

Implementierung von Skelettierungs-Algorithmen mit dem Kinect-Sensor

— Projektbericht —

Arbeitsbereich Kognitive Systeme

Fachbereich Informatik

Fakultät für Mathematik, Informatik und Naturwissenschaften

Universität Hamburg

Vorgelegt von:

Johannes Böhler (6349343),
Christopher Kroll (6367171),
Sandra Schröder (6060939)

Hamburg, den 11. März 2013

Inhaltsverzeichnis

1 Einleitung	3
1.1 Aufgabenstellung	4
1.2 Aufbau der Projektarbeit	5
2 Die Kinect	6
3 Die Skelettierung	12
3.1 Skelettierung mittels Thinning	12
3.2 Skelettierung mittels Distanztransformation	20
3.3 Weitere Verfahren	25
4 Implementierung der Algorithmen	27
4.1 Technische Umsetzung	27
4.2 Spielersegmentierung	28
4.3 Skelettierung mittels Thinning	29
4.4 Skelettierung mittels Distanztransformation	30
5 Evaluation	32
5.1 Skelettqualität	32
5.2 Laufzeiten	34
5.3 Distanztransformation - Verbesserung der Skelettqualität	35
5.4 Fazit	41
6 Zusammenfassung und Ausblick	43
Literaturverzeichnis	45
A Quellcode	47
A.1 Skelettierung - Startup	47
A.2 Spielersegmentierung	48
A.3 Skelettierung - Distanztransformation	49
A.4 Skelettierung - Thinning	50
A.5 Verbesserung der Skelettqualität	52
B Übersicht über die Autoren der einzelnen Kapitel und Abschnitte	58

1. Einleitung

Autor: Christopher Kroll

Im Master-Studiengang Informatik an der Universität Hamburg ist ein zweisemestriges Projekt vorgesehen. Die Autoren dieser Arbeit belegten im Sommersemester 2012 und Wintersemester 2012/2013 das Projekt 'Bildverarbeitung' unter der Leitung von Prof. Dr. Leonie Dreschler-Fischer.

Der Verlauf und das Ergebnis dieses Projektes werden in dem vorliegenden Dokument dargestellt.

Die Studenten sollten in diesem Projekt die Daten der Microsoft Kinect-Kamera nutzen und diese für ein Thema ihrer Wahl verarbeiten. Nach einer Vorstellung der Kinect und der Programmiersprache Python wurden die Gruppen eingeteilt und das Thema gewählt (siehe Kapitel 1.1).

Die digitale Bildverarbeitung hilft unter anderem bei der Erkennung von Objekten, wie zum Beispiel bei der vollautomatisierten Qualitätskontrolle. Um auch bei großen Datenmengen noch performant arbeiten zu können, bedient man sich oft des Mittels der Abstraktion. Das Projektthema dieser Arbeit dreht sich um eine Möglichkeit der Abstraktion, nämlich der Skelettierung.

Der Begriff 'Skelett' taucht in vielen Fachbereichen, wie zum Beispiel der Anatomie, Biologie oder Architektur auf. In der digitalen Bildverarbeitung hat er einen ähnlichen Sinn wie in diesen Bereichen: Es stellt von einem Objekt das Grundgerüst dar. Dieses Gerüst ist der zentrale Hauptbestandteil, mit dessen Hilfe sich Rückschlüsse auf das gesamte Objekt (zum Beispiel bei Dinosaurierskeletten) ergeben.

Aus Sicht der Bildverarbeitung kann ein Skelett folgendermaßen definiert werden: Ein Skelett ist ein nützlicher Deskriptor, um Informationen über die Region und den Rand eines Objektes kompakt und effizient zu kodieren und gibt die wesentlichen Grundzüge eines Objektes wieder. In der Informatik kommen noch weitere Eigenschaften des Skeletts hinzu. Das Ziel ist die Information eines Objektes zu reduzieren ohne dabei die Grundstruktur zu verletzen. So ist eine Anforderung, dass die 'Pixelkonnektivität' gewährleistet ist, also alle späteren Skelettpixel mindestens einen benachbarten Skelettpixel besitzen und das Skelett nicht unterbrochen ist.

Bei der Skelettierung muss weiterhin beachtet werden, dass die topologische Struktur des



Abbildung 1.1.: Skelettierung des Buchstabens 'A'. Bildquelle: [Buh08]

Originalbildes nicht verändert wird. Trotz eventueller Verformung bei der Skelettbildung, wie in Abbildung 1.1 dargestellt, muss die strukturelle Eigenschaft erhalten bleiben; so ist bei einer automatisierten Zeichenerkennung die Strichstärke unerheblich, lediglich der generelle Aufbau des Buchstabens ist relevant [Buh08].

Um nun ein Skelett zu erhalten, gibt es verschiedene Ansätze, die sich nach Art der Bestimmung der Skeletpunkte unterscheiden. Die meisten können in eine von drei Kategorien eingeteilt werden: Das Thinning (Ausdünnen), die Distanztransformation und der kritische Punkte Ansatz. Näheres zu den einzelnen Kategorien wird in Kapitel 3 beschrieben.

1.1. Aufgabenstellung

Unsere Gruppe entschied sich für die Datenauswertung einer Person ('Spieler'). Dabei war zunächst das Ziel den Bewegungsablauf bei Sportübungen zu analysieren und eine Rückmeldung zu geben, ob diese richtig ausgeführt wurden. So soll zum Beispiel bei Kniebeugen durch eine Messung des Winkels zwischen Ober- und Unterschenkel der Person eine Hilfestellung gegeben werden.

Um dieses Ziel zu erreichen, muss zunächst der Spieler von anderen Gegenständen im Raum getrennt, also herausgefiltert werden (Segmentierung). Für die Bewegungsanalyse ist es hilfreich, nicht den gesamten Menschen, womöglich noch mit störender Kleidung, zu betrachten, sondern nur sein Skelett. Um das Skelett zu erhalten, bot sich die Wahl zwischen schon implementierten Skelettierungsalgorithmen oder eigenen Implementierungen. Da das Thema des Projektes die Bildverarbeitung und nicht eine Anwendungsprogrammierung ist, fiel die Entscheidung auf die Konzentration auf die Skelettierung. Das Ziel war nun, verschiedene Ansätze zu implementieren und hinsichtlich Qualität und Leistung zu vergleichen.

Dabei mussten einige Kriterien beachtet werden. Zuallererst muss auf die Qualität

des erzeugten Skeletts geachtet werden. Es sollte als eine Skelettrepräsentation des Quellbildes erkannt werden, also ein zentrales Grundgerüst mit vorhandener Pixelkonnektivität. Des Weiteren ist die Leistungsfähigkeit relevant. Die Algorithmen sollen echtzeitfähig sein, damit der Spieler eine unmittelbare Rückmeldung seiner Bewegungen erhält.

1.2. Aufbau der Projektarbeit

Nach dieser Einleitung wird in Kapitel 2 die Microsoft Kinect-Kamera vorgestellt. Ihre einzelnen Komponenten werden beschrieben und Vor- und Nachteile der Kamera dargestellt.

In Kapitel 3 wird näher auf die Skelettierung eingegangen, indem die Grundidee ausgeführt und verschiedene Ansätze zur Erzeugung eines Skeletts aufgezeigt werden. In den Unterkapiteln zu Thinning und Distanztransformation werden verwandte wissenschaftliche Arbeiten, mit denen sich im Seminarteil des Projektes beschäftigt wurde, präsentiert.

Während bis dahin eher die Theorie beleuchtet wurde, geht es im vierten Kapitel um die praktische Umsetzung: Das technische Umfeld, die Spielersegmentierung und die Umsetzung der beiden Skelettierungsmethoden werden hier vorgestellt.

Kapitel 5 evaluiert die Umsetzung, indem die Skelettqualität und die Laufzeit der Algorithmen beleuchtet werden. Außerdem werden hier Überlegungen zur Verbesserung gezeigt.

Im abschließenden Kapitel 6 wird die gesamte Arbeit zusammengefasst und ein Ausblick, wie die Arbeit fortgeführt werden kann, gegeben.

Im Anhang befindet sich der Quellcode zu den erwähnenswerten Stellen der Implementation sowie eine Autorenübersicht zu den einzelnen Kapiteln, bzw. Abschnitten dieser Arbeit.

2. Die Kinect

Autor: Johannes Böhler

Die drei Haupt-Hardware-Komponenten der Kinect sind ein Infrarotprojektor, eine RGB-Kamera sowie eine Infrarotkamera.

Die Kombination aus Infrarotstrahler und Infrarotkamera ermöglicht die Gewinnung



Abbildung 2.1.: *Links:* Infrarotprojektor *Mitte::* RGB-Kamera *Rechts::* Infrarotkamera.
Bildquelle:[Bor12]

von Tiefeninformationen aus der Umgebung. Im Gegensatz zu gewöhnlichen Kameras, welche einem Pixel Farbinformation (z.B. über RGB-Farbkanäle) zuordnen, wird dem Pixel mit Hilfe der Infrarot Kamera eine Entfernung zugeordnet.

Diese ergibt sich aus der Art und Weise wie der Infrarotstrahl von dem durch den Pixel repräsentierten Bereich eines Objektes reflektiert wird.

Die Tiefenbilder welche man von der Infrarotkamera erhält, sehen aus wie komplett verrauschte Graustufenbilder. Hierbei steht jeder Grauwert eines Pixels für die entsprechende Entfernung des korrespondierenden Objektausschnittes zur Kinect.

Hohe Grauwerte (helle Pixel) repräsentieren nahe Objekte, während niedrige Grauwerte (dunkle Pixel) weiter entfernte Objekte beschreiben. In einem Tiefenbild ist die gesamte Information über die Entfernung der im Bildausschnitt erfassten Objekte zur Kinect enthalten.

Wird die Tiefeninformation dazu genutzt um die Pixel im dreidimensionalen Raum anzugeben, erhält man eine 3d-Punktwolke. Die in der 2d-Betrachtung benachbarte Pixel müssen in der 3d-Präsentation nicht miteinander verbunden sein, da die Z-Koordinaten unterschiedliche Werte aufweisen können.



Abbildung 2.2.: Tiefeninformation der Umgebung codiert in Grauwerten.
Bildquelle:[Bor12]

2.0.1. Funktionale Komponenten

Infrarotprojektor

Der Infrarotprojektor emittiert elektromagnetische Strahlen, deren Wellenlänge (830nm) außerhalb des für den Menschen sichtbaren Bereichs (380nm-780nm) liegt [Bor12]. Der Projektor strahlt zur Tiefenbestimmung ein Gitter von Infrarotpunkten (strukturiertes Licht) auf die Objekte in seiner Umgebung ab. Zur Generierung dieses Musters wird ein besonderes Verfahren implementiert. Normalerweise produzieren Filter, die diese Art von Muster generieren einen sehr hellen Punkt in der Mitte des Bildes, welcher die Leistungsfähigkeit des Infrarotprojektors limitiert. Durch das hier verwendete Verfahren entstehen statt einem einzigen sehr hellen, neun helle Punkte, welche durch unvollständige Lichtfilterung zur Mustererstellung entstehen[KBC⁺12]. Das hier eingesetzte Verfahren produziert weniger starke Artefakte und ermöglicht die Verwendung einer leistungsfähigeren Diode. Dies führt zu einer höheren Genauigkeit der Abstandsauflösung und zu einer größeren Reichweite im Bezug auf die Umgebung in welcher die Entfernungsmessung durchgeführt wird. Die Reichweite des Infrarotprojektors ist dennoch eingeschränkt, da zu hohe Intensitäten der Infrarotstrahlen Augenschäden verursachen könnten.

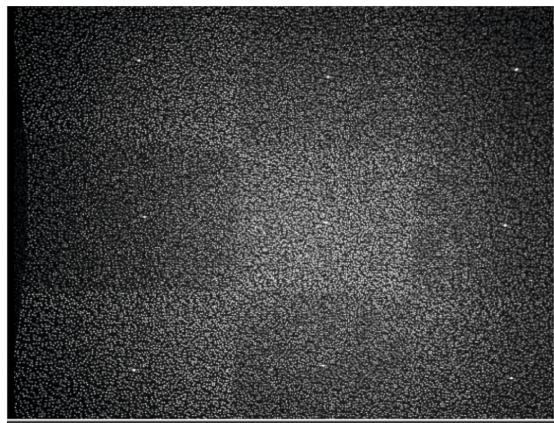


Abbildung 2.3.: Generierung des Infrarot Random-Patterns (strukturiertes Licht) mit Hilfe von 9 Bereichen Bildquelle:[KBC⁺12]

Infrarotkamera

Die Infrarotkamera nimmt Bilder mit einer nativen Auflösung von 1280x1024 Pixeln, bei einer Bildwiederholrate von 30 Hz auf. Weitergeleitet werden allerdings nur Bilder mit einer Auflösung von 640x480 Pixeln, da der USB Datenbus eine Limitierung bezüglich der Datenübertragungsrate darstellt[Bor12]. Das Blickfeld der Infrarotkamera beträgt in der Horizontalen 58 °, in der Vertikalen 45 °. Damit die Mustererkennung funktioniert, ist ein Mindestabstand von 0,8 Metern erforderlich [Bor12].

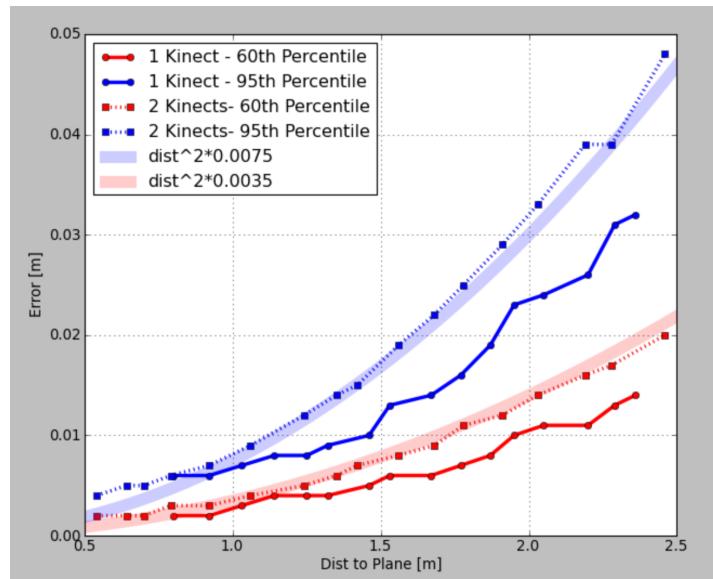


Abbildung 2.4.: Fehlerbehaftung der Tiefeninformation in Abhängigkeit von der Objektentfernung. Bildquelle: [kin]

Ab einer Entfernung von 3,5 Metern wird die Intensität der reflektierten Strahlen zu gering um Tiefeninformationen mit ausreichender Aussagekraft und Präzision zu erhalten. Bei einem optimalen Abstand von 2 Metern zum Objekt beträgt die Auflösung in der XY-Ebene 3 mm, in der Z-Ebene 1 cm [Bor12].

Die Quantisierungsauflösung liegt bei 2^{11} (2048) Bit [mic]. Es muss sichergestellt werden, dass die Infrarotkamera nur die erwünschte elektromagnetische Strahlung im 830 nm Bereich aufnimmt und nicht von Strahlen anderer Wellenlänge gestört wird. Dies wird durch einen Filter, welcher auf dem Infrarot-Kamera-Objektiv angebracht ist realisiert. Trotz Verwendung des Filters sollte die Kinect in abgedunkelten Innenräumen verwendet werden, da Sonnenlicht auch elektromagnetische Wellen im Infrarotbereich enthält.

RGB Kamera

Die RGB Kamera nimmt bei einer Wiederholrate von 30 Hz ebenfalls mit einer Auflösung von 640x480 Pixeln auf, könnte jedoch auch mit einer Auflösung von 1280x1024 Pixeln bei einer reduzierten Wiederholrate von 15 Bildern pro Sekunde angesteuert werden. Die Quantisierungsauflösung der Kamera liegt bei 2^8 (256) Bit [mic].

Mikrofon Array

Die Kinect beinhaltet vier Mikrofone, welche verteilt verbaut sind. Sie dienen sowohl der Erfassung von Ton, als auch der Lokalisierung und Unterscheidung von Soundquellen. Hierdurch ist es möglich, unterschiedliche Stimmen mehrerer Spieler eindeutig ihren jeweiligen Avataren zuzuweisen. Jedes der Mikrophone tastet mit einer Quantisierungsauflösung von 2^{16} (65536) Bit und einer Abtastrate von 16 KHz ab [mic].

2.0.2. Tiefenberechnung

Die Berechnung der Objektentfernungen (Tiefenwerte) innerhalb der Kinect erfolgt durch Aussendung von strukturiertem Licht durch die Projektionseinheit (siehe Abschnitt *Infrarotkamera*) und durch Weiterverarbeitung der durch die Infrarotkamera empfangen reflektierten Strahlen. Das Muster für das strukturierte Licht wird mit Hilfe eines Diffusors (eine Lochplatte mit fest definiertem Muster) erzeugt.

