der Bundeswehr
Universität München

Integration einer Physik-Engine zur Steigerung der Nutzerfreundlichkeit eines halb-öffentlichen Wandbildschirms

Masterarbeit

Gerrit Grauwinkel
1184922

Erstprüfer:	Prof. Dr. Michael Koch
Zweitprüfer:	Prof. Dr.-Ing. Mark Minas
Betreuer:	Julian Fietkau
Abgabetermin:	30.06.2023

Universität der Bundeswehr München
Fakultät für Informatik

Kurzfassung

Die Gestaltung von nutzerfreundlichen User-Interfaces stellt einen wichtigen Aspekt im Forschungsfeld der Mensch-Computer-Interaktion dar. Im Bereich von Videospielen ist es etabliert, Physiksimulationen zu nutzen, um den empfundenen Realismus und den Spaßfaktor des Nutzers zu erhöhen. In dieser Arbeit soll untersucht werden, inwiefern die in der Spieleentwicklung genutzten Techniken zur Physiksimulation Auswirkungen auf die Nutzerfreundlichkeit eines User-Interfaces haben, wenn dieses außerhalb des Kontexts von Spielen angewendet wird. Dabei soll im Speziellen untersucht werden, wie eine Physiksimulation in ein bestehendes User-Interface integriert werden kann, welche Aspekte der Nutzerfreundlichkeit durch diese beeinflusst werden und welche Techniken der Physiksimulation dafür geeignet sind. Dazu wird eine Physik-Engine in die Software eines halb-öffentlichen Wandbildschirms integriert und anhand einer Datenerhebung zum Nutzerverhalten ausgewertet, welche Auswirkungen die physikalisch simulierten Elemente auf die Nutzerfreundlichkeit haben. Das Ergebnis dieser Arbeit besteht darin, dass die eingeführte Physiksimulation zu einem starken Anstieg der Interaktionen mit dem User-Interface geführt hat und die implementierte Physiksimulation eine positive Wirkung auf die Nutzerfreundlichkeit des User-Interfaces hat.

Inhaltsverzeichnis

1 Einleitung

Im Forschungsfeld der Mensch-Computer-Interaktion wird sich mit Fragen über kontextgerechte und nutzerfreundliche Gestaltung von IT-Systemen beschäftigt (Koch et al., 2020). Ein wichtiger Forschungsbereich der Mensch-Computer-Interaktion liegt bei der Gestaltung, Implementierung und Evaluation von User-Interfaces (Sinha et al., 2010). Physikalisch simulierte Elemente werden in der Gaming-Industrie schon seit langem verwendet und können ein erfolgsentscheidendes Element bei der Gestaltung von Spielen sein. Physikalische Gesetze werden in die Spielmechaniken integriert, um das Spiel realistischer wirken zu lassen und um Emotionen zu vermitteln (Kumar, 2023). Im Gegensatz zu Videospielen, die der Unterhaltung dienen, existiert eine Vielfalt an IT-Systemen, die zahlreiche Anwendungsgebiete abdecken und den Nutzer nicht primär unterhalten sollen. In dieser Arbeit soll untersucht werden, inwiefern die positive Wirkung physikalisch simulierter Elemente in Videospielen bei der Gestaltung von User-Interfaces übertragbar ist, wenn der Kontext nicht unterhaltungsbezogen ist.

Durch die Arbeitsgruppe für Mensch-Computer-Interaktion an der Universität der Bundeswehr in München werden sogenannte Community Mirrors betrieben, um Langzeitstudien durchzuführen und Gestaltungsempfehlungen zu erstellen. Diese Community Mirrors sind große interaktive Wandbildschirme, die ihren Nutzern Informationen darstellen (Kraus, 2022). Die Community Mirrors befinden sich auf dem Universitätsgelände und werden von Studenten, sowie Angestellten der Universität passiert. Da die Community Mirrors über ein interaktives User-Interface verfügen und sie nicht primär der Unterhaltung, sondern der Vermittlung von Informationen dienen, eignen sie sich als Testobjekt für diese Arbeit. Am Beispiel eines Community Mirrors soll untersucht werden, ob durch das Integrieren von physikalisch simulierten Elementen die Nutzerfreundlichkeit gesteigert werden kann. Im Spezifischen soll eruiert werden, welche Aspekte der Nutzerfreundlichkeit durch Physiksimulationen beeinflusst werden können, welche Techniken dazu geeignet sind und welche Wirkung die Integration einer Physiksimulation in ein User-Interface letztendlich erzielt.

1.1 Vorgehen

Zunächst sollen in Kapitel 2 durch eine Literaturrecherche theoretische Grundlagen für diese Arbeit gelegt werden. Dazu wird der Begriff der Physik-Engine eingeführt und erklärt, welchen Nutzen diese bei der Implementierung einer Physiksimulation hat. Anschließend werden einige Kernkonzepte vorgestellt, die bei der Simulation von physikalischen Elementen zum Einsatz

kommen. Darauf aufbauend wird erläutert, welche Aspekte der Nutzerfreundlichkeit eines User-Interfaces durch physikalisch simulierte Elemente beeinflusst werden können. Zusätzlich werden Beispiele genannt, durch welche Features der jeweilige Aspekt der Nutzerfreundlichkeit verbessert werden kann.

Nachdem die theoretischen Grundlagen gelegt wurden, sollen diese nun durch die Integration einer Physik-Engine in den Community Mirror in der Praxis Anwendung finden. Dazu wird in Kapitel 3 das User-Interface des Community Mirrors und seine für diese Arbeit relevanten Funktionalitäten beschrieben. Anschließend werden Use-Cases erstellt, die festlegen, welche physikalisch simulierten Elemente dem Community Mirror hinzugefügt und welche bereits bestehenden verbessert bzw. in die Physik-Engine integriert werden sollen.

Um die aufgestellten Use-Cases zu implementieren, wird in Kapitel 4 eine dafür geeignete Physik-Engine ausgewählt. Dabei werden mehrere potenzielle Möglichkeiten vorgestellt und sich auf Grundlage ausgewählter Kriterien für eine Physik-Engine entschieden. Anschließend wird erklärt, wie die Physik-Engine in die bestehende Software integriert werden soll, welche Änderungen an bestehenden Klassen vorgenommen werden und welche Klassen neu erstellt werden sollen. Um das Kapitel abzuschließen, wird anhand von ereignisgesteuerten-Prozessketten der Ablauf der Physiksimulation erklärt.

In Kapitel 5 wird die Implementierung der Physiksimulation beschrieben. Dabei wird auf die Entwicklungsgrundlagen eingegangen, der Programmaufbau vorgestellt, Herausforderungen und Schwierigkeiten bei der Umsetzung genannt und der Code der neuen Klassen erklärt.

In Kapitel 6 werden die vorgenommenen Änderungen evaluiert, um herauszufinden, welchen Einfluss das Integrieren einer Physiksimulation auf die Nutzerfreundlichkeit bei der Interaktion mit dem Community Mirror hat. Abschließend werden die in dieser Arbeit erlangten Erkenntnisse im Fazit zusammengefasst.

1.2 Verwandte wissenschaftliche Arbeiten

Der Einsatz von Physiksimulationen bei der Gestaltung von User-Interfaces wurde bereits in anderen Arbeiten behandelt. Kim und Park (2015) haben eine Interaktionstechnik untersucht, bei der die Hände des Nutzers zur intuitiven und realistischen Interaktion durch eine Physiksimulation simuliert werden. Dafür haben sie Modelle zur Verformung der Hände des Benutzers erstellt. Durch die Modelle wurde erreicht, dass ohne Definition von Interaktionsmethoden und lediglich durch Verwendung der Physiksimulation dem Nutzer ermöglicht wurde, intuitiv und präzise mit Objekten zu interagieren. McDirmid (2009) stellt Lösungen für das Problem vor, dass eine realistische physikalische Simulation die Bedienung eines User-Interfaces beeinträchtigen kann. Dabei stellt er als Lösung vor, an ausgewählten Stellen die Regeln der Physik zu brechen, um die Gesamtwirkung der Physiksimulation zu verbessern. Er nennt einige Techniken, die dazu geeignet sind und testet diese an einer Fallstudie. Er kommt zu dem Ergebnis, dass die von ihm genannten Techniken dazu geeignet sind, zu verhindern, dass der Realismus einer

Physiksimulation, die Bedienung von User-Interfaces beeinträchtigt. Abschließend weist er darauf hin, dass zukünftige wissenschaftliche Arbeiten untersuchen können, wie spielzentrierte Physik-Engine-Technologie im Kontext eines nicht spielzentrierten User-Interfaces genutzt werden kann. Liu und Zordan (2011) stellen ein Framework vor, um physikbasierte Charakter-Animationen mit einem Natural-User-Interface (User-Interface ohne künstliche Eingabegeräte) zu kombinieren. Dadurch soll es dem Nutzer ermöglicht werden, sich in den im User-Interface dargestellten Charakter hineinzuversetzen und mit ihm zu fühlen bzw. zu handeln. Die verwandten wissenschaftlichen Arbeiten zeigen, dass die Wirkung von Physiksimulationen bei der Gestaltung von User-Interfaces ein relevantes Thema ist und diese Arbeit an bereits gesammelte Erkenntnisse anknüpft, um die Wirkung einer Physiksimulation auf die Nutzerfreundlichkeit eines halb-öffentlichen Wandbildschirms zu bestimmen.

2 Grundlagen physikalischer Simulationen im Bezug auf User-Interfaces

In diesem Kapitel werden die theoretischen Grundlagen gelegt, auf deren Basis im nächsten Kapitel eine Anforderungsanalyse mit den Use-Cases für die Integration einer Physiksimulation im Community Mirror folgt. Zu Beginn wird erklärt, was eine Physik-Engine ist und welche Vorteile sie mit sich bringt. Anschließend werden die technischen Grundlagen zur Simulation physikalischer Systeme im Kontext der Softwareentwicklung vorgestellt. Abschließend wird erörtert, wie die Integration von Physiksimulationen die Nutzerfreundlichkeit eines User-Interface beeinflussen kann.

2.1 Physik-Engines

Als Physik-Engine lässt sich Software bezeichnen, die physikalische Systeme annähernd simuliert. Sie ist der Teil einer Anwendung, der den gesamten erforderlichen Code für die Simulation von physikalischen Systemen enthält (Bourg, 2002, S. 281). Besonders ausgeprägt ist deren Einsatz im Bereich von Computerspielen und Film CGI. So nehmen Physik-Engines im Bereich der Computerspiele eine so wichtige Rolle ein, dass moderne Spiele nicht mehr ohne sie auskommen (Millington, 2007, S .1). Grundsätzlich ist die Simulation von physikalischen Eigenschaften schon vor dem weit verbreiteten Einsatz von Physik-Engines gängig gewesen. Dabei wurden jedoch nur die spezifisch für eine Anwendung benötigten Eigenschaften, wie beispielsweise die Flugbahn eines Pfeils, simuliert. Bei einer Physik-Engine hingegen, wird vergleichsweise generisch das physikalische Verhalten von Objekten beschrieben. So könnte, um auf das Beispiel zurückzukommen, nicht nur die Flugbahn eines Pfeils, sondern von Projektilen generell simuliert werden. Der Entwickler müsste dazu nur die entsprechenden Details auf das zu simulierende Objekt anpassen. (Millington, 2007, S. 2 f.) Aufgrund dieser Eigenschaft sind Physik-Engines für verschiedene Projekte, teilweise auch außerhalb ihres angedachten Anwendungsbereiches verwendbar. Somit ermöglicht eine Physik-Engines dem Entwickler Zeit und Aufwand zu sparen, da er sich über die technischen Aspekte einer physikalischen Simulation weniger Gedanken machen muss (Baba et al., 2007). Um Physik Simulationen in ein Projekt zu integrieren, bestehen grundsätzlich drei Möglichkeiten. Leistungsstarke, kommerzielle Physik-Engines können in Form einer Middleware von Drittanbietern bezogen werden (Millington, 2007, S. 1). Des Weiteren gibt es zahlreiche Open-Source-Lösungen, von denen in Abschnitt 4.1 einige vorgestellt werden.

Die letzte Möglichkeit besteht darin eine eigene Engine, angepasst an einen spezifischen Bedarf zu entwickeln.

2.2 Techniken zur physikalischen Simulation

Im Folgenden werden einige für diese Arbeit relevanten Techniken und Elemente vorgestellt, mit denen Engines Physiksimulationen durchführen. Dazu zählen Rigid-Body-Dynamics, verschiedene Kräfte, die auf einen Körper einwirken können, Kollisionen, Federn und der Physik-Engine-Loop.

2.2.1 Rigid-Body-Dynamics

Rigid-Body-Dynamics (Starrkörpersimulation) wird dazu verwendet, Bewegungen und Kollisionen von starren Objekten in einer physikalischen Umgebung zu simulieren. Sie basiert auf Gesetzen der klassischen Mechanik und behandelt Objekte als starre Körper, die sich nicht verformen können. Rigid-Body-Dynamics werden sowohl im zwei- als auch dreidimensionalen Bereich angewendet (Bourg, 2002, S. 189 ff.). Die Positionen und Bewegungen der Körper werden durch ein Koordinatensystem und entsprechende Vektoren abgebildet. Außerdem können auf die Körper einwirkende Kräfte durch Vektoren simuliert werden, wodurch Bewegung, Beschleunigung, Rotation und das Abbremsen dieser ermöglicht wird (Bourg, 2002, S. 62 ff.).

2.2.2 Kräfte und Drehmoment

Kräfte werden benötigt, um durch Körper simulierte Objekte zu beschleunigen und damit zu bewegen. Grundsätzlich lassen sich Kräfte in die Kategorien Kontaktkräfte und Feldkräfte einteilen (Bourg, 2002, S. 71 f.) Zu den Kontaktkräften gehören alle Kräfte, die durch Berührungen verursacht werden. Beispiele aus dem Alltag sind etwa das Schießen eines Balles oder das Hochheben eines Gegenstandes. Auch durch Reibung entstehende Kräfte sind den Kontaktkräften zuzuordnen. Feldkräfte hingegen können auf einen Körper einwirken, ohne diesen zu berühren. Als Beispiele für die Feldkräfte lassen sich Gravitation und elektromagnetische Kräfte nennen (Bourg, 2002, S. 71 f.). Während die aufgezählten Kräfte für die lineare Bewegung eines Körpers verantwortlich sind, wird durch das Drehmoment die Rotation eines Körpers verursacht. Sowohl Kräfte als auch Drehmoment werden durch Vektoren dargestellt. Der Vektor von Kräften verläuft von dem Punkt, an dem sie entstehen, in die Richtung, in die sie wirken. Der Vektor vom Drehmoment befindet sich hingegen an der Drehachse eines Körpers (Bourg, 2002, S. 80 f.).

In Abbildung 1 wird die Wirkung einer Feldkraft in Form von Gravitation und Drehmoment auf einen Körper in einer zweidimensionalen Umgebung dargestellt. Man sieht, wie sich der Körper in Richtung des Vektors der Gravitation bewegt und entlang seiner Drehachse in die durch das Drehmoment vorgegebene Richtung rotiert.

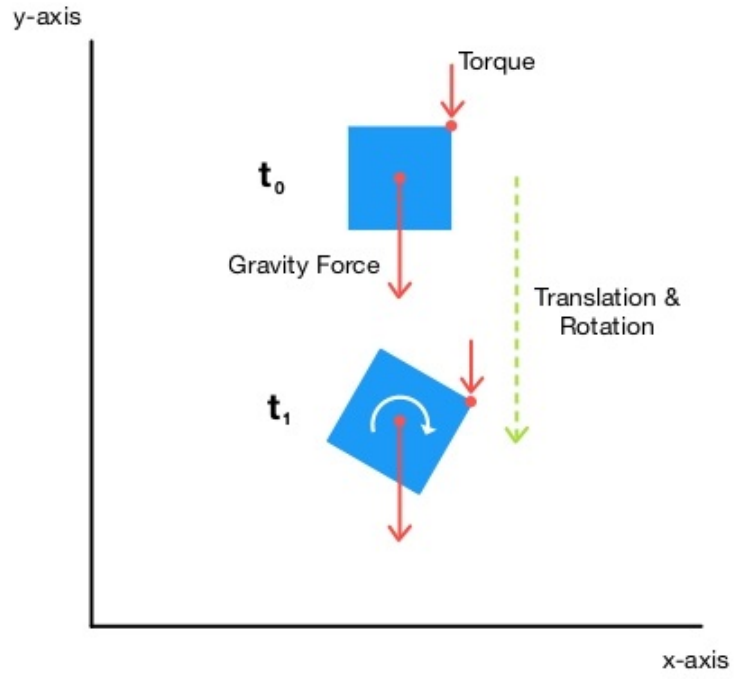

Abbildung 1: Wirkung von Kräften und Drehmoment auf einen Körper (Serrano, 2017).

2.2.3 Kollisionen

Eine Physik-Engine ermöglicht es, Kollisionen zwischen Körpern zu erkennen und davon ausgehende Reaktionen einzuleiten. Bei der Kollisionserkennung handelt es sich um ein geometrisches Problem, bei dem bestimmt wird, ob und wo zwei Körper miteinander kollidieren. Bei der Kollisionsreaktion wird berechnet, welche kinetischen Bewegungen von einer Kollision ausgehen (Bourg, 2002, S. 103). Bei der Kollisionserkennung werden geometrische Formen wie Kreise und Polygonen um den zu simulierenden Körper gezogen. Wenn die um einen Körper gezogene geometrische Form die eines anderen berührt, wird dies im Regelfall von der Engine als Kollision gewertet (Serrano, 2017). In Abbildung 2 sind Möglichkeiten abgebildet, wie die geometrischen Formen um einen Körper aussehen könnten. Je genauer eine Form an den von ihr repräsentierten Körper angepasst wird, desto akkurater können Kollisionen und die daraus entstehenden Kräfte bestimmt werden. Einfachere geometrische Formen wie beispielsweise Kreise ermöglichen hingegen eine schnellere Berechnung von möglichen Kollisionen (Serrano, 2017).

Bei der auf einer Kollisionserkennung folgenden Kollisionsreaktion werden anhand von Daten wie der Masse der aufeinander treffenden Körper, den Kollisionspunkten und der Aufschlagsgeschwindigkeit, die aus der Kollision entstehenden Kräfte berechnet (Serrano, 2017).

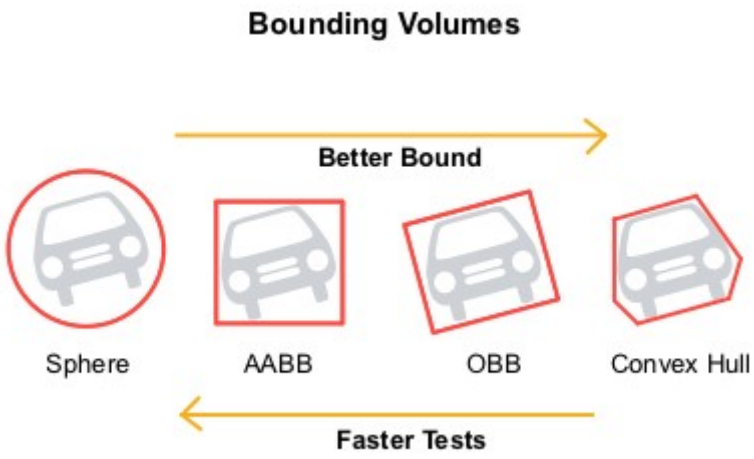

Abbildung 2: Kollisionserkennung durch geometrische Formen (Serrano, 2017).

