Hochschule
Flensburg
University of
Applied Sciences

Bachelorthesis

Thema

Kamerabasiertes Interaktionskonzept
mit virtuellen Welten

von

Philipp Borucki

Matrikel-Nr.:	**690114**
Studiengang:	**Medieninformatik PSO 2017**
Betreuer/in und Erstbewerter/in:	**Knut Hartmann**
Zweitbewerter/in:	**Simon Olberding**
Ausgabedatum:	**20.04.2023**
Abgabedatum:	**20.06.2023**

Ich versichere, dass ich die vorliegende Thesis ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen benutzt habe.

Abstract

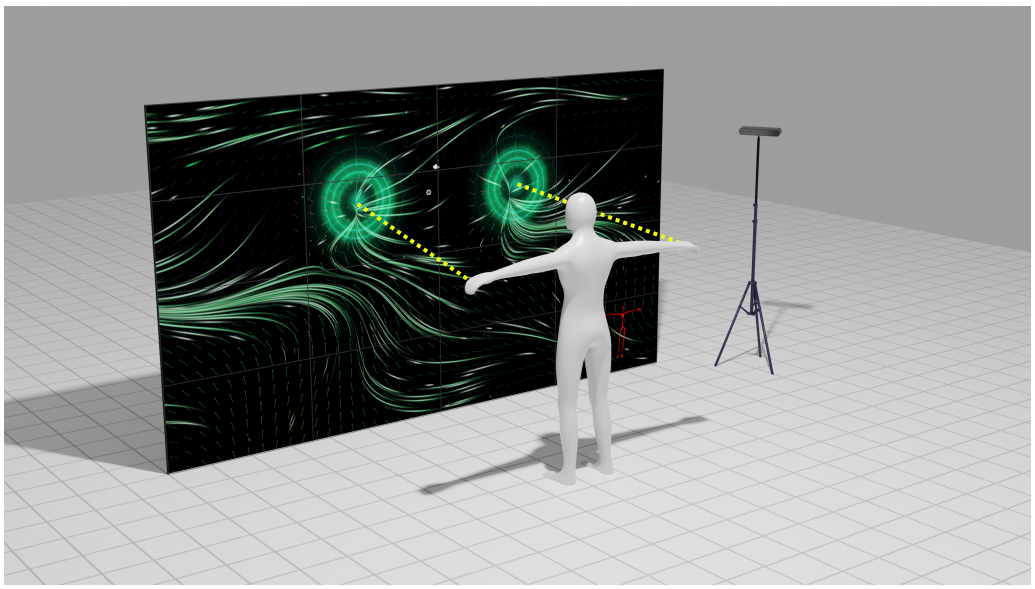
Für die Interaktion mit digitalen Ausstellungsstücken auf Messen existieren verschiedene Ansätze; neben konventionellen Eingabemethoden wie Maus und Tastatur sind dies zunehmend auch Augmented Reality (AR) und Virtual Reality (VR). Dabei beschränkt sich die Möglichkeit, die Ausstellungsstücke gleichzeitig zu nutzen, meist auf die Anzahl verfügbarer Hardware. In der vorliegenden Arbeit wird eine technische Grundlage entwickelt, auf deren Basis 2D-Anwendungen erstellt werden können, in denen mehrere Personen über ein Natural User Interface (NUI) mit dieser auf einem Bildschirm interagieren können.

Dafür wird das kamerabasierte Erkennungssystem ZED 2i von »StereoLabs« in Unity eingebunden. Eine eigene Unity-Komponente transformiert die Rohdaten der Kamera in eine gespiegelte Abbildung der Personen vor dem Bildschirm auf diesem. Dabei werden verschiedene Herangehensweisen betrachtet und bewertet. Eine exemplarische Anwendung in Form eines mit den Händen manipulierbaren Vektorfelds wird entwickelt.

Die entwickelte Unity-Komponente stellt die Basis für NUI-gestützte 2D-Anwendungen und damit Ausstellungsstücke dar, welches auch anhand eines Beispiels demonstriert wird. Gleichzeitig bildet sie den Ausgangspunkt für weitere Forschung in verschiedenen Bereichen wie Skalierbarkeit, Benutzerfreundlichkeit, erweiterter Interaktions- und Konfigurationsmöglichkeiten sowie der Verlässlichkeit des Erkennungssystems und eröffnet weitere Möglichkeiten für kreative Weiterentwicklungen und Anwendungen.

Inhaltsverzeichnis

1. Einleitung



Es gibt verschiedene Ansätze für die Interaktion mit virtuellen Systemen, wie beispielsweise digitalen interaktiven Ausstellungsstücken auf Messen. Allgemein bekannte Ansätze sind unter anderem die Verwendung von »konventionellen« Eingabemethoden wie Tastatur und Maus, der Einsatz von Touchscreens für interaktive Displays oder die Verwendung von mobilen Apps bzw. dedizierten Audioguides als Ergänzung zu bestehenden Objekten.

Um ein interaktives und immersives Erlebnis für Personen zu schaffen, werden auch zunehmend Technologien der Virtual Reality (VR) und Augmented Reality (AR) eingesetzt.[1] Die Personen können entweder mit einer komplett digitalen Umgebung interagieren und dort Objekte aus verschiedenen Blickwinkeln betrachten, sie manipulieren bzw. Informationen darüber erhalten, oder sehen eine Erweiterung der realen Umgebung mit zusätzlichen Informationen oder interaktiven Elementen.

[1] vgl. Cohen u. a., *Augmented and Virtual Reality in Operations*, S. 6.

Alle diese Verfahren bieten Möglichkeiten, aber auch Einschränkungen, wie zum Beispiel feste Limitierungen bei der Anzahl der gleichzeitig Nutzenden. Bei der Verwendung von VR-Headsets ist die Anzahl der Nutzenden beispielsweise auf die Anzahl der verfügbaren Geräte beschränkt.

Der grundlegende Gedanke dieser Arbeit besteht darin, einen immersiven Ansatz zu entwickeln, der es ermöglicht, dass verschiedene Personen ohne individuelle Vorbereitung direkt mit einem virtuellen System interagieren können. Dabei soll unter anderem auf dedizierte Eingabegeräte pro Person verzichtet werden, um die theoretische Anzahl der interagierenden Personen nicht durch die Anzahl der Eingabegeräte fest zu limitieren. Dies soll auch verhindern, dass umstehende Personen von der immersiven Interaktion ausgeschlossen werden.

Möglich wäre das durch die Verwendung eines Natural User Interface (NUI) als Interaktionsmethode, bei dem der menschliche Körper als Eingabegerät fungiert. Die Interaktion erfolgt damit durch die Körperbewegung der Personen.

Ein geeignetes Verfahren für die konkrete Problemstellung ist der Einsatz eines rein optischen kamerabasierten Erkennungssystems, welches ohne weitere Markierungen oder zusätzliche individuelle Sensorik an den betreffenden Personen auskommt. In dieser Arbeit wird, auf Basis eines solchen Systems, eine prototypische Implementierung entwickelt, mit der beispielsweise eine Person ohne Vorbereitung mit einer zweidimensionalen Anwendung oder einem Spiel auf einem gegenüberliegenden Display interagieren kann.

Für eine solche Implementierung wird auf die Laufzeit- und Entwicklungsumgebung Unity zurückgegriffen. Als Erkennungssystem wird die stereoskopisches Tiefenkamera *ZED 2i* von StereoLabs eingesetzt. Für diese Kombination wird eine Komponente in Unity entwickelt, welche die Körperbewegungsdaten der einzelnen Personen vom Erkennungssystem zur Verwendung aufbereitet und für andere Komponenten nutzbar bereitstellt. Für die Interaktion mit einer Anwendung auf einem Display gegenüber der interagierenden Person bedeutet dies, dass die Daten für einen zweidimensionalen Anwendungsfall vorbereitet werden müssen. Daher soll die Interaktion der Personen möglichst unabhängig von der genauen Ausrichtung des Erkennungssystems sein und relativ zum Bildschirm stattfinden. Der Implementierung der Komponente folgt eine exemplarische Demonstration auf Basis von Vektorfeldern. In dieser findet die Komponente Anwendung, indem eine interagierende Person das Vektorfeld manipulieren kann. Abschließend wird ein Fazit gezogen und ein Ausblick auf mögliche Weiterentwicklungen des Themas gegeben.

Im Rahmen dieser Arbeit wird eine mögliche prototypische Implementierung für eine kamerabasierte Interaktion von Personen mit einem zweidimensionalen Display durchgeführt und beispielhaft ausgearbeitet. Dabei liegt der Fokus auf der Schaffung einer technologischen Grundlage für eine solche Interaktion bzw. der Implementierung des Konzepts, ohne Wertungen über die Interaktion selbst vorzunehmen.

2. Technische Grundlagen

Zur Umsetzung der Implementierung werden verschiedene technische Grundlagen benötigt, welche im Folgenden für die Zwecke dieser Arbeit vorgestellt werden. Dabei handelt es sich unter anderem um eine Einführung in das verwendete Erkennungssystem bzw. des von dessen Hersteller bereitgestellten Software Development Kit (SDK). Da die spätere Implementierung innerhalb der Laufzeit- und Entwicklungsumgebung Unity durchgeführt wird, sind Vorkenntnisse im Umgang mit Unity erforderlich.

2.1 Stereoskopisches Tiefenkamerasystem

Als Erkennungssystem für diese Arbeit wird die stereoskopische Tiefenkamera *ZED 2i* der Firma *StereoLabs* verwendet. Bei einer solchen Kamera handelt es sich um ein System, das in der Lage ist, Informationen über die räumliche Tiefe einer Szene zu erfassen. Sie besteht aus einem Paar von Kameralinsen, welche in einem definierten Abstand zueinander angeordnet sind. Damit kann eine sogenannte passive Stereoanalyse durchgeführt werden, um aus den zwei versetzten zweidimensionalen Abbildungen nachträglich die Tiefeninformation zu triangulieren.[2]

2.1.1 Hauptkomponenten des Tiefenkamerasystem

Das ZED-System besteht aus der Kamera-Hardware und einer Softwarekomponente (dem SDK). Die Hardware stellt dafür verschiedene Sensoren in einem Gehäuse zur Verfügung. Dazu zählen die dualen Bildsensoren bzw. Kameralinsen, aber auch verschiedene Bewegungs- und Umgebungssensoren (Temperatursensor, Beschleunigungsmesser, Gyroskop, Magnetometer sowie Barometer).[3] Das SDK führt die eigentliche Interpretation der erfassten Daten auf einem angeschlossenen Hostgerät (Computer) durch, wurde in der Programmiersprache C++ entwickelt und steht mit Stand dieser Arbeit für Windows und Ubuntu zur Verfügung. Es stellt verschiedene Module bereit, auf dessen Basis Anwendungen entwickelt werden können.

[2] vgl. Sasse, »Entfernungsmessende Verfahren«, S. 34.
[3] s. Stereolabs Inc., *ZED 2i - Datasheet (Jan. 2023)*, S. 4.

Diese umfassen:

- KI-basierte Tiefenerkennung (Depth Sensing) aus stereoskopischen Aufnahmen,
- Positional Tracking (Positionsverfolgung) der Hardware,
- Spatial Mapping (Raumkartierung),
- Object Detection (Objekterkennung),
- sowie Body Tracking.[4]

Alle notwendigen Berechnungen, auch die Triangulation der Tiefeninformationen, finden dabei zur Laufzeit auf dem Hostgerät statt. Dafür wird auf die NVIDIA-Programmierschnittstelle Compute Unified Device Architecture (CUDA) zurückgegriffen, welche insbesondere hochgradig parallelisierbare Berechnungen auf NVIDIA-Grafikprozessoren ermöglicht.

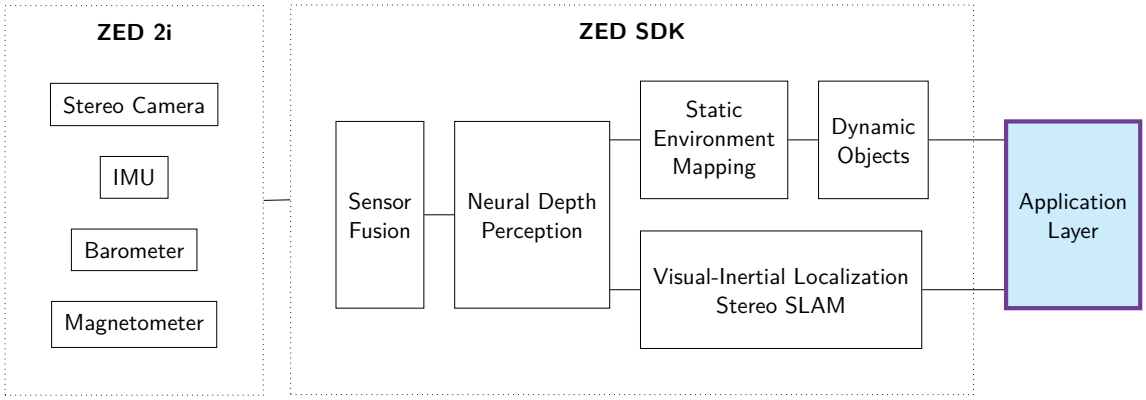


Abbildung 2.1: Funktionales SDK-Diagramm

2.1.2 »Body Tracking Module«

Zur Implementierung dieser Arbeit wird als Basis das »Body Tracking Module« des SDKs in der Version 4.0.2 verwendet.[5] Dieses KI-basierte Modul stellt zur Laufzeit auf Basis der erkannten Tiefeninformationen kontinuierlich Daten über erkannte Personen und zu diesen zugeordnete Knochenpositionen bereit. Dafür nutzt es eines von drei wählbaren neuralen Netzwerken (`FAST`, `MEDIUM`, `ACCURATE`),[6] welche sich in Performance und Genauigkeit unterscheiden. Ein erkannter Knochen wird nach der Erkennung durch zwei sogenannte Schlüsselpunkte repräsentiert. Dabei handelt es sich nach der Verarbeitung durch das Modul um

[4] s. Stereolabs Inc., *ZED 2i - Datasheet (Jan. 2023)*, S. 6.
[5] s. Stereolabs Inc., *Body Tracking Module – ZED SDK API Reference (C++)*.
[6] s. Stereolabs Inc., *Using the Body Tracking API*.

3D-Positionen von Schlüsselpunkten der erkannten Personen im Raum relativ zur Tiefenkamera.

Dafür bietet das Modul je nach Konfiguration verschiedene Körperformate bzw. Schlüsselpunktmodelle, welche unterschiedliche Detailgrade bzw. Komplexitäten aufweisen. Diese Schlüsselpunkte werden in der fest definierten Reihenfolge des jeweiligen Formats übergeben, sodass man bei jeder Aktualisierung die momentanen Positionen der einzelnen Schlüsselpunkte erhält und verarbeiten kann.

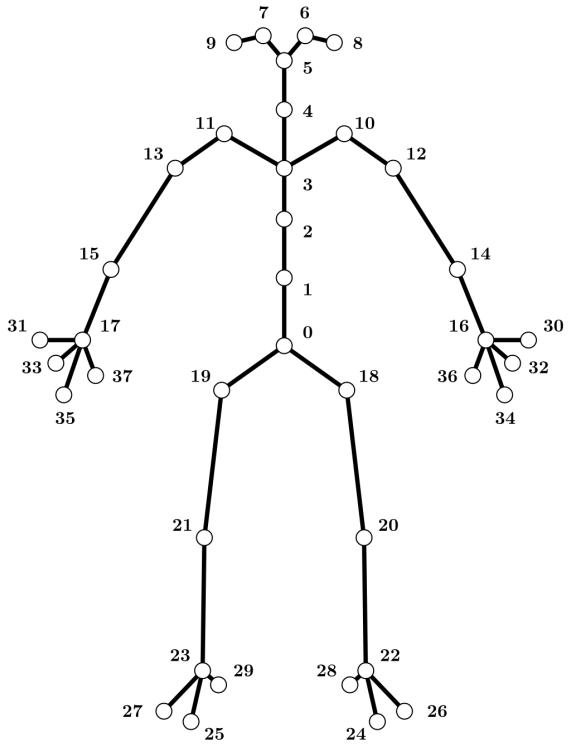


Abbildung 2.2: Körperformat mit 38 Schlüsselpunkten (mit definierter Reihenfolge)

Die erkannten Körperdaten werden als Datenstruktur verfügbar gemacht und enthalten neben der Schlüsselpunktpositionen auch weitere Informationen, welche kontinuierlich aktualisiert werden. Ein wichtiges Merkmal dabei ist die eindeutige Zuweisung von Bewegungen zu einer bestimmten Person über den Erkennungszeitraum.[7] So können aktualisierte Bewegungsdaten derselben Person bzw. demselben Körper zugeordnet werden.

[7] s. Stereolabs Inc., *Body Tracking Overview*, Abschnitt Detection Outputs.

2.2 Wrapper zur Anbindung der Tiefenkamera an Unity

Der Hersteller stellt für verschiedene Entwicklungsumgebungen sogenannte Wrapper für das SDK bereit.[8] Diese stellen eine Art Brücke zwischen dem in C++ erstellten SDK und der Zielumgebung dar, indem die externen Aufrufe und Typen auf die andere Umgebung adaptiert werden. Im Fall von Unity bedeutet das, dass ein Unity-Paket bereitgestellt wird, welches Komponenten, Skripte und Beispiele für diesen Zweck beinhaltet. Nachfolgend wird auf die Kernbestandteile des SDK-Wrappers eingegangen, welche für die Realisierung notwendig sind.[9]

2.2.1 Kamera-Prefab und ZEDManager.cs

Zur Nutzung des SDKs mit Unity existieren Prefab-Objekte (`ZED_Rig_Mono` & `ZED_Rig_Stereo`), welche zur Hierarchie der Unity-Szene hinzugefügt werden können. Es handelt sich dabei um die Repräsentation der physischen Kamera in Unity. Diese sind eigentlich als benutzerdefinierte AR-Kameras vorgesehen, welche die reguläre Unity-Kamera in einer Szene ersetzen. Diese sind dafür konzipiert, um vorbereitete virtuelle, von der Unity-Kamera gerenderte Elemente mit den durch die ZED-Kamera aufgenommenen Videodaten zu vermischen.

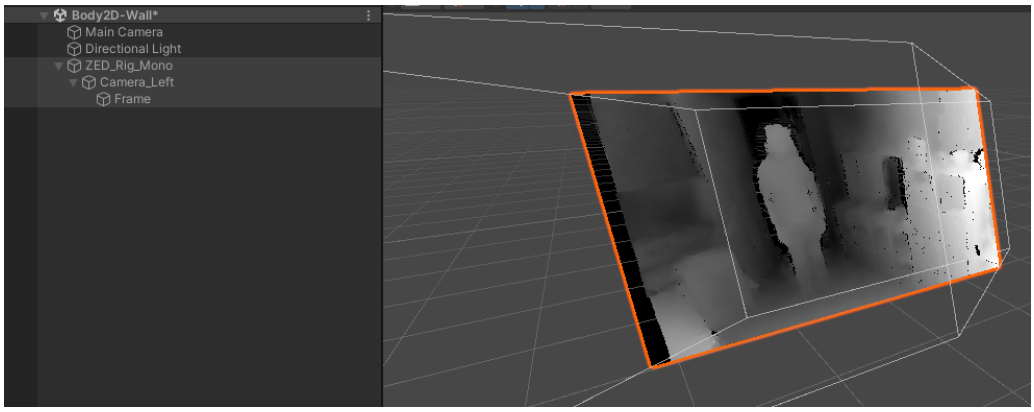


Abbildung 2.3: `ZED_Rig_Mono`-Prefab in der Szenenhierarchie

Für das Ziel dieser Arbeit ist diese AR-Funktionalität nicht erforderlich. Allerdings bildet das Kamera-Prefab-Objekt mit seinen Komponenten auch die Basis und Schnittstelle für andere Komponenten in Unity, um die ZED-Kamera bzw. das SDK zu verwenden. Daher wird das Kamera-Prefab-Objekt als Grundlage verwendet, ohne die reguläre Unity-Kamera zu ersetzen.

Das Prefab enthält dafür eine wichtige Komponente, den ZED Manager (ZEDManager.cs). Dieser Manager dient als eigentliche primäre Schnittstelle zwischen

[8] s. Stereolabs Inc., *Unity and ZED.*
[9] s. Stereolabs Inc., *Unity: Key Concepts & Scripts.*

dem ZED-System und Unity. Er steuert das Gesamtsystem (Hardware & SDK) über Unity und ermöglicht den Verbindungsaufbau, die Konfiguration von Kameraparametern, das Starten und Konfigurieren von SDK-Modulen und stellt zahlreiche Statusinformationen, Eigenschaften, Methoden und Eventhandler für die Nutzung in eigenen Implementierungen bereit. Die Konfiguration bzw. Nutzung des Managers erfolgt dabei entweder über den Unity Inspektor (Inspektor) oder mit eigenen Komponenten über eine Objekt-Referenz des ZED Managers.

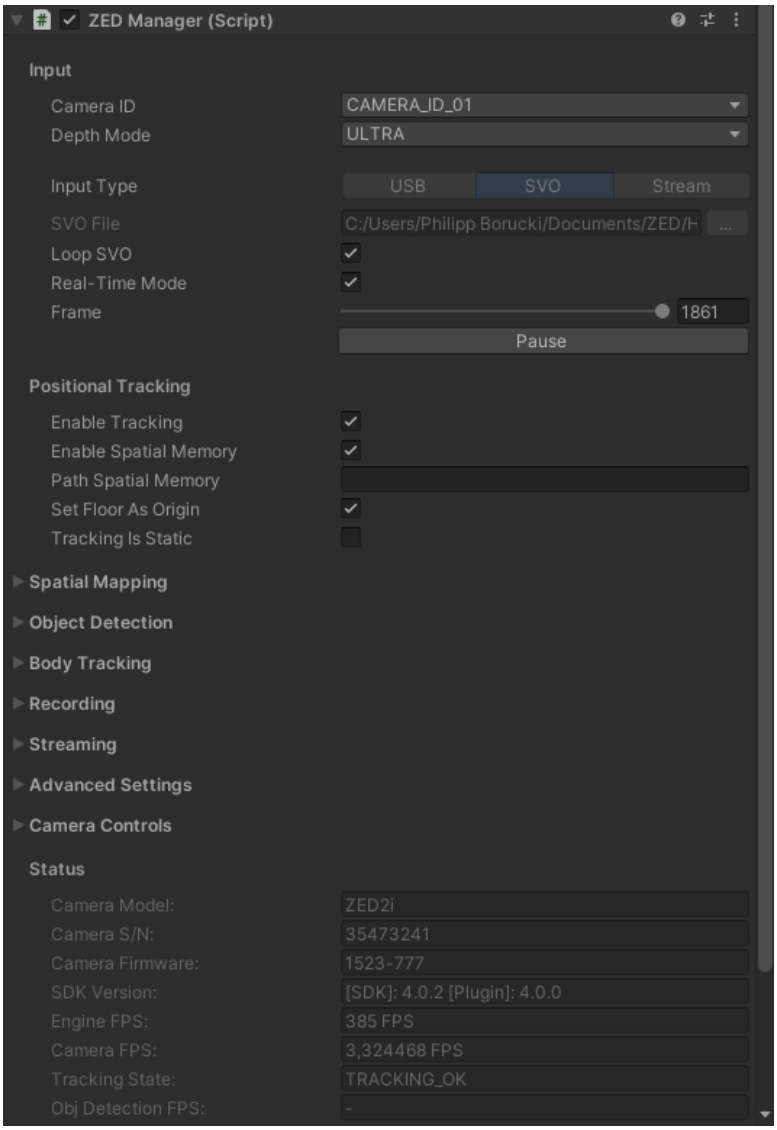


Abbildung 2.4: Konfiguration: ZED Manager (Inspektor)

Die Grundeinstellungen umfassen dabei unter anderem die Quelle der Sensordaten (Input Type) und welche Qualitätsstufe (Depth Mode) für die Tiefenberechnung sowie Auflösung (Resolution) genutzt werden soll. Des Weiteren ist es möglich, auch Aufzeichnungen anstelle einer Hardware-Kamera zu verwenden. Diese sogenannten SVO-Dateien (proprietäres Dateiformat) enthalten nicht nur die Bildinformationen aus beiden Bildsensoren, sondern auch alle anderen gesammelten Sensordaten der Hardware. Das SDK kann diese Daten identisch wie Realdaten von der Kamera-Hardware interpretieren und es ist möglich, eine vorhandene Aufnahme eines Szenarios mit nachträglich geänderten Interpretationseinstellungen erneut zu verwenden.

Das Starten oder die Konfiguration des Body Tracking Moduls kann ebenfalls im Inspektor oder über Code-Anweisungen vorgenommen werden. Die spezifische Konfiguration umfasst unter anderem die Wahl des neuralen Netzes (Body Tracking Model), des Körperformats bzw. Schlüsselpunktmodells (Body Format) und weitere Parameter.

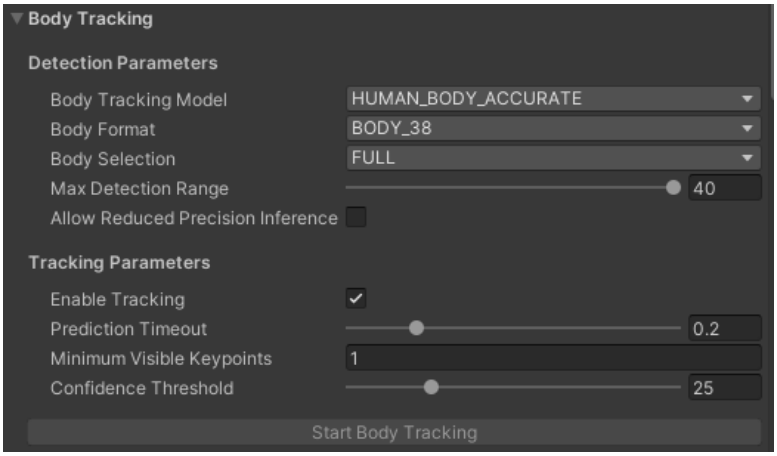


Abbildung 2.5: Konfiguration: Body Tracking im ZED Manager (Inspektor)

Zusammenfassend ermöglicht der ZED Manager die Kontrolle und Konfiguration des ZED-Systems in Unity, auch in Abhängigkeit vom Bedarf der Implementierung und der Leistungsfähigkeit des Hostgerätes. Dabei sind insbesondere die Möglichkeiten der Anpassung bzw. Abwägung zwischen Genauigkeit und Performance hervorzuheben. Der ZED Manager stellt auch verschiedene Informationen über den Zustand des SDK bzw. der Kamera zur Verfügung und bietet eventbasierte Aktualisierungen bei Zustandsänderungen an. Für das weitere Vorgehen sind dabei besonders folgende Events relevant:

- `public event OnZEDManagerReady OnZEDReady`
  signalisiert die Bereitschaft der Kamera-Hardware und des SDK.

- `public event onNewBodyTrackingTriggerDelegate OnBodyTracking;` signalisiert wenn beim SDK ein aktualisierte Frame mit »Body Tracking«-Daten vorliegt.

2.3 Zusammenfassung der verwendeten Technologien

Für die Umsetzung dieser Arbeit werden verschiedene Technologien eingesetzt, welche noch einmal zusammengefasst werden:

- Laufzeit- und Entwicklungsumgebung Unity 2021.3.23f1 LTS
- ZED-System
  - StereoLabs ZED 2i
  - StereoLabs ZED SDK 4.0.2
  - StereoLabs ZED Plugin for Unity 4.0.0
- NVIDIA CUDA 11.8

Die Kombination aus Unity, dem ZED SDK mit ZED 2i und dem »ZED Plugin for Unity« bildet somit die technologische Grundlage für die Implementierung und Durchführung dieser Arbeit.

3. 3D-2D-Transformation

Die Grundziele und die technische Grundlage der Arbeit wurden im vorherigen Kapitel ausgeführt. Im Folgenden wird die Ausgangslage und die theoretische Grundlage für die Grundimplementierung beschrieben, welche sich damit beschäftigt, wie die vom Erkennungssystem bereitgestellten Daten für die vorliegende Problemstellung nutzbar gemacht werden können. Dabei werden zwei verschiedene Ansätze in Betracht gezogen und erläutert. Anschließend wird die Implementierung des ausgewählten Verfahrens als Unity-Komponente beschrieben. Dabei werden die erforderlichen Funktionalitäten zur grundlegenden Nutzung der Komponente identifiziert und implementiert. Dieses Kapitel legt damit den Grundstein für die praktische Umsetzung des Interaktionsprototyps.

3.1 Herangehensweise

3.1.1 Beschreibung der Ausgangslage

Das ZED-System erzeugt kontinuierlich Daten von erfassten Personen und leitet diese an die Anwendung weiter. Diese Daten beinhalten unter anderem die Transformationen der Person bzw. ihrer Schlüsselpunkte als dreidimensionale Vektoren, relativ zu einem vom ZED-System definierten Ursprung. Die Kamera-Hardware muss so positioniert sein, dass sie die zu erfassenden Personen einsehen kann.

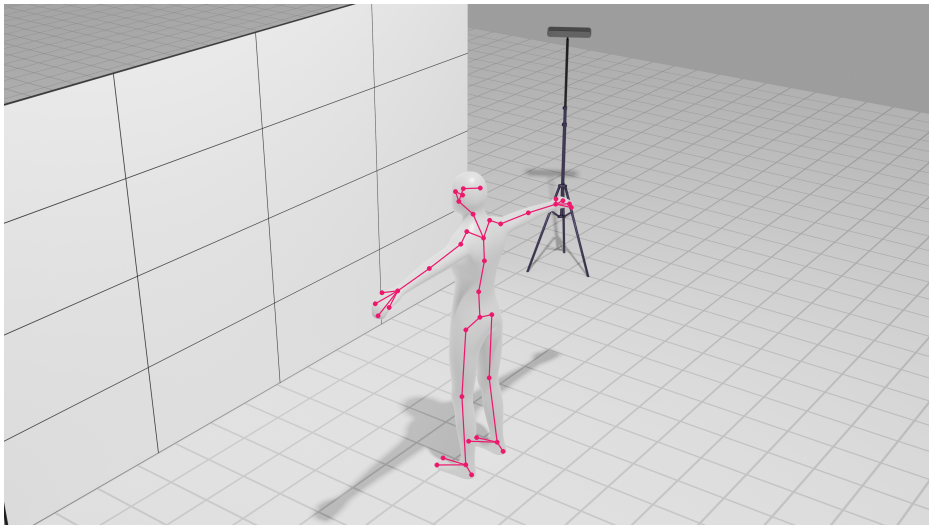


Abbildung 3.1: ZED-System ermittelt Schlüsselpunkte relativ zur Tiefenkamera

Eine oder mehrere erkannte Personen befinden sich vor einem großformatigen Bildschirm, welcher als Interaktionsziel dient und durch ein NUI gesteuert werden soll. Die Schlüsselpunkte der Realwelt (3D-Welt) sollen auf den Bildschirm (2D-Fläche) projiziert werden. Dazu muss eine frontale Perspektive auf die interagierenden Personen definiert werden, die die Positionierung dieser vor dem Bildschirm wiedergibt und auf deren Basis die 3D-2D-Projektion der Schlüsselpunkte durchgeführt werden kann. Die daraus resultierenden neuberechneten Schlüsselpunkte sollen dann von der Anwendung zur Interaktion verwendet werden.

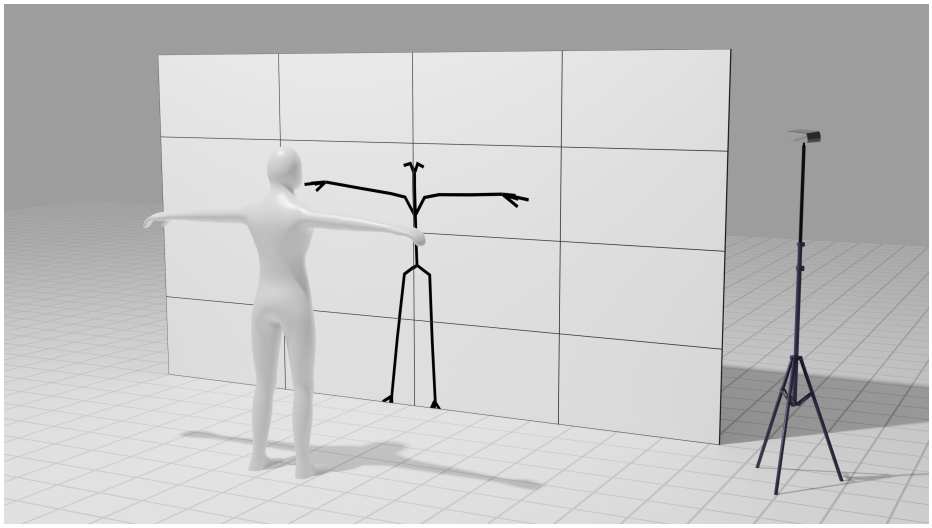


Abbildung 3.2: Spiegelartige Interaktion mit Bildschirmfläche

3.1.2 1. Lösungsansatz (Orthogonal-Kamera-Projektion)

Die Vektoren der Schlüsselpunkte der einzelnen Personen sind relativ zu einem Ursprung definiert, der durch das ZED-System festgelegt ist.

Eine virtuelle orthogonale Kamera wird in eine passende Position und Orientierung gebracht, indem diese Kamera entlang der Bildschirmflächennormale frontal auf die mit dem Bildschirm interagierenden Personen gerichtet ist. Der Sichtkörper der orthogonalen Kamera muss dabei die Personen einschließen.

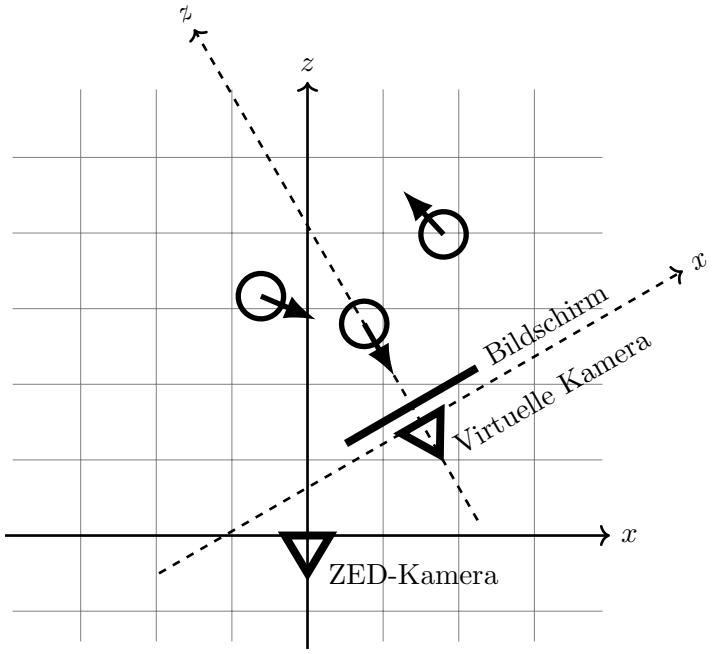


Abbildung 3.3: Weltansicht der ZED-Kamera

Die Schlüsselpunkte der einzelnen Personen im Raum (Weltkoordinaten) müssen in Bezug zur durch die virtuelle Kamera definierten Sicht gebracht werden. Das bedeutet, dass diese Schlüsselpunkte durch Transformation mit der Kameramatrix in das Koordinatensystem der virtuellen Kamera überführt werden.

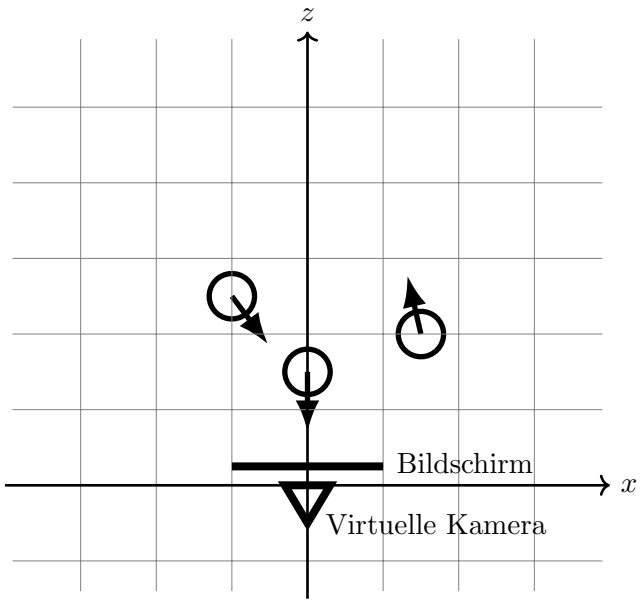


Abbildung 3.4: Weltansicht der virtuellen Kamera bzw. des Bildschirms

Anschließend müssen nun die Tiefeninformationen (Z-Achse in Kamerakoordinatensystemen) aus den überführten Schlüsselpunkten entfernt werden. Dies kann in diesem Fall mit einer zusätzlichen Transformation durch eine Orthogonalprojektionsmatrix auf die XY-Ebene, daher einer Skalierung der Tiefenachse mit 0, durchgeführt werden.