Das Prinzip des strukturierten Lichts ist an ein Verfahren angelehnt, welches sich Streifenprojektion nennt und in dieser Anwendung modifiziert wurde, um die Erfassung von beweglichen Objekten zu ermöglichen. Anstatt von Lichtstreifen wird eine Punktematrix verwendet, welche zufällig und fest definiert ist. Die Infrarotkamera sowie der Projektor müssen sich hierzu in einem vordefinierten, gleich großen Abstand zueinander befinden. Die Umgebungsreflektionen der projizierten Punktematrix werden von der Infrarotkamera erfasst. Um aus diesem Gitter von reflektierten Infrarotpunkten Tiefeninformation zu extrahieren, wird das Verfahren der aktiven Stereotriangulation verwendet.

Die Triangulation ist ein Verfahren zur optischen Abstandsmessung, welches sich hierzu

trigonometrischer Funktionen innerhalb von aufgespannten Dreiecken bedient. Es wird allgemein zwischen aktiver und passiver Triangulation unterschieden. Aktive Triangulation bedingt mindestens eine strukturierte Lichtquelle zur Abstandsberechnung, während dies bei passiver Triangulation nicht der Fall ist[tri].

Da der Infrarotprojektor ein statisches Pseudozufallsmuster emittiert, ist dieser als strukturierte Lichtquelle einzuordnen. Stereotriangulation bedeutet dass zwei unterschiedliche Bildquellen benötigt werden um die Tiefe jedes Pixels eines Bildausschnittes berechnen zu können. Eine der zwei Bildquellen ist der Diffusor (das „Lochmuster“), welcher die vom Projektor emittierten Strahlen statisch definiert. Die andere Bildquelle ist die Infrarotkamera. Das "Bild"(die Lochmaske) des Projektors ist immer identisch und statisch. Das Bild der Infrarotkamera hingegen variiert je nach Umgebung. Diese beiden Bilder sind die Grundlage für die trigonometrischen Operationen zur Berechnung der Tiefeninformationen [?].

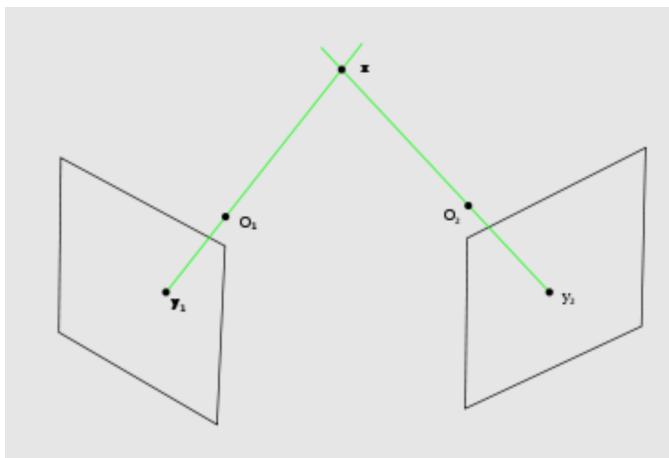


Abbildung 2.5.: Stereotriangulation Bildquelle:[ste]

Zur Gewinnung der Tiefeninformation wird die horizontale Differenz des Punktes Y_1 des von der Infrarotkamera aufgenommenen Bildes zum korrespondierenden Punkt Y_2 des virtuellen, statischen Referenzbildes des Projektors berechnet. Aus dieser Differenz lässt sich die Tiefe des betreffenden Pixels durch aufstellen der beiden Projektionslinien und Schnittpunktbestimmung derselben berechnen.

Der Grund weshalb die Pixel der statischen „Schablone“ im Projektor zufällig angeordnet sind, liegt darin, dass die unterschiedlichen lokalen Nachbarschaftsbedingungen die Pixelzuordnung zwischen dem statischen und dem dynamisch veränderten Bild erleichtern [pri].

2.0.3. Das Schattenproblem

Aufgrund der Entfernung der verbauten RGB-Kamera zur Infrarotkamera, weisen die Bilder beider Kameras einen kleinen Versatz auf. Schatten im Tiefenbild entstehen aufgrund der Entfernung des Infrarotprojektors zur Infrarotkamera.

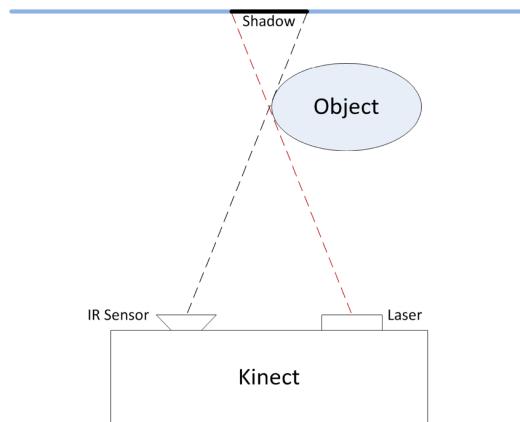


Abbildung 2.6.: Blockierte Infrarotstrahlen.Bildquelle:[AMAA]

Der Schatten im Muster macht es für den Sensor unmöglich die Tiefe festzustellen. Die Pixel in diesen Bereichen werden stattdessen auf den Wert 0 gesetzt. Das Objekt blockiert die Strahlen des Lasers. Da für die Tiefenberechnung das von der Umgebung reflektierte Muster benötigt wird, ist es für die Kinect unmöglich die Distanz in Bereichen zu berechnen, welche außerhalb der Erreichbarkeit des Infrarot-Strahlenmusters liegen.

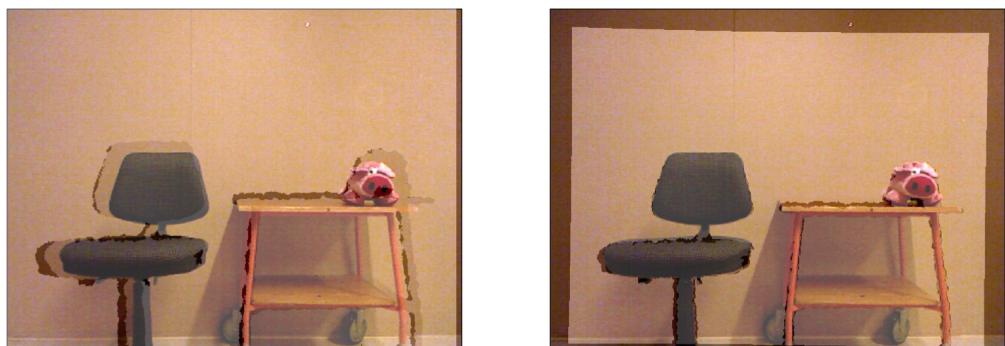


Abbildung 2.7.: Da die Infrarotkamera links des Projektors positioniert ist, treten die Schatten auch linksseitig der Objekte auf. Bildquelle:[AMAA]

3. Die Skelettierung

Autor: Sandra Schröder

Unter einem Skelett versteht man einen Deskriptor, der die topologischen Eigenschaften eines Objekts beschreibt. Das sind Merkmale, die sich nicht explizit auf die konkrete Form einer Region beziehen, sondern auf ihre strukturellen Eigenschaften, die auch bei starken Verformungen erhalten bleiben. Zur Erfassung von bestimmten Eigenschaften von Binärbildern vereinfachen Skelette die Analysen, da durch die Reduzierung der Daten unwesentliche Eigenschaften ausgeblendet werden. Diese Abstraktion von für die Anwendung unwichtigen Eigenschaften ist eine zentrale Eigenschaft des Skelettes beziehungsweise eines Deskriptors im Allgemeinen. Historisch sind im Laufe der Entwicklung diverser Skelettierungsalgorithmen immer mehr konkrete Anforderungen an Skelette entstanden.

Eine wichtige Anforderung ist die *Konnektivität* des Skeletts. Dies bedeutet, dass es keine Lücken und Unterbrechungen im Skelett gibt, denn ein zusammenhängendes Objekt sollte ein zusammenhängendes Skelett besitzen. Man spricht bei einem pixelweisen Zusammenhang im Skelett von *Pixelkonnektivität*.

Viele Skelettierungsverfahren fordern, dass ein Skelett genau ein Pixel breit ist. Dies ist beispielsweise in der Datenkomprimierung wichtig. Möchte man die Struktur eines Objektes mit möglichst wenig Pixeln speichern, genügt ein Skelett mit der minimal möglichen Pixelbreite. Eine breitere Skelettlinie könnte unter Umständen zu viele unnötige Informationen beinhalten. Des Weiteren sollte das Skelett zentriert im Objekt liegen. Das bedeutet, dass Abstand der Pixel der Skelettlinien nach links und nach rechts möglichst gleich sein sollte.

In wie weit alle Anforderungen an das Skelett erfüllt sein müssen, ist allerdings für jeden Anwendungsfällen neu zu diskutieren. Dieses Kapitel beschreibt bekannte Konzepte und Verfahren zur Skelettierung im Bereich der Bildverarbeitung. Zwei Verfahren wurden im Rahmen der Projektarbeit genauer untersucht: Die Skelettierung mittels *Thinning* und mittels *Distanztransformation*.

Um die Übersicht über die weiteren grundlegenden Verfahren und Konzepte zu vervollständigen, werden diese in einem separaten Abschnitt kurz vorgestellt.

3.1. Skelettierung mittels Thinning

Autor: Johannes Böhler, Christopher Kroll

Das Thinning, bzw. Ausdünnen ist das älteste und wohl bekannteste Verfahren der

Skeletttgenerierung, wodurch diese beiden Begriffe oft synonym verwendet werden[Buh08]. Es bezeichnet eine Kategorie von Methoden zur Skelettierung von 2d- sowie 3d-Objekten. In dieser Projektarbeit ist der Fokus ausschließlich auf die 2d-Skelettierung gerichtet, da die Kinect kein vollkommenes 3D Modell eines Objektes liefert. Sie erfasst das Objekt lediglich aus einem Blickwinkel, desshalb erhält man nur ein 2,5-dimensionales Modell. Es werden nur Tiefeninformationen bezüglich der Seite des Objektes, welche der Kinect zugewendet ist bereitgestellt. Die Tiefeninformationen der Rückseite bleiben verborgen. Um ein vollständiges 3d-Modell zu erhalten müssten mindestens 2 Kinects verwendet, sowie die Informationen beider Geräte zusammengeführt und vereinheitlicht werden.

Alle Thinning-Algorithmen verbindet das iterative Abtragen des Musters oder der Oberfläche. Dabei werden nach und nach die Konturpixel, also die Objektpixel, die mit den Hintergrundpixeln benachbart sind, überprüft. Je nach Algorithmus liegen verschiedene Kriterien vor, ob diese gelöscht, also als Hintergrundpixel markiert werden können. Ein Kriterium ist, dass die Zusammengehörigkeit der Skelettpixel bewahrt wird. Eine weitere Gemeinsamkeit der unterschiedlichen Ansätze ist, dass das Ausdünnen solange stattfindet, bis am Objekt keine Änderung mehr vorgenommen wurde.

Beim Thinning wird die Zusammengehörigkeit nicht verändert, wodurch die topologische Struktur des Objektes erhalten bleibt.

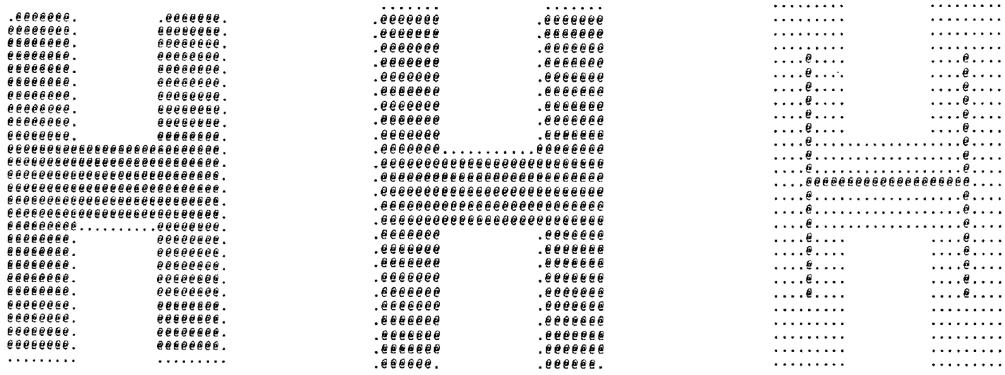
Das Verfahren lässt sich in zwei Klassen aufteilen:

Während beim sequentiellen Ausdünnen der Zustand des Bildes von allen bisherigen Iterationen unterschieden wird, betrachtet das parallele Ausdünnen lediglich die vorherige Iteration, um die Entscheidung der Pixellösung zu treffen. Der Vorteil bei letzterem Verfahren ist die Möglichkeit der parallelen Lösung, da alle Pixel unabhängig voneinander bearbeitet werden können, wobei oft mit Unterarationen gearbeitet wird[Buh08].

3.1.1. A Fast Parallel Algorithm for Thinning Digital Patterns

Autor: Johannes Böhler

Der Abschnitt bezieht sich auf das Paper *A Fast Parallel Algorithm for Thinning Digital Patterns* von T. Y. ZHANG und C. Y. SUEN und erläutert das beschriebene Verfahren zur Generierung eines Skeletts mit Hilfe von Thinning. Der gesamte Algorithmus erstreckt sich über mehrere Iterationen, dabei wird je Iteration eine Schicht (Menge an Randpixeln) des Musters abgetragen sofern bestimmte Kriterien erfüllt werden. Die Iterationen selbst, sind wiederum in zwei Subiterationen unterteilt. Das Abtragen der „Schichten“ wird somit in zwei unterschiedliche Phasen aufgespalten. Mit Hilfe der ersten Subiteration werden sowohl Süd- und Ostgrenzpunkte als auch Nordwest-Eckpunkte entfernt. Das entfernen von Nord- und Westgrenzpunkten sowie von Südosteckpunkten erfolgt in der zweiten Subiteration.



(a) Nach erster Subiteration (b) Nach zweiter Subiterati- (c) Skelett des Ursprungs-
zu Beginn on (zu Beginn) muster

Abbildung 3.1.: Zustände des Algorithmus
Bildquelle: [ZS84]

Anforderungen an den Algorithmus

- Das Rauschen, welches der Algorithmus verursacht soll so gering wie möglich sein.
- Das Skelett des Ursprungsmusters soll die Endpunkt- und Pixelverbundenheit erhalten. Endpunktverbundenheit bedeutet, dass sich zwischen zwei Endpunkten eines Skeletts keine unverbundenen Stellen befinden.
- Das Skelett soll nach Durchlaufen des kompletten Algorithmus in einer einheitlichen Dicke von einem Pixel vorliegen.
- Der Algorithmus soll möglichst schnell und effizient arbeiten um Echtzeitfähigkeit gewährleisten zu können.

Ablauf des Algorithmus

Es wird davon ausgegangen, dass zu Beginn ein binär digitalisiertes Bild vorliegt. Die Pixel werden mit Hilfe einer zweidimensionalen Matrix IT durchlaufen, deren Wert an der jeweiligen Stelle $IT(i,j)$ entweder 0 oder 1 ist. Mit Muster ist im Folgenden die Menge an Pixeln gemeint, welche den Wert eins haben. Es werden in Abhängigkeit von den 8 Nachbarpixeln (siehe Abbildung 3.2), Transformationen auf den betrachteten Pixel P_1 angewendet. Der Wert des des Pixels P_1 wird bei Erfüllung bestimmter Kriterien welche im Folgenden erläutert werden auf 0 gesetzt. Dieser Vorgang wird iterativ auf die Matrix IT angewendet. Nach einmaligem Durchlaufen der Matrix IT ist eine Subiteration abgeschlossen.

Der neue Wert eines Pixels während der n -ten Iteration hängt von dem eigenen Wert während der $(n-1)$ -ten Iteration und den Werten der acht Nachbarn während der $(n-1)$ -ten Iteration ab. Dies ermöglicht paralleles Transformieren mehrerer Bildpunkte zur selben

Zeit. Die Bedingungen, welche zum Ausführen der Transformation erfüllt sein müssen werden über ein 3x3 Pixel Fenster abgefragt. Der Punkt P1 über dessen Transformation entschieden wird, ist mit allen acht Nachbarn direkt verbunden.

P_9 $(i - 1, j - 1)$	P_2 $(i - 1, j)$	P_3 $(i - 1, j + 1)$
P_8 $(i, j - 1)$	P_1 (i, j)	P_4 $(i, j + 1)$
P_7 $(i + 1, j - 1)$	P_6 $(i + 1, j)$	P_5 $(i + 1, j + 1)$

Abbildung 3.2.: Betrachteter Pixel P1 und Nachbarumgebung. Bildquelle: [ZS84]

Der Algorithmus entfernt alle Randpunkte des Musters, außer den Pixeln welche Bestandteil des Skeletts sind. Um die Verbundenheit des Skeletts zu gewährleisten wird ein Iterationsschritt in zwei Subiterationen aufgeteilt.

In der ersten Subiteration wird der Punkt P1 aus dem Muster gelöscht, wenn er folgende Bedingungen erfüllt:

- a) $2 \leq B(P1) \leq 6$ B entspricht der Anzahl der Nachbarn von P1 welche ungleich 0 sind. Die Anzahl der Nachbarn von P1 welche den Wert 1 haben, muss somit zwischen 2 und 6 liegen.
- b) $A(P1)=1$ Anzahl der „01“-Folgen Die Anzahl der 01 Folgen in der geordneten Folge P2,P3...P9 muss genau eins betragen.
- c) $P2 * P4 * P6 = 0$ Mindestens ein Pixel der Pixelmenge P2, P4, P6 muss den Wert Null haben.
- d) $P4 * P6 * P8 = 0$ Mindestens ein Pixel der Pixelmenge P4, P6, P8 muss den Wert Null haben.

