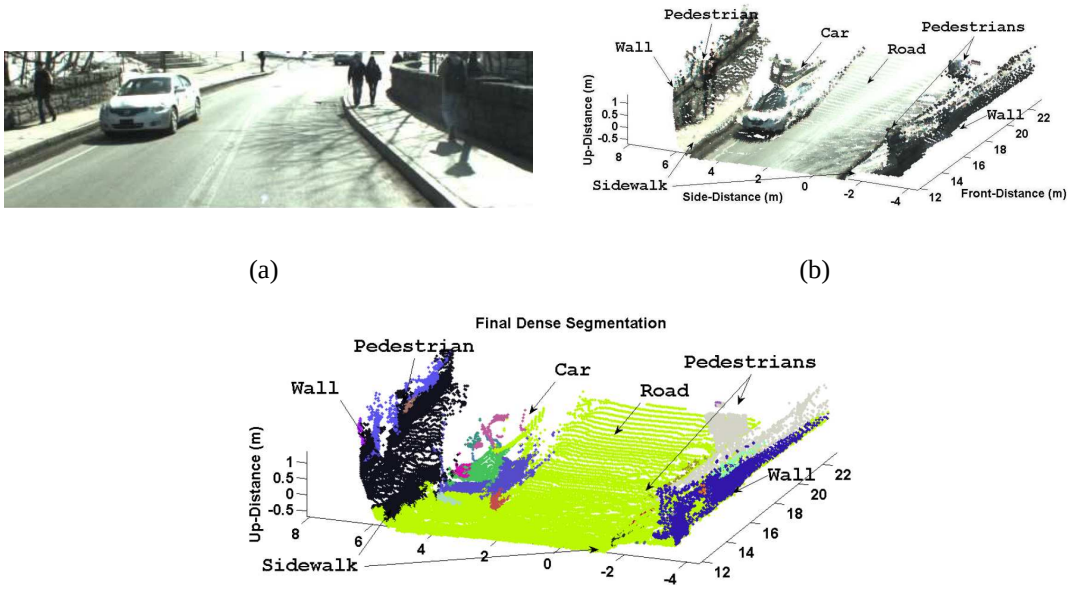
Nachdem in Kapitel 2.2 einige Clusterverfahren abgehandelt wurden, werden in diesem Kapitel bekannte Arbeiten erläutert, die sich mit einer ähnlichen Thematik befassen, wie diese Arbeit.

In [SNC10] wird sich mit der Segmentierung von Straßen/Verkehrs - Szenen befasst. Die Straßenszene wird mittels einer optischen Kamera und einem *laser range finder* aufgenommen (siehe Bild 2.2.3). Zur Segmentierung der Szene wird ein Graph-basiertes Verfahren genutzt. Der ungerichtete Graph $G(C) = \langle V, E \rangle$ bildet sich hierbei aus den Tiefeninformationen des *laser range finder*, wobei die Knoten V die 3D Punkte im Raum repräsentieren und durch Kanten E miteinander verbunden sind. Zu Beginn des Algorithmus wird das Gewicht ω einer jeden Kante zwischen den beiden Nachbarknoten i und j errechnet. ω_{ij} ist durch eine gewichteten Kombination aus euklidischen Abstand, Intensitätsdifferenz der Pixel und errechneter Oberflächennormale folgendermaßen definiert:

$$\omega_{ij} = k_e \cdot \|\vec{v}_i - \vec{v}_j\|^2 + k_I \cdot \|x_i - x_j\|^2 + k_N \cdot (1.0 - \vec{N}_i^T \vec{N}_j)$$

wobei $\vec{N}$ die errechnete Oberflächennormale des 3D Punktes ist und k_e, k_I und k_N jeweils die relativen Gewichte für den euklidischen Abstand, die Intensitätsdifferenz der Pixel und der errechneter Oberflächennormale sind. Abweichend vom Graph Cuts Verfahren in Kapitel 2.2.3, wird zu Beginn jeder Knoten als *terminal* betrachtet. Daraufhin werden die Knoten zu einem Cluster gefasst, dessen Kantengewicht unter dem jeweiligen Clusterthreshold η_i liegt. Nachdem ein Knoten zu einem Cluster hinzugefügt wurde, wird dessen Clusterthreshold neu berrechnet.

Auf einem Desktop PC mit einem 2.7GHz Intel®Core™i7 Prozessor benötige dieses Verfahren bei einer Auflösung von 337920 Pixel 1852ms und bei einer Auflösung von 84480 Pixel 513ms pro Frame [SNC10]. Auf Grund der Laufzeit-Angaben und des linearen Verhaltens ergibt sich für ein Bild der Auflösung 204^2 px eine geschätzte Laufzeit von 286ms pro Frame. Da sich somit nur 3-4 Bilder pro Sekunde segmentieren lassen, ist dieses Verfahren nicht für die Echtzeit-Segmentierung geeignet.



(a) (b)

(c)

Bild 3.1: Segmentierung einer Straßenszene: (a) Farbbild, (b) texturiertes Tiefenbild, (c) Segmentierungsergebnis [SNC10].

In [GLHL07] wird ein Segmentierungsverfahren vorgestellt, welches eine Hand aus Tiefen und Intensitätsinformationen von einer 3D TOF Kamera mit PMD Sensor segmentieren soll. Für die Regionsbildung werden hierbei jedem einzelnen Pixel eine Eigenschaft zugeordnet, nach der sie mit anderen Pixeln aus dem Pixelarray, auf Homogänität geprüft werden kann. Diese Eigenschaft wird anhand eines 2D Vectors f_{rc} repräsentiert:

$$f_{rc} = (z_{rc}, \varphi_{rc}) \tag{3.1}$$

wobei z_{rc} die Position des Objektes in z-Richtung, bezogen auf das Weltkoordinatensystem, repräsentiert. φ_{rc} hingegen, entspricht einem komplexen Wert φ aus dem Polaren Koordinatensystem. Dieser komplexe Wert wird für jede Zeile r und Spalte c des Pixelarrays folgendermaßen errechnet:

$$\varphi_{rc} = \arctan\left(\frac{d_{rc}}{g_{rc}}\right) \tag{3.2}$$

d_{rc} enspricht dabei dem normierten Distanzwert und g_{rc} dem normierten Intensitätswert des jeweiligen Pixels.

Zur Segmentierung des Bildes wurden zwei Clusterverfahren kombiniert. Diese sind das K-Means Verfahren (siehe Kapitel 2.2.2) und das *Expectation Maximization* Verfahren (kurz. EM-Verfahren), welches im Laufe dieses Abschnittes näher erläutert wird.