Formal ließe sich das so beschreiben:

$V :=$ Menge der Schlüsselpunkt-Vektoren

$$M_{OP} := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ (Orthogonalprojektionsmatrix auf die XY-Ebene)

$M_{WtC} :=$ Kameramatrix der virtuellen Kamera

Die Menge der überführten Vektoren V' wird durch die Anwendung der folgenden Berechnungsvorschrift auf die Menge V gebildet:

$$V' = \{M_{OP} \cdot M_{WtC} \cdot \overrightarrow{v} \mid \overrightarrow{v} \in V\}$$

Das Ergebnis der Operation sind zweidimensionale Koordinaten der Personenschlüsselpunkte auf einer XY-Ebene aus Sicht der virtuellen Kamera und damit des Monitors, welche gleichzeitig direkt als Weltkoordinaten zur Interaktion mit der Anwendung verwendet werden können.

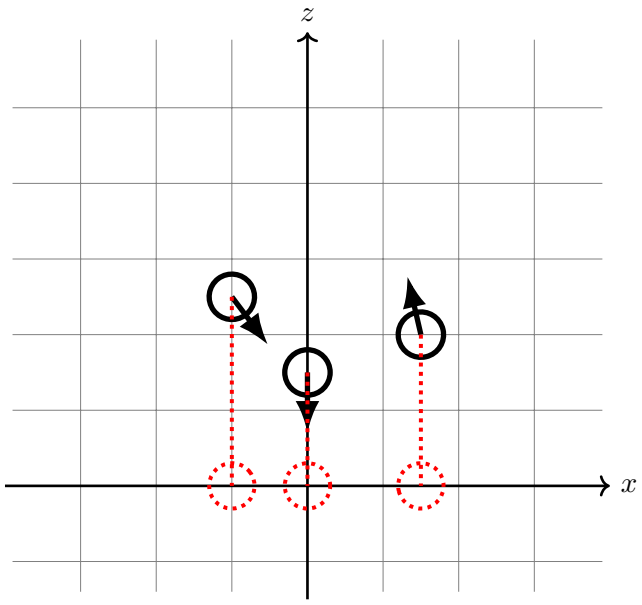


Abbildung 3.5: Nutzung von Kamerakoordinaten als Unity-Weltkoordinaten

Falls die virtuelle Kamera auch als Kamera für die Darstellung der Anwendung verwendet werden soll, müssen die errechneten Punkte stattdessen mithilfe der inversen Kameramatrix zurück in Weltkoordinaten umgewandelt werden. Das Ergebnis hierbei wären Weltkoordinaten der Schlüsselpunkte mit Ausrichtung auf die virtuelle Kamera.

3.1.3 2. Lösungsansatz (erkennungsdaten-basierte Rigid-Body-Transformation)

Der zweite betrachtete Lösungsansatz ist eine Transformation aller Schlüsselpunkte als starrer Körper bzw. Rigid-Body anhand einer auf Erkennungsdaten basierenden Transformation.

Bei einem Starrkörper handelt es sich vereinfacht um ein System aus diskreten Punkten. Dessen Transformationen können als Überlagerung einer Translation, bei der auf allen Punkten die gleiche Verschiebung angewendet wird, und einer

Rotation, bei sich alle Punkte als Einheit auf einer Kreisbahn um die Drehachse verschieben, dargestellt werden.[10]

Das bedeutet hier, dass alle Schlüsselpunkte durch die zum Weltkoordinatensystem relative Position und Orientierung einer Referenzperson transformiert werden. Diese Referenzperson dient daher als Kalibrierung für die Transformation aller weiteren Schlüsselpunkte. Die Referenzperson muss dafür während der Kalibrierung möglichst gerade und mittig vorm Interaktionsziel (dem Bildschirm) stehen und stellt den neuen Transformationsursprung aller weiteren Schlüsselpunkte dar.

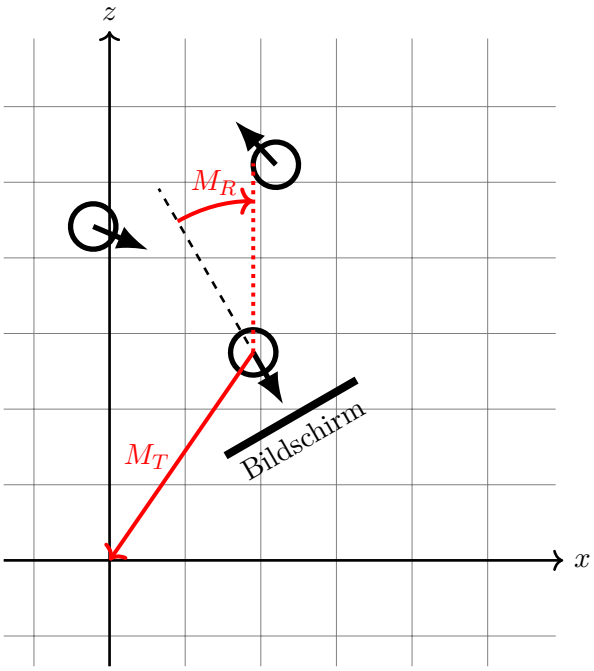


Abbildung 3.6: Erkannte Personen vor Kalibrierung

Insgesamt sollen alle Schlüsselpunkte gleichmäßig so transformiert werden, dass die Referenzperson im Ursprung des Weltkoordinatensystems steht und bei gerader Ausrichtung zum Bildschirm ebenfalls gerade entlang der Z-Achse orientiert ist. Gleichzeitig bleiben aber alle relativen Abstände zwischen den Schlüsselpunkten erhalten. Dieser Vorgang entspricht daher also einer Starrkörpertransformation bzw. Rigid-Body-Transformation, wobei die Summe der Schlüsselpunkte aller Körper den insgesamt zu transformierenden Starrkörper definiert.

Als Grundlage wird eine gemittelte relative Position der Referenzperson sowie die Orientierung ihres dedizierten Körpers (bzw. Starrkörpers) benötigt. Die daraus resultierende Rotationsmatrix wird mit der Translationsmatrix verrechnet und ergibt damit eine »Referenzperson-Transformationsmatrix«.

[10] vgl. Bruhns und Lehmann, *Elemente der Mechanik I: Einführung, Statik*, S. 81 ff.

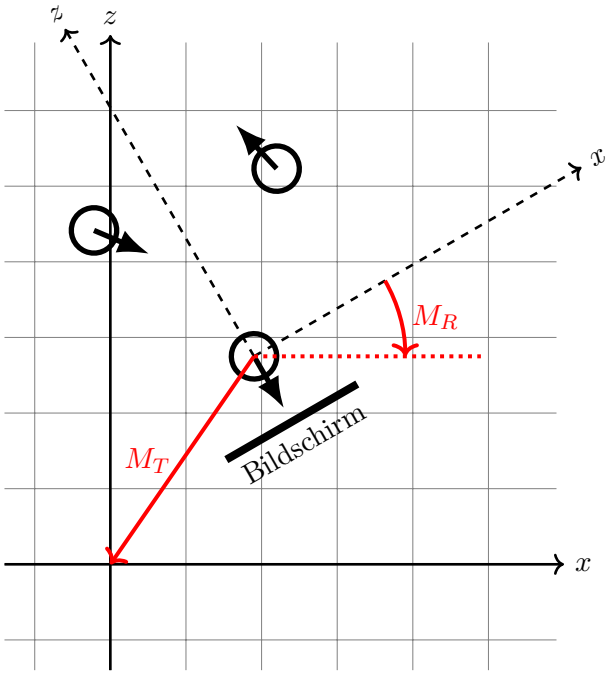


Abbildung 3.7: Transformiertes Koordinatensystem durch Referenzperson

Die Inverse der entstandenen Transformationsmatrix wird auf alle Schlüsselpunkte angewendet.

Formal bedeutet das:

$V :=$ Menge der Schlüsselpunkt-Vektoren

$M_R :=$ Rotation des Starrkörpers der Referenzperson zum Weltursprung

$M_T :=$ Translation des Starrkörpers der Referenzperson zum Weltursprung

$$M_C = (M_R \cdot M_T)^{-1}$$

Die Menge der transformierten Vektoren V' wird durch die Anwendung der folgenden Berechnungsvorschrift auf die Menge V gebildet:

$$V' = \{M_C \cdot \overrightarrow{v} \mid \overrightarrow{v} \in V\}$$

Alle Schlüsselpunkte sind nun so transformiert, dass der Mittelpunkt der Referenzperson, bei nach der Kalibrierung nicht veränderter Position und Ausrichtung, im Nullpunkt des Weltkoordinatensystems positioniert ist. Zudem entspricht die Orientierung nun einer frontalen Ausrichtung entlang der Tiefenachse des Weltkoordinatensystems. Die relativen Abstände aller Schlüsselpunkte sind dabei erhalten geblieben, daher weisen auch alle eventuell weiteren erkannten Personen die gleichen relativen Abstände und Orientierungen zur Referenzperson auf.

Die entstandene »Referenzperson-Transformationsmatrix« wird als Kalibrierung gespeichert und identisch für alle weiteren Aktualisierungen von Schlüsselpunkten angewendet. Sie stellt nun eine Art Korrekturmatrix für einen konkreten Aufbau (Position des Bildschirms und der Kamera-Hardware) dar.

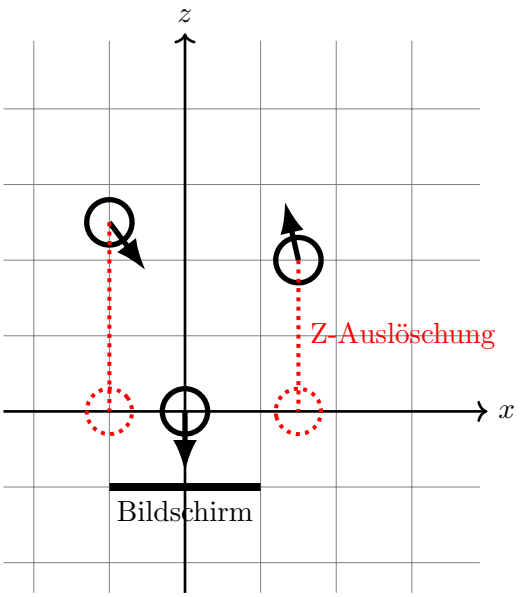


Abbildung 3.8: Erkannte Personen nach Kalibrierung

Für die Verwendung zur Interaktion mit zweidimensionalen Anwendungen kann abschließend die Tiefe aus den transformierten Schlüsselpunkten entfernt werden. Dies kann auch in diesem Ansatz mit einer zusätzlichen Transformation durch eine Orthogonalprojektionsmatrix auf die XY-Ebene (Skalierung der Tiefenachse mit 0) durchgeführt werden.

3.1.4 Festlegung

Beide betrachteten Ansätze sind prinzipiell zur Lösung des Problems geeignet. Beim ersten Verfahren werden dabei bestehende Grundlagen einer virtuellen Kamera bzw. ihre Transformationsmatrix verwendet, um ein zweidimensionales Abbild zu erzeugen. Allerdings ist bei diesem Verfahren der weitere Implementierungs- und Berechnungsaufwand höher, da bspw. die Konzeption eines stabilen Verfahrens für die manuelle oder automatische Positionierung der virtuellen Kamera in korrekter Ausrichtung noch zusätzlich berücksichtigt werden muss. Des Weiteren bringt die Verwendung einer virtuellen Kamera möglicherweise Einschränkungen mit sich, wenn zum Beispiel die Kamera sowohl als Transformationsgrundlage als auch für die Darstellung verwendet werden soll. In diesem Fall muss die Anwendung bzw. das Interaktionsszenario noch zur Darstellung in der virtuellen Kamera angepasst werden.

Das erkennungsdaten-basierte Verfahren ist unabhängig von einer dedizierten virtuellen Kamera für die 3D-2D-Transformation, da eine Referenzperson und deren Transformationsmatrix als Grundlage für eine Starrkörpertransformation der Schlüsselpunkte dient.

Durch dieses Verfahren erhält man direkt Schlüsselpunkte aller Personen relativ zur Referenzperson, welche durch die Transformation in den Ursprung der Welt (Nullpunkt) verschoben wurde. Die Ausrichtung der Gesamtheit an Personen (der Starrkörper) wird gleichzeitig so korrigiert, dass die Blickrichtung der Referenzperson in Richtung der Tiefenachse zeigt. Das bedeutet, wenn mehrere Personen gerade vor dem Bildschirm stehen, sind diese relativ zum kalibriertem Ursprung positioniert bzw. ausgerichtet und schauen daher in Richtung der Tiefenachse.

Wenn nun die darstellende Kamera der Anwendung auf den Weltursprung (Nullpunkt) fokussiert ist, können die überführten Schlüsselpunkte der Personen mit wenigen Anpassungen für eine Interaktion auf dem Bildschirm verwendet werden.

In Anbetracht der Vorteile des zweiten Lösungsansatzes wird dieser für die folgende Implementierung verwendet.

3.2 Implementierung als Unity-Komponente

Die Implementierung des Lösungsansatzes wird als Unity-Komponente durchgeführt (`BodyTrackingManager2D`). Diese soll kontinuierlich, nach dem beschriebenen Verfahren, eine wiederverwendbare Basis zur Aufbereitung und Bereitstellung der überführten Schlüsselpunkte einer jeder erkannten Person bilden. An diese Komponente sollen sich event-basiert andere Komponenten anbinden lassen, um die eingehenden Daten für die zweidimensionale Interaktion zu verarbeiten.

3.2.1 Anforderungen an die Komponente

Für die Implementierung der Unity-Komponente werden zunächst die Anforderungen an diese festgestellt, um das in der Einleitung (Kapitel 1) genannte Grundkonzept unter Verwendung des gewählten Lösungsansatzes (Unterabschnitt 3.1.3) durchzuführen. Diese werden ergänzt durch Funktionen, welche für die Nutzung in anderen Komponenten notwendig sind. Dieser Abschnitt spezifiziert dabei thematisch gruppiert die zu implementierenden funktionalen Anforderungen an die Komponente bzw. ihre Bestandteile. Die Anforderungen werden zur besseren Referenzierung fortlaufend nummeriert.

3.2.1.1 Grundlagen und Initialisierung

[A01] Grundkonfiguration und Anbindung an Events des ZED Manager Die Komponente soll bei Start den ZED Manager an sich anbinden und die Komponente dafür vorzubereiten, Daten vom ZED Manager zu empfangen.

[A02] Start der Erkennung Der Start der Erkennung des »Body Tracking Module« über das ZED-System soll automatisch oder über einen Methodenaufruf möglich sein. Dabei soll die Komponente beim Starten des Trackings die für die Konfiguration notwendigen Einstellungen bzw. Parameter des ZED Managers festlegen, um das »Body Tracking Module« des SDKs auf die Erfassung von Personen vorzubereiten.

3.2.1.2 Kalibrierung

[A03] Kalibrierungsdaten erheben Die Komponente soll die Kalibrierung entsprechend des in Kapitel 3.1.3 beschriebenen Verfahrens realisieren. Dafür soll die erste vom ZED-System erkannte Person mit ihrer gemittelter Position und Grundausrichtung verwendet werden. Diese bilden die Kalibrierungsdaten. Die Komponente soll prüfen, ob nach Start der Erkennung vom ZED-System die Körperdaten einer Person übermittelt werden, und sonst gegebenenfalls warten, bis eine Person erkannt wird. Des Weiteren soll, nachdem eine Person erkannt wurde, eine konfigurierbare Zeitspanne ablaufen, bevor die Kalibrierungsdaten endgültig erhoben werden. Dies soll dazu dienen, dass je nach Anwendung die Person auch nach Start der Kalibrierung ihre Position und Ausrichtung vor dem Bildschirm noch korrigieren kann. Die Kalibrierung soll abgebrochen und neu begonnen werden, wenn das System während dieser Zeitspanne die Erkennung der dedizierten Person verliert. Wenn diese Zeitspanne fünfmal verstreicht, ohne zu einem Kalibrierungsergebnis zu kommen, gilt die Kalibrierung als fehlgeschlagen.

[A04] Kalibrierung starten Der Start der Kalibrierung soll konfigurierbar entweder automatisch nach Start und Initialisierung der Komponente erfolgen oder über einen Methodenaufruf möglich sein. Es gilt, vorher zu prüfen, ob die Erkennung bereits gestartet wurde (siehe [A02]), und dies sonst gegebenenfalls nachzuholen.

[A05] Status der Kalibrierung Es soll ein Status der Kalibrierung bereitgestellt werden, welcher für Prozeduren in anderen Komponenten bzw. zur Visualisierung in der Anwendung genutzt werden kann. Dieser soll folgende Stufen umfassen:

- `Inactive`: Komponente noch nicht bereit zur Kalibrierung. Bspw. weil das ZED-System nicht bereit ist.
- `Ready`: Die Komponente ist bereit für die Kalibrierung anhand der ersten erkannten Person.
- `Running`: Der Kalibrierungsvorgang läuft. Dieser Status wird so lange beibehalten, bis eine Person erfolgreich kalibriert wurde oder die Kalibrierung fehlschlägt.

- `Done`: Die Kalibrierung ist fertiggestellt und es wurden die erforderlichen Daten ermittelt.
- `Failed`: Die Kalibrierung ist fehlgeschlagen, da die konfigurierte Zeitspanne fünfmal verstrichen ist, ohne zu einem Kalibrierungsergebnis zu kommen.

[A06] Caching der Kalibrierung für eine bestimmte Szene Die ermittelten Kalibrierungsdaten sollen optional für eine bestimmte Szene persistent zwischengespeichert werden können. Dies soll ermöglichen, dass nicht bei jedem Start einer konkreten Anwendung bzw. Szene eine Kalibrierung durchgeführt werden muss, sofern der Aufbau in der Realwelt (Position des Bildschirms bzw. Position der ZED-Kamera) nicht verändert wurde.

[A07] Verwendung von Kalibrierungsdaten aus Caching Vor dem Start der Kalibrierung soll optional überprüft werden, ob für die aktuelle Szene zwischengespeicherte Kalibrierungsdaten vorliegen.

Falls ein Cache vorhanden ist und verwendet werden soll, werden die Kalibrierungsdaten entnommen und für die spätere Verwendung angewendet. Der Status der Kalibrierung wechselt dann sofort zu `Done`.

[A08] Konfiguration des Caches Es soll konfigurierbar sein, ob ein vorhandener Cache genutzt werden soll bzw. ob ein vorhandener Cache überschrieben werden soll.

3.2.1.3 3D-2D-Transformation

[A09] Transformation der Schlüsselpunkte Die Komponente soll auf Basis des Verfahrens die Schlüsselpunkte einer jeden erkannten Person kontinuierlich für die zweidimensionale Verwendung transformieren. Die überführten Schlüsselpunkte sollen anschließend pro Person mit einem geeigneten Datenmodell (siehe Unterunterabschnitt 3.2.1.4) gekapselt werden, welches auch die Zuordnung zu einer bestimmten Person berücksichtigt. Die Komponente hat zusätzlich noch die Aufgabe, den Lebenszyklus der daraus resultierenden Sammlung an Datenobjekten zu verwalten. Das bedeutet, dass diese bei neu erkannten Personen hinzugefügt werden, für bestehende Personen aktualisiert werden und für nicht mehr erkannte Personen entfernt werden.

[A10] Einstellbare Modifikatoren (Offsets) Zur besseren Anpassung an die Anwendung sollen einstellbare Modifikatoren (Offsets) konfigurierbar sein, welche für Verschiebung und Skalierung der Gesamtheit aller Personen bzw. ihrer Schlüsselpunkte (des Starrkörpers) verwendet werden. Diese werden zusätzlich auf die transformierten Schlüsselpunkte angewendet und ermöglichen beispielsweise eine Vergrößerung der interagierenden Personen oder Verschiebung des Startpunkts zur Anpassung an die Anwendung.

[A11] Varianten von Schlüsselpunkten Für jede Person sollen die folgenden Varianten von Schlüsselpunkten als Array zur Verfügung stehen.

- `Keypoints2D`: Transformierte und modifizierte Schlüsselpunkte
- `RawKeypoints2D`: Transformierte aber unmodifizierte Schlüsselpunkte
- `RelativeKeypoints2D`: Relative Schlüsselpunkte der Einzelperson bzw. lokale Bewegung. Die Schlüsselpunkte sind relativ zur Person selbst und können daher zum Beispiel für positionsunabhängige Szenarien verwendet werden.

3.2.1.4 Datenmodell für transformierte Körperdaten

[A12] Grundanforderung an das Datenmodell »Body2D« Für jede erkannte und überführte Person soll eine Datenstruktur auf Basis einer Klasse existieren. Die Instanz kapselt die überführten Schlüsselpunkte als Array in verschiedenen Varianten und ordnet diesen einem eindeutigen Personenschlüssel zu. Die Gleichheit von Instanzen soll dabei durch diesen Personenschlüssel definiert werden, damit Komponenten, welche die Daten verarbeiten, direkt die Gleichheit von Personen durch boolesche Ausdrücke feststellen können.

[A13] Definition von Körperformaten Das Datenmodell kann verschiedene mögliche Körperformate als Ausgabeformate definieren und konvertiert die eingehenden Schlüsselpunkte vom ZED-System zu diesen. Ein Körperformat ist definiert durch eine bestimmte Anzahl an Schlüsselpunkten eines Körpers in einer festen Reihenfolge. Zu jedem Körpermodell sollen Datenstrukturen angeboten werden, um die Indizes der Schlüsselpunkt-Arrays zum Namen der Position zuzuordnen, zum Beispiel `Neck` $\rightarrow$ `4`.

3.2.1.5 Events

[A14] Event bei Aktualisierung des Kalibrierungsstatus Die Komponente soll bei Änderungen des Kalibrierungsstatus ein Event bereitstellen, an welchem sich andere Komponenten anbinden können, um auf Zustandsänderungen bei der Kalibrierung zu reagieren.

[A15] Event für Signalisierung, dass die Komponente bereit ist Die Komponente soll ein Event bereitstellen, das ausgelöst wird, sofern die Kalibrierung abgeschlossen ist und sie bereit ist, transformierte Körperdaten bereitzustellen.

[A16] Event bei Aktualisierung von überführten Körperdaten »Body2D« Die Komponente soll ein Event bereitstellen, welches bei jeder Aktualisierung von Körperdaten ausgelöst wird. Das Event soll dabei an die Empfänger eine Liste mit aktuellen transformierten Körperdaten übergeben.

3.2.1.6 Konfiguration

[A17] Konfiguration des Ausgabeformats Das Körperformat, welches zur Ausgabe nach der Transformation verwendet wird, soll konfigurierbar sein. Dabei sollen die durch das Datenmodell verfügbaren Körperformate wählbar sein.

[A18] Konfiguration der Erkennung Einige Optionen für die Konfiguration des ZED-»Body Tracking Module« sollen durch die Komponente verfügbar sein und bei Verwendung dieser angewendet werden.

- **Aktivierung/Deaktivierung von »Body Fitting«**:[11] Verbessert die Erkennung bei fehlenden Daten. Der Prozess nutzt die Historie jeder verfolgten Person, um fehlende Schlüsselpunkte dank der kinematischen Beschränkung des menschlichen Körpers zu ermitteln.

- **Erkennungssicherheit (»Recognition Confidence«)**:[12] Prozentualer Mindestwert für das Vertrauen in die korrekte Erkennung einer Person. Durch das ZED-System erkannte Personen unterhalb des Schwellenwerts werden nicht in die Erkennung mit einbezogen.

3.2.2 Grundlage und Aufbau der Komponente

Im Folgendem wird auf den grundlegenden Aufbau der Komponente bzw. die geplante Integration in ein Gesamtsystem eingegangen. Es geht im Kern um die Entwicklung der Komponente `BodyTrackingManager2D`. Diese ist so konzipiert, dass sie einen Aufsatz auf den ZED Manager darstellt, diesen steuert und die Daten des ZED-Systems empfängt, transformiert und mit einem angepassten Datenmodell zur Verfügung stellt.

3.2.2.1 Systemüberblick und Datenfluss

Zunächst wird das System und dessen Daten- und Steuerfluss im Überblick gezeigt. Zusätzlich wird die verwendete Datenbasis vom ZED-System betrachtet, auf welcher die `BodyTrackingManager2D`-Komponente aufsetzen wird.

[11] s. Stereolabs Inc., *Body Tracking Overview*, Abschnitt »3D BODY FITTING«.
[12] s. ebd., Abschnitt »Detection Outputs«.

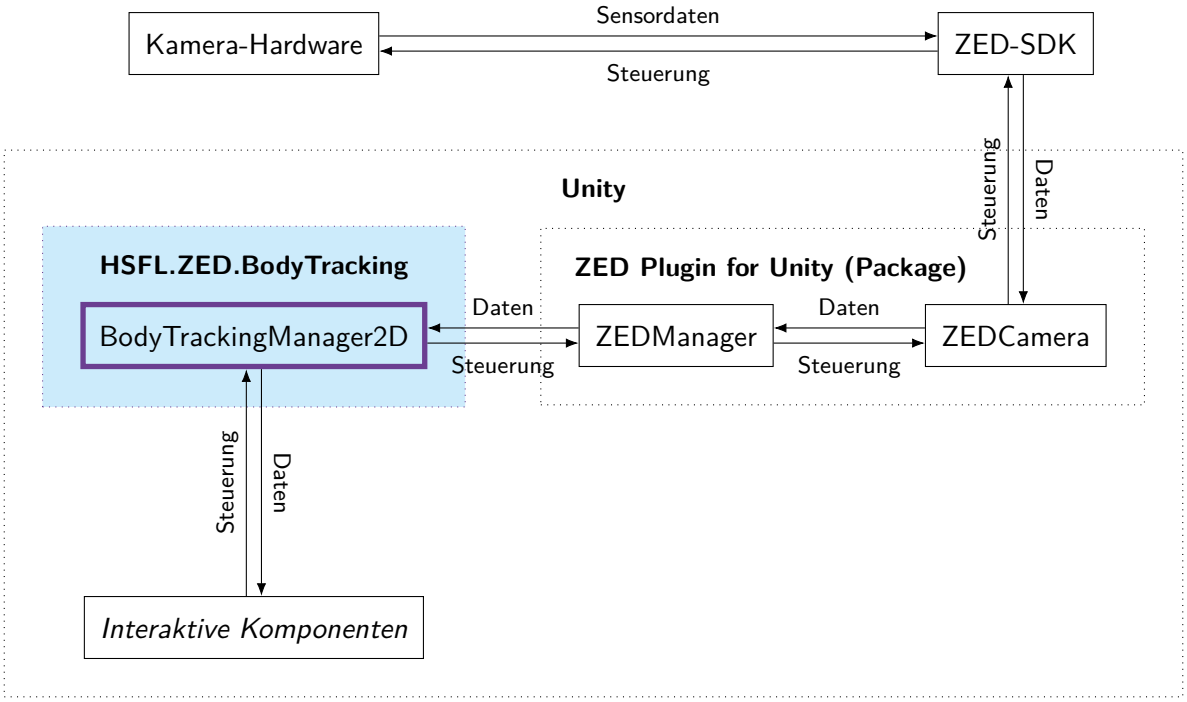


Abbildung 3.9: Systemüberblick (Datenflussmodell)

Das vollständige System besteht aus der Kamera-Hardware, dem nativen ZED-SDK und dem bereitgestellten Unity-Paket »ZED Plugin for Unity«, ergänzt um die selbst implementierte Komponente `BodyTrackingManager2D`. Verschiedene interaktive Komponenten empfangen die überführten Körperdaten von der Komponente `BodyTrackingManager2D` zur Steuerung der Anwendung. Ein wichtiger Teil des Datenflusses für die Implementierung sind die bereitgestellten Daten des `ZEDManager`s. Dieser stellt aktuelle Erkennungsdaten eventbasiert bereit. Im folgenden Abschnitt werden die Klassen bzw. Strukturen der Daten vom `ZEDManager` zum `BodyTrackingManager2D` beschrieben, welche für die Implementierung benötigt werden.

3.2.2.2 Daten vom `ZEDManager`

Der `ZEDManager` stellt die aufbereiteten Erkennungsdaten des SDKs eventbasiert bereit, wenn eine Kamera-Hardware verfügbar oder eine geeignete Aufzeichnung (SVO) vorhanden ist und das »Body Tracking Module« gestartet wurde. Wie die Anbindung an diese Daten stattfindet, folgt in den Implementierungsdetails.

Es steht keine dedizierte Dokumentation für das Unity-Paket »ZED Plugin for Unity« bereit, weshalb einige Ausführungen zu einzelnen relevanten Klassen des Unity-Paketes folgen.

BodyTrackingFrame Die Grundlage der Daten stellt dabei die kontinuierliche Übermittlung eines sogenannten `BodyTrackingFrame`s dar. Bei jedem Event wird ein aktualisiertes Objekt dieses Typs übergeben. Dieses hält unter anderem alle erkannten Körper bzw. Personen während eines einzelnen Erkennungsframes. Es handelt sich dabei um eine spezifisch für Unity entwickelte Klasse, welche die Körper[13] aus dem SDK für Unity aufbereitet zur Verfügung stellt und auch weitere Metadaten und Filterfunktionen bereitstellt.

Konkret werden folgende für die Implementierung relevanten Eigenschaften bzw. Methoden in einem `BodyTrackingFrame` bereitgestellt:

- `public Bodies rawbodies`: Körperdaten direkt aus dem ZED-SDK. Stellt alle Erkennungen dar, die während eines einzelnen Erkennungsframes erfasst wurden.
- `public int bodyCount`: Wie viele Körper insgesamt entdeckt wurden. Ohne Berücksichtigung von eventuellen Filtern.
- `public List<DetectedBody> detectedBodies`: Alle Körper als Liste von `DetectedBody`-Objekten, die innerhalb dieses Frames entdeckt wurden.
- `public List<DetectedBody> GetFilteredObjectList( bool tracking_ok, bool tracking_searching, bool tracking_off, float confidencemin = 0)`: Erzeugt eine gefilterte Liste der Körper als `DetectedBody`-Objekt, die innerhalb dieses Frames entdeckt wurden. Mögliche Filter sind hierbei:
  - `tracking_ok`: Körper, welche auf aktuell erkannten Daten basieren.
  - `tracking_searching`: Körpern, welche bisher erkannt wurden, aber welche vom SDK aktuell gesucht werden. Werden mit den letzten bekannte Daten gelistet.
  - `tracking_off`: Körper, welche bisher erkannt wurden, aber nicht mehr vom SDK mit Daten versorgt werden. *Zum Beispiel, weil die Erkennung pausiert ist oder nicht verfügbar ist.*

DetectedBody Die innerhalb eines `BodyTrackingFrame`s erkannten Körper werden als Liste von `DetectedBody`-Objekten bereitgestellt. Dabei repräsentiert jedes `DetectedBody`-Objekt einen einzelnen Körper, welcher vom »Body Tracking Module« innerhalb eines `BodyTrackingFrame` erkannt wurde. Es handelt sich dabei um eine spezifisch für Unity entwickelte Klasse, welche die Positionsdaten einer einzelnen Person[14] aus dem SDK für Unity aufbereitet repräsentiert. Dabei enthält es allerdings hauptsächlich Daten über die Position und Orientierung des gesamten Körpers bzw. dessen Ausmaße relativ zum ZED-System.

[13] s. Stereolabs Inc., *Bodies – ZED SDK API Reference (C++)*.
[14] s. Stereolabs Inc., *BodyData – ZED SDK API Reference (C++)*.

Die eigentlichen Daten über die Pose bzw. Position der Schlüsselpunkte der Person und weitere Informationen befinden sich nicht direkt im `DetectedBody`-Objekt, sondern in der Eigenschaft `public BodyData rawBodyData` des Objekts.

BodyData Bei der Struktur `BodyData` handelt es sich um die direkte Repräsentation der vom SDK erhobenen Rohdaten. Sie enthält unter anderem eine Identifikationsnummer (ID), `Vector3`-Arrays mit den Positionen der Schlüsselpunkte relativ zum ZED-System, Informationen zur aktuellen Orientierung des Körpers, Statusinformationen zur Erkennung und weitere Daten des SDKs.[15]

Es handelt es sich also um die eigentlichen Körperdaten, welche für die weitere Implementierung gebraucht werden. Die relevanten Eigenschaften sind dabei:

- `public int id`: ID, die dem Körper bei der ersten Erkennung vom SDK zugewiesen wird. Sie bleibt zwischen den Frames bestehen, wenn der Körper sichtbar bleibt und erkannt wird. Anhand dieser kann verfolgt werden, ob in einem aktualisiertem `BodyTrackingFrame` derselbe Körper gemeint ist. Das SDK stellt die Eindeutigkeit sicher.

- `public Vector3 position`: Enthält die gemittelte Position des Körpers in Relation zum ZED-System.

- `public float confidence`: Prozentualer Wert für das Vertrauen in die korrekte Erkennung einer Person. Das bedeutet, wie sicher sich das SDK ist, dass es sich bei diesem Körper tatsächlich um eine gültige Erkennung handelt. Höhere Werte sind besser. So besteht zum Beispiel bei einem Wert von 99 eine Wahrscheinlichkeit von 99 %, dass es sich tatsächlich um eine korrekt erkannte Person bzw. Pose handelt.

- `public Vector3[] keypoint`: Enthält die Liste der Schlüsselpunkte des Körpers. Diese Daten werden für das eigentliche Ziel benötigt und von der Komponente später transformiert. Sie entsprechen dabei einem Array von Schlüsselpunkten im Raum relativ zum ZED-System in einer bestimmten Reihenfolge. Die Reihenfolge leitet sich dabei vom gewählten und daher vom SDK verwendeten Körperformat ab (siehe Kapitel 2).

- `public Quaternion globalRootOrientation`: Die globale Grundausrichtung des Körpers bzw. des Starrköpers der einzelnen Person (sozusagen des Skeletts).

Weiterhin stehen im `BodyData`-Objekt auch direkt zweidimensionale Schlüsselpunkte (`public Vector2[] keypoint2D`) bereit. Allerdings sind diese für das Ziel einer frontalen Sicht ungeeignet, da es sich um zweidimensionale Punkte aus der Sicht des ZED-Systems (ähnlich Bildschirmkoordinaten) handelt und diese

[15] s. Stereolabs Inc., *BodyData – ZED SDK API Reference (C++)*.

aufgrund fehlender Tiefen- und Rotationsdaten für eine Transformation ungeeignet sind. Sie sind rein dafür konzipiert, als Überlagerung für das Kamerabild des ZED-Systems verwendet zu werden. Daher können diese nicht für den weiteren Verlauf benutzt werden.

Zusammenfassung der Datengrundlage Die genannten Klassen bzw. Strukturen stellen damit die Grundlage der Daten dar, welche das SDK bereitstellt und zur Implementierung der `BodyTrackingManager2D`-Komponente benötigt werden. Zusammengefasst befinden sich die benötigten erhobenen Rohdaten als `BodyData`-Objekte innerhalb von `DetectedBody`-Objekten, welche als Liste vom ZED-Manager in einem `BodyTrackingFrame` pro Erkennungsframe übermittelt werden.

