



Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet Graphische Datenverarbeitung

Projektseminar

Gestensteuerung einer 3D-Anwendung mittels Kinect

Autoren: Mario Janke
Peter Lindner
Patrick Stäblein

Betreuer: M. sc. Julian Meder

2. August 2017

Inhaltsverzeichnis

1	Rahmenbedingungen	2
1.1	Grundlagen und Motivation	2
1.2	Aufgabenstellung	3
1.3	Eigenschaften der Kinect	5
2	Vorüberlegungen	6
2.1	Priorisierung der Aufgaben	7
2.2	Hinzugekommene Anforderungen	8
2.3	Aufgabenverteilung im Team	9
2.4	Werkzeuge	10
3	Entwurfsentscheidungen	11
3.1	Der Master	11
3.1.1	Möglichkeiten der Master-Identifikation	11
3.1.2	Robustheit	12
3.1.3	Umsetzung	13
3.2	Gesten und ihre Wirkung	14
3.3	Zustandsmaschine	20
3.4	Robustheit und Pufferung	23
4	Bemerkungen zum Quellcode	33
4.1	Wichtige Datenstrukturen, Variablen und Funktionen	33
4.2	Ergänzungen zum Zusammenspiel und Ablaufskizze	39
4.3	Einbinden	42
5	Schlussbemerkungen	44

1 Rahmenbedingungen

Die vorliegende Ausarbeitung entstand im Rahmen des Projektseminars im Master-Studiengang Informatik an der Technischen Universität Ilmenau. Ziel und Zweck des Projektseminars ist die teambasierte Auseinandersetzung mit einem Forschungsgegenstand unter Verwendung von Fachliteratur.

Im Folgenden erläutern wir die Grundgegebenheiten, formulieren die uns gegebene Aufgabenstellungen und führen das titelgebende Tracking-System Microsoft Kinect ein.

1.1 Grundlagen und Motivation

Gegeben ist eine bereits vorhandene 3D-Anwendung, die vom Betreuer des Projektseminars, Herrn M. sc. Julian Meder, erstellt wurde. Das Programm wird zu Demonstrationszwecken z. B. am Tag der offenen Tür in den Räumlichkeiten des Fachgebiets genutzt. Innerhalb der Anwendung ist es möglich,

- Objekte einzuladen und anzeigen zu lassen sowie
- die Kamera (bzw. Kameras) zu manipulieren, d. h. zu bewegen, zu rotieren und zu zoomen.

In Erweiterung soll das Programm auch weiteren Aufgaben der Objektmanipulation dienlich sein, dies beinhaltet insbesondere das Laden mehrerer Objekte in eine Szene und die Manipulation dieser im Sinne von Translation, Rotation, Skalierung und Löschung.

Den oben genannten Demonstrationszweck nimmt das Programm über die Ankopplung des Rechners an einen 3D-Kameraaufbau wahr. Das Programm rendert zwei Ausgabefenster aus zwei verschiedenen, im Sinne der Stereoskopie angeordneten Kamerasichten. Dies wird von den beiden Projektoren im Nebenraum des Päsentationsraums genutzt um ein linkes und ein rechtes Bild von hinten auf einen Schirm zu strahlen, der in die Trennwand der beiden Räume eingelassen ist. Mittels handelüblicher aus 3D-Kinos bekannten Shutterbrillen können Benutzer und Zuschauer sodann den 3D-Eindruck wahrnehmen.

Die Steuerung des genannten Grundprogramms erfolgt durch den Vortragenden (im Weiteren zumeist Master genannt) über Tastatur und Maus bzw. einen Präsentations-

pointer. Die Unhandlichkeit sowie fehlende Immersion und Attraktivität dieser Steuerung ist eine Kernmotivation für unser Projekt. Darüber hinaus motivierend ist auch die bloße Auseinandersetzung mit der am Fachgebiet vorhandenen Technik sowie die Untersuchung der Möglichkeiten, diese zielführend einzusetzen.

1.2 Aufgabenstellung

Aus der oben angeführten Motivation heraus, ist es das Ziel unseres Projektes, die Steuerung der uns gegebenen Anwendung hinsichtlich ihrer Präsentation vor einer Zuschauergruppe zu erleichtern und intuitiv zu gestalten. Dabei wird besonderer Wert auf das Eintauchen des vorführenden Masters in die 3D-Szene gelegt. Diese Anforderung soll durch die Implementierung einer Gestensteuerung erfüllt werden. Parallel soll der zusätzliche Zweck erfüllt werden, weitere am Fachgebiet vorhandene, bislang jedoch ungenutzte Technik zu verwenden und präsentieren zu können. Dies betrifft das Trackingsystem, mit welchem die genannte Gestensteuerung umgesetzt werden soll. Hierbei standen zur Auswahl:

- ein professionelles Trackingsystem zum Tracken von Raumpunkten und
- die Verwendung einer Microsoft Kinect 2 zur Gestenerkennung.

Das genannte professionelle Trackingsystem ist ein Motion-Capture-System, das über angebrachte Marker und Wands funktioniert. Die Kinect hingegen verwendet keine am Körper angebrachten Merkmale und leitet die 3D-Information aus ihrer Sicht auf ein projiziertes Infrarotmuster ab.

Bereits sehr früh (so früh, um sie als verfeinerte Aufgabenstellung betrachten zu können) fiel die Entscheidung, die Kinect als Trackingsystem zu verwenden. Dies hatte vielerlei Gründe. Zum einen ist sie mit ihrer Herkunft aus der Spieleindustrie einem breiten Publikum bekannt und daher gegebenenfalls besser zur Präsentation geeignet. Hier kommt ebenfalls zugute, dass der Nutzer ohne jeglichen Aufwand mit dem Bedienen der Kinect beginnen kann, markerbasierte Trackingsysteme haben diesen „Plug-and-Play“-Vorteil nicht. Andererseits ist es auch der Popularität der Kinect zu verdanken, dass es eine sehr gute Quellenlage im Internet gibt. Die Kinect verfügt über eine (wenn auch nicht in großem Maße ausführliche) Online-Dokumentation und da sie mit SDK und API veröffentlicht wurde finden sich in einem doch etwas breiteren

Rahmen Lösungsdiskussionen und -präsentationen im Netz. Nicht zuletzt war sie allen Projektmitgliedern bekannt und weckte aufgrund ihrer Herkunft bereits Interesse.

Das so ausgewählte Trackingsystem soll nun also zur Implementierung einer Gestensteuerung der gegebenen Anwendung genutzt werden. Das dafür entwickelte Programm soll Folgendes leisten:

- Es soll in der Lage zu sein, sämtliche Steuerung und Manipulation, die oben beschrieben wurde durchzuführen, d. h. Kamera und Objekte steuern können.
- Die Bedienung soll sehr intuitiv und einfach sein, d. h. etwaige Gesten müssen bezüglich der ihnen zugeordneten Aktion einleuchtend und leicht auszuführen sein.
- Die Steuerung soll ihrem Zweck angemessen genau sein, am besten ist hier eine glatte 1-zu-1-Übertragung von Handbewegungen auf die Szene.
- Das Programm soll möglichst einfach eingebunden und wiederverwendet werden können.

Aus dem Anwendungszweck des Originalprogrammes heraus erwuchs die zusätzliche Anforderung, eine Mastererkennung / -verwaltung zu implementieren, das heißt ein Verfahren, das garantiert, dass auch nur die dafür vorgesehene Person das Programm steuert und niemand sonst. Im Rahmen der genannten öffentlichen Präsentation muss ausgeschlossen sein, dass ein Fremder die Kontrolle über das Programm gewinnen kann und die Vorführung dadurch – gewollt oder unbewusst – behindert. Zusätzlich soll die ausgezeichnete Person auch später wieder erkannt werden, nachdem sie etwa einen Augenblick lang nicht im von der Kamera abgedeckten Bereich war.

Damit ist die vollständige Liste der Anforderungen gegeben und wird hier zur Übersicht nochmals in ihrem Kern zusammengefasst:

Ziel ist die Entwicklung einer Software

- unter Verwendung vorhandener Technik (gemeint ist das Trackingsystem Kinect),
- die eine Gestensteuerung der gegebenen Anwendung ermöglicht und
- dabei nur einer ausgezeichneten Person diese Steuerung erlaubt.

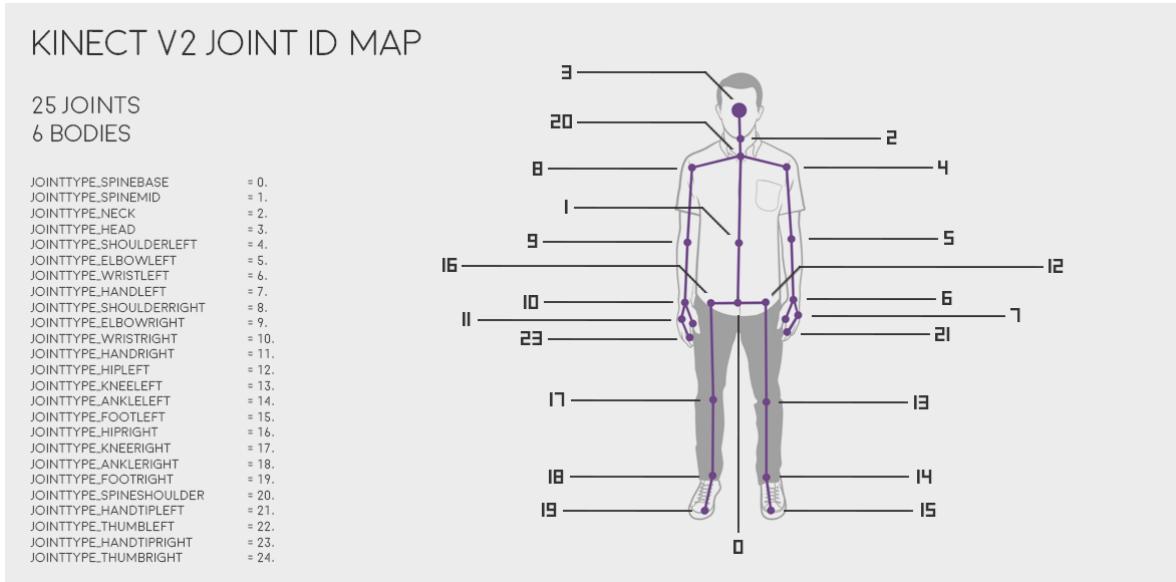


Abbildung 1: Gelenkpunkte die mit der Kinect getrackt werden können
Quelle: [7]

1.3 Eigenschaften der Kinect

Wir stellen in diesem Abschnitt nur die für uns interessanten Eigenschaften und Möglichkeiten der Kinect vor (hinsichtlich unserer Aufgabe und der Rahmenbedingungen). Die Kinect erkennt visuell den 3D-Raum vor sich. Dabei werden Personen als solche detektiert und konfidenzbasiert mit einem primitiven und grobgranularen Skelett ausgestattet. Hierbei lassen sich die Koordinaten der oben abgebildeten Gelenkpunkte mit Hilfe des Kinect SDKs abfragen. Die praktische Reichweite für die Erkennung einer Person beträgt etwa 1,2 bis 3,5 Meter. Dieses Tracking ist für bis zu sechs Personen zeitgleich möglich. Weiterhin wird für beide Hände einer getrakten Person ein „Handzustand“ erkannt, nämlich ob die Hand offen oder geschlossen ist, oder die sogenannte Lassogeste gebildet wird (etwa nur zwei Finger ausgestreckt). Kann einer Hand keiner dieser Zustände zugeordnet werden, ist ihr Status unbekannt. Diese Daten (Skelett und Status pro getrakter Person) können unter Verwendung der USB-Schnittstelle und des Kinect-SDKs abgegriffen werden. Sie werden dafür 30 mal in der Sekunde zur Verfügung gestellt.

2 Vorüberlegungen

Für unser Vorgehen zentral sind die folgenden beiden Bereiche:

(i) Die Interaktion mit dem Benutzer, d. h.

- das Entwerfen intuitiver und eingängiger sowie leicht auszuführender und gut unterscheidbarer Gesten für die verschiedenen Zwecke und
- das Auszeichnen einer getrackten Person als „Master“, der das Programm steuert.

(ii) Die technische Umsetzung, d. h.

- das korrekte Erkennen und Werten von Gesten einer ausgezeichneten getrackten Person,
- das Wiedererkennen der ausgezeichneten Person,
- das korrekte Berechnen notwendiger Bewegungsparameter bei der Steuerung und
- die Einbindung in die bestehende Applikation.

Wir stellen in diesem Abschnitt die wichtigen und unmittelbaren Beobachtungen vor, die sich aus der Aufgabenstellung und dem Versuchsaufbau ziehen lassen.

Ausgehend von der Aufgabenstellung kann man abstrahierend zwischen zwei primitiven Steuerungsmodi unterscheiden:

- einem Modus, in dem die Kamera verschoben und rotiert werden kann &
- einem Modus, in welchem Objektmanipulationen möglich sind.

Der Benutzer sollte sich zu jedem Zeitpunkt nur in maximal einem dieser Modi aufhalten, d. h. gleichzeitige Kamera- und Objektmanipulationen sind ausgeschlossen. Diese Vereinfachung treffen wir, da damit weniger komplexe Gesten benötigt werden und eine solche simultane Manipulation keine praktische Relevanz besitzt. Für Manipulationen, die man sowohl für die Kamera, als auch für Objekte haben will, bietet dies zudem eine geeignete Kapselung, da z. B. Rotationsparameter berechnet werden und dann nur entschieden werden muss, ob sie auf die Kamera oder ein Objekt angewendet werden, je nach Modus. Dies kann, falls so umsetzbar, die Gesamtzahl nötiger Gesten reduzieren.

Die Kinect ermöglicht ein Tracking des gesamten Körpers für mehrere (genauer sechs) Personen. Wir beschränken uns jedoch nur auf einen Teil dieses Spektrums:

- Wir benötigen nur eine Person, die die Anwendung (möglichst ungestört) steuert. Eine genauere Auswertung der restlichen Personen, ihrer Skelette etc. ist zumeist unnötig.
- Die in unserem Anwendungsfall intuitiven Gesten werden ausschließlich mit den Händen (bzw. Armen) durchgeführt. Dazu gibt es einige Constraints in den Gestendefinitionen, die noch eine Reihe weiterer Körpermerkmale verwenden.
- Ebenso wird für die Mastererkennung ein bestimmtes Set von Körpermerkmalen verwendet.

Primitive Erkennungsmöglichkeiten eines Masters kann man etwa aus der Entfernung der getrackten Personen zur Kamera und der Position der Personen im Raum gewinnen, in der oben angedeuteten Vergleichsvariante wird eine Sammlung von Körpermerkmalen zusammengestellt, die später zum Abgleich dienen kann. Genauere Erklärungen folgen weiter unten.

Intuitive Gesten für Verschiebungen imitieren das Verschieben eines großen Gegenstands, etwa einer imaginären Box, sodass hier etwa ein Verschieben der flachen Hand in der Luft naheliegt. Für eine intuitive Drehgeste eignet sich die Vorstellung eines imaginären Lenkrads, genauer gesagt einer Lenkkugel, bei der die Rotation um eine Raumachse nach dem Lenkradprinzip erfolgt. Eine intuitive Geste zur Objektauswahl ist offenbar eine Greifgeste.