Das EM-Verfahren lässt sich in zwei Arbeitsschritte aufteilen. Einem Erwartungsschritt (E-Schritt) und einem Maximierungsschritt (M-Schritt). Im Erwartungsschritt wird für jedes Pixel p_n ein so genannter Erwartungswert ausgerechnet, welcher aussagt, wie optimal p_n zum jeweiligen Clusterschwerpunkt μ_k passen könnte. Im Maximierungsschritt werden mit den Erwartungswerten die Parameter neu berechnet, die für die Errechnung der Erwartungswerte im Erwartungsschritt nötig sind. Da der E-Schritt und der M-Schritt somit voneinander abhängig sind, wird das EM-Verfahren in der Initialisierung mit festgelegten Parametern eingeleitet. Der E- und M-Schritt werden daraufhin so oft wiederholt, bis der Algorithmus eine Maximum-Likelihood-Schätzung ausgibt.
Das EM-Verfahren wurde folgendermaßen implementiert:

- Initialisierung: Die Parameter, die errechnet werden sollen, werden in diesem Schritt vorerst initialisiert. Diese Parameter sind der Clusterschwerpunkt μ_k, die Kovarianz Σ_k und der Mischkoeffizient π_k.

- Erwartung: In diesem Schritt werden die Erwartungswerte $E(z_{nk})$ für jedes Pixel p_n errechnet.

$$E(z_{nk}) = \frac{\pi_k N(p_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(p_n|\mu_j, \Sigma_j)}$$

- Maximierung: Nachdem die Erwartungswerte $E(z_{nk})$ errechnet wurden, werden die oben genannten Parameter neu bestimmt:

$$\begin{aligned} \mu_k^{new} &= \frac{1}{N_k} \sum_{n=1}^{N} E(z_{nk}) p_n \\ \Sigma_k^{new} &= \frac{1}{N_k} \sum_{n=1}^{N} E(z_{nk})(p_n - \mu_k^{new})(p_n - \mu_k^{new})^T \\ \pi_k^{new} &= \frac{N_k}{N} \end{aligned}$$

  wobei

$$N_k = \sum_{n=1}^{N} E(z_{nk})$$

- Auswertung: In diesem Schritt wird die logarithmische Wahrscheinlichkeit und die Konvergenz der Parameter ausgewertet. Falls das Konvergenzkriterium dabei nicht erfüllt wird, beginnt der Algorithmus wieder beim Erwartungsschritt.

Das K-Means Verfahren ist ein distanzabhängiger Algorithmus, d.h. die Pixel werden anhand ihrer Entfernung zu den Clusterschwerpunkten zugeordnet. Daher können kleine Positionsveränderungen eines Punktes dazu führen, dass das dazugehörige Pixel zu einem anderen Cluster zugeordnet wird. Falls nur das EM-Verfahren verwendet wird, kann es passieren, dass, durch ungünstige Parameter

in der Initialisierung, ungünstige Clusterergebnisse entstehen. Um dies zu vermeiden, wurde das K-Means Verfahren mit dem EM-Verfahren kombiniert (Zusammen: KEM). Dabei liefert K-Means im ersten Schritt die Clusterzentren μ_k. Das EM-Verfahren benutzt diese für die Initialisierung und sucht in den Folgeiterationen das lokale Maximum.

Kapitel 4

Clustern & Tracking

Dieses Kapitel befasst sich mit dem für meine Arbeit umgesetzten System. Dieses System soll aus Tiefen- und Intensitätsinformationen, welche durch eine TOF-Kamera aufgenommen wurden, die Hände der aufgenommenen Person erkennen und daraufhin verfolgen. Dazu wird das Bild zuallererst geclustert, um die einzelnen Hände zu segmentieren. Anschließend werden aus der Menge an Regionen die Hände identifiziert und verfolgt.

4.1 Clustern

Um das Bild in einzelne Regionen zu clustern, wurde das hierarchische Clusterverfahren verwendet (siehe Kapitel 2.2.1), welches um ein paralleles Merging-Verfahren erweitert wurde. Im Gegensatz zum sequentiellen hierarchischen Verfahren, wird hier parallel überprüft, ob sich die zwei zu fusionierenden Regionen gegenseitig als optimalen Merge-Partner ansehen. Nur wenn das der Fall ist, können beide Regionen miteinander vereinigt werden. Die Funktionsweise des parallelen Merging-Verfahrens wird in Kapitel 4.1.1 erläutert. Wie der Algorithmus implementiert wurde, steht anschließend in Kapitel 4.1.2.

Das Merge-Kriterium, nach der die Regionen auf Homogänität verglichen werden, basiert auf der komplexen Größe f_{rc} (siehe Formel 3.1), allerdings wurde φ auf andere Weise berechnet. Da φ aus dem Verhältnis zwischen dem mittleren Distanz- und Intensitätswert der jeweiligen Region errechnet wird, lässt die Berechnung von φ in Formel 3.2 allerdings annehmen, dass die Intensität I einer Fläche, mit steigender Entfernung d zur Kamera, linear wächsts. Jedoch verhält sich die Intensität zum Abstand der Lichtquelle folgendermaßen:

$$I \propto \frac{1}{d^2} \Rightarrow \frac{1}{\sqrt{I}} \propto d$$

somit ist der Wert $\frac{d}{\frac{1}{\sqrt{I}}} = d\sqrt{I}$ ungefähr konstant für jede Fläche. Der Winkel φ wird daher, für diese

Arbeit, wie folgt berechnet:

$$\varphi_R = arctan(d_R\sqrt{I_R}) \tag{4.1}$$

d_R entspricht dabei der normierten Distanz von R und I_R der normierten Intensität von R. Die Eigenschaft einer Region wird somit durch folgenden 2D Vektor dargestellt:

$$f_R = (z_R, \varphi_R) \tag{4.2}$$

wobei z_R den durchschnittlichen Positionswert von R in z-Richtung, bezogen auf das Weltkoordinatensystem, repräsentiert.