3.2.2.3 Aufbau und Bestandteile der Komponente

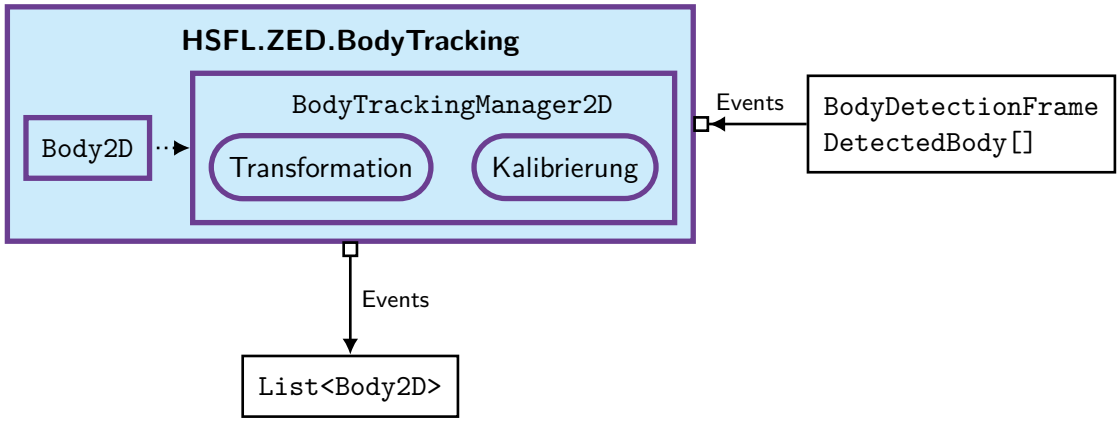


Abbildung 3.10: Systemzerlegung (`BodyTrackingManager2D`-Komponente)

Zusammengenommen besteht die Implementierung dieses Teilaspekts der Arbeit aus einer Unity-Komponente, welche die `ZEDManager`-Komponente steuert und die `BodyTrackingFrame`-Objekte von dieser empfängt. Diese wird im Folgenden als `BodyTrackingManager2D`-Komponente bezeichnet. Sie führt den Kalibrierungsprozess aus und transformiert anschließend kontinuierlich die Schlüsselpunkte der eingehenden Körperdaten pro Person und überführt diese in die gewünschten Schlüsselpunktvarianten (siehe Unterunterabschnitt 3.2.1.3 [A11]). Die überführten Schlüsselpunkte werden in einer geeigneten Datenstruktur (`Body2D`) gekapselt (siehe Unterunterabschnitt 3.2.1.4) und intern verwaltet. Bei jeder Aktualisierung werden die verwalteten bzw. aktuell vorhandenen `Body2D`-Objekte als Liste per Event bereitgestellt. Diese Daten können dann von anderen Komponenten für interaktive Konzepte verwendet werden und stellen damit auch die Basis des späteren prototypischen Beispiels dar.

Im Folgenden wird nun die konkrete Implementierung erläutert; es wird primär auf den Bereich für die Implementierung des Verfahrens eingegangen.

3.2.3 Anbindung an das ZED-System

Um Daten vom ZED-System in der Komponente `BodyTrackingManager2D` empfangen und es steuern zu können, muss dieses angebunden werden. Dafür wird eine Referenz auf eine Instanz des `ZEDManager`s in der Szene benötigt. Zusätzlich wird beim Start der Komponente überprüft, ob diese vorhanden ist, und gegebenenfalls versucht, die Referenz über eine Objektsuche in der Szene zu finden.

```
// BodyTrackingManager2D.cs
// ...
public ZEDManager zedManager;

// ...
private void Start()
{
    // Search for ZED Manager if not set
    zedManager = zedManager ? zedManager :
       FindObjectOfType<ZEDManager>();
    if (zedManager == null)
    {
        throw new Exception("ZED Manager not found in
           scene.");
    }

    // ZED Manager is available when code reaches this
       point

    zedManager.OnZEDReady += OnReady;
    zedManager.OnBodyTracking += OnBodyTrackingUpdate;
}
// ...
```

Mit der Referenz kann direkt eine Anbindung an die notwendigen Events der `ZEDManager`-Komponente stattfinden. Dies ist essenzieller Bestandteil der Implementierung und bereitet den beschriebenen Datenfluss (siehe Unterunterabschnitt 3.2.2.1) zwischen `ZEDManager` und dieser Komponente vor. Die beiden Event-Handler-Methoden werden den Events zugeordnet und entsprechen den folgenden Signaturen:

- `private void OnReady;` Wird aufgerufen, sobald der `ZEDManager`, die Kamera-Hardware und das SDK bereit sind.
- `private void OnBodyTrackingUpdate(BodyTrackingFrame bodyTrackingFrame);` wird bei jedem vom SDK bereitgestellten Erkennungframe aufgerufen und bekommt dabei das aktuelle `BodyTrackingFrame` übergeben.

Damit ist die grundlegende Anbindung an die `ZEDManager`-Komponente abgeschlossen.

Die Komponente soll bei Bedarf die Erkennung automatisch starten, sobald das ZED-System bereit ist. Dafür eignet es sich, den eben erschaffenen und angebundenen Event-Handler zu verwenden, um eine geeignete Startmethode für die Erkennung aufzurufen.

```
// BodyTrackingManager2D.cs
// ...

[Header("System Control")]
[Tooltip("Auto start tracking when play mode is on and
   ZED ready.")]
public bool startTrackingAutomatically = true;

// ...
/// <summary>
/// Called when depth camera SDK is ready.
/// </summary>
private void OnReady()
{
    if (startTrackingAutomatically)
    {
        StartTracking();
        // ...
    }
}
// ...
```

Die Konfiguration und das Starten der Erkennung kann nun in eine eigene öffentliche Methode ausgelagert werden. Dies ermöglicht auch den gewünschten manuellen Start der Erkennung (siehe Unterunterabschnitt 3.2.1.1 [A01]). Um die Erkennung zu starten, wird die Konfiguration des SDKs bzw. »Body Tracking Modules« über die `ZEDManager`-Komponente angepasst. Grundlegend sollen die Einstellungen des ZED-Systems (wie zum Beispiel die Erkennungsqualität) direkt im Inspektor der `ZEDManager`-Komponente angepasst werden (siehe Kapitel 2). Die Komponente `BodyTrackingManager2D` soll nur notwendige Einstellungen, welche für einen nutzbaren Erkennungs- und Transformationsprozess hilfreich sind, überschreiben.

```
// BodyTrackingManager2D.cs
// ...

[Tooltip("Disables or enabled the body fitting while body
   detection running.")]
public bool bodyFitting = true;

// ...

/// <summary>
/// Configures the ZED Manager, starts the calibration
   and the tracking.
/// </summary>
public void StartTracking()
{
    if (!zedManager.IsBodyTrackingRunning)
    {
        // Not necessary for the purpose and saves
           resources when deactivated.
        zedManager.bodyTracking2DMask = false;
        // Improves tracking of a person over multiple
           frames.
        zedManager.bodyTrackingTracking = true;
        // Improves tracking of missing points with human
           kinematic constraints.
        zedManager.enableBodyFitting = bodyFitting;

        zedManager.bodyFormat = BODY_FORMAT.BODY_38;

        zedManager.StartBodyTracking();
        // ...
    }

    // ...

    onBody2DTrackingStarted.Invoke();
}

// ...
```

Die Konfiguration steht dabei in Form von Eigenschaften des `ZEDManager`-Objekts zur Verfügung. Eine grundlegende Einstellung stellt die Wahl des Körperformats dar, welches vom ZED-System als Ausgabeformat verwendet werden soll. Es stehen verschiedene Körperformate zur Auswahl.[16] Diese sind mit dem aktuellen Stand des SDKs:

- `BODY_18`: Enthält 18 Schlüsselpunkte, die der COCO18-Skelettdarstellung entsprechen.
- `BODY_34`: Enthält 34 Schlüsselpunkte.
- `BODY_38`: Entspricht dem `BODY_34`-Format, mit mehr Schlüsselpunkten an den Extremitäten.
- `BODY_70`: Entspricht dem `BODY_38`-Format, mit einer zusätzlichen detaillierten Erkennung der Hand

3.2.3.1 Ausgabekörperformat des ZED-Systems (`BODY_38`)

In dieser Implementierung wird das `BODY_38`-Format verwendet, welches einen für zweidimensionale Anwendungen geeigneten Detailgrad bietet, ohne einen größeren Leistungsverlust zu verursachen.

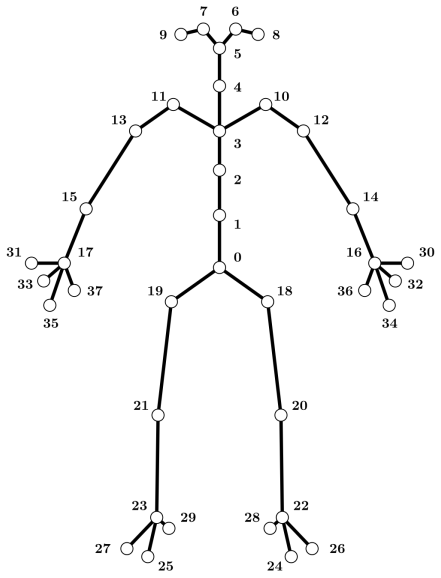


Abbildung 3.11: Körperformat mit 38 Schlüsselpunkten (mit definierter Reihenfolge)

[16] s. Stereolabs Inc., *Body Tracking Overview*, Abschnitt »How It Works«.

Die Schlüsselpunkte der übergebenen Bewegungsdaten sind entsprechend der Reihenfolge in der folgenden Tabelle indiziert:

Index	Bezeichnung	Index	Bezeichnung	Index	Bezeichnung
0	PELVIS	13	RIGHT_SHOULDER	26	LEFT_SMALL_TOE
1	SPINE_1	14	LEFT_ELBOW	27	RIGHT_SMALL_TOE
2	SPINE_2	15	RIGHT_ELBOW	28	LEFT_HEEL
3	SPINE_3	16	LEFT_WRIST	29	RIGHT_HEEL
4	NECK	17	RIGHT_WRIST	30	LEFT_HAND_THUMB_4
5	NOSE	18	LEFT_HIP	31	RIGHT_HAND_THUMB_4
6	LEFT_EYE	19	RIGHT_HIP	32	LEFT_HAND_INDEX_1
7	RIGHT_EYE	20	LEFT_KNEE	33	RIGHT_HAND_INDEX_1
8	LEFT_EAR	21	RIGHT_KNEE	34	LEFT_HAND_MIDDLE_4
9	RIGHT_EAR	22	LEFT_ANKLE	35	RIGHT_HAND_MIDDLE_4
10	LEFT_CALVICLE	23	RIGHT_ANKLE	36	LEFT_HAND_PINKY_1
11	RIGHT_CLAVICLE	24	LEFT_BIG_TOE	37	RIGHT_HAND_PINKY_1
12	LEFT_SHOULDER	25	RIGHT_BIG_TOE		

Tabelle 3.1: Tabelle der Schlüsselpunkte für das `BODY_38`-Format

3.2.3.2 Start der Erkennung

Der Startvorgang des »Body Tracking Modules« wird über den entsprechenden Methodenaufruf `zedManager.StartBodyTracking()` durchgeführt, sofern die Kamera-Hardware verfügbar bzw. in der `ZEDManager`-Komponente eine kompatible Aufzeichnung eingestellt ist. In der vorliegenden Version des »ZED Plugin for Unity« führt der Start des Moduls dazu, dass die Unity-Anwendung für einige Sekunden einfriert. Dies liegt daran, dass dieser Vorgang auf dem Haupt-Thread von Unity ausgeführt wird und diesen bis zur Rückmeldung vom SDK blockiert.

Nach dem Startvorgang beginnt das ZED-System, aktuelle Erkennungframes zu generieren und bereitzustellen. Diese werden von der `ZEDManager`-Komponente nun eventbasiert an die `BodyTrackingManager2D`-Komponente weitergegeben.

Nun wird die definierte `OnBodyTrackingUpdate`-Methode für jeden Erkennungsframe des SDKs aufgerufen, sodass die übergebenen Daten nun für die Weiterverarbeitung genutzt werden können.

3.2.4 Kalibrierung

Nachdem nun Erkennungsdaten pro Person verfügbar sind, können diese genutzt werden, um die Kalibrierung durchzuführen. Für die Basisimplementierung wird dafür die erste vom ZED-System erkannte Person, also der erste erkannte Körper, verwendet.

3.2.4.1 Kalibrierungsstatus

Die Kalibrierung durchläuft verschiedene Zustände (`CalibrationStatus`), welche wie entsprechend der Anforderungen (siehe Unterunterabschnitt 3.2.1.2 [A05]) wie folgt definiert sind:

```
// BodyTrackingManager2D.cs
// ...

public UnityEvent<CalibrationStatus>
    onCalibrationStatusChanged = new();
// ...

// Calibration
private CalibrationStatus _calibrationStatus =
    CalibrationStatus.Inactive;
// ...

/// <summary>
/// Status of the calibration process of this component.
/// </summary>
public CalibrationStatus CalibrationStatus
{
    get => _calibrationStatus;
    private set
    {
        _calibrationStatus = value;
        onCalibrationStatusChanged.Invoke(value);
    }
}

public enum CalibrationStatus
{
    Inactive,
    Ready,
    Running,
    Done,
    Failed
}
```

Jede Änderung des Kalibrierungsstatus löst automatisch ein Event aus, welches von anderen Komponenten verarbeitet werden kann (siehe Unterunterabschnitt 3.2.1.5 [A14]). Sobald die `ZEDManager`-Komponente bereit ist (und damit die `OnReady`-Methode aufgerufen wird), kann der Kalibrierungsstatus auf `Ready` gesetzt werden.

3.2.4.2 Start des Kalibrierungsvorgang

Die Kalibrierung kann entweder automatisch starten oder durch einen Methodenaufruf durchgeführt werden. Dies geschieht durch das Hinzufügen einer Methode namens `StartCalibration`, welche entweder manuell oder automatisch, nachdem das ZED-System bereit ist, aufgerufen wird. Für den automatischen Start wird die `OnReady`-Methode ergänzt.

```
// BodyTrackingManager2D.cs
// ...

[Tooltip("Start calibration automatically after tracking
   has been automatically started.")]
public bool startCalibrationAutomatically = true;

// ...

private void OnReady()
{
    if (startTrackingAutomatically)
    {
        StartTracking();
        if (startCalibrationAutomatically)
        {
            StartCalibration();
        }
    }
}
// ...
```

Der Start des Kalibrierungsvorganges wird durch eine Statusänderung ausgelöst. Dies ist notwendig, da die für die Kalibrierung erforderlichen Daten kontinuierlich an die definierte `OnBodyTrackingUpdate`-Methode übergeben werden und dort entschieden wird, was mit den eingehenden Körperdaten durchgeführt wird. Es verhält sich damit ähnlich zu der `Update`-Methode in Unity, mit dem Unterschied, dass diese pro Erkennungsframe aufgerufen wird.

```
// BodyTrackingManager2D.cs
// ...

public void StartCalibration()
{
    // If tracking has not been started;
    if (!zedManager.IsBodyTrackingRunning)
    {
```

```
            StartTracking();
        }

        // Start calibration if necessary. Status is only
           ready if no cache should be used.
        if (CalibrationStatus == CalibrationStatus.Ready)
        {
            CalibrationStatus = CalibrationStatus.Running;
        }
    }
    // ...
```

3.2.4.3 Kalibrierungsvorgang

Damit die Kalibrierung durchgeführt werden kann, wird die Event-Handler-Methode `OnBodyTrackingUpdate` vorbereitet:

```
// BodyTrackingManager2D.cs
// ...

/// <summary>
/// Called by the ZED SDK for each new BodyTrackingFrame.
/// </summary>
private void OnBodyTrackingUpdate(BodyTrackingFrame
   bodyTrackingFrame)
{
    if (CalibrationStatus == CalibrationStatus.Running)
    {
        // If in calibration mode and calibration is
           running
    }
    else if (CalibrationStatus is CalibrationStatus.Done
       or CalibrationStatus.Failed)
    {
        // If calibration is completed or completely
           failed, prepare data and send it to event
           handlers.
    }

    // ...
}
// ...
```

Es existieren nun zwei Anweisungsblöcke, jeweils für die laufende Kalibrierung und die spätere Überführung der Schlüsselpunkte. Der Kalibrierungsblock wird nun durch die Kalibrierungsprozedur bzw. ihrer Vorbereitung ersetzt und es werden einige Instanzvariablen deklariert bzw. initialisiert.

```
// BodyTrackingManager2D.cs
// ...

// Calibration
// ...
private Vector3 _calibratedPositionOffset = Vector3.zero;
private Quaternion _calibratedRotationOffset =
   Quaternion.identity;
private int? _idOfCalibrationCandidate = null;
private int _calibrationAttempts = 0;
private BodyTrackingFrame _lastFrame = null;
// ...

private void OnBodyTrackingUpdate(BodyTrackingFrame
   bodyTrackingFrame)
{
    if (CalibrationStatus == CalibrationStatus.Running)
    {
        // Preserve last frame for calibration coroutine
        _lastFrame = bodyTrackingFrame;

        if (_idOfCalibrationCandidate == null &&
           bodyTrackingFrame.bodyCount > 0)
        {
            _idOfCalibrationCandidate =
               bodyTrackingFrame.detectedBodies[0]
                .rawBodyData.id;
            _calibrationAttempts++;

            StartCoroutine(Calibration());
        }
        else if (_calibrationAttempts > 5)
        {
            CalibrationStatus = CalibrationStatus.Failed;
        }
    }
    // ...
}
// ...
```

Die Variablen haben später die folgenden Funktionen:

- `_calibratedPositionOffset`: Kalibrierte Verschiebung, welche durch die Translation der Referenzperson zum Unity-Weltursprung ermittelt wird.
- `_calibratedRotationOffset`: Kalibrierte Rotation, welche durch die Orientierung der Referenzperson zum Unity-Weltursprung ermittelt wird.
- `_idOfCalibrationCandidate`: ID des aktuellen Referenzperson-Kandidaten. Wird verwendet, um nach Ablauf der Kalibrierungszeit zu prüfen, ob die Person noch vorhanden ist.
- `_calibrationAttempts`: Zähler für Kalibrierungsversuche.
- `BodyTrackingFrame`: Hält den letzten Erkennungsframe vom ZED-System.

Sofern noch kein Referenzperson-Kandidat festgelegt wurde und mindestens eine Person erkannt wurde, wird dieser für den aktuellen Kalibrierungsversuch festgelegt.

Die eigentliche Kalibrierung findet in einer Unity-Coroutine statt, da diese eine gewisse Zeit für die Positionierung der Person vor der Kalibrierung warten soll und dieser Vorgang nicht die Anwendung einfrieren soll.

```
// BodyTrackingManager2D.cs
// ...

[Tooltip("Time in seconds the calibration should wait
   before locking the calibration")]
public int calibrationTime = 10;

// ...

private IEnumerator Calibration()
{
    yield return new WaitForSeconds(calibrationTime);

    // Check if "same" person is still there
    var trackedBody =
       _lastFrame.detectedBodies.FirstOrDefault(
        body => body.rawBodyData.id ==
           _idOfCalibrationCandidate);

    if (trackedBody is not null)
    {
        _calibratedPositionOffset =
           trackedBody.rawBodyData.position;
```

```
            _calibratedRotationOffset =
                trackedBody.rawBodyData.globalRootOrientation;

            // ...

            CalibrationStatus = CalibrationStatus.Done;
        }
        else
        {
            // Set candidate to null and restart
            _idOfCalibrationCandidate = null;
        }
    // ...
    }
```

Zunächst wird überprüft, ob nach Ablauf der Zeit dieselbe Referenzperson bzw. ein Körper mit der identischen ID vorhanden ist. Für die eigentliche Kalibrierung stellt das ZED-System geeignete Werte pro Körper zur Verfügung. Aus den Rohdaten eines Körpers kann dessen gemittelte Position (`position`) sowie seine Grundrotation (`globalRootOrientation`) relativ zum Weltursprung entnommen werden. Diese Werte stellen die spätere Grundlage für die Translation aller Werte dar und die entsprechende Variablen (`_calibratedPositionOffset` & `_calibratedRotationOffset`) für die Kalibrierung werden auf diese Werte gesetzt. Nach diesem Vorgang gilt die Kalibrierung als abgeschlossen und der Status wird dementsprechend angepasst.

Sollte kein Körper mit der identischen ID zum Kalibrierungszeitpunkt mehr vorhanden sein, wird der Kandidat zurückgesetzt und die Coroutine beendet. Der Vorgang beginnt daraufhin erneut, solange, bis die Anzahl der maximalen Kalibrierungsversuche (`_calibrationAttepts > 5`) erreicht ist.

3.2.4.4 Caching der Kalibrierung

Damit die ermittelten Kalibrierungswerte nicht bei jedem Start der Szene verloren gehen, werden diese persistent zugeordnet zur aktuellen Szene gespeichert. Dies geschieht, indem die Variablen und die Bezeichnung der Szene serialisiert werden. Der genaue Vorgang des Cachings ist für das Ziel der Komponente sekundär und wird im Folgenden zusammengefasst beschrieben.

Im Editor-Modus von Unity können die Kalibrierungswerte als »Scriptable Object« (`CalibrationData` bzw. `CalibrationDataSerializable`) persistiert werden.

```
// CalibrationData.cs

[Serializable]
[CreateAssetMenu(fileName = "CalibrationData", menuName =
   "HSFL/ZED/BodyTracking/Calibration", order = 1)]
public class CalibrationData : ScriptableObject,
   IEquatable<CalibrationData>
{
    [SerializeField]
    public string sceneIdentifier;
    [SerializeField]
    public Vector3 calibratedPositionOffset =
       Vector3.zero;
    [SerializeField]
    public Quaternion calibratedRotationOffset =
       Quaternion.identity;

    #region Equality Check

    // ...

    #endregion
}
```

Zur Laufzeit ohne Editor-Umgebung in Unity ist die Erstellung und damit der Einsatz von »Scriptable Objects« für diesen Zweck nicht möglich. Daher wird eine Klasse ergänzt, welche die Kalibrierungsdaten auf dem Host-System serialisiert (JSON-Format) und sie im Anwendungsdatenverzeichnis sichert bzw. wieder lädt.

Die notwendigen Ergänzungen für beide Verfahren zum Persistieren nach abgeschlossener Kalibrierung werden der Kalibrierungs-Coroutine hinzugefügt. Zusätzlich wird vor dem Start der Kalibrierung überprüft, ob zwischengespeicherte Kalibrierungsdaten vorliegen und diese gegebenenfalls angewendet und die Kalibrierung dadurch vorzeitig beendet bzw. ausgesetzt.

Die Vorgänge für Speichern, Laden oder Überschreiben können einzeln aktiviert oder deaktiviert werden. Dafür werden der Komponente einige Eigenschaften zur Konfiguration hinzugefügt:

```
// BodyTrackingManager2D.cs
// ...

[Space(5)] [Header("Calibration Configuration")]
// ...
```

```
[Tooltip("Enables the usage of cached calibration when
   available. Either the calibration cache object or one
   in the serialized database.")]
public bool useCachedCalibration = true;

[Tooltip("The cache data which should be used when in
   cache modes.")]
public CalibrationData calibrationCache = null;

[Tooltip("If the calibration routine has a cached one
   available for the scene, it will be discarded and
   overwritten by a new one.")]
public bool ignoreAndOverwriteCachedCalibration = false;

// ...
```

3.2.5 Überführung der Schlüsselpunkte

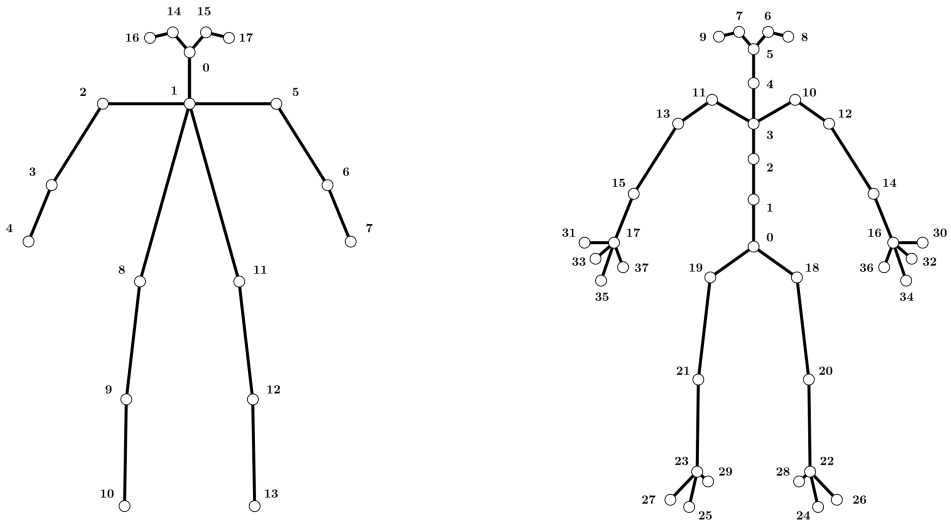
Nachdem die Kalibrierung abgeschlossen ist, können nun die Schlüsselpunkte von allen erkannten Körpern korrigiert werden und in die gewünschten Varianten überführt werden.

3.2.5.1 Datenmodell (»`Body2D`«)

Vor der Erläuterung des eigentlichen Transformationsvorgangs wird das Datenmodell `Body2D` definiert, in welchem die überführten Schlüsselpunkte pro Körper gehalten werden. Dabei werden die Schlüsselpunkte für jeden erkannten Körper in drei Varianten (siehe Unterunterabschnitt 3.2.1.3 [A11]) gehalten. Das Datenformat definiert auch verschiedene Ausgabe-Körperformate (`BodyFormat2D`) und kann daher die eingehenden Schlüsselpunkte abhängig vom Quell-Körperformat auf ein anderes Ausgabeformat konvertieren. Das SDK übergibt unabhängig vom gewünschten Quell-Körperformat immer Schlüsselpunkt-Arrays mit einer Länge von 70. Wobei die ungenutzten Indizes mit Nullvektoren besetzt sind. Die Konvertierung bzw. Kürzung geschieht durch einen sogenannten Adapter, welcher standardmäßig das `BODY_38`-Format vom ZED-System beibehält und die ungenutzten Indizes entfernt oder diese in ein anderes Ausgabeformat umstrukturiert. Dadurch erhält man gleichzeitig eine Abstraktionsschicht: Falls sich die Quell-Körperformate ändern, kann dies im Adapter korrigiert werden, sodass nicht alle darauf bauenden interaktiven Komponenten angepasst werden müssen und sich das Datenmodell prinzipiell auch für andere Erkennungssysteme eignet.

Standardmäßig ist ein Adapter für das `BODY_38`-Format des ZED-Systems vorhanden. Es können eigene Adapter durch Erweiterung einer abstrakten Klasse `Adapter` geschaffen werden.

Folgende Ausgabeformate werden von `Body2D` definiert:



(a) `Body18` (b) `Body38`

Abbildung 3.12: Ausgabeformate (`Body2D`) für überführte Schlüsselpunkte

Die Reihenfolge für das Ausgabeformat `Body38` ist identisch zu der genannten Reihenfolge des Körperformats `BODY_38` des ZED-Systems (siehe Tabelle 3.1). Das Ausgabeformat `Body18` entspricht dem COCO18-Format[17] und ist wie folgt definiert:

Index	Bezeichnung	Index	Bezeichnung
0	NOSE	9	RIGHT_KNEE
1	NECK	10	RIGHT_ANKLE
2	RIGHT_SHOULDER	11	LEFT_HIP
3	RIGHT_ELBOW	12	LEFT_KNEE
4	RIGHT_WRIST	13	LEFT_ANKLE
5	LEFT_SHOULDER	14	RIGHT_EYE
6	LEFT_ELBOW	15	LEFT_EYE
7	LEFT_WRIST	16	RIGHT_EAR
8	RIGHT_HIP	17	LEFT_EAR

Tabelle 3.2: Tabelle der Schlüsselpunkte für das `Body18`-Format

[17] s. Hidalgo u. a., *OpenPose: PoseModel – Namespace Reference.*

Die Definition der Eigenschaften von der `Body2D`-Klasse entspricht dabei folgender Grundstruktur:

```
// Body2D.cs

public class Body2D : IEquatable<Body2D>
{
    public int Id { get; }
    public BodyFormat2D BodyFormat2D { get; private set; }

    // Keypoints with calibration and included offsets
    public Vector3[] Keypoints2D { get; private set; }

    // Keypoints with calibration and without offsets
    public Vector3[] RawKeypoints2D { get; private set; }

    // Keypoints relative to skeleton root which is
       always x=0 & y =0.
    public Vector3[] RelativeKeypoints2D { get; private
       set; }

    public float CurrentTrackingConfidence;

    // Default Adapter
    public Adapter Adapter = new Zed38BodyFormatAdaptor();

    // ...
}

public enum BodyFormat2D
{
    Body18,
    Body38
}
```

Diese Struktur berücksichtigt alle Daten, welche pro überführtem Körper gekapselt werden müssen. Sie enthält Eigenschaften für die eindeutige ID eines Körpers, das gewählte Ausgabekörperformat, alle Schlüsselpunktvarianten, die aktuellen Erkennungssicherheit sowie den genutzten Adapter zur Konvertierung von Schlüsselpunkten in das Ausgabekörperformat.

Bei der Konstruktion einer `Body2D`-Instanz wird die ID des Körpers, das gewünschte Ausgabekörperformat sowie, falls notwendig, erste Werte der Schlüsselpunkte übergeben.

```
// Body2D.cs
// ...

public Body2D(int id, BodyFormat2D bodyFormat2D)
{
    Id = id;
    BodyFormat2D = bodyFormat2D;
    switch (bodyFormat2D)
    {
        case BodyFormat2D.Body18:
            Keypoints2D = new Vector3[18];
            break;
        case BodyFormat2D.Body38:
            Keypoints2D = new Vector3[38];
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }
}

public Body2D(int id, BodyFormat2D bodyFormat2D,
   Vector3[] keypoints, Vector3[] rawKeypoints2D,
    Vector3[] relativeKeypoints) : this(id, bodyFormat2D)
{
    Update(keypoints, rawKeypoints2D, relativeKeypoints);
}
```

Des Weiteren existiert eine `Update`-Methode zur Aktualisierung der Instanz, welche vom `BodyTrackingManager2D` zur Laufzeit verwendet werden kann um die Daten zu aktualisieren.

```
// Body2D.cs
// ...

public void Update(Vector3[] keypoints, Vector3[]
   rawKeypoints, Vector3[] relativeKeypoints)
{
    switch (BodyFormat2D)
    {
        case BodyFormat2D.Body18:
            Keypoints2D = Adapter.AdaptArrayToBody2D(
                keypoints, BodyFormat2D.Body18);
            RawKeypoints2D = Adapter.AdaptArrayToBody2D(
                rawKeypoints, BodyFormat2D.Body18);
```

```
                RelativeKeypoints2D =
                    Adapter.AdaptArrayToBody2D(
                        relativeKeypoints, BodyFormat2D.Body18);
                break;
            case BodyFormat2D.Body38:
                Keypoints2D = Adapter.AdaptArrayToBody2D(
                    keypoints, BodyFormat2D.Body38);
                RawKeypoints2D = Adapter.AdaptArrayToBody2D(
                    rawKeypoints, BodyFormat2D.Body38);
                RelativeKeypoints2D =
                    Adapter.AdaptArrayToBody2D(
                        relativeKeypoints, BodyFormat2D.Body38);
                break;
            default:
                throw new ArgumentOutOfRangeException(
                    nameof(BodyFormat2D));
        }
    }
```

Zur vereinfachten Handhabung sind noch Hilfsstrukturen vorhanden, für eine einfache Zuordnung von Bezeichnung und Array-Index für die Ausgabeformate.

- `public enum KeypointNamesBody18`: Enthält die Bezeichnungen der einzelnen Schlüsselpunkte für das `Body18`-Ausgabeformat
- `public enum KeypointNamesBody38`: Enthält die Bezeichnungen der einzelnen Schlüsselpunkte für das `Body38`-Ausgabeformat

Damit ist das Datenmodell zur Kapselung der überführten Schlüsselpunkte definiert. Es stellt die Basis dar, um die notwendigen Daten zur späteren Verwendung strukturiert anderen Komponenten zur Verfügung zu stellen.

3.2.5.2 Transformation

Für die Implementierung der zweiten Kernaufgabe und zentralen Aufgabe der `BodyTrackingManager2D`-Komponente, wird der zweite Block der Event-Handler-Methode `OnBodyTrackingUpdate` implementiert, welche nun kontinuierlich eingehende Erkennungsframes überführen kann.

Das Endziel der Transformation ist die Ausgabe einer Liste von `Body2D`-Objekten bzw. deren eventbasierte Verteilung pro Erkennungsframe. Wobei jedes Objekt einen erkannten und überführten Körper darstellt. Daher verwaltet die Komponente auch den Lebenszyklus der `Body2D`-Objekte welche von ihr verwaltet werden müssen (siehe Unterunterabschnitt 3.2.1.3 [A09]).

```
// BodyTrackingManager2D.cs
// ...
public float recognitionConfidence = 40;
public UnityEvent<List<Body2D>> onBody2DTrackingUpdate =
    new();
// ...
private void OnBodyTrackingUpdate(BodyTrackingFrame
    bodyTrackingFrame)
{
     // ...
     else if (CalibrationStatus is CalibrationStatus.Done
        or CalibrationStatus.Failed)
     {
          var detectedBodies =
              bodyTrackingFrame.GetFilteredObjectList(true,
                 true, false, recognitionConfidence);

          // Remove missing bodies from data set
          _bodies2D.RemoveWhere(body2D =>
              _bodies2D.Select(body => body.Id).ToList()
                  .Except(detectedBodies.Select(body =>
                     body.rawBodyData.id))
                  .Contains(body2D.Id));

          foreach (var detectedBody in detectedBodies)
          {
              var existingBody2D =
                 _bodies2D.FirstOrDefault(body2D =>
                 body2D.Id == detectedBody.rawBodyData.id);
              if (existingBody2D != null)
              {
                  UpdateBody(detectedBody, existingBody2D);
              }
              else
              {
                  CreateBody(detectedBody);
              }
          }

          onBody2DTrackingUpdate.Invoke(_bodies2D.ToList());
     }
}
```

Dafür wird die aktuelle Liste erkannter Körper um die Mindest-Erkennungssicherheit gefiltert. Falls ein Körper bzw. eine ID eines Körpers im aktualisierten Erkennungsframe nicht mehr vorkommt, wird das korrespondierende `Body2D`-Objekt aus der internen Liste entfernt. Für alle anderen wird festgestellt, ob ein bestehendes `Body2D`-Objekt existiert und dieses aktualisiert. Neu erkannte Körper werden als neue Objekte angelegt.

Diese beiden Funktionalitäten werden in die Methoden `CreateBody` sowie `UpdateBody` ausgelagert.

```
// BodyTrackingManager2D.cs
// ...

private void CreateBody(DetectedBody detectedBody)
{
    var newBody = new Body2D(detectedBody.rawBodyData.id,
       outputFormat2D,
        ApplyCalibrationToRawData(
            detectedBody.rawBodyData.keypoint),
        ApplyCalibrationToRawData(
            detectedBody.rawBodyData.keypoint, false),
        CalculateRootRelativeKeypoints(
            ApplyCalibrationToRawData(
                detectedBody.rawBodyData.keypoint,
                   false)));
    newBody.CurrentTrackingConfidence =
       detectedBody.confidence;

    _bodies2D.Add(newBody);
}

private void UpdateBody(DetectedBody detectedBody, Body2D
   body2D)
{
    body2D.CurrentTrackingConfidence =
       detectedBody.confidence;

    body2D.Update(
        ApplyCalibrationToRawData(
            detectedBody.rawBodyData.keypoint),
        ApplyCalibrationToRawData(
            detectedBody.rawBodyData.keypoint, false),
        CalculateRootRelativeKeypoints(
            ApplyCalibrationToRawData(
```

```
                detectedBody.rawBodyData.keypoint,
                    false)));
}
```

Die eigentliche Überführung bzw. Transformation der Schlüsselpunkte findet in der Methode `ApplyCalibrationToRawData` statt. Für alle übergebenen Schlüsselpunkte wird zunächst die Starrkörpertransformation durchgeführt. Dafür wird der jeweilige Punkt zuerst verschoben – der Ursprung des Koordinatensystems wird an die kalibrierte Position verschoben – und dann um den neuen Ursprung rotiert – entgegen der kalibrierten Rotation, sodass sich die Rotation der Referenzperson ausgleicht und diese in Richtung der Tiefenachse schaut. Anschließend wird die Tiefenachse der Punkte für die zweidimensionale Verwendung auf der Z-Achse um 0 skaliert (Orthogonalprojektion).

