Projektseminar: Gestensteuerung einer 3D-Anwendung mittels Kinect

Mario Janke
Peter Lindner
Patrick Stäblein

Inhaltsverzeichnis

1 Rahmenbedingungen

1.1 Grundlagen & Technik

Gegeben ist eine bereits vorhandene 3D-Anwendung, die zu Demonstrationszwecken genutzt wird. Innerhalb der Anwendung ist es möglich,

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

zusätzlich geplant ist später

- geladene Objekte manipulieren, in diesem Falle skalieren oder löschen zu können.

Das Programm rendert dabei zwei Ausgabefenster, in denen die Szene dargestellt ist, wobei die Kameras 3D-Aufbau bilden.
Die so beschriebene Ausgabe wird über zwei Projektoren von hinten auf eine Projektionsfläche geworfen – ein Projektor für die linke Kamera und einer für die rechte. Wird die „Leinwand“ von vorne durch eine Shutterbrille betrachtet, entsteht der 3D-Eindruck.
Die Steuerung der Anwendung erfolgt über Tastatur und Maus bzw. Präsentationspointer.

1.2 Aufgabenstellung

Ziel des Projektseminars ist es, die Steuerung der Anwendung hinsichtlich einer Präsentation vor einer Zuschauergruppe zu erleichtern und intuitiv zu gestalten, sodass parallel an der Universität vorhandene (und bislang ungenutzte) Technik verwendet und präsentiert werden kann. In diesem Sinne geeignet und vorgeschlagen sind

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

Das damit entwickelte Programm soll Folgendes leisten:

- Es soll in der Lage zu sein, sämtliche Steuerung und Manipulation, die oben beschrieben wurde durchzuführen.

- Die Bedienung soll sehr intuitiv und einfach sein, d. h. etwaige Gesten müssen bezüglich der ihnen zugeordneten Aktion einleuchtend und leicht auszuführen sein.
- Das Programm soll möglichst einfach eingebunden und wiederverwendet werden können.

1.3 Eigenschaften der Kinect

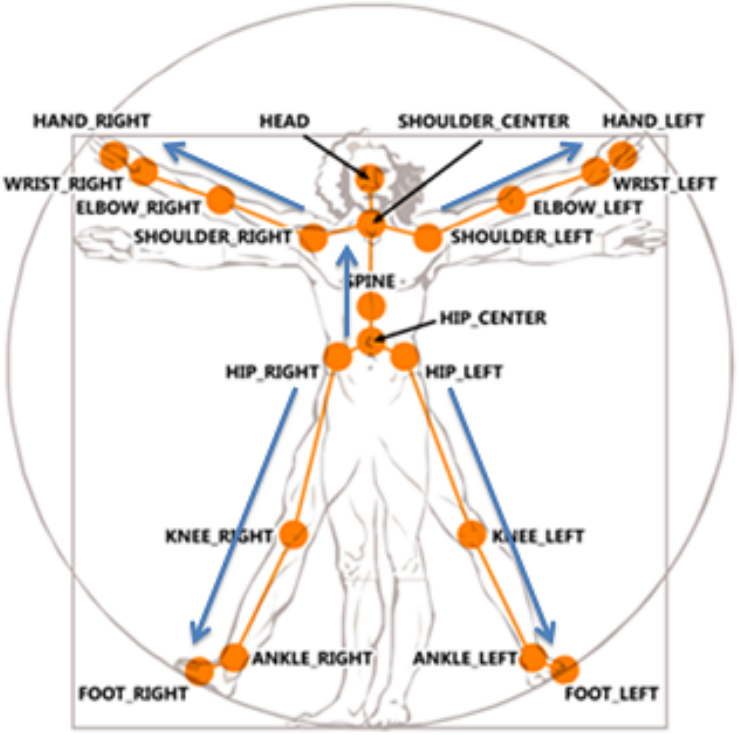

Abbildung 1: Gelenkpunkte die mit der Kinect getrackt werden können
Quelle: [4]

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 getrackten 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 getrackter 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:

1. die technische Umsetzung, d. h.
    - das korrekte Erkennen und Werten von Gesten einer ausgezeichneten getrackten Person
    - das korrekte Berechnen notwendiger Bewegungsparameter
    - die Einbindung in die bestehende Applikation
2. die Interaktion mit dem Benutzer, d. h.
    - das Entwerfen intuitiver und eingängiger Gesten für die verschiedenen Zwecke
    - das Auszeichnen einer getrackten Person als „Master“, der das Programm steuert

Wir stellen in diesem Abschnitt die zentralen 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 Objektmanipulation wird 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 reduziert die Gesamtzahl nötiger Gesten.