4.1.1 Parallels Merging-Verfahren

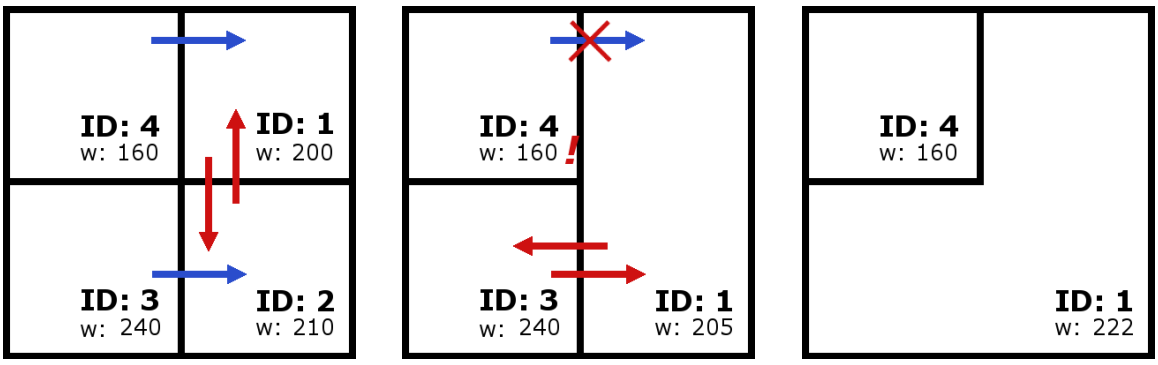
Wie am Ende von Kapitel 2.2.1 schon erwähnt wurde, werden für jede Region alle Nachbarregionen gesucht, die ein bestimmtes Homogänitätskiterium erfüllen. Aus der Menge an Nachbarregionen wird daraufhin die Nachbarregion als optimaler Merge-Partner gewählt, die das Homogenitätskriterium am besten erfüllt, d.h. mit der die Differenz Δf minimal ist.
Für die Findung des optimalen Merge-Partners und für die eigentliche Fusion zweier Regionen, müssen folgende Regeln beachtet werden [WLR89]:

1. Jede Region kann sich pro Durchlauf nur mit einer einzigen Nachbarregion, die dass Homogenitätskriterium am besten erfüllt, zu einer Region fusionieren.

2. Wenn eine Region mehrere Nachbarregionen als optimale Merge-Partner ansieht, wird die Nachbarregion gewählt, die die größere bzw. die kleinere (je nachdem wie zu Beginn festgelegt) Regions-ID besitzt.

3. Zwei Regionen können nur miteinander fusionieren, wenn sie sich gegenseitig als optimalen Merge-Partner sehen.

Diese Regeln erlauben einer Region sich bei jedem Iterationsschritt mit nur einem einzigen Nachbar zu vereinigen. Andernfalls könnte die neue Region das Homogenitätskriterium verletzen, wie Bild 4.1 zeigt. Die Übernahme der kleineren (bzw. größeren) ID als neue Region-ID verhindert, in Verbindung mit Regel 2, mögliche Deadlocks. Nach der Fusionierung zweier Nachbarregionen wird die Regionseigenschaft der neuen Region neu berechnet.
Konnte eine Region sich nicht mit seinem optimalen Merge-Partner vereinen, da dieser eine andere Region als optimalen Merge-Partner wählte, hat sie vielleicht im nächsten Iterationsschritt die Chance dazu (siehe Bild 4.1).



(a) (b) (c)

Bild 4.1: t = 40. (a) Da Region 1 und 2 sich gegenseitig als optimalen Merge-Partner ansehen und pro Iteration nur jeweils eine Region mit einer anderen fusionieren darf, müssen Region 3 und 4 auf den nächsten Iterationsschritt warten. (b) Nun kann Region 3 mit der neuen Region 1 fusionieren. Region 4 im Gegensatz erfüllt nicht mehr das Homogenitätskriterium. (c) entspricht der finalen Regionsbildung.

Die einzelnen Schritte werden so oft wiederholt, bis kein optimaler Merge-Partner für eine beliebige Region vorhanden ist.

4.1.2 Implementierung des Clusterns

Zu Beginn, werden die 204^2 Tiefen- und Intensitätsinformationen des jeweiligen Frames, von der PMD CamCube ausgelesen. Da der Parallel-Merging-Algorithmus auf der GPU implementiert wurde [CK11] und Shadern keine Arrays übergeben werden können, werden die Informationen via FBOs direkt in die Farbkanäle von 204^2px großen Texturen gerendert. Folgende Texturen werden für das Clustern benötigt:

- ID-Textur: In der ID-Textur ist für jedes Pixel die Regions-ID des optimalen Merge-Partners gespeichert.
- Regions-Textur: Die Regions-Textur beinhaltet für jedes Pixel die aktuelle ID der zugehörigen Region.
- Data-Textur: Für jedes Pixel wird hier der φ-Wert, sowie die $(x, y, z)^T$ Koordinaten des Clusterschwerpunkts m der eigenen Region gespeichert.
- Data2-Textur: In der Data2-Textur ist unter anderem für jedes Pixel die Pixelanzahl des zugehörigen Clusters gespeichert.

Jede Textur ist hierbei doppelt vorhanden, da eine Textur als Input der Daten dient, wohingegen in die zweite Textur die Ergebnisse gespeichert werden.

Die Daten aus der jeweiligen Pixelposition jeder Textur repräsentiert dabei dasselbe Pixel aus dem aufgenommenem Bild. Da die Regions-IDs, nach einem gewissen Schema verteilt wurden, kann anhand der Funktion vec2 map(int regionID) die Pixelposition des Pixels ausgegeben werden, dessen ursprüngliche Regions-ID der eingegebenen ID entspricht. Durch die Pixelposition können in der jeweiligen Textur die Cluster-Daten ausgegeben werden, in dem sich das Pixel befindet.

Der Clustering-Algorithmus ist in folgende Schritte eingeteilt. Diese werden so oft wiederholt bis kein optimaler Merge-Partner für eine beliebige Region vorhanden ist, bzw. eine festgelegte maximale Iterationstiefe erreicht wurde:

Bestimme den optimalen Merge-Partner: Die einzelnen Vertecies der Texturen werden zu Beginn vom Vertex-Shader an den Geometry-Shader durchgereicht. Im Geometry-Shader wird nun für jedes Pixel der optimale Merge-Partner gesucht. Dabei wird jedes Pixel vorerst mit seinen Nachbarpixeln verglichen, ob sie zum selben Cluster gehören. Falls dies nicht der Fall ist, beginnt die Homogänitätskontrolle (siehe Listing 4.1).

```
for (int i = 0; i < 4; i++) {

  vec2 nextCoord = coord + diff[i];

  // prüfe Gültigkeit der Koordinaten
  //
  [...]

  // Hole die Regions-ID des Nachbarpixels und prüfe auf Gültigkeit
  //
  int nextID = int(round(texture2DRect(regions, nextCoord).r));
  if (nextID < 1) continue;

  // suche möglichen Merge-Partner
  //
  if (currID != nextID) {

    // hole Daten (Clusterschwerpunkt und Phi) von nextID
    //
    vec4 nextData = texture2DRect(data, map(nextID));

    [...]

    // Errechne die Differenz (Kosten) der Phi Werte
    // und der Position in z-Richtung
    //
    float err_z = abs(currData.z - nextData.z);
    float err_phi = abs(currData.a - nextData.a);
```

```
    // Sind die Differenzen unter den jeweiligen Thresholds?
    //
    if (err_z > (dist_th))
    continue;
    if (err_phi > phi_th)
    continue;

    // normiere err_z und err_phi
    //
    err_z = err_z/max_dist;
    err_phi = err_phi/max_phi;

    // Errechne die gewichteten Gesamtkosten
    //
    float err = err_phi*phi_weight + err_z*dist_weight;

    // nimm Nachbarn mit kleinsten Kosten, bei gleichen Kosten, nimm die
    // Region mit größerer ID
    if ((err < minErr) || (err == minErr && nextID > minIndex)) {
    minIndex = nextID;
    minErr = err;
    }
  }
}
```