Sollen definierte Modifikatoren (Offsets) eingerechnet werden (siehe Unterunterabschnitt 3.2.1.3 [A10]), findet dies nach der Starrkörpertransformation statt. Da es sich um eine Transformation bzw. Projektion ins Zweidimensionale handelt und es sich deswegen bei den Modifikatoren um `Vector2`-Werte handelt, ist bei deren Nutzung als `Vector3` der Z-Wert standardmäßig 0. Daher kann der Skalierungsmodifikator die einfache Skalierung der Z-Achse auf 0 zur Auslöschung der Tiefe vollständig ersetzen.

```
// BodyTrackingManager2D.cs
// ...

public Vector2 positionOffset = new Vector2(0, 0);
public Vector2 scale = new Vector2(1, 1);

// ...

private Vector3[] ApplyCalibrationToRawData(
    IEnumerable<Vector3> rawData,
    bool calculateOffsets = true)
{
    var totalScale = calculateOffsets ? (Vector3)scale :
       new Vector3(1, 1, 0);
    var totalTranslation = calculateOffsets ?
       (Vector3)positionOffset : Vector3.zero;

    return rawData
        .AsParallel()
        .Select(point =>
            Vector3.Scale(
                Quaternion.Inverse(
                  _calibratedRotationOffset)
```

```
                    * (point - _calibratedPositionOffset),
                // since scale is set to 0 on Z it also
                    eliminates the depth
                totalScale
            ) + totalTranslation)
        .ToArray();
}
```

Die Modifikatoren ermöglichen eine Anpassung der Werte an die jeweilige Anwendung bzw. Interaktion (siehe Unterunterabschnitt 3.2.1.3 [A10]). Damit steht die Berechnungsgrundlage für zwei von drei gewünschten Schlüsselpunktvarianten.

Zur Bildung der Werte für relative Schlüsselpunkte müssen diese nach der Transformation mit der `CalculateRootRelativeKeypoints`-Methode umgerechnet werden. Diese berechnet übergebenen Vektoren relativ zum ersten der Liste und eignet sich so auch universell für alle Körperformate.

```
// BodyTrackingManager2D.cs
// ...

private static Vector3[]
    CalculateRootRelativeKeypoints(Vector3[]
    rawKeypoints2D)
{
     return rawKeypoints2D.ToList()
         .Select(relativeKeypoint => relativeKeypoint -
            rawKeypoints2D[0]).ToArray();
}
```

Wenn die Transformation für alle Körper abgeschlossen ist, wird die aktuelle interne Liste an `Body2D`-Objekten über das Event `onBody2DTrackingUpdate` verteilt und angebundene Komponenten können die aktualisierten Daten verarbeiten. Damit ist der Transformations- bzw. Projektionsvorgang beendet.

3.3 Verwendung der Komponente

Die fertiggestellte Komponente kann nun beliebig verwendet werden, um für weitere interaktive Komponenten in einer Szene mit zweidimensionalen Körperdaten zu arbeiten. Um die Verwendung zu beschreiben und gleichzeitig die Funktion zu prüfen, wird im Folgenden ein Beispiel definiert.

Als Basis dafür ist es nur notwendig, eine Szene mit einem Kamera-Prefab (und damit einer `ZEDManager`-Komponente) und der implementierten Komponente `BodyTrackingManager2D` zu erstellen.

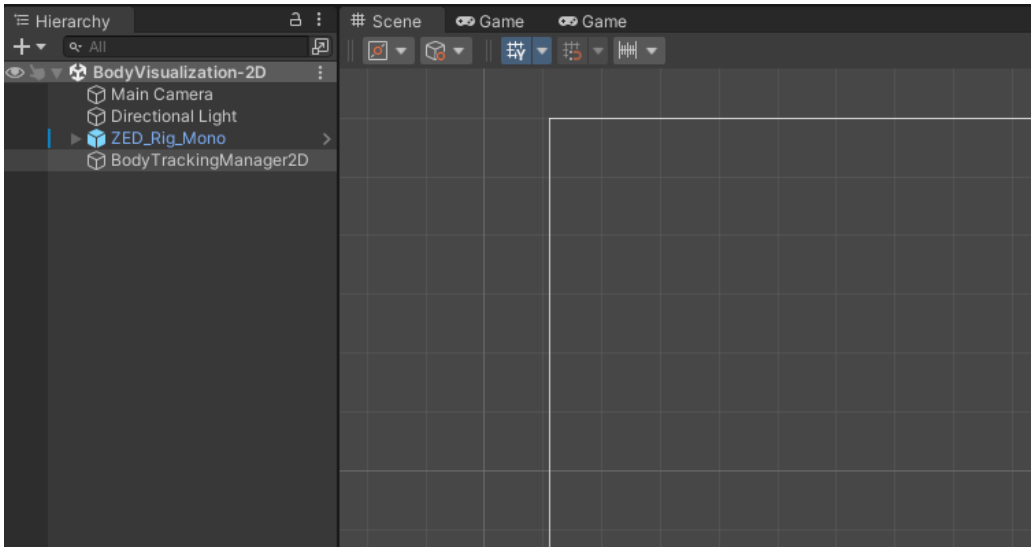


Abbildung 3.13: Vorbereitete Szene für die Verwendung des `BodyTrackingManager2D`

Die `BodyTrackingManager2D`-Komponente kann nun im Inspektor vorbereitet werden. Dort kann das Kamera-Prefab-Objekt als Referenz für die `ZEDManager`-Komponente hinterlegt werden und die weitere Konfiguration vorgenommen werden.

Standardmäßig startet die Komponente das Tracking und die Kalibrierung beim Start der Szene automatisch. Alternativ kann die Komponente auch programmatisch (über die Methoden `StartTracking` bzw. `StartCalibration`) gesteuert werden. Für die Erkennung bzw. die Erkennungsdaten können das Körperformat und eine Option zur Verbesserung der Erkennung bei fehlenden Daten eingestellt werden. Die erkannten Körper können zum einen durch eine Mindest-Erkennungssicherheit, zum anderen durch den Erkennungszustand im ZED-System gefiltert werden. So gilt eine Person, die aktuell erkannt wird, als *On* und eine Person, die kürzlich nicht mehr erkannt werden konnte, als *Searching*. Die Kalibrierungscache-Option wird meist zur Laufzeit verwendet, um den aktuell vorhandenen Cache darzustellen. Mit der Kalibrierungszeit kann für den automa-

tischen Start eingestellt werden, wie lange nach dem Start des Kalibrierungsvorgangs gewartet werden soll, bis die Orientierung und Position der Referenzperson als Kalibrierungswerte festgesetzt werden. So hat die Person die entsprechende Zeit zur Verfügung, sich optimal zu platzieren.

Die bereitgestellten Events können nun verwendet werden, um eine eigene interaktive Komponente anzubinden.

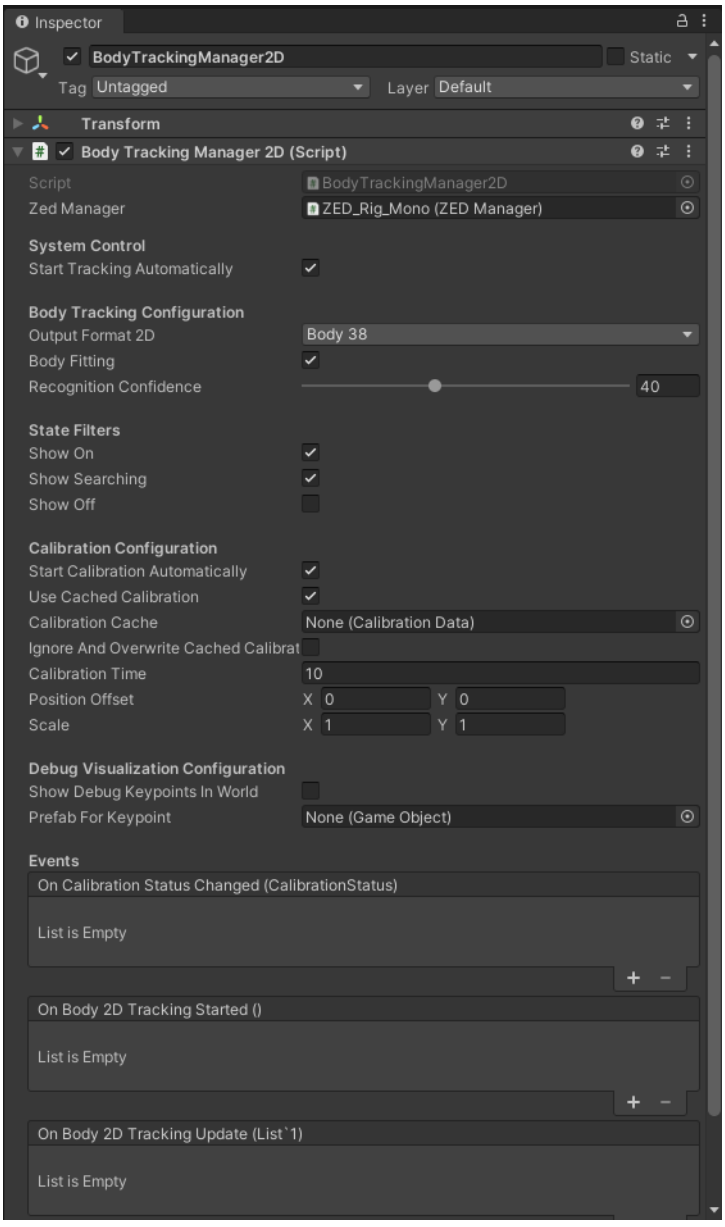
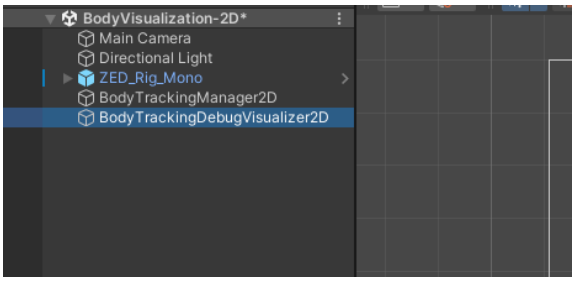


Abbildung 3.14: Inspektor-Ansicht des `BodyTrackingManager2D`

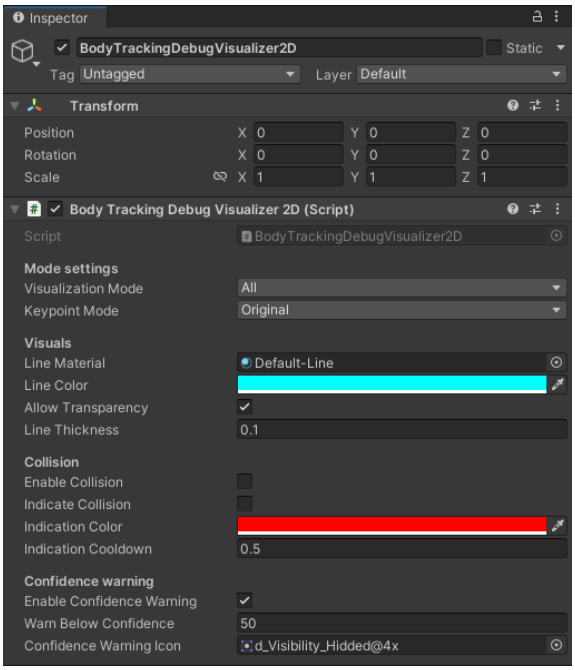
3.3.1 LineRenderer-basierter Visualizer

Zur Überprüfung der Funktion eignet sich die Implementierung einer interaktiven Komponente, welche die überführten Personen darstellen kann. Dafür wird die Komponente `BodyTrackingDebugVisualizer2D` entwickelt, welche folgende Funktionen erfüllt:

- Verwendung und Darstellung von empfangenen Listen mit `Body2D`-Objekten in Form von einer schematischen Darstellung (Strichmännchen)
- Visualisierung der verschiedenen Schlüsselpunktvarianten
- Rudimentäre Kollision durch einen selbstdefinierten `LineCollider2D` mit visueller Darstellung der Kollision



(a) Objekt in der Szenenhierarchie



(b) Inspektor-Ansicht

Abbildung 3.15: `BodyTrackingDebugVisualizer2D`-Objekt

Die entwickelte Komponente stellt also die empfangenen Daten je nach Körperformat als zusammenhängende Linien dar. Auch ist bereits eine rudimentäre Interaktion möglich, bei Aktivierung der Kollision. Die Anbindung erfolgt dabei, indem in der `BodyTrackingDebugVisualizer2D`-Komponente eine Methode definiert wird, welche als Event-Handler zum kontinuierlichen Empfangen von einer Liste aus `Body2D`-Objekten dient.

```
// BodyTrackingDebugVisualizer2D.cs
// ...

public void VisualizeBody2D(List<Body2D> list)
{
    // Compute Body2D-Data here
}

// ...
```

Es kann dadurch über den Inspektor oder programmatisch als Event der Komponente `BodyTrackingManager2D` hinzugefügt werden und ist damit an den Datenfluss (siehe Unterunterabschnitt 3.2.2.1) als interaktive Komponente angebunden.

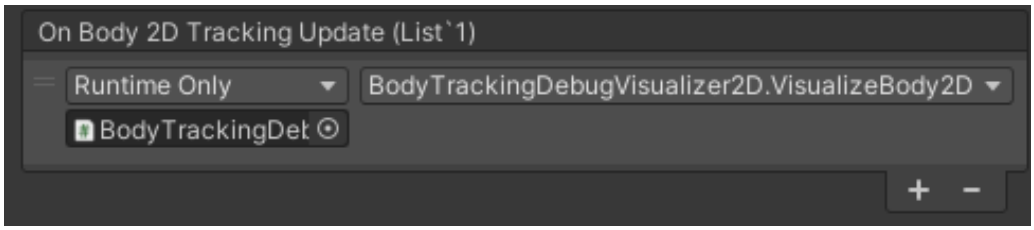


Abbildung 3.16: Update-Event im `BodyTrackingManager2D`

3.3.2 Anwendungsbeispiel der 3D-2D-Transformation

Die Szene ist nun für eine Demonstration vorbereitet und kann zum Zeigen des Transformationsprozesses bzw. der Überführung verwendet werden. Nachdem die Szene gestartet wurde und das ZED-System bereit ist, beginnt die Erkennung bzw. der Kalibrierungsprozess.

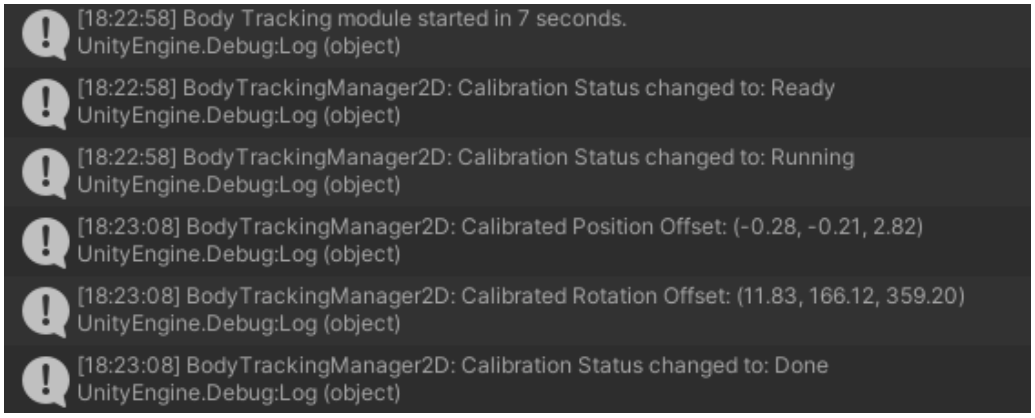


Abbildung 3.17: Status der Kalibrierung im `BodyTrackingManager2D`

Nachdem der Kalibrierungsprozess beendet ist, beginnt die Komponente mit der Übermittlung von den Erkennungsdaten per Event an angebundene interaktive Komponenten.

Zur besseren Visualisierung wird der Transformations- bzw. Überführungsvorgang ausgehend von den dreidimensionalen Daten des ZED-Systems gezeigt. Das Kamerabild gilt als Ausgangslage, welches vom ZED-System verarbeitet wird. Es wurden die dreidimensionalen rohen Körperdaten als Skelett überlagert und die zentralen Orientierung (rot) kenntlich gemacht. Die Person ist dabei frontal auf einem Monitor ausgerichtet (nicht im Bild zu sehen) und wurde als Referenzperson zur Kalibrierung eingesetzt. Man sieht die Abweichung der Orientierung zum Kamerabild des ZED-Systems.

Abbildung 3.18: Kamerabild mit 3D-Skelett aus Rohdaten überlagert

In der Szenenansicht wurde der Ursprung kenntlich gemacht (Visualisierung der aufspannenden Vektoren des Koordinantensystems im Nullpunkt). Hier die abweichende Verschiebung und Orientierung deutlich.

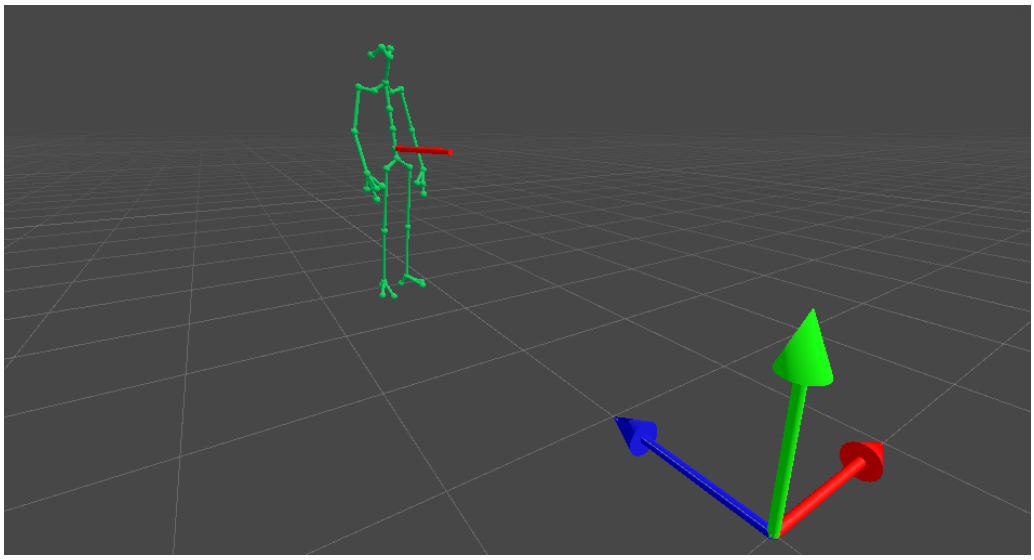
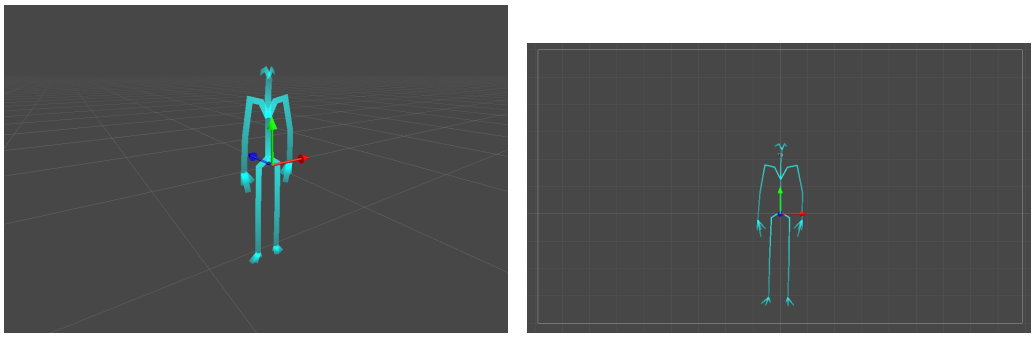


Abbildung 3.19: 3D-Skelett aus Rohdaten in der Szene mit Weltursprung

Jetzt wird ein durch die `BodyTrackingDebugVisualizer2D`-Komponente dargestellte überführte Ansicht gezeigt. Es handelt sich um die selbe Pose, welche durch die `BodyTrackingManager2D`-Komponente als neues Zentrum kalibriert wurde. Es ist erkennbar, dass sich die Person nun im Ursprung befindet und frontal zur Tiefenachse ausgerichtet ist. Alle Bewegungen bzw. Orientierungsänderungen relativ zum Kalibrierungspunkt in der Realwelt, finden damit relativ zum Nullpunkt bzw. der frontalen Ausrichtung statt.



(a) Perspektivisch (b) 2D

Abbildung 3.20: Überführtes 2D-Skelett mit Weltursprung

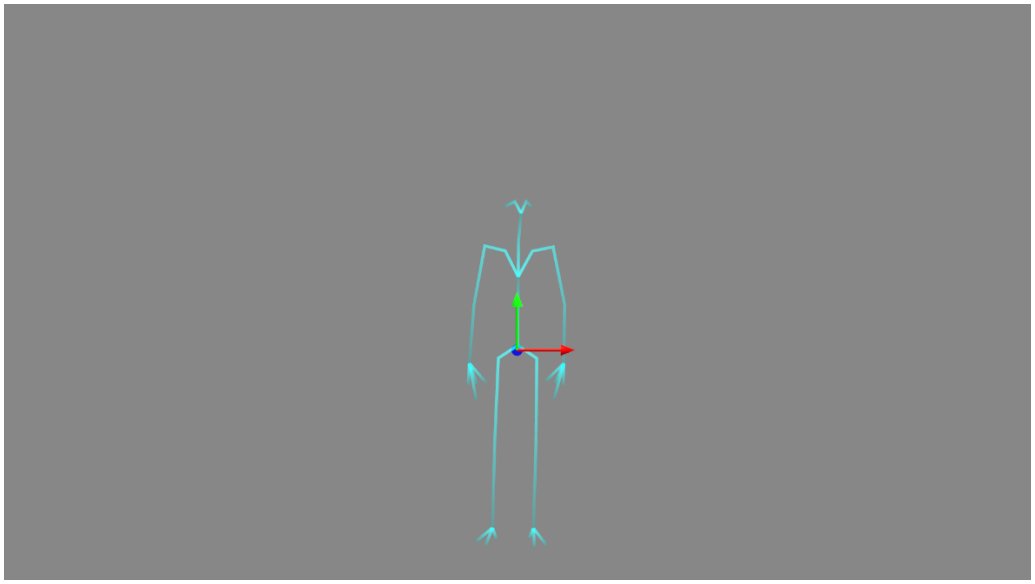
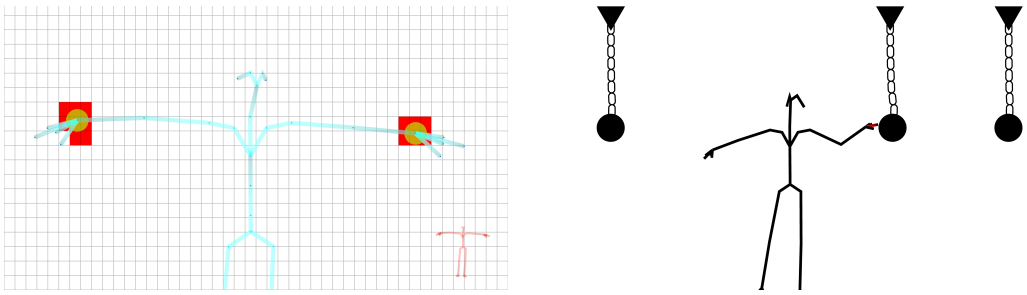


Abbildung 3.21: Überführtes 2D-Skelett mit Weltursprung (Playmode)

Die Position eines jeden Schlüsselpunkts bzw. jede Person kann nun als Datenquelle für eine Interaktion mit zweidimensionalen Anwendungen in Unity verwendet werden. Dabei können abhängig von der gewünschten Interaktion entweder alle Schlüsselpunkte pro Person oder nur eine Auswahl verwendet werden.

Auf Basis dieser entstandenen Szene werden weitere Beispiele implementiert, um die Funktionalität der Komponente nachzuvollziehen.



(a) Body2D-Wall (b) Body2D-BallHingeInteraction

Abbildung 3.22: Beispielszenen auf Basis `BodyTrackingDebugVisualizer2D`

Bei Abbildung 3.22a handelt es sich um eine von der Handposition abhängige Interaktion mit dem Hintergrund; bei Abbildung 3.22b handelt es sich um eine kollisionsbasierte Interaktion mit hängenden Kugeln.

4. Interaktionsprototyp »Vektorfelder«

Nachdem die Grundlage für die Interaktion als Unity-Komponente erschaffen ist, wird im Folgenden die Konzeption und Implementierung eines Interaktionsprototyps auf Basis eben dieser Komponente betrachtet. Dieser Prototyp soll ein exemplarisches Interaktionsbeispiel auf Basis von Vektorfeldern und Partikelsystemen sein, welches durch die positionsbasierte Interaktion durch die Hände des Benutzers verändert werden kann. Dafür folgt zunächst eine Einführung in Vektorfelder und ihre Anwendungsmöglichkeiten für diese Arbeit. Anschließend werden die Ziele des Interaktionsprototyps, die dafür notwendigen Schritte sowie die Integration in Unity-Partikelsysteme betrachtet und Beispiele gezeigt.

4.1 Einführung in Vektorfelder und ihre Anwendungsmöglichkeiten

Mathematisch kann ein Vektorfeld (sofern es sich um ein gleichdimensionales Vektorfeld handelt) als Funktion $\overrightarrow{F} : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ betrachtet werden. Das bedeutet, dass das Vektorfeld $\overrightarrow{F}$ Vektoren aus dem Definitionsbereich D in den n-dimensionalen euklidischen Raum R^n abbildet.[18] Kurzgesagt, jedem Punkt eines n-dimensionalen Raumes ist ein n-dimensionaler Vektor zugeordnet. Wenn der zugrundeliegende Raum der Definition ein zweidimensionaler Raum ($\mathbb{R}^2$) ist und das Vektorfeld jedem Punkt in diesem Raum einen Vektor im zweidimensionalen Raum zuordnet, spricht man daher von einem Vektorfeld im zweidimensionalen Raum.

Es ist möglich, für Punkte auf dem Vektorfeld zu berechnen, wie sich diese im Raum im Laufe der Zeit unter dem Einfluss des Vektorfeldes bewegen. Formal beschrieben, sei $\overrightarrow{F}$ ein Vektorfeld und p ein Punkt in diesem Raum. Der Fluss des Vektorfeldes $\overrightarrow{F}$ zum Zeitpunkt t ist eine Funktion $\phi(t, p)$, die den Punkt angibt, zu dem sich p unter dem Einfluss des Vektorfeldes $\overrightarrow{F}$ zum Zeitpunkt t entwickelt. Damit ist es möglich, die Bewegung von Teilchen oder Punkten im Raum zu beschreiben, die vom gegebenen Vektorfeld beeinflusst werden.

[18] vgl. Neher, »Vektorfelder«.

Vektorfelder können grafisch durch Pfeile oder ähnliches visualisiert werden. Die Länge und Richtung des Pfeils repräsentieren die Größe und Richtung des Vektors an diskreten Punkten im Feld.

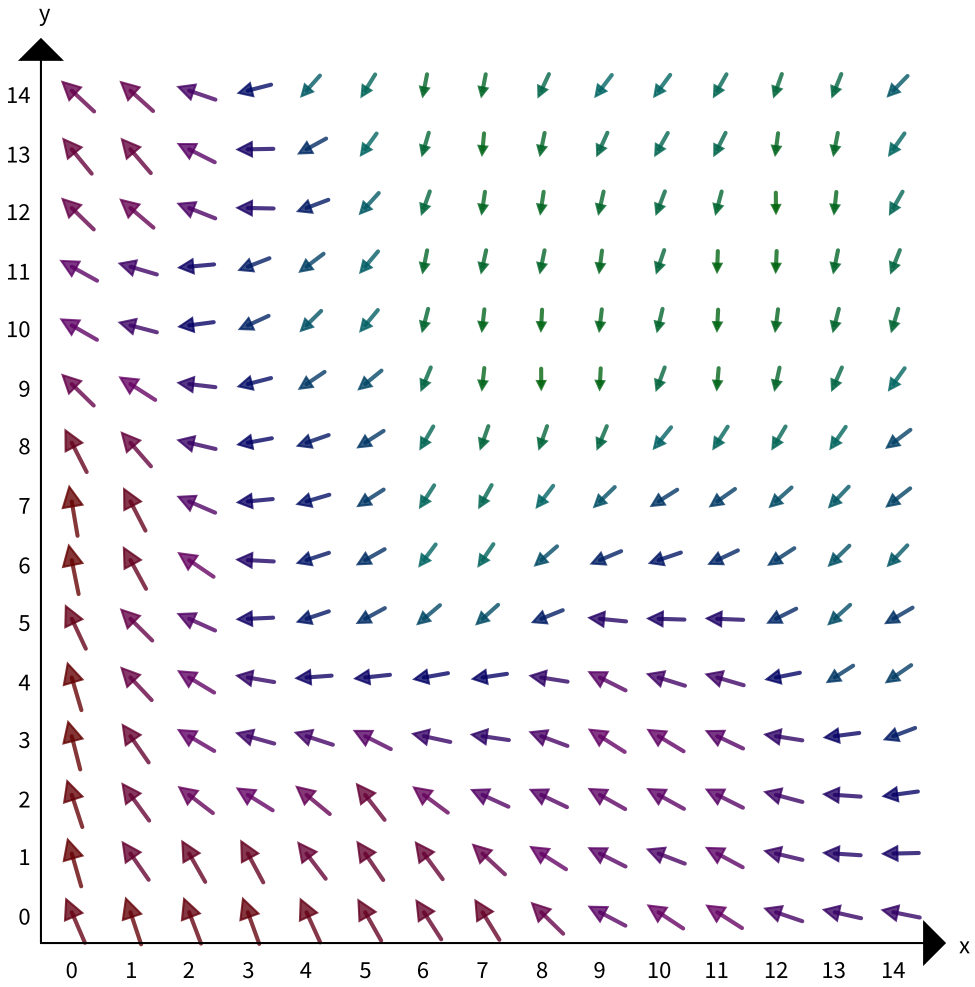


Abbildung 4.1: Vektorfeld im zweidimensionalen Raum

Solche Vektorfelder werden in vielen wissenschaftlichen und technischen Bereichen verwendet. Beispielweise in der Physik, um die Geschwindigkeit und Richtung von Kräften (zum Beispiel Magnetismus) zu beschreiben. In der Strömungsmechanik werden Vektorfelder verwendet, um die Strömung von Flüssigkeiten oder Gasen zu definieren. Auch in der Meteorologie werden Vektorfelder verwendet, um beispielsweise Windströmungen zu analysieren bzw. visualisieren.[19]

[19] s. »Vektorfeld«.

Vordefinierte Vektorfelder werden in der Computergrafik bzw. insbesondere in der Spieleentwicklung verwendet, um beispielsweise Partikel bestimmten Formen folgen zu lassen bzw. diese Formen gezielt zu umströmen (siehe Abbildung 4.2[20]).

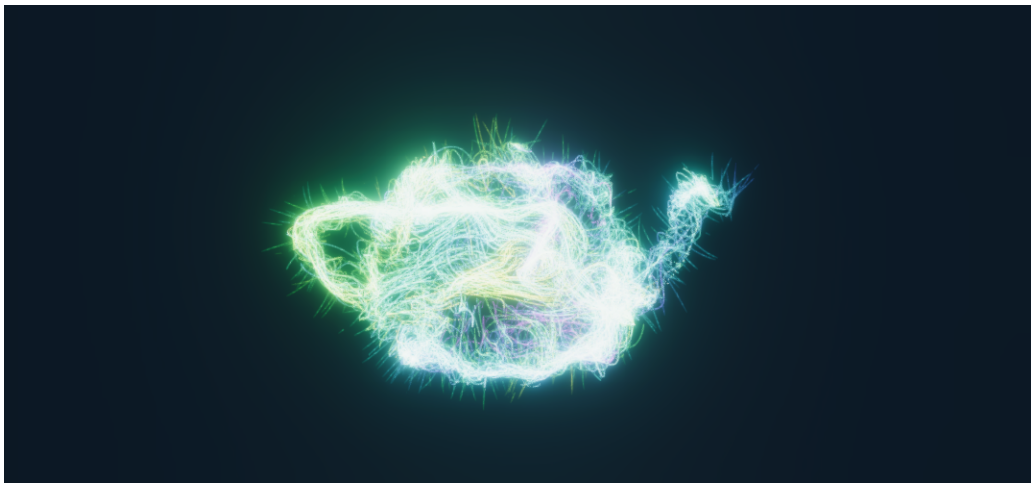


Abbildung 4.2: Utah-Teekanne als Grundlage für einen Partikelstrom

4.2 Anforderungen an den Prototypen

Für den Interaktionsprototypen soll so ein zweidimensionales dynamisch anpassbares Vektorfeld als Grundlage implementiert werden, um die Richtung und Geschwindigkeit von Partikeln in Unity zu beeinflussen. Die dabei wirkenden Kräfte werden durch dieses Vektorfeld definiert und sollen auf die Partikel je nach ihrer korrespondierenden Position auf dem Feld angewendet werden.

Für die mögliche Implementierung eines dynamischen Vektorfeldes in Unity wird eine angepasste Form der Vektorfeld-Definition verwendet. Zur besseren Handhabung innerhalb von Unity handelt es sich dabei um ein diskretes und begrenztes Vektorfeld. Statt eine Funktion zu berechnen und in Echtzeit zu aktualisieren, werden die Vektorwerte bereits im Voraus berechnet und in Form eines äquidistanten Voxelgitters (bzw. Pixelgitters im zweidimensionalen Raum) gespeichert. Dies erleichtert unter anderem die Verwaltung, Manipulation und Anzeige des Vektorfeldes in Unity.

Das Vektorfeld soll später als Basis dafür genutzt werden, Partikel aus einem Partikelsystem zu beeinflussen. Dabei ist ein Ziel, ein in Unity integriertes Partikelsystem dafür nutzen zu können. Dies bietet mehrere Vorteile: Die integrierten Partikelsysteme von Unity wurden speziell für die Unity-Engine optimiert. Dadurch können Partikeleffekte mit einer hohen Anzahl von Partikeln performant gerendert werden, ohne die Leistung stark zu beeinträchtigen. Des Weiteren ist

[20] s. Torrence, »Martin Newell's original teapot«.

bereits ein breites Spektrum an Anpassungs- bzw. Konfigurationsmöglichkeiten vorhanden, um verschiedenste Effekte zu erzielen.

In Unity gibt es mit Stand dieser Arbeit zwei Hauptpartikelsysteme. Das »Shuriken Particle System« sowie den »Visual Effects Graph (VFX-Graph)«. Es sind verschiedene Ansätze denkbar, wie auf die von den Systemen verwalteten Partikel eingewirkt werden kann. Ein Beispiel wäre die direkte manuelle Manipulation der Position bzw. Geschwindigkeit der Partikel. Beide Systeme sind aber auch grundlegend in der Lage, die von ihnen verwalteten Partikel durch externe Kräfte effizient selbst zu beeinflussen. Dafür können dreidimensionale Texturen verwendet werden, welche auf Basis des datenbasierten Vektorfeldes generiert werden müssen. Die Partikel des jeweiligen Systems sollen dann den entsprechend definierten Kräften bzw. Richtungen folgen.

Zur Demonstration der `BodyTrackingManager2D`-Komponente soll eine Interaktion zur Laufzeit mit dem Vektorfeld möglich sein. Dafür soll das Feld mit einem kreisförmigen Manipulator interaktiv beeinflusst werden. Der zu manipulierende Bereich soll dabei durch den Radius des Manipulationskreises kontrolliert werden. Sofern dieser Manipulator mit der Position von Pixeln bzw. Voxeln des Vektorfeldes in der Unity-Welt überlappt, sollen diese bzw. deren Vektoren innerhalb des Manipulationsfeldes manipuliert werden. Dabei soll es verschiedene Arten der Manipulation geben.