Die genannten Gesten und Umsetzungen folgten unseren Überlegungen nach ersten Tests und wurden im Laufe des Projekts noch überarbeitet.

2.1 Priorisierung der Aufgaben

Da eine primitive Mastererkennung nach Entfernung entlang der Tiefenachse leicht zu implementieren ist, konzentriert sich die Arbeit zunächst auf die Gesten, genauer auf die Steuerung der Kamera. Im ersten Schritt sind Gesten und ein Rechenmodell zu entwerfen, das sich zur Steuerung eignet. Dies ist dann für Verschiebung und anschließend Drehung der Kamera zu implementieren. Anschließend findet selbiges für die Objektmanipulation statt. Ist dies in einem zufriedenstellendem Maße gegeben,

kann die Mastererkennung ausgebaut werden. Steht die Grundfunktionalität, können Optimierungen und Verbesserungen diskutiert werden.

2.2 Hinzugekommene Anforderungen

Wie bei solch praktisch orientierten Projekten gewöhnlich, fielen in den Tests und der Arbeit mit dem Programm bestimmte Features auf, die einen äußerst positiven Effekt auf die Bedienerfahrung des Programmes hätten. So kamen zu den in Abschnitt 1.2 genannten Aufgaben zusätzliche Anforderungen hinzu.

In der original angedachten Variante einer Masterfestlegung musste ein Knopf betätigt werden, der den Einspeichervorgang startet. Dies kann ein Partner des Vortragenden am Rechner vornehmen oder aber der Vortragende selbst per Präsentationspointer. Eine Verbesserung dieses Prinzips ist eine Art Selbstauslösermechanik. Dazu soll der vorgesehene Master das Einspeichern am Rechner anstoßen können, das jedoch noch nicht beginnt, solange nicht eine Person (in passender Pose) zum Einspeichern bereit steht. Dies gibt ihm die Möglichkeit, sich ins Aufnahmefeld zu bewegen um dort als Master festgelegt zu werden. Die Notwendigkeit einer zweiten Person, die den Rechner bedient (falls kein Präsentationspointer vorhanden) entfällt damit.

Ein weiterer Punkt ist das großräumigere Navigieren durch die Szene. Entsprechend der oben genannten Ideen zur Kamerabewegung ist für das Zurücklegen einer größeren Strecke ein ständiges Nachgreifen nötig. Dies ließe sich durch Erschaffen eines „Flug-Modus“ vermeiden. Konzeptuell soll sich die Kamera dabei solange nach vorne bewegen, wie eine bestimmte Geste präsentiert wird. Dazu erscheinen nach vorne ausgestreckte Arme einleuchtend, wobei mit Kippbewegungen und der Zeigerichtung auch die Flugrichtung manipuliert werden kann.

Hinsichtlich der Masterfestlegung fiel weiterhin auf, dass es für den Anwender nützlich ist, die Länge (und damit die Genauigkeit) der Einspeicherung beeinflussen zu können. Dafür eignet es sich, einen Zusammenhang zwischen der Länge des Tastendrucks für Einspeicherung und der Einspeicherungszeit selbst herzustellen.

Schließlich fiel bei der Arbeit schnell auf, dass die Entwicklung einer solchen Anwendung sehr auf visuelles Debugging angewiesen ist. In der Version unseres Programmes das am Ende ausgeliefert werden soll gab es nach den A-priori-Überlegungen keinerlei Rückkopplung zum Hauptprogramm, die über den Erfolg oder Misserfolg von Einzelschritten Auskunft hätte geben können. So kam als Anforderung hinzu, ein Eventsys-

tem in Form von Stubs zu implementieren, das später dazu ausgebaut werden kann, bestimmte Statusmeldungen von der Gestensteuerung und Mastererkennung, wie etwa „Master festgelegt“, „Master verloren“ oder die Angabe des aktuellen Steuerungsmodus zurückzugeben.

2.3 Aufgabenverteilung im Team

Das Projekt wurde zu viert begonnen, wobei einer der Teilnehmer gleich zu Beginn wieder absprang. Die Aufgabenverteilung im Team wurde ab der Einarbeitungsphase mit dem Kinect-System dynamisch vorgenommen. So kristallisierten sich im Laufe der Zeit diverse Zuständigkeitsbereiche heraus, die grob so umrissen werden können:

Mario Janke kümmerte sich vor allem um die Mathematik im Hintergrund, präziser um die diversen Berechnungen von Parametern aus den vorliegenden Kinect-Daten. Als klar wurde, dass das Projekt aufgrund der Eigenschaften der Kinect zusätzliche Robustheitsmechanismen benötigte, wurde das Management unserer Puffer zusätzlicher Aufgabenbereich.

Peter Lindner sorgte vorderrangig für die Strukturierung des Codes. Dies erstreckt sich auf die Umsetzung des objektorientierten Programmierparadigmas mit der Ausarbeitung und Erstellung der Klassenhierarchie. Die im Zuge dessen entstandene Zustandsmaschine wurde in der Folge von ihm verwaltet. Da für deren Existenz zu großen Teilen das beabsichtigte Wirkungsprinzip der Gesten war, schloss sich dem Aufgabenbereich das Gestenmanagement an.

Patrick Stäblein war für das Ansammeln grundlegender Informationen zur Kinect – gerade in der Anfangsphase – verantwortlich und programmierte einen Großteil der Mastererkennungsmechanismen in der zweiten Phase des Projekts.

Parallel zur Arbeit am Programmcode entstand das vorliegende Dokument und später die Vorbereitung der Abschlusspräsentation. Dabei orientierten sich die vom jeweiligen Projektmitglied bearbeiteten Themengebieten an ihren Zuständigkeiten und damit Wissenschwerpunkten aus der Programmierung.

2.4 Werkzeuge

In diesem Abschnitt sollen die Tools genannt und erklärt werden, die für die Entwicklung unserer Software vordergründig waren.

Das im Zentrum stehende Trackingsystem *Microsoft Kinect Version 2* war durch das Fachgebiet gegeben. Zur Arbeit damit stand ein Raum bereit, der über die Kinect und einen 3D-Kamera-Aufbau zur Projektion verfügte. Ferner durfte die Kinect für Heimtests auch ausgeliehen werden.

Mit dem Kinect SDK wird eine Softwarelösung namens *Kinect Studio* ausgeliefert, die sich bei der Kinect-Programmierung zum visuellen Debugging eignet. Im Kinect Studio können die verschiedenen Sensoraufnahmen der Kinect nebst der interpretierten Skelette und Hand-States sowie der festgestellten Tiefensituation betrachtet werden. Diese Features können getoggelt oder umgeschaltet werden. Die Tiefenkarke wird per Falschfarbendarstellung und, sofern erwünscht, sogar dreidimensional präsentiert. Das Kinect Studio erwies sich als sehr hilfreich, um die Güte der Kinect-Daten zu überprüfen und zu erkennen, ob die Kinect einen Teil des Aufnahmebereiches fehlinterpretiert. Dies war zum Teil notwendig, um bei der Programmierung schnell und einfach zwischen Fehlern des Programmes und fehlerhaften Kinect-Daten unterscheiden zu können.

Ebenfalls zum Kinect-SDK gehörig ist der sogenannte *Visual Gesture Builder*, mit dem Gesten(folgen) aufgenommen und in Programme eingespeist werden können. Da wir uns (s. u.) für einen anderen Weg der Gestenimplementierung entschieden haben, fanden hiermit nur kleinere Tests in der Anfangsphase statt.

Die Kinect-API kann mit JavaScript, C++ oder C# verwendet werden. Da das uns im Rahmen der Aufgabenstellung übermittelte Programm, für das unsere Gestensteuerung gedacht ist, in C++ geschrieben war, verwendeten wir aus Kompatibilitäts- und Einheitlichkeitsgründen heraus ebenfalls C++ und entwickelten und debuggten mit dem *Microsoft Visual Studio 2015* unter *Microsoft Windows 10*.

Zur Versionsverwaltung nutzten wir das Kommandozeilentool git mit [GitHub](#).

3 Entwurfsentscheidungen

3.1 Der Master

Der Master ist die Person (unter den getrackten Personen), der es obliegt, die Anwendung zu steuern, d. h. in unserem Anwendungsfall der Präsentation ist der Master der Präsentierende. Es muss gewährleistet werden, dass nur der Master das Programm steuert und dabei von weiteren Personen im Raum nicht (bzw. nicht ohne weiteres) gestört werden kann. Die Erkennung muss robust gegen Jittering der Kinectdaten sein.

3.1.1 Möglichkeiten der Master-Identifikation

Grundsätzlich kamen für die Festlegung des Masters zwei Ideen auf. Zunächst sollte bei jedem Frame, die getrackte Person, identifiziert werden, die der Kinect bzgl. der z-Koordinate am nächsten ist und diese als Master festgelegt werden. Der Master könnte hierbei bei jedem Frame zwischen den getrackten Personen wechseln.

Die zweite Möglichkeit war die Festlegung des Masters auf eine bestimmte Person, von der zunächst bestimmte Identifikations-Merkmale eingespeichert werden und die dann anhand dieser als Master reidentifiziert werden kann. Sofern diese Festlegung erst einmal geschehen ist, bleibt diese Person Master, selbst nachdem sich diese zwischenzeitlich in einem ungetrackten Zustand (beispielsweise beim Herausgehen aus dem getrackten Bereich) befunden hat und dann wieder als getrackt erkannt wird. Bei einer Recherche, welche Merkmale sich aus den von der Kinect gelieferten Daten extrahieren lassen ließen, um hierfür in Frage zu kommen, stießen wir hierbei auf verschiedene Möglichkeiten, von denen einige jedoch aufgrund ihrer Unpraktikabilität ausschieden (Erkennung anhand des Gangs oder anhand der Stimme würde bei unserer Anwendung keinen Sinn machen, da die Master-Person während der Bedienung kaum umher läuft und diese hierfür nicht zu sprechen braucht). Schließlich stiessen wir auch auf Verfahren die die Skelettdaten der Kinect zur Identifikation nutzen. Dies schien die für unsere Zwecke praktikabelste Lösung zu sein, wenngleich unser Ansatz die Skelettdaten zu nutzen im Vergleich zu denjenigen in den gefundenen Arbeiten stark vereinfacht wurde.

Grundlegendstes Prinzip für die Identifizierung einer Person ist hierbei das Auslesen der Skelettkoordinatenpunkte mit Hilfe des Kinect SDKs und daraus der Ermittlung diverser Längen als Körperproportionen mittels der Berechnung des euklidischen Abstands

zwischen den entsprechenden Skelettpunkten. Beispielsweise wird die rechte Oberarmlänge als Abstand zwischen dem rechten Schulterpunkt und dem rechten Elbogenpunkt ermittelt. Weitere Körperproportionen die verwendet wurden sind unter anderem die Schulterbreite, Hüftbreite, Unterarmlänge, Abstand zwischen Hals und Kopf.

3.1.2 Robustheit

Nach einigen Experimenten mit den Körperproportionen wurde festgestellt, dass einige Proportionen übermäßig große Abweichungen aufwiesen, wenn die gleiche Person in unterschiedlichen Posen mit der Kinect vermessen wurde bzw. dass sich einige Skelettpunkte "verschieben", wenn man mit einem Körperteil darüber fährt. (Beispielsweise die gemessene Schulterbreite, welche sich signifikant zwischen der Standard-Pose und einer Pose bei der die Arme über den Kopf gestreckt sind unterschied.) Aus diesem

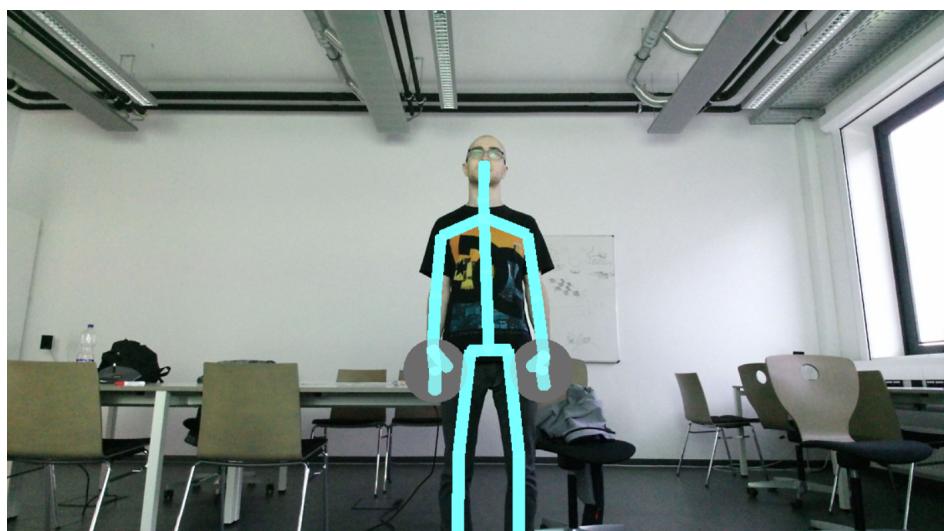


Abbildung 2: Die Standardpose, in der sich eine Person befinden muss, damit sie vermessen werden kann.

Grund wird die Vermessung einer Person nur durchgeführt, wenn sich die entsprechende Person über eine bestimmte Anzahl von Frames (etwa 20) durchgehend in der festgelegten Standard-Pose befindet. Falls während der Sammlung das Einhalten der Standard-Pose unterbrochen wird, wird die Sammlung erneut begonnen.

Ein weiteres Problem das zu lösen war bestand darin, dass selbst beim Stillhalten, jedoch hauptsächlich bei der Bewegung einer getrackten Person, ein Skelettpunkte kurzzeitig auf einen anderen weiter entfernten Raumknoten springen konnten, was teils

zu unrealistischen Messungen einer Körperproportion für einzelne Frames führte (z.B. Messung einer Oberarmlänge von 2 Metern). Um derartige Ausreißer zu eliminieren wird der während der Vermessung einer Person zur Einspeicherung als Master über mehrere Frames der Mittelwert sowie die Standardabweichung jeder Körperproportion berechnet und jeder Wert der nicht innerhalb des Intervalls um den Mittelwert mit der Ausdehnung der Standardabweichung liegt (d.h. des Intervalls (Mittelwert-Standardabweichung*Faktor, Mittelwert+Standardabweichung*Faktor)) ignoriert. Weiterhin wird darauf geachtet, dass alle für die Vermessung relevanten Körperproportionen mit ausreichender Konfidenz von der Kinect getrackt werden. Die Kinect liefert hierfür für jeden Skelettpunkt einen von drei möglichen Konfidenzwerten, der angibt, wie wahrscheinlich die von der Kinect gelieferte Koordinatenwerte für diesen Punkt dem tatsächlichen Wert entsprechen. (Die drei Konfidenzwerte heißen Tracked, Inferred, NotTracked; wobei Tracked die höchste Konfidenz und NotTracked die niedrigste Konfidenz für einen Skelettpunkt bezeichnet). Ist in einem Frame der Konfidenzwert von einem der beiden Skelettpunkte aus denen eine Körperproportion berechnet wird nicht Tracked, so wird diese Körperproportion für diesen Frame nicht berücksichtigt bzw. mit Null gewichtet.