2.2.4 Federn und Gelenke

Federn und Gelenke sind Strukturelemente, durch die zwei Körper miteinander verbunden werden können. Durch diese Strukturelemente werden Kräfte auf Körper erzeugt, um diese realistisch miteinander interagieren zu lassen. So können Ansammlungen von Körpern durch Federn elastisch zusammengehalten werden (Bourg, 2002, S. 79 f.). Physik-Engines bieten verschiedene Arten von Gelenken an. Durch diese ermöglicht eine Physik-Engine es, realistische Bewegungen wie Pendeln, Schwenken oder Drehen auf Körper anzuwenden. Beispiele dafür sind etwa Radgelenke, Schiebegelenke und Scharniergelenke (Catto, 2023). In Abbildung 3 ist exemplarisch ein Radgelenk dargestellt, wie es in der Physik-Engine Box2D implementiert ist. Dort sieht man zwei Körper, die durch eine Feder miteinander verbunden sind, damit sie zusammengehalten werden und am unteren Ende der Feder befindet sich eine Achse, durch die der untere Körper rotieren kann.

2.2.5 Physik-Engine-Loop

Die Aufgebe einer Physik-Engine besteht darin, Beschleunigung, Geschwindigkeiten und Verschiebungen eines Objekts auf Grundlage der auf einen Körper wirkenden Kräfte und Drehmoment zu berechnen. Der Physik-Engine-Loop stellt eine Endlosschleife (üblicherweise Update-Methode) dar, in der Positionen und Auswirkungen von den vorgestellten Elementen der Physiksimulation berechnet werden. Bei jedem Aufruf der Schleife werden Kollisionen erkannt, Reaktionen bestimmt und Feldkräfte wie die Gravitation einbezogen, um die Bewegungen von Körpern zu bestimmen (Serrano, 2017).

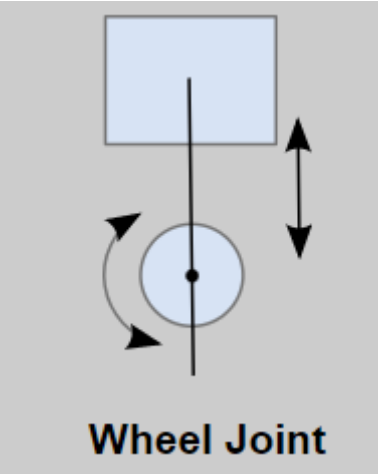

Abbildung 3: Radgelenk in Box2D (Catto, 2023).

2.3 Einfluss von Physik auf die Nutzerfreundlichkeit

In folgendem Abschnitt wird erläutert, inwiefern physikalische Simulationen die Nutzerfreundlichkeit beeinflussen können. Dabei wird ein besonderer Schwerpunkt auf die Nutzerfreundlichkeit von User-Interfaces gelegt. Dazu wird zunächst geklärt, wie Nutzerfreundlichkeit im Rahmen dieser Arbeit definiert ist und welche Elemente sie beinhaltet. Anschließend werden verschiedene Aspekte mit passenden Beispielen aufgezählt, durch die Physiksimulationen die Nutzerfreundlichkeit eines User-Interface aufwerten können. Abschließend wird noch ein möglicher Nachteil von schlecht umgesetzten Physiksimulationen erläutert.

2.3.1 Nutzerfreundlichkeit in der Mensch-Computer-Interaktion

Um zu prüfen, inwiefern durch das Integrieren einer Physiksimulation die Benutzerfreundlichkeit einer Software gesteigert werden kann, muss zunächst geklärt werden, aus welchen Bestandteilen die Benutzerfreundlichkeit im Sinne der Mensch-Computer-Interaktion besteht. Grundsätzlich ist in der Fachliteratur in Bezug auf Benutzerfreundlichkeit meistens der Begriff *Usability* im Gebrauch (Richter & Flückiger, 2013, S. 3f.). Beide Begriffe werden teilweise synonym verwendet. *Usability* bedeutet im Zusammenhang der Mensch-Computer-Interaktion, dass ein System vom Nutzer im Kontext der Aufgabenerfüllung einfach und effektiv zu bedienen ist (Richter, 1997, S. 4). Durch technologische Fortschritte in den letzten Jahrzehnten, existieren in einem Anwendungsbereich oft viele konkurrierende Softwareprodukte, welche alle die Anforderungen im Sinne der *Usability* erfüllen (Moser, 2012, S. 1). Deshalb reicht es nicht aus, wenn ein Produkt einfach und effektiv bei der Aufgabenerfüllung zu bedienen ist, sondern es muss sich zusätzlich durch andere Argumente von der Konkurrenz abgrenzen. Eine Möglichkeit besteht darin, positive Erlebnisse bzw. Emotionen bei der Nutzung auszulösen. Da positive Emotionen Glücksgefühle auslösen, wird der Nutzer dieses Erlebnis wiederholen wollen, wohingegen er negative Emotionen

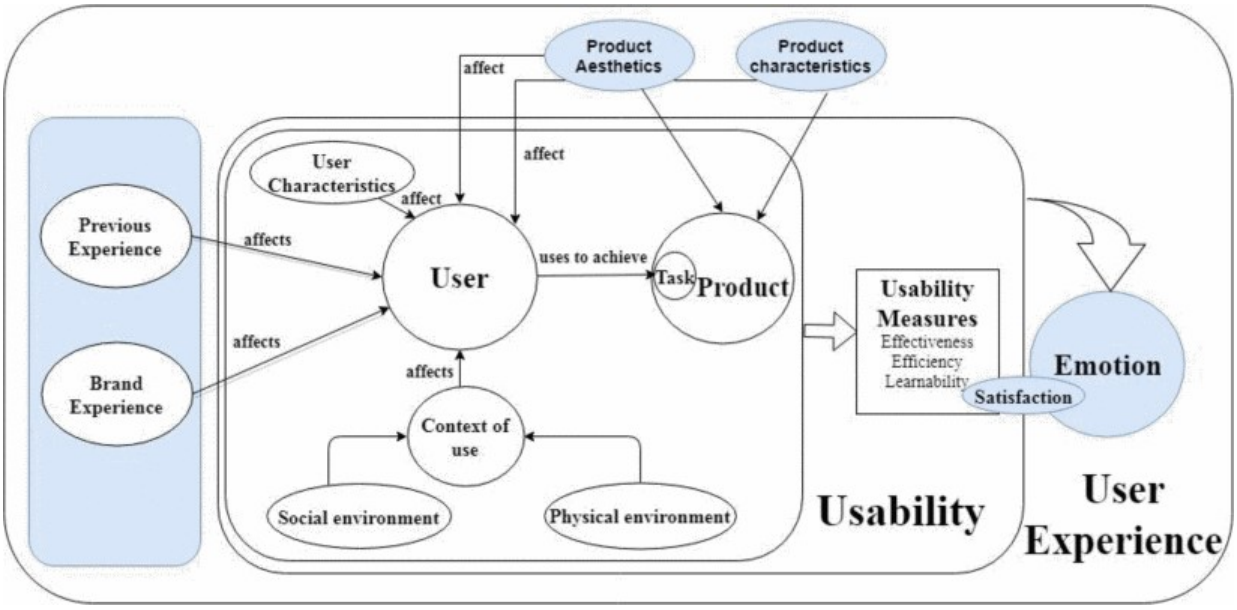

Abbildung 4: Aspekte der *User Experience* (Hassan & Galal-Edeen, 2017).

lieber vermeidet (Moser, 2012, S. 1). Dieser Aspekt der Nutzerfreundlichkeit lässt sich in den Bereich *User Experience* einordnen. Die *User Experience* erweitert die hauptsächlich auf technische Aspekte beschränkte *Usability* insofern, als sie das Gesamterlebnis bei der Nutzung von Software umfasst und somit auch emotionale Faktoren berücksichtigt (Richter & Flückiger, 2013, S. 156). In Abbildung 4 wird das eben beschriebene noch ein Mal visuell zusammengefasst. Im Zentrum der Abbildung befindet sich die *Usability*. Der Nutzer erfüllt durch Verwenden des Produkts eine Aufgabe. Bei diesem Prozess kann die *Usability* durch Werte wie Effektivität und Effizienz gemessen werden. Außerhalb des Bereichs der *Usability* sind in der Abbildung noch weitere Aspekte dargestellt. So sieht man, dass vergangene Erfahrungen beim Nutzen des Produkts und dessen Ästhetik bzw. Charakteristik den Nutzer beeinflussen. Alle genannten Punkte haben letztendlich Einfluss auf die Zufriedenheit des Nutzers und damit auf die Emotionen, welche durch die Nutzung des Produkts hervorgerufen werden (Hassan & Galal-Edeen, 2017).

2.3.2 Physik im Interface Design

Durch den Einsatz von physikalischen Simulationen kann die *User Experience* sowohl im Bereich der *Usability* als auch auf emotionaler Ebene verbessert werden. Die Interaktion mit einem User-Interface kann durch Physiksimulationen für den Nutzer intuitiver gestaltet werden, da der Nutzer sein Wissen über physikalische Systeme aus der realen Welt, auf die Interaktion mit dem User-Interface anwenden kann (McDirmid, 2009). Besonders bei physikalischen Eingabemethoden wie Touch oder Neigung spielt physikalische Simulation eine zentrale Rolle bei den Auswirkungen der entsprechenden Eingaben (McDirmid, 2009). Dem

Nutzer fällt die Interaktion mit einem User-Interface natürlich leichter, wenn seine Eingaben auch die Auswirkungen haben, die er aus seinen Erfahrungen in der realen Welt erwartet. Diese Imitation gewohnter Wahrnehmungsprozesse steigert die Nachvollziehbarkeit von Abläufen auf dem User-Interface und gibt dem Nutzer einen Bezugspunkt im Umgang mit diesen (Weber & Immich, 2009). Des Weiteren können physikalische Animationen dem Nutzer bei der Orientierung auf einem User Interface unterstützen und seine Aufmerksamkeit auf ausgewählte Elemente lenken. Um die Aufmerksamkeit des Nutzers zu beeinflussen, werden durch Animationen Fotorezeptoren des Nutzers angesprochen, die besonders sensitiv auf bewegte Reize reagieren. Somit können kritische Ereignisse auf dem User-Interface dem Nutzer durch entsprechende Animationen kenntlich gemacht werden (Weber & Immich, 2009). Besonders beim Auftauchen oder Verschwinden von Elementen auf dem User-Interface besteht die Gefahr, dass der Nutzer diese übersieht. Diese Übergänge können dem Nutzer durch Animationen wie Vergrößerung und Schrumpfen besser erkennbar gemacht werden (Weber & Immich, 2009). So könnte ein Element auf dem User Interface, dass ausgeblendet werden soll, zunächst vergrößert werden, um die Aufmerksamkeit des Nutzers zu erregen und anschließend explodieren, ausgeblendet werden oder durch Kollision mit einem anderen Element aus dem User Interface verdrängt werden. Eine Möglichkeit, die Orientierung des Nutzers auf dem User Interface zu unterstützen, besteht in der Anwendung von virtuellen Gelenken bzw. Federn. Bei dieser Methode ist ein Objekt durch starre Gelenke oder bewegliche Federn mit anderen Objekten verbunden (Wilson et al., 2008). Dadurch können beispielsweise verwandte Elemente auf einem User Interface durch Federgelenke miteinander verbunden werden, um dem Nutzer zu verdeutlichen, dass diese inhaltlich miteinander zusammenhängen.

Zusätzlich zu den eben aufgeführten Möglichkeiten, die *Usability* eines User-Interface zu verbessern, kann das Anwenden von Physik Simulationen auch den vom Nutzer empfundenen Realismus bei der Interaktion mit dem User Interface erhöhen. Dadurch wird der Nutzer stärker in die Interaktion einbezogen und empfindet das User-Interface als ästhetischer, wodurch die *User Experience* verbessert wird (McDirmid, 2009). Um ein Gefühl von Realismus zu vermitteln, ist es zuträglich, wenn User-Interfaces Verhaltensweisen wie beispielsweise Bewegungen so widerspiegeln, wie der Nutzer es aus der Realität gewohnt ist. Dieser Realismus lässt den Nutzer Vertrautheit bei der Interaktion mit dem User Interface spüren, wodurch die Interaktion positiver wahrgenommen wird. Zusätzlich können spielerische Elemente eingebaut werden, um die *User Experience* zu verbessern (Agarawala & Balakrishnan, 2006). Um auf das Beispiel der realistischen Bewegung zurückzukommen, könnte das Implementieren von Kollisionen einzelner Elemente und die Möglichkeit diese per Drag and Drop zu verschießen die *User Experience* verbessern. So könnte der Nutzer während einer Leerlaufphase, in der er eigentlich nichts zu tun hätte, die spielerischen Elemente des User-Interface nutzen, um die Zeit zu überbrücken (Agarawala & Balakrishnan, 2006).

Trotz der Vorteile, die das Einbinden von Physiksimulationen im User-Interface-Design bietet, bedeutet das nicht, dass User-Interfaces so realistisch wie möglich zu gestalten sind. Im

Gegensatz zu Videospielen soll die Physik dem Nutzer bei der Bedienung von User-Interfaces keine zusätzliche Herausforderung bieten und die *Usability* muss gewahrt bleiben. Zum Beispiel sollte der Nutzer nicht für schlechtes Zielen bestraft werden (McDirmid, 2009). Dementsprechend darf die UI-Physik nicht ausschließlich realistisch sein, sondern muss an manchen Stellen die Regeln der Physik brechen, um die *Usability* aufrechtzuerhalten. Beispiel dafür sind der *Lucky Shot* und *Force Fields*. Bei ersterem wird dafür gesorgt, dass ein vom Nutzer bewegtes Element sein Ziel trifft, obwohl es streng genommen nicht getroffen hätte. Durch *Force Fields* wird die Bewegung von bestimmten Elementen durch unnatürliche Barrieren beschränkt, sodass diese nicht den für sie vorgesehenen Bereich verlassen können (McDirmid, 2009). Wie eben erläutert können Physiksimulationen ein User-Interface in vielerlei Hinsicht aufwerten, wobei zu beachten gilt, dass zu viel Realismus die *Usability* unter Umständen auch einschränken kann.

3 Anforderungsanalyse

Der Begriff Anforderungsanalyse bezeichnet eine Vorgehensweise, bei der alle relevanten Aspekte des Anwendungsgebietes und der Aufgabenstellung einer zu entwickelnden Software erfasst, konsolidiert und dokumentiert werden (Broy & Kuhrmann, 2021, S. 203 f.). Dabei werden sowohl funktionale-, als auch nicht-funktionale Anforderungen festgelegt. Um ein Grundverständnis über die Funktionsweise des Community Mirrors aufzubauen, werden einleitend anhand eines Screenshots Begrifflichkeiten und der grobe Aufbau des Community Mirrors erklärt. Anschließend werden Features genannt, die bereits in der Software implementiert sind, jedoch wichtig für das Festlegen von weiteren Use-Cases sind. Danach werden die Ergebnisse aus Unterabschnitt 2.3.2 aufgegriffen, um daraus konkrete Use-Cases ableiten zu können, mit denen die *User Experience* durch physikalisch simulierte Elemente verbessert werden kann. Anschließend werden die festgelegten Use-Cases mit einem geeigneten Diagramm im Gesamtkontext der Software dargestellt. Abschließend werden noch nicht-funktionale Anforderungen festgelegt.

3.1 Grundaufbau Community Mirror

In diesem Abschnitt wird der Grundaufbau des Community Mirrors erklärt. Dazu wird ein Screenshot (vgl. Abbildung 5) genutzt. Die verschiedenen Elemente des Community Mirrors werden durch die Klasse *FlowView* in dieser Ansicht dargestellt. Die in blau bzw. gelb eingerahmten Kreise sind die *FlowVisualItems*. Im weiteren Verlauf dieser Arbeit werden diese auch allgemein als *VisualItems* bezeichnet. Diese bewegen sich horizontal von einer Seite des Bildschirms zur anderen und verlassen den Bildschirm daraufhin. Ein *VisualItem* kann per *Drag and Drop* bewegt und verschossen werden, woraufhin es nach einiger Zeit abbremst und aus dem Bildschirm ausgeblendet wird. Wenn ein *VisualItem* angeklickt wird, werden verwandte *VisualItems* wie in der Mitte des Screenshots zu sehen durch schwarze Linien mit dem angeklickten *Visual Item* verbunden angezeigt. Der große Textblock auf der linken Seite ist die *TeaserComponent*. Durch Anklicken dieser schließt sie sich und stattdessen erscheint das entsprechende *VisualItem* an der Stelle und verhält sich so wie die restlichen *VisualItems*. Die anderen auf dem Screenshot sichtbaren Elemente sind für diese Arbeit nicht relevant und deshalb wird an dieser Stelle nicht weiter auf sie eingegangen. Für weitere Informationen zum Community Mirror empfiehlt es sich im entsprechenden Wiki (`https://publicwiki.unibw.de/display/MCI/Mensch-Computer-Interaktion+Home`) nachzulesen.

Abbildung 5: Screenshot des Community Mirrors

3.2 Bereits implementierte Features

Da der Community Mirror vor dem Erstellen dieser Arbeit bereits über physikalisch simulierte Elemente verfügt, werden hier nun einige vorgestellt, die dazu geeignet sind im Zuge dieser Arbeit weiterentwickelt bzw. genutzt zu werden.

3.2.1 Mouse Events

Im Community Mirror ist es bereits möglich, die *Visual Components* anzuklicken, um Aktionen auszulösen oder sie per *Drag and Drop* zu bewegen bzw. zu verschießen (Kraus, 2022). Bestehende Anwendungen für dieses Feature liegen im Einklappen der *Teaser Component* und dem Bewegen der *VisualItems*. Durch das Anklicken eines *VisualItems* werden außerdem verwandte Elemente angezeigt und durch Federn verbunden. Dieses Feature wird in Unterabschnitt 3.2.3 noch ausführlicher vorgestellt. Der dazu benötigte *Event Handler* wird durch JavaFx zur Verfügung gestellt und in einigen Klassen des Community Mirrors verwendet. Um Einheitlichkeit zu wahren und die bereits existierenden Features des Community Mirrors in die Entwicklung von physikalisch simulierten Elementen einzubinden, bietet es sich an, die bestehenden *Mouse Events* zu nutzen und lediglich an die zu integrierende Physik-Engine anzupassen.