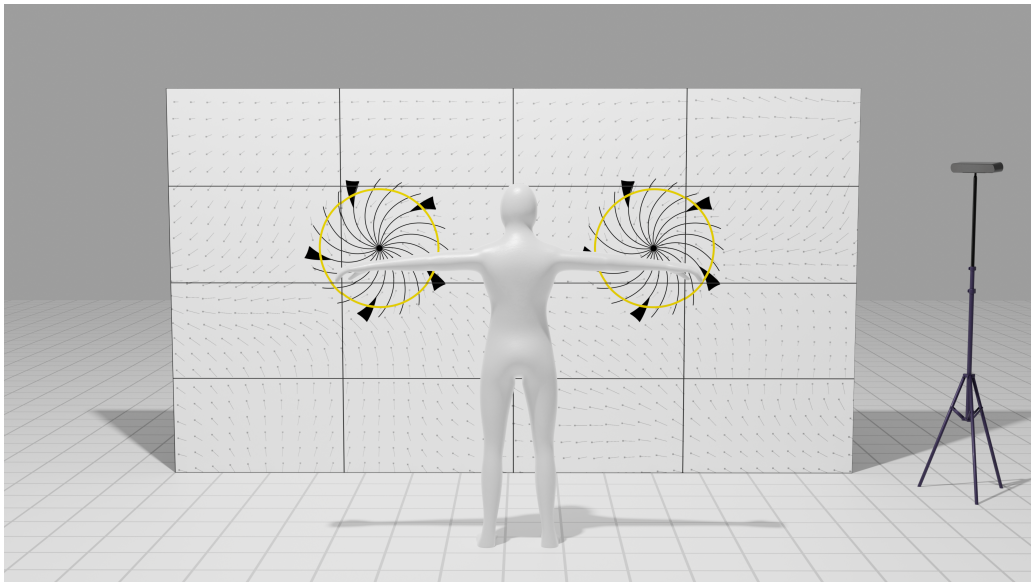


Abbildung 4.3: Konzeptvisualisierung der Manipulation

Für das eigentliche prototypische Anwendungsbeispiel, basierend auf der zuvor erstellen `BodyTrackingManager2D`-Komponente, soll die Position des Vektorfeld-Manipulators auf dem Vektorfeld mit den Händen der interagierenden Person

relativ zum Bildschirm kontrolliert werden und dadurch das Verhalten der Partikel bzw. des Partikelflusses relativ zur Handposition anpassen.

4.3 Implementierung von Vektorfeldern in Unity

Auf Basis der genannten Anforderung werden zwei Komponenten entwickelt: Die `VectorField`-Komponente, welche in der Lage ist, Vektorinformationen zu generieren, in einem dreidimensionalen Gitter zu speichern, zu manipulieren, darzustellen und in einem verwendbaren Format (`Texture3D`) für die Beeinflussung eines Partikelsystems bereitzustellen oder auf ein solches anzuwenden. Es soll möglich sein, verschiedene Parameter des Vektorfelds über den Inspektor zu konfigurieren. Dazu gehören die Breite, Höhe und optional Tiefe des Vektorfelds und die Art des Vektorfelds, die den Inhalt des Vektorfeldes bei der Initialisierung konfiguriert (leer oder auf Basis von Rauschen bzw. einer Noise-Funktion). Für die Initialisierung auf Basis einer Noise-Funktion sollen weitere Parameter wie die Skalierung und Verschiebung des Rauschens und dessen Animation konfigurierbar sein. Die Implementierung erfordert für den geplanten Interaktionsprototypen prinzipiell nur ein zweidimensionales Vektorfeld, allerdings ist es für die weitere Implementierung günstig, Vektorfelder dreidimensional zu betrachten und zu implementieren.

Im Weiteren soll die Komponente eine Funktionalität für die Einwirkung der genannten kreisförmigen Manipulatoren bieten, welche das Vektorfeld abhängig von ihrer Größe und Position in der Unity-Welt manipuliert. Darüber hinaus soll die Komponente so implementiert werden, dass das Feld optional langsam in seine ursprüngliche Form zurückkehrt, wenn der Manipulator verschoben oder entfernt wird.

Die notwendige andere Komponente `VectorFieldManipulator` soll ein kreisförmiges Manipulator-Objekt repräsentieren, welches auf ein bestimmtes Vektorfeld bzw. `VectorField`-Objekt einwirkt. Der Manipulationsradius und Modus sollen konfigurierbar sein.

4.3.1 Grundlegendes Datenmodell

Das Datenmodell für die `VectorField`-Komponente besteht aus den Eigenschaften und Variablen für die Konfiguration und aus den internen Variablen des Vektorfeldes.

Die Grundkonfiguration stellt dabei die Definition der Feldgröße und der daraus resultierenden festen Anzahl an Voxeln dar.

```
// VectorField.cs
// ...

[Header("Vector field parameter")]
[Tooltip("The width of the VectorField voxels.")]
```

```
public uint width = 44;
[Tooltip("The height of the VectorField voxels.")]
public uint height = 25;
[Tooltip("The depth of the VectorField voxels. Default is
    1 for 2D.")]
public uint depth = 1;

// ...
```

Zur Verwaltung der Daten eines zweidimensionalen bzw. dreidimensionalen diskreten Vektorfeldes wird ein Datenmodell benötigt, welches eine einfache Handhabung und Manipulation vom Vektorfeld ermöglicht. Dafür wird zur Repräsentation der Vektoren eines Feldes ein mehrdimensionales `Vector3`-Array verwendet. Der Definitionsbereich (also die Punkte) werden dabei durch dessen Indizes repräsentiert, während die Werte in Form der enthaltenen `Vector3`-Strukturen im Array abgebildet werden. Die Indizierung entspricht dabei der konventionellen Reihenfolge eines dreidimensionalen Vektortupels (x, y, z).

Für die Implementierung des gewünschten Verhaltens werden mehrere solcher Array-Strukturen verwendet:

```
// VectorField.cs
// ...

// Three dimensional array for current vector field values
private Vector3[,,] _vectorField;
// Shadow vector field for lerping back
private Vector3[,,] _baseVectorField;
// Accumulated manipulation of all manipulators
private Vector3[,,] _manipulationField;
// Accumulated influence of manipulation fields
private float[,,] _influenceField;

// ...
```

Diese haben die folgenden Funktionen: ein Array für den jeweils aktuellen Zustand des Vektorfeldes (`_vectorField`), eine Kopie des unmanipulierten Zustands des Vektorfeldes (`_baseVectorField`) als Basisvektorfeld, um den früheren Zustand nach einer Manipulation wiederherstellen zu können, die Kumulation aller Manipulationen (`_manipulationField`), welche auf das Basisvektorfeld angewendet werden, sowie die Kumulation aller Einflussfelder (`_influenceField`), welche später dafür verwendet werden, um gewichtet die Manipulationen auf das Basisvektorfeld anwenden zu können. Diese Datenstrukturen bilden die Basis des dynamischen Vektorfeldes.

4.3.2 Erzeugung des Basisvektorfeldes mit »Perlin Noise« bzw. »Simplex Noise«

Das Vektorfeld wird initial mit Werten besetzt, um eine Grundlage für die spätere Bewegung der Partikel bzw. Manipulation zu bieten.

Diese Initialisierung des Vektorfeldes ist entweder ein einfaches leeres Vektorfeld (alle Vektoren sind Nullvektoren) oder das Feld wird auf Basis von Rauschfunktionen generiert.

Als Basis eigenen sich Rauschfunktionen wie »Perlin Noise« und »Simplex Noise«. Dabei erzeugen beide eine kontinuierliche, unregelmäßig wellenförmige Funktion. Die Funktion gibt für jeden Punkt im Raum einen Zufallswert zurück, der von den benachbarten Punkten abhängt. Dies ermöglicht die Erzeugung von kohärenten, organisch aussehenden Strukturen und Texturen.

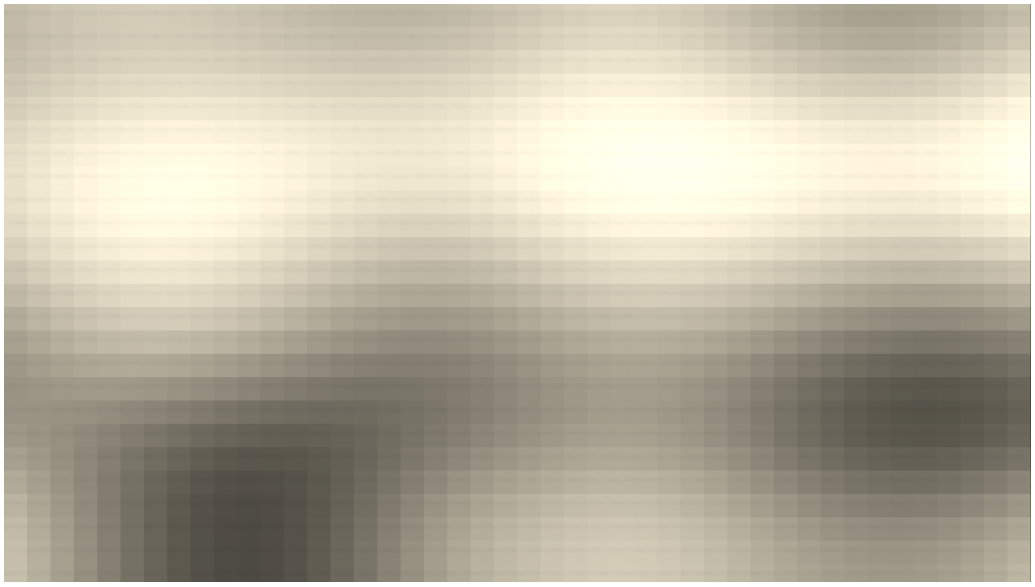


Abbildung 4.4: Beispiel: Diskrete Abtastung »Perlin Noise«

Für die Initialisierung des Vektorfeldes eignet sich die Anwendung solcher Funktionen, da auf deren Basis ein initiales Vektorfeld erzeugt werden kann, welches sanfte Übergänge über das Voxelgitter hinweg aufweist. Durch die Variation der Intensität und Werte des Rauschens können verschiedene Ergebnisse erzielt werden. Dies ermöglicht eine flexible und kontrollierte Gestaltung des Partikelflusses, der in späteren Schritten auf das Vektorfeld angewendet werden kann.

Dies wurde wie folgt implementiert:

```
private static Vector3[,,]
    GenerateNoiseBaseVectorFieldData(uint width, uint
    height, uint depth, float noiseScale, Vector3
    noiseOffset)
```

```
{
    var vectorField = new Vector3[width, height, depth];

    for (var x = 0; x < width; x++)
    {
        for (var y = 0; y < height; y++)
        {
            for (var z = 0; z < depth; z++)
            {
                var noiseX = (x + noiseOffset.x) / width
                    * noiseScale;
                var noiseY = (y + noiseOffset.y) / height
                    * noiseScale;
                var noiseZ = (z + noiseOffset.z) / depth
                    * noiseScale;

                var noiseValue = noise.cnoise(new
                    float3(noiseX, noiseY, noiseZ)) + 1;


                var vector = new
                    Vector3(Mathf.Cos(noiseValue *
                    Mathf.PI), Mathf.Sin(noiseValue *
                    Mathf.PI),
                     depth == 1 ? 0 : Mathf.Cos(noiseValue
                        * Mathf.PI));

                var noiseForLength = (noise.snoise(new
                    float3(noiseX, noiseY, noiseZ)) + 1) /
                    2;

                vector = vector.normalized *
                    noiseForLength;

                vectorField[x, y, z] = vector;
            }
        }
    }

    return vectorField;
}
```

Für jeden Voxel des Vektorfeldes wird ein Basiswert pro Komponente als Parameter der Rauschfunktion ermittelt. Dieser ist abhängig von der Gitterposition

des aktuellen Voxels und kann durch Parameter modifiziert bzw. variiert (Verschiebung und Skalierung des Rauschens) sowie animiert werden. Die Richtung des Vektors wird dabei mit dem Ergebnis der Rauschfunktion in Verbindung mit trigonometrischen Funktionen berechnet. Im Falle eines zweidimensionalen Vektorfeldes (wie für diesen Prototyp notwendig) wird die Z-Achse immer mit 0 besetzt, um ausschließlich zweidimensionale Richtungen im Vektorfeld zu erhalten. Dies ist besonders wichtig, damit der später folgende Partikelfluss für den geplanten zweidimensionalen Anwendungsfall korrekt funktioniert, und auf die Partikel nur Kräfte auf der zweidimensionalen Ebene angewendet werden und diese folglich darauf beschränkt bleiben.

Die Länge des Vektors wird separat mit einer weiteren Rauschfunktion bestimmt. Dadurch wird unter anderem erreicht, dass die Länge des Vektors unabhängig von seiner Ausrichtung ist.

4.3.2.1 Visualisierung des Vektorfeldes

Die Datengrundlage des Vektorfeldes ist bereit und kann nun erstmals in Unity visualisiert werden. Dafür können Gizmos verwendet werden, welche jeden Vektor im Gitter sowie das Gitter selbst visualisieren.

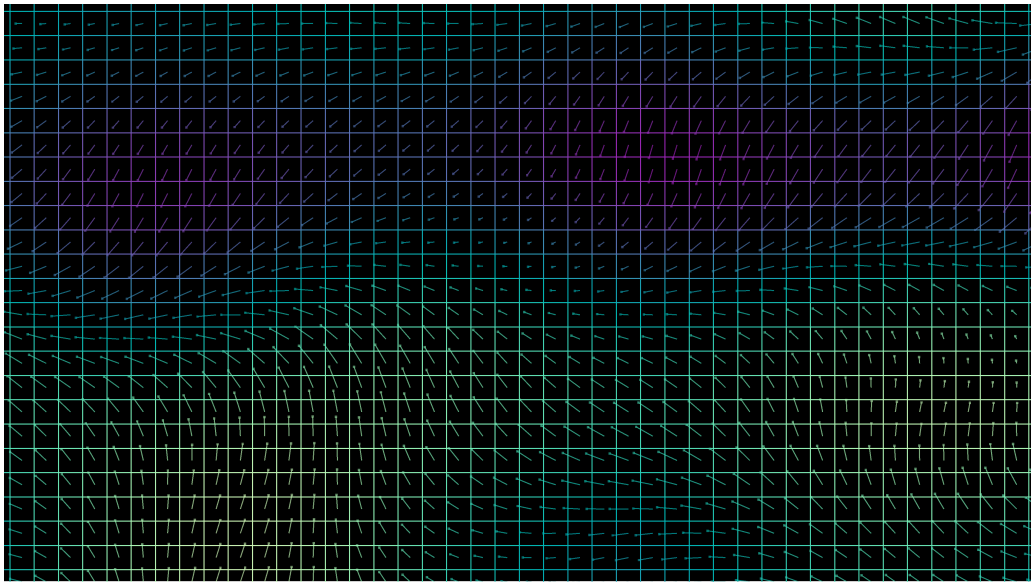


Abbildung 4.5: Visualisierung eines auf der Funktion basierenden Vektorfeldes mit Gizmos in Unity

In der Abbildung 4.5 ist die Ausrichtung der Vektoren sowie ihre Länge pro Voxel bzw. Pixel (Gitterpunkt) erkennbar. Das Ergebnis ist ein Vektorfeld mit kohärenten bzw. sanften Übergängen, sowohl in der Orientierung als auch der Einflusskraft (Länge der Vektoren).

4.3.3 Entwicklung des kreisförmigen Manipulators

Als Vorleistung für die spätere Interaktion ist ein Manipulationswerkzeug notwendig, welches das Vektorfeld in Abhängigkeit von seiner Position manipuliert. Dies wird in diesen Interaktionsprototypen durch einen kreisförmigen Manipulator realisiert. Dabei soll der Manipulator, wie bereits beschrieben, die Vektoren des Vektorfeldes beeinflussen, die mit der Position (in der Unity-Welt) des Manipulators korrespondieren bzw. innerhalb seines Einflussradius liegen.

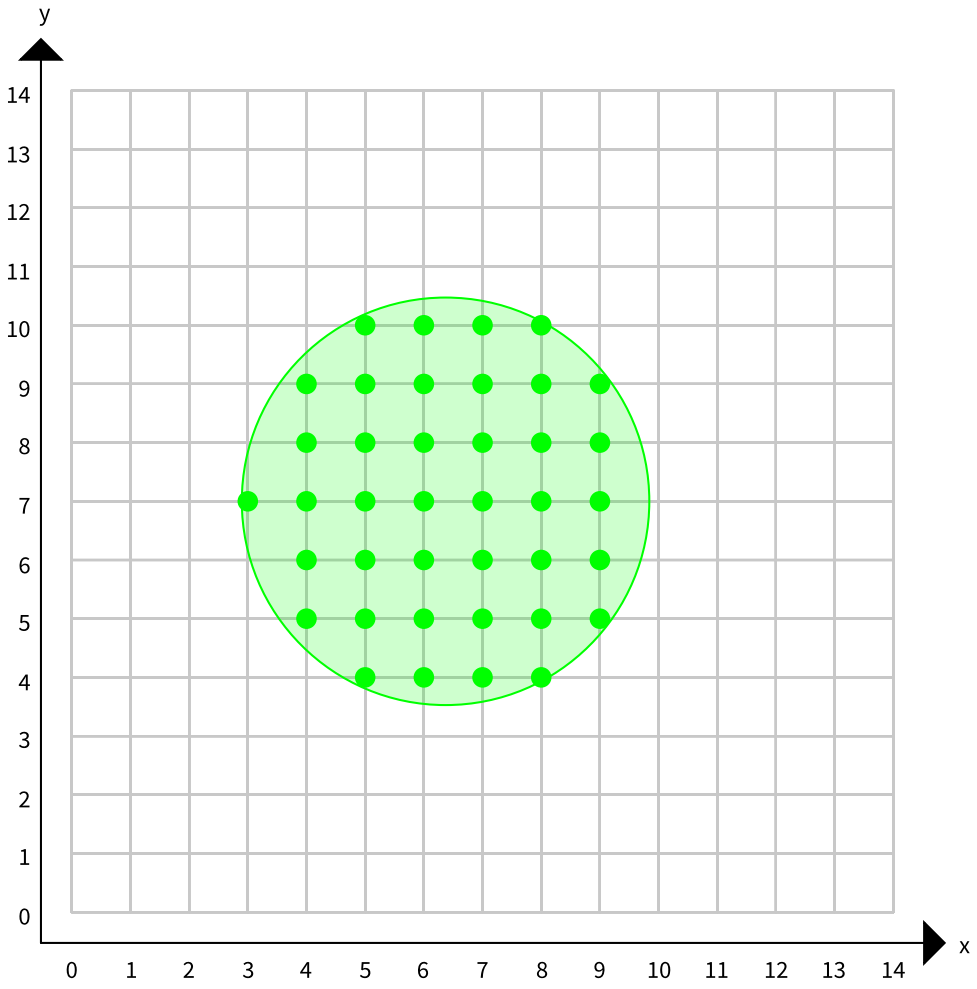
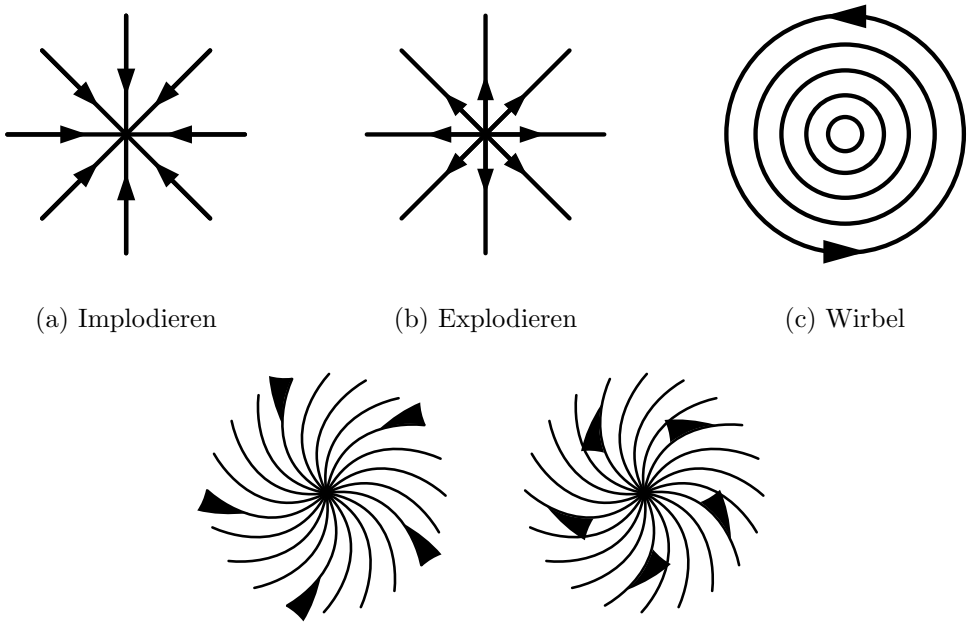


Abbildung 4.6: Schema der beeinflussten Voxel durch den Manipulator

Die Vektoren im Manipulationsradius sollen durch verschiedene Modi manipuliert werden können und dadurch wahrnehmbare kritische Punkte im Vektorfeld erzeugen.



(a) Implodieren (b) Explodieren (c) Wirbel

(d) Anziehender Wirbel (e) Abstoßender Wirbel

Abbildung 4.7: Schematische Darstellung der Manipulationsmodi

- Implodieren (`Implode`): Im Manipulationsradius werden die Vektoren in Richtung des Zentrums ausgerichtet. Die Partikel werden folglich eingezogen.

- Explodieren (`Explode`): Im Manipulationsradius werden die Vektoren ausgehend vom Zentrum nach außen zeigend ausgerichtet. Die (umströmenden) Partikel werden folglich abgestoßen.

- Wirbel (`Vortex`): Vektoren werden ringförmig als Kreisbahn und das Zentrum des Manipulators ausgerichtet und stellen damit eine Zirkulation um das Zentrum dar.

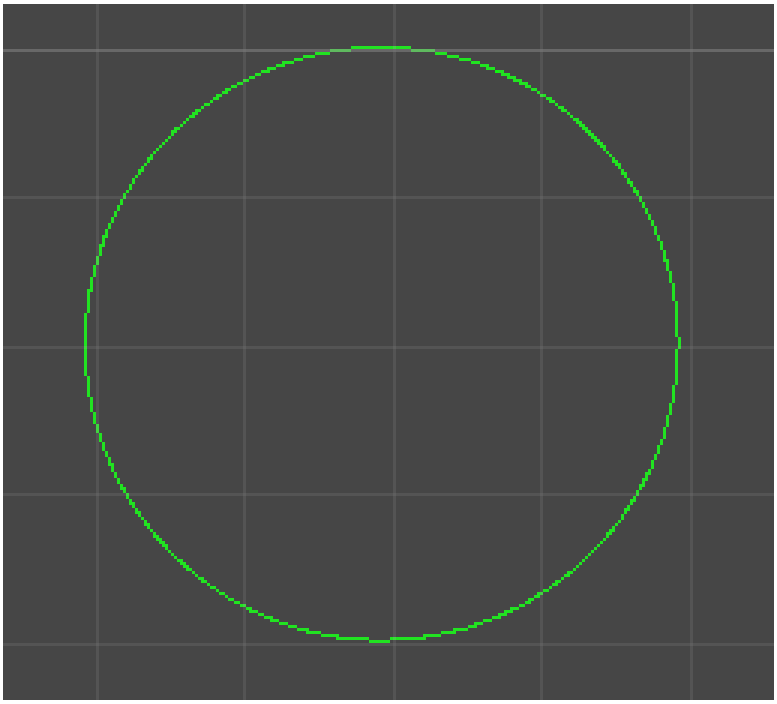
- Anziehender Wirbel (`AttractingVortex`): Vektoren werden so ausgerichtet, dass ein spiralförmiger Fluss ins Zentrum (den Fokuspunkt) des Manipulators entsteht.

- Abstoßender Wirbel (`RepellingVortex`): Vektoren werden so ausgerichtet, dass ein spiralförmiger Fluss vom Zentrum des Manipulators nach außen entsteht.

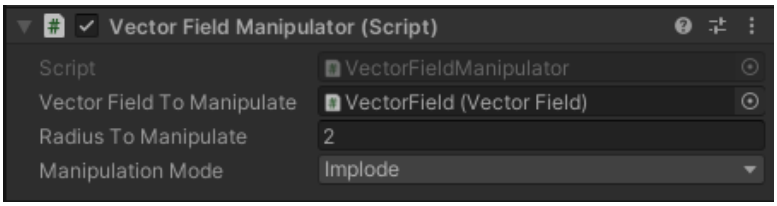
4.3.3.1 Datenmodell des Manipulators

Zur Repräsentation eines Manipulators in Unity wird eine einfache Komponente (`VectorFieldManipulator`) implementiert, welche einen Radius definiert, sowie einen der von der `VectorField`-Komponente bereitgestellten Modus annimmt. Die Verarbeitung der Manipulation wird vollständig durch die `VectorField`-Komponente durchgeführt.

```
public class VectorFieldManipulator : MonoBehaviour
{
    public VectorField vectorFieldToManipulate;
    public float radiusToManipulate = 1;
    public ManipulationMode manipulationMode =
       ManipulationMode.Explode;

    private void Start()
    {
        vectorFieldToManipulate.RegisterManipulator(this);
    }

    private void OnDestroy()
    {
        vectorFieldToManipulate.RemoveManipulator(this);
    }

    private void OnDrawGizmosSelected()
    {
        #if UNITY_EDITOR
        Handles.color = Color.green;
        Handles.DrawWireDisc(transform.position,
           Vector3.back, radiusToManipulate);
        #endif
    }
}
```



(a) `Manipulator in Editor-Ansicht`



(b) Manipulator im Inspektor

Abbildung 4.8: Repräsentation des Manipulators in Unity

4.3.3.2 Grundlage der Manipulation

Für die tatsächliche Anwendung von mehreren Manipulatoren wird zunächst ein manipuliertes Vektorfeld aus allen Manipulationen berechnet, welches diese Anwendung auf das Rohfeld (Rauschen oder Nullvektoren) beschreibt. Dafür werden zwei weitere Felder benötigt: Zum einen das Manipulationsfeld, welches die Feldwerte aller Manipulatoren enthält, zum anderen das Einflussfeld, welches bestimmt, zu welchem Anteil die Basisfeldwerte zu den Manipulationswerten linear interpolieren.

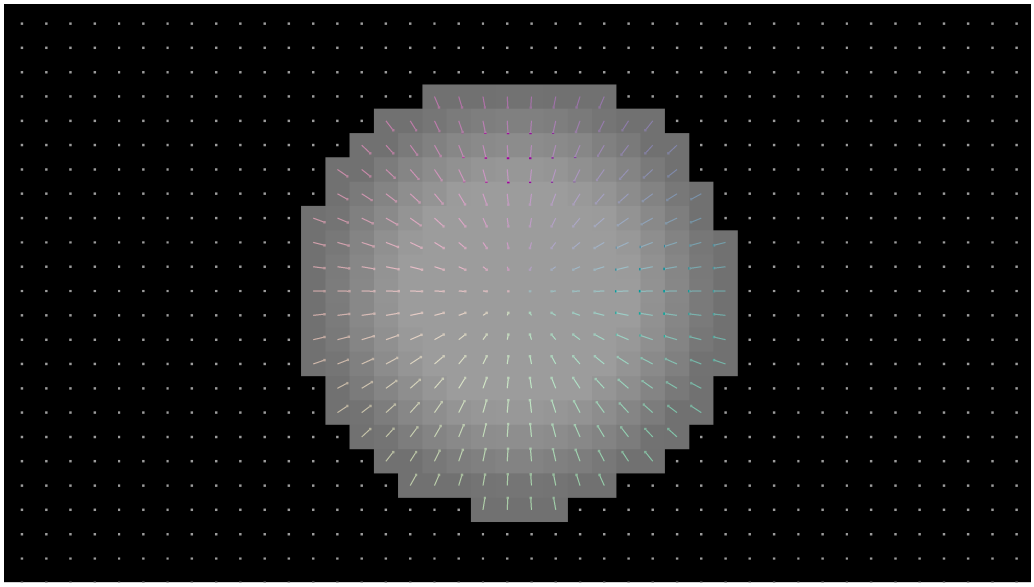


Abbildung 4.9: Manipulationsfeld mit Einflussfeld (Grauwert-Overlay)

Für die Berechnung des Manipulationsfelds und des Einflussfelds werden nacheinander die Werte für jeden einzelnen Manipulatoren berechnet. Die Werte eines Manipulators berechnen sich dabei auf Basis der Entfernung eines Voxels zum Zentrum des Manipulators. Dabei wird ähnlich einer linearen Interpolation die Entfernung ins Verhältnis zum Radius des Manipulators gesetzt. Damit werden außen andere Werte als weiter innen erzielt und zwischen diesen einen weichen Übergang geschaffen.

4.3.3.3 Berechnung des Manipulations- bzw. Einflussfeldes eines Manipulators

Die Berechnung des Manipulations- bzw. Einflussfeldes pro Manipulator erfolgt dabei separat in einer Methode `ManipulateWithCircle`. Diese erstellt ein kombiniertes Manipulations- und Einflussfeld (`manipulatorField`) in denselben Ausmaßen wie das Basisfeld. Dabei wird allerdings ein vierdimensionaler Vektor verwendet, um in der W-Komponente den Einflusswert unterbringen zu können. Der Einflusswert dient bei der späteren Anwendung der Manipulation auf das Vektorfeld dazu, das prozentual angegeben werden kann, wie stark der Vektor des Manipulationsfeldes mit dem ursprünglichen Vektor verrechnet werden soll. Als Parameter dienen die aktuelle Position des Manipulators in der Unity-Welt, der Radius des Manipulationskreises sowie der Modus der Manipulation.

```
// VectorField.cs
// ...
private Vector4[,,] ManipulateWithCircle(Vector3
   position, float radius, ManipulationMode mode)
```

```
{
    Vector4[,,] manipulatorField = null;
    var bounds = GetRealWorldBounds();
    bounds.Expand(2 * radius);

    if (bounds.Contains(position))
    {
        manipulatorField = new Vector4[width, height,
            depth];

        var localMatrix = transform.worldToLocalMatrix;
        var localPosition =
            localMatrix.MultiplyPoint(position);
        var localScale = _transform.localScale;
        var localRadius = radius /
            Mathf.Max(localScale.x, localScale.y);

        var minX = Mathf.Max(0,
            Mathf.FloorToInt(localPosition.x -
            localRadius));
        var minY = Mathf.Max(0,
            Mathf.FloorToInt(localPosition.y -
            localRadius));
        var minZ = Mathf.Max(0,
            Mathf.FloorToInt(localPosition.z -
            localRadius));
        var maxX = Mathf.Min(width - 1,
            Mathf.CeilToInt(localPosition.x +
            localRadius));
        var maxY = Mathf.Min(height - 1,
            Mathf.CeilToInt(localPosition.y +
            localRadius));
        var maxZ = Mathf.Min(depth - 1,
            Mathf.CeilToInt(localPosition.z +
            localRadius));

        // ...
```

Es wird zunächst festgestellt, ob die übergebene Position und Ausmaße des konkreten Manipulators sich überhaupt im Bereich des Vektorfeldes befinden bzw. das Feld schneiden. Für die anschließende Berechnung wird die Welt-Position des Manipulators auf lokale Vektorfeldkoordinaten umgerechnet. Dadurch ist nun bekannt, an welcher lokalen Position des Vektorfeldes sich der Manipulator befindet und welcher Bereich manipuliert werden soll. Der Radius wird auf eine evtl.

Skalierung des Gesamtfeldes in der Unity-Welt angepasst. Anschließend wird ein quadratischer Bereich auf dem entstehenden Manipulationsfeld ermittelt, welcher von der Beeinflussung betroffen sein wird. Dies sorgt für eine bessere Effizienz im nächsten Schritt, da nur innerhalb dieses Quadrates überprüft werden muss, ob sich eine Vektorfeldkoordinaten auch innerhalb des Manipulationskreises befindet.

Am Beispiel des `Implode`-Manipulators erfolgt die Generierung eines Wertes im Manipulationsfeldes wie folgend beschrieben. Für jeden Punkt im vorher festgelegten Quadrat wird festgestellt, ob dieser sich auch im Manipulationsradius befindet, und der prozentuale Abstand zum Zentrum des Manipulators berechnet:

```
// VectorField.cs
// ...

// For each coordinate in manipulation area x, y, z
var coordinateOfCell = new Vector3(x, y, z);
var distance = Vector3.Distance(coordinateOfCell,
   localPosition);

if (distance <= localRadius)
{
    var relativeDistance = Mathf.InverseLerp(0,
       localRadius, distance);

    // ...
    case ManipulationMode.Implode:
        manipulatorField[x, y, z] = (localPosition -
           coordinateOfCell).normalized * Mathf.Lerp(0,
           1, relativeDistance);
        manipulatorField[x, y, z].w =
            LerpWithBounds(1, 0.5f, 0.5f, 0.9f,
               relativeDistance);
        break;
    // ...
```

Im Falle von `Implode` wird ein Vektor von den Koordinaten der Zelle in Richtung des Zentrums des Manipulators berechnet. Die Länge des Vektors ist dabei abhängig von berechneten prozentualen Distanz der Zelle zum Zentrum, daher wird von außen nach innen wird die Einflusskraft schwächer. Der Einflusswert für die spätere Verrechnung mit dem Basisfeld hängt ebenfalls von der prozentualen Distanz ab. Dadurch entsteht ein Wert, der im Außenbereich des Manipulationskreises einen weicheren Einfluss aufweist, damit dort der Anteil des Basisfelds höher ist. Die Einflüsse sind dabei auf minimale und maximale Werte beschränkt. Dafür wird eine Erweiterung einer linearen Interpolation verwendet, welche den

Interpolationswert zusätzlich so beschränkt, dass dieser nicht wie gewöhnlich zwischen 0 und 1 liegt, sondern parametrisierbare Maxima bzw. Minima aufweist und Interpolationswerte darüber oder darunter auf den jeweiligen maximalen oder minimalen Ergebniswert beschränkt. (`LerpWithBounds`).

Das Ergebnis ist ein temporäres Feld für einen Manipulator, welches die Manipulation beschreibt und gleichzeitig den gewünschten Einfluss bei der folgenden Verrechnung mit dem Basisfeld enthält (siehe Abbildung 4.9).

4.3.3.4 Anwendung der Manipulation auf das Basisfeld

Die Felder der einzelnen Manipulatoren werden so addiert, dass die Einflusswerte und die Länge der Manipulationswerte bzw. -vektoren nie größer als 1 sind, und bilden damit das Manipulationsfeld bzw. Einflussfeld.

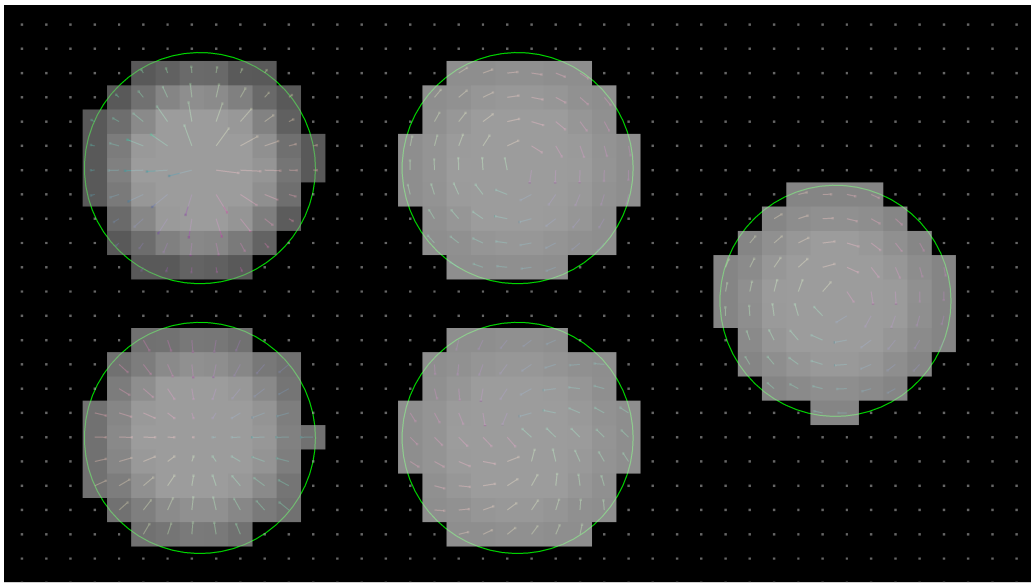


Abbildung 4.10: Kumulierte Manipulationsfelder mit Einflussfeld (Grauwert-Overlay)

Zwischen dem daraus entstandenen kumulierten Manipulationsfeld und dem Basisfeld wird in Abhängigkeit vom Einflussfeld linear interpoliert. Daraus resultiert das aktuelle Feld (daher die Kombination aus dem Basisfeld mit angewendeter Manipulation). Dabei werden nur Werte aus dem Manipulationsfeld berücksichtigt, welche einen Einfluss (> 0) haben.