Die Kinect ermöglicht ein Tracking des gesamten Körpers für mehrere (genauer sechs) Personen. Wir beschränken uns aus naheliegenden Gründen jedoch 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 unnötig.
- Die in unserem Anwendungsfall intuitiven Gesten werden ausschließlich mit den Händen (bzw. Armen) durchgeführt.

Primitive Erkennungsmöglichkeiten eines Masters kann man etwa aus der Entfernung der getrackten Personen zur Kamera und der Position der Personen im Raum gewinnen. 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.

2.1 Priorisierung der Aufgaben

2.2 Aufgabenverteilung im Team

Grob: Mario – Algorithmik und Berechnung; Peter – Backend, Struktur; Patrick – Master, Kinect-Basics; Anmerkung, dass wir zu viert angefangen haben.

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.
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 nähesten 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 praktibalste 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 Körper proportionen 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

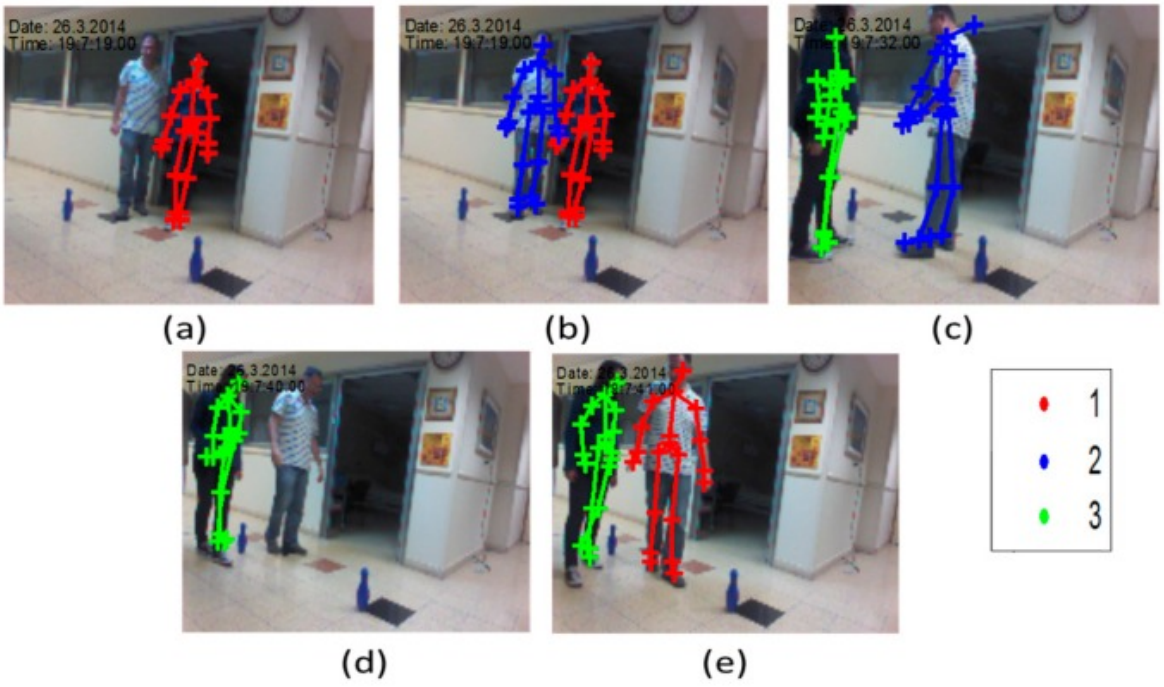

Abbildung 2: Neuzuordnung von IDs nach zwischenzeitlichem „Verlust“ von Skeletten, Quelle:[2]

ermittelt. Weitere Körperproportionen die verwendet wurden sind unter anderem die Schulterbreite, Hüftbreite, Unterarmlänge, Abstand zwischen Hals und Kopf.

3.2 Gesten und ihre Wirkung

Für die eingangs erwähnte Aufgabenstellung war es notwendig, bestimmte Programmfunktionalitäten mit Gesten zu verbinden. Einerseits hätte die Möglichkeit bestanden, 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. Stattdessen 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 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.