Sind alle Bedingungen a, b ,c und d erfüllt so wird der Wert des Pixels auf 0 gesetzt. Dies bedeutet dass er kein Teil des Skelett-Musters mehr ist. Wird eine der Bedingungen nicht erfüllt, so bleibt der Pixelwert bei 1.

In der zweiten Subiteration wird P1 gelöscht falls folgende Bedingungen gelten:

- a) $2 \leq B(P1) \leq 6$
- b) $A(P1)=1$
- c) $P2 * P4 * P8 = 0$

0	0	1
1	P_1	0
1	0	0

Abbildung 3.3.: Bedingung B: Anzahl 01 folgen in zyklischer Reihenfolge. Bildquelle: [ZS84]

- d) $P_2 * P_6 * P_8 = 0$

Nur die Bedingungen c und d haben sich im direkten Vergleich zur ersten Subiteration verändert.

Um die Bedingungen der ersten Subiteration zu erfüllen, muss $P_4 = 0$ oder $P_6 = 0$ oder ($P_2 = 0$ und $P_8 = 0$) erfüllt sein. Dies impliziert dass P_1 entweder Süd- oder Ost-Grenzpunkt, oder Nordwesteckpunkt ist. Um die Bedingungen der zweiten Subiteration zu erfüllen muss $P_2 = 0$ oder $P_8 = 0$ oder ($P_4 = 0$ und $P_6 = 0$) sein. P_1 ist somit Nord- oder West-Grenzpunkt oder Südosteckpunkt.

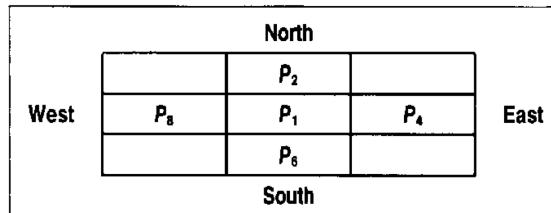


Abbildung 3.4.: Betrachteten Nachbarpunkte in den Bedingungen c und d. Bildquelle: [ZS84]

Während mit Bedingung A ($2 \leq B(P_1) \leq 6$) die Endpunkte des Skeletts erhalten werden, so wird mit Bedingung B ($A(P_1) = 1$) die Auslöschung von Punkten zwischen den Endpunkten der Skelettlinie verhindert.

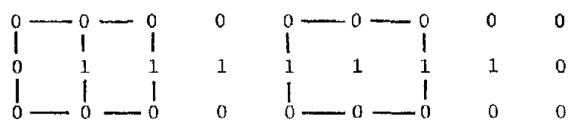


Abbildung 3.5.: Gewährleistung der Pixelverbundenheit. Bildquelle: [ZS84]

Der Algorithmus im Überblick

Diese Abschnitt nimmt Bezug auf Abbildung 3.6, und stellt den Algorithmus im Gesamtzusammenhang dar. In der Matrix Search M befinden sich während der ersten Iteration alle Pixel die gelöscht werden dürfen, da sie den Bedingungen der ersten Iteration genügen. Ist dies nicht der Fall, so ist der Counter gleich Null und der Algorithmus beendet, da es keine zu löschen Pixel mehr gibt. Die Skelettierung ist somit beendet.

Falls der Counter ungleich Null ist, werden die Pixel welche den Bedingungen genügen von der Matrix IT (Skelett-Muster) abgezogen, der Counter wird Null gesetzt und es wird zur zweiten Iteration fortgeschritten. Dort findet der Ablauf mit veränderten Bedingungen c und d wiederholt statt. Ist der Counter auch nach dem Durchlaufen der zweiten Subiteration ungleich Null so wird der Vorgang iterativ fortgeführt bis der Counter am Ende einer Subiteration den Wert Null hat und damit das Ende der Skelettierung bezeichnet.

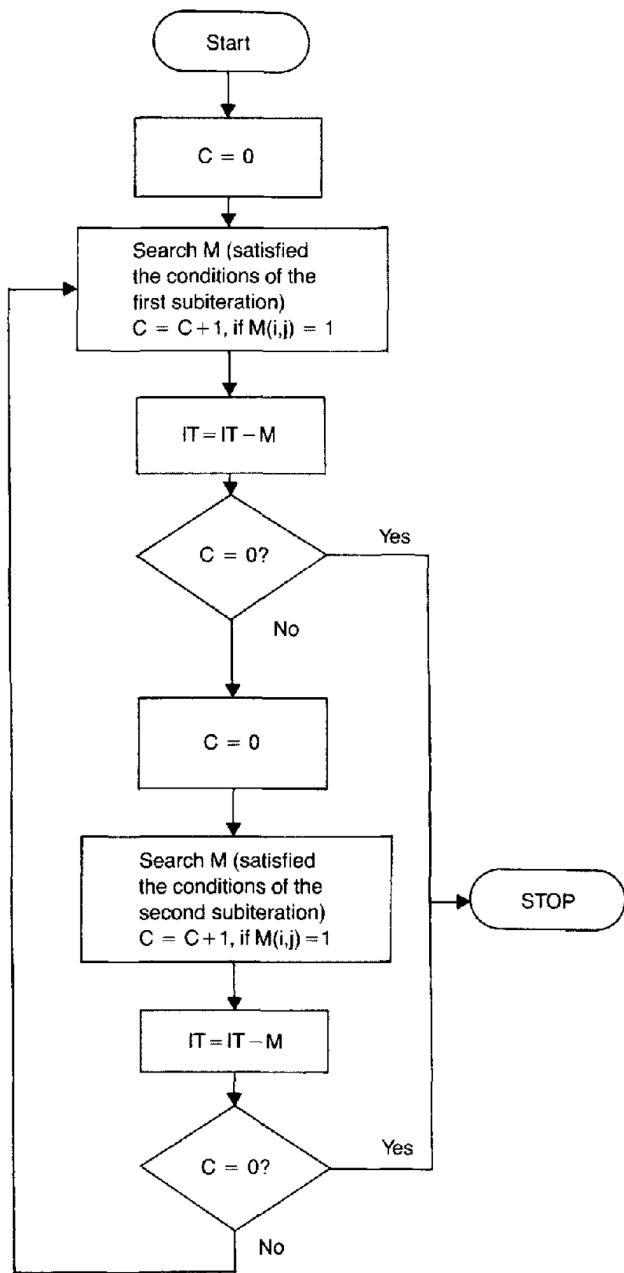


Abbildung 3.6.: Gesamter Algorithmus in der Übersicht. Bildquelle: [ZS84]

Resultate

- Der Algorithmus erzielt sehr gute Ergebnisse in Bezug auf Verbundenheit und Rauschverhalten der Randpunkte.
- Die Bedingungen welche zum Auffinden der zu löschen Randpunkte führen sind sehr simpel.

- Durch den Bezug auf die n-1te Iteration zur Abfrage der Bedingungen werden Warteabhängigkeiten vermieden und ein performantes Verarbeiten der Bildpixel gewährleistet.

Nach Betrachtung des Algorithmus wurde die Entscheidung getroffen, diesen vor allem aufgrund der Kriterien Performanz und Implementierbarkeit umzusetzen.

3.2. Skelettierung mittels Distanztransformation

Autor: Sandra Schröder

Die Distanztransformation eines Binärbildes enthält Informationen über den Abstand der Objektpixel zum Hintergrund. Der Abstand zum Hintergrund wird für jeden Objektpixel bestimmt und in einem weiteren Bild (gleiche Dimension und Größe wie das Binärbild) als Grauwert an der Stelle des Objektpixels gespeichert. Das Ergebnis ist die sogenannte *Distance Map* (Abbildung 3.7).

Zur Bestimmung des Abstands benötigt man Metriken. Eine gängige Metrik - die für den Algorithmus im Rahmen dieser Arbeit auch genutzt wurde - ist die euklidische Metrik d_2 :

$$d_2(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} \quad (3.1)$$

Die Definition einer Nachbarschaft spielt ebenfalls eine Rolle. Wählt man eine 4er-Nachbarschaft, wird der Abstand vom Objektpixel zur linken und zur rechten Seite, sowie nach oben und nach unten berechnet. Man erhält dementsprechend vier Abstände. In der Distance Map wird der minimale Abstand von den vier Werten gespeichert. Bei einer 8er-Nachbarschaft kommen die diagonalen Richtungen dazu. Abhängig von der gewählten Nachbarschaft und der Metrik ergeben sich unterschiedliche Distance Maps.

Die Pixel mit dem größten Abstand zum ersten Pixel des Hintergrunds sind die hellsten Pixel in der Distance Map. Die umgebenden Pixel haben einen kleineren Grauwert und der Grauwert der Pixel verringert sich, je näher die Pixel am Hintergrund liegen. Dementsprechend beschreiben die hellsten Pixel des *Grauwertgebirges* eine skelettförmige Struktur, die im Folgenden aus der Distance Map extrahiert werden soll. Wir verfolgen einen Ansatz, der auf der Bildung des Gradienbetrages der Distance Map beruht.

Ein Graubild kann als skalare Funktion $f(x, y)$ beschrieben werden mit $x, y \in \mathbb{N}$ und



Abbildung 3.7.: Beispiel einer Distance Map. Links: Originalbild. Dieses wurde zuerst invertiert, damit das Objekt (Person) weiß markiert ist. Rechts: Resultat der Distanztransformation. Aufällig ist das Maximum in der Mitte.

$0 \leq f(x, y) \leq 255$. Die Idee ist, den Gradienten beziehungsweise den Gradientenbetrag der Distance Map zu bestimmen. Der Gradient (Gleichung 3.2) ist ein Differentialoperator

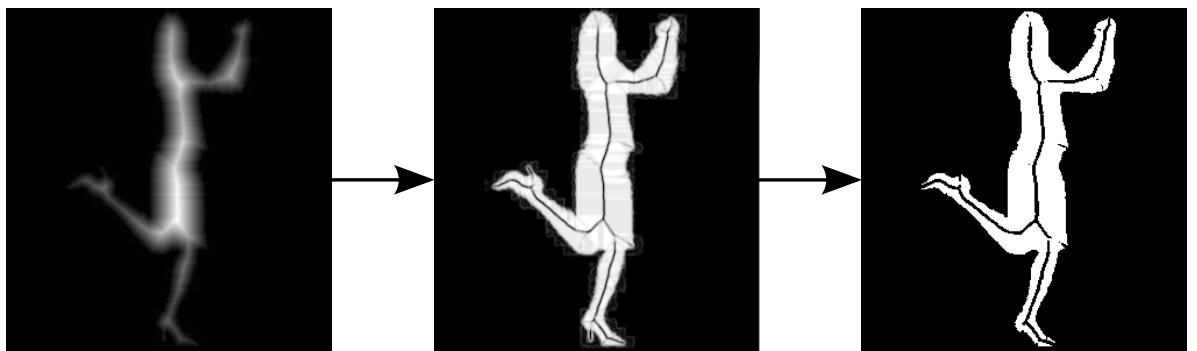


Abbildung 3.8.: Segmentierung des Gradientenbetrags aus der Distance Map. Von links nach rechts: Distance Map, Gradientenbetrag der Distance Map, Segmentiertes Gradientenbetragsbild

und liefert, angewandt auf ein Skalarfeld, die Richtung des stärksten Anstiegs, sowie die Amplitude des Anstiegs (Gradientenbetrag):

$$grad(f(x, y)) = \nabla f(x, y) = \begin{bmatrix} \frac{df}{dx} \\ \frac{df}{dy} \end{bmatrix} \quad (3.2)$$

Entsprechend der Definition des Gradienten, ist an den lokalen Maxima des Grauwertgebirges der Gradientenbetrag gleich null.

Speichert man den Gradientenbetrag ebenfalls als Grauwertbild mit gleicher Dimension und Größe wie die Distance Map, wird der Gebirgskamm als Linie mit kleinen Grauwerten (fast Null) kodiert. Diese markiert das Skelett. Eine schwellwertbasierte Segmentierung des Gradientenbetragbildes bewirkt, dass die Skelettlinien schwarz sind (Abbildung 3.8). Andere Bereiche, die größer als der Schwellwert sind und somit nicht zum Skelett gehören, erhalten den Grauwert 255 (weiß).

Um endgültig nur die Skelettlinie zu erhalten, wird die Differenz zwischen der Distance Map und dem segmentierten Gradientenbetragsbild gebildet. Der Teil der Distance Map der nicht auf der höchsten Stelle des Grauwertgebirges liegt, hat entweder einen Grauwert kleiner oder gleich 255. Bei der Differenzbildung kann der Grauwert dieser Pixel nur kleiner gleich 0 werden, da von diesen Grauwerten der Grauwert 255 abgezogen wird (weißer Bereich des segmentierten Gradientenbetragsbildes). Negative Grauwerte werden auf 0 gesetzt. Als Ergebnis erhält man die Skelettlinie mit Grauwerten ungleich 0. Um ein binärkodiertes Skelett zu erhalten, wird auf dem Differenzbild anschließend eine schwellwertbasierte Segmentierung ausgeführt (Abbildung 3.9). Im Folgenden wird das aus der Distanztransformation bestimmte Skelett zur Abgrenzung zum Thinning-Algorithmus als *Distanzskelett* bezeichnet.

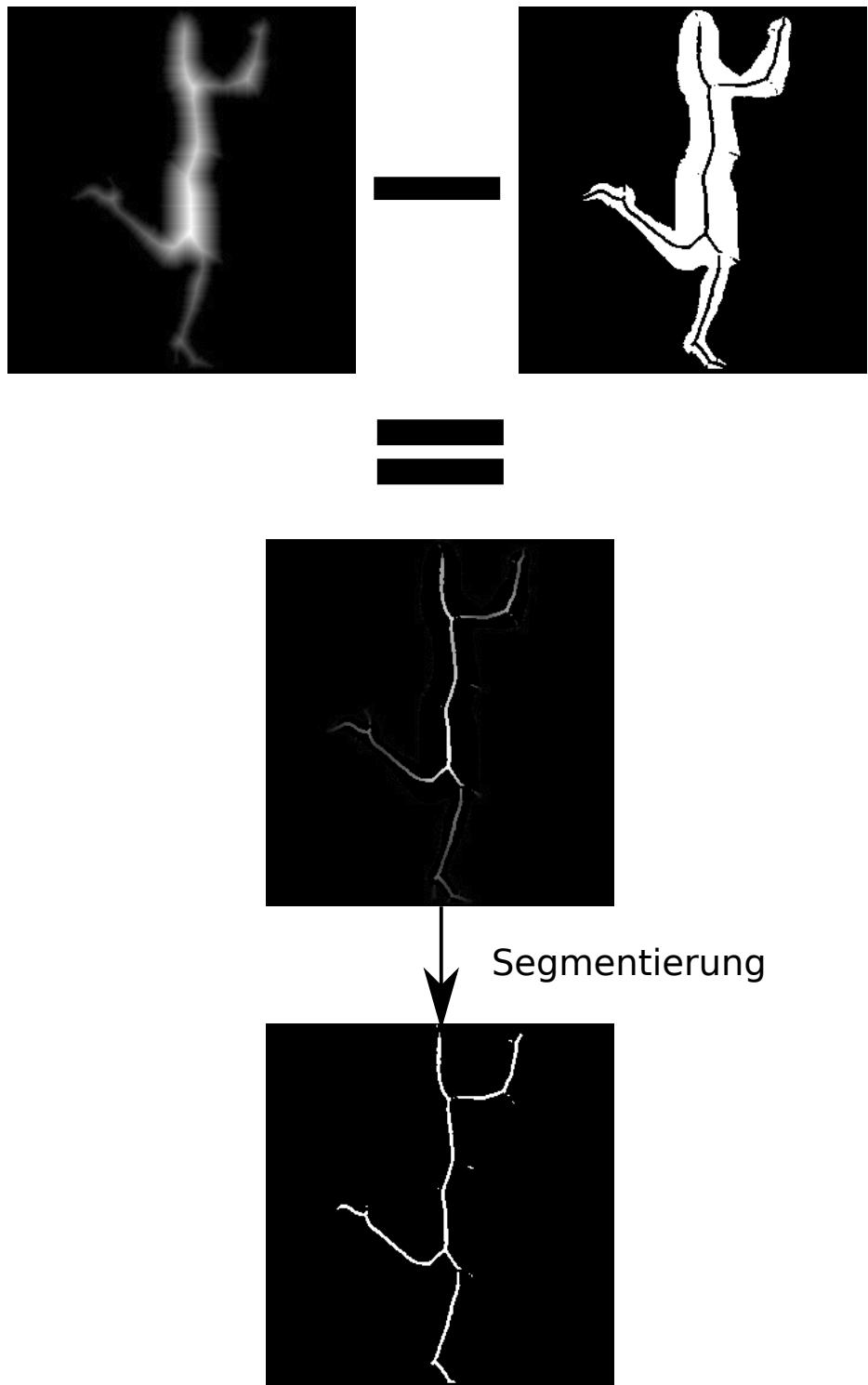


Abbildung 3.9.: Differenzbildung zwischen dem segmentieren Gradientenbetragsbild und der Distance Map.