Listing 4.1: Ausschnitt aus dem Geometry-Shader: Lokale Suche nach dem optimalen Merge-Partner pro Pixel

Aus der Menge an gültigen Nachbarpixeln wird das Pixel als optimaler Merge-Partner gewählt, welches minimale Gesamtkosten hat. Haben mehrere Pixelnachbarn die gleichen minimalen Gesamtkosten, wird der Nachbar mit der größeren Regions-ID als möglicher Merge-Partner gewählt. Nun müsste mit den restlichen Pixeln aus dem Cluster verglichen werden, die an anderen Clustern angrenzen, welche Region als optimaler Merge-Partner gewählt wurde. Da der Z-Buffer-Test aber nicht zwei Werte gleichzeitig vergleichen kann, muss ein zweiter Durchlauf eingeleitet werden. Damit die ganzen Berechnungen aus Geometry-Shader nicht doppelt durchgeführt werden, wird das Ergebnis (Vertices) in einem Transform-Feedback-Buffer gespeichert und imitiert diesen beim zweiten Durchlauf. Beim Vertices mit gleichen Gesamtkosten wird im zweiten Durchlauf der Vertex mit der kleinsten ID bestimmt und in der ID-Textur gespeichert. Wurde kein optimaler Merge-Partner gefunden, wird in die ID-Textur der Wert 0 als ID gespeichert.

Mergen: Nachdem in die ID-Textur für jedes Pixel, die Regions-ID des optimalen Merge-Partners gespeichert wurde, wird nun kontrolliert, welche Regionen miteinander vereint werden können. Dies geschieht im Fragment-Shader. Für jedes Pixel wird aus der Regions-Textur die aktuelle Regions-ID

(currID) entnommen und verglichen, ob der optimale Merge-Partner (nextID) auch currID als seinen optimalen Merge-Partner (nghbID) ansieht (siehe Listing 4.2).

```
  vec2 coord = gl_TexCoord[0].st;

  // hole aktuelle Regions-ID aus der Regions-Textur (regions)
  int currID = int(round(texture2DRect(regions, coord).r));

[...]

  // get Merge-Partner and check if it is valid
  int nextID = int(round(texture2DRect(IDs, map(currID)).r));

[...]

   // wenn kein Merge-Partner gefunden wurde
   //
   if (nextID == 0) {
    gl_FragColor = vec4(currID);
    return;
  }

  // ID des optimalen Merge-Parters,
  // vom optimalen Merge-Partner der aktuellen Region
  //
  int nghbID = int(round(texture2DRect(IDs, map(nextID)).r));

  // sehen sich beide Cluster als optimalen Merge-Partner?
  if (currID == nghbID)
    gl_FragColor = vec4(min(currID, nextID));
  else
    gl_FragColor = vec4(currID);
```

Listing 4.2: Ausschnitt aus dem Fragment-Shader: Mergen der beider Cluster, die sich gegenseitig als optimalen Merge-Partner sehen. Die Regions-ID des betrachteten Pixels, wird mit der kleineren der beiden Regions-IDs überschrieben.

Falls zwei Cluster sich gegenseitig als optimalen Merge-Partner ansehen, wird an der aktuellen Pixelposition der Regions-Textur, die kleinere der beiden Regions-ID gespeichert. Falls nicht, wird die aktuelle Regions-ID gespeichert.

Update der Regionseigenschaften: Nachdem die neuen Cluster gebildet wurden, werden nun der neue Clusterschwerpunkt, der mittlere φ-Wert, sowie die Clustergröße (Anzahl der Pixel) des neuen Clusters berechnet. Hierfür wird im Fragment-Shader die ursprüngliche Regions-ID des aktuellen Pixels in currID gespeichert sowie die ID des optimalen Merge-Partners (nextID) und dessen optima-

ler Merge-Partners (nghbID). Falls sich beide Cluster als optimalen Merge-Partner sehen (currID = nghbID), wird desweiteren kontrolliert, ob das aktuelle Pixel die kleinere von beiden ursprünglichen Regions-IDs besitzt. Falls dies der Fall ist, werden die geupdateten Werte in in diese Pixelposition gespeichert. Dadurch werden nur die aktuellen Clusterwerte in das Pixel gespeichert, dessen ursprüngliche Regions-ID gleichzeitig, die aktuelle Regions-ID des Clusters ist. In allen anderen Pixelpositionen werden die Werte mit 0 überschrieben (siehe Listing 4.3).

```
void main()
{
  vec2 coord = gl_TexCoord[0].st;
  int currID = map(coord);

  vec4 currData   = texture2DRect(data, coord);
  vec4 currData2 = texture2DRect(data2, coord);

  // hole optimalen Merge-Partner
  int nextID = int(texture2DRect(IDs, map(currID)).r);

  // Kontrolliere ob Daten und IDs gültig sind, sowie ob
  // ein Merge-Partner existiert
  [...]

  int nghbID = int(texture2DRect(IDs, map(nextID)).r);

  // mergen --> in Abhängigkeit der Regionsgröße
  //
  if (currID == nghbID) {
    if (currID < nextID) {

     // Update der Clusterwerte und in 'result' speichern
     [...]

      gl_FragData[0] = result;
    }
    else
      gl_FragData[0] = vec4(0.0); // entferne die Clusterwerte
  }
  else {
    gl_FragData[0] = currData;
  }
}
```

Listing 4.3: Ausschnitt aus dem Fragment-Shader: Die Daten des neuen Clusters werden neu berrechnet. Die neuen Daten werden nur an die Pixelposition geschrieben, dessen urpsrüngliche Regions-ID die aktuelle Regions-ID ist.

Der Clusterschwerpunkt m_{new} der neuen Region, sowie der neue φ-Wert φ_{new}, werden in Abhängigkeit der Clustergrößen s der beiden Regionen R_i und R_j folgendermaßen errechnet:

$$m_{new} = \frac{m_i s_i + m_j s_j}{s_{new}}$$

$$\varphi_{new} = \frac{\varphi_i s_i + \varphi_j s_j}{s_{new}}$$

wobei

$$s_{new} = s_i + s_j$$

s_{new} repräsentiert die Pixelanzahl des neuen Clusters und wird in die Data2-Textur gespeichert.