3.1.3 Umsetzung

Eingebettet in unsere Hauptschleife geht die Master-Identifikation folgendermassen von statthen: Beim Start des Programms, ist zunächst einmal die Person Master die bzgl. der Kinect die geringste z-Koordinate aufweist. Hierfür wird bei jedem Schleifendurchlauf abgefragt, welche der 6 (potenziell) getrackten Personen, den geringsten z-Wert hat. Der Master wird somit bei jedem Frame neu bestimmt.

Durch die Betätigung einer vorher festgelegten Taste wird schließlich die Master-Festlegung auf eine bestimmte Person aktiviert. Hierzu werden die Körperproportionen der Person, die als erstes in Standardpose erkannt wird, über mehrere Frames aus den Skelettdaten extrahiert, gepuffert und schließlich aus diesen für jedes Körpermerkmal der Mittelwert sowie die Standardabweichung berechnet. (Sofern beim Tastendruck noch keine Person im getrackten Bereich war, wird gewartet bis eine Person getrackt wird und diese die Standard-Pose einnimmt.) Der Mittelwert und die Standardabweichung werden dazu verwendet Ausreißer zu identifizieren und zu entfernen, um schließlich die Mittelwerte erneut ohne die Ausreißer zu berechnen. Neben diesen Mittelwerten für die nun als Master festgelegte Person, wird außerdem die von der Kinect vergebene

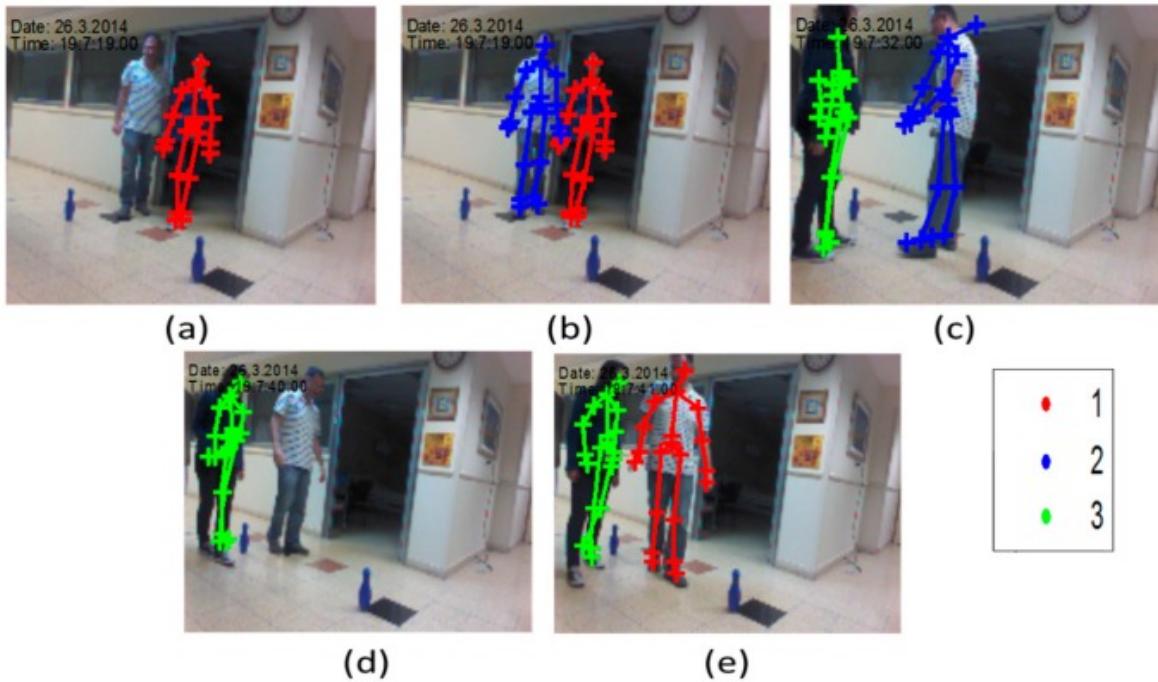


Abbildung 3: Neuzuordnung von IDs nach zwischenzeitlichem „Verlust“ von Skeletten,
Quelle:[2]

ID der Person eingespeichert. Diese bleibt solange dieser Person zugeordnet, solange sie durchgehend getrackt wird. Verlässt der Master den getrackten Bereich oder wird anderweitig vorübergehend nicht getrackt (beispielsweise wenn dieser von einer anderen Person verdeckt wird) und kommt danach wieder in den getrackten Bereich zurück, hat dieser eine neue ID. In diesem Fall muss der Master neu identifiziert werden.

Solange der Master noch nicht gefunden wurde, werden hierfür für jede Person die in Standard-Pose erkannt wird die Körperproportionen über eine gewisse Anzahl (etwa 20) an Frames mit denen der für den Master eingespeicherten Werte verglichen. Schließlich wird aus den frameweise berechneten Abweichungen der Durchschnitt berechnet. Ist diese durchschnittliche Abweichung kleiner als ein im Programm festgelegter Wert ist diese als Master identifiziert worden. Diese kann nun mit der Steuerung des Programms fortfahren.

3.2 Gesten und ihre Wirkung

Für die eingangs erwähnte Aufgabenstellung war es notwendig, bestimmte Programm-funktionalitäten mit Gesten zu verbinden. Einerseits hätte die Möglichkeit bestan-

den, mit dem Kinect-eigenen Visual Gesture Builder Gesten aufzunehmen und einzulernen. Diese Gesten werden dann als Datenbank ins Programm geladen und bei Vorführung erkannt. Dies erspart natürlich primitive aber umständliche Low-Level-Erkennungsmechanismen. Weiterhin sind hierdurch einige weiterführende Möglichkeiten gegeben wie etwa die Rückgabe, bis zu welchem Punkt eine Geste bereits ausgeführt wurde (in Bezug zur Gesamtgeste, d. h. beispielsweise wieviel Prozent einer Armbewegung vordefinierte Länge bereits ausgeführt wurde). Anderseits wiederum können durch die Kinect-Rohdaten auch eigene Erkennmechanismen implementiert werden. Dies bietet dem Programmierer die vollständige Kontrolle über seinen Gestenkatalog. Änderungen können kurzfristig und schnell vorgenommen werden und für einfache Projekte ist die Zusatzfunktionalität, die der Visual Gesture Builder gestattet nicht vonnöten, der eher für komplexere Gestenfolgen ausgelegt zu sein scheint. Demgegenüber ist für diese Direktimplementierung von Gesten aber die bereits erwähnte Low-Level-Erkennung zu implementieren, d. h. ein Extrahieren von Bewegungen und Bewegungsrichtungen aus den Skelett- und Gelenkdaten, die die Kinect bestimmt. Dennoch entschieden wir schließlich uns kraft dieser Gegenüberstellung (nebst einigen Versuchen mit dem Visual Gesture Builder, die uns nicht von seinem Mehrwert in unserer konkreten Anwendungssituation überzeugen konnten) für eine direkte Gestenerkennung.

Auch bei der Low-Level-Erkennung gibt es jedoch verschiedene Ansätze bzw. Ausprägungen. Es ist sogar das Implementieren nicht ganz primitiver Gestenfolgen möglich, indem eine Geste zeitlich und räumlich in verschiedene Segmente unterteilt wird. Dies sei an einem Beispiel erläutert: Es soll eine Winkgeste der rechten Hand erkannt werden. Die Geste wird in zwei Segmente geteilt. Ein Wechsel zwischen den Segmenten findet statt, wenn die horizontale Position der Hand und des Ellenbogens wechseln. Wird dieser Übergang dreimal in Folge erkannt, so wurde die Winkgeste präsentiert. Eine genauere Auseinandersetzung mit der Aufgabenstellung und unseren Vorstellungen von intuitiven Gesten für die zu realisierenden Funktionalitäten zeigte jedoch auf, dass auch eine Segmenteinteilung von Gesten für das Projekt nicht notwendig ist. Statt dessen sind die gegebenen Aufgaben in ihrer Struktur simpel genug, um die verschiedenen Wirkungen mit diskreten Gesten zu erzeugen, d. h. es genügt die Erkennung einer Geste durch bestimmte Zustände der Kinect-Rohdaten zu einem einzigen Zeitpunkt. Um die Wirkung jedoch zu erzielen, ist natürlich auch eine Betrachtung der Geste über mehrere Frames notwendig.

Im Folgenden erklären wir unseren Gestenkatalog und gehen dabei darauf ein, was der

Benutzer vorführen muss, damit die Geste erkannt wird und wie die Geste genutzt wird, um in der Anwendung die Kamera oder Objekte zu manipulieren:

TRANSLATE_GESTURE (siehe Abb. 4)

Der Benutzer hat beide Hände geöffnet, mit den Handflächen zur Kamera (wichtig ist nur, dass die Kinect beide Hände als offen erkennt, die genaue Haltung ist dabei egal). Ein paralleles Verschieben der beiden Hände in eine Richtung bewirkt ein zur Bewegungsgeschwindigkeit proportionales Verschieben der Kamera in diese Richtung.

Diese Geste war allen Projektteilnehmern unmittelbar einleuchtend und intuitiv und bedurfte keiner weiteren Diskussionen.

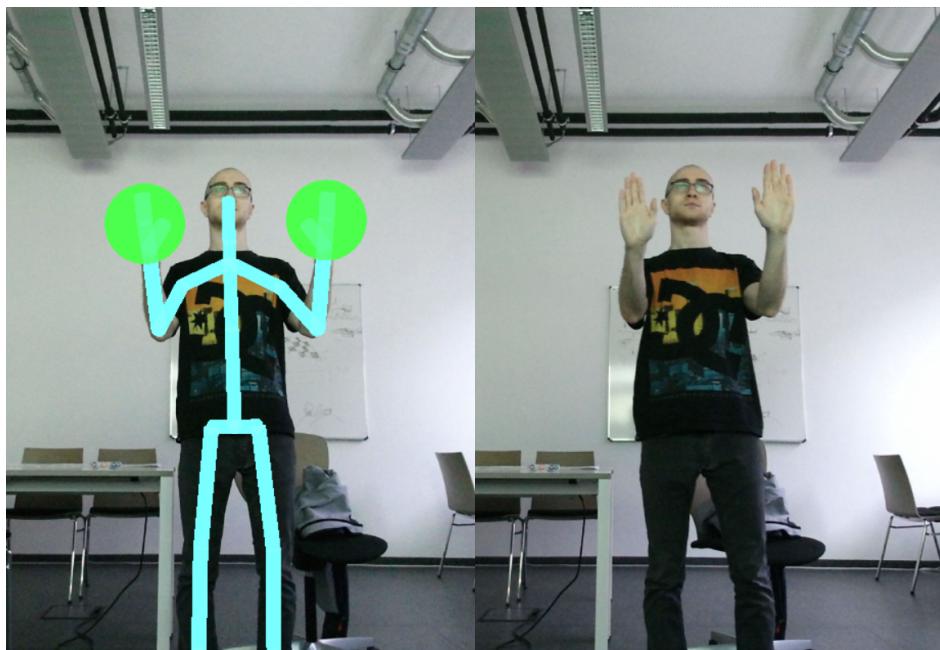


Abbildung 4: Die (Kamera-)Translations-Geste, mit und ohne eingezeichnetes Skelett und HandStates.

ROTATE_GESTURE (siehe Abb. 5)

Der Benutzer hat beide Fäuste geballt. Dann bewirkt eine gleichzeitige Bewegung der Hände auf einer Kreisbahn eine Rotation der Kamera um die Senkrechte des zugehörigen Kreises. Dies ist genau die intuitive Art der Steuerung, mit der man etwa ein Aussichts- bzw. Münzfernrohr steuern würde.

Wie die TRANSLATE-Geste war auch diese Geste von Anfang an im Team umstritten.

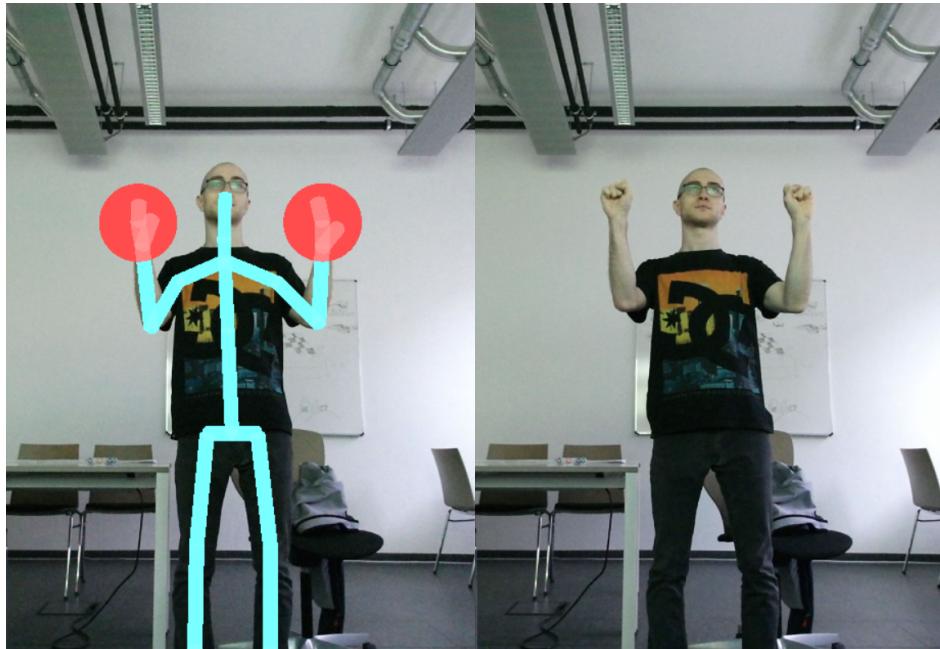


Abbildung 5: Die (Kamera-)Rotations-Geste, mit und ohne eingezeichnetes Skelett und HandStates.

GRAB_GESTURE (siehe Abb. 6)

Zunächst war angedacht, dass die Objektmanipulation dieselben Gesten verwendet wie die Kameramanipulation und die Unterscheidung, was manipuliert wird durch einen globalen Zustand gefällt wird. Bei näherer Betrachtung dieses Ansatzes und ersten Tests dessen fiel auf, dass es so schwierig ist, zwischen Kamera- und Objektmanipulation zu wechseln. Weiterhin schien es während des Testens weniger intuitiv als zuvor angenommen, ein Objekt auf diese Art und Weise zu manipulieren. Es mussten also andere Ansätze gefunden werden.

In das Problem der Objektmanipulation eingeschlossen ist das Problem des Object-Pickings, d. h. die Auswahl des zu manipulierenden Objekts vom Bildschirm. Auch dies wäre mit der oben beschriebenen Methode, die die Gesten der Kameramanipulation verwendet, nur schwierig und umständlich realisierbar gewesen. Wir näherten uns dem Finden eines neuen Weges diesmal auf einem anderen Weg, nämlich nicht über die Manipulation, sondern über das Picking des

Objekts. Schnell einigten wir uns auf das Greifen eines Objekts (eine Hand ist erhoben und geschlossen – dies motiviert auch den Namen „GRAB“-Geste) als intuitivste Möglichkeit dafür. Von der Idee her sollte ein Hin- und Herbewegen dieser „Kontroll-Hand“ auch das Objekt hin- und herbewegen. Nachdem dies zufriedenstellend eingebaut war, widmeten wir uns der Objektrotation, was schnell eine fundamentale Schwäche dieser Geste offenbarte: Die Rotation des Objekts sollte der Rotation der geschlossenen Hand folgen, jedoch ist die Erkennung der Rotation einer geschlossenen Hand durch die Kinect viel zu schlecht, um an dieser Stelle sinnvoll Verwendung zu finden.