3.2.1. Verwandte Arbeit [Cha07]: Extracting Skeletons From Distance Maps

Autor: Sandra Schröder

Zur oben beschriebenen Berechnung des Skelettes aus einer Distance Map gibt es verwandte Arbeiten, von denen nun eine vorgestellt wird. In dem Paper *Extracting Skeletons from Distance Maps* beschreibt der Autor einen Algorithmus, der effizient und schnell aus einem distanztransformierten Bild ein Skelett extrahiert [Cha07]. Der Autor legt besonders hohen Wert darauf, dass die Extraktion keine komplizierten Berechnungen beinhaltet. Er möchte vor allem auf die Berechnung von Ableitungen höherer Ordnung und die Auswertung von komplexen Gleichungen verzichten. Das Verfahren, welches der Autor zur Extraktion der Skelettlinien eines Objekt vorstellt, ist die sogenannte *Ridge Point Detection* (deutsch: *Gebirgskammdetektion*). Dabei nutzt der Autor eine grundlegende Eigenschaft der Distance Map. Wie in Abschnitt 3.2 beschrieben, ist die Distance Map ein Grauwertgebirge, wobei der Gebirgskamm zentriert im Objekt liegt. Betrachtet man nur diesen Teil der Distance Map und projiziert ihn auf das Originalbild, ist eine skelett-artige Beschreibung des Objekts zu erkennen.

Die Gebirgskammdetektion ist ein gradientenbasiertes Verfahren. Der Gradient zeigt nach Definition in die Richtung des stärksten Anstiegs (Gleichung 3.2). Dies bedeutet, dass der Gradient eines Punktes, der nicht auf dem Kamm liegt, in die Richtung des Kamms zeigt. Wählt man nun Punkte näher am Kamm, so wird der Anstieg geringer, da die Differenz zwischen dem Grauwert des betrachteten Gebirgspunktes und dem aktuell gewählten Punkt kleiner wird. Überquert man den Kamm, kehrt sich die Richtung des Gradienten um und zeigt wieder zum Kamm. Hier findet ein Vorzeichenwechsel des Gradientenbetrags statt. Der Punkt auf dem Gebirgskamm bildet eine *sign barrier* zwischen den Gradientenbeträgen der sich gegenüberliegenden Punkte, wobei sich der dieser Punkt zwischen den beiden Punkten befindet. Diese Beobachtung ist in Abbildung 3.10 dargestellt. Es ist ein Querschnitt eines Grauwertgebirges abgebildet. Legt man nun eine Projektionslinie genau durch das lokale Maximum (roter Punkt), und projiziert die Gradienten (violette Pfeile) der sich gegenüberliegenden Punkte (blau) auf diese Linie, so kann man erkennen, dass die Projektionen der Gradienten (orangene Pfeile) genau in die entgegengesetzte Richtung zeigen.

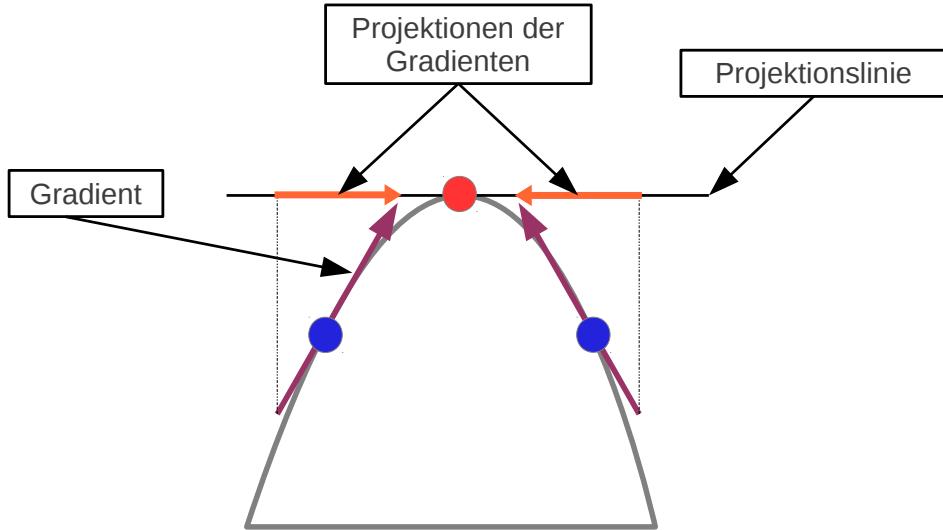


Abbildung 3.10.: Ridge Point Detection. Der rote Punkt ist ein lokales Maximum auf dem Gebirgskamm. Die blauen Punkte liegen sich gegenüber und umschließen den roten Punkt. Ihre auf die Projektionlinie projizierten Gradientenrichtungen zeigen in entgegengesetzte Richtungen. Der rote Punkt bildet somit eine *sign barrier* für die beiden Richtungen.

Die Idee des Algorithmus ist, Projektionslinien durch die Distance Map zu legen und das Verhalten der Gradientenbeträge auf diesen Linien zu beobachten. Man stellt fest, dass sich dabei mehrere Muster von Vorzeichenwechsel der Gradientenbeträge erkennen lassen. Diese Muster können dabei ein Indiz für einen Gebirgspunkt und somit für einen Punkt des Skeletts sein.

Dabei stellt sich die Frage, wieviele Richtungen mit diesen Linien untersucht werden sollen. Man kann beobachten, dass es einen Vorzeichenwechsel in den Gradientenbeträgen gibt, wenn die Projektionslinie den Gebirgskamm schneidet. Dementsprechend gibt es keinen Vorzeichenwechsel, wenn die Projektionslinie parallel zum Kamm verläuft. Findet man also in einer Richtung keinen Gebirgskamm, so muss einer in der orthogonalen Richtung liegen. Deshalb genügt es, zwei zueinander senkrechte Richtungen zu untersuchen. Dabei wählt man die eine Richtung parallel zur x-Achse und die andere Richtung parallel zur y-Achse des untersuchten Bildes. Es existieren insgesamt vier Muster, die auf einen Gebirgskamm deuten. Zwei davon sind in Abbildung 3.11 zu sehen. Die Muster beschreiben, in welcher Weise Vorzeichenwechsel zwischen zwei benachbarten Pixeln auftreten können. Die Symbole + und – die Richtungen der Gradienten. + ist eine positive Richtung (bergauf), – eine negative Richtung (bergab) auf einer Projektionslinie. Das Symbol \circ besagt, dass sich in diesem Bereich der Gradient nicht ändert, da der Grauwert im

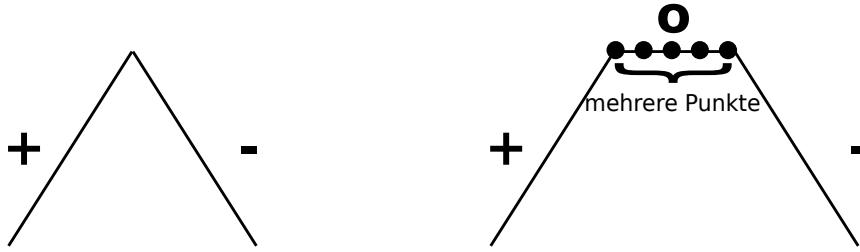


Abbildung 3.11.: Muster für Hinweise auf einen Gebirgskamm. Diese beiden Muster sind ein starkes (*strong*, linke Abbildung) und ein gutes (*good*, rechte Abbildung) Indiz für einen Punkt auf dem Gebirgskamm.

nächsten Nachbarpunkt gleich ist. Die linke Abbildung entspricht genau der Beobachtung, wie sie anhand Abbildung 3.10 beschrieben wurde. Schneidet die Projektionslinie einen Punkt auf dem Gebirgskamm, erzeugt dieser Punkt einen Vorzeichenwechsel zwischen den beiden Punkten, die den Punkt auf dem Kamm umschließen. Die rechte Abbildung zeigt, dass es mehrere Punkte hintereinander in der Distance Map mit dem gleichen Grauwert geben kann, aber auch ein Hinweis für einen Gebirgskamm sind. Dies entspricht im Grauwertgebirge einem Plateau.

Der Algorithmus sucht nun in x -und in y Richtung - von oben nach unten und von links nach rechts - in der Distance Map nach diesen Mustern und markiert die Punkte nach den Eigenschaften *strong*, *good*, *weak* und *none*. Diese Markierung gibt die Stärke der Sicherheit des Punktes wieder, ein Punkt auf dem Gebirgskamm zu sein und somit zum Skelett zu gehören.

Wurden alle Punkte in beide Richtungen untersucht, hat der Algorithmus zu jedem Punkt das richtige Label gefunden und alle Punkte, die zu einem Gebirgskamm gehören [Cha07]. Diese Labels werden weiter benutzt, um eine Graphenrepräsentation des Skeletts zu erstellen.

3.3. Weitere Verfahren

Autor: Christopher Kroll

Neben Thinning und Distanztransformation bilden Algorithmen, die auf kritische Punkte basieren, die dritte Kategorie. Hier werden kritische, bzw. wichtige Punkte detektiert, die dann zu einem Skelett verbunden werden. Zwei Vertreter dieser Kategorie sind der einfache Kritische-Punkte-Ansatz und die Triangulations-Technik.

Beim einfachen Kritische-Punkte-Ansatz wird das Quellbild spalten- oder zeilenweise durchgegangen. Werden Objektpixel gefunden, so wird die Mitte dieses 'Objektpixel-blocks' markiert. Dies sind die kritischen Punkte, die nach der Markierung verbunden werden und somit das Skelett bilden. Ein Nachteil dieses Ansatzes ist, dass abhängig von der Lage des Objektes und der Wahl, ob das Quellbild zeilen- oder spaltenweise

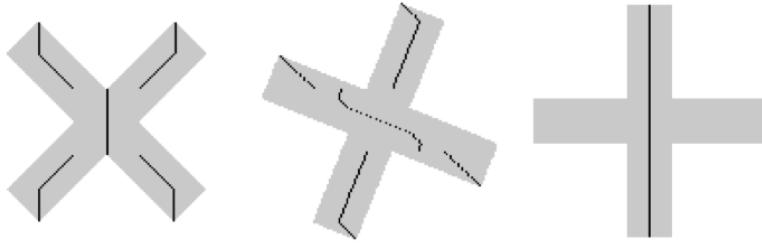


Abbildung 3.12.: Skelettierung beim einfachen Kritische-Punkte-Ansatz und die Veränderung bei der Drehung. Bildquelle: [Buh08]

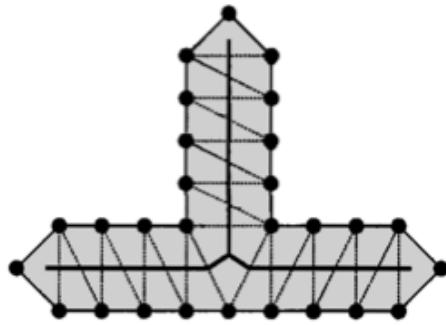


Abbildung 3.13.: Triangulation. Bildquelle: [Buh08]

durchlaufen wird unterschiedliche Ergebnisse erzielt werden. Das Verfahren ist somit nicht isotrop. Abbildung 3.12 zeigt die Skelettierung bei zeilenweisem Durchlaufen des Bildes. Abhängig von der Lage des Objektes ergibt sich ein anderes Skelett. Wenn dieses Objekt spaltenweise durchlaufen werden würde, würde man beim rechten Kreuz eine horizontale statt vertikale Linie erhalten. Dieses Verfahren ist deswegen dafür geeignet, wenn man die Lage des Objektes und das gewünschte Ergebnis vorher kennt, bzw. die Lage vor der Skelettierung beeinflussen lassen kann.

Flexibler als der obere Ansatz ist die Triangulation. Da dieser jedoch um einiges komplexer ist und für das weitere Vorgehen keine Rolle spielt, wird sie nur oberflächlich der Vollständigkeit halber erläutert. Abbildung 3.13 stellt alle Schritte der Triangulation dar: Zunächst werden Punkte an der Kontur des Objektes markiert. Diese Punkte werden dann so verbunden, dass Dreiecke im Objekt entstehen. Je nach Lage des Dreiecks werden die Skelettpunkte erstellt, entweder am Schwerpunkt oder auf der Kante. Die Skelettpunkte werden dann für das endgültige Skelett verbunden. Näheres zu diesem Verfahren ist in [Buh08] beschrieben.

Zu erwähnen ist außerdem, dass es neben diesen drei vorgestellten Skelettierungsategorien noch weitere Algorithmen gibt, die jedoch sehr speziell sind und sich dadurch schwer kategorisieren lassen (zum Beispiel das Hamilton-Jacobi-Skelett).

4. Implementierung der Algorithmen

Autor: Sandra Schröder

Im Rahmen des Projekts wurden zwei Algorithmen für die Skelettierung implementiert. Dies ist zum einen die Skelettierung nach Thinning nach dem Algorithmus der in Abschnitt 3.1.1 beschrieben wurde. Die Skelettierung mittels Distanztransformation wurde nach einer eigenen Idee entwickelt und umgesetzt.

Die Skelettierung läuft in zwei Schritten ab. Erst wird der Spieler segmentiert. Man erhält als Ergebnis ein Binärbild, welches im zweiten Schritt weiterverarbeitet wird, um ein Skelett zu extrahieren. Die Segmentierung des Spielers ist bei beiden Ansätzen identisch. In diesem Kapitel wird die Umsetzung der Algorithmen anhand signifikanten Codeausschnitten der Implementierungen vorgestellt. Ein Überblick über die technische Umsetzung gibt einen Eindruck über die verwendeten Programmiersprachen und Arbeitsumgebungen.

4.1. Technische Umsetzung

Autor: Christopher Kroll

Für die Implementierung wurden die bereitgestellten iMacs am Informatikum in Stellingen gewählt. Der Vorteil war neben der Performanz die schon eingerichtete Arbeitsumgebung. Um die Verbindung zu der Kinect herzustellen, wurde das quelloffene Framework libfreenect der OpenKinect-Gruppe benutzt. Libfreenect bietet Treiber und Bibliotheken, um zum Beispiel die Bilder der Kamera anzuzeigen oder den eingebauten Motor zu steuern (siehe Abbildung 4.1).

Um die empfangenen Bilder verarbeiten zu können wurde die Bibliothek OpenCV eingesetzt. Sie unterstützt unter anderem beim Speichern und Laden von Bildern und bei der Segmentierung. Wichtige Aufrufe von OpenCV-Funktionen werden im Folgenden an geeigneter Stelle erwähnt.

Eine weitere wichtige Bibliothek, die auch OpenCV benutzt, ist NumPy. Es ist eine Python-Erweiterung und wurde entwickelt, um Operationen bezüglich mathematischen Funktionen und multidimensionalen Arrays effizienter zu gestalten [num]. So wurde NumPy benutzt, um Arraywerte bei der Segmentierung logisch zu verknüpfen (siehe Anhang A.2).

Der Großteil der Programmierung erfolgte in der Sprache Python. Hierbei handelt

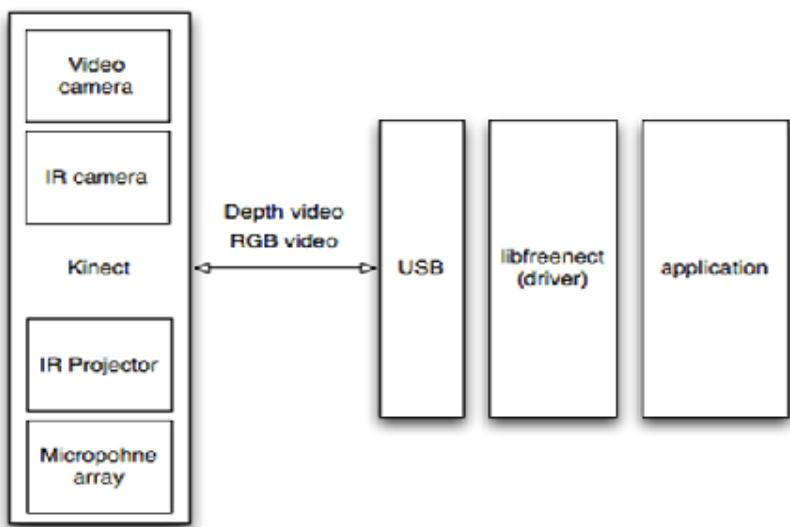


Abbildung 4.1.: Libfreenect [lib]

es sich um eine leicht zu erlernende Interpretersprache. Allerdings traten im Laufe des Projektes Performanzprobleme bei der Implementierung des Thinning-Algorithmus auf. Aus diesem Grund wurde über einen Python-Wrapper der in C++ implementierte Algorithmus eingebaut. Näheres dazu wird in Kapitel 4.3 beschrieben.

Als Entwicklungsumgebung wurde Spyder für die Python-Programmierung und Apples Xcode für die C++-Programmierung benutzt.

4.2. Spielersegmentierung

Autor: Sandra Schröder

Es wird anhand der Tiefeninformation segmentiert, die die Kinect liefert. Somit kann der Spieler in einfacher Weise vom Hintergrund und von anderen Objekten, die nicht skelettiert werden sollen, getrennt werden. Der Open-Source-Treiber für die Kinect - *Freenect* - bietet Funktionen für den Zugriff auf die Tiefenwerte. Die Funktion `pretty_depth` des Moduls `frame_convert` normiert die Tiefenwerte auf das Intervall $[0, \dots, 255]$.