4.2 Tracking

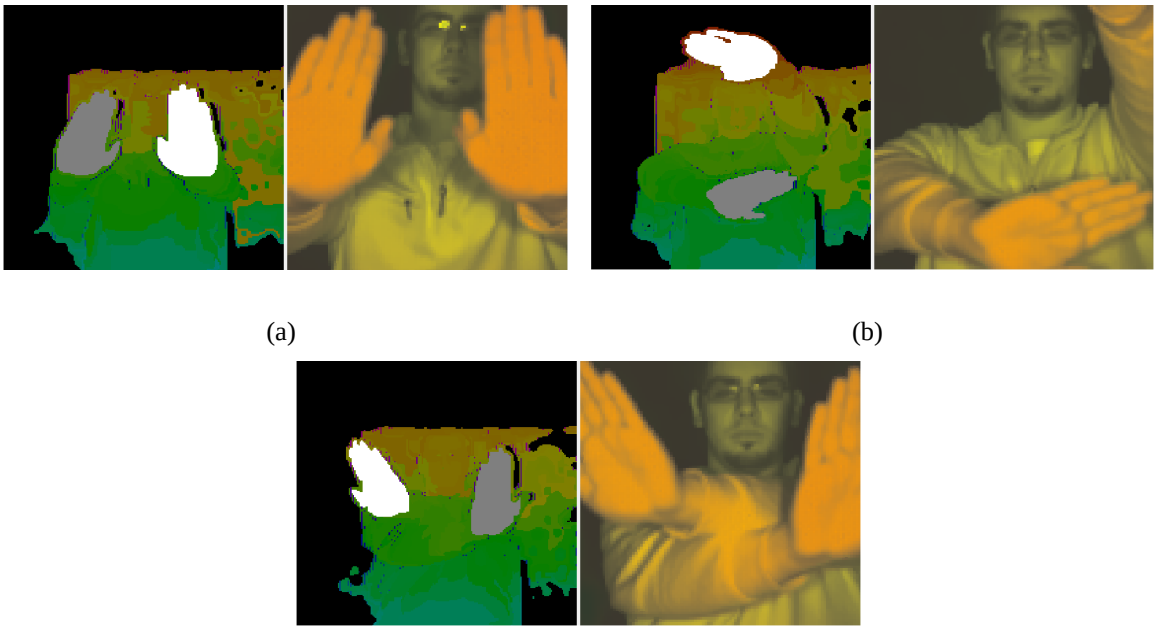
Der vorgestellte Tracking-Algorithmus wurde auf der CPU implementiert und umfasst zwei Schritte. Der erste Schritt ist die Initialisierungs, in der die Hände aus der Menge an Clustern identifiziert werden. Schritt zwei umfasst das Tracking der Hände.

4.2.1 Schritt Eins: Initialisierung

In der Initialisierung werden aus der Gesamtmenge von Regionen, folgende Regionen als Hände identifiziert, die:

- am nächsten zur TOF-Kamera sind und
- deren Größe über einem Threshold liegen

Beide Regionen werden als Hand 1 und Hand 2 gekennzeichnet. Solange beide Regionen die oben genannten Kriterien erfüllen, können beide Hände auch im Folgebild eindeutig zugeordnet werden. Das liegt daran, da beide Regionen, via *nearest neighbor* (nächster Nachbar) Verfahren, den zwei neuen Regionen im Folgebild zugeordnet werden (siehe Bild 4.2).



(a) (b)

(c)

Bild 4.2: Die Hände können, solange sie die Kriterien der Initialisierung einhalten, richtig zugeordnet werden. Hand 1 (weiß) wird somit hier der linken Hand und Hand 2 (grau) der rechten Hand zugeordnet. Links ist jeweils das geclusterte Bild, rechts das Intensitätsbild zu sehen.

Nach einer vorgegebenen Anzahl an Frames endet die Initialisierung und das eigentliche Tracking wird im nächsten Frame eingeleitet.

Im letzten Frame der Initialisierung werden folgende Werte der beiden Handcluster gespeichert:

- $regionID_{h1}$ und $regionID_{h2}$:
  Die Regions-ID der beiden Handcluster
- φ_{h1} und φ_{h2} :
  Der φ-Wert des jeweiligen Handclusters
- $size_{h1}$ und $size_{h2}$:
  Die Clustergröße, gegeben durch die Pixelanzahl des jeweiligen Handclusters
- pos_{h1} und pos_{h2} :
  Die Position der Clusterschwerpunkte von Hand 1 und Hand 2 im Raum, durch die Koordinaten $(x, y, z)^T$. Der Koordinatenursprung $(0, 0, 0)^T$ ist der Standpunkt der Kamera.

Diese Werte sind für das Tracking in Schritt 2 relevant.

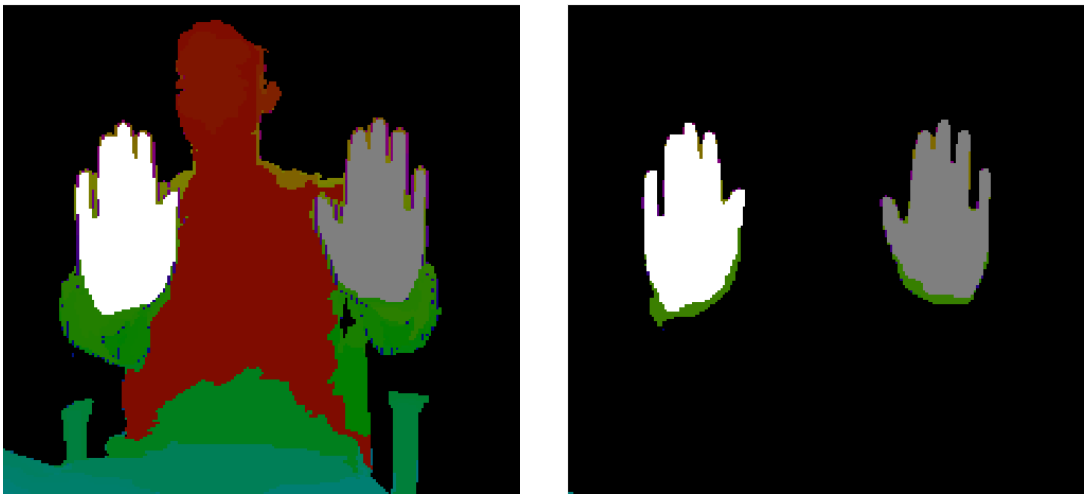
4.2.2 Schritt Zwei: Tracking

Beim Einleiten von Schritt Zwei wird der maximal r_{max} und minimal r_{min} zu erfassende Bereich der Kamera auf die Hände eingegrenzt. D.h. alle Bildpunkte die vor r_{min} oder hinter r_{max} sind, werden beim Clustern nicht mehr beachtet. Das hat den Vorteil, dass weniger Pixel zum Clustern vorhanden sind, wodurch das System weniger Rechenaufwand hat. r_{min} und r_{max} werden für jedes Frame folgendermaßen berechnet:

$$r_{min} = min(d_{h1}, d_{h2}) - r_{th}$$

$$r_{max} = max(d_{h1}, d_{h2}) + r_{th}$$

wobei, d_{h1} die durchschnittliche Distanz aller Pixel von Hand 1, d_{h2} die durchschnittliche Distanz aller Pixel von Hand 2 und r_{th} ein Distanzschwellwert ist. Nach jedem Frame wird r_{min} und r_{max} neu berechnet.
Der Distanzschwellwert wird benötigt, damit sich die vordere Hand auch weiter nach vorne bzw. die hintere Hand auch weiter nach hinten bewegen darf. Für $r_{th} = 0$ würde die jeweilige Hand bei leichten Distanzänderungen im nächsten Frame weggeclippt werden.