3.2.2 Ausblendung

Die aktuelle Version des Community Mirrors verfügt über ein *Fade out and die* Verhalten. Dieses sorgt dafür, dass *VisualItems* ausgeblendet und zerstört werden können. In Unterabschnitt 2.3.2

wurde bereits erläutert, dass dieses Feature eine gute Möglichkeit darstellt, um die Aufmerksamkeit des Nutzers auf einen bestimmten Punkt auf dem Display zu lenken. Dementsprechend soll auch dieses Feature nach Möglichkeit übernommen werden. Dabei würde sich beispielsweise anbieten, das Zerstören von Objekten in der Physik-Engine mit dem *Fade out and die* Verhalten zu verknüpfen.

3.2.3 Federn für verwandte Elemente

Wenn ein Nutzer auf ein *Visual Item* klickt, bleibt dieses stehen und es erscheinen weitere *VisualItems*, die thematisch mit dem angeklichten verwandt sind. Die verwandten Elemente erscheinen um das angeklickte *Visual Item* herum und werden durch schwarze Striche mit dem Ursprungselement verbunden. In Abbildung 6 ist dieses Feature abgebildet.

Abbildung 6: Verwandte Elemente eines *VisualItems* durch Federn verbunden.

Durch das Anzeigen und Verbinden der verwandten Elemente soll die Orientierung des Nutzers verbessert werden (vgl. Unterabschnitt 2.3.2). Jedoch hat das Feature im Zustand vor der Durchführung dieser Arbeit noch einige Schwächen. Zunächst kommt es dazu, dass die *VisualItems* sich überschneiden, da es keine Kollisionserkennung gibt. An sich ist eine hohe Dichte an Informationen nicht zwingend schädlich für die *User Experience* (Staggers, 1993). Durch das Überschneiden von Texten und Grafiken wird dem Nutzer das Lesen der Informationen jedoch erschwert. Dies lässt sich in Abbildung 6 schon erahnen, wird jedoch in

Abbildung 7 noch klarer deutlich. Abbildung 7 ist ein Screenshot der verwandten Elemente eines *VisualItems* nachdem dieses bewegt wurde. Die dargestellten Informationen können nicht mehr gelesen werden. Da das Feature grundsätzlich gut geeignet ist, um die Orientierung des Nutzers zu verbessern, soll es im Zuge dieser Arbeit aufgegriffen und möglichst verbessert werden.

Abbildung 7: Verwandte Elemente eines *VisualItems* nach *Drag and Drop*.

3.3 Entwurf von Use-Cases

In diesem Abschnitt werden die Use-Cases für die Integration der Physik-Engine festgelegt. Dabei wird beachtet, dass diese gemäß den Ergebnissen aus Unterabschnitt 2.3.2 geeignet sind, um die *User Experience* des Nutzers zu verbessern. Außerdem werden bereits bestehende Features der Software berücksichtigt, um diese weiterverwenden zu können.

3.3.1 Verbesserung des empfundenen Realismus durch Kollisionen

Um den vom Nutzer empfundenen Realismus zu erhöhen, spielt die Simulation von physikalischen Eigenschaften eine zentrale Rolle (Wages et al., 2004, S. 217). In einem zweidimensionalen Display bestehen dafür deutlich weniger Möglichkeiten, als beispielsweise einem dreidimensionalen Flug-Simulator. Im Bereich Software kann der Begriff Realismus daher aus einem bestimmten Bezugspunkt betrachtet werden. So sollte die Welt des Bezugspunktes (hier der Community Mirror) in sich stimmig sein, um auf den Nutzer einen realistischen Eindruck zu hinterlassen (Wages et al., 2004). Eine Möglichkeit dazu besteht im Einbau von Kollisionen und passenden Reaktionen. Die *VisualItems* bilden einen Großteil der auf dem Community Mirror dargestellten Informationen ab und fliegen in Form eines Kreises von der einen Seite des Mirrors zur anderen,

woraufhin sie den Mirror wieder am Rand verlassen. Durch physikalisch simulierte Kollisionen, in denen Geschwindigkeit, Masse und Richtungsvektoren der *VisualItems* einbezogen werden, soll der vom Nutzer empfundene Realismus erhöht werden. Um zu verhindern, dass die *VisualItems* den Mirror am oberen oder unteren Bildschirmrand verlassen, nachdem sich ihre Flugbahn durch eine Kollision geändert hat, werden am oberen und unteren Rand Wände hinzugefügt, von denen die *VisualItems* abprallen. Um den Bewegungsfluss der *VisualItems* zu wahren, soll die Kollisionsreaktion mit Wänden so angepasst werden, dass die Bewegung auf der x-Achse weiter in die gleiche Richtung geht. So soll ein *Visual Item*, das von unten links gegen die obere Bildschirmwand stößt, nach der Kollision nach unten rechts abprallen.

3.3.2 Steigerung der Interaktionen durch Gamification

Unter Gamification wird die Verwendung von Designelementen aus Spielen in einer nicht als Spiel angedachten Software verstanden. Dadurch soll ein spielerisches Nutzererlebnis geschaffen werden, dass die Freude an der Nutzung erhöht und der Nutzer zu bestimmten Verhalten motiviert werden (Deterding et al., 2013). Anders als bei Spielen geht es bei der Gamification nicht vorrangig darum, dass der Nutzer sich durch Spielen die Zeit vertreibt, sondern darum eine aus Spielen bekannte Technik einzusetzen, um etwas anderes zu erreichen (Joachim & Kupka, 2013, S. 7). Anbieter für Gamification Produkte behaupten, durch ihre Lösungen Interaktionen von Kunden mit einem bestehenden Produkt wie beispielsweise einer Website deutlich erhöhen zu können (Gonzales-Scheller, 2013, S. 34). Um diesen Effekt auch im Kontext des Community Mirrors zu erzielen, soll dort auch eine Gamification eingebaut werden. Die *VisualItems* auf dem Community Mirror sind bereits mit der Maus bzw. per Touch beweg- und verschießbar. Da sie durch ihre runde Form vom Aussehen Bällen ähneln, bietet es sich an als Gamification eine Art Tor oder Korb zu erstellen, in den die *VisualItems* geschossen werden können. Dieser Korb könnte gleichzeitig als eine Möglichkeit dienen, *VisualItems* vom Bildschirm zu entfernen. So würde die Gamification den Nutzer auf spielerische Weise auf die Möglichkeit aufmerksam machen, mit dem Community Mirror zu interagieren. Im Laufe dieser Arbeit ist eine Kooperation mit einer Forschergruppe aus Studenten entstanden, die testet, inwiefern das Hinzufügen einer Gamification dazu führt, dass mehr Leute mit dem Community Mirror interagieren. Nach Absprache mit der Forschergruppe wurde beschlossen, einen Basketballkorb am unteren Bildschirmrand einzuführen, in den die *VisualItems* per *Drag and Drop* oder durch ihre standardmäßige Flugbahn hineinfliegen können. Letzteres soll dazu dienen, dass der Nutzer schon bevor er mit dem Bildschirm interagiert auf das Feature aufmerksam gemacht wird. Durch die in Unterabschnitt 3.3.1 festgelegten Use-Cases zur Kollision der *VisualItems* miteinander und mit der oberen- und unteren Bildschirmkante soll die Gamification noch weiter durch Realismus aufgewertet werden.

3.3.3 Bessere Orientierung durch Gelenke und Federn

Das Anzeigen von *VisualItems* als verwandte Elemente, die durch Federn bzw. Gelenke miteinander verbunden sind, ist in der aktuellen Version des Community Mirrors noch ausbaufähig. Dazu soll dieses Feature in Zukunft mit einer Physik-Engine umgesetzt werden. Das Ziel dabei besteht einerseits darin, Überschneidungen der *VisualItems* durch Kollisionen zu verhindern, andererseits sollen die Bewegungen der miteinander verbundenen Elemente natürlicher wirken.

3.4 Nicht-funktionale Anforderungen

Um zu gewährleisten, dass die im Zuge dieser Arbeit entstandenen Änderungen am Code des Community Mirrors für weitere Projekte und wissenschaftliche Arbeiten nutzbar ist, werden folgende nicht-funktionale Anforderungen festgelegt. Die Performance der Software ist kein Schwerpunkt dieser Arbeit, dennoch soll weiterhin ein flüssiger Ablauf der Software gewährleistet sein. Zusätzlich soll eine gute Wartbarkeit der Software Ziel der Entwicklung sein. Dazu sollen die eingeführten Änderungen ausreichend dokumentiert werden und möglichst sinnvoll in den bestehenden Code integriert werden. Näheres dazu folgt in Abschnitt 4.2. Da die Software auf den Community Mirrors der Universität der Bundeswehr München installiert werden soll, besteht eine weitere Anforderung darin, dass die im Zuge dieser Arbeit entwickelten Features geeignet sind, um auf diesen installiert und betrieben zu werden.

4 Konzeption

In diesem Kapitel wird ein Konzept zur Implementierung der Use-Cases erstellt. Dazu wird zunächst eine geeignete Physik-Engine ausgewählt. Anschließend wird erklärt, wie diese sich in die bestehende Software einbinden lässt und wie die neuen Features implementiert werden sollen. Abschließend wird anhand ereignisgesteuerter-Prozessketten der Ablauf der Physiksimulation erklärt.

4.1 Auswahl der Physik-Engine

Die Auswahl der Physik-Engine hat einen großen Einfluss auf den weiteren Verlauf dieser Arbeit. Um die optimale Physik-Engine für die geplanten Use Cases zu finden, werden zunächst Kriterien aufgestellt, die eine Engine zwangsläufig erfüllen muss, um in Betracht gezogen zu werden.

- Die Physik-Engine muss in Java implementiert sein bzw. über eine entsprechende Portierung verfügen
- Kompatibilität zu JavaFX Anwendungen muss gewährleistet sein
- Die Physik-Engine muss dafür geeignet sein, die erstellten Use Cases zu implementieren
- Es soll sich um eine Open-Source Lösung handeln, also muss sie kostenlos und mit einsehbaren Quellcode zur Verfügung gestellt werden

Im Folgenden werden einige Physik-Engines genannt, welche die festgelegten Kriterien erfüllen. Im Wiki des Community Mirrors (`https://publicwiki.unibw.de/display/MCI/Mensch-Computer-Interaktion+Home`) werden bereits drei Engines vorgeschlagen, die in Betracht gezogen werden können. Dazu zählen JBox2D (Java Portierung der Physik-Engine Box2D), Phys2D und javafx-ik. In der Industrie verbreitete Physik-Engines, die oben festgelegte Kriterien erfüllen, sind Box2D, PhysX und Bullet Physics (Templet, 2021, S. 40 ff.). Nach zusätzlicher Online-Recherche ist außerdem noch die Engine dyn4j als geeignet befunden worden. Hier eine Auflistung der Websites von potenziellen Physik-Engines, die in den Community Mirror integriert werden könnten.

- `https://dyn4j.org/`

- http://jbullet.advel.cz/
- http://www.jbox2d.org/
- https://github.com/FourSeventy/phys2d
- https://github.com/netopyr/javafx-ik
- https://www.nvidia.com/de-de/drivers/physx/physx-9-21-0713-driver/

4.1.1 Vergleich der Physik-Engines

Um von der vielfältigen Auswahl an Physik-Engines eine für das Projekt dieser Arbeit zu bestimmen, werden diese nun auf die Qualität der Dokumentation, Einsteigerfreundlichkeit und ggf. existierende Produkte, die mit dieser Engine realisiert wurden, geprüft. Da die Software des Community Mirrors regelmäßig von mehreren Entwicklern weiterentwickelt wird, besteht das Hauptaugenmerk bei der Einsteigerfreundlichkeit und der Qualität der verfügbaren Dokumentation.

So lassen sich zunächst die Physik-Engines javafx-ik und Phys2D ausschließen. Bei einer Google-Suche nach Phys2D lässt sich ausschließlich die Software auf Github und einige veraltete Artikel (älter als 10 Jahre) finden. Bei javafx-ik gibt es zwar einen Entwickler-Blog, in dem die Engine vorgestellt wird, jedoch ist auch hier eine Dokumentation durch eine Google-Suche nicht zu finden. Die Engine PhysX verfügt über eine ausführliche Dokumentation, jedoch nicht über einen Starter-Guide, wodurch gerade der Einstieg in die Engine mit einer steilen Lernkurve verbunden ist (Templet, 2021, S. 50). Bei Bullet Physics handelt es sich um eine Physik-Engine, die in C++ implementiert wurde, jedoch mit JBullet über eine Java Portierung verfügt. Sie verfügt über mehrere Quellen zur Dokumentation, darunter einen Starter-Guide, eine vollständige Klassendokumentation, die durch das Tool Doxygen generiert wurde und einen Nutzer-Guide (Templet, 2021, S. 57 f.). Der Einstieg in die Engine wird dadurch erschwert, dass die benötigten Schritte zum Erstellen einer Basis-Anwendung nicht konkret zusammengefasst aufgelistet sind, die Dokumentation teilweise spezifische Fachbegriffe aus dem Gebiet Robotik verwendet und die Dokumentationen teilweise an verschiedene Programmiersprachen angepasst sind, was dem Entwickler eine zusätzliche Translation abverlangt (Templet, 2021, S. 58 f.).

Die Engine JBox2D ist eine Java Portierung von Box2D. Auf der Website von Box2D gibt es eine ausführliche Dokumentation über die Features der Engine mit Beispielcode, einen Starter-Guide und eine vollständige Klassendokumentation. Diese Dokumentation ist jedoch auf C++ ausgelegt. Die Erklärungen sind auch für Entwickler mit wenig Vorkenntnissen im Bereich der Physiksimulationen leicht zu verstehen und die Engine bietet eine eigene GUI, in der die Features der Engine isoliert getestet werden können (Templet, 2021, S. 42). Box2D verfügt über einen gewissen Bekanntheitsgrad, da es von bekannten Spiele-Engines wie Unity genutzt wird und auch bekannte Spiel wie Angry Birds und Shovel Knight wurden mit Nutzung von Box2D

entwickelt (Templet, 2021, S. 40 f.). Diese Bekanntheit führt dazu, dass es auf Plattformen wie Youtube vergleichsweise mehr lehrreiche Inhalte über Box2D gibt, als über die anderen Physik-Engines. Außerdem beweist die Implementierung von Box2D in den genannten Produkten, die Tauglichkeit der Engine in einer zweidimensionalen Software. Die Engine dyn4j verfügt ebenfalls über eine Website mit Starter-Guide, Beispielsoftware und einer Klassendokumentation und ist damit auch eine mögliche Wahl.

4.1.2 Entscheidung über die Wahl der Physik-Engine

Unter Berücksichtigung der in Abschnitt 4.1 aufgeführten Punkte wird JBox2D als Physik-Engine für diese Arbeit ausgewählt. Der einzige Nachteil besteht darin, dass die Dokumentation für C++ ausgelegt ist und dementsprechend bei der Nutzung, der Java Portierung eine gewisse Transferleistung vom Entwickler benötigt wird. Folgende Aspekte waren bei der Wahl von JBox2D ausschlaggebend:

- Bekannte Spiele und Game-Engines nutzen Box2D
- Gesamte Dokumentation auf einer Website
- Hohe Anzahl von Videos und Beispielprojekten über Box2D
- Sehr hohe Beginnerfreundlichkeit durch ausführliche Dokumentation und Testbed.

4.2 Einbindung in die bestehende Software

Nach dem Festlegen von Use-Cases und der Auswahl der Physik-Engine sollen diese nun in die bestehende Software integriert werden. In Abbildung 8 sind die für diese Arbeit relevanten Klassen der Community Mirror Software dargestellt. Dabei gilt zu beachten, dass aus Platzgründen nur öffentliche- nicht geerbte Methoden aufgelistet sind. Die Klasse *FlowView* stellt die JavaFX Benutzeroberfläche bereit. Sie erstellt die *VisualItems* mit der Methode *createFlowVisualItem*. Außerdem verfügt sie über die Methode *onUpdate*, durch die wiederum eine Update-Methode aller in der *FlowView* dargestellten Objekte (inklusive die *VisualItems*) aufgerufen wird. Zusätzlich wird in der Methode *onUpdate* auch die *Teaser Component* ein bzw. ausgeblendet. Der erste Schritt nach der Installation von JBox2D besteht daher darin, den Physik-Engine-Loop in die *FlowView* einzubinden. In der momentanen Version des Community Mirrors besitzen die *VisualItems* über die Methode *move*, mit der sie ihre Position verändern können (wird in Abbildung 8 nicht angezeigt, da vererbte Methoden aus Platzgründen dort nicht dargestellt sind). Durch die Integration von JBox2D soll die Bewegung der Elemente des Community Mirrors jedoch innerhalb des Physik-Engine-Loops berechnet werden. Deshalb besteht der nächste Schritt darin, Methoden zu erstellen, mit denen *VisualItems* auch innerhalb von JBox2D erstellt und simuliert werden. Die Möglichkeiten zur Interaktion des Nutzers mit

dem Community Mirror sind momentan in verschiedenen *Behaviour* Klassen implementiert. In Abbildung 8 sind exemplarisch für die *Behaviours* zwei abstrakte Oberklassen dargestellt. Das Anzeigen der verwandten Elemente eines *VisualItems* wird durch die Klassen *Graph*, *GraphNode* und *GraphEdge* umgesetzt. Ein weiterer Schritt bei der Integration der Physik-Engine besteht darin, Features wie die Interaktion durch *Drag and Drop* und das Anzeigen der verwandten Elemente eines *VisualItems* mit JBox2D umzusetzen.

4.2.1 Einbindung von JBox2D in die *FlowView*

Um die Funktionalitäten von JBox2D in die *FlowView* zu integrieren, soll die Klasse *PhysicsEngine* erstellt werden, welche in der *FlowView* als Attribut gespeichert wird. Beim Initialisieren der *FlowView* soll dann auch eine Instanz der *PhysicsEngine* erstellt werden. JBox2D benutzt die Klasse *World*, in der alle erstellten Körper physikalisch simuliert werden (Templet, 2021, S. 43). Dementsprechend soll im Konstruktor der Klasse *PhysicsEngine* eine neue Instanz von *World* erstellt werden, in die später die *VisualItems* eingefügt werden können. Außerdem soll die Klasse *PhysicsEngine* über eine Update-Methode (Physik-Engine-Loop) verfügen, die bei jedem Aufruf der Methode *onUpdate* aus der *FlowView* ebenfalls aufgerufen wird. Innerhalb der Update-Methode werden dann die Bewegungen der erstellten Körper simuliert und deren Positionen berechnet.