Listing 4.1: Tiefenwerte zurückgeben

```

1 def get_depth():
2     return frame_convert.pretty_depth
3     (freenect.sync_get_depth()[0])

```

Um nicht für jeden einzelnen Pixel die Bedingung zu überprüfen, ob er den Schwellwert für die Segmentierung überschreitet beziehungsweise unterschreitet, wird die effiziente Numpy-Funktion `logical_and` benutzt, die global auf dem Bild arbeitet und für das gesamte Bild die Schwellwertbedingung prüft. Eine pixelweise Überprüfung wäre mit Python eine ineffiziente Lösung.

Für den Schwellwert werden zwei Werte definiert, um ein Intervall festzulegen, in dem

sich das Objekt befinden darf. In Listing 4.2 legen die Variablen `current_depth` und `threshold` das Intervall fest.

Listing 4.2: Spielersegmentierung in Python

```

1 #Segmentierung des Tiefebildes
2 depth = 255 * np.logical_and
3     (depth >= current_depth - threshold,
4      depth <= current_depth + threshold)

```

Da die Funktion auf Numpy-Arrays arbeitet, muss das Bildobjekt zuvor in ein Array umgewandelt werden. Es wurden vorgefertigte Funktionen von OpenCV benutzt, die diese Konvertierung vornehmen.

4.3. Skelettierung mittels Thinning

Autor: Christopher Kroll

Da man sich schon durch den Seminarteil mit dem Algorithmus nach [ZS84] beschäftigte, fiel die Wahl des Thinning-Algorithmus zunächst auf diesen.

Der Algorithmus arbeitet mit Binärbildern, also alle Objektpixel sind auf 1, bzw. 255 gesetzt und alle Hintergrundpixel auf 0. Die in Kapitel 4.2 vorgestellte Spieler-Segmentierung lieferte jedoch anfangs nur ein Grauwertbild. So war nun der erste Schritt dieses Bild in ein Binärbild umzuwandeln. Die Idee war das zweidimensionale Grauwertbildarray mit zwei Schleifen zu durchlaufen und jeden Wert, der größer als 128 ist auf 255 zu setzen. Jeder Wert der kleiner oder gleich 128 ist, wird auf 0 gesetzt (siehe Listing 4.3).

Listing 4.3: Binarisierung des Grauwertbildes.

```

1 for(int y=0; y < height; y++) {
2     for(int x=0; x < width; x++) {
3         if(img(x,y)>128){
4             img(x,y)=255;
5         } else{
6             img(x,y)=0;
7         }
8     }
9 }

```

So erhalten wir ein Array, das nur aus den Werten 0 und 255 besteht. Das Problem dabei war allerdings die nicht ausreichende Leistungsfähigkeit von Python. Schon bei diesen zwei ineinander verschachtelten Schleifen bestand keine Echtzeitfähigkeit mehr. Ohne auch nur den iterativen - und somit rechenintensiven - Thinning-Algorithmus eingesetzt zu haben stockte das Ergebnisbild zu stark. Python ist zwar ähnlich schnell wie PHP, Perl oder Smalltalk, kann in Sachen Geschwindigkeit mit kompilierten Sprachen jedoch nicht mithalten [pyt].

Aus diesem Grund musste eine Alternative zu Python gefunden werden. Die Wahl fiel auf die leistungsstärkere Compilersprache C++.

Nun sollten jedoch nicht zwei unterschiedliche Programme entstehen - Distanztransformation mit Python und Thinning mit C++ und die Spieler-Segmentierung in Python sollte

auch für die Thinning-Skelettierung benutzt werden. Die Lösung war die Möglichkeit der Anbindung von Python an eine andere Programmiersprache, in diesem Fall C++. So wurde der in Kapitel 3 vorgestellte Thinning-Algorithmus in C++ geschrieben und mit einem Makefile kompiliert. Der kompilierte Code wurde nun mit einem Wrapper eingebunden. Die Python-Datei `pythonWrapper.py` lädt dafür die vom Makefile erstellte Bibliothek, um auf die kompilierten Daten zuzugreifen (Listing 4.4) .

Listing 4.4: Laden der Wrapper-Bibliothek.

```
1 lib = N.ctypeslib.load_library('libpython-wrapper', '..')
```

Die C++-Funktion kann nun mit Python aufgerufen werden (Listing 4.5).

Listing 4.5: Aufruf der C++-Funktion.

```
1 lib.vigra_reflectimage_c(arr, arr2, arr.shape[1], arr.shape[0], reflect_mode)
```

Da nun mit dem Algorithmus nach [ZS84] trotz mehrstündiger Fehlersuche und Optimierung jedoch keine guten Ergebnisse erzielt werden konnten, wurde auf den verwandten Guo-Hall-Algorithmus zurückgegriffen. Auch er hat vier Kriterien, die zutreffen müssen, damit ein Pixel gelöscht werden kann. Aufgrund der Pixelbenennung im 3x3-Pixel-Fenster aus Abbildung 3.2 ergeben sich folgende Kriterien in der C++-Implementation:

Listing 4.6: Guo-Hall-Kriterien in C++

```
1 int C = (!p2 & (p3 | p4)) + (!p4 & (p5 | p6)) + (!p6 & (p7 | p8)) + (!p8 & (p9 |
2   ↪ p2));
3 int N1 = (p9 | p2) + (p3 | p4) + (p5 | p6) + (p7 | p8);
4 int N2 = (p2 | p3) + (p4 | p5) + (p6 | p7) + (p8 | p9);
5 int N = N1 < N2 ? N1 : N2;
6 int m = iter == 0 ? ((p6 | p7 | !p9) & p8) : ((p2 | p3 | !p5) & p4);
7 if (C == 1 && (N >= 2 && N <= 3) & m == 0)
8   marker.at<uchar>(i, j) = 1;
```

Treffen die Kriterien in Zeile 7 zu, wird der Pixel in Zeile 8 zum Löschen markiert.

4.4. Skelettierung mittels Distanztransformation

Autor: Sandra Schröder

Der theoretische Ablauf der Skelettierung wurde bereits im Kapitel 3 beschrieben. Zur Rekapitulation werden die Schritte kurz aufgezählt.

Die Skelettierung anhand der Distanztransformation läuft folgendermaßen ab:

- Bestimmen der Distanztransformation des Binärbildes
- Berechne den Gradientenbetrag auf der Distance Map und führe die Segmentierung auf dem Gradientenbetragsbild aus.
- Differenz zwischen dem Gradientenbild und der Distance Map bilden

Die Distance Map kann mit einer Funktion aus der Bildverarbeitungsbibliothek *OpenCV* einfach berechnet werden. Die Funktion (Listing 4.7) erwartet als Eingabe das Originalbild (`img`) und ein Bild (`dist_img`), um das Ergebnis speichern zu können (gleiche Größe und Dimension wie das Originalbild). Eine weitere Möglichkeit, die die Funktion bietet, ist die Angabe einer Metrik, nach der der Abstand eines Pixels zum Hintergrund bestimmt wird. Es wurde die euklidische Metrik benutzt.

Listing 4.7: Berechnen der Distance Map des Binärbildes des Spielers.

```
1 cv.DistTransform(img, dist_img, distance_type=cv.CV_DIST_L2)
```

Zur Bestimmung des Gradientenbetrages der Distance Map wurde die Numpy-Funktion `gaussian_gradient_magnitude` genutzt (Listing 4.8). Die Funktion berechnet den Gradientenbetrag mit Ableitungen der Gaussfunktion. Die Variable `sigma` ist die Standardabweichung des Gaussfilters. Das Ergebnis dieser Funktion wird in ein Bildobjekt konvertiert und entsprechend festgelegter Schwellwerte (`lowerbound`, `upperbound`) segmentiert.

Listing 4.8: Gradientenbetrag der Distance Map und Segmentierung des Gradientenbetragsbildes.

```
1 #Gradienten-Berechnung
2 ndimage.gaussian_gradient_magnitude(distancemap,
3     sigma,gradient_image)
4 #Vor -und Nachbearbeitungen
5 #Segmentierung
6 cv.InRangeS(gradient_image,lowerbound,upperbound,
7     threshed_gradient_image)
```

Zur Differenzbildung und endgültigen Berechnung des Distanzskellettes werden die Bildobjekte in Arrays umgewandelt. Diese Arrays können einfach voneinander abgezogen werden (Listing 4.9).

Listing 4.9: Differenz zwischen Distance Map und segmentiertem Gradientenbetrag

```
1 #Berechnung Differenzbild
2 diff = distancemap_array - threshed_gradient_array
3 diff = diff * 1.0
```

Bei der Implementierung wurden in keinem Fall Operationen ausgeführt, die auf einzelne Pixel zugreifen. Situationen, in denen pixelweise Operationen durchgeführt werden könnten, wurden umgangen, indem Arrayoperationen oder Funktionen aus der OpenCV-Bibliothek benutzt wurden. Ein pixelweiser Zugriff könnte bei einer Interpretersprache wie Python zu einer sehr langsamen Ausführung der Skelettberechnung führen.

5. Evaluation

Autor: Sandra Schröder

In diesem Kapitel werden die beiden entwickelten Algorithmen gegenüber gestellt. Zwei zentrale Kriterien zur Evaluation der Algorithmen werden thematisiert.

- Skelettqualität
- Echtzeitfähigkeit

Zur Bewertung der Skelettqualität greifen wir auf die gewünschten Eigenschaften eines Skeletts zurück (Kapitel 3) und überprüfen, ob die Algorithmen sie realisieren. Auch die Echtzeitfähigkeit ist von Bedeutung. Grundlage für eine verzögerungsfreie Interaktion des Spielers mit der Anwendung (dem Spiel) ist eine schnelle Berechnung des Skeletts. Denn trotz nachfolgender Verarbeitungsschritte soll eine hohe Anzahl von Bildern schnell (Größenordnung 20 Bilder pro Sekunde) prozessiert werden.

5.1. Skelettqualität

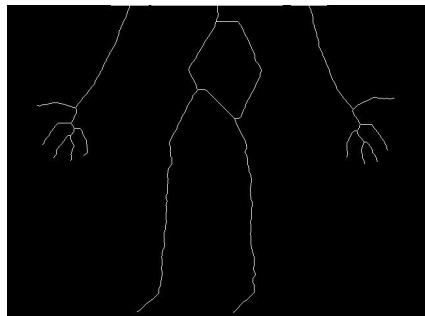
Zum Vergleich der Algorithmen wurden Posen des Spielers aufgenommen (Abbildung 5.1). Bei gleichen Eingabedaten weisen die beiden Algorithmen Resultate von unterschiedlicher Qualität auf. Pose 1 zeigt den Unterkörper des Spielers. Beide Skelette beschreiben die Grundstruktur des Spieler sehr gut. Arme und Beine werden jeweils durch eine im ursprünglichen Objekt mittig liegende Linie beschrieben. Die Hände des Spielers sind deutlich vom Körper gestreckt und die Finger gespreizt. Der Thinning-Algorithmus ist in der Lage die Topologie der Hände und Finger wiederzugeben, während bei der Distanztransformation das Skelett der Hände schlechter zu erkennen ist. Die zweite getestete Pose (Pose 2 in Abbildung 5.1) erzeugt ebenfalls Skelette von unterschiedlicher Qualität. Auffallend sind die Lücken in den Skelettlinien des Skeletts, das aus der Distanztransformation entstanden ist. Der Thinning-Algorithmus hat stets ein Skelett mit zusammenhängenden Skelettkomponenten als Ergebnis. Jedoch ist beim Skelett des Thinning-Algorithmus auf Bauchhöhe ein horizontaler Ausläufer zu erkennen, der nicht die eigentliche Form des Objektes beschreibt. Das aus der Distanztransformation resultierende Skelett gibt den Oberkörper bis zu den Schultern als eine Linie ohne Ausläufer wieder.

Eine weitere Eigenschaft einer Skelettlinie ist ihre Breite, die sich idealerweise auf einen Pixel beschränkt. Dies ist beim Thinning der Fall, bei der Distanztransformation fällt die Skelettlinie breiter aus. Wird das Skelett zur Datenkomprimierung genutzt, enthalten

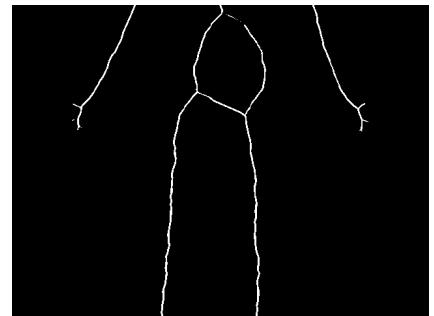
Pose 1



Spieler (segmentiert)



Thinning-Skelett

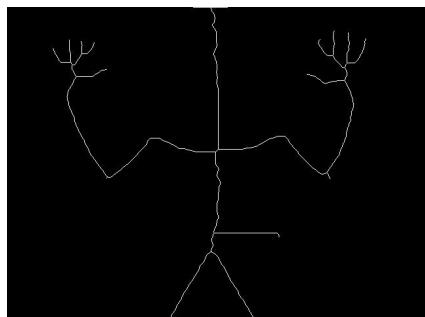


Distanzskelett

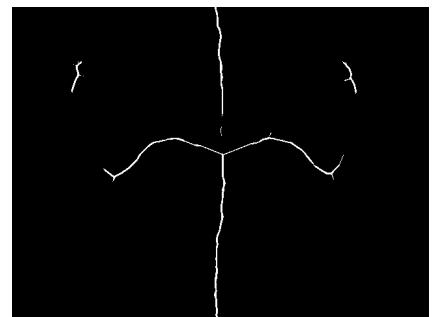
Pose 2



Spieler (segmentiert)



Thinning-Skelett



Distanzskelett

Abbildung 5.1.: Vergleich zweier Posen. Der Thinning-Algorithmus und die Skelettierung mittels Distanztransformation weisen jeweils Resultate von unterschiedlicher Qualität auf.

unnötige Pixel mehr Informationen, als eigentlich benötigt wird. Für eine konkrete Anwendung muss diskutiert werden, ob eine 1-Pixel-Breite unbedingt nötig ist. Bezuglich der Skelettqualität liefert der Thinning-Algorithmus sehr gute Skelette. Hier sind keine weiteren Verbesserungen mehr nötig. Bei dem Distanzskelett sind Verbesserungen, besonders bei der Konnektivität des Skelettes, möglich. In Abschnitt 5.3 werden zwei Verbesserungsmöglichkeiten vorgestellt.

5.2. Laufzeiten

Die Laufzeit wurde mit dem Profiling-Tool *CProfile* gemessen. Es erstellt für Python-Programme eine Statistik über die Laufzeiten eines Programms und seine einzelnen Funktionen. Es wurde jeweils eine Statistik für die Implementierung des Thinning-Algorithmus und des Algorithmus der Distanztransformation erstellt. Die wichtigsten Funktionen der Programme wurden aus der Statistik entnommen. Die Tabellen 5.1 und 5.2 geben einen Überblick über die Laufzeiten.

Obwohl der Thinning-Algorithmus bezüglich der geforderten Eigenschaften von Skeletten die besten Ergebnisse liefert, ist er deutlich langsamer als die Skelettierung mittels Distanztransformation. Zur Initialisierung der Algorithmen wird einmalig zu Beginn eine Startfunktion ausgeführt. Diese beansprucht weniger als 30 Sekunden und kann gegenüber der Gesamlaufzeit der Anwendung (Minuten bis Stunden) vernachlässigt werden. Die Spielersegmentierung und die Skelettberechnung wird pro Bild einmalig ausgeführt. Die Segmentierung unterscheidet sich bei den beiden Ansätzen nicht. Die Abweichung zwischen den Laufzeiten ist auf Messungenauigkeiten zurück zu führen. Die Skelettierung nimmt bei beiden Ansätzen deutlich mehr CPU-Zeit in Anspruch als die Segmentierung. Pro Bild ergibt sich ein Verhältnis der Laufzeiten von:

$$\frac{\text{Laufzeit Skelett aus Thinning}}{\text{Laufzeit Skelett aus Distanztransformation}} = \frac{0.003\text{ s} + 0.904\text{ s}}{0.004\text{ s} + 0.076\text{ s}} \approx 11 \quad (5.1)$$

Die Berechnung des Distanzskeletts ist also um mehr als eine Größenordnung schneller als die Berechnung mittels Thinning. Der Thinning-Algorithmus ist maschinennah in C++ programmiert, und die Distanztransformation ist in der Interpretersprache Python umgesetzt. Es ist also anzunehmen, dass die Distanztransformation durch die Verwendung einer Compiler-Sprache deutlich beschleunigt werden kann.

46371 Funktionsaufrufe in 16.723 CPU-Sekunden			
Funktion	Anzahl Aufrufe	Gesamtzeit [s]	Pro Aufruf [s]
Startfunktion	1	16.723	16.723
Spielersegmentierung	126	0.453	0.004
Berechnung des Distanzskeletts	126	9.594	0.076

Tabelle 5.1.: Laufzeiten Skelettierung mittels Distanztransformation

1424 Funktionsaufrufe in 28.159 CPU-Sekunden			
Funktion	Anzahl Aufrufe	Gesamtzeit [s]	Pro Aufruf [s]
Startfunktion	1	28.159	28.159
Spielersegmentierung	29	0.088	0.003
Skelettierung nach Thinning	29	26.224	0.904

Tabelle 5.2.: Laufzeiten Skelettierung mittels Thinning

5.3. Distanztransformation - Verbesserung der Skelettqualität

Im Folgenden wird ein Ansatz zur Verbesserung der Skelettqualität vorgestellt.