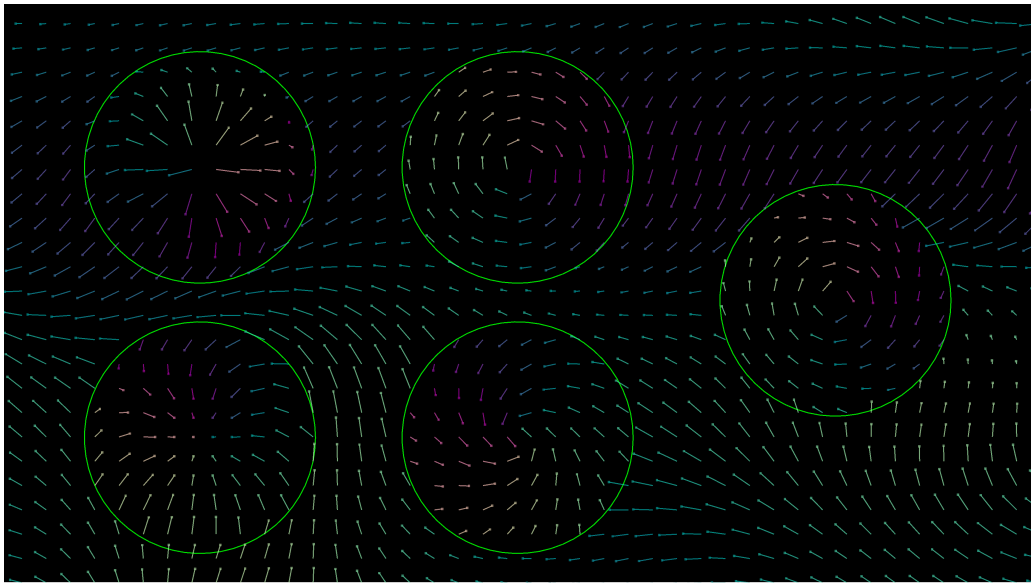


Abbildung 4.11: Vektorfeld nach Anwendung aller Manipulatoren

Bei Bewegung oder Entfernung eines Manipulators besteht die Möglichkeit, dass nicht mehr betroffene Voxel allmählich zum Zustand des Basisvektorfelds zurückkehren sollen. Dies wird erreicht, indem die nicht manipulierten Bereiche im Laufe der Zeit zurück zum Zustand des Basisfelds linear interpoliert werden.

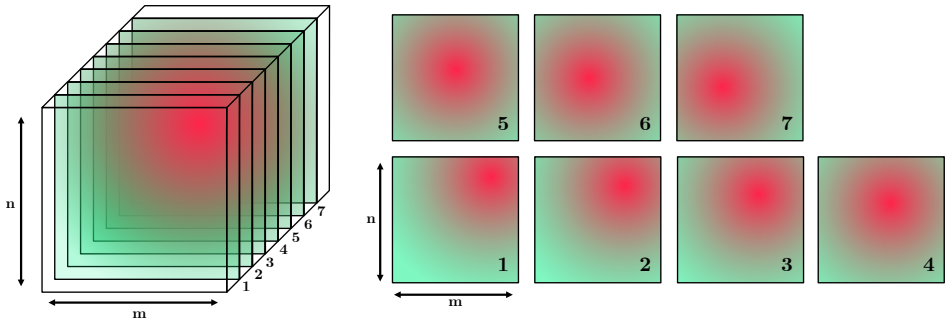
4.4 Integration des manipulierten Vektorfeldes in Unity-Systeme

Bei den bisherigen Ergebnisbildern handelt es sich um die Visualisierung der Datenstruktur eines manipulierbaren Vektorfeldes. Es existiert also die Möglichkeit, ein dynamisches interaktives Vektorfeld in der Unity-Welt zu platzieren, dieses positionsabhängig durch einen Manipulator mit unterschiedlichen Modi zu beeinflussen und dies darzustellen.

4.4.1 Verwendung einer `Texture3D`

Der nächste Schritt ist es, die entstandenen Daten in eine Form zu bringen, welche es ermöglicht, das Partikel aus einem Unity-Partikelsystem den durch das Vektorfeld definierten Flussverläufen bzw. Kräften folgen können. In Unity unterstützen sowohl das »Shuriken Particle System« sowie der »VFX-Graph« eine Einwirkung von Kräften auf den von ihnen verwalteten Partikeln. Dies geschieht normalerweise in Form von einer statischen dreidimensionalen Volumentextur (`Texture3D`), welche ein Vektorfeld repräsentiert.

Im Kontext von Vektorfeldern kann eine `Texture3D` als eine dreidimensionale Textur betrachtet werden, die Informationen über das Vektorfeld enthält. Eine Textur wird normalerweise als zweidimensionales Bild betrachtet, bei dem jedem Pixel ein Farbwert zugeordnet ist. Bei einer `Texture3D` gibt es jedoch zusätzlich zur Breite und Höhe auch eine Tiefe, sodass jedem Punkt im Raum ein Farbwert zugeordnet werden kann.



Volumentextur (3D) Korrespondierende Flachtexturen (2D) pro Schicht

Abbildung 4.12: Schematische Darstellung einer Volumentextur

Zur Verwendung der integrierten Beeinflussungswerkzeuge der genannten Partikelsysteme muss daher die Datenstruktur des Vektorfeldes zur Laufzeit in eine solche `Texture3D` umgewandelt werden. Bei der Implementierung des prototypischen Beispiels wird eigentlich ein zweidimensionales Vektorfeld für die zweidimensionale Interaktion verwendet, welches aber trotzdem aus Kompatibilitätsgründen in eine (`Texture3D`) gewandelt werden muss.

Die Informationen des Vektorfeldes liegen momentan als dreidimensionales Array bzw. ein Gitter von `Vector3`-Strukturen vor. Jeder Voxel in diesem Gitter enthält einen Vektorwert, der die Richtung und den Betrag (Kraft) des Vektorfeldes an diesem Punkt darstellt. Jeder dieser Vektoren besteht wiederum aus drei Komponenten, welche auch als Farbwert für eine RGB-Farbe interpretiert werden können. Wobei dort folgende Zuordnung gilt: Roter Kanal entspricht X-Wert, grüner Kanal entspricht Y-Wert und blauer Kanal entspricht Z-Wert. Dies stellt die Grundlage für die Repräsentation des Vektorfeldes in Form einer `Texture3D` dar.

Für `Texture3D`-Texturen gibt es verschiedene Formate. Ein Format beschreibt dabei, in welcher Form die Farbinformationen in einer Textur gespeichert sind bzw. wie diese zu interpretieren sind. Für die Überführung des Vektorfeldes in eine Textur muss das Texturformat gewisse Anforderungen erfüllen. Die Vektoren des Vektorfeldes sind normalerweise Fließkommazahlen in einem Wertebereich zwischen -1 und 1 für jede Komponente. Daher muss das Texturformat sowohl mit

negativen Werte als auch Fließkommazahlen im gegebenen Wertebereich umgehen können. Ein dafür in Unity geeignetes Format stellt das `RGBAFloat`-Format dar. Dieses Format unterstützt die benötigten Farbinformationen mit einem passenden Wertebereich.

4.4.1.1 Erzeugung einer `Texture3D`

Unity unterstützt die Instanziierung und Aktualisierung solcher Textur-Objekte zur Laufzeit. In dieser Implementierung wird aus Effizienzgründen einmalig zur Laufzeit eine Textur-Instanz erstellt und wie folgt konfiguriert:

```
// VectorField.cs
// ...

private static Texture3D GenerateEmptyTexture3D(uint
    width, uint height, uint depth)
{
    // Configure the texture
    const TextureFormat format = TextureFormat.RGBAFloat;
    const TextureWrapMode wrapMode =
        TextureWrapMode.Clamp;

    // Create the texture and apply the configuration
    var texture = new Texture3D((int)width, (int)height,
        (int)depth, format, false)
    {
        wrapMode = wrapMode
    };

    // Apply the changes to the texture and upload the
        updated texture to the GPU
    texture.Apply();

    return texture;
}
```

Diese Funktion generiert eine leere vorkonfigurierte dreidimensionale Textur einer festen Größe. Diese wird bei der Initialisierung des Vektorfeldes in dessen Maßen angelegt. Zur Laufzeit kann dann jede Änderung des Vektorfeldes wie folgt in die Textur geschrieben werden:

```
// VectorField.cs
// ...

private void RefreshTexture3D()
{
    var colors = new Color[width * height * depth];
    for (var z = 0; z < depth; z++) {
        for (var y = 0; y < height; y++) {
            for (var x = 0; x < width; x++) {
                var value = _vectorField[x, y, z];
                colors[x + y * width + z * width *
                   height] = new Color(value.x, value.y,
                   value.z, 1f);
            }
        }
    }

    _texture3D.SetPixels(colors);
    _texture3D.Apply(true);
}
```

Für jeden Vektor des Vektorfeldes wurde der mit dessen Position korrespondierender Pixel der Textur angepasst.

4.4.1.2 Ergebnis der Umwandlung

In diesem Abschnitt wird nun das Ergebnis der Umwandlung von Vektorfelddaten in `Texture3D`-Texturen gezeigt. Da für den Prototyp nur zweidimensionale Vektorfelder notwendig sind, können die generierten Texturen auch als zweidimensionale Abbildung gezeigt werden.

Zunächst wird ein Basisvektorfeld (generiert mit Rauschfunktion) betrachtet, welches mit dem beschriebenen Verfahren in eine `Texture3D` umgewandelt wurde.

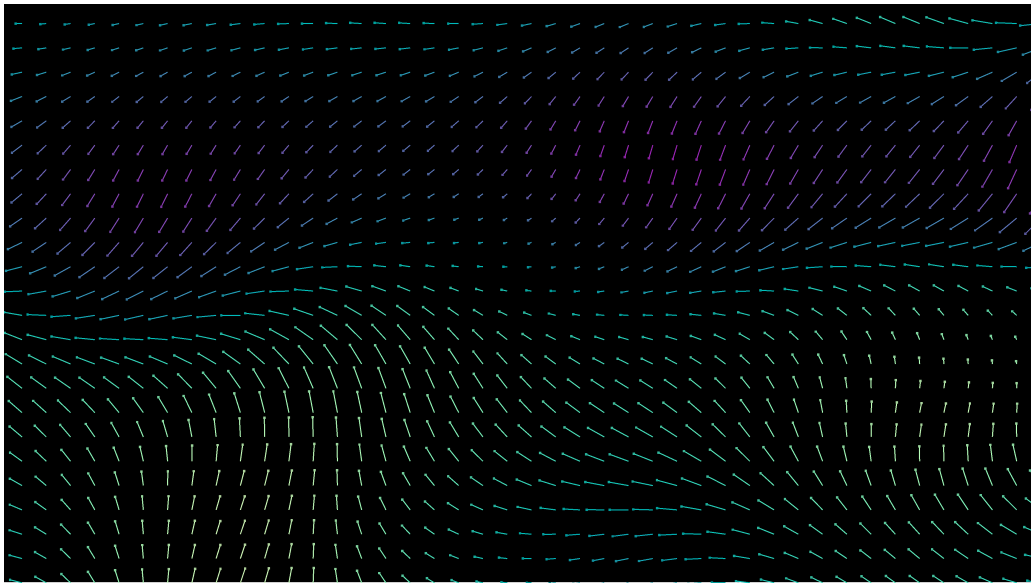


Abbildung 4.13: Visualisierung eines Basisvektorfeld

Das Ergebnis (Abbildung 4.14) der Umwandlung ist eine Textur, welche optisch mit dem Vektorfeld korrespondiert. Die Umwandlung ist mit dieser abgeschlossen und das Ergebnis kann bereits für die spätere Anwendung auf ein Partikelsystem verwendet werden.

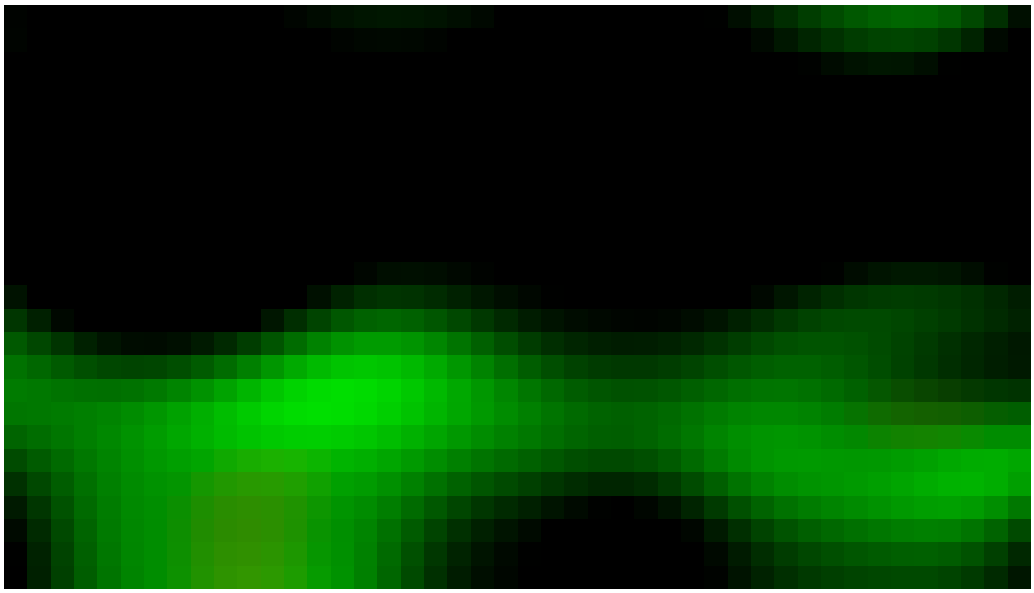
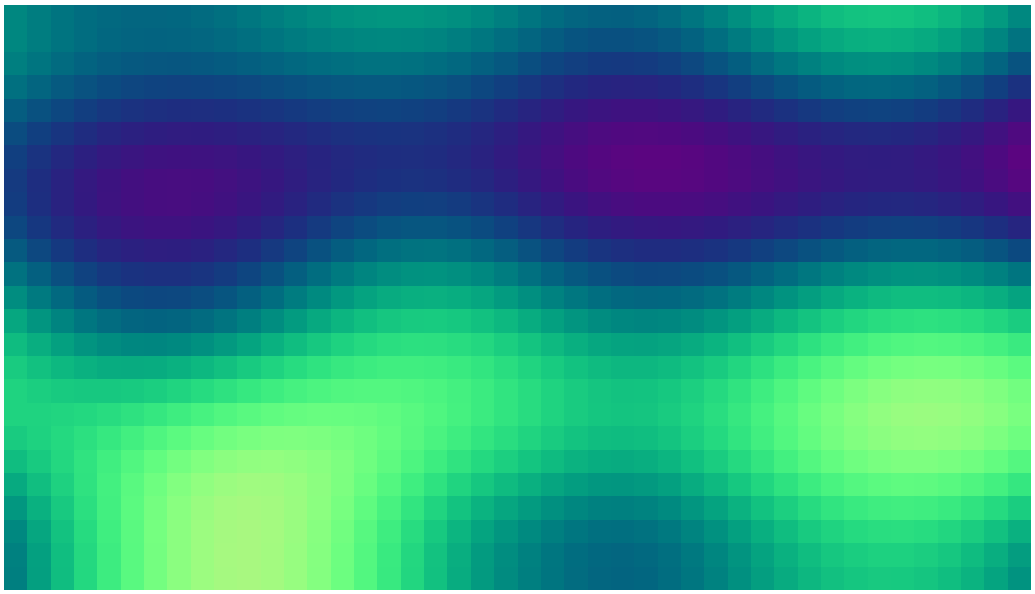
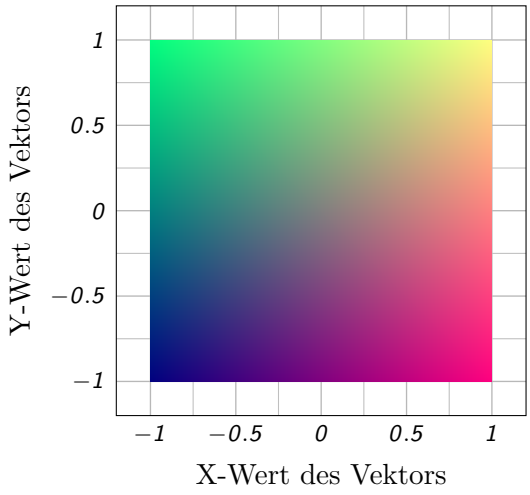


Abbildung 4.14: Ergebnis des Basisvektorfeld als `Texture3D`

Zum besseren Verständnis bzw. zur besseren Interpretation der Ergebnistextur folgt nun eine farbverschobene Darstellung mit einer Legende. Dies wurde aus dem Grund umgesetzt, da Vektoren mit negativen Komponenten bzw. negativen Richtungen ansonsten im RGB-Farbraum nur unzureichend visualisiert können. Die betroffenen Vektoren sind in der vorherigen Darstellung unvollständig sichtbar, da negative Komponenten bei der Darstellung im RGB-Farbraum als Nullwerte (und damit als Schwarz) interpretiert werden.



(a) `Texture3D` (Farbverschoben)



(b) Legende

Abbildung 4.15: Ergebnis des Basisvektorfeld als `Texture3D` (Farbverschoben)

Durch die Farbverschiebung kann mithilfe von Abbildung 4.15b aus der Farbe die Ausrichtung und Stärke der Vektoren im Vektorfeld interpretiert werden. Dadurch erkennt man, dass das Vektorfeld aus Abbildung 4.13 mit der `Texture3D` aus Abbildung 4.15a übereinstimmt und die Umwandlung erfolgreich durchgeführt wurde.

Bei einem manipulierten Vektorfeld ist die Manipulation auf der farbverschobenen Ergebnistextur deutlich erkennbar (Abbildung 4.16 und Abbildung 4.17).

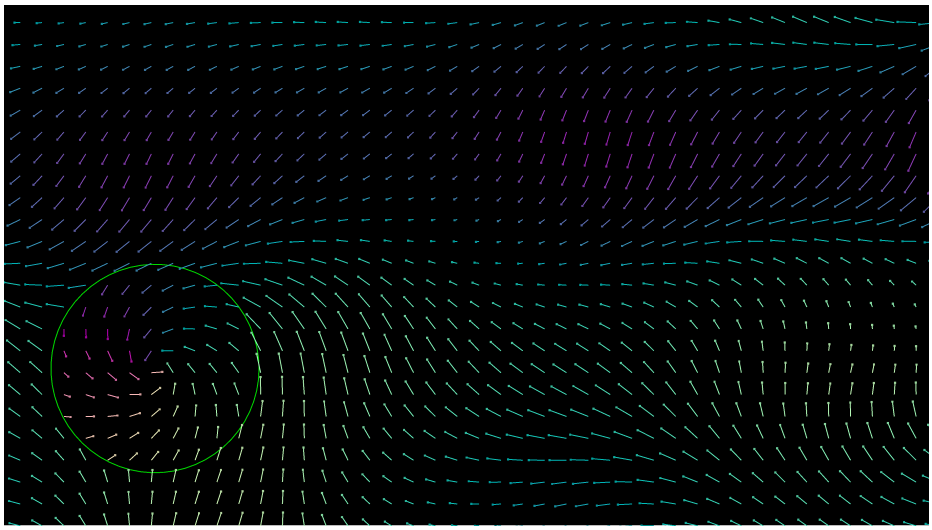


Abbildung 4.16: Visualisierung eines manipulierten Vektorfeldes

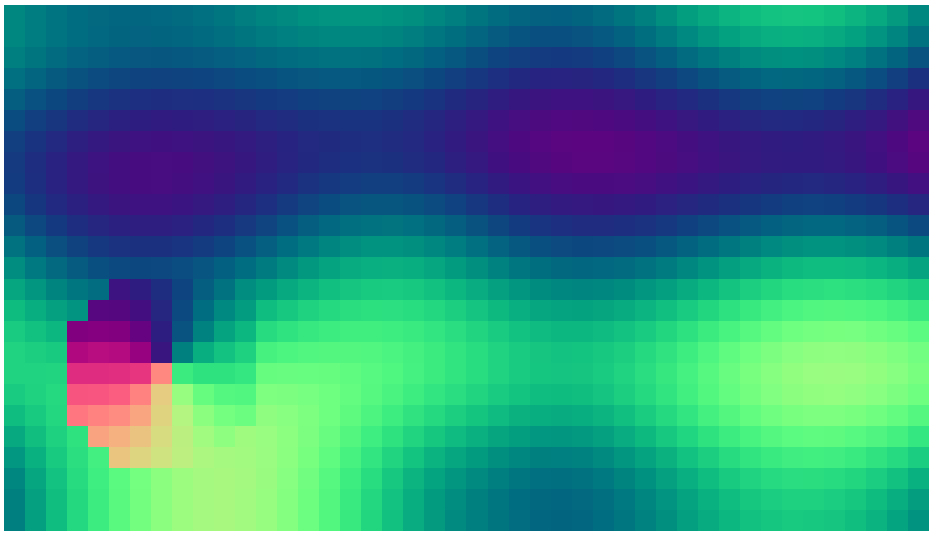


Abbildung 4.17: Ergebnis des manipulierten Vektorfeldes als `Texture3D` (Farbverschoben)

4.4.2 Einsatz des Vektorfeldes in Partikelsystemen

Die erzeugte Repräsentation des Vektorfeldes als `Texture3D` kann nun zur Beeinflussung von Partikeln verwendet werden. Dabei ist diese Textur theoretisch für beide Hauptpartikelsysteme nutzbar. Im Folgenden wird die Anwendung für beide Systeme beschrieben. Das »Shuriken Particle System« unterstützt zwar prinzipiell die Verwendung von Vektorfeldern auf Basis von `Texture3D` als externe Kraft. Es wurde allerdings so konzipiert, dass kontinuierliche Aktualisierungen einer bestehenden `Texture3D`-Instanz nicht mehr auf das Partikelsystem angewendet werden bzw. Änderungen nicht im tatsächlichen Partikelfluss reflektiert werden. Daher ist dieses im Kontext einer Manipulierung zur Laufzeit nicht verwendbar, sondern nur für statische Felder nutzbar. Eine Möglichkeit, dies zu umgehen, ist es, für jede Aktualisierung eine neue `Texture3D`-Instanz zu erstellen. Davon sollte abgesehen werden, da die Instanziierung von `Texture3D`s einen ressourcenintensiven Prozess darstellt, weil dabei unter anderem fortwährend Speicherallokierungen sowohl im Hauptspeicher als auch in der GPU durchgeführt werden. Für den tatsächlichen Prototyp wird daher nur vom »VFX-Graph« als Partikelsystem Gebrauch gemacht.

4.4.2.1 Shuriken Particle System

Das Shuriken Particle System ist ein komponentenbasiertes System, das auf dem Emitter-Partikel-Modell basiert. Es verwendet vorgefertigte Module, um die Eigenschaften von Partikeln zu steuern, wie zum Beispiel Geschwindigkeit, Lebensdauer, Farbe und Größe. Es unterstützt die Beeinflussung durch externe Kräfte durch das integrierte `External Forces module`. Dieses Modul ist ursprünglich ausschließlich dafür gedacht eine Unterstützung für die Einwirkung von zwei integrierten Unity-Komponenten zu bieten und kann einfach in einer bestehenden Partikelsystem-Komponente aktiviert werden.

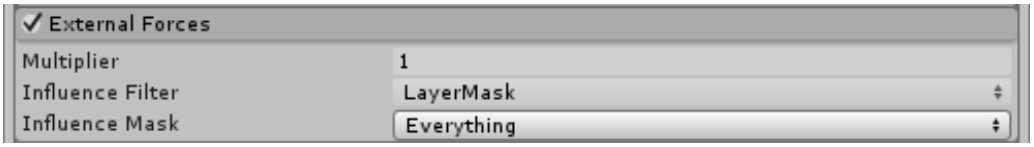


Abbildung 4.18: `External Forces module` im Inspektor

Nach der Aktivierung können die Kräfte aus den unterstützten Komponenten auf das Partikelsystem einwirken. Bei den unterstützen Komponenten des Moduls handelt es sich einmal um die »Wind Zones«-Komponente des Unity-Terrain-Systems, welche eine konfigurierbare Windfeldkurve generiert und auch auf Partikel des »Shuriken Particle Systems« angewendet werden kann. Zusätzlich zu dieser existiert eine Unterkomponente des Partikelsystems `Particle System Force Field`. Dabei handelt es sich um ein skalierbares Feld, welches verschiedene Kräfte (wie Richtungsänderungen, Gravitation, Rotation oder Reibung) auf

die eingeschlossenen Partikel des Partikelsystems anwenden kann. Eine weitere für dieses Kapitel wichtige Funktion ist die Möglichkeit eine Volumen-Textur (`Texture3D`) eines Vektorfeldes als Kraft anzuwenden.

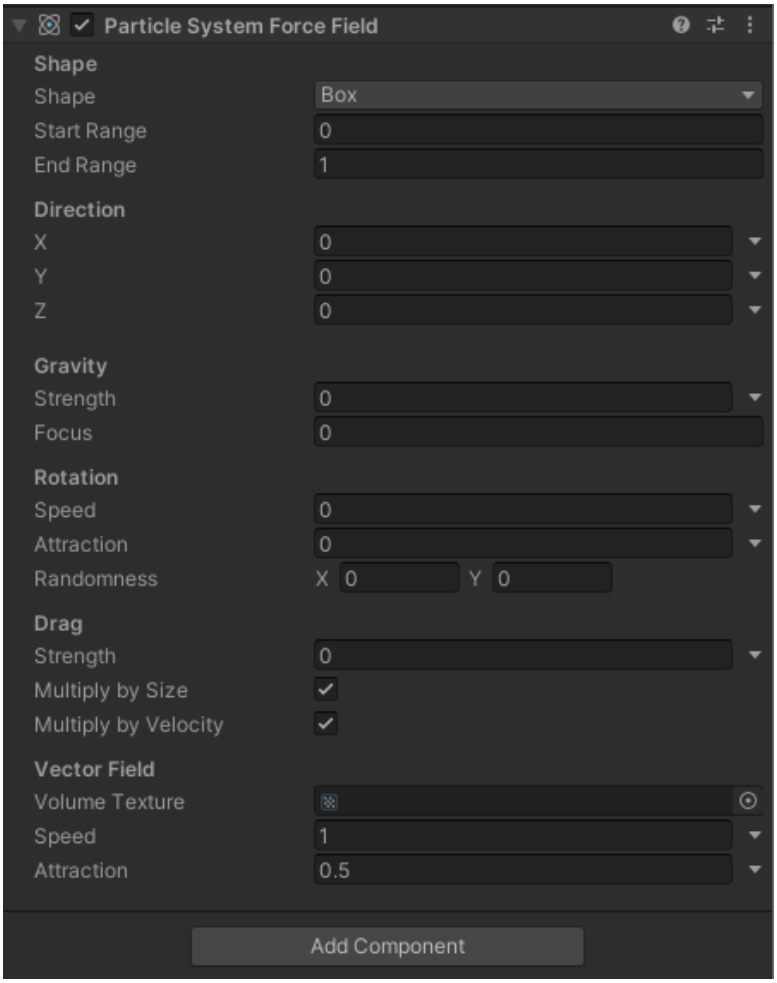


Abbildung 4.19: `ParticleSystemForceField` im Inspektor

Für die Verwendung der im vorherigen Unterkapitel erzeugten Textur wird die Komponente `Particle System Force Field` entsprechend konfiguriert. Die Form des Einflusses wird auf eine Box gestellt und das `Particle System Force Field` auf die gewünschten Maße skaliert. In diesem Fall stimmen die Maße mit der visuellen Repräsentation des Vektorfelds überein.

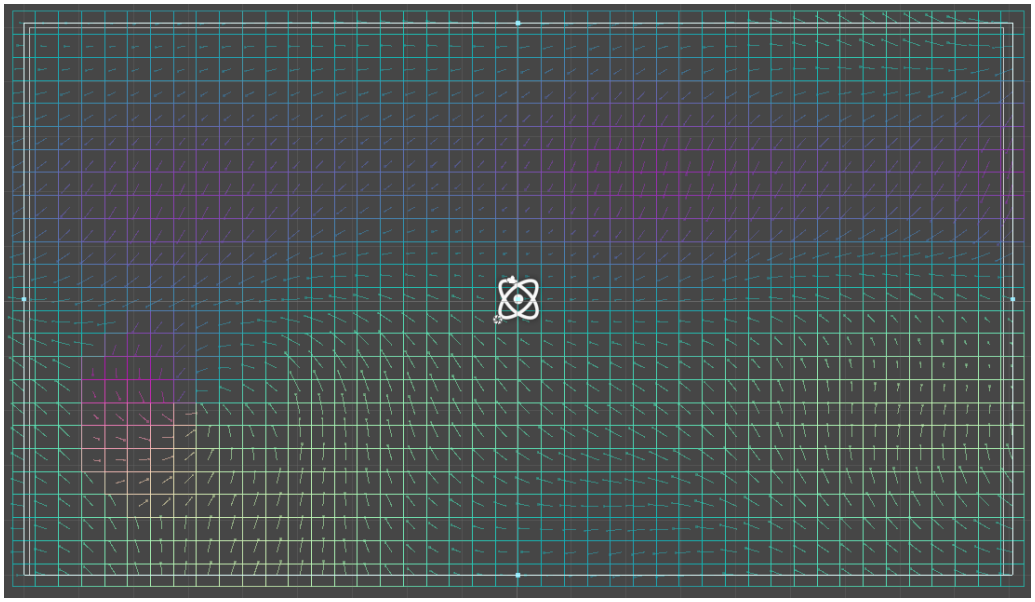


Abbildung 4.20: `Particle System Force Field` im Editor

Die generierte Textur muss nur auf das `Particle System Force Field` angewendet werden und Partikel beginnen den Kräften des Vektorfeldes entsprechend zu folgen.

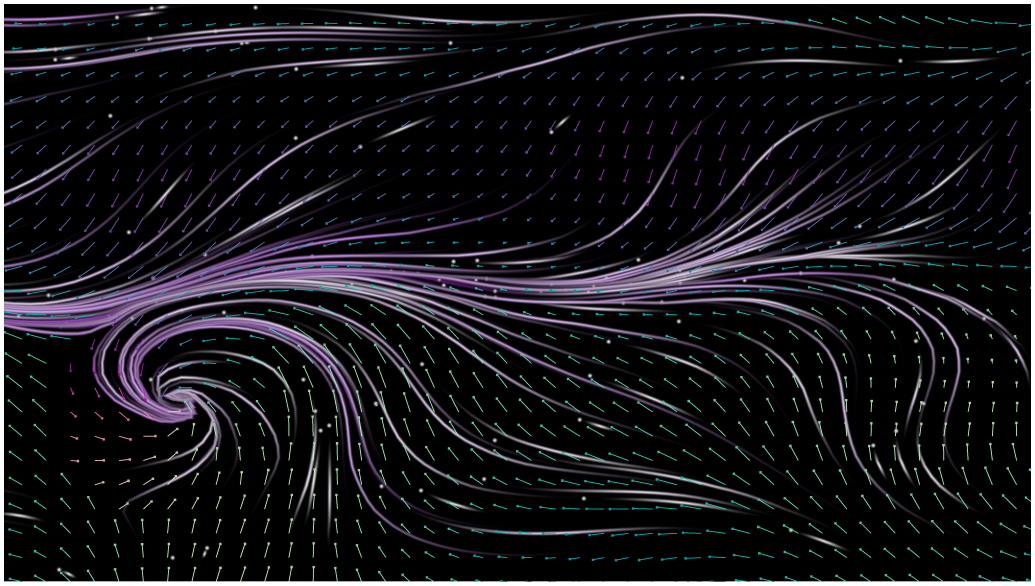


Abbildung 4.21: Partikelfluss anhand des Vektorfeldes (»Shuriken Particle Systems«)

Die Partikel folgen erkennbar dem Fluss des Vektorfeldes. Aufgrund der genannten Einschränken des `Particle System Force Field` reagiert der Partikelfluss nicht mehr auf Änderungen des Feldes.

4.4.2.2 VFX-Graph

Der VFX-Graph ist mit Stand dieser Arbeit ein relativ neues Partikelsystem in Unity. Es handelt sich dabei um ein grafisches System, das auf knotengestützter bzw. node-gestützter Shader-Programmierung basiert (ähnlich zum »Unity Shader Graph«). Es ermöglicht eine feine Kontrolle über die visuellen Effekte, indem komplexe Shader-Graphen erstellt werden können, die die Partikelinteraktionen und -simulationen steuern. Ein markanter Unterschied ist, dass der VFX-Graph im Unterschied zum »Shuriken Particle System« primär auf der GPU ausgeführt wird.

Im VFX-Graph steht der Knoten oder Node `Vector Field Force` zur Verfügung, um die Kräfte, eines durch eine `Texture3D` definierten Vektorfelds, auf Partikel anzuwenden. Obwohl der Knoten eigentlich für die Anwendung von statischen Vektorfeldern konzipiert ist, ermöglicht er auch die Umsetzung von Änderungen der `Texture3D`-Instanz zur Laufzeit.

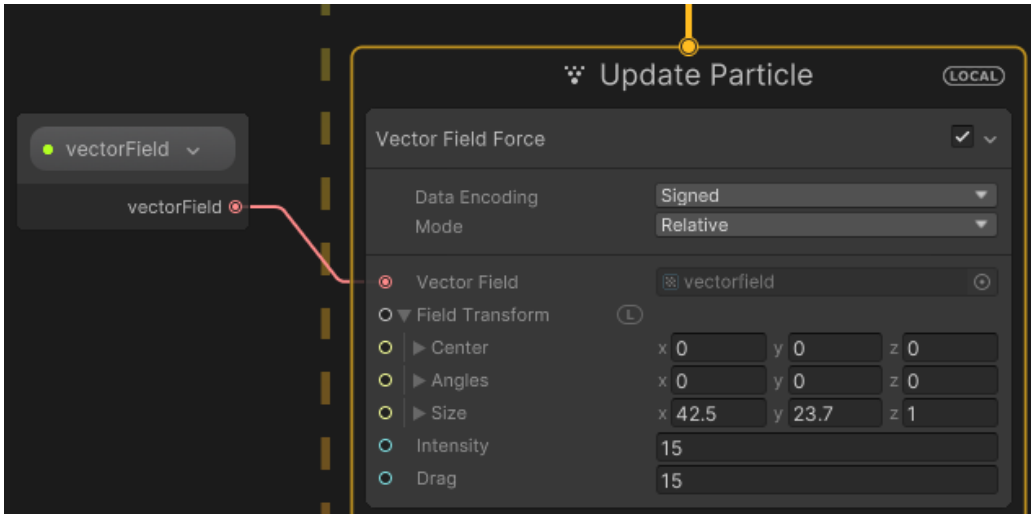


Abbildung 4.22: `Vector Field Force`-Node

Die Vermaßung (`Field Transform`) kann, ähnlich wie beim »Shuriken Particle System«, mit der visuellen Repräsentation des Vektorfeldes überlagert werden.