4.2.2 *VisualItems* durch JBox2D erstellen und simulieren

Die *VisualItems* werden von der *FlowView* erstellt. In JBox2D wird ein Körper durch die Klasse *Body* dargestellt, in der Position, Form, Bewegungsvektoren und sonstige physikalische Eigenschaften wie Reibung und Masse festgelegt werden können (Catto, 2023). Um ein *Visual Item* als *Body* in die *World* von *Physics-Engine* hinzuzufügen, soll die Methode *add* erstellt werden. Diese soll von der *FlowView* aufgerufen werden, wenn ein neues *Visual Item* erstellt wird. Mit der Position, dem Radius und der Bewegungsrichtung des *VisualItems* soll in der Methode *add* ein *Body* erstellt werden. Dieser wird der *World* Instanz übergeben und dementsprechen wird sein Verhalten durch den Physik-Engine-Loop simuliert. Damit die *VisualItems* auch die neue Positionsberechnung verwenden, bekommen sie ihren entsprechenden *Body* als Attribut und übernehmen in ihrer eigenen Update-Methode seine Position. in JBox2D müssen Körper manuell zerstört werden (Templet, 2021, S. 44). Deshalb soll beim Zerstören eines *VisualItems* durch die Methode *destroy* zusätzlich der entsprechende *Body* zerstört werden.

4.2.3 Anpassung der bestehenden *Behaviours* an JBox2D

Die zum Implementieren der Use Cases relevanten *Behaviours* sind die Klassen *DragBehaviour* und *DriftBehaviour*. Letztere ist für die Bewegung der *VisualItems* verantwortlich. Da diese Funktion nun von der Klasse *PhysicsEngine* übernommen wird ist *DriftBehaviour* im Kontext

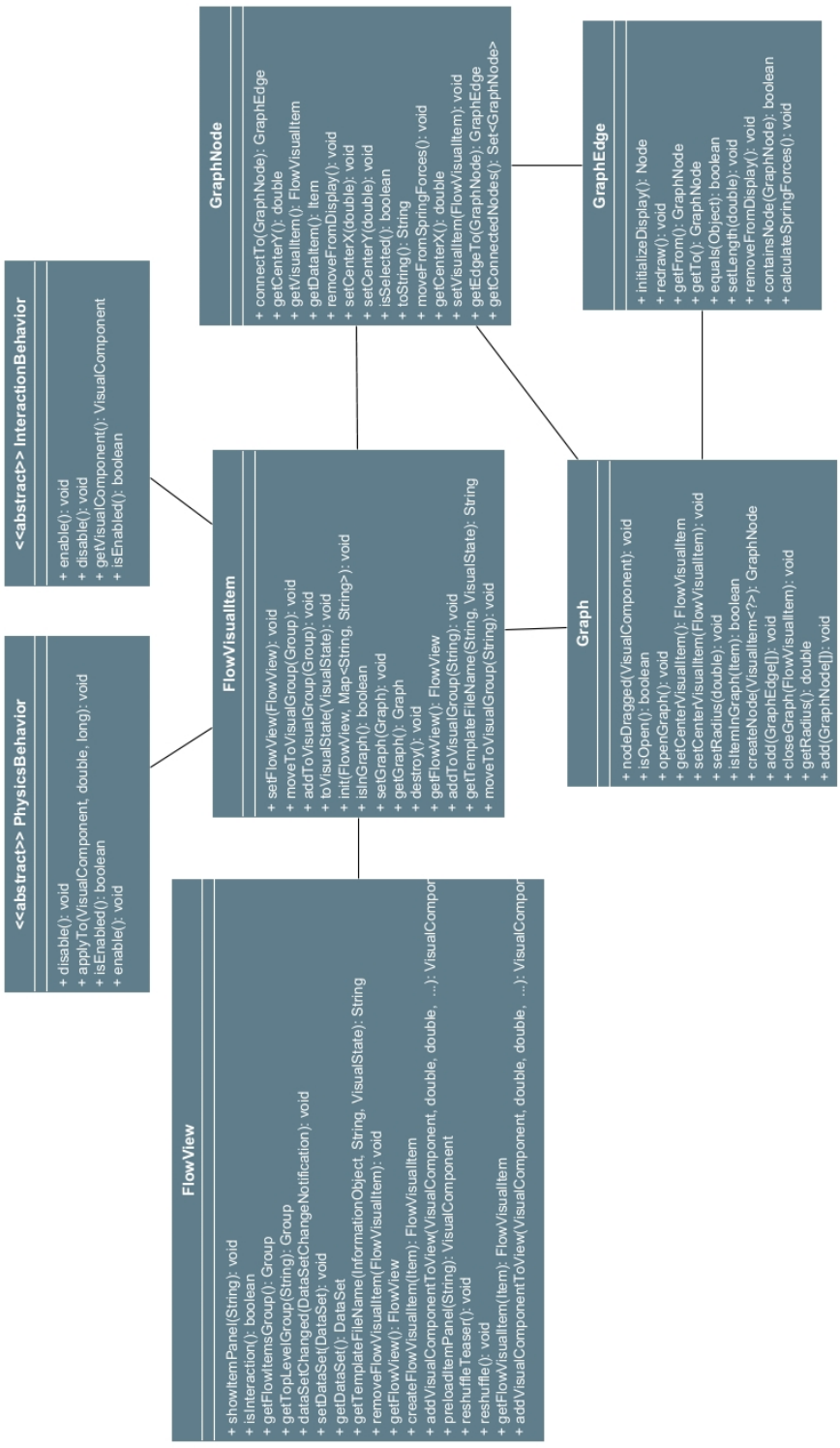

Abbildung 8: Klassendiagramm des für die Integration relevanten Bereiches

der *VisualItems* redundant geworden und soll nicht mehr verwendet werden. Die Klasse *DragBehaviour* ist für das Bewegen und Verschießen der *VisualItems* per *Drag and Drop* verantwortlich. Diese Funktionalität lässt sich in zwei Bereiche einteilen. Zunächst das Bewegen der *Visualitems* mit der Maus bzw. Touch. Die *VisualItems* nehmen dabei die Position de Mauscursors ein. Um dies mit JBox2D umzusetzen, soll *DragBehaviour* über eine getter-Methode auf den *Body* des *VisualItems* zugreifen und ihn an die Position des Mauscursors bewegen. Der zweite Bereich ist das Verschießen des *VisualItems* nach Loslassen der Maustaste. In der aktuellen Version berechnet die *DragBehaviour* einen Bewegungsvektor und wendet ihn auf *VisualItem* an. Stattdessen soll auch hier der Vektor auf den *Body* des *VisualItems* angewendet werden, damit die Physik-Engine die Bewegung realisieren kann.

4.2.4 Anzeigen der verwandten Elemente mit JBox2D

Für das Anzeigen der verwandten Elemente eines *VisualItems* sind die Klassen *Graph*, *GraphNode* und *GraphEdge* verantwortlich. In der Klasse *Graph* werden mit den Methoden *openGraph* und *closeGraph* die verwandten Elemente entsprechend angezeigt bzw. ausgeblendet. Die Klasse *GraphEdge* ist dafür verantwortlich, die Federn durch die *FlowView* anzeigen zu lassen und die durch die Federn ausgeübten Kräfte zu berechnen. Letztere Funktionalität soll durch JBox2D übernommen werden. JBox2D bietet verschiedene Arten von Federn und Gelenken an, die von der abstrakten Klasse *Joint* erben (Catto, 2023). Vom Verhalten kommt die Klasse *RopeJoint* den Federn im Community Mirror am nächsten, weshalb diese im Folgenden verwendet werden soll. Dafür soll die Klasse *PhysicsEngine* über eine Methode verfügen einen solchen *RopeJoint* zwischen zwei *VisualItems* zu erstellen. Diese Methode kann dann von *openGraph* für die Verbindung der verwandten Elemente durch Federn aufgerufen werden. Der visuelle Teil der Verbindung soll nach wie vor durch *GraphEdge* realisiert werden. In der Methode *closeGraph* müssen dann lediglich die *RopeJoint*-Instanzen wieder zerstört werden.

4.3 Neue Features

Zusätzlich zu den Features, die bereits existieren und nur in JBox2D integriert werden müssen, erfordern die Use-Cases auch das Implementieren von neuen Features. Namentlich sind das die Kollisionen der *VisualItems* miteinander, das Erstellen von Banden am oberen und unteren Bildschirmrand, die Gamification und das Kollidieren der *VisualItems* mit den Banden, wobei jedoch die Bewegung auf der x-Achse weiterhin in dieselbe Richtung gehen soll.

4.3.1 Kollisionen und Banden am Bildschirmrand

Da zu *VisualItems* analog auch eine *Body*-Instanz erstellt wird (vgl. Unterabschnitt 4.2.2), können diese durch JBox2D Methoden eine Form als Attribut haben. Diese Form wird in

JBox2D durch die Klasse *Shape* realisiert und es gibt mehrere Arten von Formen wie beispielsweise Kreise und Vielecke (Catto, 2023). Für jede *Body*-Instanz, die über ein *Shape*-Attribut verfügt, werden Kollisionen durch die Physik-Game-Loop erkannt und die entsprechenden Maßnahmen getroffen. Um die Banden an den Bildschirmrändern zu erstellen, sollen in der Klasse *PhysicsEngine* entsprechende statische *Body*-Instanzen erstellt werden. Die Kollisionsreaktionen sind von mehreren Attributen der kollidierenden Körper und der *World*-Instanz, in der die Simulation stattfindet, abhängig. Die gewünschte Kollision von den Banden, bei der die horizontale Bewegung bestehen bleiben soll, entspricht nicht der Kollisionsreaktion, welche der Physik-Engine-Loop berechnen würde. JBox2D bietet mit der Klasse *ContactListener* manuelle Aktionen zu Beginn und zum Ende einer Kollision auszuführen (Catto, 2023). Um die passenden Kollisionsreaktionen nach Kollision zwischen *Visual Item* und dem Bildschirmrand zu erhalten, soll ein *ContactListener* in den Physik-Engine-Loop integriert werden.

4.3.2 Gamification

Die Gamification soll in Form eines Basketballkorbes am unteren Bildschirmrand umgesetzt werden. Für die visuelle Dartstellung muss lediglich ein entsprechendes Bild in der *onInit* Methode der *FlowView* eingefügt werden. Bei der Umsetzung durch JBox2D soll die Bande am unteren Bildschirmrand aus zwei nebeneinander liegenden Körpern mit einem Loch zwischen ihnen bestehen, durch das die *VisualItems* nachdem sie im Korb gelandet sind, den Bildschirm verlassen können. Um dafür zu sorgen, dass *VisualItems* nicht einfach durch den Korb durchfliegen, soll die Physik-Engine-Loop prüfen, ob diese sich in der Hitbox des Korbes befinden. Ist dies der Fall, sollen sie einen neuen Bewegungsvektor bekommen, durch den sie den Bildschirm verlassen.

4.3.3 Zusammenfassung der neuen Klassen

Nach den bisherigen Ergebnissen der Konzeptionsphase, werden zur Implementierung der Use-Cases zwei neue Klassen benötigt, die in Abbildung 9 abgebildet sind. Außerdem müssen eine Vielzahl an Methoden aus den bereits existierenden Klassen geändert bzw. hinzugefügt werden. Eine Übersicht auf alle neuen und überarbeiteten Methoden inklusive einer Beschreibung derer Funktion wird in Kapitel 5 durch Tabelle 1 und Tabelle 2 gegeben. Die erste neue Klasse *PhysicsEngine* ist dafür verantwortlich, für die Inhalte in der *FlowView* entsprechende JBox2D Objekte zu erstellen. Dazu gibt es die Methoden *createRope*, *createEdge* und *add* wodurch entsprechende Objekte für Federgelenke, Banden am Bildschirmrand und *VisualItems* erstellt werden. Das Verhalten der Objekte wird in der Methode *update* simuliert, welche den Physik-Engine-Loop darstellt. Dabei nutzt sie die Klasse *ContactListener*. Diese wird für jede erkannte Kollision aufgerufen und erkennt in der Methode *beginContact*, ob die Kollision zwischen Bande und *Visual Item stattfindet*. Ist dies der Fall, wird die gewünschte Kollisionsreaktion ausgeführt. Durch die Methode *endContact* können ggf. Aktionen ausgelöst werden, wenn

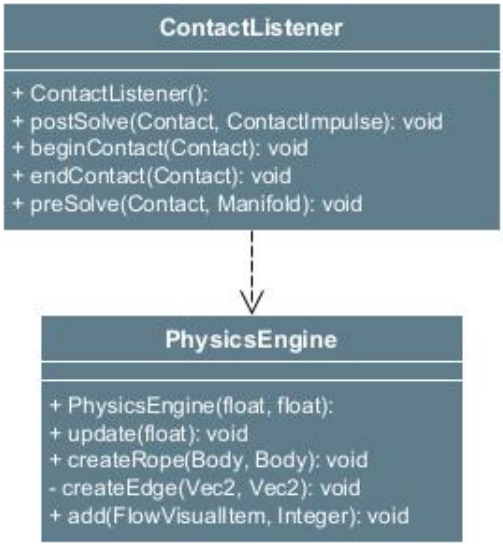

Abbildung 9: Neue Klassen

sich die kollidierenden Körper nicht mehr berühren. Die Methoden *preSolve* und *postSolve* sollen keine Funktion haben und sind nur für die Vollständigkeit im Klassendiagramm (*Contact Listener* implementiert ein Interface, in dem die Methoden bereits vorgegeben sind).

4.4 Ablauf der Physiksimulation

Im Folgenden wird der Ablauf der Physiksimulation beschrieben. Dabei wird anhand von ereignisorientierten Prozessketten beschrieben, wie das Verhalten von *VisualItem* bestimmt wird. Anschließend wird dargestellt, wie der *ContactListener* zum Auslösen einer Kollisionsreaktion verwendet wird.

4.4.1 Verhalten von *VisualItems*

In Abbildung 10 wird dargestellt, wie das Verhalten von *VisualItems* beeinflusst wird. Zunächst wird es erstellt und bekommt eine Startgeschwindigkeit. Dann wird eine Schleife durchlaufen, in der immer wieder das Verhalten des *VisualItems* bestimmt wird. Die Ausgangssituation dieser Schleife besteht darin, dass ein *VisualItem* sich auf dem Bildschirm bewegt. Es können drei mögliche Ereignisse eintreten. Das erste Ereignis besteht darin, dass ein Nutzer ein *VisualItem* anklickt. Daraufhin wird geprüft, ob dieses bereits verwandte Elemente anzeigt. Ist dies nicht der Fall, werden verwandte *VisualItems* und Federgelenke erstellt. Werden verwandte *VisualItems* bereits angezeigt, werden diese samt der zugehörigen Federgelenke zerstört. Anschließend tritt wieder die Ausgangssituation ein, in der sich das *VisualItem* auf dem Bildschirm bewegt. Ein weiteres Ereignis besteht darin, dass der Nutzer ein *VisualItem* per *Drag and Drop*

bewegt hat. Als Reaktion darauf bekommt das *VisualItem* einen neuen Bewegungsvektor und die Ausgangssituation tritt wieder ein. Das letzte mögliche Ereignis, wodurch das Verhalten von *VisualItems* beeinflusst werden kann, besteht aus durch die Physiksimulation ausgelöste Reaktionen. Dazu gehören ausgeübte Kräfte, die durch Kollisionen und die Gelenke, welche die Körper verbinden, ausgeübt werden. Als Reaktion darauf bekommt das *VisualItem* einen neuen Bewegungsvektor und die Ausgangssituation tritt wieder ein. Die letzte Reaktion durch die Physiksimulation tritt ein, wenn ein *VisualItem* im Basketballkorb ist. Dann bekommt es einen neuen Bewegungsvektor, durch den es den Bildschirm verlässt.

4.4.2 Verhalten des *ContactListeners*

In Abbildung 11 wird das Verfahren zum Bestimmen von Kollisionsreaktionen dargestellt. Die Ausgangssituation besteht darin, dass beim Durchlaufen vom Physik-Engine-Loop eine Kollision erkannt wurde. Anschließend wird die Art der kollidierenden Körper bestimmt. Es gibt die Möglichkeit, dass zwei *VisualItems* aufeinander getroffen sind oder dass ein *VisualItem* auf den Bildschirmrand getroffen ist. Bei der erstgenannten Möglichkeit wird die Default-Kollisionsreaktion von der Physiksimulation ausgeführt. Wenn ein *VisualItem* auf einen Bildschirmrand trifft, wird die Kollisionsreaktion beeinflusst, sodass die Bewegungsrichtung und Geschwindigkeit auf der x-Achse erhalten bleiben. Durch die angewendete Kollisionsreaktion trennen sich die kollidierten Körper voneinander und die Kollision ist beendet. Nun soll das Attribut *Restitution* der Körper geprüft werden, welches festlegt, ob eine Kollision elastisch oder unelastisch ist. Besitzt ein Körper ein unelastisches Kollisionsverhalten, so wird er in dem Moment vom Nutzer mit der Maus bewegt (wird in Abschnitt 5.4 weiter ausgeführt). In diesem Fall wird eine durch Kollision verursachte Bewegung des Körpers gestoppt. Für einen Körper mit elastischen Kollisionsverhalten werden keine Maßnahmen ergriffen.