Das Skelett, welches mit der Methode der Distanztransformation bestimmt wurde, weist Lücken zwischen den Skelettteilen auf. Um die Topologie und geometrische Eigenschaften des Objekts gut wiederzugeben, ist ein lückenloses Skelett wünschenswert.

Die Segmentierung des Gradientenbetrages der Distance Map erzeugt Lücken im Distanz-skeletts. Dies wird anhand von Abbildung 5.2 deutlich. Im Gradientenbild (Teilabbildung b) sind noch durchgehende Skelettlinien zu erkennen. Per Schwellwert-Filter wird das Skelett aus dem Gradientenbild extrahiert. Für das gezeigte Beispielbild konnte allerdings kein geeigneter Schwellwert gefunden werden, der sowohl Skelettkonnektivität gewährleistet als auch Artefakte verhindert. Teilabbildung c zeigt die Segmentierung des Gradientenbildes. In diesem Schritt ist die Skelettkonnektivität erstmals nicht gewährleistet. Auch in den folgenden Schritten wird die Skelettkonnektivität nicht wieder hergestellt.

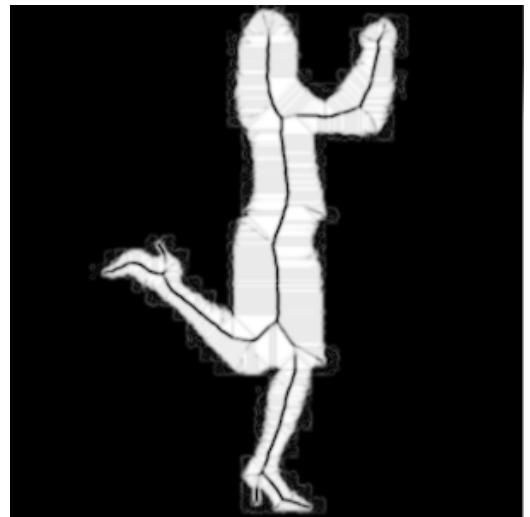
In den nächsten Abschnitten werden Methoden zur Verbesserung des Skeletts vorgestellt. Dies umfasst zum einen die Herstellung der Konnektivität der Skelettkomponenten beziehungsweise dem Füllen der Lücken zwischen den Skelettlinien. Zuerst wurden markante Punkte auf dem Skelett bestimmt. Diese wurden genutzt, um mittels Breitensuche oder Tiefensuche Pfade zwischen ihnen zu finden und sie zu verbinden. Punkte werden miteinander verbunden, wenn sie nah genug beieinander sind. Das Ergebnis ist ein Skelett mit zusammenhängenden Skelettlinien.

5.3.1. Berechnung von Features auf dem Skelett

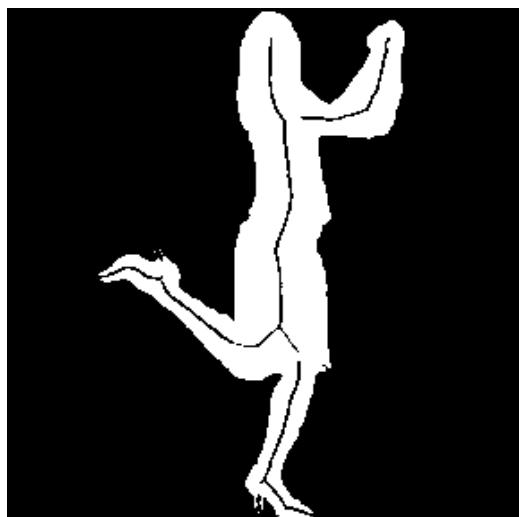
Die Bildverarbeitungsbibliothek *OpenCV* bietet Funktionen zur Bestimmung von markanten Punkten, sogenannten *Features*, in einem Bild. Die Funktion `goodFeaturesToTrack` [ST94] bestimmt die stärksten Ecken in einem Bild oder Bildausschnitt. Mittels eines Qualitätsmaßes wird entschieden, ob die Stärke einer Ecke an einem bestimmten Pixel ausreicht, um in die Featuremenge aufgenommen zu werden. Für die Funktion können der Wert des Qualitätsmaßes, den eine Ecke erfüllen muss, die Anzahl der Ecken, die gefunden werden sollen und die minimale Distanz zwischen den stärksten Ecken übergeben werden. Die Qualität einer Ecke wird anhand von Eigenwerten bestimmt. Die Eigenwerte beziehen



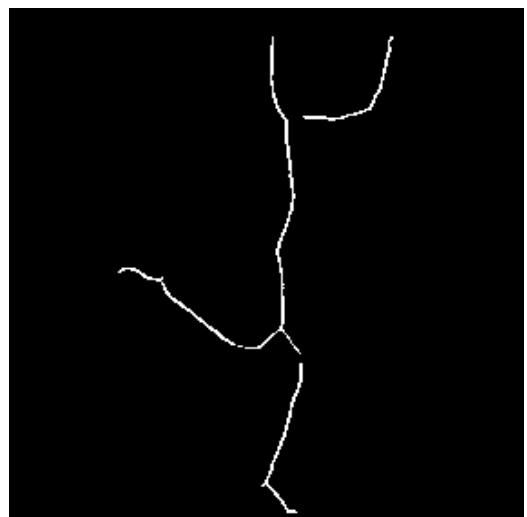
(a)



(b)



(c)



(d)

Abbildung 5.2.: Bestimmung des Skelett mittels Distanztransformation. (a) Originalbild
(b) Gradientenbild des Distanzskeletts (c) Gradientenbild segmentiert
(d) Distanzkelett

sich dabei auf die Kovarianzmatrix von Ableitungen einer festgelegten Umgebung eines Pixels. Es wird der minimale Eigenwert für die Eckendetektion weiterverwendet. Ist der minimale Eigenwert einer Ecke kleiner als das gewünschte Qualitätsmaß, wird diese Ecke verworfen. Die verbleibenden Ecken werden nach ihrer Qualität absteigend sortiert. Anschließend wird überprüft, ob es in der spezifizierten Distanz Ecken gibt, die stärker sind. Die stärksten Ecken werden dann endgültig in die Feature-Menge aufgenommen. Abbildung 5.3 zeigt das Ergebnis der Berechnung. Die Kreise markieren die Feature-

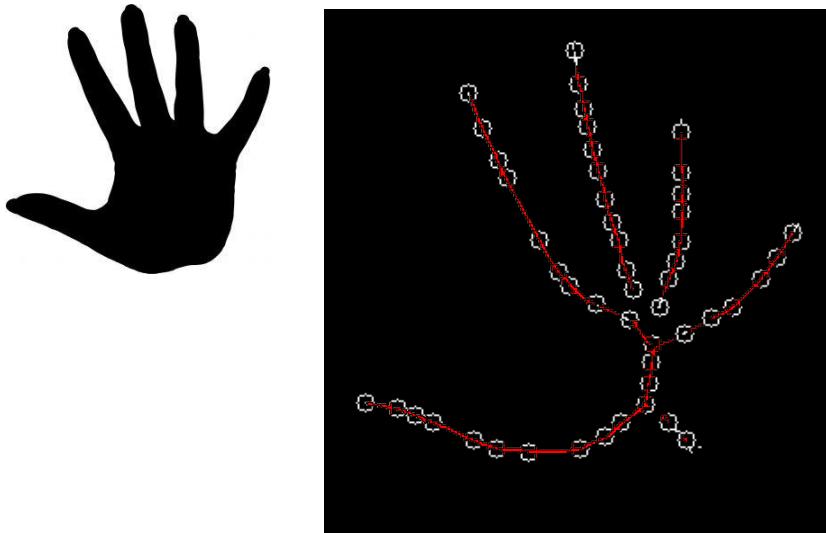


Abbildung 5.3.: Ergebnis der Funktion `goodFeaturesToTrack`. Links: Originalbild. Rechts: Berechnetes Distanzskelett mit Features. Die roten Linien markieren das Rohskelett, die weißen Kreise markieren die Features, die auf dem Skelett gefunden wurden.

Punkte. Wie man erkennen kann, befinden sich die Features auf der Skelettlinie. Dies ist eine Grundlage für die weiteren Verbesserungen des Skeletts. Befänden sich Features außerhalb der Skelettlinien, könnte die ursprüngliche Form des Skeletts und die Topologie des Objekts verfälscht werden. Diese Problematik wurde bei dem gewählten Algorithmus [ST94] nicht beobachtet.

5.3.2. Breitensuche

Der Breitensuche-Algorithmus durchsucht einen Graphen ausgehend von einem Startpunkt nach weiteren Knoten. Der Algorithmus sucht zunächst nur nach direkt nachfolgenden Knoten und somit in die Breite des Graphen. Knoten, die bereits besucht wurden, werden markiert. Wurde ein Knoten noch nicht besucht, wird er in eine *Queue* (Warteschlange) aufgenommen.

Mittels Breitensuche sollen Pfade zwischen den markanten Punkten auf dem Skelett gefunden werden. Die markanten Punkte entsprechen Knoten in einem Graphen. Für die Nachbarschaftsbeziehung zwischen zwei Knoten wird ein Nachbarschaftsmaß definiert.

Abbildung 5.4 zeigt, wie überprüft wird, ob ein Punkt in einem festgelegten Intervall des Punktes (x, y) liegt. Es wird eine Suchdistanz für beide Richtungen festgelegt. Sie wird entsprechend der minimalen Distanz, die zwischen markanten Punkten erlaubt ist (Abschnitt 5.3.1), gewählt.

Erst wird in x-Richtung gesucht, dann in y-Richtung. Fällt der Punkt in das Intervall, wird er als besucht markiert und mit dem Punkt (x, y) verbunden. In Anhang A befindet sich der Quellcode zur Breitensuche (Listing A.6, Zeile 78/79).

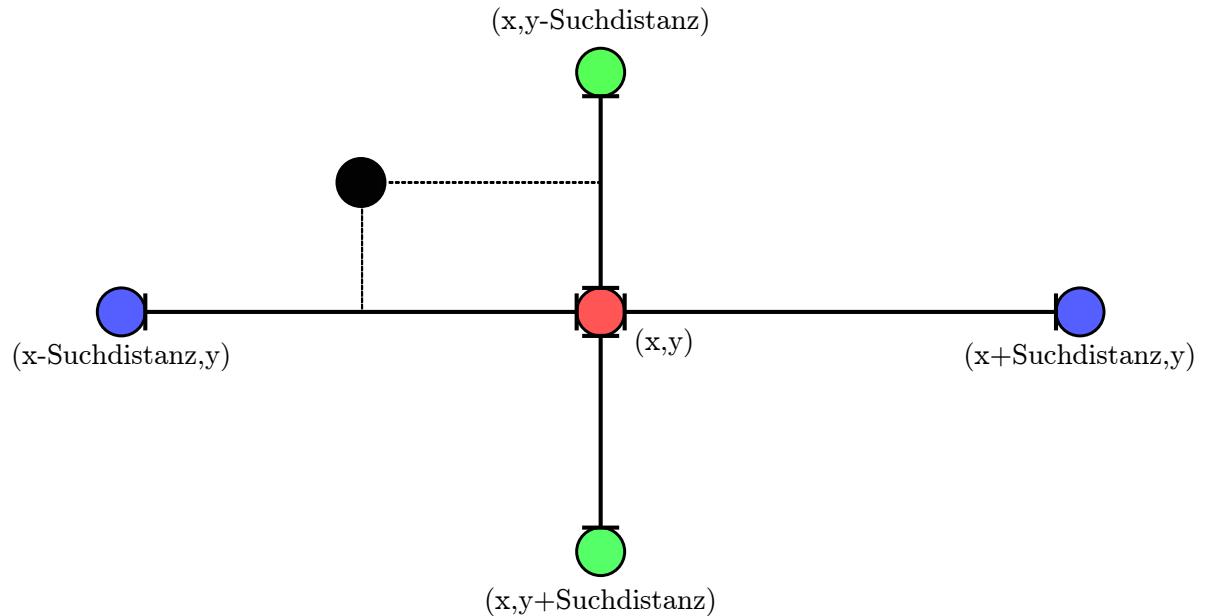


Abbildung 5.4.: Das aufgespannte Suchkreuz von Punkt (x, y) (rot). Für den schwarzen Punkt wird überprüft, ob er in den festgelegten Suchintervallen des Punktes (x, y) liegt.

Das Ergebnis des Algorithmus ist in Abbildung 5.5 im Vergleich zum vorigen Skelett zu sehen. Das Skelett besitzt nun zusammenhängende Komponenten. Auffällig sind die Ausläufer, die auf die Vorgehensweise des Breitensuchealgoritmus zurückzuführen sind. Wird die Suchdistanz zu groß gewählt, findet der Algorithmus mehrere Nachfolger, die er dann mit dem aktuellen Punkt verbindet. Das Ergebnis sind fächerartige Ausläufer.



Abbildung 5.5.: Ergebnis der Breitensuche. Links: Distanzskelett. Rechts: Verbessertes Skelett.

5.3.3. Tiefensuche

Wie bei der Breitensuche wird ausgehend von einem Startknoten ein Graph nach weiteren Knoten durchsucht. Im Gegensatz zur Breitensuche erfolgt das Traversieren des Graphens in die Tiefe. Wurde ein Nachfolgeknoten gefunden, wird für diesen weiter geprüft, ob er ebenfalls einen Nachfolgeknoten besitzt. Dies wird solange wiederholt (rekursiv) bis kein Nachfolgeknoten mehr gefunden werden kann. Mittels *Backtracking* wird der Pfad zurückverfolgt. Anschließend wird ein neuer Knoten als Startpunkte gesucht, der noch nicht besucht wurde und das Durchsuchen nach nächsten Nachbarn wird für diesen Knoten wiederholt. Die Suche nach dem nächsten Nachbarn funktioniert wie bei der Breitensuche (Abbildung 5.4).

Das Ergebnis ist in Abbildung 5.6 zu sehen. Es fällt auf, dass es noch Lücken im Skelett gibt, was auf die Suchdistanz zurückzuführen ist. Ist sie zu klein, können keine weiteren Punkte im Umkreis des aktuellen Punktes gefunden werden und der Algorithmus endet für diesen Pfad.



Abbildung 5.6.: Ergebnis der Tiefensuche. Links: Distanzskelett. Rechts: Verbessertes Skelett.

Um die Konnektivität des Skeletts zu gewährleisten, muss das Skelett nachbearbeitet

werden.

Das Skelett, welches aus der ersten Stufe der Verbesserung entstanden ist, besitzt Zusammenhangskomponenten. Innerhalb der Zusammenhangskomponenten ist die Skelettkonnektivität gewährleistet, die globale Konnektivität zwischen den Komponenten jedoch nicht. Die Idee des nächsten Verbesserungsschrittes ist es, die Zusammenhangskomponenten zu verbinden, die am nächsten beieinander liegen. Dazu werden die Punkte aus den Zusammenhangskomponenten extrahiert, die die Endpunkte der Komponenten sind. In Graphenrepräsentation entsprechen diese Punkte den Wurzeln und Blättern des Waldes. Das sind entweder Punkte ohne Vorgänger oder ohne Nachfolger. Für n Komponenten müssen $n - 1$ Verbindungen gefunden werden, um vollständige Konnektivität zu gewährleisten. Dafür werden die kürzesten Verbindungen zwischen Endpunkten verschiedener Komponenten bestimmt: Es wird komponentenweise vorgegangen. Für die Endpunkte der aktuellen Komponente wird der nächste Nachbarpunkt bestimmt, der ein Endpunkt einer anderen Komponente ist. Für die Bestimmung des Abstandes wird die euklidische Metrik d_2 (Gleichung 3.1) verwendet. Welcher Punkt auf welcher Komponente liegt, wird bereits bei der Tiefensuche bestimmt. Der Quellcode befindet sich in Anhang A.

In Abbildung 5.7 wurden die Endpunkte farbig markiert. Dies sind alles Kandidaten zum Verbinden der Komponenten. Das Skelett besitzt in dieser Abbildung fünf Zusammenhangskomponenten, wobei eine Zusammenhangskomponente aus genau einem Punkt besteht (auf dem Ringfinger der Hand). Dieser Punkt besitzt weder einen Vorgänger noch einen Nachfolger.

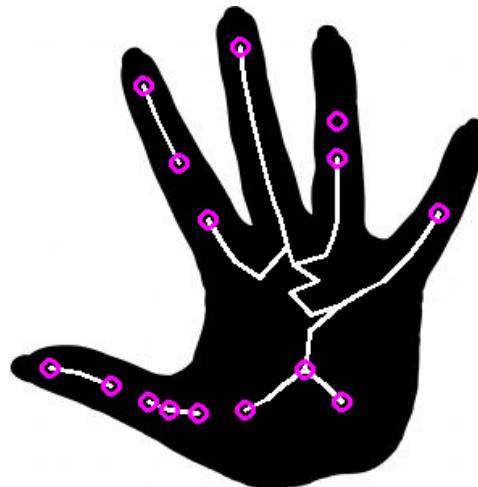


Abbildung 5.7.: Zusammenhangskomponenten (weiß) und ihre Endpunkte.