(a) (b)

Bild 4.3: Geclustertes Bild mit geclippten Clusterbereich. (a) zeigt das letzte Frame von Schritt Eins (Initialisierung) und (b) das erste Frame von Schritt Zwei (Tracking)

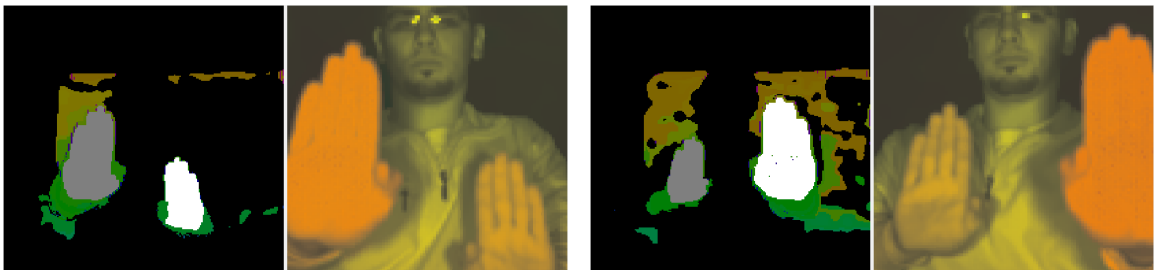
Nachdem der für das Clustern relevante Bereich festgelegt wurde, werden aus der Menge von Handclustern des neuen unbearbeiteten Frames die beiden Hände gesucht. Dazu wird jedes Cluster des aktuellen Frames, mit den beiden Handclustern des vergangenen Frames, verglichen.

```
// Durchlaufe alle Cluster und kontrolliere zu welcher Hand
//  vom vergangenen Frame es zugeordnet werden kann
//
for(int i = 0; i < cluster.size; i++)
{

// Hat das cluster einen ähnlichen Phi-Wert wie die Hand 1?
// Ist das gefundene Cluster groß genug?
//
if( ( abs( hand1.phi , cluster[i].phi ) < delta_phi_threshold)
    && ( cluster[i].size > min_size ) )
    {
        // wie weit ist die Position der Hand vom letzen Frame
        // und das aktuelle Cluster entfernt?
        //
        dist_hand1ToCluster = distance( hand1.position , cluster.position )

            found_potential_hand1 = true;
    }
        else
        {
                found_potential_hand1 = false;
        }

    // die obige if-Abfrage nun auch für hand 2
    [...]
}
```

Listing 4.4: Pseudocode für die Suche des optimalen Nachfolgecluster für das Handcluster des vergangenen Frames.

Aus der Menge an potentiellen Handclustern wird via *nearest neighbor* Verfahren bestimmt, welches Cluster der jeweiligen Hand zugeordnet wird. Nachdem die neuen Handcluster bestimmt wurden, werden die Trackinginformationen ($regionID, \varphi, size,$ und pos) der alten Händecluster mit den neuen überschrieben.

Dieser gesamte Vorgang wird für jedes Frame durchgeführt.



(a) (b)

Bild 4.4: Tracking bei Vor- und Zurückbewegung, beider Hände

4.2.3 Sonderfall: Eine Hand verdeckt die andere

Das in 4.2.2 erläuterte Verfahren reicht aus, um zwei im Bild sichtbare Hände zu tracken. Beim Überkreuzen der Hände würde dies allerdings zu Problemen führen, da die hintere Hand von der vorderen verdeckt werden würde. Der Algorithmus würde jedoch nach zwei Clustern suchen.
Um daher zu gewährleisten, dass sich zwei Hände überkreuzen können, kommt folgendes Kontrollverfahren hinzu.

Nachdem die Hände in der Initialisierung (siehe. 4.2.1) erkannt wurden, wird in jedem Frame kontrolliert, welchen Abstand die beiden Hände zueinander haben. Anhand diesen Abstandes wird eingestuft, ob sich die beiden Hände in einer Gefahrenzone Z befinden, in der sie sich überkreuzen könnten. Z ist dabei folgendermaßen definiert:

$$Z(h1,h2) = \begin{cases} true, & |pos_{h1} - pos_{h2}| < d_{min} \\ false, & \text{sonst .} \end{cases}$$

d_{min} entspricht dabei dem minimalen Abstand.

Solange beide Hände diesen minimalen Abstand einhalten, wird nur nach dem Verfahren von Kapitel 4.2.2 vorgegangen. Für $Z(h1,h2) = true$, wird folgendes Verfahren eingeleitet:

Kontrolle auf mögliches Verschwinden einer Hand: Nachdem für jedes Handcluster ein neues Nachfolgecluster gefunden wurde, wird kontrolliert, wie weit der jeweils neue Clusterschwerpunkt von dem jeweiligen alten Handclusterschwerpunkt entfernt ist. Ist die Entfernung vom alten Handcluster zum neuen größer als eine bestimmte Minimaldistanz, kann das Cluster nicht zur Hand gehören,

woraus resultiert, dass diese Hand hinter der anderen Hand verschwunden sein muss.

Wiederfindung der verschwundenen Hand: Wenn eine Hand durch die andere verdeckt wird und somit vom Bild verschwindet, werden die letzten Trackinginformationen, als die Hand noch sichtbar war, zur Wiederfindung genutzt. Die Vorderhand hf wird weiterhin nach dem Tracking-Algorithmus von Kapitel 4.2.2 getrackt. Aus den restlichen Clustern wird daraufhin die zweite Hand gesucht.

Das Cluster der wieder hervorkommenden Hand muss dabei folgende Kriterien erfüllen:

- Der Clusterschwerpunkt muss in z-Achsenrichtung hinter dem Clusterschwerpunkt der vorderen Hand liegen:
  $z_C < z_{hf}$
- Der Höhenunterschied in y-Achsenrichtung vom Clusterschwerpunkt zur vorderen Hand, muss unter einem Höhenthreshold y_{th} liegen:
  $|y_C - y_{hf}| < y_{th}$
- Der φ-Wert der Hinterhand φ_{hb} und der des Clusters φ_C
  $\Delta\varphi = |\varphi_C - \varphi_{hb}| < t_\varphi$
- Die Clustergröße muss über einem festgelegten Minimalwert liegen:
  $size_C > size_{min}$

Aus der Menge an Clustern, die diese Kriterien erfüllen, ist das Cluster mit dem kleinsten $\Delta\varphi$, die neue wiedergefundene Hand. Erfüllt kein Cluster diese Kriterien, bleibt die Hand für dieses Frame verschwunden. Dieser Vorgang wird für jedes Frame durchgeführt, bis ein Cluster die oberen Kriterien erfüllt.