Die Ergebnisse wurden direkt sehr gut, als wir dazu übergingen, die GRAB-Geste durch eine gehobene und offene (!) Hand zu definieren, da die Kinect so – wie auch naheliegend – viel besser erkennen kann, wie die Handfläche gekippt bzw. gedreht ist. Intern behielten wir jedoch den semantischen Namen „GRAB“-Geste bei.

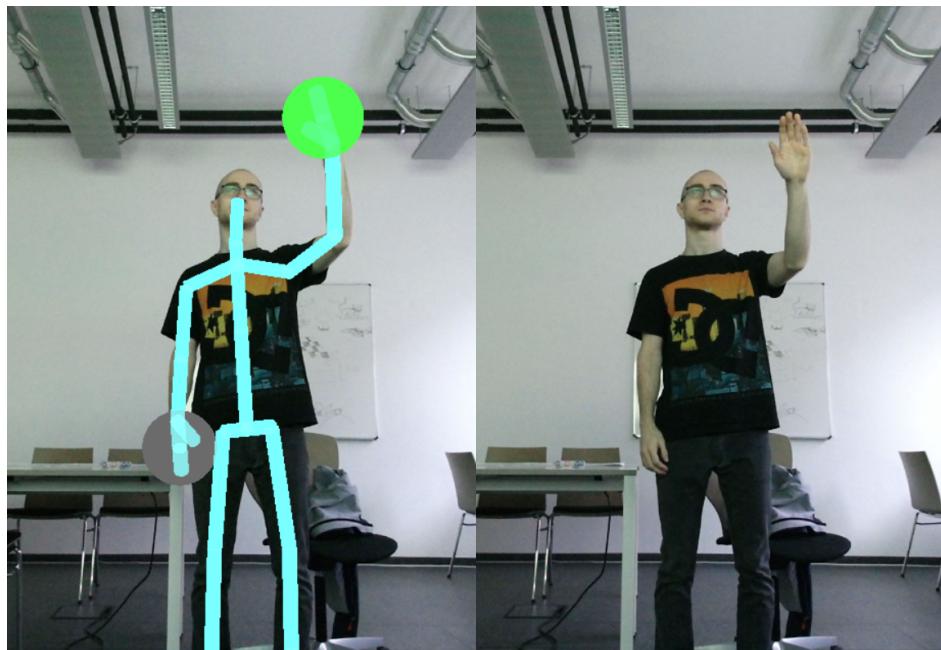


Abbildung 6: Die Objektmanipulations-Geste, mit und ohne eingezeichnetes Skelett und HandStates.

FLY_GESTURE (siehe Abb. 7)

Im Rahmen der Tests mit einem Beispielobjekt wurde schnell deutlich, dass es

auch eine einfache Möglichkeit geben sollte, sich über weitere Strecken durch den Raum zu bewegen, ohne dabei ständig zwischen dem Vorführen einer Geste und einem „Nachgreifen“ wechseln zu müssen. Als sinnvoll erschien hier, dass das Vorführen einer besonderen Geste bewirkt, dass die Kamera losfährt und erst anhält, wenn die Geste nicht mehr präsentiert wird.

Die FLY-Geste entspricht dem Ausstrecken beider Arme vor den Körper, sodass sich die Hände mehr oder weniger am selben Punkt im 3D-Raum befinden. Passiv findet bei dieser Geste im Programm eine Bewegung nach vorne statt. Durch Schwenken der Arme soll der Nutzer dabei die Richtung der Bewegung beeinflussen können, d. h. ein Zeigen der Arme nach oben bewirkt, dass die Bewegung immer weiter nach oben gezogen wird, während man mit einem Zeigen nach links oder rechts eine Kurve fliegen kann. Dabei bestimmt der Ausschlag der Arme beim Zeigen (verglichen mit der Ausgangsposition, in der beide Arme genau nach vorne gerichtet sind) die Stärke der Richtungsänderung. Um eine sogenannte Fassrolle durchzuführen oder sich in Kurven legen zu können, kann der Nutzer nebenbei seine Schulterpartie in die entsprechende Richtung kippen. Insgesamt ist die Steuerung des Flugmodus in ihrem Funktionsumfang damit ähnlich zu üblichen Steuerungen von Flugzeugen in Actionsspielen oder Simulationen.

Nicht zuletzt daher ist die assoziierte Geste für den Nutzer angenehm und intuitiv und ermöglicht eine im Gegensatz zur Bewegung über Drehen und Schieben einfache Möglichkeit, sich etwa durch ein System von Gängen in einer 3D-Szene zu bewegen und generell Strecken zurückzulegen, statt Objekte zu betrachten. Ein, wenn auch in der Art der Gesamtanwendung begründetes Problem, ist jedoch, dass das Ausführen dieser Geste durch die nach vorne ausgestreckten Arme schnell anstrengend wird. Für Anwendungen, in denen – nicht wie in unserem Fall – der Hauptfokus tatsächlich auf dem Überwinden von längeren Strecken liegt, etwa in einem Rennspiel o. Ä., wäre es vermutlich notwendig, diesem Punkt erneut Beachtung zu schenken. Für unsere Aufgabenstellung ist die genannte Gestenvariante jedoch völlig ausreichend und besticht durch Intuition und Immersion.

UNKNOWN Dies enthält alles, was als keine der anderen Gesten erkannt wird. Die Kamera und geladene Objekte sollen, solange diese Geste gezeigt wird, stillstehen. Neben der naheliegenden Motivation, dass der Nutzer die Szene gegebenenfalls

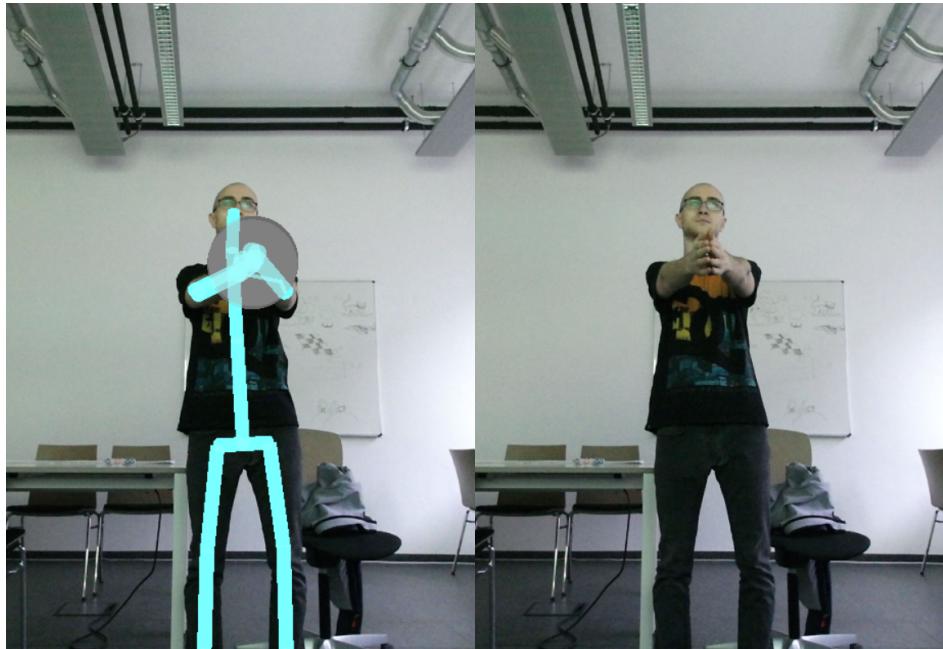


Abbildung 7: Die Flug-Geste, mit und ohne eingezeichnetes Skelett und HandStates.

auch bei Ruhe betrachten möchte, dient diese „Geste“ (oder besser „Nicht-Geste“) darüber hinaus noch einem anderen Zweck. Sie kann in andere Gesten, wie beispielsweise Translationen, eingebaut werden, um diese aufzubrechen und „nachgreifen“ zu können. Erst dies gestattet dem Nutzer, während der Bedienung an Ort und Stelle stehen bleiben zu können.

Tests mit der Kinect haben ergeben, dass es notwendig ist, bei derartig selbst implementierten Gesten auch eigene Robustheitsmechanismen einzubauen, die die Geste-nerkennung gegen Schwankungen der Kinecterkennung (etwa des Status einer Hand) abhärten. Für genauere Informationen hierzu verweisen wir auf Abschnitt 3.4.

3.3 Zustandsmaschine

Das Programm besteht aus zwei Grundmodi der Manipulation: Einerseits der Manipulation der Kamera und andererseits jener des Objekts. Wir können diese beide Modi als zwei Superzustände auffassen, innerhalb derer sich wiederum unterscheidet, auf welche Art und Weise wir manipulieren. Die Zustandsmaschine dient einerseits der Kapselung und Modularisierung der von uns bereitgestellten Funktionen und bildet andererseits die Struktur unserer Manipulationsmodi abstrakt ab.

Die Zustandsmaschine befindet sich zu jedem Zeitpunkt in einem Zustand. In diesem Zustand findet eine Berechnung der Parameter statt, die unser Programm zurückgibt, die wiederum die Manipulation beschreiben, die ausgeführt werden soll. Dies geschieht durch Auswertung der gesehenen Geste und die Berechnung entscheidender Größen, u. U. unter Einbeziehung der Werte vergangener Frames. Schließlich erfolgt basierend auf der präsentierten Geste ein Zustandswechsel am Ende eines Berechnungsschritts. Die Zustandsmaschine ist in Abb. 8 zu sehen. Im Folgenden erklären wir die Zustände, ihre Semantik und die enthaltenen Berechnungen etwas näher:

IDLE Dieser Zustand entspricht dem Initialzustand unserer Zustandsmaschine. Er ist eine Art Default-Zustand, in dem keine Kamera- und auch keine Objektmanipulation (genauer: keine Berechnung überhaupt) vorgenommen wird. Der Zustand wird betreten, wenn keine der vordefinierten Gesten sicher genug erkannt wurde. Durch Ausführung der entsprechenden Gesten gelangt man zurück in die anderen Zustände.

CAMERA_TRANSLATE Dieser Zustand gehört zur Kameramanipulation. In ihm werden gemäß der oben erklärten Geste die Parameter zur Kamerabewegung bestimmt. Ziel ist die direkte Übertragung der Handbewegungen des Benutzers auf die Kamerabewegung, die im lokalen Raum der Kamera stattfindet. Wir berechnen dazu aus den gepufferten Positions値en von linker und rechter Hand die diskrete Ableitung, die uns ein Maß für die Geschwindigkeit der Bewegung liefert. Ebenso erhält man daraus die Richtung, in die die Hände bewegt wurden. Die Geschwindigkeit wird dann mit der vergangenen Zeit zwischen dem aktuellen und dem vorhergehenden Frame multipliziert, um wieder einen Entfernungswert zu erhalten. Daraus entstehen die Translationsparameter für die x -, y - und z -Richtungen, die für diesen Zustand unsere motionParameters definieren.

CAMERA_ROTATE Dieser Zustand gehört ebenfalls zur Kameramanipulation. Analog zu oben wird hier die Rotation vorbereitet. Die Rotation ist konzeptionell nahe in der Translation. Während die Geste aktiv ist, wird eine Achse zwischen linker und rechter Hand des Benutzers bestimmt. Bewegt der Benutzer die Hände, so verändert sich die Achse. Ein Quaternion, der die Achsen ineinander überführt, wird berechnet, und analog zur Translation als Geschwindigkeit

interpretiert. Dieser proportional zur verstrichenen Zeit zum vorhergegangenen Frame verkleinert und auf die Kamerarotation angewandt.

OBJECT_MANIPULATE Die Objektmanipulation realisiert das einhändige Packen eines virtuellen Objektes mit einer einzigen Hand. Dabei werden Aspekte der Translation und Rotation abgewandelt verwendet. Die Bewegung der packenden Hand wird sehr ähnlich zur Kamera-Translation direkt übertragen. Im Gegensatz zu CAMERA_TRANSLATE wird diese allerdings auf das Objekt angewandt. Außerdem soll jede Rotation der packenden Hand auf das Objekt übertragen werden. Dazu wird Kinect::JointOrientation genutzt, welches die Orientierung der Hand im Raum als Quaternion bereitstellt. Es wird eine Art Differenz zwischen der Orientierung eines Frames und den gepufferten Vorgängerorientierungen gebildet. Dies findet statt, indem die Vorgängerorientierungen invertiert auf die aktuelle Orientierung angewandt werden. Wie zuvor kann diese Differenz als Geschwindigkeit interpretiert werden. Bei der Anwendung auf das Objekt muss jedoch beachtet werden, dass eine Drehung der Hand beispielsweise um ihre Z-Achse nicht unweigerlich auch einer Drehung des Objektes um dessen Z-Achse haben soll. Für natürliches Verhalten muss die Rotation selbst noch abhängig von Objekt- sowie Kamerarotation gedreht werden.

FLY Dieser Zustand wurde nachträglich eingeführt, als die Notwendigkeit eines Flug-Modus deutlich wurde. Er wird mittels Vorführung der FLY-Geste betreten und analog zu den anderen Zuständen verlassen. Der Flugmodus soll eine durchgehende Bewegung realisieren, ohne dass der Benutzer sich permanent bewegt. Sie bildet primär eine Alternative zum wiederholten Anwenden von CAMERA_TRANSLATE. Während die Geste aktiv ist, wird eine Translation mit festen Werten nach vorn vorgenommen. Außerdem ist es möglich, durch die Neigung der nach vorn gestreckten Arme zu lenken. Dazu wird die mittlere Handposition mit einem Referenzpunkt auf dem Körper des Benutzers verglichen. Die Abweichung der Position von einer neutralen Ausgangsposition bestimmt die Lenkrichtung. Die Stärke der Auslenkung bestimmt dabei die Stärke der Rotation. Um schnellere Drehungen zu gewährleisten, werden starke Auslenkungen noch zusätzlich verstärkt. Analog zur CAMERA_ROTATE wird ein Quaternion berechnet, welcher die neutrale Position auf die gegebene Position abbildet. Er wird wie üblich angewandt. Außerdem wird die Neigung der Achse von linker Schulter zu rechter

Schulter bestimmt. Diese wird in eine kontinuierliche Seitwärtsrolle übersetzt.

Zur Verdeutlichung sei darauf hingewiesen, dass von jedem Zustand zu jedem anderen übergegangen werden kann, wobei dieser Übergang lediglich anhand erkannter Gesten erfolgt: Wird eine unserer Gesten erkannt (Details siehe Abschnitt 3.4), so wird der zugehörige Zustand betreten. Da unser Programm darauf ausgelegt ist, während des Event-Loops einer Hauptanwendung zu laufen, besteht die Zustandsmaschine ab ihres Starts permanent (bzw. bis zum Ende der Hauptanwendung) und besitzt keinen Finalzustand.

Genaueres zum Aussehen der State-Machine als Datenstruktur ist in Abschnitt 4.1 zu finden.