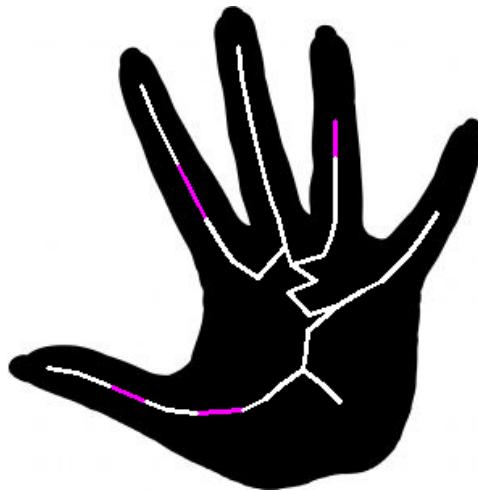


Abbildung 5.8.: Ergebnis des zweiten Verarbeitungsschrittes der Tiefensuche. Die weiße Linie ist das Skelett, das aus dem ersten Verarbeitungsschritt entstanden ist. Die magenta-farbenen Linien sind die Verbindungen zwischen den Zusammenhangskomponenten.

Abbildung 5.8 zeigt, dass die Zusammenhangskomponenten miteinander verbunden wurden, die am nächsten beieinander liegen. Nun ist die Skelettkonnektivität gegeben.

5.4. Fazit

Beide Algorithmen (Thinning und Distanztransformation) liefern Skelette. Diese realisieren die gewünschten Eigenschaften eines Skelettes. Dabei ist die Skelettqualität unterschiedlich. Der Thinning-Algorithmus liefert bezüglich Skelettkonnektivität, 1-Pixel-Breite und Wiedergabe der Topologie hervorragende Ergebnisse. Bei der Skelettierung mittels Distanztransformation ist vor allem auffällig, dass die Skelettkonnektivität nicht vollständig gegeben ist.

Es wurden Verbesserungsmöglichkeiten diskutiert, die zur globalen Skelettkonnektivität bei einem Distanzskelett führen. Die Grundlage der Verbesserung war die Berechnung von markanten Punkten, die mittels Breitensuche oder Tiefensuche verbunden wurden. Die Tiefensuche liefert nicht nur bei der Skelettkonnektivität sehr gute Ergebnisse. Auch die topologischen Eigenschaften des Objektes werden hervorragend beschrieben. Bei der Breitensuche fielen vor allem die fächerartigen Ausläufer auf. Diese sind nicht Teil der Topologie des ursprünglichen Objekts. Es muss auch hier abhängig vom konkreten Anwendungsfall diskutiert werden, ob die Ausläufer stören. Das durch die Tiefensuche verbesserte Skelett realisiert am besten die in Kapitel 3 genannten Qualitätsanforderungen.

Da diese Verbesserungen nach der offiziellen Projektarbeitszeit entstanden sind, konnten keine detaillierten Laufzeit-Messungen mit der Kinect vorgenommen werden. Die Laufzeiten der beiden Algorithmen wurden deshalb mit dem Tool *CProfile* auf statischen Bildern

gemessen. Die Laufzeiten pro Bild fallen bei beiden Algorithmen sehr gering (< 0.001 s) aus. Das Berechnen der Features dauert 0.01 s und damit bleibt die Skelettierung mittels Distanztransformation auch mit den Verbesserungen immer noch deutlich schneller als der Thinning-Algorithmus.

6. Zusammenfassung und Ausblick

Autor: Christopher Kroll

Für alle Teammitglieder war das Arbeiten mit Python und der Kinect Neuland. In der Anfangsphase des Projektes musste sich zunächst in die Programmiersprache und die Funktionsweise der Kamera eingearbeitet werden. Welche Daten die Kinect auf welche Weise liefert und wie diese mit den Frameworks aufgenommen und benutzt werden können, musste untersucht werden.

Damit nicht das komplette Blickfeld der Kamera, sondern nur das relevante Objekt, also der Mensch vor der Kamera, skelettiert wird, musste dieser segmentiert werden.

Aufbauend auf diesem Ergebnis konnte sich mit der Skelettierung und den unterschiedlichen Verfahren gekümmert werden. In dieser Phase wurde das Ziel der Projektarbeit angepasst. Wurde zunächst angepeilt ein Bewegungsanalyseprogramm zu schreiben, wurde nun die Konzentration auf eben diese Verfahren gelegt und das Vorhaben war es jetzt Skelettierungsalgorithmen zu implementieren und zu vergleichen.

Die Entscheidung fiel auf eine Skelettierung mittels Distanztransformation und ein Thinning-Verfahren. Während bei der Distanztransformation ein eigener Ansatz gewählt wurde, entschied man sich durch die Papervorstellung im Seminar einen bereits etablierten Algorithmus für das Thinning zu wählen. Trotz einigen Schwierigkeiten, wie dem Laufzeitproblem in Python und dem Wechsel des Algorithmus, konnten am Ende gute Ergebnisse erzielt und die beiden Verfahren miteinander verglichen werden.

Es stellte sich heraus, dass das Thinning zwar ein gutes Skelett liefert, aber durch den iterativen Prozess zu langsam ist. Der Wechsel der Programmiersprache brachte zwar einen Leistungszuwachs, dieser war jedoch noch immer zu unbefriedigend bei Anwendung auf Bewegtbilder. Für statische Bilder ist dieser Ansatz jedoch geeignet.

Die Distanztransformation ist dagegen echtzeitfähig. Allerdings liefert dieser Ansatz nicht so ein gutes Skelett wie das Thinning. So ist die Pixelkonnektivität nicht gegeben.

Der Vergleich der Algorithmen hat gezeigt, dass sich abhängig vom Algorithmus die Skelettqualität unterscheidet. Auch die Laufzeit ist ein wichtiges Unterscheidungsmerkmal. Vor allem im Hinblick auf die Anforderung Echtzeitfähigkeit.

Die Skelette, die mit den beiden Algorithmen bestimmt wurden, eignen sich beide für weitere Anwendungen. Allerdings muss bei beiden Algorithmen optimiert werden. Beim Thinning-Algorithmus ist dies die Laufzeit, bei der Skelettierung mittels Distanz-

transformation ist dies die Skelettqualität. Die Verbesserungen des Distanzskeletts sind vielversprechend. Der nächste Schritt wäre die Integration in die Kinect-Umgebung. Es muss getestet werden, wie performant dies ist.

Ist die Skelettierung optimiert, kann man das Ergebnis nutzen, um Anwendungen, wie zum Beispiel die erwähnte und zunächst vorgesehene Bewegungsanalyse zu erstellen.

Literaturverzeichnis

- [AMAA] P. Lisouski A.K. Mortensen M.K. Hansen T. Gregersen Authors: M.R. Andersen, T. Jensen and P. Ahrendt. Kinect depth sensor evaluation for computer vision applications.
- [Bor12] G. Borenstein. *Making Things See: 3D Vision with Kinect, Processing, Arduino, and MakerBot*. Make: Books. O'Reilly Media, Incorporated, 2012.
- [Buh08] Christoph Buhlmann. Vergleich und anwendung von skelettierungsalgorithmen in der digitalen bildverarbeitung. 2008.
- [Cha07] Sukmoon Chang. Extracting Skeletons from Distance Maps. *International Journal of Computer Science and Network Security*, 7, 2007.
- [KBC⁺12] Jeff Kramer, Nicolas Burrus, Daniel Herrera C., Florian Echtler, and Matt Parker. *Hacking the Kinect*. Apress, Berkely, CA, USA, 1st edition, 2012.
- [kin] Kinect acuracy. http://www.ros.org/wiki/openni_kinect/kinect_accuracy.
- [lib] Kinnext. <http://alixedi.wikispaces.com/Kinnext> (zuletzt aufgerufen am: 11.03.2013).
- [mic] Microsoft SDK Documentation.
- [num] Wikipedia-artikel numpy (zuletzt aufgerufen am: 10.03.2013). <http://en.wikipedia.org/wiki/NumPy>.
- [pri] How kinect works with primesense. <http://ntuzhchen.blogspot.de/2010/12/how-kinect-works-prime-sense.html>.
- [pyt] Wikipedia-artikel python (zuletzt aufgerufen am: 10.03.2013). [http://de.wikipedia.org/wiki/Python_\(Programmiersprache\)](http://de.wikipedia.org/wiki/Python_(Programmiersprache)).
- [ST94] Jianbo Shi and C. Tomasi. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, pages 593 –600, jun 1994.
- [ste] Stereotriangulation. [http://en.wikipedia.org/wiki/Triangulation_\(computer_vision\)](http://en.wikipedia.org/wiki/Triangulation_(computer_vision)).

- [tri] 3d-objekterfassung mit lasertriangulation. http://wiki.zimt.uni-siegen.de/fertigungsautomatisierung/index.php/3D-Objekterfassung_mit_Lasertriangulation.
- [ZS84] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Commun. ACM*, 27(3):236–239, March 1984.

A. Quellcode

Hier werden die wichtigsten Ausschnitte des Quellcodes gezeigt.

A.1. Skelettierung - Startup

Listing A.1: Startup der Skelettierung. Hier werden die Spielersegmentierung und die Skelettierungsalgorithmen ausgeführt.

```
1 """
2 Startup der Skelettierung. Hier findet
3 die Segmentierung des Spielers und
4 die Skelettierung (nach Thinning oder
5 Distanztransformation) statt.
6
7 @author: Sandra Schroeder, Johannes Boehler,
8 Christopher Kroll
9 """
10
11 import player_segmentation
12 import cv
13 import skeletonization
14 import image_conversion
15 import numpy
16 import depth
17 import pythonWrapper
18
19 def run(algorithm):
20     #Schwellwerte fuer die Segmentierung des Spielers
21     threshold_value = 40
22     depth_value = 170
23     skeleton = skeletonization.Skeleton()
24     while True:
25         #Tiefenbild erzeugen
26         depthvalues = depth.get_better_depth()
27         #Spielersegmentierung
28         depth_seg, depth_image = player_segmentation.player_segmentation(
29             depthvalues, threshold_value, depth_value)
30         #Glaetten des segmentierten Bildes
31         cv.Smooth(depth_seg, depth_seg, smoothtype=cv.CV_MEDIAN,
32                 param1=5, param2=5)
33
34         #Distance Map berechnen und das Skelett daraus extrahieren
35         if algorithm=="distanztransform":
36             diff_img, dist_gradient = skeleton.distance_skeleton(depth_seg)
37             diff = image_conversion.cv2array(diff_img)
38             diff = 255.0 * numpy.logical_and(diff >= 0.2, diff <= 1)
39             diff = diff.astype(numpy.uint8)
40             diff_img = image_conversion.array2cv(diff)
41
42         #Skelettierung mittels Thinning. Innerhalb von Python-Wrapper
43         #wird ein C++-Programm aufgerufen, in dem das Thinning
44         #implementiert ist.
45         elif algorithm=="thinning":
```

```

46         greyscale_array = image_conversion.cv2array(depth_seg)
47         new_g = greyscale_array.reshape((480,640))
48         new_g = pythonWrapper.reflectimage_band(new_g,1)
49
50         #Anzeige des Spielers (segmentiert) und des Distanz-Skeletts
51         cv.ShowImage('Spieler', depth_seg)
52         cv.ShowImage('Distanz-Skellett', diff_img)
53
54         if cv.WaitKey(10)==27:
55             break
56
57     #Auswahl des Algorithmus und Ausfuehrung
58     algorithm="distanctransform"
59     #algorithm="thinning"
60     run(algorithm)

```

A.2. Spielersegmentierung

Listing A.2: Spielersegmentierung

```

1     """Spielersegmentierung anhand von Tiefeinformationen
2
3     @author: Sandra Schroeder, Johannes Boehler, Christopher Kroll"""
4
5     import cv
6     import image_conversion
7     import numpy as np
8
9     """Hauptfunktion fuer die Spielersegmentiert. Hier findet die eigentliche
10    Segmentierung und nachtraegliche Verbesserungen des segmentierten
11    Bildes statt."""
12    def player_segmentation(depth_image_input,
13        threshold_value,depth_value):
14
15        #Umwandlung in ein Numpy-Array
16        depth = image_conversion.cv2array(depth_image_input)
17
18        #Schwellwerte
19        threshold = threshold_value
20        current_depth = depth_value
21
22        #Segmentierung
23        depth = 255 * np.logical_and(depth >= current_depth - threshold,
24                                     depth <= current_depth + threshold)
25
26        #depth in ein Bild umwandeln (ist bis hierhin ein Numpy-Array)
27        depth = depth.astype(np.uint8)
28        depth_image = cv.CreateImageHeader((depth.shape[1], depth.shape[0]),
29                                         cv.IPL_DEPTH_8U,1)
29        #Mit den Daten aus dem Array fuellen
30        cv.SetData(depth_image, depth.tostring(),
31                   depth.dtype.itemsize * depth.shape[1])
32        #Glaettten
33        cv.Smooth(depth_image, depth_image, smoothtype=cv.CV_GAUSSIAN,
34        param1=3, param2=0, param3=0, param4=0)
35        #Dilatation um Loecher und Rauschen zu mindern
36        depth_seg = dilate_image(depth_image)
37
38        return depth_seg, depth_image
39
40    """Dilatation"""
41    def dilate_image(img):

```

```

42 #Kernels sind die strukturierten Elemente fuer die Dilatation.
43 #Hier wurden rechteckige und elliptische Elemente gewaehlt.
44 #kernel=cv.CreateStructuringElementEx(3, 3, 0, 0, cv.CV_SHAPE_RECT)
45 kernel=cv.CreateStructuringElementEx(5, 5, 0, 0, cv.CV_SHAPE_ELLIPSE)
46 #In einem neuen Bild speichern
47 img_dil = cv.CreateImage(cv.GetSize(img),8,1)
48 #Dilatation
49 cv.Dilate(img,img_dil,kernel,iterations=2)
50 return img_dil

```

A.3. Skelettierung - Distanztransformation

Listing A.3: Skelettierung mittels Distanztransformation. Implementierung der in Abschnitt 3.2 beschriebenen Schritte zur Extraktion eines Skeletts.

```

1 """
2 @author: Sandra Schroeder, Johannes Boehler, Christopher Kroll
3
4 Skelettierung mittels Distanztransformation. Zur Extraktion
5 der Skelettlinie wird der Gradientenbetrag der Distance-Map bestimmt.
6 Eine anschliessende Differenzbildung zwischen Gradientenbetrag und
7 Distance-Map ergibt die Skelettlinie.
8
9 """
10
11 import cv
12 import image_conversion
13 from scipy import ndimage
14 import numpy
15
16 class Skeleton():
17
18     """Berechnung des Distanzskeletts. Vier Schritte: Berechnen der
19     Distance Map, bestimmen des Gradientenbetrags der Distance Map,
20     segmentieren des Gradientenbetrags und Differenzbildung zwischen
21     Distanzmap und Gradientenbetrag (segmentiert)"""
22     def distance_skeleton(self,img):
23         #Ein Zielbild bereitstellen, um das Ergebnis der Distance
24         #Transformation zu speichern
25         dist_img = cv.CreateImage(cv.GetSize(img),32,1)
26         #Distanztransformation
27         cv.DistTransform(img, dist_img, distance_type=cv.CV_DIST_L2)
28         #Konvertierung von CV-Bildobjekt zu Numpy-Array
29         dist_img_mat = image_conversion.cv2array(dist_img)
30         #Normalisierung der Distance Map
31         max_of_dist = dist_img_mat.max()
32         dist_img_mat = dist_img_mat/max_of_dist
33         #Zurueck konvertieren
34         dist_img = image_conversion.array2cv(dist_img_mat)
35         #Erste Stufe fuer das Pruning: Gradientenbild berechnen und
36         #segmentieren des Gradientenbildes
37         dist_gradient = self.pruning(dist_img,1)
38         dist_gradient_mat = image_conversion.cv2array(dist_gradient)
39         dist_img_mat = image_conversion.cv2array(dist_img)
40         #Zweite Stufe fuer das Pruning: Differenzbild aus Distanzbild und
41         #segmentiertem Gradientenbild
42         diff = dist_img_mat - dist_gradient_mat
43         diff = diff * 1.0
44         diff_img = image_conversion.array2cv(diff)
45
46     return diff_img, dist_img
47

```

```

48 #Bestimmen des Gradientenbetrages des Differenzbildes
49 def pruning(self,skeleton_img,sigma):
50     skeleton_img_mat = image_conversion.cv2array(skeleton_img)
51     #Ausgabe-Array fuer das Ergebnis der Gradientberechnung
52     gradient_output = numpy.empty_like(skeleton_img_mat)
53     #Gradienten-Berechnung
54     ndimage.gaussian_gradient_magnitude(skeleton_img_mat,
55                                         sigma,gradient_output)
56     #Normalisierung
57     gradient_output /= gradient_output.max()
58     #Array ins Bild umwandeln
59     grad_img = image_conversion.array2cv(gradient_output)
60     #Schwellwertbasierte Segmentierung des Gradientbildes
61     dist_gradient_thresh = cv.CreateImage(cv.GetSize(grad_img),8,1)
62     cv.InRangeS(grad_img,0.6,1,dist_gradient_thresh)
63
64     return dist_gradient_thresh