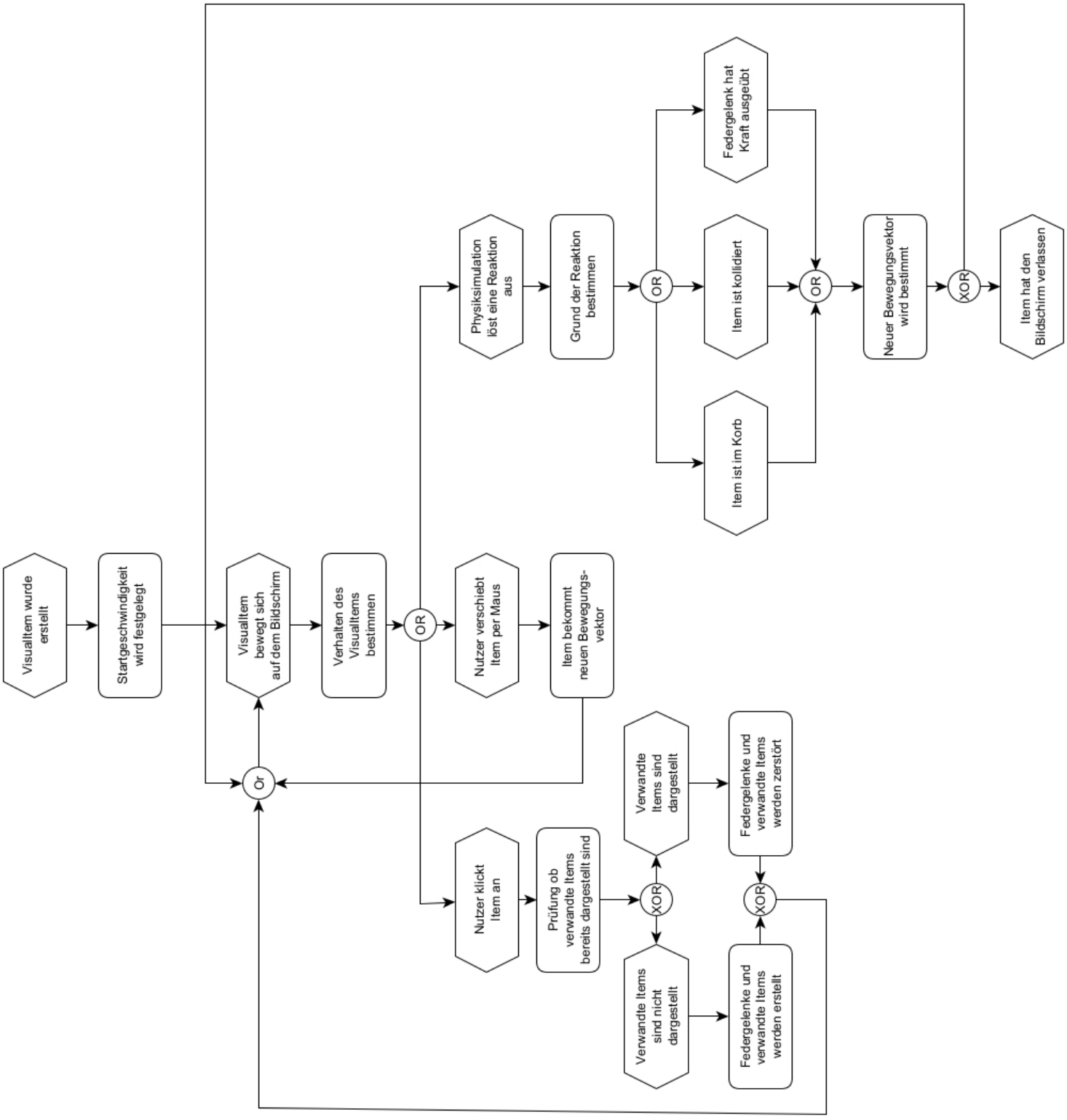

Abbildung 10: EPK zum Verhalten von *VisualItems*

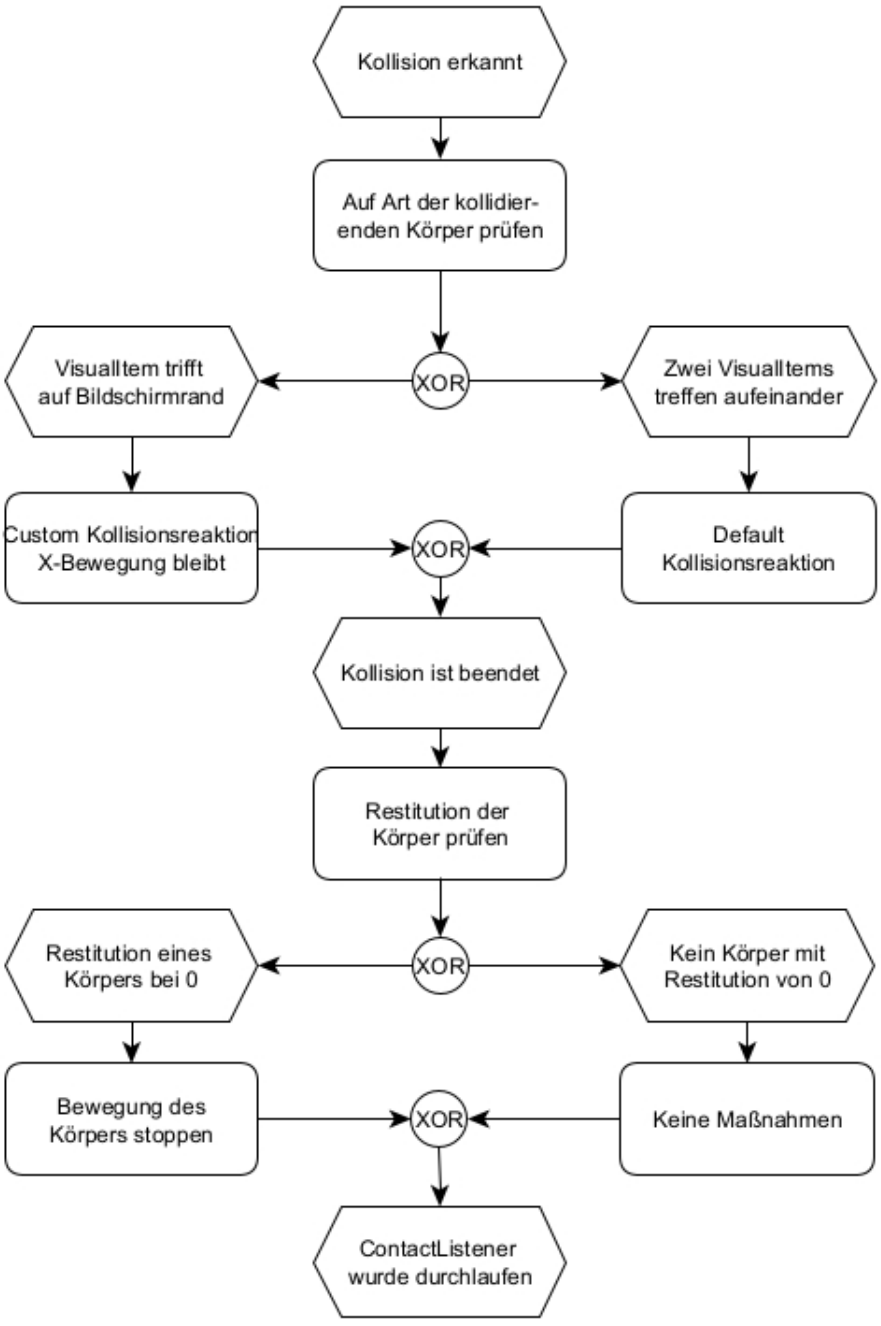

Abbildung 11: Verhalten des *ContactListeners*

5 Implementierung

In diesem Kapitel wird die Implementierung der Physiksimulation beschrieben. Dafür werden die Entwicklungsgrundlagen aufgeführt, der Programmaufbau beschrieben und Herausforderungen und Schwierigkeiten bei der Umsetzung genannt. Zum Schluss werden wichtige Codeausschnitte der Physiksimulation beschrieben.

5.1 Entwicklungsgrundlagen

In diesem Abschnitt werden die Rahmenbedingung bei der Implementierung der Anwendung dargestellt. Dazu wird folgend auf die Programmiersprache, die Entwicklungsumgebung und die verwendete Physik-Engine eingegangen.

5.1.1 Programmiersprache

Als wichtigste Entwicklungsgrundlage gilt die Programmiersprache zu erwähnen. Sie stellt die Grundlage der Implementierung dar. Der Community Mirror ist in Java implementiert. Dabei wird die JDK 11 verwendet. Die Client-Anwendung wird durch das Framework JavaFX realisiert.

Java

Java ist eine objektorientierte Programmiersprache zum Erstellen von Anwendungen. Mit Java erstellte Programme können auf jedem System laufen, dass die Java Laufzeitumgebung JRE samt der Java Virtual Machine implementiert. Neben dem Erstellen von Desktop-Anwendungen wird Java auch überwiegend bei der Entwicklung von Smartphone-Anwendungen auf Android-Systemen verwendet (Steyer, 2014, S. 2ff.). Laut der Online-Plattform Statista aus dem Jahr 2022 liegt Java mit 33,3% auf Platz sechs der meistgenutzten Programmiersprachen von Softwareentwicklern (vgl. `https://www.statista.com/statistics/793628`).

JavaFX

JavaFX ist eine Erweiterung für Java, die dem Erstellen von visuellen Anwendungen auf Basis von Java dient. Der Schwerpunkt von JavaFX liegt bei der bestmöglichen Gestaltung von Benutzerschnittstellen (Steyer, 2014, S. 7). Durch den Scene Builder stellt JavaFX

ein Tool bereit, mit dem grafische Benutzerschnittstellen ohne große Vorkenntnisse in Java oder XML erstellt werden können (Steyer, 2014, S. 15). Die wichtigsten Konzepte von JavaFX in Bezug auf den Community Mirror sind *Stage* (`https://openjfx.io/javadoc/11/javafx.graphics/javafx/stage/Stage.html`), *Scene* (`https://openjfx.io/javadoc/11/javafx.graphics/javafx/scene/Scene.html`), *Node* (`https://openjfx.io/javadoc/11/javafx.graphics/javafx/scene/Node.html`) und *Group* (`https://openjfx.io/javadoc/11/javafx.graphics/javafx/scene/Group.html`) (Kraus, 2022). Die *Stage* ist eine Art Bühne für den Inhalt der Anwendung und enthält ein Objekt der Klasse *Scene*. Die *Scene* enthält alle Objekte, die von der *Stage* gerendert werden sollen. Diese Objekte werden vom Typ *Node* abgeleitet. Eine *Group* enthält eine Liste von Objekten die gerendert werden soll (Kraus, 2022). Im Community Mirror wird zunächst die *Stage* selbstständig von der JavaFX-Runtime erzeugt, nachdem die Anwendung gestartet wird. Das *Scene-Objekt* wird beim Erstellen der *FlowView* erstellt und in der Klasse *Community Mirror* der *Stage* zugeordnet. Die Objekte die gerendert werden sollen, wie beispielsweise *VisualItems* werden als Teil einer *Group* in der *FlowView* der *Scene* zugeordnet (Kraus, 2022).

5.1.2 Entwicklungsumgebung

Eine integrierte Entwicklungsumgebung oder auch IDE stellt einen wichtigen Baustein zur Softwareentwicklung dar. Sie bietet Programmierern eine Sammlung der wichtigsten Werkzeuge zur Softwareentwicklung. Dazu zählen beispielsweise Editor mit Quelltextformatierung und Syntaxhervorhebung, Compiler und Linker, Debugger, Interpreter, Werkzeuge für das Erstellen von grafischen Oberflächen, Versionsverwaltungen und weitere Tools (Luber, 2023). In dieser Arbeit wurde Intellij IDEA der Firma JetBrains als Entwicklungsumgebung verwendet (`https://www.jetbrains.com/de-de/idea/`). Intellij IDEA ermöglichte eine schnelle und einfache Installation der benötigten Software und verfügt über eine integrierte Versionsverwaltung, die für die Entwicklung verwendet wurde. Die gewählte Entwicklungsumgebung unterstützte die Entwicklung mit einem auf Java spezialisierten, ausgeprägten Editor und Möglichkeiten zum Debugging.

5.1.3 Physik-Engine

In dieser Arbeit wurde die Physik-Engine JBox2D in die Anwendung des Community Mirrors integriert. Bei JBox2D handelt es sich um eine Java-Portierung der ursprünglich in C++ entwickelten Physik-Engine Box2D. In diesem Unterabschnitt werden die Kernkonzepte von Box2D und damit auch JBox2D erklärt. Dazu wird das Wiki von Box2D (Catto, 2023) als Leitfaden verwendet. Box2D ist eine zweidimensionale Starrkörpersimulation für Spiele, mit dem Ziel Objekte realistisch zu Bewegen und die Spielwelt interaktiver zu gestalten. Box2D verwendet die in Unterabschnitt 2.2.1 eingeführten Rigid-Body-Dynamics. Ein Festkörper (*Body*) kann zusätzlich über eine in Unterabschnitt 2.2.3 eingeführte Form (*Shape*) verfügen, die

durch eine sogenannte *Fixture* mit dem Körper verbunden wird. Eine *Fixture* kann dem Körper zusätzliche Eigenschaften wie Dichte und Reibung hinzufügen. Durch die *Fixtures* wird eine Form in das Kollisionssystem von Box2D hinzugefügt. Box2D nutzt die Klasse *World*, die eine Sammlung der Körper, *Fixtures* und Gelenken enthält, die von ihr simuliert werden. Um eine schnelle Speicherverwaltung zu gewähleisten, bietet *World* Factory-Methoden zum Erstellen von Körpern und Gelenken. Eine Factory-Methode ist ein Entwurfsmuster, dass eine Schnittstelle zum Erstellen von Objekten durch Aufruf einer Methode, statt dem Aufruf eines Konstruktors ermöglicht (Musch, 2023, S. 205 ff.). Wenn man in Box2D einen Körper erstellen möchte, muss man zunächst eine Definition von diesem erstellen. Diese enthält alle Informationen, die zum Erstellen des Körpers notwendig ist. Die Definitionen können durch das Aufrufen der Factory-Methoden der *World-Klasse* erstellt und in die Physiksimulation eingefügt werden. In Abschnitt 5.4 wird noch eingehender auf das Erstellen von Körpern eingegangen.

5.2 Programmaufbau

In diesem Abschnitt wird der Programmaufbau, der für die Physiksimulation relevant ist, beschrieben. Die Struktur der Physiksimulation wird in Abbildung 12 dargestellt.

FlowView

Die *FlowView* ist für die grafische Darstellung des Community Mirrors verantwortlich. Sie verfügt über die *JavaFX-Scene* und rendert damit die Elemente, die die auf dem Community Mirror dargestellt werden. Zusätzlich erstellt die *FlowView* die Physiksimulation und aktualisiert diese regelmäßig. Sie erstellt *VisualItems* und fügt diese in die Physiksimulation ein.

VisualItems

VisualItems bewegen sich über den Bildschirm und bieten dem Nutzer die Möglichkeit, mit ihnen zu interagieren. Sie verfügen in der Physiksimulation über einen mit ihnen assoziierten Körper, dessen Position sie annehmen. Durch Nutzerinteraktionen oder andere Einflüsse ausgelöste Verhaltensweisen werden durch die *Behaviour-Klassen* gehandelt. Ein *VisualItem* zu dem es verwandte Elemente gibt, kann durch die Klasse *Graph* und andere mit ihr assoziierten Klassen die verwandten Elemente anzeigen lassen.

Behaviours

Behaviours sind Klassen, die Verhaltensweisen der *VisualItems* umsetzen. So zum Beispiel die Interaktionen des Nutzers. Wenn die *Behaviours* Einfluss auf die Position eines *VisualItems* nehmen, wird dessen Körper in der Physiksimulation von JBox2D entsprechend angepasst, woraufhin sich auch die Position des *VisualItems* ändert.

Graph

Die Klassen aus dem Paket Graph sind dafür verantwortlich, die verwandten Elemente eines *VisualItems* anzuzeigen und in der Physiksimulation entsprechende Federgelenke zu erstellen. Des Weiteren sind sie auch dafür verantwortlich, diese wieder auszublenden.

JBox2D

Dieses Paket enthält die Physiksimulation und einen *ContactListener*. In ihr werden Körper und Gelenke erstellt und simuliert. Dabei werden Kollisionen und durch die *Behaviour-Klassen* weitergegebene Aktionen berücksichtigt. Auf dieser Grundlage können *VisualItems* ihre Position bestimmen.

5.3 Herausforderungen und Schwierigkeiten

In diesem Abschnitt werden Herausforderungen und Schwierigkeiten, die während des Projekts aufgetreten sind erwähnt und die entsprechenden Lösungen vorgestellt. Zu den Herausforderungen zählen das Vermeiden einer Überladung des Community Mirrors, den Realismus nicht auf Kosten der *Usability* zu steigern und eine möglichst nahtlose Integration der Physiksimulation zu erreichen.

5.3.1 Überladung Vermeiden

In der Analysephase dieser Arbeit wurden Use-Cases erstellt, die festgelegt haben, welche Features durch die Integration einer Physik-Engine implementiert werden sollen. Die ausgewählte Physik-Engine JBox2D bietet viele Features wie beispielsweise Schwerkraft, Sensoren und zahlreiche Gelenktypen, die in dieser Arbeit nicht verwendet wurden, um das User-Interface nicht zu überladen. Eine Herausforderung dieser Arbeit lag deshalb darin, neue sinnvolle Features in den Community Mirror zu implementieren, ohne diesen zu überladen. Es gibt verschiedene Möglichkeiten, durch die ein User-Interface auf den Nutzer überladen wirken kann und somit seine Zufriedenheit senken könnte. Zwei dieser Möglichkeiten sind Information Overload (Informationsüberflutung) und Choice Overload (Auswahlparadox) (Bollen et al., 2010, S. 63). Da durch die Physiksimulation dem User-Interface keine weiteren Informationen hinzugefügt werden, wird im Weiteren nur das Auswahlparadox betrachtet. Der Begriff Auswahlparadox stammt aus einer Studie zum Kaufverhalten von Kunden. Diese führte zu dem Ergebnis, dass Kunden, die zu viele Auswahlmöglichkeiten haben, lieber keine Kaufentscheidung trafen, als eine falsche zu treffen. Bei Kunden mit einer kleineren Auswahl war die Wahrscheinlichkeit, dass diese etwas kaufen, höher (Iyengar & Lepper, 2001). In einer Studie von 2022 wurde untersucht, inwiefern das Auswahlparadoxon Auswirkungen auf die Nutzerzufriedenheit bei der Interaktion mit User-Interfaces hat (Luber, 2023). Das Ergebnis dieser Studie zeigte, dass User-Interfaces je nach Gestaltung Choice Overload beim Nutzer hervorrufen und damit die Nutzerzufriedenheit

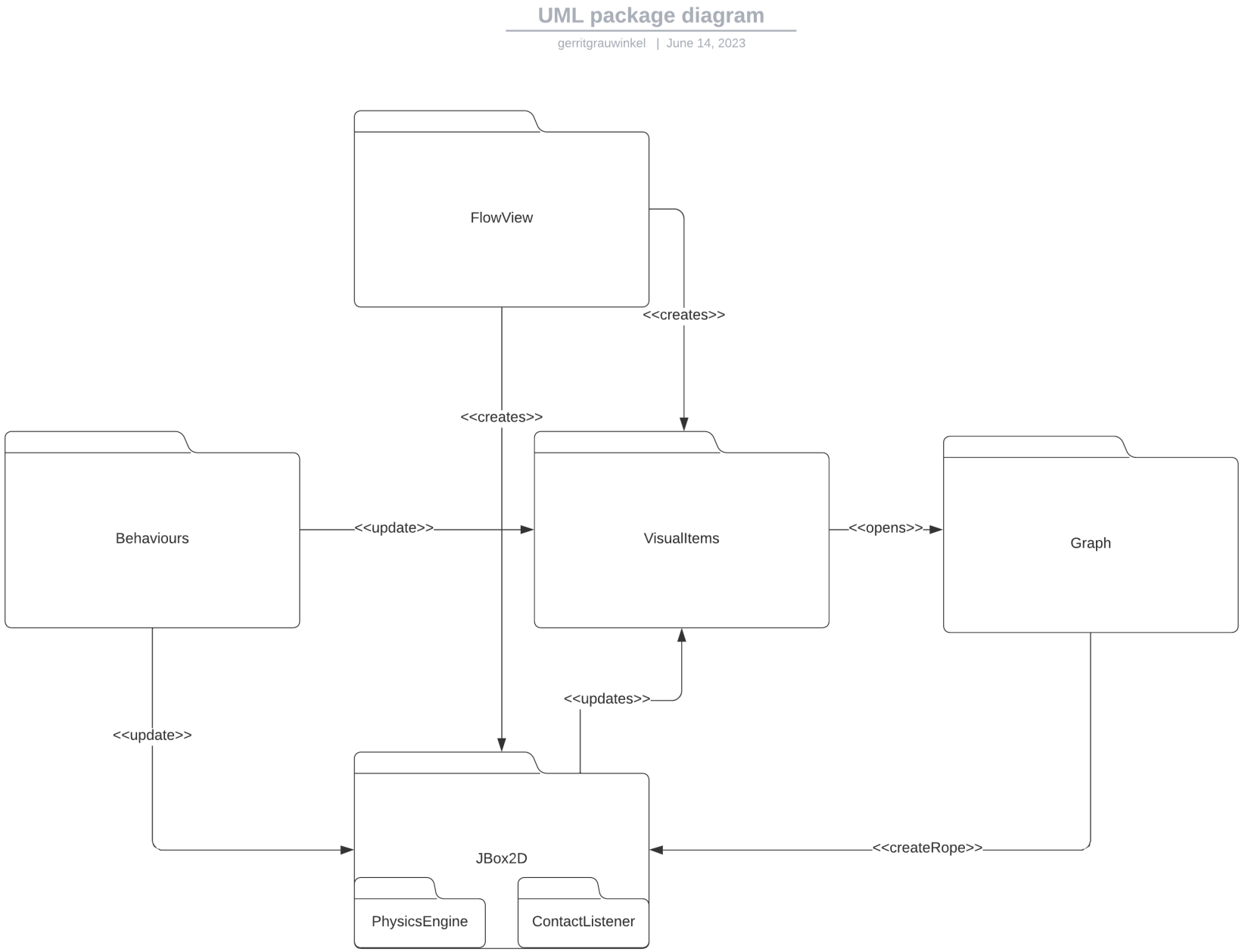

Abbildung 12: Paketdiagramm der Physiksimulation

senken können. An diese Studie anknüpfend sollte es in dieser Arbeit vermieden werden, durch zu viele Gamifications oder neue Interaktionsmöglichkeiten Choice Overload beim Nutzer hervorzurufen. Deshalb wurde sich auf die Einführung von einer Gamification beschränkt und statt neue Interaktionsmöglichkeiten einzuführen, die bereits bestehenden ausgebaut.