3.4 Robustheit und Pufferung

Wie wir vorangegangen festgestellt haben, sind einige der Mechanismen, die wir implementieren wollen anfällig gegenüber qualitativ niedrigwertigen Kinectdaten. Tests mit der Kinect haben folgende kritische Situationen ergeben:

- (i) **Gelenke und Skelettbestandteile in der Nähe von Objekten und anderen Personen.** Siehe zur Anschauung vor allem Abbildung 9 auf Seite 25. Diese können falsch oder verzerrt erkannt werden. So kann etwa die erkannte Handposition zwischen zwei Kinectframes Raumunterschiede von mehreren Metern aufweisen und zurückspringen. Dies kann auch Körperteile betreffen, die vor oder hinter anderen Körperteilen liegen. Für hinter Körpern oder Objekten versteckte Teile ist klar, dass diese von der Kinect nur geraten werden können. Gliedmaßen, die sich vor Körpern (seltener Gegenständen) befinden können durch die Kinect-Optik zum Teil nicht hinreichend von den weiter hinten befindlichen Körperregionen differenziert werden. Das Problem verschärft sich mit zunehmender optischer und räumlicher Ähnlichkeit (Rückstrahlverhalten im sichtbaren Licht und Infrarotlicht sowie annähernd gleiche Entfernung zur Kamera). Ferner steigt die Ungenauigkeit, je weiter man vom „perfekten Abdeckbereich“ der Kinect entfernt ist.

Ein Sonderfall fehlerhafter Skelette erwächst aus der Lichtabhängigkeit der Kinect in Verbindung mit dem von ihr verwendeten Verfahren der Tiefenfeststellung, siehe Abbildung 10 auf Seite 26. Das wiederholungsfreie Muster, das dabei in

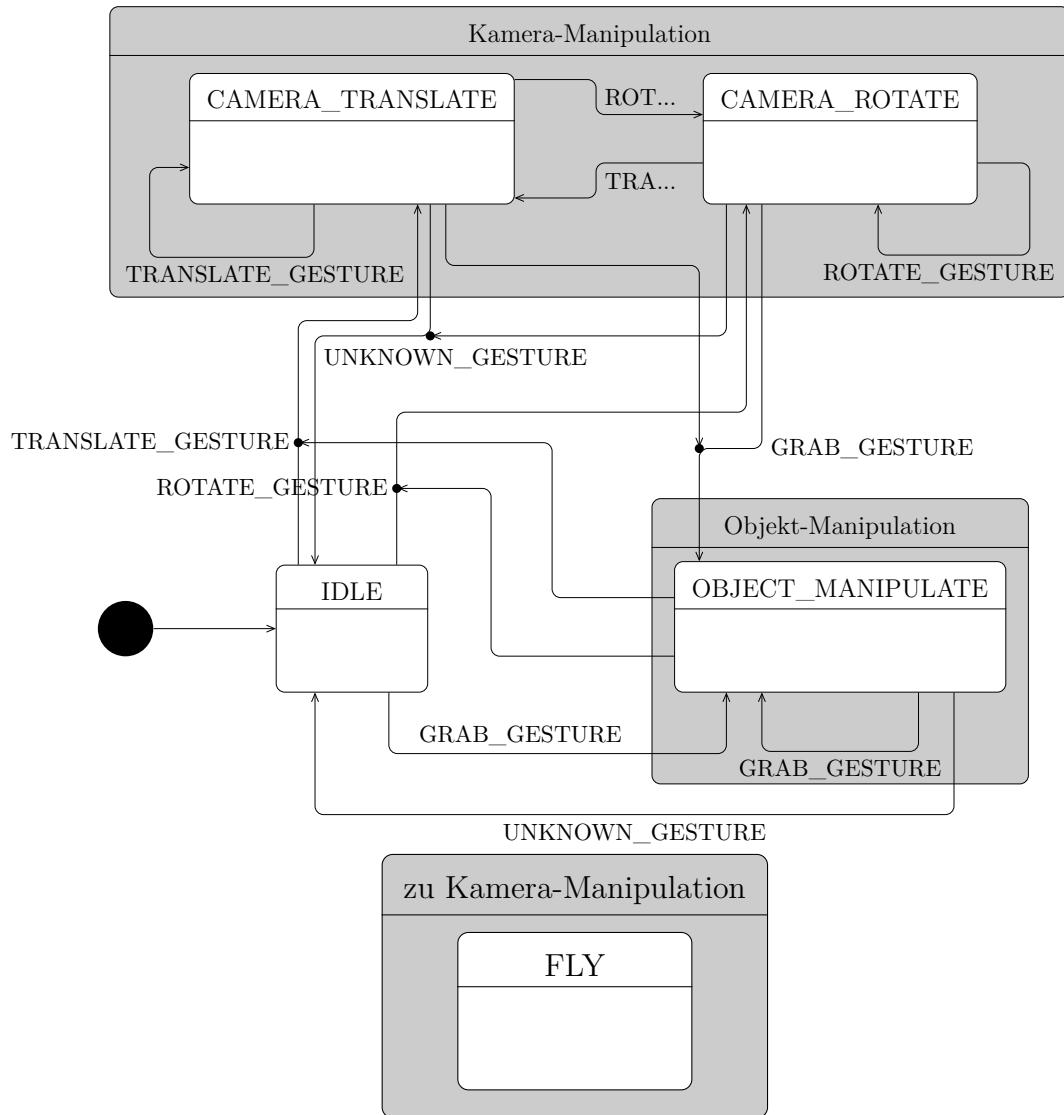


Abbildung 8: Die Zustandsmaschine. Der nachträglich eingefügte Zustand FLY ist mit allen anderen Zuständen über die entsprechende Geste verbunden und wird von allen Zuständen durch Präsentieren der FLY-Geste erreicht. Aus Gründen der Übersichtlichkeit wurde auf das Einzeichnen dieser Kanten verzichtet.

den Raum gestrahlten, kann bei Spiegelungen unvorhergesehene Nebeneffekte haben. In einer Testsituation in Verbindung mit Kleidung (hier Hosen) von hohem Infrarot-Rückstrahlvermögen spiegelte dabei der Boden das Detektionsmuster auf den Hosen zurück. Dies erzeugte einen „Infrarotschatten“ vor den Beinen des Probanden, der die gleichen erkannten Tiefenmerkmale wie der Restkörper

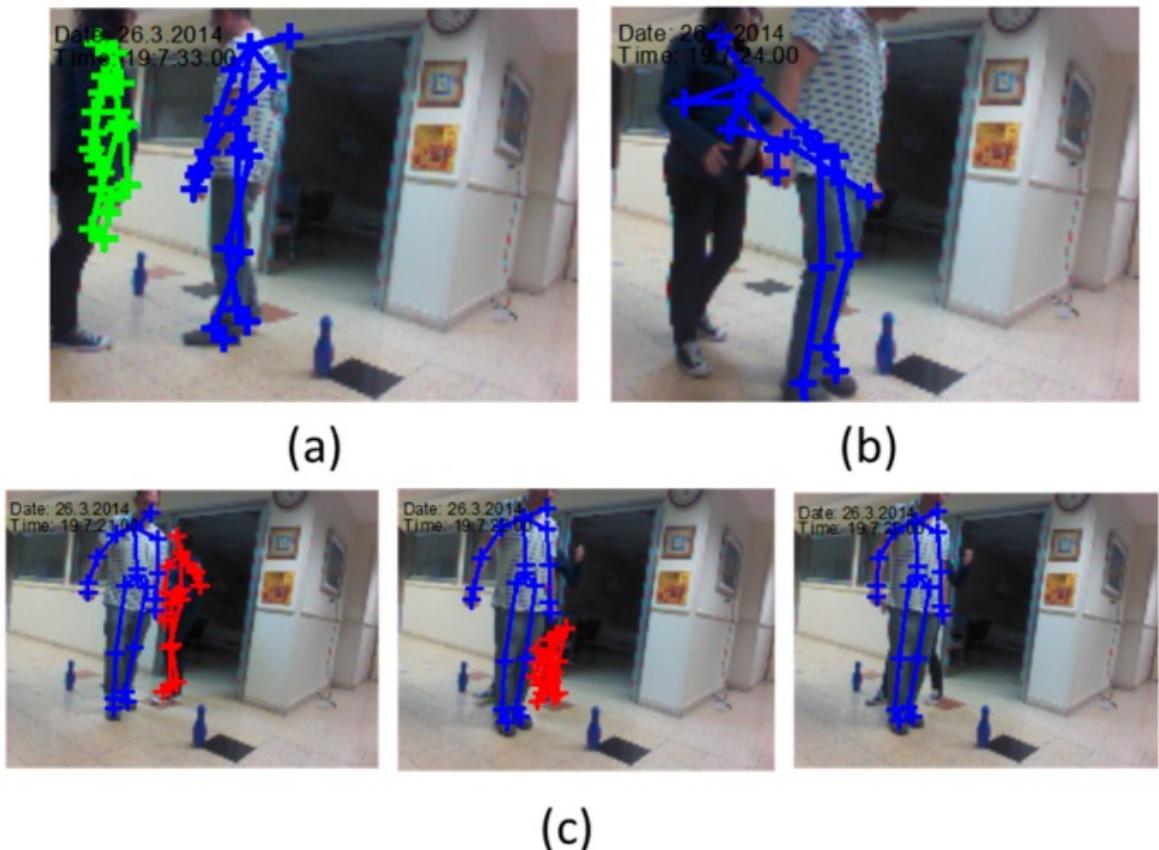


Abbildung 9: Diverse falsch erkannte Skelette.

- (a) degenerierte Skelette
- (b) verschmolzenes Skelett
- (c) Skelettverlust durch Verdeckung

Quelle: [2]

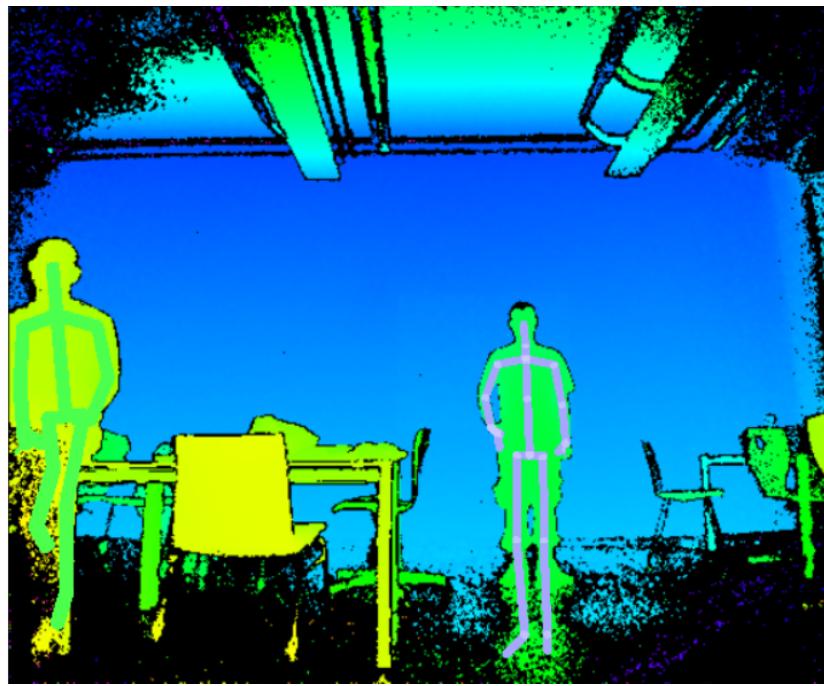


Abbildung 10: Im Sonderfall von Punkt eins beschriebene Situation eines Infrarotschatzens nebst zugewiesenen (fehlerhaften) Skelett.

hatte. Daher wurden dem Skelett des Probanden beim Test unnatürlich lange Beine zugewiesen. Hierbei sei angemerkt, dass der Boden in der Situation bei weitem kein idealer Spiegel war.

- (ii) **Status der Hände.** Auch bei durchgängiger Aufrechterhaltung eines Handzustands kann es passieren, dass die Kinect vereinzelt falsche Zuweisungen trifft oder keine Zuweisung möglich ist. Besonders schlecht wird die Erkennung, wenn sich die Hände vor dem Körper befinden. Sind die Hände selbst vollständig oder auch nur teilweise verdeckt, ist selbstverständlich ebenfalls keine sinnvolle Erkennung des Handstatus möglich. Hier liegen weitestgehend dieselben Mechanismen zugrunde, die auch die Probleme von Punkt eins verursachen.
- (iii) **Jitterfehler.** Die Kinectdaten sind verrauscht und weisen bspw. von Frame zu Frame kleine Ungenauigkeiten und Abweichungen der Gelenkpositionen in beliebige Richtungen auf. Diese Fehler nehmen ebenfalls umgekehrt proportional zur Kinect-Sicherheit zu, d. h. treten vermehrt auf, wenn die genaue Position nicht richtig erkannt wird und ein „Rateeinfluss“ vorhanden ist. Insbesondere ist dies wieder bei Verdeckung (egal in welcher Reihenfolge) mit optischer und räumli-

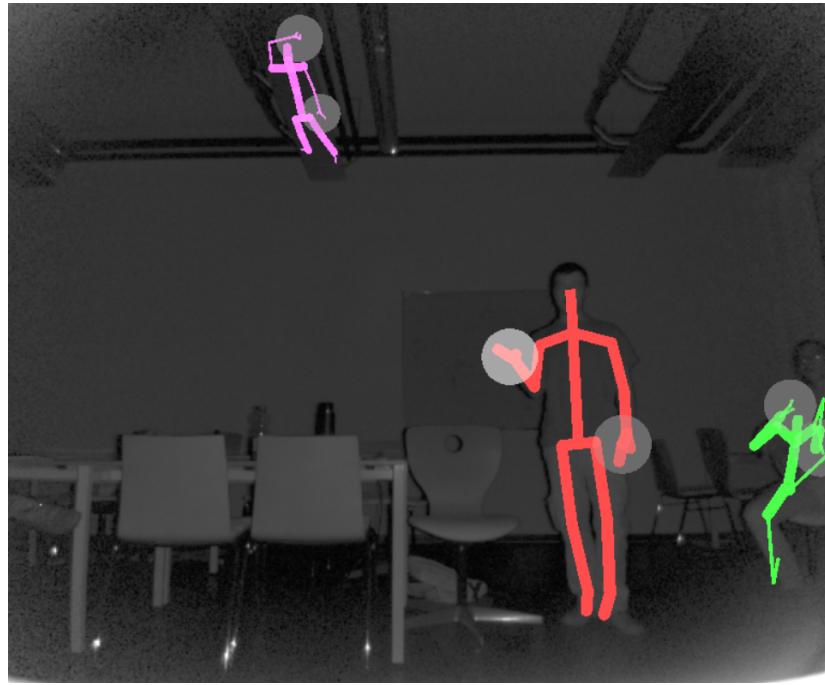


Abbildung 11: Infrarotbild der Kinect mit Phantom (oben links entlang der Lampe erkannt). Insbesondere werden Teile des Phantoms von der Kinect mit hoher Konfidenz erkannt (verdeutlicht durch die dicken Linien).