ROTATE_GESTURE 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. Wie die TRANSLATE-Geste war auch diese Geste von Anfang an unumstritten und alternativlos.

GRAB_GESTURE 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 aus 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 (wie auch naheliegend) viel besser erkennen kann, wie die Handfläche gekippt bzw. gedreht ist. Intern behielten wir jedoch den semantischen Namen „GRAB“-Geste bei.

FLY_GESTURE 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. Durch Schwenken der Arme soll auch die Kamera während der Fahrt schwenken.

UNKNOWN Dies enthält alles, was als keine der anderen Gesten erkannt wird.

Tests mit der Kinect haben ergeben, dass es notwendig ist, bei derartig selbst implementierten Gesten auch eigene Robustheitsmechanismen einzubauen, die die Gestenerkennung 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. ?? zu sehen. Im Folgenden erklären wir die Zustände, ihre Semantik und die enthaltenen Berechnungen etwas näher:

IDLE Dieser Zustand entspricht dem Initialzustand unserer Zustandsmachine. 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. Wir berechnen dazu aus den gepufferten Positionswerten 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. Aus diesen Größen berechnen wir 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.

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.

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 Situtationen ergeben:

- Gelenke und Skelettbestandteile in der Nähe von Objekten und anderen Personen. Diese können falsch oder verzerrt erkannt werden. So kann etwa die erkannte