5.3.2 Realismus, ohne die *Usability* einzuschränken

In Unterabschnitt 2.3.2 wurde bereits erwähnt, dass Physiksimulationen im User-Interface-Design nicht das Ziel haben, so realistisch wie möglich zu sein, sondern stets die *Usability* wahren sollen. In dieser Arbeit wurde dem Community Mirror ein Basketballkorb als Gamification-Element hinzugefügt. Da der Community Mirror nicht als Basketball-Simulation konzipiert wurde, würde es der *Usability* schaden, wenn es durch Realismus zu schwer wäre, *VisualItems* in den Korb zu befördern. So würde sein eigentlicher Nutzen, der im Entfernen von *VisualItems* aus dem Bildschirm besteht, zu schwierig werden. Um diese Herausforderung zu lösen, wurde das ebenfalls in Unterabschnitt 2.3.2 eingeführte *Force Field* verwendet. Dieses sorgt dafür, dass ein *VisualItem* auf direktem Weg den Bildschirm verlässt, sobald es sich im Bereich des Basketballkorbes aufhält. Nur durch eine Nutzerinteraktion kann das *VisualItem* dann noch aus dem Korb herausgezogen werden. In Abbildung 13 wird zum Verständnis dargestellt, welche Bereich das *Force Field* einnimmt und in welche Richtung es Kräfte ausübt.

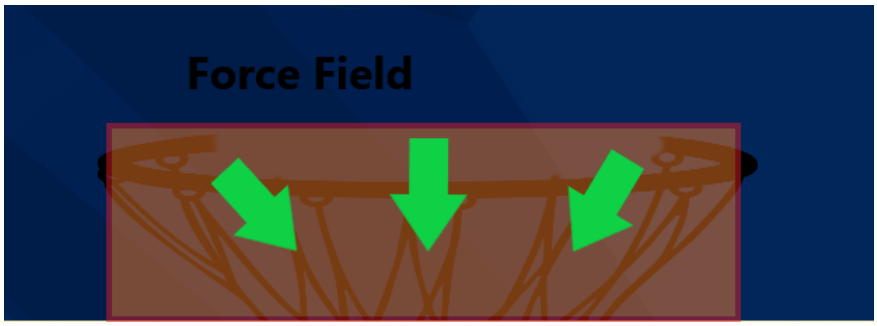

Abbildung 13: *Force Field* für Gamification

5.3.3 Integration

Da in dieser Arbeit keine eigenständige Software entwickelt, sondern stattdessen eine bestehende Software weiterentwickelt wurde, bestand eine Herausforderung darin, sich im bestehenden Projekt zurechtzufinden und die Physiksimulation an den passenden Stellen zu integrieren. Ein Beispiel dafür ist das Hinzufügen der *VisualItems* in die Physiksimulation. Der Großteil dieser wird durch die Methode *addFlowVisualItem* in der *FlowView* erstellt. Deshalb wurde in diese Methode der Methodenaufruf um in der Physiksimulation einen passenden Körper zu erzeugen integriert. Dadurch wurden jedoch nicht alle *VisualItems* in die Simulation aufgenommen, weil diese teilweise auf anderem Wege erstellt werden. So werden *VisualItems*, die durch das

Anklicken der *TeaserComponent* oder durch das Anzeigen verwandter Elemente entstehen, nicht durch die Methode *addFlowVisualItem* erstellt. Dementsprechend wurden für diese Sonderfälle eigene Wege entwickelt, um die *VisualItems* in die Physiksimulation aufzunehmen. Ein Beispiel ist die Methode *addFlowVisualItemToPhysics* welche das durch Anklicken der *TeaserComponent* entstandene *VisualItem* in die Physiksimulation aufnimmt. Das Einbauen solcher Sonderlösungen macht die Implementierung etwas unübersichtlicher, ist aber nicht vermeidbar, wenn die Grundstruktur der Software aufrechterhalten werden soll. Nicht unbedingt eine Schwierigkeit oder Herausforderung, aber trotzdem ein nennenswerter Punkt besteht darin, dass JBox2D und der Community Mirror bei der Angabe von Zahlen unterschiedliche Datentypen nutzen. Der Community Mirror nutzt den Datentyp Double und JBox2D Float. Dementsprechend musste bei Schnittstellen zwischen den vorher existierenden Klassen und der Physiksimulation oft der passende Datentyp gecastet werden.

5.4 Codebeschreibung

In diesem Abschnitt werden die wichtigsten Codeabschnitte der Physiksimulation beschrieben. Dazu gehören die Initialisierung der Physiksimulation, das Erstellen der benötigten Körper und Gelenke, Reaktionen auf Nutzerinteraktionen und der Physik-Engine-Loop. In der Codebeschreibung werden nur Methoden aus den Klassen beschrieben, die im Rahmen dieser Arbeit neu erstellt worden sind. Zusätzlich sind im Zuge dieser Arbeit auch neue Methoden in bereits bestehenden Klassen erstellt worden und einige Methoden wurden auch verändert. Auf alle Änderungen im Detail einzugehen, würde den Rahmen dieser Arbeit sprengen. Um trotzdem einen Überblick auf alle Änderungen zu geben, sind in Unterabschnitt 5.4.5 alle Änderungen mit einer kurzen Beschreibung dokumentiert.

5.4.1 Initialisierung der Physiksimulation

Im folgenden Codeabschnitt ist der Konstruktor der Klasse *PhysicsEngine* dargestellt. Dabei gilt zu beachten, dass die in Zeilen 1-8 bestimmten Attribute eigentlich in der Definition der Klasse bestimmt werden und nur hier zum Zweck der Dokumentation in den Konstruktor aufgenommen wurden. Die *PhysicsEngine* wird von der *FlowView* erstellt und stellt die Hauptkomponente der Physiksimulation dar. Zunächst wird ein Vektor mit der Gravitation der *World* und anschließend die *World* selber erstellt. Da im Community Mirror keine Schwerkraft simuliert werden soll beträgt die Gravitation hier null. Die *physicsTime* stellt die Laufzeit dar und wird für die Physik-Game-Loop benötigt. Zu Beginn wird diese deshalb auf 0 gesetzt. Die Attribute *physicsTimeStep*, *velocityIterations* und *positionIterations* werden von der Physik-Game-Loop verwendet, um Geschwindigkeiten und Positionen von Objekten zu bestimmen. Hier werden empfohlene Werte aus dem Wiki von Box2D verwendet. Anschließend wird die Bildschirmauflösung gespeichert, um Objekte systemunabhängig an der richtigen Stelle positionieren zu können. Als Nächstes werden

mit der Methode *createEdge* die Bildschirmränder erstellt, da diese statisch sind und somit nur zu Beginn der Simulation erstellt werden müssen. Abschließend bekommt die *World* einen *ContactListener* zugeteilt, mit dem sie Kollisionsreaktionen berechnen soll. Nach Durchlaufen der Konstruktor-Methode, verfügt die Physiksimulation nun über ein Grundgerüst, dem Körper und Gelenke hinzugefügt werden können.

```
1   public PhysicsEngine(float width, float height) {
2
3       this.gravity = new Vec2(0, 0);
4       this.world = new World(gravity);
5       this.physicsTime = 0.0f;
6       this.physicsTimeStep = 1.0f / 60.0f;
7       this.velocityIterations = 8;
8       this.positionIterations = 3;
9       screenWidth = width;
10      screenHeight = height;
11      this.world.setContactListener(new ContactListener());
12
13
14      createEdge(new Vec2(0, height * (-0.1f)), new Vec2(width, height * (-0.1f)
            ));
15      createEdge(new Vec2(0, height * 0.87f), new Vec2(width * 0.4f, height *
            0.87f));
16      createEdge(new Vec2(width * 0.68f, height * 0.87f), new Vec2(width,
            height * 0.87f));
17
18  }
```

5.4.2 Erzeugen von Körpern und Gelenken

Für die Physiksimulation der *World* drei Arten von Elementen hinzugefügt werden können. Die statischen Körper für die Bildschirmränder, die dynamischen Körper für die Simulation der *VisualItems* und Federgelenke zum Anzeigen der verwandten Elemente.

Bildschirmränder

Im nachfolgenden Codeabschnitt ist dargestellt, wie die Körper der Bildschirmränder erstellt und in die *World* hinzugefügt werden. Zunächst wird eine *Shape-Instanz* erstellt. Zum Darstellen der Bildschirmränder wird die in JBox2D enthaltene Klasse *EdgeShape* verwendet. Statt einer Kreis- oder Vieleckform stellt die Klasse *EdgeShape* ein Liniensegment zwischen zwei

Punkten dar. Die zwei Positionen zwischen denen das Liniensegment verlaufen soll, bekommt die Methode als Parameter mitgeliefert. Anschließend wird eine Definition für einen statischen *Body* und einer den *EdgeShape* enthaltenden *Fixture* erstellt. Die Position der *BodyDef* ist hier (0, 0) da die Position des Körpers in diesem Kontext irrelevant ist, da die Banden nur zur Kollisionserkennung dienen und dafür die erstellte *EdgeShape-Instanz* ausreicht. Abschließend wird durch eine Factory-Methode von *World* der Festkörper erstellt und bekommt die *Fixture* mit dem Liniensegment zur Kollisionserkennung als Attribut übergeben. Nun ist der erstellte Bildschirmrand ein Teil der Physiksimulation.

```
1   private void createEdge(Vec2 pos1, Vec2 pos2) {
2      EdgeShape edgeShape = new EdgeShape();
3      edgeShape.m_vertex1.set(pos1);
4      edgeShape.m_vertex2.set(pos2);
5
6      BodyDef bodyDef = new BodyDef();
7      bodyDef.type = BodyType.STATIC;
8      bodyDef.position.set(0, 0);
9
10     FixtureDef fixtureDef = new FixtureDef();
11     fixtureDef.shape = edgeShape;
12     fixtureDef.density = 1;
13     fixtureDef.friction = 0.3f;
14
15     Body lowerEdgeBody = this.world.createBody(bodyDef);
16     lowerEdgeBody.createFixture(fixtureDef);
17  }
```

Visual Items

Das Einfügen der *VisualItems* in die Physiksimulation wird unter diesem Absatz abgebildet und funktioniert ähnlich wie das Erstellen der Bildschirmränder. Es werden *Shape*, *FixtureDef* und *BodyDef* erstellt, woraufhin per Factory-Methode die *Body-Instanz* erstellt und der Physiksimulation hinzugefügt wird. Der Körper ist im Gegensatz zu den Bildschirmrändern dynamisch und seine Position wird von dem *VisualItem*, das er repräsentieren soll, übernommen. Als Form wird hier die Klasse *CircleShape* benutzt. Diese stellt einen Kreis dar und ist daher gut geeignet, die kreisförmigen *VisualItems* in der Kollisionserkennung zu repräsentieren. Bei der *FixtureDef* werden einige Attribute festgelegt, die sich auf die Physiksimulation auswirken. Das Attribut *density* legt die Dichte eines Körpers fest. Die Dichte wird von JBox2D zur Berechnung der Masse verwendet (Catto, 2023). Körper mit einem größeren Radius bekommen hier eine größere Dichte zugeordnet, damit sie bei Kollisionen kleinere Körper noch stärker verdrängen, als es

sonst der Fall wäre. Mit dem Attribut *friction* wird die Reibung der Körper auf 0 gestellt, damit sie ihre Geschwindigkeit nicht verlieren. Das Attribut *restitution* wird mit dem Wert 1 belegt. *Restitution* legt fest, wie Objekte auf einen Aufprall reagieren (Catto, 2023). Bei einem Wert von 0 würde der Körper sich bei einer Kollision unelastisch verhalten. Bei einem Wert von 1 hingegen wird die Geschwindigkeit des Körpers exakt reflektiert, was einen elastischen Aufprall darstellt. Im Kontext des Community Mirrors sollen elastische Kollisionen stattfinden, damit die *VisualItems* nach einer gewissen Zeit nicht alle ihre Geschwindigkeit verloren haben. Nachdem der Körper erstellt wurde, wird sein Bewegungsvektor festgelegt. Zu Beginn sollen sich die *VisualItems* nur auf der x-Achse bewegen. Dazu wird ein zufälliger Wert als Geschwindigkeit bestimmt und die Bewegungsrichtung hängt vom Parameter *directionSwitcher* ab, der beim Erstellen des *VisualItems* von der *FlowView* festgelegt wird. Abschließend bekommt das *VisualItem* den für ihn erstellten *Body* durch den Methodenaufruf *item.setRawBody(body)* als Attribut übergeben, um darauf beispielsweise zur Positionsbestimmung zugreifen zu können.

```
1  public void add(FlowVisualItem item, Integer directionSwitcher) {
2
3        BodyDef bodyDef = new BodyDef();
4        bodyDef.type = BodyType.DYNAMIC;
5        bodyDef.position.set((float) item.getPositionX(), (float) item.
             getPositionY());
6
7
8        CircleShape shape = new CircleShape();
9        float scaleX = (float) item.getScaleX();
10       float scaleY = (float) item.getScaleY();
11       float shapeWidth = (0.5f * (float) item.getBoundsInLocal().getWidth()) *
             scaleX;
12       float shapeHeight = (0.5f * (float) item.getBoundsInLocal().getHeight()) *
              scaleY;
13       shape.m_radius = (float) ((shapeHeight * sqrt(2)) * 0.7);
14
15
16       FixtureDef fixtureDef = new FixtureDef();
17       fixtureDef.shape = shape;
18       fixtureDef.density = 1 + shape.m_radius;
19       fixtureDef.friction = 0;
20       fixtureDef.restitution = 1;
21
22
```

```
23        Body body = this.world.createBody(bodyDef);
24        body.createFixture(fixtureDef);
25        double randomNumber = Math.random() * 10;
26        body.setLinearVelocity(new Vec2(-10f * directionSwitcher * (float)
              randomNumber, 0));
27        item.setRawBody(body);
28    }
```

Federgelenke

Federgelenke werden zwischen zwei verschiedenen Körpern erstellt. Deshalb hat die unter diesem Abschnitt abgebildete Methode *createRope* zwei *Body-Instanzen* als Parameter, die jeweils mit einem der zu verbindenden *VisualItem* assoziiert sind. Zum Erstellen der Federgelenke wird die Klasse *RopeJoint* von JBox2D verwendet. Diese Klasse legt einen maximalen Abstand zwischen zwei Körpern fest (Catto, 2023). Ähnlich wie bei Körpern muss auch zum Erstellen von Gelenken in JBox2D zunächst eine entsprechende Definition erstellt werden. In dieser werden die zwei zu verbindenden Körper und ihr maximaler Abstand festgelegt. Durch Setzen des Booleans *collideConnected* auf true wird festgelegt, dass verbundene Körper miteinander kollidieren können. Abschließend wird ein Gelenk durch Aufrufen der Factory-Methode in die Physiksimulation eingefügt.

```
1  public void createRope(Body center, Body child){
2        RopeJointDef ropeJointDef = new RopeJointDef();
3        ropeJointDef.bodyA = center;
4        ropeJointDef.bodyB = child;
5        // Center and child body can collide which eachother.
6       ropeJointDef.collideConnected = true;
7
8       ropeJointDef.maxLength = 350;
9
10      RopeJoint ropeJoint = (RopeJoint) world.createJoint(ropeJointDef);
11    }
```

5.4.3 Physik-Game-Loop

Der unten abgebildete Codeausschnitt der Methode *update* übernimmt die Rolle des Physik-Engine-Loops. Jedes Mal, wenn die *FlowView* ihre Methode *onUpdate* ausführt, um die JavaFX-Anwendung zu aktualisieren, wird auch der Physik-Engine-Loop durchlaufen. Der übergebene Parameter *dt* entspricht der Zeit, die seit dem letzten Aufruf vergangen ist. Bei jedem Aufruf der Methode wird *world.step* aufgerufen. Durch das Aufrufen von *world.step* werden die Körper in

der Physiksimulation anhand der auf sie einwirkenden Kräfte simuliert und Kollisionen erkannt, woraufhin entsprechende Reaktionen ausgelöst werden. An dieser Stelle werden die Vorteile einer Physik-Engine besonders gut ersichtlich. Nachdem Körper und Gelenke in die Simulation eingefügt wurden, reicht ein Aufruf der Methode *world.step* und JBox2D führt die vollständige Physiksimulation durch. Im zweiten Teil der Methode *update* soll geprüft werden, ob sich ein Körper im Basketballkorb befinden. Dazu wird mit *world.getBodyList()* eine Liste angelegt, die alle Körper in der Physiksimulation enthält. Anschließend wird über jeden Körper iteriert. Zunächst wird abgefragt, ob der Körper dynamisch ist. Das führt dazu, dass statische Körper wie die Bildschirmränder nicht berücksichtigt werden. Anschließend wird abgefragt, ob sich die Position des Körpers innerhalb des Basketballkorbs befindet. Ist dies der Fall, wird für den Körper ein neuer Bewegungsvektor bestimmt, durch den er den Bildschirm auf direktem Weg verlassen kann.

```
1  public void update(float dt) {
2        physicsTime += dt;
3        if (physicsTime >= 0.0f) {
4           physicsTime -= physicsTimeStep;
5           world.step(physicsTimeStep, velocityIterations, positionIterations);
6
7           // If a FlowVisualItem is in the Basket it will leave the screen.
8           Body list = world.getBodyList();
9           for (Body b = list; b != null; b = b.getNext()) {
10             if (b.m_type == BodyType.DYNAMIC) {
11                Vec2 position = b.getPosition();
12                if (position.x > screenWidth * 0.42 && position.x < screenWidth *
                       0.64) {
13                   if (position.y > 0.71 * screenHeight && position.y < 0.99 *
                          screenHeight) {
14                      Vec2 targetVelocity = new Vec2((float) (screenWidth * 0.54
                             - position.x), screenHeight - position.y);
15                      b.setLinearVelocity(targetVelocity);
16                   }
17                }
18             }
19          }
20       }
```

5.4.4 ContactListener

Um Kollisionsreaktionen bestimmen zu können, wird von JBox2D bei jeder erkannten Kollision automatisch eine Instanz der Klasse *Contact* erstellt und nach der Kollision wieder gelöscht. *Contact-Instanzen* können nicht manuell erstellt werden. Um auf eine *Contact-Instanz* zuzugreifen, können direkte Abfragen an die Physiksimulation beispielsweise mit der Methode *world.getContactList* gestellt werden. Eine andere Möglichkeit dazu besteht in der Implementierung eines *ContactListeners*. *ContactListener* wird von JBox2D als Interface bereitgestellt (Catto, 2023). Das Interface stellt Methoden zur Verfügung, die während dem Durchlaufen des Physik-Game-Loops bei einer erkannten Kollision aufgerufen werden. Dazu muss der *ContactListener* lediglich durch die Methode *world.setContactListener(ContactListener listener)* in die Physiksimulation eingebunden werden. Unter diesem Textabschnitt ist der Code abgebildet, mit dem in dieser Arbeit das Interface *ContactListener* implementiert wurde. Durch das Interface werden vier Methoden bereitgestellt, die implementiert werden müssen. Diese sind *beginContact*, *endContact*, *preSolve* und *postSolve*. Die beiden zuletzt aufgezählten Methoden werden in dieser Arbeit nicht verwendet und deshalb nicht weiter behandelt.