```

A.4. Skelettierung - Thinning

A.4.1. PythonWrapper

Listing A.4: Wrapper in Python, um C++ Programme einzubinden.

```

1 import numpy as N
2 import image_conversion
3 import cv
4
5 lib = N.ctypeslib.load_library('Libpython-wrapper','.')
6
7
8 ##### Reflect image #####
9 ##### Reflect image #####
10 lib.vigra_reflectimage_c.restype=int
11 lib.vigra_reflectimage_c.argtypes = [ N.ctypeslib.ndpointer(N.uint8, ndim=2,
12     ↪ flags='aligned, contiguous, writeable'), N.ctypeslib.ndpointer(N.uint8,
13     ↪ ndim=2, flags='aligned, contiguous, writeable'), N.ctypeslib.ctypes.c_int,
14     ↪ N.ctypeslib.ctypes.c_int, N.ctypeslib.ctypes.c_int]
15 def reflectimage_band(arr, reflect_mode):
16     arr = N.require(arr, N.uint8, ['ALIGNED'])
17     arr2 = N.require(arr,N.uint8,['ALIGNED'])
18     lib.vigra_reflectimage_c(arr, arr2, arr.shape[1], arr.shape[0], reflect_mode)
19     return arr2
20
21 def reflectimage(arr,reflect_mode):
22     arr2 = N.zeros_like(arr)
23     for dim in range(arr.ctypes.shape[0]):
24         arr2[dim,:,:] = reflectimage_band(arr[dim,:,:], reflect_mode)
25     return arr2
26
27 def static():
28     #Invertierung der Farbe, damit Objekt weiss und Hintergrund schwarz
29     img = cv.LoadImage("hand.jpg")
30     grey_img = cv.CreateImage(cv.GetSize(img),8,1)
31     grey_img_mat = image_conversion.cv2array(grey_img)
32
33     for i in xrange(len(grey_img_mat[:,1])):
34         for j in xrange(len(grey_img_mat[1,:])):
35             if grey_img_mat[i,j] == 255:
36                 grey_img_mat[i,j] = 0

```

```

35         else:
36             grey_img_mat[i,j] = 1
37
38     print(lib.static_function(grey_img_mat, grey_img_mat.shape[1], grey_img_mat.shape[0]))

```

A.4.2. Thinning in C++

Listing A.5: Thinning

```

1 #include <vigra/basicimageview.hxx>
2 #include <cstdlib>
3 #include <stdlib.h>
4
5
6 #ifdef _WIN32
7 #define LIBEXPORT extern "C" __declspec(dllexport)
8 #else
9 #define LIBEXPORT extern "C"
10#endif
11
12
13
14 #include </opt/local/include/opencv2/highgui/highgui.hpp>
15 #include </opt/local/include/opencv2/imgproc/imgproc.hpp>
16 #include <iostream>
17
18 /**
19 * Perform one thinning iteration.
20 * Normally you wouldn't call this function directly from your code.
21 *
22 * @param im Binary image with range = 0-1
23 * @param iter 0=even, 1=odd
24 */
25 void thinningGuoHallIteration(cv::Mat& im, int iter)
26 {
27     cv::Mat marker = cv::Mat::zeros(im.size(), CV_8UC1);
28     for (int i = 1; i < im.rows; i++)
29     {
30         for (int j = 1; j < im.cols; j++)
31         {
32
33             uchar p2 = im.at<uchar>(i-1, j);
34             uchar p3 = im.at<uchar>(i-1, j+1);
35             uchar p4 = im.at<uchar>(i, j+1);
36             uchar p5 = im.at<uchar>(i+1, j+1);
37             uchar p6 = im.at<uchar>(i+1, j);
38             uchar p7 = im.at<uchar>(i+1, j-1);
39             uchar p8 = im.at<uchar>(i, j-1);
40             uchar p9 = im.at<uchar>(i-1, j-1);
41
42             int C = (!p2 & (p3 | p4)) + (!p4 & (p5 | p6)) +
43             (!p6 & (p7 | p8)) + (!p8 & (p9 | p2));
44             int N1 = (p9 | p2) + (p3 | p4) + (p5 | p6) + (p7 | p8);
45             int N2 = (p2 | p3) + (p4 | p5) + (p6 | p7) + (p8 | p9);
46             int N = N1 < N2 ? N1 : N2;
47             int m = iter == 0 ? ((p6 | p7 | !p9) & p8) : ((p2 | p3 | !p5) & p4);
48
49             if (C == 1 && (N >= 2 && N <= 3) & m == 0)
50                 marker.at<uchar>(i, j) = 1;
51         }
52     }
53     im &= ~marker;
54 }

```

```

55 /**
56 * Function for thinning the given binary image
57 *
58 * @param im Binary image with range = 0-255
59 */
60 void thinningGuoHall(cv::Mat& im)
61 {
62     im /= 255;
63     cv::Mat prev = cv::Mat::zeros(im.size(), CV_8UC1);
64     cv::Mat diff;
65     do {
66         thinningGuoHallIteration(im, 0);
67         thinningGuoHallIteration(im, 1);
68         cv::absdiff(im, prev, diff);
69         im.copyTo(prev);
70     }
71     while (cv::countNonZero(diff) > 0);
72     im *= 255;
73 }
74
75 /**
76 * This is an example on how to call the thinning function above.
77 */
78 LIBEXPORT int vigra_reflectimage_c( int *arr, int *arr2, const int width, const
79                                     → int height, const int reflect_method){
80
81     cv::Mat source(height, width, CV_8UC1, arr);
82     if (source.empty())
83         return -1;
84
85     cv::Mat bw = cv::Mat::zeros(source.size(), CV_8UC1);
86     cv::threshold(source, bw, 10, 255, CV_THRESH_BINARY);
87     thinningGuoHall(source);
88     cv::imshow("Thinning", source);
89     return 0;
90 }
```

A.5. Verbesserung der Skelettqualität

A.5.1. Breitensuche

Listing A.6: Implementierung der Verbesserung der Skelettqualität mittels Breitensuche.

```

1 """
2 Verbesserung der Skelettqualitaet des Distanzskeletts mittels
3 Breitensuche.
4
5 @author: Sandra Schroeder
6 """
7 import cv
8
9 """Bestimmen der Features auf dem Bild image."""
10 def calcGoodFeatures(image):
11     #Zum Speichern der Eigenwerte
12     eigenvalueImage = cv.CreateImage(cv.GetSize(image), cv.IPL_DEPTH_32F, 1)
13     tempImage = cv.CreateImage(cv.GetSize(image), cv.IPL_DEPTH_32F, 1)
14
15     corners = []
16     #Spezifikationen:
```

```

17     # Wieviele Ecken sollen gefunden werden?
18     # Welche Qualitaet?
19     # Was ist die minimale Distanz zwischen den Ecken?
20     cornerCount = 10
21     qualityLevel = 0.1
22     minDistance = 5
23
24     #Funktion zur Berechnung der Features (OpenCV Funktion)
25     corners = cv.GoodFeaturesToTrack(image,eigenvalueImage,tempImage,
26                                         cornerCount,qualityLevel,minDistance)
27
28     return corners
29
30 """Zeichnen der Features features in Bild image. Features werden mit
31 einem Kreis mit dem Radius 5 Pixel markiert."""
32 def drawFeatures(features,image):
33
34     for point in features:
35         center = int(point[0]), int(point[1])
36         cv.Circle(image,(center),5,(255,0,0))
37
38 """Startfunktion der Breitensuche. Auswahl eines Startpunktes, bei dem
39 die Breitensuche beginnen soll und dann Aufruf einer weiteren Funktion,
40 die die eigentliche Breitensuche ausfuehrt."""
41 def startConnect(features,searchDistance,img):
42     startpoint = features.pop()
43     #Breitensuche
44     neighbours = connectFeatures(startpoint,features,searchDistance,img)
45     return neighbours
46
47 """Breitensuche. Beginnt mit dem Punkt startpoint und sucht nach naechsten
48 Nachbarn in der Liste features in einer Distanz searchDistance."""
49 def connectFeatures(startpoint,features,searchDistance,img):
50
51     #Nachbarn des Punktes startpoint
52     neighbours = []
53     #Knoten, die bereits besucht wurden. Werden dann nicht mehr besucht.
54     visited = []
55     neighbours.append(startpoint)
56     while len(neighbours) != 0:
57         current = neighbours.pop()
58         visited.append(current)
59         #Koordinaten des aktuellen Punktes
60         x = current[0]
61         y = current[1]
62         #Fuer jeden Punkt in der Liste der Features pruefen, ob er in der
63         #Naehe des aktuellen (fest gewahlten) Punktes current liegt.
64         for f in features:
65             if f not in visited:
66                 #Koordinaten des Features f
67                 x1 = f[0]
68                 y1 = f[1]
69                 #Befuellen der temporaeren Liste.
70                 #Haelt die Punkte, die am naechsten dran vom
71                 #aktuellen Punkt sind.
72                 #Es zunaechst in x-Richtung, dann in y-Richtung geprueft.
73                 #Es wird ein Suchkoordinatenkreuz mit dem aktuellen
74                 #Punkt im Ursprung.
75                 if x1>=x-searchDistance and x1<=x or x1<=x+searchDistance and x1>=x:
76                     if y1>=y-searchDistance and y1<=y or y1<=y+searchDistance and
77                         y1>=y:
78                         neighbours.append((x1,y1))
79                         #Verbinden des aktuellen Punktes mit dem Nachbarn,
80                         #der im Intervall liegt.
81                         cv.Line(img, (int(current[0]),int(current[1])),
82                               (int(f[0]),int(f[1])), (255,255,255), thickness=2,

```

```
81     ↣ lineType=8, shift=0)  
82     #Nachbar wurde besucht und als besucht markiert.  
83     #Er wird dann nicht noch einmal besucht.  
84     visited.append(f)  
85  
return visited
```

A.5.2. Tiefensuche

Listing A.7: Verbinden der Zusammenhangskomponenten, die mit der Tiefensuche gefunden wurden.

```

1  """
2
3 @author: Sandra Schroeder
4
5 Idee: Alle Komponenten mit Komponente 0 verbinden.
6 Ansatz: Jede Komponente (ausser die erste) einzeln durchgehen
7 und die Kuerzeste Verbindung zwischen einem offenen Ende der Komponente
8 und einem offenen Ende einer anderen Komponente
9 suchen. Dann die Komponenten vereinen. Nachdem alle Komponenten bearbeitet
10 wurden ist sind alle Komponenten miteinander verbunden.
11 """
12
13 #features: Die berechneten Features auf dem Skelett
14 #img: Originalbild von dem das Skelett berechnet wurde
15 #searchDistance: Suchdistanz. Definiert Intervall, in dem nach dem
16 #naechsten Nachbar gesucht wird.
17
18 #####BEGINN: Erste Verarbeitungsstufe. Ausfuehren der Tiefensuche#####
19 #Zuordnung Features zu Knoten
20 nodes = []
21 for i in range(len(features)):
22     #Speichern der Knoten in einer Liste nodes
23     #Node bezeichnet eine Klasse. Featurepunkte werden wie Knoten in
24     #einem Graphen behandelt.
25     nodes.append(Node(features,i))
26 #Die Tiefensuche
27 nodes = DFS(nodes,searchDistance)
28 #Pfade zeichnen, die mit der Tiefensuche gefunden wurden.
29 drawDFSPPath(nodes,img)
30 #####ENDE: Erste Verarbeitungsstufe#####
31
32 #####BEGINN: Zweite Verarbeitungsstufe. Verbinden der
33     ↪ Zusammenhangskomponenten#####
34 OpenEnds = []
35 #Endpunkte der Zusammenhangskomponenten finden
36 OpenEnds = GetOpenEnds(nodes)
37 #Wieviele Zusammenhangskomponenten gibt es
38 NumComps = NumComponents(OpenEnds)
39
40 #Jede Komponente nacheinander durchgehen
41 for i in range(NumComps):
42     #Komponente 0 ist Referenz-Komponente.
43     if i > 0:
44         #Offene Enden aus der zur Zeit betrachteten Komponente i
45         OpenEndsInComponentToConnect = GetNodesWithComponent(OpenEnds,i)
46         #Moegliche Verbindungsknoten zu anderen Komponenten (Knoten aus OpenEnds)
47         #ungleich der aktuellen Komponente
48         OpenEndsInOtherComponents = GetNodesWithNotComponent(OpenEnds,i)
49         #Kuerzeste Verbindung zu einer anderen Komponente
50         bestlink = (0,"nil")
51         #Knoten der aktuellen Komponente, mit dem die Verbindung eingegangen wird.
52         ConnectNode = "nil"
53         # Alle Kandidaten durchgehen und die beste Verbindung waehlen.
54         for n in OpenEndsInComponentToConnect:
55             #Kuerzeste Verbindung vom aktuellen Knoten zu einer anderen Komponente
56             PossibleLink = findNextFeature(n,OpenEndsInOtherComponents)
57             #Verbindung ist kurzer als alle zuvor vorgeschlagenen
58             if bestlink[1] is "nil" or bestlink[0]>PossibleLink[0]:
59                 bestlink = PossibleLink
59                 ConnectNode = n

```

```

60         cv.Line(img, (int(ConnectNode.feature[0]),int(ConnectNode.feature[1])),
61             (int(bestlink[1].feature[0]),int(bestlink[1].feature[1])), 
62             (0,255,0), thickness=2, lineType=8, shift=0)
63     #Update der Komponente fuer die OpenEnds der aktuellen Komponente
64     for n in OpenEndsInComponentToConnect:
65         n.setComponent(bestlink[1].component)
66
67
68 #####ENDE: Zweite Verarbeitungsstufe#####
69
70 #####BEGINN: Ueberpruefen der Konnektivitaet#####
71 for n in OpenEnds:
72     ComponentNodes = GetNodesWithComponent(nodes, n.component)
73     for m in ComponentNodes:
74         m.setComponent(n.component)
75 ComponentNodes = GetNodesWithComponent(nodes, 0)
76 components = []
77 for n in ComponentNodes:
78     components.append(n.component)
79 if len(components)==components.count(0):
80     print "Pixelkonnektivitaet ist gegeben! :-) :-) :-)"
81
82 #####ENDE: Ueberpruefen der Konnektivitaet#####

```

Listing A.8: Die Tiefensuche.

```

1 """Tiefensuche. Der Aufruf der Tiefensuche erfolgt ueber die Funktion
2 DFS(nodes, searchDistance). Wenn ein Knoten gefunden wurde, der noch
3 nicht besucht wurde, wird ein neuer Baum erzeugt. Der Knoten bildet
4 dann die Wurzel des Baumes. Der Baum ist eine Zusammenhangskomponente."""
5
6 # Teil der Tiefensuche. Ein Baum.
7 def DFSVisit(node,nodes,searchDistance,Component_number):
8     node.setVisited()
9     node.setComponent(Component_number)
10    #FindNeighbours wie in der Breitensuche (Suchkreuz)
11    neighbours = findNeighbours(node,nodes,searchDistance)
12    for neighbour in neighbours:
13        if neighbour.visited is False:
14            neighbour.predecessor = node
15            node.postdecessor = neighbour
16            DFSVisit(neighbour,nodes,searchDistance,Component_number)
17
18 # Tiefensuche Startup.
19 def DFS(nodes,searchDistance):
20     component = 0
21     for i in range(len(nodes)):
22         if nodes[i].visited == False:
23             DFSVisit(nodes[i],nodes,searchDistance,component)
24             #Wenn keine Knoten mehr gefunden werden koennen, wird
25             #nach einem weiteren Knoten gesucht, der noch nicht
26             #besucht wurde (solange es noch
27             #unbesuchte Knoten gibt). Diese Komponente ist jetzt fertig.
28             #Neue Komponente anfangen.
29             component = component + 1
30     return nodes

```

Listing A.9: Finden des nächstgelegenen Punktes.

```

1 """Finde den naechstgelegenen Knoten aus einer Knoten-Liste von dem Knoten
2 node aus."""
3 def findNextFeature(node,nodeList):
4     distance_min = 10000
5     distances = []
6     distance = 0

```

```

7     node_candidate = ""
8     for n in nodeList:
9         if n is not node:
10            #euklidischer Abstand
11            distance = ((node.feature[0]-n.feature[0])**2 +
12                         → (node.feature[1]-n.feature[1])**2)**0.5
13            distances.append((n,distance))
14        #Welcher Knoten hat den minimalen Abstand zu Knoten d?
15        for d in distances:
16            node_d = d[0]
17            node_dist = d[1]
18            if node_dist < distance_min:
19                distance_min = node_dist
20                node_candidate = node_d
21                n.distance = distance_min
21
21     return (distance_min,node_candidate)

```

B. Übersicht über die Autoren der einzelnen Kapitel und Abschnitte

Name	Kapitel/Abschnitt
Johannes Böhler	Kapitel 2 Abschnitt 3.1 Abschnitt 3.1.1
Christopher Kroll	Kapitel 1 Abschnitt 3.1 Abschnitt 3.3 Abschnitt 4.1 Abschnitt 4.3 Kapitel 6
Sandra Schröder	Abschnitt 3.2 Einleitungstext Kapitel 3 Abschnitt 3.2.1 Einleitungstext Kapitel 4 Abschnitt 4.2 Abschnitt 4.4 Kapitel 5

Tabelle B.1.: Übersicht