Ferner verzerrtes Skelett am rechten Rand (verursacht durch Verlassen des Aufnahmebereichs).

cher Nähe der Fall. Dieser Punkt sei jedoch von Fehlern nach Punkt eins insofern abgegrenzt, dass wir uns hier auf ständig auftretende, von ihrer Natur her kleine Fehler beziehen, obwohl Punkt eins natürlich auch bewirken kann, dass die Kinectdaten in beliebige Richtungen „zittern“.

- (iv) **Phantome.** Mitunter kann es passieren, dass ein ganzes Skelett an einem Gegenstand hängenbleibt, siehe Abbildungen 11 (Seite 27) und 12 (Seite 28). Dies geschieht, wenn eine getrackte Person einen Gegenstand passiert und dabei nicht korrekt erkannt wird – etwa weil sich die Situation sehr nahe am Rand des Aufnahmebereichs abspielt oder der Gegenstand eine nahezu humanoide Form hat. Sobald die Person wieder richtig erkannt wird, erhält sie ein neues Skelett. Die Skelettdaten dieser Phantome sind dann abgesehen von der Orientierung an menschenähnlichen Merkmalen komplett geraten und variieren über die Zeit sehr stark und willkürlich.

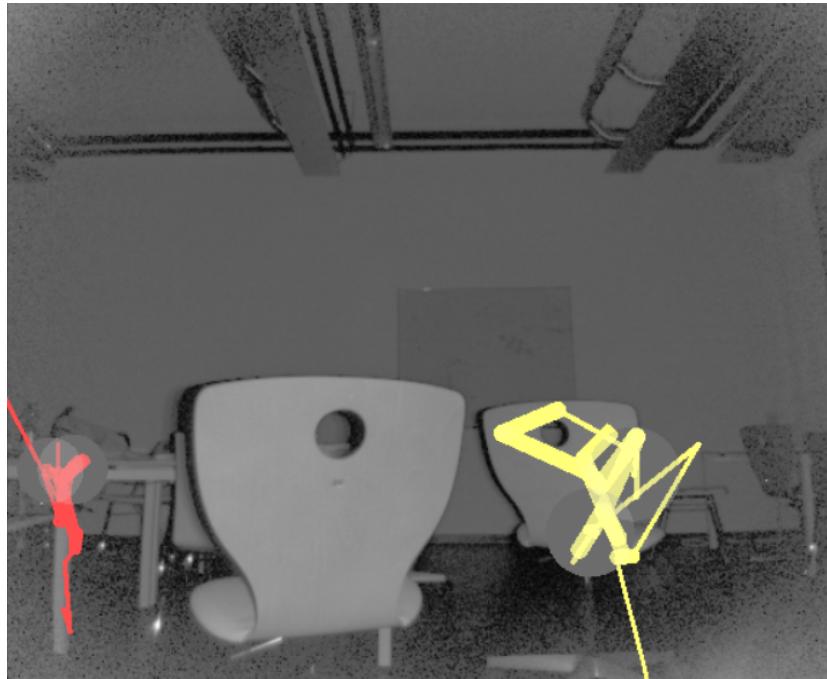


Abbildung 12: Infrarotbild der Kinect mit an Stühlen entstandenen Phantomskeletten.

- (v) **Fehlerhafte Skelettschätzungen.** Besonders problematisch für unsere Mastererkennungsmechanismen ist ein anderer Fall von Fehlerskeletten, der von der Problematik aus Punkt eins zu unterscheiden ist: In dem Augenblick, in dem eine neue Person im Aufnahmebereich erkannt wird, bekommt sie ein Skelett zugewiesen. In diese Zuweisung fließen Faktoren ein, die nichtdeterministisch scheinen. Vorstellbar ist dies als Identifizieren des Körpers mit einem geschätzten Skelett. Das Problem hierbei ist, dass eine erneute Schätzung bei zwischenzeitlichem Verlust des Skeletts anders ausfallen kann und der Körper somit leicht andere Proportionen hat. Derartiges lässt sich während der Anwendung ohne die visuellen Debug-Möglichkeiten des Kinect Studios nur schwer verhindern. Der Effekt scheint sich jedoch mildern zu lassen, wenn die betroffenen Personen sich bewegen und der Kinect damit mehr Schätzungsmöglichkeiten und die Möglichkeit zur Korrektur bieten.

Diese Punkte können gravierende Einschränkungen bezüglich der Programmbedienbarkeit mit sich ziehen. Eine fehlerhafte Erkennung von Positionen gemäß des **ersten Punktes** kann zu einem gänzlichen Verlust der gegenwärtigen Position im virtuellen Raum führen: Im naiven Ansatz wird ein hoher Differenzwert zwischen den die Be-

wegung (oder Drehung) bestimmenden Handpositionen festgestellt, der die Stärke der Manipulation besimmt und demzufolge auch eine extrem starke Manipulation bewirkt. Bei der Arbeit mit der Objektsteuerung ist ein weiteres Fehlerszenario aufgefallen, welches man als Sonderfall dieses Punktes betrachten kann: Ist die Hand des Nutzers, mit welcher dieser das Objekt bewegt oder dreht, durch die Kinect genau im Profil zu sehen, d. h. die Handfläche ist bezüglich der Höhenachse 90 Grad verdreht, so kann die Kinect nicht genau erkennen, in welche Richtung sie verdreht ist (m. a. W., ob sich der Daumen vorne oder hinten befindet). Solange dies der Fall ist, kann es passieren, dass die Kinect zwischen beiden Möglichkeiten hin- und herspringt, was sich in einer plötzlichen und äußerst starken Drehung wiederspiegelt, der jedoch keinerlei bewusste Nutzereingabe zugrunde liegt.

Eine fehlerhafte Erkennung nach **Punkt drei** verursacht durch die vielen willkürlichen kleinen Bewegungen eine als „zittrig“ wahrgenommene Steuerung der Anwendung: So nimmt ein bewegtes Objekt etwa eine Vielzahl kleiner Bewegungen bzw. Drehungen in verschiedene Richtungen vor, ohne dass der Nutzer eine entsprechende Geste präsentiert hat.

Das vorübergehende Verlieren (oder Missinterpretieren) des vorgeführten HandStates (**Punkt zwei** von oben) äußert sich bei der Programmsteuerung dagegen in einem Stottern, d. h. dass die ursprünglich fortlaufend präsentierte Geste zu den Zeitpunkten der Fehlerkennung nicht wirkt und daher z. B. eine kontinuierlich angedachte Bewegung mehrfach abrupt unterbrochen wird. Siehe Abb. 9 für eine Illustration.

Die Erzeugung eines Phantoms (**Punkt vier**) ist vor allem dann kritisch, wenn das Phantom aus dem augenblicklichen Master hervorgeht. In diesem Falle übernimmt das Phantom die Programmsteuerung, wodurch einerseits der eigentliche Master die Kontrolle verliert, aber auch andererseits das Programm gänzlich chaotische Bewegungen und Drehungen vornehmen könnte. Letzteres ist wegen der Mechanismen zur Gestenerkennung jedoch unwahrscheinlich, da das Phantomskelett die entsprechenden, relativ strengen Constraints erfüllen müsste, um den Idle-Modus zu verlassen, nach seiner Erzeugung das Programm aber sehr schnell in den Idle-Modus bringt.

Schließlich ist klar, dass Fehler nach **Punkt fünf** die Mastererkennung erschweren, da der Master gegebenenfalls mit fehlerhaften Proportionen eingespeichert wird.

Diese Probleme üben einen negativen Einfluss auf die Erfahrung aus, die der Nutzer mit der Software macht. Insbesondere Fehler nach dem erstgenannten Schema können

dem Nutzer das Erreichen seines Ziels – etwa des Annavigierens eines Objektes – unmöglich machen. Die weiteren Punkte werden dagegen einfach als störend empfunden. Die verschiedenen (und auch üblichen) von uns angewendeten Mechanismen, um diese Probleme zu beheben, sind weiter unten erklärt.

Die genannten Schwierigkeiten ergeben sich aus den üblichen Problemen von Sensoren, wobei hier hinzukommt, dass die Kinect (wegen der primären Anwendung, die die Spieleindustrie zum Ziel hatte) über vergleichsweise preiswerte Sensoren verfügt. In diversen Arbeiten, die sich mit ähnlichen Problemstellungen beschäftigen, finden die Grenzen der Kinect nahezu durchgängig Erwähnung und ein wesentlicher Punkt in der Auseinandersetzung mit der Kinect und ihrer Anwendung in unserem und ähnlichen Szenarien widmet sich einer möglichst fehlerarmen Auswertung der qualitativ durchwachsenen Daten. In der Regel wird dabei auf Verzerrungen und schlechte Werte eingegangen, die sich durch den eingeschränkten „Abdeckbereich“ der Kinect und den Einfluss von Licht ergeben (siehe [2] und [5]). Ferner wird darauf hingewiesen, dass das Detektions- und Trackingproblem generell von Beleuchtung, Blickwinkel, Distanz und weiteren Faktoren abhängt (vgl. [6]). Ferner ist ein gerade für uns wichtiger Punkt die Abhängigkeit der Kinect-Daten von der Pose, was etwa bereits in [1] festgestellt wurde. So ist es beispielsweise möglich, durch ungünstiges Verdecken von Körperpartien die durch die Kinect erkannten Gelenkpositionen zu verschieben. Im Test konnten wir so eine Verschiebung des Genicks (gemeint ist der Gelenkpunkt zwischen Schultern und Hals bzw. Kopf) um mehrere Zentimeter reproduzieren, indem die Hände vor dieser Stelle auf und ab bewegt werden. Besonders kritisch ist dies vor allem dann, wenn die Verdeckung nach dem Verschieben aufgehoben wird und der Gelenkpunkt an seine eigentliche Position „zurückschnappt“. Der ursprüngliche Ansatz, Pufferung und Mittelung, eliminiert die jitterartigen Fehler, mit denen die Kinectdaten häufig belastet sind. Hierzu wird ein Puffer vorher festgelegter Länge verwendet und während des Programmablaufs mit den für den Anwendungszweck wichtigen Daten, hier den Handpositionen des Nutzers gefüllt. Wenn unser Programm schließlich die Rückgabeparameter für die Manipulationen bestimmt, wird dieser Puffer ausgewertet. Wir bilden dabei ein exponentiell gewichtetes Mittel der gepufferten Positionen. Die neuesten Puffereinträge werden am stärksten gewichtet. Dieser Puffer dient dabei noch gleich einem anderen Zweck: Tests haben ergeben, dass das Steuern angenehmer ist, wenn die Übertragung nicht vollständig direkt von den Handpositionen erfolgt. Die Pufferlänge wurde genau so angelegt, dass das dadurch erzeugte Delay dem Nutzer nicht unange-

nehm auffällt und gleichzeitig die Kontrolle über das Programm per Gestensteuerung wesentlich glatter und angenehmer erfolgen kann.

Die eben beschriebene Glättung mag zwar kleine Jitterfehler ausmerzen, versagt jedoch bei Kinectdaten, die sehr stark von den eigentlichen Realdaten abweichen. Ein Beispiel für dieses immer wieder auftauchende Problem ist etwa ein weiterer Nutzer der sich im Hintergrund des steuernden Nutzers bewegt. In einem solchen Fall (und ähnlichen Fällen) kann es passieren, dass die Kinect Körperteile dieses zweiten Nutzers falsch interpretiert und dem Steuernden zuordnet. Dadurch können z. B. Positionsdaten entstehen, die um mehrere Meter von der Realität abweichen. Diese Fehler benötigen eine eigene Ausreißerbehandlung: Werte, die eine zu große Abweichung von den zuletzt ermittelten Werten aufweisen (etwa eine Änderung der Handposition um mehrere Meter in aufeinanderfolgenden Frames) und daher unplausibel sind, werden auf eine vordefinierte Maximalabweichung geclipt. Ohne eine solche Behandlung hätten diese Ausreißer dazu führen können, dass der Nutzer seine aktuelle Position in der 3D-Welt ohne sein Zutun mit großer Geschwindigkeit verlässt (falls er sich etwa im Kamerabewegungsmodus befand).

Mit den gerade besprochenen Methoden haben wir also eine Reihe von Robustheitsmechanismen, was fehlerhafte Kinectdaten hinsichtlich der Position von Joints (Gelenkpunkten) angeht. Dies ist nicht der einzige Aspekt der Anwendung, der solche Sonderbehandlungen verlangt. Wir hatten oben bereits als einen derartigen Punkt die Erkennung der „Hand States“ genannt. Hier ist insbesondere kritisch, dass eine Fehlererkennung nach dem ursprünglichen Modell, das nur Handzustände zu einem bestimmten Zeitpunkt diskret ausgewertet hat, zu sofortigen Zustandswechseln der Zustandsmaschine führen konnte. Besonders häufig erkennt die Kinect Handzustände in den Fehlersituationen gar nicht (und drückt dies durch „Erkennen“) des Zustands „Unknown“ aus), teils – wenn auch deutlich seltener – werden jedoch auch die „echten“ Zustände „offen“, „geschlossen“ und „Lasso“ falsch zugeordnet.

Wir wollen ferner Folgendes bemerken: Obwohl die Kinect (auch nicht intern) über keine eigenen Identifikationsmechanismen verfügt (vgl. [2]), legt die durch das im SDK enthaltene Kinect Studio (siehe [3]) nahe, dass wenigstens von der Kinect mitgelieferte Konfidenzwerte für die Güte der Rückgabedaten verfügbar sind. Diese Überlegung drängte sich auf, da schlecht erkannte Gelenke (bzw. eher Gelenkverbindungen) im Studio dünner dargestellt werden, als jene, bei denen die Kinect-Daten gut zu sein scheinen: Dünne Verbindungen treten überwiegend bei Verdeckung und am Sichtfens-

terrund auf. Es stellte sich jedoch heraus, dass alles, was die Kinect in dieser Hinsicht bereitstellt aus drei Status pro Gelenkpunkt besteht: Jeder Gelenkpunkt ist getrackt („Tracked“), nicht getrackt („NotTracked“) und vermutet („Inferred“). Über den letzten Status ist der Dokumentation (siehe [4]) nur zu entnehmen, dass das Vertrauen in die Richtigkeit der Daten „sehr gering“ ist.

Abschließend sei noch ein Szenario genannt, gegen das unsere Robustheitsmechanismen keinen hinreichenden Schutz bieten: Das der gezielten Manipulation. Wie bereits bemerkt, sind die Kinectdaten z. T. ungenau, etwa bezüglich der Gelenkpositionen. Durch (gegebenenfalls bewusstes) Verdecken oder Unkenntlichmachen von Körperteilen ist es möglich, die von der Kinect erkannten Jointkoordinaten zu verschieben. Dies kann durch den Einsatz von der Hände und der Körperhaltung, aber auch z. B. durch weite Kleidung hervorgerufen werden. Wir erklären das Problem an einem Beispiel: In unserer Mastererkennung verwenden wir neben anderen Körpermerkmalen etwa die Torsolänge. Ein Nutzer, der von der Kinect getrackt wird, kann jedoch die an ihm erkannte Torsolänge (genauer den Abstand zwischen den entsprechenden erkannten Gelenkpunkten) verändern, indem er sich beispielsweise streckt und „groß macht“ (dies verlängert die erkannte Torsolänge) oder aber sich leicht nach vorne beugt (was die erkannte Torsolänge staucht). Dies eröffnet ihm einen Spielraum bezüglich des genannten Merkmals, in dem er die vom eingespeicherten Master bekannte Torsolänge annähern kann. Ähnliches ist für andere Körperteile reproduzierbar, etwa durch leichtes Beugen der Arme oder Anheben und Hängenlassen der Schultern. Eine weitere, aber weniger relevante Möglichkeit ist auch das Ausnutzen von Kinect-Ungenauigkeiten am Rande ihres Aufnahmebereiches, z. B. weit weg von der Kamera. Ihre kleinere Relevanz liegt in der Schwierigkeit begründet, *bewusst* diverse Effekte hervorzurufen, da die Randungenauigkeiten aus Anwendersicht willkürlich und ohne Muster sind.