Kapitel 5

Ergebnis

Nachdem in Kapitel 4 die Funktionsweise des für diese Arbeit entwickelten Systems erklärt wurde, werden in diesem Kapitel die Ergebnisse dieses Verfahrens präsentiert. Clusterergebnisse früherer lokale Merge-Kriterien werden in Kapitel 5.1 erläutert. Welche Einschränkungen eingehalten werden müssen, um eine optimale Nutzung des Programms zu ermöglichen, werden anschließend in Kapitel 5.2 aufgeführt.

5.1 Lokales Merge-Kriterium

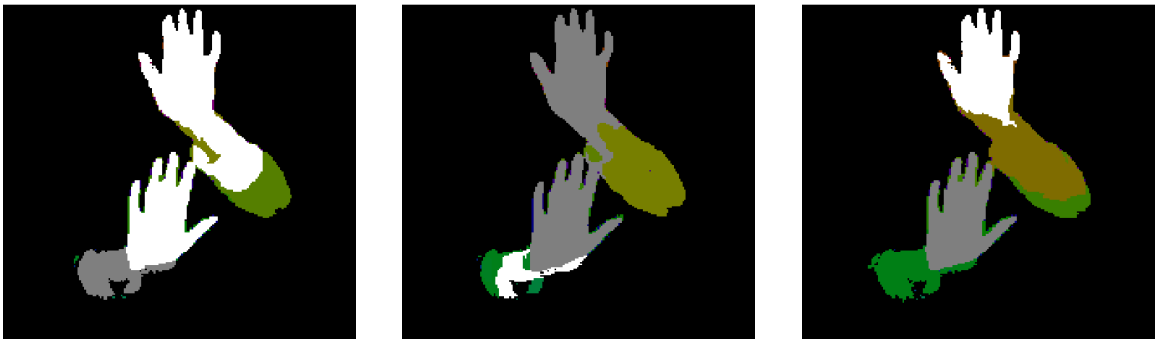
Bevor die Eigenschaften einer Region durch einen 2D Vektor f_R aus Formel 4.2 dargestellt wurde, war ein erster Ansatz die Regionseigenschaften anhand von 3 Werten zu repräsentieren. Diese sind die durchschnittliche Distanz d_R, die durchschnittliche Intensität I_R und die durchschnittliche Normale $\vec{n}_R$, der jeweiligen Region. Um allerdings eine akzeptable Regionsbildung zu erhalten, muss für den Homogenitätsvergleich der Normalen ein sehr hohes Threshold t_n gewählt werden. Die Differenz Δn_{ij} der beiden zu vergleichenden Normalen $\vec{n}_i$ und $\vec{n}_j$ ist geben durch:

$$\Delta n_{ij} = 1 - \langle \vec{n}_i, \vec{n}_j \rangle$$

Wie auf Bild 5.1(a) und 5.1(b) zu sehen ist, kann ein annehmbares Ergebnis durch $t_n = 0.9$ erziehlt werden, was einem Winkelunterschied von $81°$ entspricht. Da dadurch Regionen zusammengefasst werden können, deren Normalen beinahe Rechtwinklig zueinander stehen, kann diese Komponente auch aus dem Homogenitätsvergleich gestrichen und somit die Berechnungszeit der Normalen für jedes Pixel pro Frame eingespart werden, wie der Vergleich in Bild 5.1(c) zeigt.

(a) (b) (c)

Bild 5.1: Gleiche geclusterte Szene. (a) Clustern nach Distanz, Intensität und Normale mit $t_n = 0.1$. (b) Clustern nach Distanz,Intensität und Normale mit $t_n = 0.9$. (c) Clustern nach Formel 4.2



(a) (b) (c)

Bild 5.2: (a) Clustern mit φ aus Formel 3.2. (b) Clustern mit φ aus Formel 5.1.(a) und (b) zeigt, beide Hände vermischen sich, durch eine Verbindung über den Ärmel, zu einem Cluster. Das Handgelenk, bzw der Ärmel wird dadurch versehentlich als Hand angesehen. (c) Clustern mit φ aus Formel 4.1. Hände werden genauer geclustert.

Wie schon in Kapitel 4.1 erwähnt, basiert das finale lokale Merge-Kriterium auf der Formel 3.1. Da φ aus dem Verhältnis der mittleren Distanz und der Intensität jeden Pixels errechnet wird und die Intensität einer Fläche mit steigender Entfernung von der Lichtquelle abnimmt, wurde der Winkel φ_R in meiner Arbeit vorerst folgendermaßen berechnet:

$$\varphi_R = arctan(\frac{d_R}{1 - I_R}) \tag{5.1}$$

Da diese Formel, ähnlich wie Formel 3.1, allerdings nur eine lineare Annäherung der Abhängigkeit von Intensität zur Distanz ist, kann dies zu ungenauen Clusterergebnissen führen, wie in Bild 5.2 zu

sehen ist.

5.2 Einschränkungen

Nachdem in dem Initialisierungsschritt die Hände erkannt wurden, werden sie getrackt, allerdings gibt es Einschränkungen, die eingehalten werden müssen. Die Handflächen dürfen, in der xy-Ebene in der sie sich befinden, mit keinem Objekt in Kontakt treten, welches eine ähnliche Intensität wie die Handfläche hat, da sie sonst zu einer Region geclustert werden. Wenn der Arm der jeweiligen Handfläche auf derselben xy-Ebene wie die Handfläche liegt, wird diese mit der Hand zusammen als eine Region geclustert (siehe Bild 5.3(a)).