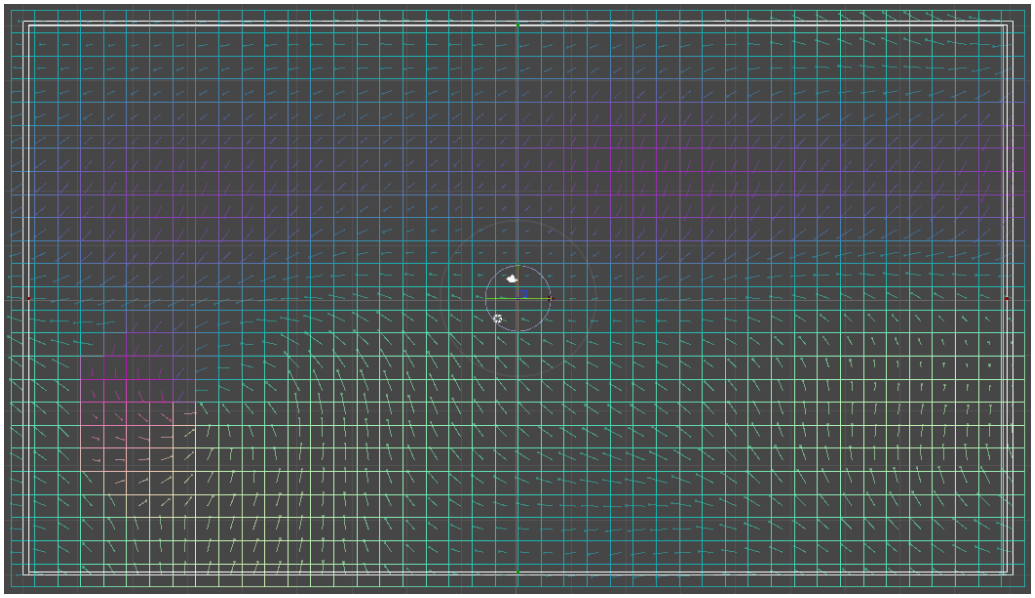


Abbildung 4.23: `Vector Field Force` Ausmaße im Editor

Zur Anwendung wird auf dem sogenannten »Blackboard« des VFX-Graphen eine Variable für ein Objekt des Typs `Texture3D` angelegt. Die generierte Textur muss nun über die eben erstellte Variable auf die `Vector Field Force`-Node angewendet werden und die Partikel beginnen, den Kräften des Vektorfeldes entsprechend zu folgen. In diesem Fall werden auch die kontinuierlichen Änderungen des dynamischen Vektorfeldes berücksichtigt.

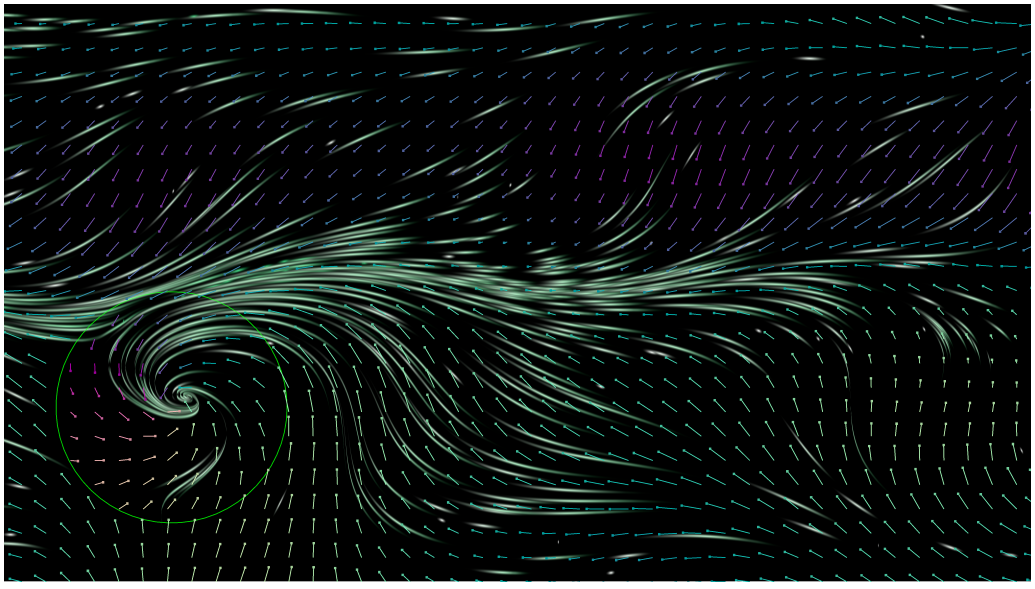


Abbildung 4.24: Partikelfluss anhand des Vektorfeldes (VFX-Graph)

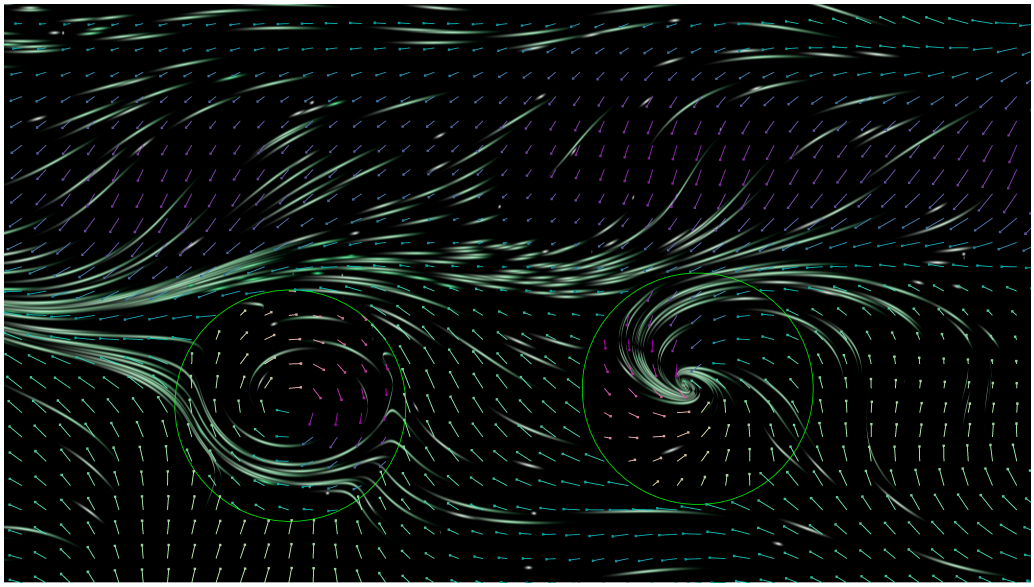


Abbildung 4.25: Veränderter Partikelfluss anhand des Vektorfeldes (VFX-Graph)

4.4.3 Ergebniskomponente und ihre Anwendung auf den Partikelfluss

Das Ergebnis der Implementierung stellt nun zusammenfassend die komponentenbasierte Grundlage für durch einen positionsabhängigen Manipulator manipulierbare Vektorfelder.

Die entstandene `VectorField`-Komponente ermöglicht dabei eine Konfiguration und Platzierung von diskreten Vektorfeldern in der Unity-Welt.

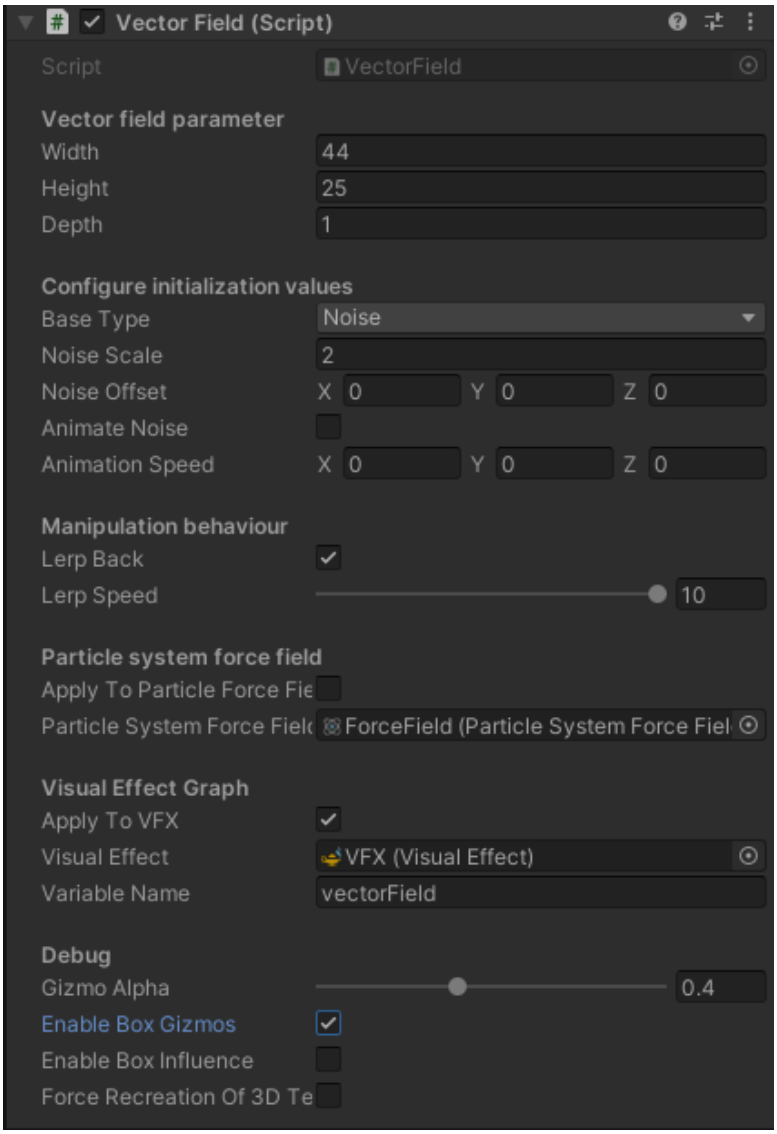


Abbildung 4.26: `VectorField`-Komponente im Inspektor

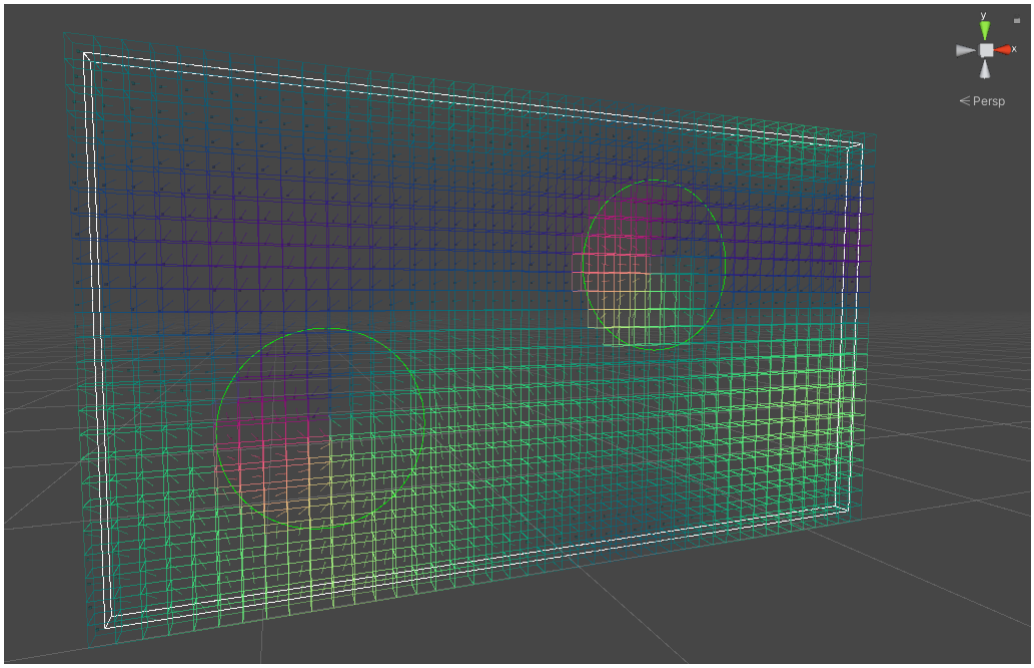


Abbildung 4.27: `VectorField`-Komponente plaziert in Unity-Welt

Diese manipulierbaren Vektorfelder können zur Beeinflussung von Partikelsystemen auf Basis einer zur Laufzeit generierten Abbildung des Vektorfeldes als `Texture3D` verwendet werden und diese auf ein Partikelsystem anwenden. Die folgenden Ergebnisbilder verwenden den VFX-Graph.

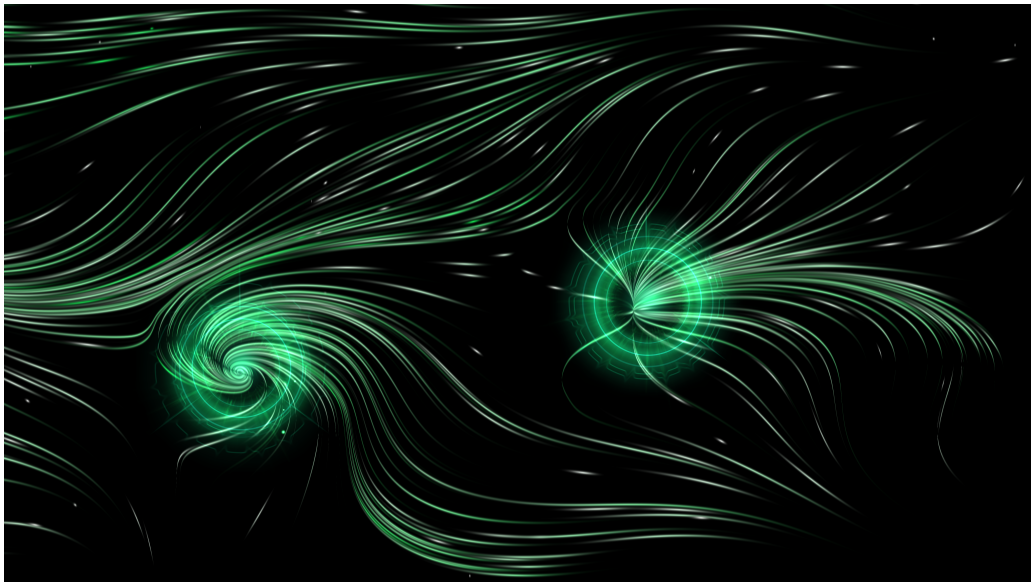


Abbildung 4.28: Partikelfluss mit zwei Manipulatoren im Playmode

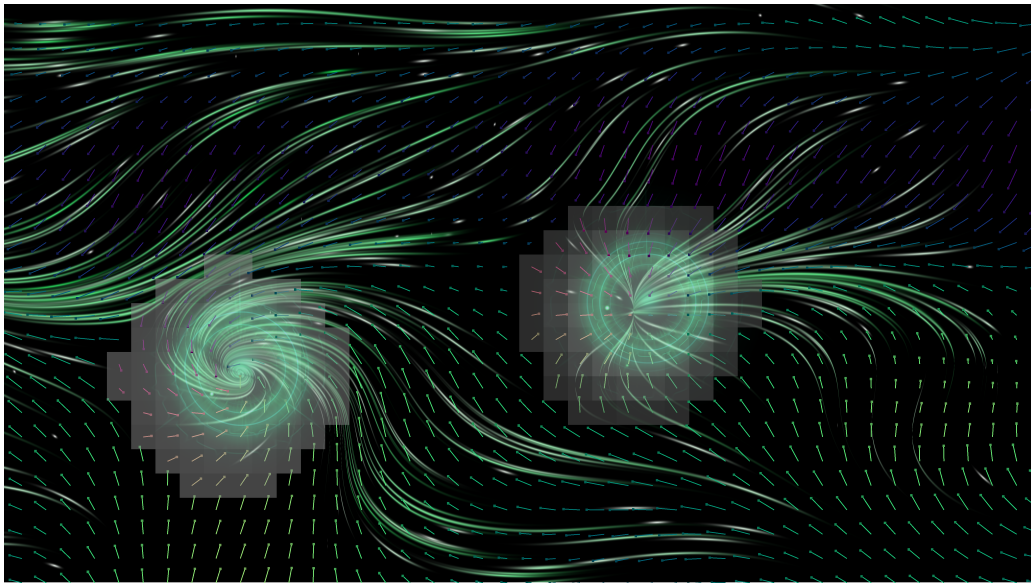


Abbildung 4.29: Partikelfluss mit zwei Manipulatoren (Überlagertes Vektorfeld und Einflussfeld)

Zur besseren Visualisierung des Manipulators wurden diese mit einem weiteren visuellen Effekt (Wurmlochartiger Effekt bei Manipulatoren) ausgestattet, um den Manipulator zur Laufzeit erkennbar zu machen. Die Grundlage des Interaktionsprototyps wurde damit gebildet.

4.5 Einbindung der Interaktionskomponente

Zur Komplettierung des Interaktionsprototyps wird die Szene des Vektorfeldes noch um die im vorherigen Kapitel (siehe Kapitel 3) implementierte Komponente `BodyTrackingManager2D` sowie das notwendige Kamera-Prefab des ZED-Systems ergänzt (siehe Abschnitt 3.3).

Eine Komponente wird erstellt, die zur Laufzeit für die Hände erkannter Personen ein Manipulator-Objekt erzeugt und dessen Position auf dem Vektorfeld in Abhängigkeit der überführten Positionsdaten vom ZED-System bzw. des `BodyTrackingManager2D` aktualisiert. Diese bildet die Grundlage für ein Spawner-Objekt in der Szene.

Die Anbindung an das ZED-System verläuft ähnlich wie in Abschnitt 3.3 bei der Anbindung an die Komponente `BodyTrackingDebugVisualizer2D`. Die `BodyTrackingManager2D`-Komponente wird entsprechend für die Szene konfiguriert (Skalierung der übergebenen Körperdaten zur Anpassung der Größenverhältnisse an die Szene) und das erstellte Spawner-Objekt wird an diese angebunden.

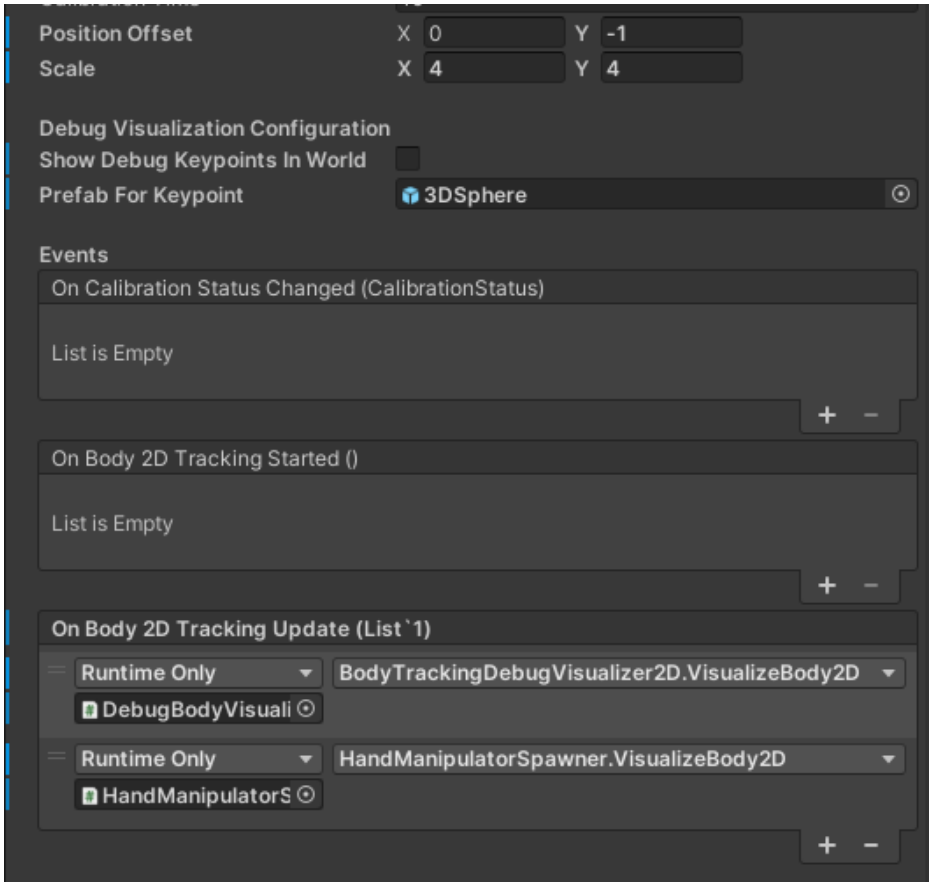


Abbildung 4.30: Anbindung und Konfiguration des `BodyTrackingManager2D` für die Vektorfeldmanipulation

Damit wird nun für jede vor dem Bildschirm erkannte Person ein Manipulator pro Hand erstellt, der den Positionsveränderungen der jeweiligen Hand folgt.

5. Ergebnisse

Diese Arbeit beschäftige sich mit der Schaffung bzw. Implementierung eines kamerabasierten Interaktionskonzeptes für zweidimensionale virtuelle Welten bzw. Anwendungen. Dafür wurde ein Konzept geschaffen, welches eine direkte Interaktion mit einer zweidimensionalen Anwendung auf Basis eines NUI als Erkennungssystem in der Laufzeit- und Entwicklungsumgebung Unity ermöglicht. Die Interaktion sollte dabei auf einer Bildschirmfläche gegenüber der Nutzenden stattfinden und ohne zusätzliche Vorbereitung von diesen (durch weitere Hardware bzw. Eingabegeräte) möglich sein. Als technologische Basis wurde dafür zu Beginn dieser Arbeit das Erkennungssystem in Form einer stereoskopischen Tiefenkamera (ZED 2i) und des dazugehörigen SDKs (ZED-System) erläutert (siehe Kapitel 2). Das ZED-System erzeugt dabei auf Basis des KI-basierten SDK-Moduls »Body Tracking Module« die Datengrundlage für die Körperbewegungsdaten der Nutzenden.

Zur Lösung der Grundproblemstellung und damit zur Ermöglichung der gewünschten Interaktion auf einem Bildschirm war es notwendig, die perspektivischen dreidimensionalen Daten des Erkennungssystems aufzubereiten. Das ZED-System ist eigentlich für die Nutzung mit VR- bzw. AR-Anwendungen aus Sicht der Kamera konzipiert. Daher wurde dafür eine technologische Grundlage in Form einer Unity-Komponente (`BodyTrackingManager2D`) implementiert, welche die dreidimensionalen Körperdaten des ZED-Systems empfängt und für die Nutzung dieser in zweidimensionalen interaktiven Komponenten überführt (siehe Kapitel 3). Auf Basis dieser Komponente wurde anschließend ein Beispiel für dessen Nutzung implementiert, bei welchem es sich um einen Interaktionsprototyp in Form eines dynamischen, durch die Hände des Nutzenden manipulierbares Vektorfelds handelt. Der Nutzer erhält damit die Möglichkeit, einen auf diesem Vektorfeld basierenden Partikelstrom händisch zu beeinflussen (siehe Kapitel 4).

5.1 3D-2D-Transformation (`BodyTrackingManager2D`)

Zur Realisierung des `BodyTrackingManager2D` und damit der Anforderungen aus Unterabschnitt 3.2.1, wurden zunächst mögliche theoretische Lösungsansätze vorgestellt, um eine Grundlage für die Nutzung in zweidimensionalen Anwendungen relativ zu einer Bildschirmfläche zu erhalten. Das ZED-System liefert Körperdaten von Personen in seinen Erkennungsbereich. Dabei handelt es sich um drei-

dimensionale Schlüsselpunktpositionen verschiedener Skelett-Fixpunkte einer jeden erkannten Person (siehe beispielsweise Tabelle 3.1). Diese dreidimensionalen Schlüsselpunkte sind relativ zur Kamera-Hardware positioniert bzw. orientiert und werden auch dementsprechend so in Unity empfangen.

Ausgehend davon, dass der Bildschirm eine zweidimensionale Fläche ist, die keine Tiefenachse bzw. Z-Achse besitzt und so auf der XY-Ebene auch in Unity repräsentiert werden soll, entspricht die Ausrichtung der ZED-Kamera jedoch nicht zwangsläufig der Ausrichtung des Bildschirms – die ZED-Kamera muss bzw. kann nicht unbedingt direkt hinter dem Bildschirm stehen und so auf die Personen vor dem Bildschirm frontal ausgerichtet sein. Die Ausrichtung der Personen in der Unity-Welt muss also in dem Sinne korrigiert werden, dass die Tiefenachse der Kamera mit der Tiefenachse von Unity bzw. der auf dem Bildschirm dargestellten virtuellen Kamera übereinstimmt. Des Weiteren muss die Position so korrigiert werden, dass eine mittig vor dem Bildschirm stehende Person auch mittig in der Anwendung auf dem Bildschirm repräsentiert wird. Da in Unity die virtuelle Kamera bei zweidimensionalen Unity-Anwendungen normalerweise auf den Unity-Weltursprung ausgerichtet ist und die XY-Ebene darstellt, müssen die empfangenen Daten so korrigiert werden, dass eine zentral zum Bildschirm positionierte und ausgerichtete Person einen spiegelartigen Eindruck während der Interaktion erhält. Zur Lösung dieses Problems wird eine Starrkörpertransformation aller Schlüsselpunkte durchgeführt (siehe Abschnitt 3.1) um diese an diese in die benötigte Position und Ausrichtung zu transformieren. Die empfangenen Schlüsselpunkte der Personen werden dafür transformiert und anschließend an andere Komponenten für die Weiterverarbeitung bzw. der Schaffung von Interaktionen übermittelt.

Die entwickelte Komponente etabliert dafür einen Kalibrierungs- und Transformationsprozess. Für einen gegebenen Aufbau eines Interaktionssystems (daher die Positionierung vom ZED-System und des Bildschirm) werden vor Beginn der Interaktion Kalibrierungsdaten von einer Referenzperson erfasst. Die erste erkannte Person wird als Referenzperson verwendet und positioniert und orientiert sich dafür mittig zur Bildschirmfläche. Die von der Referenzperson erhobenen Positions- bzw. Orientierungsdaten relativ zum ZED-System dienen als Grundlage für die Transformation, bei welcher die Referenzperson den neuen Ursprung in Position und Ausrichtung definiert und alle weiteren erkannten Personen relativ zu dieser transformiert sind. Dadurch bleiben die relativen Abstände zwischen den Schlüsselpunkten anderer Personen und der Referenzperson erhalten. Das Ergebnis ist, dass der Ursprung und Ausrichtung des auf der Referenzperson basierenden Koordinaten-Systems und des Unity-Weltkoordinatensystems aneinander angeglichen wurden. Die Referenzperson befindet sich praktisch im Nullpunkt der Unity-Welt und ist in Richtung der Tiefenachse orientiert. Durch die Entfernung der Tiefe aller Schlüsselpunkte werden diese abschließend für zweidimensionale Anwendungen auf der XY-Ebene vorbereitet. Die erfassten Kalibrierungsdaten werden im Verlauf verwendet, um die vom ZED-System bereitgestellten dreidi-

mensionalen Schlüsselpunkte kontinuierlich in, entsprechend dem genannten Prozess, transformierte zweidimensionale Schlüsselpunkte zu überführen und anderen interaktiven Komponenten eventbasiert bereitzustellen. Dabei ermöglicht die Komponente, dass Nutzende spontan das daraus resultierende interaktive System benutzen können, da neue Personen zur Laufzeit erkannt und überführt werden bzw. bei Verlust der Erkennung auch wieder entfernt werden.

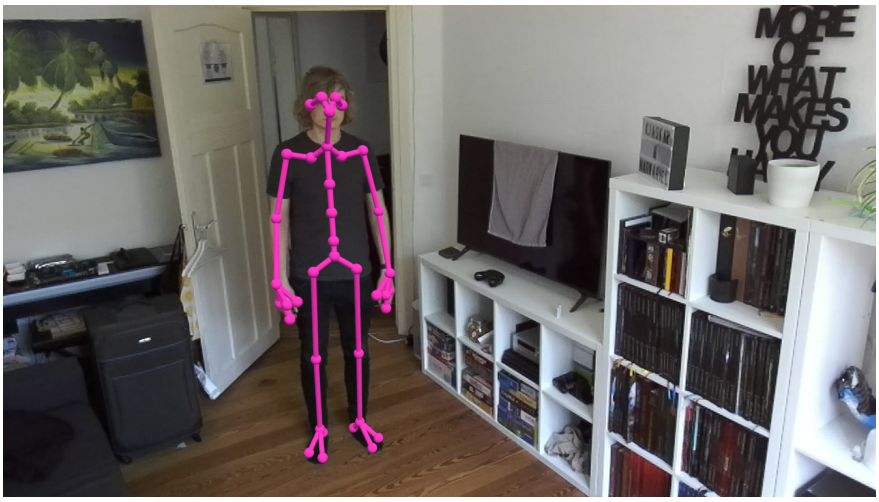
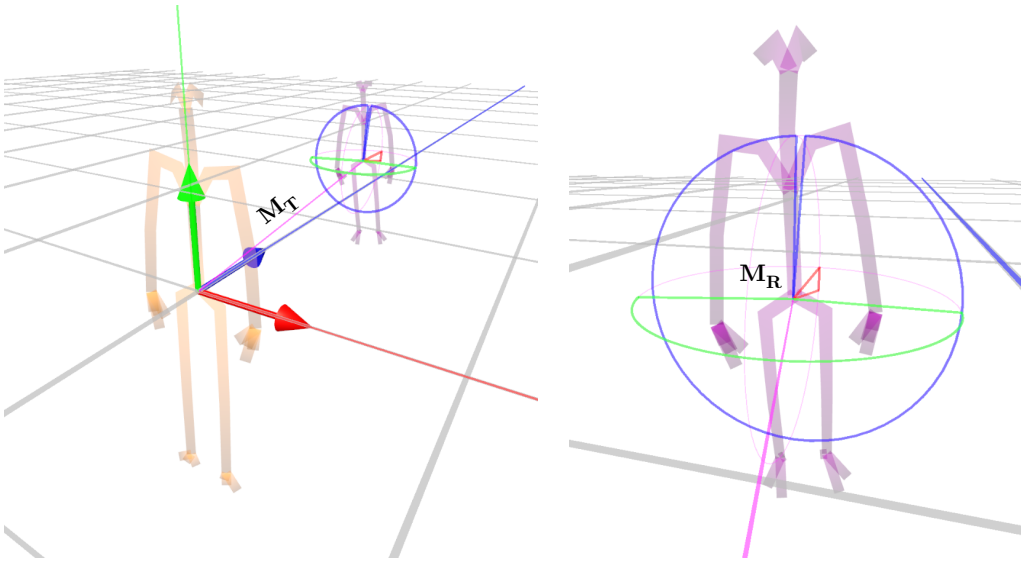


Abbildung 5.1: Referenzperson aus Sicht des ZED-System



(a) Kalibrierte Translation

(b) Kalibrierte Rotation

Abbildung 5.2: Kalibrierung durch Referenzperson (siehe Abbildung 5.1)

Die von der `BodyTrackingManager2D`-Komponente bereitgestellten Körperbewegungsdaten können damit als Grundlage für die Entwicklung interaktiver Komponenten für zweidimensionale Anwendungen verwendet werden. Dabei kann ein Entwickler einer solchen Komponente erwarten, dass eine erkannte Person, welche sich relativ zum Kalibrierungspunkt befindet, sich damit relativ zum Nullpunkt der Unity-Welt bewegt. Dies ermöglicht die Planung der Anwendung bzw. der Welt um den Nullpunkt herum auf der XY-Ebene. Die überführten Schlüsselpunkte können damit für verschiedenste Interaktionsszenarien verwendet werden. Zur besseren Anpassung an bestimmte Interaktionen stehen die Schlüsselpunkte in verschiedenen Detailgraden zur Verfügung (Körperformate) und können als Ganzes skaliert bzw. verschoben werden. Letzteres ermöglicht auch eine Anpassung an bestimmte Realgrößenverhältnisse des Interaktionssystems (insbesondere die Bildschirmgröße). Körperbewegungsdaten von Personen, die den Erkennungsbereich zur Laufzeit betreten oder verlassen, werden dabei hinzugefügt oder entfernt.

5.1.1 Visualisierung auf Basis von `LineRenderer`

Um das Ergebnis erstmalig nach der Umwandlung zu visualisieren, wurde eine Testkomponente (`BodyTrackingDebugVisualizer2D`) entwickelt, welche die überführten Körperdaten des `BodyTrackingManager2D` für Testzwecke direkt visualisiert. Diese erstellt und aktualisiert auf Basis der erhaltenen Daten die Personen bzw. ihre zugeordneten Schlüsselpunkte zur Laufzeit in Form einer vereinfachten schematischen Darstellung (Strichfigur) auf Basis von Unitys `LineRenderer`-Komponenten. Für die grundlegende Interaktion mit Objekten in der Unity-Szene wurde auch ein Collider `LineCollider2D` implementiert.

Ein erstes Beispiel auf Basis der beiden Komponenten zeigt die Repräsentation einer erkannten Person, welche durch ihre Position bzw. Bewegung mit dem Hintergrund interagieren kann bzw. diesen je nach Handposition zellenweise aufleuchten lassen kann. Dafür wurde zunächst der Kalibrierungsprozess durchgeführt, indem die Referenzperson sich vor dem Bildschirm mittig und gerade zu diesem positioniert und orientiert hat. Anschließend ist eine Interaktion relativ zum Bildschirm möglich.

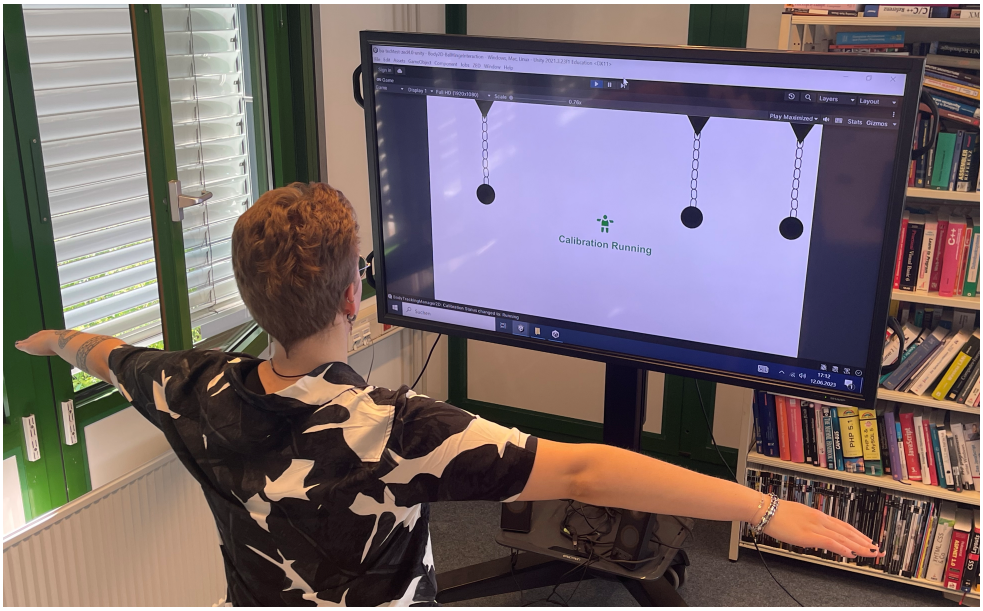


Abbildung 5.3: Kalibrierungsprozess in der Realwelt

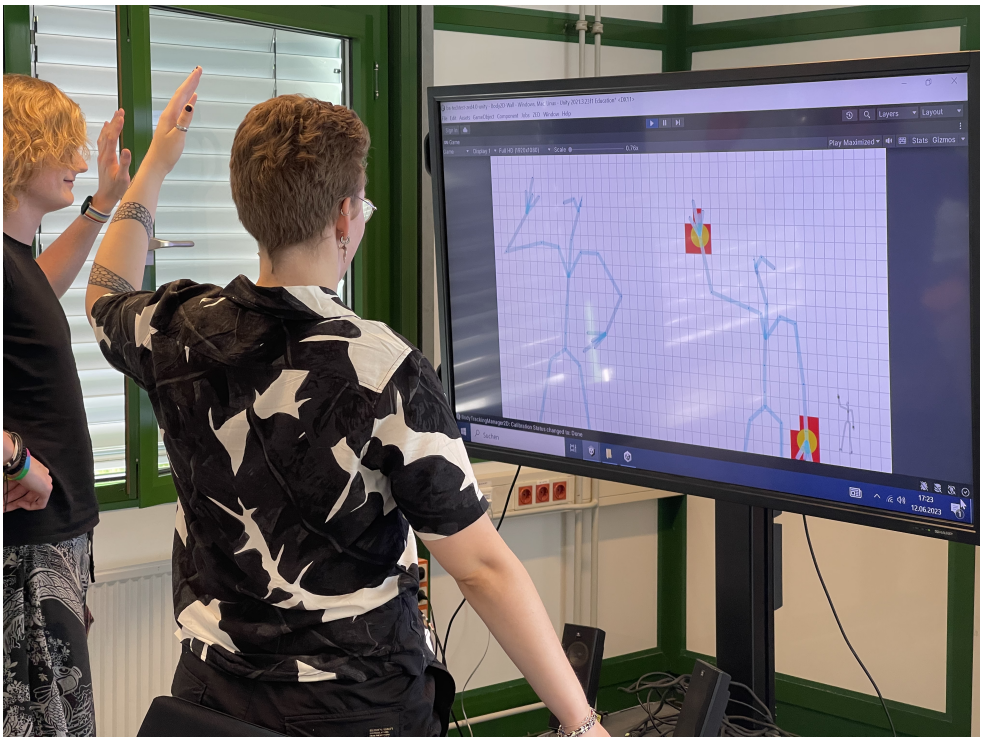


Abbildung 5.4: Mehrere Personen interagieren mit Wand-Szene

Die Ausrichtung bzw. Position der Person wurde durch die Kalibrierung korrekt übernommen und wird nun kontinuierlich auf erkannte Personen angewendet. Für die Szene werden die überführten Datenpunkte skaliert, um die Interaktion an die Bildschirmgröße und die primäre Interaktion im Oberkörperbereich anzupassen.