Für Nutzer, die sich ohnehin aufgrund ihrer Körpermerkmale recht ähnlich sind, ist es bei solchen wie eben beschriebenen Manipulation schließlich nicht mehr sicher möglich, eine korrekte Entscheidung zu fällen. Es sei jedoch noch einmal darauf hingewiesen, dass der Beeinflussungsspielraum relativ gering und damit nur für a priori ähnliche Skelette von Belang ist. Ferner ist es augenscheinlich unmöglich, die Manipulation ohne Debugausgaben der genauen Werte bewusst und gezielt durchzuführen.

4 Bemerkungen zum Quellcode

In diesem Abschnitt wollen wir wesentliche Stellen bzw. Strukturen des Programmcodes etwas technischer erläutern. Wir gehen dazu auf die verwendeten Datenstrukturen, Variablen und Funktionen ein und erläutern grob ihr Zusammenspiel. Schließlich wird in diesem Abschnitt auch auf das Einbinden unseres Programmes zur Verwendung in fremder Software eingegangen.

4.1 Wichtige Datenstrukturen, Variablen und Funktionen

Auf dem folgenden Bild ist zunächst die Dateistruktur unseres Programmes zu sehen: Dabei korrespondieren die Header- mit ihren zugehörigen .cpp-Dateien zu den wichtigsten

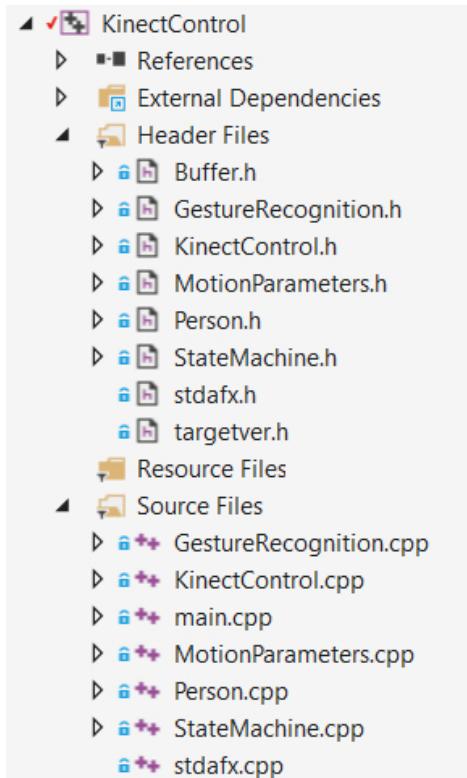


Abbildung 13: Ausschnitt aus dem Solution Explorer der Visual Studio IDE.

Klassen in unserem Programm. Diese sollen nun, sofern dies nicht in den vorangegangenen Kapiteln bereits geschehen ist, hinsichtlich ihrer Intention, ihrer Umsetzung und ihrer Verwendung erläutert werden. Die Reihenfolge, in der wir sie vorstellen orientiert sich ihren logischen Abhängigkeiten zueinander.

KinectControl. Dies ist unsere Managerklasse, die sowohl als Einstiegspunkt für Aufrufe durch andere Programme dient, als auch zur Gesamtkoordination und Arbeit mit der Kinect Verwendung findet. Wegen ihrer zentralen Funktion, sind alle anderen Header-Dateien in dieser Klasse direkt oder indirekt eingebunden. Insbesondere ist Kinect.h aus dem Kinect-SDK eingebunden, um überhaupt mit der Kinect arbeiten zu können.

KinectControl stellt eine `init()`-Funktion bereit, die die Datenstrukturen, die für die Kinect-Kommunikation gebraucht werden initialisiert und bereitet diverse Puffer durch Speicherallokation und Füllen mit Default-Einträgen vor. Dazu wird der KinectSensor geholt und „geöffnet“, d. h. seine Nutzung ermöglicht. Anschließend kann über den KinectSensor auf die BodyFrameSource des Sensors zugegriffen werden, mittels derer man dann durch Öffnen eines Readers den Stream der von der Kinect interpretierten Körperinformationen abgreifen kann. Wir sparen weitere technische Details der Funktion an dieser Stelle aus.

Weiterhin spielt die `run()`-Funktion der KinectControl-Klasse eine wichtige Rolle: Sie entspricht dem „Main Loop“ unseres Programmes. Alle Berechnungen und Aufrufe haben ihren Ursprung in ihrem Rumpf. Im Wesentlichen werden die wichtigen Bestandteile des aktuellen Zustands bzw. bisheriger Berechnungen ausgelesen und mit den von der Kinect erhaltenen neuen Daten abgeglichen oder verrechnet. Einerseits ist unsere Mastererkennung implementiert, zum Anderen findet auch das Puffern der für die Steuerung wichtigen Handpositionen hier statt und schließlich wird von hier aus auch die Zustandsmaschine gesteuert.

```

42     GetDefaultKinectSensor(&kinectSensor);
43     kinectSensor->Open();
44     kinectSensor->get_BodyFrameSource(&bodyFrameSource);
45     bodyFrameSource->get_BodyCount(&numberOfTrackedBodies);
46     bodyFrameSource->OpenReader(&bodyFrameReader);
  ↴

```

Abbildung 14: Ausschnitt aus der init-Funktion.

Buffer. An dieser Stelle sei auf die Pufferklasse verwiesen. Sie folgt dem bekannten Schema und dient unserem Projekt lediglich als Werkzeug. Auf Details verzichten wir. Es gibt folgende Funktionen:

- Einen Konstruktor, der einen Puffer fester Größe erzeugt; dazu einen Destruktor.

- Diverse Zeiger (`begin`, `end` und `next`).
- Eine Push-Funktion, sowie eine Funktion zum Leeren des Buffers.
- Abfragen auf Gefülltheit und Elemente an gegebenen Positionen.

Es sei angemekkt, dass die Push-Funktion eine Ringpufferfunktionalität implementiert, d. h. bei vollem Puffer wird wieder von vorne beginnend überschrieben.

MotionParameters. Dies ist eine Hilfsklasse, die eine für uns wichtige Sammlung an Informationen kapselt: Die Bewegungsparameter. Ein `MotionParameters`-Objekt besteht aus drei Floats, die eine Bewegung in die drei Achsenrichtungen beschreiben, einem Quaternion, der die Rotation enthält und einem „`MotionTarget`“ – einem booleschen Wert, der angibt, ob die Kamera oder ein Objekt manipuliert wird. Darüber hinaus extistieren eine Vielzahl an Gettern und Settern für einzelne oder auch mehrere Klassenvariablen, da es z. B. zweckmäßig ist, die Translationsparameter zusammen setzen zu können (vgl. Zeile 12 in Abb. 15).

```

1  #pragma once
2  #include "Eigen/Dense"
3
4  class MotionParameters {
5  public:
6      enum MotionTarget {
7          TARGET_OBJECT = false,
8          TARGET_CAMERA = true
9      };
10
11     void setMotion(float translateX, float translateY, float translateZ, Eigen::Quaternionf rotate, MotionTarget target);
12     void setTranslation(float translateX, float translateY, float translateZ);
13     void setRotation(Eigen::Quaternionf rotate);
14     void setTarget(MotionTarget target);
15     void resetMotion();
16     void resetTranslation();
17     void resetRotation();
18
19     float getTranslateX();
20     float getTranslateY();
21     float getTranslateZ();
22     Eigen::Quaternionf getRotation();
23     MotionTarget getTarget();
24
25     MotionParameters();
26 private:
27     float translateX;
28     float translateY;
29     float translateZ;
30     Eigen::Quaternionf rotate;
31     MotionTarget target; //0-verändere Model, 1-verändere Kamera
32 }
```

Abbildung 15: Inhalt der Header-Datei MotionParameters.h.

Person. Die Person-Klasse kapselt Informationen, die logisch zu einer von der Kinect erkannten Person gehören. Wir speichern selbstverständlich so gut wie ausschließlich jene Informationen, die wir im Laufe unseres Programmes auch benötigen. Zu einer Person gehören als wohl wichtigstes Element die Gelenkdaten der Kinect – ein Array von Joints. Ferner werden von diesen Gelenkdaten auch die Orientierungen, gespeichert in einem separaten Array, benötigt. Zentral sind weiter die Puffer für die Positionen der linken und rechten Hand, ein Puffer für die per Geste vorgeführten Rotationen, sowie die HandStates der beiden Hände und eine „ControlHand“ für die einhändige Objektmanipulation. Dazu verwenden wir eine ID für die Person und verfolgen ihre z-Koordinate.

```
--  
18  □ Person::Person()  
19  {  
20      id = -1;  
21  
22      leftHandCurrentPosition = { 0,0,0 };  
23      rightHandCurrentPosition = { 0,0,0 };  
24  
25      leftHandLastPosition = { 0,0,0 };  
26      rightHandLastPosition = { 0,0,0 };  
27  
28      leftHandState = HandState_Unknown;  
29      rightHandState = HandState_Unknown;  
30  
31      z = FLT_MAX;  
32  
33      // Handpositionenbuffer  
34      leftHandPositionBuffer = new Buffer<CameraSpacePoint>(POS_BUFFER_SIZE);  
35      rightHandPositionBuffer = new Buffer<CameraSpacePoint>(POS_BUFFER_SIZE);  
36  
37      // Rotationenbuffer  
38      rotationBuffer = new Buffer<Eigen::Quaternionf>(ROT_BUFFER_SIZE);  
39
```

Abbildung 16: Ausschnitt aus dem Person-Konstruktor mit Initialisierung der wesentlichen Merkmale.

Die weiteren Klassenvariablen dienen verschiedenen Zwecken, unter Anderem stellen sie die Strukturen und Funktionen bereit, die später bei der Mastererkennung dienlich sind. Zentral für die Erkennung ist hierbei die Vermessung der gezeigten Körperproportionen.

```
65  □ enum BODY_PROPERTIES {  
66      LEFT_UPPER_ARM_LENGTH, RIGHT_UPPER_ARM_LENGTH, LEFT_LOWER_ARM_LENGTH, RIGHT_LOWER_ARM_LENGTH,  
67      SHOULDER_WIDTH, HIP_WIDTH, TORSO_LENGTH, NECK_TO_HEAD, NUMBER_OF_BODY_PROPERTIES  
68  };
```

Abbildung 17: Körperproportionen, die zur Erkennung einer Person gespeichert werden

GestureRecognition. Hierbei handelt es sich erneut um eine Hilfsklasse. Enthalten sind zunächst die Grundstrukturen für die Arbeit mit unseren selbstdefinierten Gesten. Neben der Definition dieser Gesten selbst als Enumeration ist hier die Struktur „`GestureConfidence`“ gespeichert. Ein Behälter, der für unsere verschiedenen Gesten Floats enthält, die angeben werden, wie sicher eine Erkennung der zugehörigen Geste war. Die auf Abb. 18 ebenfalls zu sehende Enumeration `ControlHand` ist wieder für die einhändige Objektsteuerung nötig, als Angabe, welche Hand steuert.

```
- 4  class GestureRecognition {
5  public:
6  enum Gesture {
7      UNKNOWN,
8      TRANSLATE_GESTURE,
9      ROTATE_GESTURE,
10     GRAB_GESTURE,
11     FLY_GESTURE
12 };
13
14 struct GestureConfidence {
15     float unknownConfidence;
16     float translateCameraConfidence;
17     float rotateCameraConfidence;
18     float grabConfidence;
19     float flyConfidence;
20 };
21
22 enum ControlHand {
23     HAND_LEFT = 0,
24     HAND_RIGHT = 1
25 };
```

Abbildung 18: Ausschnitt aus dem GestureRecognition-Header.

Die Hauptfunktion dieser Klasse ist schließlich, neben dem Rahmen der Struktur `GestureConfidence`, für genau diese Konfidenzen einen Puffer nebst Auswertungsfunktion bereitzustellen, welche schließlich anhand der gepufferten Konfidenzen eine Endkonfidenz pro Geste bestimmt und so letztendlich die vermutlich vorgeführte Geste ermittelt, vergleiche Abb. 19. Dies ist die Geste mit der höchsten ermittelten Konfidenz. Die eben genannte „Ermittlung“ ist dabei einfach eine Glättung der Pufferdaten, Näheres siehe Abschnitt 3.4.

StateMachine. Die Zustandsmaschine ist das Muster, das die Gestensteuerung implementiert. Dem zugrunde liegt die natürliche Abbildung von Steuerungsmodus auf Zustand: Die Zustandsmaschine ist stets gemäß präsentierter Geste in genau einem

```

58 //Ergebniskonfidenz der Auswertung
59 GestureRecognition::GestureConfidence finalConfidence = { 0,0,0,0 };
60
61 //Gehe durch den Puffer und glätte alle Komponenten
62 for (int i = 0; i < GESTURE_BUFFER_SIZE; i++) {
63     float iSmoothFactor = gestureSmooth[i] / gestureSmoothSum;
64     currentConfidence = *confidenceBuffer->get(i);
65     finalConfidence.flyConfidence += currentConfidence.flyConfidence * iSmoothFactor;
66     finalConfidence.grabConfidence += currentConfidence.grabConfidence * iSmoothFactor;
67     finalConfidence.translateCameraConfidence += currentConfidence.translateCameraConfidence * iSmoothFactor;
68     finalConfidence.rotateCameraConfidence += currentConfidence.rotateCameraConfidence * iSmoothFactor;
69     finalConfidence.unknownConfidence += currentConfidence.unknownConfidence * iSmoothFactor;
70 }
71
72 //Werte aus, welche Komponente den höchsten Konfidenzwert hat
73 float maxConfidence = finalConfidence.unknownConfidence; Gesture maxConfidenceGesture = UNKNOWN;
74 if (maxConfidence < finalConfidence.flyConfidence) { maxConfidence = finalConfidence.flyConfidence; maxConfidenceGesture = FLY_GESTURE; }
75 if (maxConfidence < finalConfidence.grabConfidence) { maxConfidence = finalConfidence.grabConfidence; maxConfidenceGesture = GRAB_GESTURE; }
76 if (maxConfidence < finalConfidence.translateCameraConfidence) { maxConfidence = finalConfidence.translateCameraConfidence; maxConfidenceGesture = TRANSLATE_GESTURE; }
77 if (maxConfidence < finalConfidence.rotateCameraConfidence) { maxConfidence = finalConfidence.rotateCameraConfidence; maxConfidenceGesture = ROTATE_GESTURE; }
78
79 setRecognizedGesture(maxConfidenceGesture);

```

Abbildung 19: Ausschnitt aus der evaluateGestureBuffer-Funktion.

ihrer Zustände und wertet die gesehenen Daten zu Parametern aus. In der `run()`-Methode von Kinect-Control wird stets ein Berechnungsschritt, d. h. Berechnungen und Zustandswechsel, der State Machine durchgeführt.