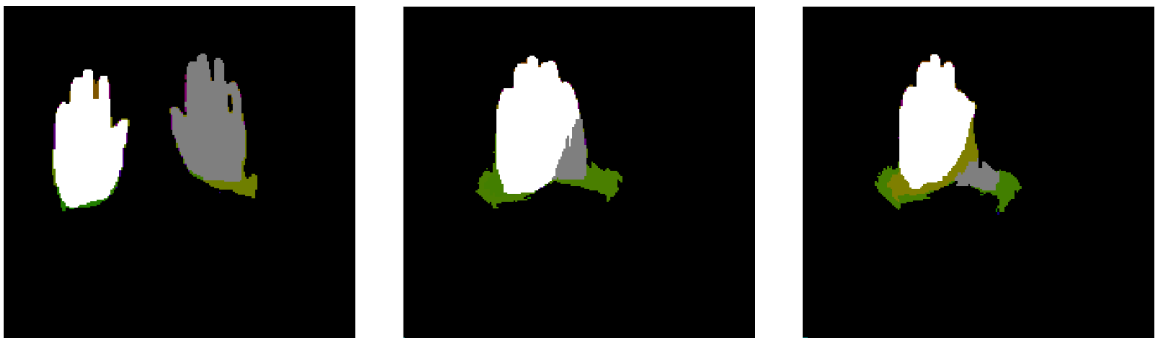
(a) (b)

Bild 5.3: Die gefilmte Person hält beide Hände hoch. Jede Handflächen ist mit dem jeweiligen Arm auf einer Ebene parallel zur Kamera. Die beiden Handcluster sind das graue und das weisse Cluster. In (a) trägt die gefilmte Person einen dunkelbraunen Baumwollpullover. Die Arme werden hier mit zum Handcluster gezählt. In (b) wird eine dunkelbraune matte Lederjacke getragen. Nur die Hände werden hier als Handcluster identifiziert.

Um dies zu vermeiden müssen die Arme entweder auf einer anderen xy-Ebene als die Hände liegen oder eine andere Bekleidung gewählt werden, die eine unterschiedliche Intensität hat, als die Handfläche (sieh Bild 5.3(b))

Eine ähnliche Problematik ereilt sich beim Überkreuzen beider Hände. Wenn während des Überkreuzungsvorgangs beide Hände keinen bestimmten Minimalabstand in z-Richtung einhalten, kann es passieren, dass beide Hände als ein Region geclustert werden. D.h., je näher beide Hände beim überkreuzen sind, umso höher ist die Gefahr, dass beide Hände nicht richtig separat geclustert werden

(siehe Bild: 5.4).
Derartige Fehler beim Clustern können zu Trackingfehlern führen. Dadurch, dass sich der Clusterschwerpunkt der hinteren Hand stark verschiebt, können Cluster mit einem ähnlichen φ-Wert, wie die hintere Hand, als neues Handcluster verwechselt werden, da sie nahe diesem neuen Clusterschwerpunkt liegen. (siehe Bild 5.4(c))



(a) (b) (c)

Bild 5.4: Die Bilder (a)-(c), zeigen die beiden Handcluster während sich die Hände überkreuzen.(b) Da die Hände nahe beieinander sind, wird ein Teil der hinteren Hand zum vorderen Handcluster gezählt. (c) Das Handgelenkcluster der hinteren Hand wird als neues Handcluster gesehen.

Dank der genaueren Berechnung von φ, ist dieser Minimalabstand allerdings kleiner als mit Formel 5.1.

Kapitel 6

Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, ein einfaches System zu entwickeln, welches die automatische Detektion sowie das zeitliche Tracking beider Hände eines Users in einem Stream von Time-of-Flight Tiefeninformationen ermöglicht. Quelle dieser Tiefeninformation ist die Tiefenkamera PMD CamCube. Diese ermöglicht die Echtzeit-Aquisition einer Szene von 204^2 px. Anvisiert war zu Beginn ein einfaches Tracking beider Hände, welches auch umgesetzt wurde. Nachdem beide Hände in der Initialisierung detektiert wurden, werden diese daraufhin getrackt. Als Zusatz wurde zum einfachen Tracking noch die Möglichkeit hinzugefügt, die Hände überkreuzen zu können, sodass eine Hand beim Überkreuzungsvorgang von der Anderen verdeckt werden kann und beim wieder Hervorkommen, vom System detektiert und weiter getrackt werden kann. Die Schwächen dieses Systems sind allerdings, dass das Clustern und Tracken nur dann erfolgreich funktioniert, wenn gewisse Einschränkungen eingehalten werden. Wie in Kapitel 5.2 erwähnt sind die Lage der Arme, die Wahl der Kleidung, sowie ein Minimalabstand der Hände zu anderen Objekten mit ähnlicher Intensität relevant für das Clustern. Auch beim Tracking kann es passieren, dass wenn eine Hand von der anderen verdeckt wird, das Handgelenk als Hand verwechselt werden kann.
Ein Verbesserungsvorschlag zur Stabilisierung des Systems wäre, zu der PMD CamCube, noch eine optische Kamera zu verwenden. Dadurch könnte man die Tiefeninformationen der PMD CamCube und die Farbinformationen der optischen Kamera zum Clustern und Tracken verwenden (ähnlich wie in [SNC10]). Durch Verwendung von Farbinformationen anstatt von Intensitätsinformationen, können ein Großteil der Einschränkungen aufgehoben werden. Die Farbe der Kleidung wäre allerdings dennoch relevant für das Clustern. Ein weiterer Verbesserungsvorschlag wäre, das Tracking durch eine Zustandsübergangs-Vorhersage zu erweitern. Dadurch könnte vermieden werden, dass wenn die eine Hand von der anderen überdeckt wird, das Handgelenk versehentlich als Hand detektiert wird. Durch die Zustandsübergangs-Vorhersage würde gespeichert werden, in welche Richtung beide Hände sich bewegen. Anhand dieser Richtungsangabe würde die Wahrscheinlichkeit ausgerechnet werden, welches Cluster am Wahrscheinlichsten das nachfolgende Handcluster sein könnte.

Literaturverzeichnis

[BVZ01] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE,*, 23(11):1222–1239, November 2001.

[Chi10] Iurie Chiosa. *Efficient and High Quality Clustering*. PhD thesis, Universität Siegen, 2010.

[CK11] Iurie Chiosa and Andreas Kolb. Gpu-based multilevel clustering. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):132–144, 2011.

[GLHL07] S. E. Ghobadi, O. E. Loepprich, K. Hartmann, and O. Loffeld. Hand segmentation using 2d/3d images. In *Proc. of Image and Vision Computing New Zealand*, pages 64–69, Dezember 2007.

[Lin10] Marvin Lindner. *Calibration and Real-Time Processing of Time-of-Flight Range Data*. PhD thesis, Universität Siegen, 2010.

[Rin07] Thorsten Ringbeck. A 3d time of flight camera for object detection. In *Optical 3-D Measurement Techniques*, volume 9, pages 1–10, 2007.

[SNC10] Jonathan R. Schoenberg, Aaron Nathan, and Mark Campbell. Segmentation of dense range information in complex urban scenes. In *The 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2033 – 2038, Taipei, Taiwan, Oktober 2010.

[WLR89] Marc Willbeek-LeMair and Anthony P. Reeves. Region growing on a hypercube multiprocessor. In *C3P Proceedings of the third conference on Hypercube concurrent computers and applications*, volume 2, pages 1033–1042. ACM, 1989.