BeginContact

beginContact wird jedes Mal aufgerufen, wenn eine Kollision erkannt wurde. In der Umsetzung für diese Arbeit wird für beide an der Kollision beteiligten Körper die Form abgefragt. Dadurch lässt sich feststellen, ob es sich bei dem Körper um eine Repräsentation eines *VisualItems* oder des Bildschirmrands handelt. Sollte die Kollision zwischen einer Kreisform (*CircleShape*) und einem Liniensegment (*EdgeShape*) stattgefunden haben, ist also ein *VisualItem* auf den Bildschirmrand gestoßen. Ist dies der Fall, bekommt der Körper, der das *VisualItem* repräsentiert, einen neuen Bewegungsvektor, damit seine Bewegungsrichtung auf der x-Achse sich nicht ändert.

EndContact

Diese Methode wird aufgerufen, sobald die zwei kollidierenden Körper sich nicht mehr berühren. Im Kontext dieser Arbeit wird die Methode dafür genutzt, dass *VisualItems*, die vom Nutzer mit der Maus bewegt werden, nicht durch Kollisionen weggestoßen werden. Wenn der Nutzer mit einem *VisualItem* interagiert, wird für den Zeitraum, in dem das Objekt per Maus verschoben wird, das Attribut *restitution* für die *Body-Instanz* des bewegten *VisualItems* auf 0 gesetzt. Damit soll erreicht werden, dass die Kollisionen mit dem per Maus bewegten *VisualItem* unelastisch sind. Diese Implementierung hat jedoch nicht so funktioniert wie geplant und per Maus bewegte *VisualItems* wurden durch Kollisionen immernoch leicht von der Position des Mauszeigers verschoben. Um dies zu verhindern, wird in der Methode *endContact* abgefragt, ob ein an der Kollision beteiligter Körper für das Attribut *restitution* den Wert 0 hat. Ist dies der Fall, handelt es sich um ein vom Nutzer bewegtes *VisualItem*. Dieser Körper erhält dann einen Nullvektor als Bewegungsvektor, damit er durch die Kollision nicht bewegt wird.

```
1 @Override
2    public void beginContact(Contact contact) {
3       Fixture fixtureA = contact.getFixtureA();
4       Fixture fixtureB = contact.getFixtureB();
5
6       if (fixtureA.getShape() instanceof CircleShape && fixtureB.getShape()
            instanceof EdgeShape){
7         Body body = fixtureA.getBody();
8         body.setLinearVelocity(new Vec2(body.getLinearVelocity().x, body.
              getLinearVelocity().y*(-1)));
9       }
10      if (fixtureB.getShape() instanceof CircleShape && fixtureA.getShape()
            instanceof EdgeShape){
11         Body body = fixtureB.getBody();
12         body.setLinearVelocity(new Vec2(body.getLinearVelocity().x, body.
               getLinearVelocity().y*(-1)));
13      }
14   }
15
16   @Override
17   public void endContact(Contact contact) {
18      Fixture fixtureA = contact.getFixtureA();
19      Fixture fixtureB = contact.getFixtureB();
20      if (fixtureA.getRestitution() == 0f){
21         fixtureA.getBody().setLinearVelocity(new Vec2(0,0));
22      }
23      if (fixtureB.getRestitution() == 0f){
24         fixtureB.getBody().setLinearVelocity(new Vec2(0,0));
25      }
26   }
```

5.4.5 Neue und geänderte Methoden in bestehenden Klassen

In diesem Unterabschnitt soll ein Überblick auf alle Änderungen in den bereits bestehenden Klassen gegeben werden. Dazu wird in Tabelle 1 abgebildet, welche neuen Methoden in bestehenden Klassen hinzugefügt wurden und in Tabelle 2 wird dargestellt, welche Methoden überarbeitet wurden. Für jede Methode wird die Klasse angegeben, zu der sie gehört und ihre Funktion prägnant beschrieben.

Klasse	Methode	Beschreibung
FlowView	addFlowVisualItemtoPhysics	Ruft in der Klasse PhysikEngine die Methode add auf.
FlowView	createJointInPhysics	Ruft die Methode createRope in der Klasse PhysicsEngine auf.
FlowVisualItem	getRawBody	Getter Methode um die Body-Instanz eines FlowVisualItems zu erhalten.
FlowVisualItem	setRawBody	Setter für die zugehörige Body-Instanz eines FlowVisualItems.
FlowVisualItem	onUpdate	Eigene Update Methode eines FlowVisualItems, durch die es seine Position verändert.

Tabelle 1: Übersicht der neuen Methoden in bereits bestehenden Klassen

Klasse	Methode	Neuerung
FlowView	onInit	Erstellt nun zusätzlich eine Instanz der PhysicsEngine und fügt den Basketballkorb in die View ein.
FlowView	OnUpdate	Ruft nun den Physik-Game-Loop auf.
FlowView	addFlowVisualItem	Ruft beim Erstellen eines FlowVisualItems die add Methode in der Klasse PhysicsEngine auf.
FlowVisualItem	destroy	Zerstört nun auch den zum FlowVisualItem zugehörigen Body.
DragBehaviour	handleMouseDragged	Durch die Maus bewegte Objekte können nicht durch Kollisionen weggestoßen werden.
DragBehaviour	setNewPosition	Die Positionsänderung wird an die Klasse PhysicsEngine übergeben.
DragBehaviour	handleMouseReleased	Der Bewegungsvektor wird an die Klasse PhysicsEngine übergeben, anstatt eine DriftBehaviour auszulösen.
TouchCreate FlowItem-Behavior	doTouchAction	Das erstellte VisualItem bekommt durch den Methodenaufruf von addFlowVisualItemtoPhysics einen *Body* erstellt und zugewiesen.
Graph	openGraph	Federn werden nun durch die Klasse PhysicsEngine erstellt.
Graph	closeGraph	Federn werden nun über JBox2D Methoden zerstört.

Tabelle 2: Übersicht der überarbeiteten Methoden

6 Evaluation

In diesem Kapitel soll das Resultat der Umsetzung evaluiert werden. Die Evaluation von Software dient dem Testen und ggf. dem Verbessern einer Benutzerschnittstelle im Kontext der *Usability* (Hegner, 2003, S. 7). Es gibt mehrere unterschiedliche Evaluationsmethoden, die sich in ihrer Art voneinander unterscheiden. Es wird zwischen objektiven und subjektiven Evaluationsmethoden unterschieden (Hegner, 2003, S. 15). Subjektive Evaluationsmethoden knüpfen an subjektiven Eindrücken von einzelnen Personen an. Objektive Evaluationsmethoden versuchen subjektive Einflüsse auszuschalten und verwenden quantitative Daten, wie beispielsweise Ausführungszeiten und Fehlerraten (Hegner, 2003, S. 15 f.). Zusätzlich zu der eben genannten Unterscheidung von Evaluationsmethoden, kann auch zwischen einer Evaluation mit- und ohne Nutzerbeteiligung unterschieden werden (Riihiaho et al., 2000, S. 7). Evaluationen ohne Nutzerbeteiligung sind nicht-empirisch und liefern daher weniger valide Resultate. Ihr Vorteil liegt darin, dass sie weniger Vorbereitungszeit brauchen und sich leicht in einen Entwicklungsprozess integrieren lassen. Sie dienen nicht dazu, empirische Methoden zu ersetzen, sondern sollen diese ergänzen (Riihiaho et al., 2000, S. 9).

Die Evaluation dieser Arbeit soll mehrere der eben genannten Bereiche abdecken und setzt sich deshalb folgendermaßen zusammen: Der erste Teil der Evaluation dieser Arbeit besteht daraus, die ursprünglichen Use-Cases mit dem tatsächlichen Resultat zu vergleichen. Ein weiterer Teil der Evaluation besteht aus der Auswertung eines Experiments, in dem eine Datenerhebung über Nutzerverhalten durchgeführt wurde. Zusätzlich zu der Datenerhebung wurden auch Umfragen durch Nutzer ausgefüllt. Dementsprechend beinhaltet die folgende Evaluation einen subjektiven Teil ohne Nutzerbeteiligung und einen sowohl objektiven als auch subjektiven Teil mit Nutzerbeteiligung.

6.1 Reflexion der Umsetzung

In diesem Abschnitt wird ein Überblick darauf gegeben, inwiefern die Use-Cases umgesetzt wurden und wie gut die neuen Features funktionieren.

6.1.1 Kollisionen

Durch Kollisionserkennung und davon ausgehende Reaktionen sollte der durch den Nutzer empfundene Realismus erhöht werden. Durch die Integration von JBox2D wurde dieser Use-Case erfüllt. Kollisionen werden erkannt und auf Grundlage von physikalischen Werten wie

Masse, Geschwindigkeit und Winkel des Aufpralls werden Reaktionen eingeleitet. Diese treten optisch auch tatsächlich dann ein, wenn sich die Ränder von zwei *VisualItems* berühren. Auch der Sonderfall bei der Kollision eines *VisualItems* mit dem Bildschirmrand wurde erfolgreich implementiert. Auch wenn diese Kollisionsreaktion nicht physikalischen Berechnungen entspricht, trägt sie dazu bei, dass ein gewisser Bewegungsfluss erhalten bleibt. Ein Bug, der vereinzelt auftritt, besteht darin, dass ein *VisualItem* manchmal an einem unsichtbaren Körper abprallt. Ein weiterer Nachteil, der durch die Kollisionen verursacht wird, liegt darin, dass ein *VisualItem*, das gerade erstellt wurde, durch eine Kollision möglicherweise wieder direkt aus dem Bildschirm befördert wird. Dies stellt jedoch im Kontext des Community Mirrors kein Problem dar, da *VisualItems* mit gleichem Inhalt nach einiger Zeit wieder auf dem Bildschirm erscheinen.

6.1.2 Gamification

Da die Auswirkungen der Gamification im zweiten Teil der Evaluation noch ausgiebig behandelt wird, soll an dieser Stelle nur eine kurze Bewertung stattfinden. Der Korb macht mögliche Nutzer auf die Interaktionsmöglichkeit mit dem Community Mirror aufmerksam, da einige *VisualItems* per Zufall in ihm landen. Des Weiteren wird auch die Funktion als Papierkorb erfüllt.

6.1.3 Gelenke und Federn

Die Federgelenke durch die *VisualItems* mit ihren verwandten Elementen verbunden werden, nachdem der Nutzer sich anklickt, sollten durch die Integration in JBox2D realistischer gestaltet werden. Durch die eingeführten Kollisionen wird verhindert, dass sich die verwandten Elemente überschneiden. Dadurch wurde die Orientierung auf dem Community Mirror verbessert. Die Gelenke werden wie gewollt durch die Physik-Engine simuliert und tragen somit zu einem realistischeren Verhalten bei.

6.1.4 Nicht-funktionale-Anforderungen

Durch die Integration von JBox2D ist die Performance nicht merkbar zurückgegangen. Da wo die Möglichkeit bestand, wurden Funktionen von den bestehenden Klassen weiterhin verwendet. Die einzige Klasse, die durch die Integration von *JBox2D* keinen Nutzen mehr hat, ist *Drift-Behaviour*. Somit hat sich außerhalb des Bereichs der Positionsbestimmung nichts am Code geändert. Dementsprechend ist die Wartbarkeit der Software weiterhin gewährleistet und die neu eingefügten Codeabschnitte werden durch die Artefakte in dieser Arbeit dokumentiert. Im Zuge des im nächsten Abschnitt beschriebenen Experiments wurde die Software des Community Mirrors auf die in dieser Arbeit entstandene Version upgedatet. Dementsprechend erfüllt die Lösung auch die Anforderung, für Installation und Betrieb auf dem Community Mirror geeignet zu sein.

6.2 Datenerhebung durch Experiment

Die durch diese Arbeit im Community Mirror integrierte Physiksimulation wurde im Kontext eines Forschungsprojekts von einer Gruppe aus Studenten verwendet, um ein Experiment durchzuführen (Buhl et al., 2023). In diesem Experiment sollte ergründet werden, wie Gamifications genutzt werden können, um Interaktionen mit halböffentlichen-Wandbildschirmen zu erhöhen. Die Gamification, mit der das Experiment durchgeführt wurde, ist der Basketballkorb, der durch diese Arbeit entstanden ist. Dieser wird von allen in dieser Arbeit erstellten Features beeinflusst. Deshalb können die Ergebnisse aus dem Experiment dazu dienen, Thesen über den Einfluss von physikalisch simulierten Elementen auf die Nutzerfreundlichkeit im Kontext von User-Interfaces zu unterstützen bzw. zu widerlegen.

6.2.1 Versuchsbeschreibung

Das Ziel des Versuchs bestand darin, durch das Implementieren einer Gamification mehr Nutzerinteraktionen mit dem Community Mirror zu erzielen. Dafür wurde die im Zuge dieser Arbeit erstellte Software an einem Community Mirror im Gebäude des Instituts für Softwaretechnologie an der Universität der Bundeswehr in München installiert. Der Community Mirror sammelt bereits selbstständig Interaktionsdaten wie Bildschirmberührungen und das Ziehen von Objekten. Das Experiment lief über einen Zeitraum von 21 Tagen, in denen entsprechende Interaktionsdaten aufgenommen wurden und zusätzlich noch einige Nutzer Umfragen ausgefüllt haben. Die Ergebnisse der Interaktionsdaten wurden nach der Laufzeit des Experiments mit den Interaktionsdaten aus der Zeit davor verglichen. Die Daten aus den ersten drei Tagen des Versuchs wurden nicht berücksichtigt, um durch den Neuheitseffekt verursachte Interaktionen möglichst auszublenden.

6.2.2 Gesammelte Daten

Das Ergebnis der Daten bestand darin, dass die Interaktionen deutlich gestiegen sind. Dabei ist im täglichen Durchschnitt der Bildschirm öfters berührt worden und Objekte wurden öfters über den Bildschirm gezogen, als in dem Zeitraum vor dem Experiment. So stieg die Anzahl der durchschnittlichen täglichen Bildschirmberührungen von 105,77 auf 287,75 an. Der Tagesdurchschnitt von durch Nutzer gezogene Objekte hat sich von 35,91 auf 214,19 erhöht.

6.2.3 Geführte Umfragen

Zusätzlich zu den automatisch generierten Daten wurden auch Umfragen mit insgesamt 13 Nutzern durchgeführt, die täglich an dem Community Mirror vorbeigehen. Von den Befragten haben nur vier Personen die Neuerung wahrgenommen. Acht der befragten Personen gaben an, grundsätzlich nicht mit dem Community Mirror zu interagieren, da sie keinen Grund dazu sehen. Die Personen, welche die Änderung wahrgenommen haben, gaben an sich mit dieser

auseinandergesetzt zu haben. Zusätzlich haben sie auch mit Arbeitskollegen über die Neuerung gesprochen. Des Weiteren ergab die Umfrage, dass die durchschnittliche Zahl an Interaktionen pro Person mit der Gamification zwischen zwei und fünf lag.

6.2.4 Auswertung

In der Auswertung des Versuchs werden im Folgenden zunächst die gezogenen Schlüsse der Forschergruppe genannt. Anschließend wird erörtert, inwiefern sich die Ergebnisse auf die Forschungsfrage dieser Arbeit anwenden lassen.

Auswertung der Gruppe

Die Forschergruppe kommt in der Auswertung ihres Versuchs zu der Erkenntnis, dass die gestiegene Interaktionszahl ein Indiz für eine größere Motivation der Nutzer mit dem Bildschirm darstellt. Ein weiteres Indiz liegt darin, dass mit dem Ziehen von Objekten die Interaktionsart, die für das Interagieren mit der Gamification benötigt wird, mit 496% deutlich stärker gestiegen ist, als das Berühren des Bildschirms mit 172%. Die Gruppe kommt zu dem Fazit, dass durch Einführen einer Gamification Passanten dazu motiviert werden können, mit dem Bildschirm zu interagieren. Auch aus den Umfragen wurde erkenntlich, dass alle Personen, die das Feature wahrgenommen haben, mit diesem interagiert haben. Den Grund dafür, dass nur weniger als die Hälfte der befragen Personen die Gamification wahrgenommen haben, sieht die Gruppe in der visuellen Gestaltung ebendieser.

Anwendung der Ergebnisse auf diese Arbeit

Da der Versuch rein auf der Bestimmung des Effekts von Gamifications ausgelegt war, wurden andere in dieser Arbeit implementierten Features bei der Versuchsauswertung nicht berücksichtigt. Da diese während der Durchführung des Versuchs, auf dem Community Mirror jedoch vorhanden waren, können anhand der gesammelten Daten durchaus Schlüsse auf den Nutzen der anderen Features gezogen werden.