Aus datenstruktureller Sicht ist das `State`-Struct zentral, das die bereits konzeptuell bekannten Zustände `IDLE`, `CAMERA_TRANSLATE`, `CAMERA_ROTATE`, `OBJECT_MANIPULATE` und `FLY` definiert. Wichtige Klassenvariablen sind der aktuelle Zustand „`state`“, die eingespeicherte Person „`master`“, sowie eine `GestureRecognition`-Instanz für die Gestenerkennung und die aktuellen berechneten Rückgabeparameter „`motionParameters`“. Darüber hinaus enthält sie zahlreiche Konstanten, die als Gewichte, Constraints oder empirisch ermittelte Verstärkungs- bzw. Abschwächungsfaktoren Verwendung finden. Für einen Arbeitsschritt der StateMachine sind die Funktionen `bufferGestureConfidence()`, `compute()` und `switchState()` ausschlaggebend. Diese Funktionen werden in genau dieser Reihenfolge im Main-Loop von KinectControl aufgerufen. In `bufferGestureConfidence()` wird aus den Kinectdaten unter Prüfung diverser Constraints eine Fuzzy-Geste in Form eines Konfidenztupels bestimmt und gepuffert. Dieser Puffer wird anschließend direkt ausgewertet und liefert die erkannte Geste, die in `recognizedGesture` gespeichert wird. Die `compute()`-Funktion errechnet daraufhin je nach aktuellem Zustand aus den gesehenen Skelettdaten und HandStates entsprechend vorangegangener Erklärungen die Bewegungsparameter und legt sie gebündelt in `motionParameters` ab. Schließlich entscheidet `switchState()` anhand der `recognizedGesture` über den Folgezustand und nullt gegebenenfalls die `motionParameters` der StateMachine.

Das als Erweiterung der Aufgabenstellung eingeführte Eventsystem liegt aufgrund der Klassenhierarchie ebenfalls in der StateMachine-Klasse.

```
436     switch (recognizedGesture) {
437         case GestureRecognition::Gesture::ROTATE_GESTURE:
438             // Initialisiere den Rotationsbuffer mit (0,0,0,1)-Werten
439             for (int i = 0; i < Person::ROT_BUFFER_SIZE; i++) {
440                 rotationBuffer->push(Eigen::Quaternionf::Identity());
441             }
442             motionParameters.setTarget(MotionParameters::MotionTarget::TARGET_CAMERA);
443             newState = CAMERA_ROTATE;
444             break;
445         case GestureRecognition::Gesture::TRANSLATE_GESTURE:
446             motionParameters.setTarget(MotionParameters::MotionTarget::TARGET_CAMERA);
447             newState = CAMERA_TRANSLATE;
448             break;
449         case GestureRecognition::Gesture::GRAB_GESTURE:
450             if (currentState != OBJECT_MANIPULATE)
451                 master.setLastHandOrientationInitialized(false);
452             master.setControlHand(master.getRisenHand());
453             // Initialisiere den Rotationsbuffer mit (0,0,0,1)-Werten
454             for (int i = 0; i < Person::ROT_BUFFER_SIZE; i++) {
455                 rotationBuffer->push(Eigen::Quaternionf::Identity());
456             }
457             widget->pickModel(0, 0); // löst Ray cast im widget aus
458             motionParameters.setTarget(MotionParameters::MotionTarget::TARGET_OBJECT);
459             newState = OBJECT_MANIPULATE;
460             break;
461         case GestureRecognition::Gesture::FLY_GESTURE:
462             motionParameters.setTarget(MotionParameters::MotionTarget::TARGET_CAMERA);
463             newState = FLY;
464             break;
465         default: //Übergang zu IDLE, entspricht UNKNOWN
466             motionParameters.setTarget(MotionParameters::MotionTarget::TARGET_CAMERA);
467             newState = IDLE; //Zustand nicht wechseln
468             motionParameters.resetMotion();
469             break;
470     }
471
472     setState(newState);
473     if (newState != currentState) motionParameters.resetMotion();
474 }
```

Abbildung 20: Ausschnitt aus der `switchState()`-Funktion.

Anmerkung: Die `pickModel()`-Funktion in Zeile 457 nutzt eine Stubfunktion, die in einer möglichen Erweiterung der Software zum Object Picking verwendet werden kann.

4.2 Ergänzungen zum Zusammenspiel und Ablaufskizze

In diesem Abschnitt wird der Ablauf des Programms skizziert. Wir gehen dabei davon aus, dass KinectControl bereits instanziert und durch Aufruf der `init()`-Funktion initialisiert wurde. Wir stellen uns weiter vor, dass die `run()`-Methode im Main-Loop der umgebenden Software ausgeführt wird. Genaueres zur Einbindung ist Abschnitt [4.3](#) zu entnehmen.

Sollte kein Master bestimmt sein, so werden für den Master zunächst eindeutige Defaultwerte für ID (der negative Wert -1) und z -Entfernung zur Kamera (der

maximale Wert des Floatdatentyps) angenommen.

Wir versuchen dann, über den BodyFrameReader der Kinect, einen aktuellen Kinect-Frame abzugreifen. Sollte dies fehlschlagen, so werden die alten Bewegungsparameter einfach weiterverwendet. Im Falle des Erfolgs versuchen wir auf die Körpertracking-Daten der Kinect zuzugreifen, was im Fehlerfall analog behandelt wird. Gerade die erste Art von Fehlschlag (der Versuch, einen Kinect-Frame zu holen, obwohl kein solcher vorhanden ist) tritt dabei tatsächlich häufig ein, da die Kinect nur etwa 30 Frames pro Sekunde liefern, wohingegen der Tick des Main Loops überlicherweise deutlich darüber liegt.

Wir gehen den weiteren Programmablauf nun zunächst dafür durch, dass kein Master eingespeichert wird oder wurde. In diesem Falle ist der nächste relevante Schritt das Iterieren über die Körper, die die Kinect zurückgeliefert hat. Dabei holen wir uns die genauen Körperdaten, namentlich die Positionen und Orientierungen der Gelenke. In der genannten Situation (vor jeglicher Masterfestlegung) verwenden wir den primitiven *z*-Test zur Festlegung einer Person, die das Programm steuert. Dazu wird das Master-Objekt einfach mit den Werten des Körpers belegt, dessen Kopf den niedrigsten *z*-Wert aufweist. Im Programm schließt an die Masterbehandlungsphase die Steuerung an.

Wenden wir uns zunächst aber noch den anderen Fällen dieser Masterbehandlung zu: Das Programmverhalten ändert sich, wenn durch Betätigen der Einspeichertaste X ein Master festgelegt werden soll. Eine entsprechende Abfrage im Main-Loop des unterliegenden Programmes ruft die `assignMaster()`-Funktion von KinectControl auf, die die für die Masterfestlegung wichtigen Parameter initialisiert. Vor allem werden die booleschen Variablen `masterDetermined` und `collectFrames` auf `true` gesetzt und je nach Länge des Tastendrucks eine Variable hochgezählt, die die Anzahl der Samples für die Vermessung bestimmt. Fall zwei bei der Masterbehandlung entspricht dann der Belegung der beiden Variablen `masterDetermined` und `collectFrames` mit `true`. In diesem Falle wird beim Iterieren über die Körper zunächst ein Skelett in Standardpose gesucht. Die ID dieses Skeletts wird gespeichert und die zugehörige Person soll als Master eingespeichert werden. Befinden sich mehrere Skelette in Standardpose, so wird das indexmäßig erste im `trackedBodies`-Array für die Masterfestlegung verwendet. Nachdem die genannte ID und damit der künftige Master festgelegt ist, werden solange Körpermerkmale gesammelt, wie durch den Framezähler aus `assignMaster()` vorgegeben. Dazu rufen wir die `collectBodyProperties()`-Funktion aus der `Person`-Klasse auf, die wie weiter oben ausgeführt ein festes Set an Körpermerkmalen puffert. Sollte der

designierte Master dabei die Standardpose verlassen, wird die bisherige Sammlung mittels `deleteCollectedBodyProperties()` vollständig verworfen und die ID-Festlegung durch Standardposensuche vom Anfang erneut durchgeführt. War der Frame hingegen gut, wird die Anzahl noch zu sammelnder Frames dekrementiert. Sind genau so viele Frames gesammelt, wie durch `assignMaster()` vorgegeben wurde, so wird die Masterfestlegung durch Rücksetzen von `collectFrames` auf `false` und Aufruf von `calculateBodyProperties()` (ebenfalls aus der Person-Klasse) beendet. Konzeptuell erstellt letztere Funktion aus den vorher gesammelten Daten einen charakteristischen Vektor für die eingespeicherte Person, den sie im Master-Objekt ablegt.

Der dritte Fall in Sachen Masterfestlegung schließt sich sodann an, wenn also ein Master bestimmt wurde (`masterDetermined` ist `true`), jedoch keine Samples mehr gesammelt werden müssen (`collectFrames` ist `false`), weil wir den charakteristischen Vektor bereits haben und verwenden können. Dann muss hinsichtlich des Masters nur noch etwas getan werden, falls der Master zwischenzeitlich aus dem Tracking verschwunden ist, was vor der Iteration über die Körper überprüft wird. In diesem Falle wird `searchForMaster` gesetzt. Für die 0-1-Flanke von `searchForMaster` wird ein spezielles `lostMaster`-Flag gesetzt. In der Iteration über die Körper wird dann nach Körpern in Standardpose Ausschau gehalten und für diese Körper eine Sammlung von Abweichungswerten zum charakteristischen Vektor des Masters aufgebaut. Auf Codeebene wird dies durch ein Array von Puffern, `deviationBuffer[]`, gelöst. Die Abweichungen werden durch Aufruf der `compareBodyProperties()`-Methode der Person-Klasse berechnet. Das korrekte Weitersammeln in einem angefangenen Puffer kann über die von der Kinect vergebene, eindeutige `trackingId` gesichert werden. Die Puffer haben eine feste Größe und werden, wenn sie voll – aktuell entspricht dies 20 Samples – sind, durch Aufruf von `evaluateDeviationBuffer()` gemittelt. Ergebnis ist ein gemittelter Abweichungswert vom charakteristischen Vektor des Masters. Unterschreitet dieser eine empirisch festgelegte Grenze, nehmen wir an, das die präsentierte Person der Master ist und setzen das `master`-Objekt unserer Zustandsmaschine entsprechend.

Als letzter Punkt in Sachen Masterbehandlung ist noch das Verhalten im Falle eines Masterverlusts (`lostMaster` ist dann `true`) zu klären. In diesem Falle werden sämtliche Bewegungsparameter und der Zustand der State-Machine zurückgesetzt. Außerdem wird `lostMaster` wieder auf `false` gesetzt, was den Effekt hat, dass die Variable tatsächlich genau die 0-1-Flanke von `searchForMaster` einfängt.

Betrachten wir nun die Masterauslesung zur Steuerung des Programms. Sie findet statt,

falls wir einen aktiven Master haben. Von diesem werden sodann die Gelenke mit ihren Orientierungen und die Status der Hände ausgelesen. Die entsprechenden und wichtigen Merkmale werden im `master`-Objekt (z. T. nach Plausibilitätsüberprüfungen und gepuffert) abgelegt. Dann findet die Kernberechnung der Zustandsmaschine statt (siehe Abb. 21). In `bufferGestureConfidence()` wird aus den Joints und HandStates unter Verwendung eines Puffers ein Konfidenzvektor für die Gesten erstellt. In `compute()` findet die Berechnung der `motionParameters` entsprechend des aktuellen Zustands und der Gelenkdaten statt. Die Funktion `switchState()` nimmt schließlich anhand des Gestenkonfidenzvektors gegebenenfalls einen Zustandsübergang vor. Konzeptuelle Erläuterungen zur Funktionsweise dieser Funktionen sind in den vorangegangenen Abschnitten zu finden.

```

474   ****
475  * State-Machine-Berechnungsschritt
476  ****
477  stateMachine.bufferGestureConfidence();
478  stateMachine.compute();
479  stateMachine.switchState();

```

Abbildung 21: Der Code, der den Berechnungsschritt der State-Machine darstellt.

Am Ende der `run()`-Methode wird der Frame „releas“ und die bei `compute()` berechneten `motionParameters` zurückgegeben. Wie genau sie schließlich verwendet werden, entscheidet das unterliegende Programm.

4.3 Einbinden

Das vorliegende Projekt kompiliert eine statische Bibliothek (.lib). Diese ist (in Visual Studio) als „Additional Dependency“ für das Linking einzutragen.

Das Programm, das die Library verwendet, sollte mittels

```
KinectControl kinectControl;
```

ein `KinectControl`-Objekt anlegen und in seiner Initialisierung durch Aufruf seiner `init()`-Funktion mitinitialisieren.

Im Main-Loop oder Event-Loop des Programms werden per

```
MotionParameters motionParameters = kinectControl.run();
```

die Parameter (Translation, Rotation und Ziel) geholt und können ausgewertet und entsprechen angewendet werden.

Weiterhin sollte `kinectControl.assignMaster()` irgendwie angestoßen werden können, z. B. über ein bestimmtes Tastendruck-Event.

5 Schlussbemerkungen

Literatur

- [1] R. M. Araujo, G. Graña und V. Andersson. “Towards skeleton biometric identification using the microsoft kinect sensor”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. Zugriff 12.6.17. ACM. 2013, S. 21–26. URL: https://www.researchgate.net/profile/Ricardo_Araujo2/publication/237064051_Towards_Skeleton_Biometric_Identification_Using_the_Microsoft_Kinect_Sensor/links/0046351b1d1cc4c5a0000000.pdf.
- [2] G. Blumrosen u.a. “A Real-Time Kinect Signature-Based Patient Home-Monitoring System”. In: *Sensors (Basel)* (2016). Zugriff 12.6.17. URL: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5134624/>.
- [3] Microsoft. *Kinect für Windows 2.0 SDK*. <https://developer.microsoft.com/de-de/windows/kinect/tools>. Zugriff 12.6.17.
- [4] Microsoft Developer Network. *TrackingState Enumeration*. <https://msdn.microsoft.com/en-us/library/microsoft.kinect.kinect.trackingstate.aspx>. Zugriff 12.6.17. 2017.
- [5] R. Seggers. “People Tracking in Outdoor Environments: Evaluating the Kinect 2 Performance in Differend Lighting Conditions”. Zugriff 12.6.17. Bachelorarbeit. University of Amsterdam, 2015. URL: <https://staff.fnwi.uva.nl/a.visser/education/bachelorAI/thesisSeggers.pdf>.
- [6] L. Susperregi u.a. “On the Use of a Low-Cost Thermal Sensor to Improve Kinect People Detection in a Mobile Robot”. In: *Sensors (Basel)* (2013). Zugriff 12.6.17. URL: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3871095/>.
- [7] Jason Walters. *Kinect v2 Joint Map*. <http://glitchbeam.com/2015/04/02/kinect-v2-joint-map>. Original nicht mehr verfügbar, gecacht abgerufen von <http://web.archive.org/web/20151031033003/http://glitchbeam.com:80/2015/04/02/kinect-v2-joint-map/>, Zugriff 4.7.2017.