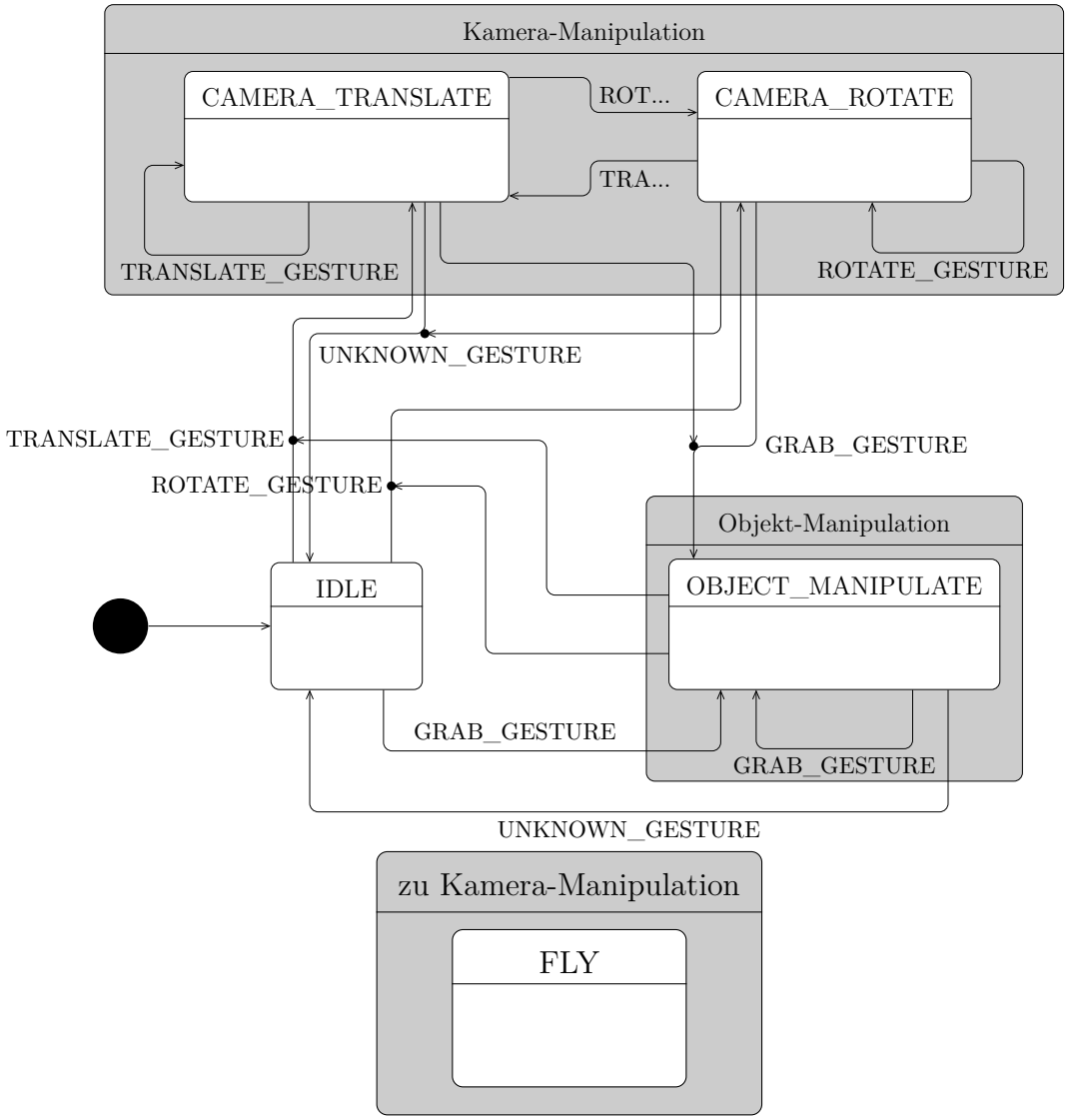

Abbildung 3: Die Zustandsmaschine. Der nachträglich eingefügte Zustand FLY ist mit allen anderen Zuständen über die entsprechende Geste verbunden und wird von allen Zuständen durch Präsentieren der FLY-Geste erreicht. Aus Gründen der Übersichtlichkeit wurde auf das Einzeichnen dieser Kanten verzichtet.

Handposition zwischen zwei Kinectframes Raumunterschiede von mehreren Metern aufweisen und zurückspringen.

- 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.

- Jitterfehler. Die Kinectdaten sind verrauscht und weisen bspw. von Frame zu Frame kleine Ungenauigkeiten und Abweichungen der Gelenkpositionen in beliebige Richtungen auf.

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 die Bewegung (oder Drehung) bestimmtenden Handpositionen festgestellt, der die Stärke der Manipulation besimmt und demzufolge auch eine extrem starke Manipulation bewirkt. Ferner bewirkt eine fehlerhafte Erkennung nach Punkt drei 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. 4 für eine Illustration.
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 Zieles – 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 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