Als weiteres Beispiel wurde eine kollisionsbasierte Interaktion erschaffen, bei welcher hängende Kugeln durch virtuelle Berührung angeschoben werden können. Die Unity2D-Physik-Engine weist allerdings Limitierungen bei der Umsetzung rein positionsbasierter Änderungen von Objekten bzw. Körpern in der Szene auf, weswegen die physikalische Interaktion mit den Kugeln nur eingeschränkt funktionierte.

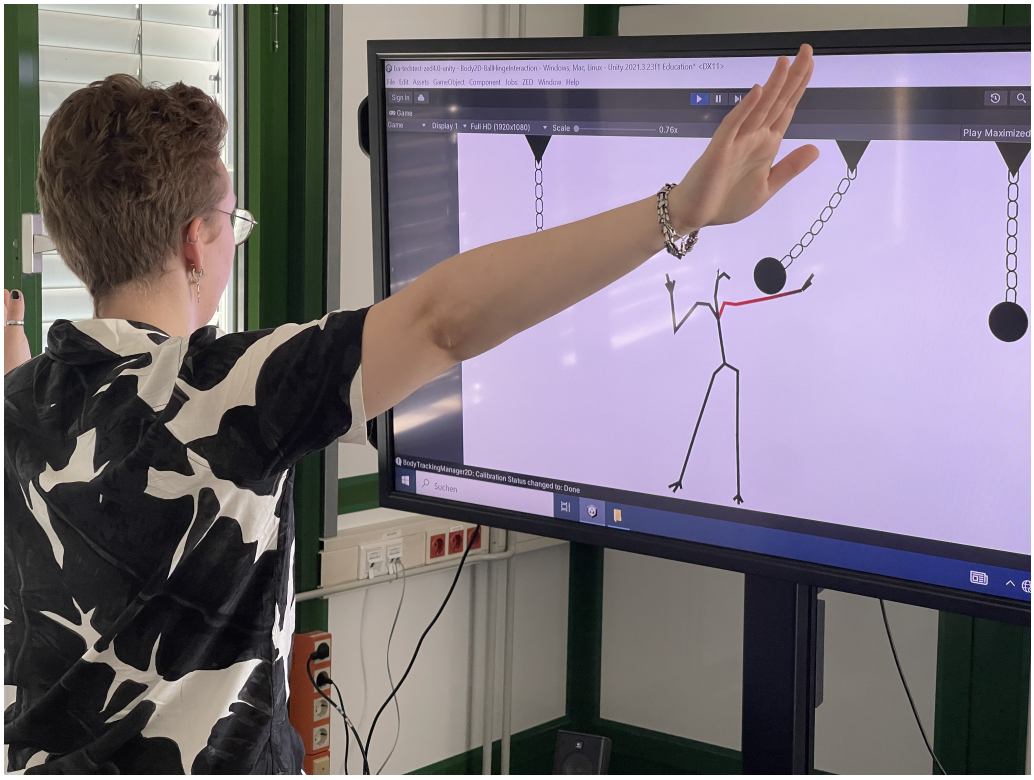


Abbildung 5.5: Person interagiert mit Ball-Szene

5.2 Interaktionsprototyp »Vektorfelder«

Die implementierte technische Grundlage wurde zur Schaffung des Interaktionsprototyps verwendet. Dabei wurde eine Herangehensweise gewählt, welche direkt auf dieser Grundlage aufsetzt und die Position von überführten Schlüsselpunkten verwendet und die notwendigen Anforderungen umsetzt (siehe Abschnitt 4.2).

Die Idee für den Interaktionsprototyp basiert dabei auf durch den Nutzenden beeinflussbaren Vektorfeldern bzw. Partikelströmen auf einer gegenüberliegenden Bildschirmfläche. Dafür wurde eine Komponente mit einem Datenmodell zur Repräsentation von begrenzten diskreten Vektorfeldern und ihren Manipulationen erschaffen. Die Komponente kann dabei ein parametrisierbares Basisvektorfeld durch Rauschfunktionen generieren und dieses in der Unity-Welt visualisieren. Es unterstützt dabei verschiedene Anpassungsoptionen betreffend Größe, Parameter der Rauschfunktion und weitere Verhaltensoptionen (siehe Abbildung 4.26).

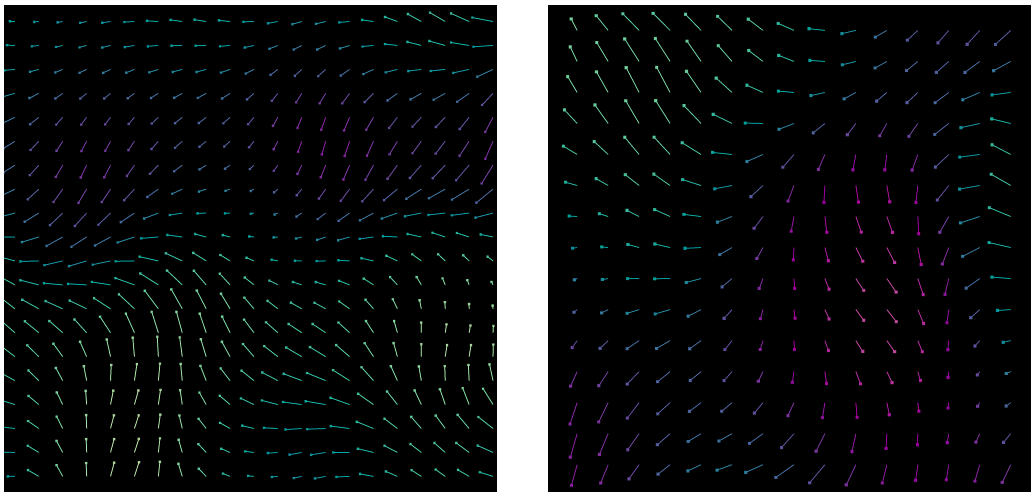


Abbildung 5.6: Vektorfelder mit verschiedenen Parametern

Die Komponente wurde anschließend so erweitert, dass das Vektorfeld durch multiple Manipulator-Objekte mit verschiedenen Manipulationsmodi in Echtzeit beeinflusst werden kann. Das Vektorfeld kehrt an nicht mehr manipulierten Bereichen langsam zurück zum Basisvektorfeld. Diese Manipulator-Objekte bilden die Grundlage der Interaktion.

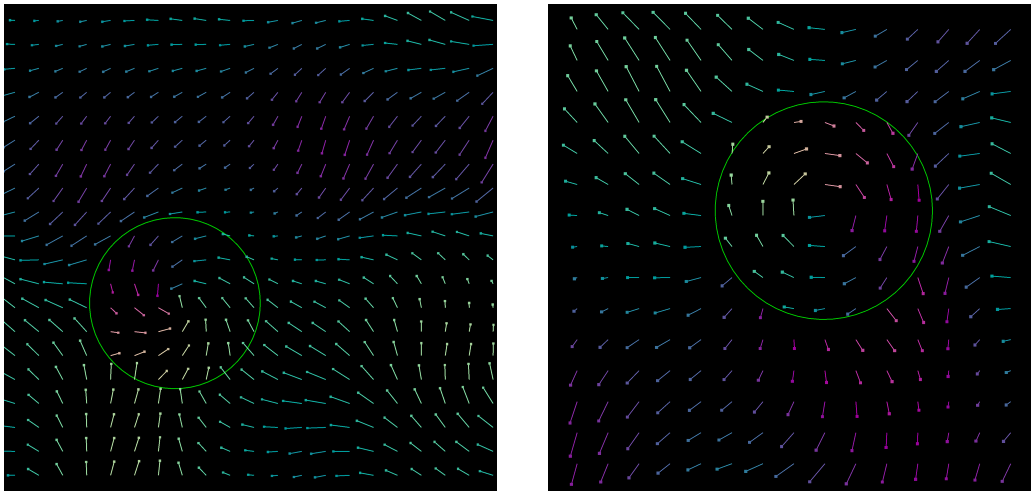
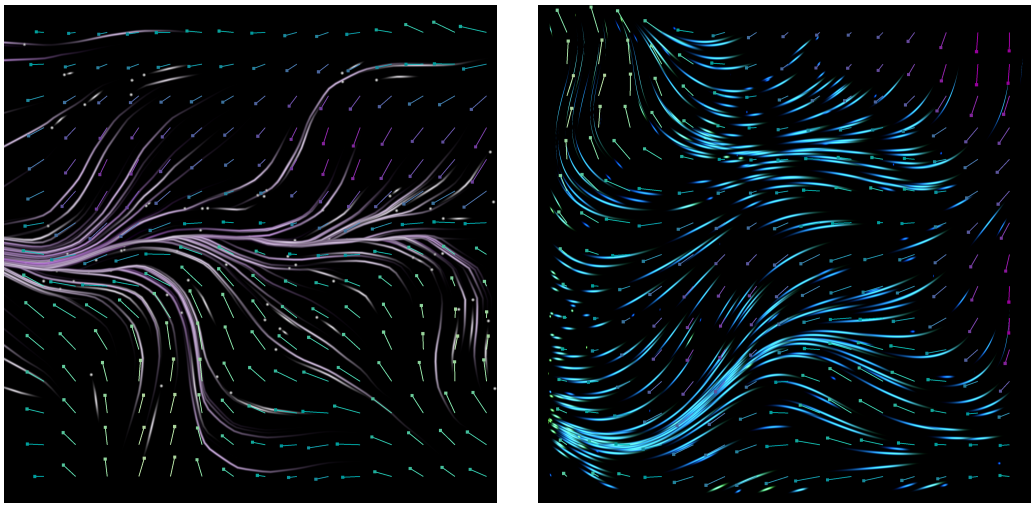


Abbildung 5.7: Manipulationen der Vektorfelder

Damit das entstandene dynamische Vektorfeld in Kombination mit den in Unity integrierten Partikelsystemen genutzt werden kann, wurde eine Prozedur geschaffen, welche das Vektorfeld zur Laufzeit in eine Volumentextur (`Texture3D`) umwandelt. Diese kann auf jeweils auf verschiedene Art und Weise in den beiden Hauptpartikelsystemen (»huriken Particle System« und VFX-Graph) von Unity verwendet werden.



(a) »Shuriken Particle System« (b) VFX-Graph

Abbildung 5.8: Partikelfluss auf Vektorfeldern

Für den eigentlichen Interaktionsprototyp wurde nur der VFX-Graph verwendet, da nur dieser auf Veränderungen in der Volumentextur zur Laufzeit reagiert.

Dadurch ist mindestens eines der in Unity integrierten Partikelsysteme vollumfänglich für den Interaktionsprototyp nutzbar.

Abschließend wurde die Vektorfeldkomponente an die Interaktionskomponente angebunden. Dabei werden für die Handpositionen einer jeden Person im Erkennungsbereich zur Laufzeit Manipulatoren mit zufälligen Manipulationsmodi erzeugt und folgen anschließend deren Handbewegungen.

Abbildung 5.9: Person interagiert mit Vektorfeld

Abbildung 5.10: Mehrere Personen interagieren mit Vektorfeld

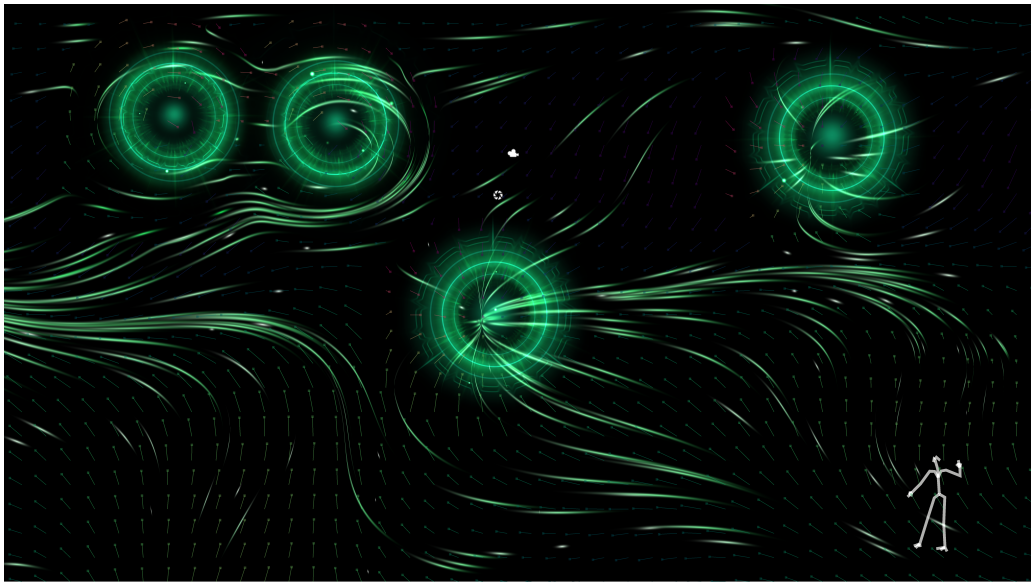


Abbildung 5.11: Screenshot von Interaktion auf Abbildung 5.10

5.3 Diskussion

Das Ergebnis (insbesondere die `BodyTrackingManager2D`-Komponente) bildet eine funktionierende Basis für das übergeordnete Ziel dieser Arbeit. Dabei sollte ein immersiver Ansatz entwickelt werden, damit verschiedene Personen ohne individuelle Vorbereitung direkt mit einem virtuellen System interagieren können. Im Fall dieser Arbeit handelte es sich im Speziellen um die Interaktion mit Anwendungen auf einem gegenüberliegenden Bildschirm, was insbesondere für interaktive Ausstellungsstücke auf Messen interessant sein könnte.

Das ZED-System diente dafür als technische Grundlage für den Empfang von Erkennungsdaten in Unity. Es ist durchaus in der Lage, dabei stabile Ergebnisse bei der Erkennung von Personen bzw. ihren Schlüsselpunkten im Raum zu liefern, sofern das verbundene Hostgerät die dafür notwendige Leistung insbesondere bei der GPU bereitstellen kann und die Tiefenkamera so ausgerichtet ist, dass sie die Personen optimal erfassen kann. Während der Entwicklung ist zusätzlich aufgefallen, dass für die Weiterführung definitiv eine potentere Hardware über den empfohlenen Anforderungen des Herstellers vorhanden sein sollte, da dies sich ansonsten stark auf die Erkennungsgeschwindigkeit und -genauigkeit auswirkt. Dies führte während der Arbeit zu einigen Herausforderungen bei praktischen Tests für die Implementierung. Da es sich um ein rein optisches System handelt, kommt es in ungünstigen Situationen (betrifft insbesondere Platzverhältnisse, Beleuchtung, Schatten und/oder Ausrichtung der Tiefenkamera) häufiger zu Erkennungsfehlern, welche das ZED-System nicht mehr selbst ausgleichen kann. Verbesserungen durch Aktualisierungen des SDKs sind zu erwarten, aber für einen Messebetrieb wäre beispielsweise trotzdem für die zukünftige Weiterentwicklung ratsam, die Erkennungsdaten mehrerer Tiefenkameras zu fusionieren, um Messungenauigkeiten ausgleichen zu können. Dies ist mit aktuellem Stand des ZED-Systems grundsätzlich vorbereitet, müsste allerdings für die Verwendung in Unity portiert werden und erfordert leistungsstarke Hardware (linearer Anstieg im Ressourcenbedarf, insbesondere im Bereich USB-Bus-Bandbreite sowie GPU-Speicher). Zusätzlich könnte man das Konzept auch mit anderen Erkennungssystemen, welche zum Beispiel auf dedizierte Tiefensensoren zurückgreifen, evaluieren. Alternativ könnte auch die Findung von Verfahren zur nachträglichen Korrektur der Erkennungsdaten interessant sein. Zur Verwendung des ZED-Systems musste eine Kalibrierungs- bzw. Transformationskomponente (`BodyTrackingManager2D`-Komponente) geschaffen werden, um die perspektivischen dreidimensionalen Körperdaten für das Interaktionsziel anzupassen. Dies ermöglicht, dass die Positionsdaten der Schlüsselpunkte von erkannten Personen als Grundlage zur Eingabe in zweidimensionale Interaktionskomponenten verwendet werden können. Im aktuellen Stand wird, zur Vorbereitung des Transformationsprozesses, die vom ZED-System als erste erkannte Person als Referenzperson für die Kalibrierung benutzt. Da dieser Prozess bei jedem neuen Aufbau des Interaktionssystems (insbesondere der Aufstellung des Bildschirms und der ZED-Kamera) neu durchgeführt werden muss, wurde eine

Möglichkeit implementiert, die Kalibrierung zu persistieren und bei gleichem Aufbau wiederzuverwenden. Die aktuelle Art der Kalibrierung erfordert dabei, dass im Idealfall zur Zeit der Kalibrierung keine weitere Person im Erkennungsbereich steht. Dies ist unter Umständen bei Messen nicht immer möglich, daher wäre es empfehlenswert, die `BodyTrackingManager2D`-Komponente um ein Verfahren zu erweitern, sodass das Kalibrierungsziel bzw. die Referenzperson zur Laufzeit bestimmt werden kann, wenn mehrere Personen im Erkennungsbereich festgestellt wurden. Das Konzept des Kalibrierungsprozess ist technisch prinzipiell funktionsfähig, kann aber noch in der Verwendung verbessert werden. Insbesondere wäre eine Evaluation sinnvoll, die sich mit der Praxistauglichkeit des Kalibrierungsvorgangs befasst und Möglichkeiten zur Verbesserung für die spontane Nutzung auf Messen untersucht.

Nach der Kalibrierung werden kontinuierlich überführte Schlüsselpunkte separat als Liste für jede erkannte Person über Events bereitgestellt, damit andere Komponenten diese verwenden können. Die Referenzperson bildet dabei den Ursprung und alle weiteren erkannten Personen befinden sich relativ zu ihr bzw. dem Ursprung auf der XY-Ebene. Dafür besteht des Weiteren die Möglichkeit, Offsets zu definieren, welche eine zusätzliche Verschiebung bzw. Skalierung aller Schlüsselpunkte ermöglichen, um diese zum Beispiel besser an die gewünschte Szene oder Interaktion anzupassen. Verschiedene Varianten der Schlüsselpunkte pro Person sorgen dabei dafür, dass sowohl Schlüsselpunkte mit oder ohne Anpassung durch Offsets zeitgleich zur Verfügung stehen, als auch Schlüsselpunkte relativ zur einer jeden Person selbst. Letzteres ermöglicht auch theoretisch die reine Interpretation von Posen unabhängig von der Position einer Person.

Insgesamt stehen damit aktuell zweidimensionale Positionsdaten der Schlüsselpunkte zur Verfügung. Dadurch sind einfache positionsbasierte Interaktionskonzepte möglich. Die Dimensionen der Interaktionen sind damit allerdings noch eingeschränkt. Sie bieten aber eine Ausgangslage für komplexere interaktive Komponenten, welche die Positionsdaten des Skeletts nicht nur verwenden, sondern auch interpretieren und damit tiefere Interaktionen ermöglichen. Beispiele dafür wären das Auslösen von Aktionen durch Interpretation von Gesten. Das können zum Beispiel Wischgesten oder ähnliches sein oder die Interpretation von Handgesten zur Initiierung von Aktionen. Eine weitere Möglichkeit besteht darin die empfangenen Positionsdaten für eine Nutzung mit virtuellen physikalischen Objekten in der Unity-Welt zu interpretieren. Damit wären beispielsweise durch den Interagierenden nachvollziehbare kollisionsbasierte Interaktion durch eine Repräsentation als Figur möglich.

Aufgrund der Natur von zweidimensionalen Anwendungen und des angewendeten Verfahrens ist dabei die Größe der interagierenden Personen entfernungsunabhängig. Das bedeutet, dass sich die relative Größe einer Person nicht verändert, wenn diese ihren Abstand zum Monitor verändert. Für viele Anwendungsfälle ist das ausreichend, da der Bezug der Interaktion anhand der korrespondierenden Bewegungen von Körper und virtueller Repräsentation herstellbar ist. Allerdings

ist es denkbar, dass es auch Anwendungsfälle gibt, bei denen ein simulierter Tiefeneindruck durch dynamische Skalierung jeder einzelnen Person sinnvoll ist. Eine mögliche Herangehensweise wäre beispielsweise, dafür ihren Abstand zum Bildschirm zu berücksichtigen.

Die Erkennungsreichweite des ZED-Systems beträgt ca. 8 bis 15 Meter.[21] Je nach Anwendung ist es nicht unbedingt gewünscht, dass jede Person im Erfassungsbereich berücksichtigt wird. Aktuell ist es möglich, die Anzahl der Personen seitens der jeweiligen interaktiven Komponente zu begrenzen und nur bei Verlust einer Person zu ersetzen. Das ist allerdings nicht für jeden Anwendungszweck ausreichend. In manchen Situationen ist es zum Beispiel sinnvoll den Erkennungsbereich einzuschränken, wenn womöglich vorbeilaufende Personen nicht direkt Teil der Interaktion werden sollen. In diesem Kontext wäre auch die alternative oder zusätzliche Evaluation von Verfahren zur Aufmerksamkeitserkennung für weitere Entwicklungen beachtenswert. Im Zuge solcher Verfahren wäre unter anderem dann möglich, festzustellen, ob eine erkannte Person wirklich mit dem Interaktionssystem interagieren möchte oder nur ein Zuschauer ist bzw. zufällig vorbeikommt.

Unter Berücksichtigung der bisherigen Diskussionsgegenstände ist es sinnvoll, zu prüfen, wie das Ausgabedatenmodell `Body2D` der `BodyTrackingManager2D`-Komponente um zusätzliche überführte oder angepasste Informationen zu jeder Person erweitert werden kann. Derzeit ist das Modell auf interpretierbare Daten zu den Positionen der Schlüsselpunkte in verschiedenen Variationen beschränkt. Dies bietet eine solide Basis für darauf aufsetzende interaktive Komponenten. Es könnten aber Ergänzungen vorgenommen werden, beispielsweise durch die Hinzufügung von dreidimensionalen Orientierungs- oder Abstandsdaten zum Kalibrierungspunkt bzw. zum Bildschirm (zum Beispiel für Konzepte zur Aufmerksamkeitserkennung) sowie anderen Daten, die bei der Interpretation als Eingaben hilfreich sein könnten.

Mit dem Interaktionsprototyp wurde durch manipulierbare Vektorfelder beispielhaft veranschaulicht, wie eine rein positionsbasierte Interaktion auf Basis der entwickelten `BodyTrackingManager2D`-Komponente aussehen könnte. Die interagierenden Personen können dabei mit ihren Händen ein Manipulator-Objekt positionieren. Dabei werden aktuell die Hände in der Szene visuell hervorgehoben, sowie die fließenden Partikel dargestellt. Für eine Weiterentwicklung des Prototyps sollte geprüft werden, inwiefern das visuelle Feedback der Interaktion insbesondere bei mehreren Personen ausreicht, um beispielsweise festzustellen, welche Manipulatoren zu welcher Person zugeordnet sind. Hinzukommt, dass der Partikelfluss zur Verdeutlichung als Spur angezeigt wird und dass dadurch eine minimale Trägheit bei der Reaktion der Partikel auftritt, auch wenn die Manipulation des Vektorfeldes längst erfolgt und angewendet wurde. Daher könnte man die Visualisierung des Feldes bzw. des Flusses prinzipiell verbessern, um ein rascheres Feedback zu erhalten und damit evtl. das Interaktionsgefühl zu verbessern.

[21] s. Stereolabs Inc., *ZED 2i - Datasheet (Jan. 2023)*, S. 3.

Ein schon genannter Punkt stellt auch die Einschränkungen durch die reine positionsbasierte Steuerung dar. Durch weitere Interpretationsmöglichkeiten von bestimmten Gesten, Posen oder ähnlichem wäre eine individuellere Steuerung der Manipulation möglich. Dies könnte beispielsweise der spontane Wechsel des Manipulationsmodus sein oder die Anpassung der Größe des Manipulationsfeldes.

Von der technischen Seite betrachtet, wurde ein eher imperativer Ansatz bei der Realisierung des Vektorfeldes verfolgt. Dies bietet einige Vorteile betreffend Performanz, da nur diskrete vorberechnete Werte des Feldes manipuliert werden müssen und jede vorberechnete Zelle direkt für die Generierung eines Voxelwerts der Volumentextur verwendet werden kann. Ein deklarativer Ansatz wäre die formelle Betrachtung des Vektorfeldes und seiner Manipulationen, für die Generierung der Volumentextur würden dann diskrete Punkte pro Voxel von der daraus resultierenden Funktion berechnet werden. Allerdings würde dies auch bedeuten, dass jede Manipulation eine komplette Neuberechnung des gesamten Feldes erforderlich machen würde und dadurch eine Aktualisierung der Volumentextur zur Folge hat. Das würde voraussichtlich die Implementierung simplifizieren, allerdings wäre dafür wahrscheinlich eine dedizierte Performanzoptimierung notwendig. Ein zusätzlicher Optimierungsansatz wäre die Überführung der Berechnungen in einen Shader zur Ausführung auf der GPU, da diese für parallele Berechnung optimiert ist. Im imperativen Ansatz, bei dem diskrete vorberechnete Werte des Vektorfeldes manipuliert werden, erfolgt die Berechnung der Volumentextur auf der CPU. Die CPU ist zwar für allgemeine Zwecke geeignet, aber sie ist nicht so effizient wie eine GPU, wenn es um Massenverarbeitung von Daten in parallelen Operationen geht. Auch in diesem Ansatz könnte die Generierung der Volumentextur auf die GPU ausgelagert werden. Insbesondere bei größeren Interaktionssystemen oder bei vielen interagierenden Personen können solche Optimierungen durchaus sinnvoll sein, da je nach Leistungsfähigkeit des Hostsystems ab einer gewissen Anzahl an Manipulatoren die Performanz sinkt.

Die gesamte Implementierung funktioniert mindestens für eine kleinere Anzahl von Personen im Erkennungsbereich. Insgesamt wäre noch zu überprüfen, ob und welche Optimierungen in funktionaler Natur oder der Performanz für die Nutzung mit einer erheblich größeren Personenanzahl notwendig wären. Es werden unter anderem an vielen Stellen Berechnungen auf größere Datenstrukturen angewendet, welche auch schon teilweise durch funktionale bzw. parallele Ansätze durchgeführt werden, jedoch noch auf weitere Bereiche in der Anwendung oder durch andere Berechnungsverfahren ausgeweitet werden könnten.

6. Fazit

Diese Arbeit befasst sich mit der Schaffung und Implementierung eines kamerabasierten Interaktionskonzepts für zweidimensionale virtuelle Welten bzw. Anwendungen. Ein Konzept wurde entwickelt, das eine direkte Interaktion mit einer zweidimensionalen Anwendung auf Basis eines NUI durch ein Erkennungssystem in der Entwicklungsumgebung Unity ermöglicht. Dafür wurde zunächst eine technische Grundlage in Form einer Unity-Komponente entwickelt, welche die Positionsdaten von Personen und ihrer Pose für eine zweidimensionale Nutzung aufbereitet und bereitstellt.

Zur Lösung dieser Problemstellung wurde zunächst das ZED-System als technologische Basis vorgestellt, das auf einer stereoskopischen Tiefenkamera und dem dazugehörigen SDK basiert. Eine Unity-Komponente `BodyTrackingManager2D` wurde implementiert, um die dreidimensionalen Körperdaten des ZED-Systems für die Nutzung in zweidimensionalen interaktiven Komponenten zu transformieren. Anschließend wurde ein Interaktionsprototyp in Form eines dynamischen Vektorfelds entwickelt, bei dem die Nutzenden den Partikelstrom durch ihre Handbewegungen beeinflussen können.

Die `BodyTrackingManager2D`-Komponente ermöglicht dafür eine kontinuierliche Transformation der Körperbewegungsdaten von erkannten Personen, sodass Entwickler interaktive Komponenten für zweidimensionale Anwendungen erstellen können. Die Komponente bietet einen Kalibrierungs- und Transformationsprozess, bei dem die Position und Ausrichtung einer Referenzperson relativ zum Bildschirm erfasst werden. Basierend auf diesen Kalibrierungsdaten werden die Schlüsselpunkte anderer erkannter Personen transformiert und anderen Komponenten zur Verfügung gestellt. Die überführten Körperbewegungsdaten können für verschiedenste Interaktionsszenarien verwendet werden und die Schlüsselpunkte stehen in verschiedenen Detailgraden bzw. Varianten zur Verfügung. Damit stellt die Komponente eine erste Grundlage für die Implementierung von weiteren interaktiven Komponenten auf Basis dieser Körperbewegungsdaten dar.

Der Interaktionsprototyp »Vektorfelder« basiert auf der entwickelten technischen Grundlage und ermöglicht es den Nutzenden, ein Vektorfeld auf einer Bildschirmfläche zu manipulieren. Das Vektorfeld wird durch Manipulator-Objekte beeinflusst, die den Partikelstrom des Feldes verändern. Die Komponente zur Darstellung des Vektorfeldes kann an verschiedene Einstellungen angepasst werden,

und das Feld kann in den in mindestens einem in Unity integrierten Partikelsystem verwendet werden.

Aufgrund des begrenzten Zeitrahmens, der einer Bachelorarbeit zur Verfügung steht, ist der Umfang dieser Arbeit naturgemäß beschränkt. So müsste die Skalierbarkeit des Konzepts für Messen noch weiter untersucht und ggf. optimiert werden; dies gilt auch für die Nutzbarkeit im Sinne der Benutzerfreundlichkeit (UX) sowie die in dem Zusammenhang fehlenden Funktionen, mit denen die Interaktion möglichst intuitiv und immersiv gestaltet werden kann. Des Weiteren könnten die für interaktive Komponenten bereitgestellten Körperdaten noch erweitert werden, da sie nur zweidimensionale Positionsdaten beinhalten; eine Interpretation dieser, zum Beispiel als Gesten, für erweiterte Arten der Interaktion wäre als Weiterentwicklung denkbar. Insbesondere für eine Interpretation ist aber die Zuverlässigkeit der Erkennungsdaten wichtig. Daher sollte geprüft werden, wie verlässlich die Erkennung ist und ob diese den Herausforderungen einer realen Messeumgebung, wie zum Beispiel schlechte Beleuchtung oder störende Objekte, gewachsen ist und inwiefern man dies ggf. verbessern kann. Insgesamt ermöglicht aber die entwickelte Lösung eine kamerabasierte Interaktion mit zweidimensionalen virtuellen Welten und Anwendungen. Der Interaktionsprototyp »Vektorfelder« zeigt dabei eine mögliche Art zur Nutzung dieser Technologie auf.

Die bisherigen Fortschritte bieten eine Grundlage für weitere Forschung und Entwicklung in diesem Bereich. Die vorgestellte Technologie des Erkennungssystems bietet zudem ein breites Spektrum an Erweiterungsmöglichkeiten, welche sicherlich erforscht werden sollten. Insbesondere für den Einsatz interaktiver Ausstellungsstücke lohnt sich eine weitere Betrachtung des Bereichs, da es Personen die Möglichkeit bietet, spontan in eine Interaktion eingebunden werden zu können, und damit Potenzial für interessante Weiterentwicklungen und kreative Anwendungen in der Zukunft aufzeigt.

Glossar

AR Augmented Reality

bspw. beispielsweise

bzw. beziehungsweise

CUDA (Compute Unified Device Architecture) Parallele Berechnungsarchitektur von NVIDIA durch Nutzung der Rechenleistung geeigneter Grafikprozessoren (Trademark von NVIDIA).

ID Identifikationsnummer

Inspektor (Unity Inspektor) Der Inspektor ist ein Werkzeug zum Anzeigen und Bearbeiten von Eigenschaften und Einstellungen für die meisten Bereiche des Unity-Editors, einschließlich GameObjects, Unity-Komponenten, Assets, Materialien sowie Einstellungen und Voreinstellungen im Editor.

KI Künstliche Intelligenz

NUI Natural User Interface

SDK Software Development Kit

Unity Unity ist eine Laufzeit- und Entwicklungsumgebung für Spiele des Unternehmens Unity Technologies mit Hauptsitz in San Francisco.

VFX-Graph Visual Effects Graph

VR Virtual Reality

ZED 2i Stereoskopische Tiefenkamera der Firma »StereoLabs«

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

Bruhns, Otto und Theodor Lehmann. *Elemente der Mechanik I: Einführung, Statik*. Studium Technik. Wiesbaden: Vieweg+Teubner Verlag, 1993. ISBN: 9783322928993. DOI: 10.1007/978-3-322-92899-3.

Cohen, Lanny u. a. *Augmented and Virtual Reality in Operations*. URL: https://www.capgemini.com/wp-content/uploads/2018/09/AR-VR-in-Operations.pdf (besucht am 17. 05. 2023).

Hidalgo, Ginés u. a. *OpenPose: PoseModel – Namespace Reference*. 22.04.2023. URL: https://cmu-perceptual-computing-lab.github.io/openpose/web/html/doc/namespaceop.html (besucht am 28. 05. 2023).

Neher, Markus. »Vektorfelder«. In: *Anschauliche Höhere Mathematik für Ingenieure und Naturwissenschaftler 2*. Springer Vieweg, Wiesbaden, 2018, S. 167–168. DOI: 10.1007/978-3-658-19422-2_10. URL: https://link.springer.com/chapter/10.1007/978-3-658-19422-2_10.

Sasse, Ralph. »Entfernungsmessende Verfahren«. In: *Bestimmung von Entfernungsbildern durch aktive stereoskopische Verfahren*. Vieweg+Teubner Verlag, Wiesbaden, 1994, S. 25–57. DOI: 10.1007/978-3-322-88814-3_3. URL: https://link.springer.com/chapter/10.1007/978-3-322-88814-3_3.

Stereolabs Inc. *Bodies – ZED SDK API Reference (C++)*. 17.05.2023. URL: https://www.stereolabs.com/docs/api/classsl_1_1Bodies.html (besucht am 26. 05. 2023).

— *Body Tracking Module – ZED SDK API Reference (C++)*. 17.05.2023. URL: https://www.stereolabs.com/docs/api/group__Body__group.html (besucht am 18. 05. 2023).

— *Body Tracking Overview*. 17.05.2023. URL: https://www.stereolabs.com/docs/body-tracking/ (besucht am 19. 05. 2023).

— *BodyData – ZED SDK API Reference (C++)*. 17.05.2023. URL: https://www.stereolabs.com/docs/api/classsl_1_1BodyData.html (besucht am 26. 05. 2023).

— *Unity and ZED*. 17.05.2023. URL: https://www.stereolabs.com/docs/unity/ (besucht am 19. 05. 2023).

— *Unity: Key Concepts & Scripts*. 17.05.2023. URL: https://www.stereolabs.com/docs/unity/basic-concepts/ (besucht am 19. 05. 2023).

— *Using the Body Tracking API*. 17.05.2023. URL: https://www.stereolabs.com/docs/body-tracking/using-body-tracking/ (besucht am 19. 05. 2023).

Stereolabs Inc. *ZED 2i - Datasheet (Jan. 2023)*. 31.01.2023. URL: https://cdn2.stereolabs.com/assets/datasheets/ZED%202i%20Datasheet%20Jan2023.pdf (besucht am 18. 05. 2023).

Torrence, Ann. »Martin Newell's original teapot«. In: *ACM SIGGRAPH 2006 Emerging technologies*. Hrsg. von John Finnegan. ACM Digital Library. New York, NY: ACM, 2006, S. 29. ISBN: 1595933646. DOI: 10.1145/1180098.1180128.

»Vektorfeld«. In: *Lexikon der Physik: Band 5*. Heidelberg und Heidelberg [u.a.]: Spektrum, Akad. Verl. und Spektrum- Verl., 2000. ISBN: 3827402638.