Ein Use-Case dieser Arbeit bestand darin, die Nutzerinteraktionen durch eine Gamification zu erhöhen. In Unterabschnitt 2.3.2 wurde beschrieben, dass das Einführen von spielerischen Elementen eine geeignete Möglichkeit ist, die Nutzerzufriedenheit zu erhöhen. Die Ergebnisse des Versuchs zeigen, dass eine Gamification dazu geeignet ist, Interaktionszahlen zu erhöhen. Da alle befragten Nutzer, die die Gamification wahrgenommen haben, sich mit ihren Kollegen darüber ausgetauscht haben und im Schnitt zwei bis fünf Mal mit ihr interagiert haben, deutet dies darauf hin, dass sie das Feature gut aufgenommen haben und ihre Nutzerzufriedenheit bei der Interaktion mit dem Community Mirror dadurch gesteigert wurde.

Ein weiterer Use-Case dieser Arbeit bestand darin, durch die Physiksimulation den wahrgenommenen Realismus des Nutzers zu erhöhen. Dazu sollten physikalisch simulierte Kollisionen implementiert werden. Auch wenn in den Umfragen nicht auf den empfundenen Realismus

eingegangen wurde, haben nur ein kleiner Teil der Befragten die Gamification wahrgenommen. Demnach ist es durchaus möglich, dass die gestiegene Zahl der Nutzerinteraktionen auch damit zusammenhängt, dass Nutzer, welche die Gamification nicht direkt erkannt haben, den Community Mirror durch die physikalisch simulierten Elemente als realistischer wahrgenommen haben und somit eher mit diesem interagierten.

Der letzte funktionale Use-Case dieser Arbeit bestand darin, die Orientierung auf dem Bildschirm durch den verbesserten Einsatz von physikalisch simulierten Federgelenken zu verbessern. Wenn ein Nutzer ein *VisualItem* interessant findet und weitere Informationen dazu haben möchte, muss er dieses anklicken. Dementsprechend spricht die Anzahl der Bildschirmberührungen dafür, wie oft nähere Informationen zu einem *VisualItem* angefragt wurden. Die Steigerung dieser Interaktionsart um 172% bedeutet also, dass Nutzer deutlich häufiger weitere Informationen zu einem *VisualItem* sehen wollten. Dies könnte darauf hinweisen, dass durch das verbesserte Anzeigen von verwandten Elementen dem Nutzer die Orientierung auf dem Bildschirm leichter fällt und er dementsprechend durchschnittlich mehr *VisualItems* anklickt, als es vor dem Versuch der Fall war.

Zusammenfassend können die gestiegenen Interaktionen mit dem Bildschirm also als Indiz für die positive Wirkung der Physiksimulation auf die Nutzerzufriedenheit gesehen werden. Auf Grundlage des Versuchs lässt sich nicht bestimmen, welches Feature welchen Einfluss auf die Nutzerzufriedenheit hatte. In Kontext dieser Arbeit wäre das jedoch schwierig, da die Features untereinander auch Wechselwirkungen haben und somit nicht isoliert betrachtet werden können. Das Gesamtpaket, welches in dieser Arbeit erstellt wurde, scheint jedoch gut geeignet, um die Nutzerzufriedenheit im Kontext des Community Mirrors zu erhöhen.

7 Zusammenfassung und Fazit

Das Ziel dieser Arbeit bestand darin, herauszufinden, ob die Integration einer Physiksimulation in ein User-Interface dazu geeignet ist, dessen Nutzerfreundlichkeit zu erhöhen. Dazu wurde eine Literaturrecherche durchgeführt, um theoretische Grundlagen über die Wirkung von Physik im Interface-Design aufzubauen. Anschließend wurde auf dieser Grundlage ein Konzept erstellt, eine Physiksimulation in die Software des Community Mirrors zu integrieren, der als halb-öffentlicher Wandbildschirm ein geeignetes User-Interface zum Untersuchen der Forschungsfrage darstellt. Das erstellte Konzept wurde implementiert und auf einem Community Mirror installiert. In Kooperation mit einer Forschergruppe aus Studenten wurde ein Versuch durchgeführt, um das Ergebnis der Implementierung zu evaluieren. Im Folgenden werden zunächst die Ergebnisse dieser Arbeit zusammengefasst. Um die Arbeit abzuschließen, werden die Ergebnisse und Methoden diskutiert und anschließend ein Ausblick auf mögliche weitere Forschungen über das Thema Physik im User-Interface-Design gegeben.

7.1 Zusammenfassung der Ergebnisse

In Kapitel 2 wurde Physik-Engines als geeignete Möglichkeit vorgestellt, um Entwickler beim Programmieren einer Physiksimulation zu unterstützen. Kernelemente einer Physik-Engine sind Rigid-Body-Dynamics, Kräfte und Drehmoment, Erkennung von Kollisionen, Federn und Gelenke sowie der Physik-Engine-Loop. Im Bereich der Nutzerfreundlichkeit kann Physik Auswirkungen auf die *Usability* durch bessere Orientierung und höhere Intuitivität der Interaktion haben. Zusätzlich kann ein durch Physik gesteigerter Realismus dem Nutzer positive Emotionen vermitteln, wodurch dieser eher mit dem User-Interface interagiert. Darauf aufbauend wurde in Kapitel 3 festgelegt, dass physikalisch berechnete Kollisionen zum Steigern des Realismus implementiert werden sollen und die Orientierung des Nutzers durch eine bessere Implementierung von Federn und Gelenken erreicht werden soll. Um den Nutzer zusätzlich zur Interaktion mit dem Community Mirror zu motivieren und positive Emotionen bei ihm auszulösen, wurde beschlossen eine Gamification zu implementieren, welche die Physiksimulation nutzt. In Kapitel 4 wurde JBox2D als Physik-Engine ausgewählt, da sie aufgrund ihrer guten Dokumentation und Einsteigerfreundlichkeit leicht zu nutzen ist und trotzdem gute Ergebnisse erreicht, wie an den vielen Referenzen ihrer Nutzung ersichtlich wird. Bei der Erstellung des Konzepts für die Integration der Physiksimulation wurde ersichtlich, dass die Gestaltung der Schnittstellen zwischen Physik-Engine und bestehender Software den Schwerpunkt der Integration darstellt.

Kapitel 5 beschreibt, wie die Physiksimulation implementiert wurde. Dabei wurden die in Kapitel 2 genannten Vorteile einer Physik-Engine ersichtlich. Nachdem der Physik-Engine-Loop in die JavaFX-Anwendung integriert war, wurde die Physiksimulation fast vollständig durch die Physik-Engine übernommen. Vor allem im Bereich Skalierbarkeit stellt eine Physik-Engine eine gute Alternative zum selbstständigen Entwickeln einer Physiksimulation dar. Herausforderungen beim Integrieren der Physiksimulation bestanden darin, eine Überladung des User-Interfaces zu vermeiden, Realismus nicht auf Kosten der *Usability* zu erzeugen und saubere Schnittstellen zwischen der bestehenden Anwendung und der Physik-Engine zu implementieren. Um das Resultat der Implementierung zu evaluieren, wurden in Kapitel 6 mehrere Methoden verwendet, um die implementierten Änderungen hinsichtlich Nutzerfreundlichkeit zu bewerten. Das Resultat dieser Evaluation besteht darin, dass die in dieser Arbeit erstellten Änderungen am Community Mirror einen positiven Effekt auf die Nutzerfreundlichkeit hatten. Vor allem die Zahl der Interaktionen konnte stark erhöht werden. Dementsprechend lässt sich die Frage, ob physikalisch simulierte Elemente in einem User-Interface die Nutzerfreundlichkeit steigern, für den Bereich halb-öffentlicher Wandbildschirme bejahen.

7.2 Diskussion und Ausblick

In dieser Arbeit wurde die positive Wirkung einer Physiksimulation auf die Nutzerfreundlichkeit eines User-Interfaces am Beispiel eines halb-öffentlichen Wandbildschirms demonstriert. Dabei bleiben jedoch noch einige Fragen offen. In der Evaluation wird nicht berücksichtigt, welches neue Feature die Nutzer zu einer erhöhten Zahl an Interaktionen verleitet hat. Dementsprechend zeigt die Datenerhebung zwar, dass das Gesamtpaket an Features einen positiven Effekt hat, jedoch nicht die unterschiedlichen Wirkungen der einzelnen Features. Um die Auswirkungen von einzelnen Features einer Physiksimulation zu untersuchen, könnten in Zukunft ggf. noch weitere Versuche durchgeführt werden.

Für diese Arbeit wurde der Community Mirror als Testobjekt gewählt. Dieser stellt einen halb-öffentlichen Wandbildschirm zum Abbilden von Informationen dar. Da User-Interfaces eine Vielzahl von Anwendungsgebieten haben, kann dies bedeuten, dass physikalisch simulierte Elemente auf ein User-Interface in einem anderen Anwendungsbereich nicht die gleichen Auswirkungen haben. Deshalb lassen sich die Ergebnisse dieser Arbeit nur im Kontext halb-öffentlicher Wandbildschirme anwenden. Dementsprechend könnte in der Zukunft untersucht werden, welche Auswirkungen physikalisch simulierte Elemente auf andere Arten von User-Interfaces haben.

Die Nutzerfreundlichkeit bei der Gestaltung von User-Interfaces spielt eine wichtige Rolle. Ein Softwareprodukt, das sich gegen konkurrierende Produkte durchsetzen soll, muss die Nutzer auf möglichst viele Arten positiv ansprechen. In dieser Arbeit wurde am Beispiel halb-öffentlicher Wandbildschirme gezeigt, wie eine Physiksimulation dazu beitragen kann und dass es sich durchaus lohnen könnte, weitere Forschung in diesem Bereich durchzuführen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

Agarawala, A., & Balakrishnan, R. (2006). Keepin' It Real: Pushing the Desktop Metaphor with Physics, Piles and the Pen, 1283–1292. ISBN: 1595933727. https://doi.org/10.1145/1124772.1124965

Baba, S. A., Hussain, H., & Embi, Z. C. (2007). An overview of parameters of game engine [Zuletzt besucht am 30.05.2023]. *IEEE Multidisciplinary Engineering Education Magazine*, *2*(3), 10–12. https://www.researchgate.net/profile/Hanafizan-Hussain/publication/265105409_An_Overview_for_Parameters_of_Game_Engine/links/55f23f8e08aef559dc49357c/An-Overview-for-Parameters-of-Game-Engine.pdf

Bollen, D., Knijnenburg, B. P., Willemsen, M. C., & Graus, M. Understanding Choice Overload in Recommender Systems. In: In *Proceedings of the Fourth ACM Conference on Recommender Systems*. RecSys '10. Barcelona, Spain: Association for Computing Machinery, 2010, 63–70. ISBN: 9781605589060. https://doi.org/10.1145/1864708.1864724.

Bourg, D. M. (2002). *Physics for game developers: Enriching game content with physics-based realism* (1. ed.). O'Reilly. ISBN: 0596000065.

Broy, M., & Kuhrmann, M. (2021). *Einführung in die Softwaretechnik*. Springer Berlin Heidelberg. ISBN: 978-3-662-50263-1. https://doi.org/10.1007/978-3-662-50263-1_5

Buhl, W., Buller, V., & Engel, K.-F. Gaming for attention – How gamification can be used to increase interaction with public displays (P. Fröhlich & V. Cobus, Hrsg.) [In Begutachtung]. In: *Mensch und Computer 2023 – Workshopband* (P. Fröhlich & V. Cobus, Hrsg.). Hrsg. von Fröhlich, P., & Cobus, V. In Begutachtung. Bonn, Deutschland: Gesellschaft für Informatik e.V., 2023. https://doi.org/10.18420/muc2023-mci-ws13-344.

Catto, E. (2023). Box2D A 2D physics engine for games [Zuletzt besucht am 30.05.2023]. https://box2d.org/documentation

Deterding, S., Björk, S. L., Nacke, L. E., Dixon, D., & Lawley, E. Designing Gamification: Creating Gameful and Playful Experiences. In: In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '13. Paris, France: Association for Computing Machinery, 2013, 3263–3266. ISBN: 9781450319522. https://doi.org/10.1145/2468356.2479662.

Gonzales-Scheller, P. (2013). Trendthema Gamification: Was steckt hinter diesem Begriff? In J. Diercks & K. Kupka (Hrsg.), *Recrutainment: Spielerische Ansätze in Personalmarketing und -auswahl* (S. 33–51). Springer Fachmedien Wiesbaden. ISBN: 978-3-658-01570-1. https://doi.org/10.1007/978-3-658-01570-1_3

Hassan, H. M., & Galal-Edeen, G. H. (2017). From usability to user experience, 216–222. https://doi.org/10.1109/ICIIBMS.2017.8279761

Hegner, M. (2003). *Methoden zur Evaluation von Software* (Bd. 29). Informationszentrum Sozialwissenschaften.

Iyengar, S., & Lepper, M. (2001). When Choice is Demotivating: Can One Desire Too Much of a Good Thing? *Journal of personality and social psychology*, *79*, 995–1006. https://doi.org/10.1037/0022-3514.79.6.995

Joachim, D., & Kupka, K. (2013). Recrutainment – Bedeutung, Einflussfaktoren und Begriffsbestimmung. In J. Diercks & K. Kupka (Hrsg.), *Recrutainment: Spielerische Ansätze in Personalmarketing und -auswahl* (S. 1–18). Springer Fachmedien Wiesbaden. ISBN: 978-3-658-01570-1. https://doi.org/10.1007/978-3-658-01570-1_1

Kim, J.-S., & Park, J.-M. Physics-based hand interaction with virtual objects. In: In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, 3814–3819. https://doi.org/10.1109/ICRA.2015.7139730.

Koch, M., Ziegler, J., Reuter, C., Schlegel, T., & Prilla, M. (2020). Mensch-Computer-Interaktion als Zentrales Gebiet der Informatik – Bestandsaufnahme, Trends und Herausforderungen. *Informatik Spektrum*, *43*(6), 381–387. https://doi.org/10.1007/s00287-020-01299-8

Kraus, M. (2022). Mensch-Computer-Interaktion Home [zuletzt besucht am 05.06.2023]. https://publicwiki.unibw.de/display/MCI/Mensch-Computer-Interaktion+Home

Kumar, V. (2023). 19 best physics games of 2023: For all platforms [zuletzt besucht am 05.06.2023]. https://www.rankred.com/best-physics-game/

Liu, C. K., & Zordan, V. B. Natural User Interface for Physics-Based Character Animation (J. M. Allbeck & P. Faloutsos, Hrsg.). In: *Motion in Games* (J. M. Allbeck & P. Faloutsos, Hrsg.). Hrsg. von Allbeck, J. M., & Faloutsos, P. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, 1–14. ISBN: 978-3-642-25090-3.

Luber, S. (2023). *Was ist eine ide?* [Zuletzt besucht am 16.06.2023]. https://www.dev-insider.de/was-ist-eine-ide-a-399da9ffb32c9a1e2796c436c6c66061/

McDirmid, S. (2009). The Magic of UI Physics [zuletzt besucht am 19.06.2023]. https://www.microsoft.com/en-us/research/publication/the-magic-of-ui-physics/

Millington, I. (2007). *Game Physics Engine Development* (First edition). CRC Press. ISBN: 978-0123819765.

Moser, C. (2012). User Experience Design. In *User Experience Design: Mit erlebniszentrierter Softwareentwicklung zu Produkten, die begeistern* (S. 1–22). Springer Berlin Heidelberg. ISBN: 978-3-642-13363-3. https://doi.org/10.1007/978-3-642-13363-3_1

Musch, O. (2023). *Design Patterns with Java : An Introduction*. Springer Fachmedien Wiesbaden. ISBN: 978-3-658-39829-3. https://doi.org/10.1007/978-3-658-39829-3_16

Richter, M. (1997). Kriterien der Benutzerfreundlichkeit [Zuletzt besucht am 16.06.2023]. http://www.michaelrichter.ch/literat_97.pdf

Richter, M., & Flückiger, M. D. (2013). *Usability Engineering kompakt: benutzbare Produkte gezielt entwickeln*. Springer. https://doi.org/https://doi.org/10.1007/978-3-8274-2329-0

Riihiaho, S., et al. (2000). Experiences with usability evaluation methods [zuletzt besucht am 19.06.2023]. *Licentiate thesis. Helsinki university of technology. Laboratory of information processing science*. https://www.researchgate.net/profile/Sirpa-Riihiaho/publication/2485077_Experiences_with_Usability_Evaluation_Methods/links/5575834f08aeb6d8c01968d4/Experiences-with-Usability-Evaluation-Methods.pdf

Serrano, H. (2017). How does a physics engine work? an overview - harold serrano - game engine developer [Zuletzt besucht am: 30.05.2023]. https://www.haroldserrano.com/blog/how-a-physics-engine-works-an-overview

Sinha, G., Shahi, R., & Shankar, M. Human Computer Interaction. In: In *2010 3rd International Conference on Emerging Trends in Engineering and Technology*. 2010, 1–4. https://doi.org/10.1109/ICETET.2010.85.

Staggers, N. (1993). Impact of screen density on clinical nurses' computer task performance and subjective screen satisfaction. *International Journal of Man-Machine Studies*, *39*(5), 775–792. https://doi.org/https://doi.org/10.1006/imms.1993.1083

Steyer, R. (2014). *Einführung in JavaFX*. Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-02836-7_1

Templet, R. M. (2021). *Game Physics: An Analysis of Physics Engines for First-time Physics Developers* [Zuletzt besucht am 16.06.2023]. https://scholarworks.calstate.edu/downloads/z890s004h

Wages, R., Grünvogel, S. M., & Grützmacher, B. How Realistic is Realism? Considerations on the Aesthetics of Computer Games (M. Rauterberg, Hrsg.). In: *Entertainment Computing – ICEC 2004* (M. Rauterberg, Hrsg.). Hrsg. von Rauterberg, M. Springer Berlin Heidelberg, 2004, 216–225. ISBN: 978-3-540-28643-1. https://doi.org/10.1007/978-3-540-28643-1_28

Weber, M., & Immich, T. (2009). *Animationen im Interface Design – Mehr als nur „Eye Candy“* (H. Brau, S. Diefenbach, M. Hassenzahl, K. Kohler, F. Koller, M. Peissner, K. Petrovic, M. Thielsch, D. Ullrich & D. Zimmermann, Hrsg.). Fraunhofer Verlag. https://doi.org/10.1524/icom.2009.0038

Wilson, A. D., Izadi, S., Hilliges, O., Garcia-Mendoza, A., & Kirk, D. (2008). *Bringing Physics to the Surface*. Association for Computing Machinery. ISBN: 9781595939753. https://doi.org/10.1145/1449715.1449728

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe, dass keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden und dass alle Zitate ordnungsgemäß gekennzeichnet worden sind.

Ferner habe ich vom Merkblatt über die Verwendung der Masterarbeit Kenntnis genommen und räume der Universität der Bundeswehr München das einfache Nutzungsrecht an meiner Masterarbeit ein.

Neubiberg, den 25.06.2023

.....

Gerrit Grauwinkel