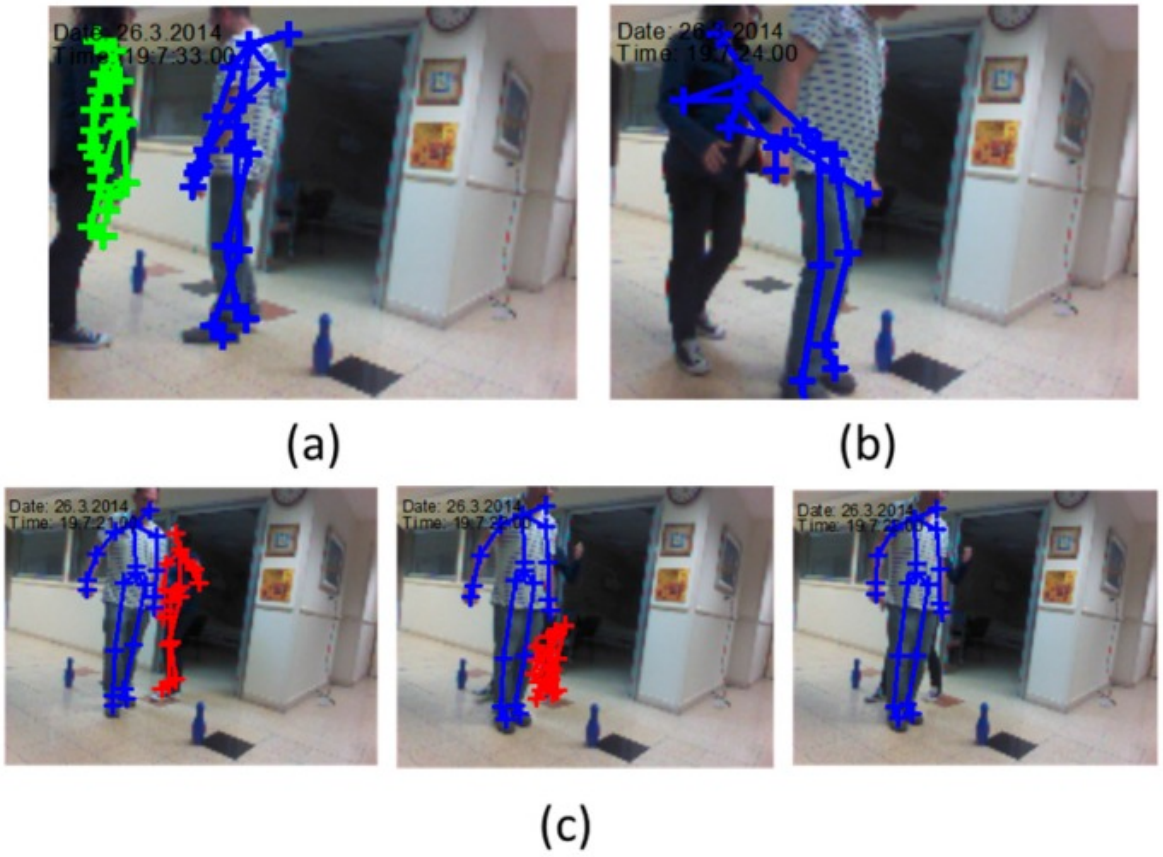

Abbildung 4: Diverse falsch erkannte Skelette.

(a) degenerierte Skelette

(b) verschmolzenes Skelett

(c) Skelettverlust durch verdeckung

Quelle: [2]

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 [6]). Ferner wird darauf hingewiesen, dass das Detektions- und Trackingproblem generell von Beleuchtung, Blickwinkel, Distanz und weiteren Faktoren abhängt (vgl. [7]). 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 unangenehm 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 Realtität abweichen. Diese Fehler benötigen eine eigene Ausreißerbehandlung: Werte, die eine zu große Abweichung von den zu-

letzt ermittelten Werten aufweisen (etwa eine Änderung der Handposition um mehrere Meter in aufeinanderfolgenden Frames) und daher unplausibel sind, werden auf eine vordefinierte Maximalabweichung geclippt. 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 Fehlerkennung 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 Sichtfensterrand 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 [5]) nur zu entnehmen, dass das Vertrauen in die Richtigkeit der Daten „sehr gering“ ist.

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 zugehören .cpp-Dateien zu den wichtigsten

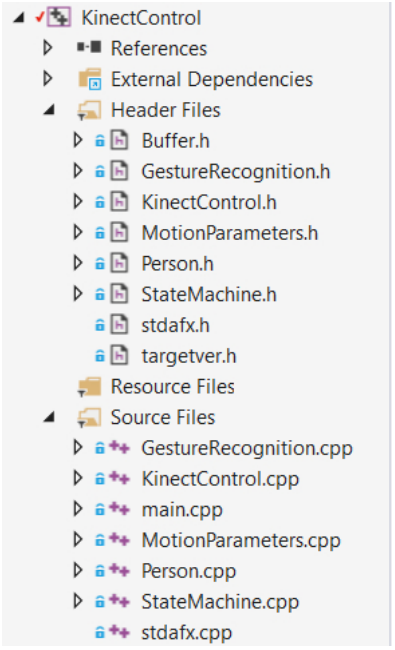

Abbildung 5: Ausschnitt aus dem Solution Explorer der Visual Studio IDE.

Klassen in unserem Programm. Diese sollen nun, sofern dies nicht in den vorangegangen 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 6: 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 angemekt, 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. 7).

```
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 7: 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. Die weiteren Klassenvariablen dienen verschiedenen Zwecken, unter Anderem stellen sie die Strukturen und Funktionen bereit, die später bei der Mastererkennung dienlich sind.

```
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 8: Ausschnitt aus dem Person-Konstruktor mit Initialisierung der wesentlichen Merkmale.

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. 9 ebenfalls zu sehende Enumeration ControlHand ist wieder für die einhändige Objektsteuerung nötig, als Angabe, welche Hand steuert.
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. 10. Dies ist die Geste mit der höchsten ermittelten Konfidenz. Die eben

```
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 9: Ausschnitt aus dem GestureRecognition-Header.

```
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 10: Ausschnitt aus der evaluateGestureBuffer-Funktion.

genannte „Ermittlung“ ist dabei einfach eine Glättung der Pufferdaten, Näheres siehe Abschnitt 3.4.

StateMachine.

4.2 Details zum Zusammenspiel

4.3 Einbinden

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. *Tracking Users with Kinect Skeletal Tracking*. https://msdn.microsoft.com/en-us/library/jj131025.aspx. Zugriff 25.4.2017. 2017.

[5] Microsoft Developer Network. *TrackingState Enumeration*. https://msdn.microsoft.com/en-us/library/microsoft.kinect.kinect.trackingstate.aspx. Zugriff 12.6.17. 2017.

[6] 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.

[7] 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/.