Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Test Fuzzing for Virtual Reality Applications

Alexander Bernhardt

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Prof. Dr. Michael Sedlmair
Beginn am:	5. April 2021
Beendet am:	5. Oktober 2021

Kurzfassung

Durch die steigende Leistung der Grafikprozessoren und der wachsenden Anzahl an Headsets und Controllern, steigt auch das Interesse an Virtual Reality (VR)-Anwendungen. VR findet heutzutage in den unterschiedlichsten Sektoren Anwendung und die Entwicklung entsprechender Software ist dafür Voraussetzung. Es gibt unterschiedliche Testmethoden innerhalb der Software-Entwicklung. Durch ihren hohen Automatisierungsgrad, nimmt Fuzzing heute eine zentrale Rolle ein. Der Aufwand der Testerzeugung ist gering, die Effektivität hoch und die Methode kann auch bei Systemen angewandt werden, bei denen kein Quellcode zur Verfügung steht. Trotz dieser Vorteile wurde Fuzzing bisher nicht bei der Entwicklung von VR-Anwendungen angewandt.

In der vorliegenden Arbeit soll nun geprüft werden, ob die Testmethode auch geeignet ist, Schwachstellen in VR-Software zu identifizieren. Es wurde hierzu ein eigener Protoyp entwickelt: in drei selbst erstellten, unterschiedlich komplexen Szenarien in Unity sollte ein synthetisch, mehrfach eingebauter Fehler gefunden werden. Das Ganze wurde im Anschluss mit einem Real Data Szenario wiederholt. Die Ergebnisse zeigen, dass es in allen drei Szenarien tatsächlich eine hohe Rate an Fehlerentdeckung gab. Es traten jedoch auch False Negatives auf, bei denen ein Fehler nicht als solcher erkannt wurde. Im Real Data Szenario wurde kein Fehler gefunden, jedoch gab es False Positives, das heißt es wurden fälschlicherweise Fehler gemeldet.

Der Prototyp ist ein vielversprechender Ansatz, der jedoch durch seine einfach gewählten Methoden an einigen Punkten an seine Grenzen kam.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Das Testen von Software im Rahmen der Entwicklung nimmt heute einen immer zentraleren Stellenwert ein. Grund ist, dass zum einen Softwareprojekte immer größer und komplexer werden und in nahezu allen Sektoren und Anwendungsfeldern vorkommen. Zum anderen bedeutet das Aufkommen von mobilen Computern, IoT-Geräten und dergleichen, dass sich die Umgebungen mehr denn je unterscheiden. Ein einziges Programm soll meist auf all diesen Plattformen ausführbar sein. Dies bedeutet, dass mehr Tests notwendig sind.

Die Herausforderung besteht darin, sichere und zuverlässige Software zu entwickeln. Die nichtfunktionalen Anforderungen (z.B. Performance oder Benutzbarkeit) sollten weiterhin erfüllt werden. Auch hierfür ist das Testen der Software enorm wichtig. Es gibt unterschiedliche Methoden, wie Software-Tests durchgeführt werden. Eine Form des automatisierten Testens ist *Fuzzing*. Im Jahre 1988 hat Professor Barton Miller erstmals das Thema Fuzzing aufleben lassen, als er an der Universität von Wisconsin-Madison in einer Programmierübung seine Studenten die ersten Fuzzer entwickeln lies [ZGB+21]. Mittlerweile ist das Verfahren gängig und erzielt gute Ergebnisse. *ClusterFuzz* ist ein Beispiel von Google, welches seit Februar 2021 circa 29.000 Fehler in Google-Produkte (z.B. Google Chrome) und über 26.000 Fehler in über 400 Open-Source Projekte gefunden hat [Goo21]. Auch weitere bekannte Unternehmen, z.B. Adobe [Ado21], Cisco [Cis21] oder Microsoft [Mic21] setzen Fuzzing für die zuverlässige und sichere Ausführung ihrer Produkte ein.

Einen wichtigen Platz in der Softwareentwicklung nehmen heutzutage Augmented Reality (AR) und Virtual Reality (VR) ein. Durch die immense Herstellung von Headsets und Controller-Hardware, aber auch Grafikprozessoren mit immer mehr Leistung, wird VR immer interessanter für Spezialanwendungen, aber auch für den Massenmarkt. Dies resultiert in einer steigenden Anzahl von VR-Software in den verschiedensten Bereichen. So kommt bespielsweise VR bei Brandschutzübungen [AG21] im Unternehmen, bei Flugsimulationen [Luf21] oder bei einer virtuellen Ausstellung eines Museums [Fra21] zum Einsatz. Aber auch bei der Schweißerausbildung [Sol21] oder bei der Traumatherapie [Med21] kann heutzutage auf die VR-Technologie zurückgegriffen werden.

In der vorliegenden Arbeit soll nun der Frage nachgegangen werden, ob die Testmethode Fuzzing auch auf VR-Anwendungen übertragbar ist. Und falls ja - wie gut? Damit werden die Felder Software-Testen und VR-Anwendungen verbunden.

1.2 Gliederung der Arbeit

Die Masterarbeit ist in sieben Abschnitte unterteilt. In Kapitel 2 wird dem Leser ein Grundverständnis zu den beiden Themen VR und der Testmethode Fuzzing vermittelt. Dabei wird anhand einer grundlegenden Fuzzing-Architektur und deren Komponenten die Funktionsweise eines Fuzzing-Tools bzw Fuzzer erläutert. Des Weiteren werden grundlegende Vorgehensweisen bei der Entwicklung von VR-Anwendungen vorgestellt und mögliche Herausforderungen diskutiert.

Kapitel 3 stellt einen Überblick über verwandte Arbeiten zu den Themen Testen und Debugging von VR-Systemen dar.

In Kapitel 4 werden die Inhalte, Ergebnisse und Erkenntnisse aus den durchgeführten Interviews präsentiert. Mit den Interviews wurde das Ziel verfolgt, eine Übersicht über typische Problemen und Fehler bei der Nutzung und Entwicklung von VR-Anwendungen zu erhalten. Daraus wurde ein bestimmte Fehler ausgewählt und in die eigene Anwendung eingebaut.

Kapitel 5 beinhaltet die grundlegende Idee, die Architektur und die Umsetzung des entwickelten Prototyps. Zuvor getroffene Annahmen werden in diesem Kapitel ebenfalls aufgeführt. Abschließend wird die Integration und Nutzung des Prototyps in andere VR-Anwendungen erklärt.

In Kapitel 6 wird der in Kapitel 5 entwickelte Ansatz evaluiert. Dazu wurden vier Szenarien erstellt, anhand deren der Prototyp getestet wurde. Die gewonnenen Ergebnisse werden im Anschluss dargestellt und diskutiert.

Kapitel 7 schließt die Arbeit mit einer Zusammenfassung ab und gibt einen Ausblick für zukünftige Arbeiten.

2 Hintergrund

In diesem Kapitel werden zentrale Aspekte zu den beiden Themengebieten VR und Fuzzing erläutert, was für ein besseres Verständnis der Arbeit beitragen soll. Es wird darauf eingegangen, was man unter den beiden Begriffen VR und Fuzzing versteht. Des weiteren wird dem Leser die grundlegende Idee, die Funktionsweise und mögliche Herausforderungen bei der Entwicklung und Nutzung von VR bzw. Fuzzing vermittelt.

2.1 Virtual Reality

Craig et al. definieren Virtual Reality (VR) als „ein Medium, das aus interaktiven Computersimulationen besteht, welche die Position und die Handlungen des Teilnehmers wahrnehmen und eine synthetische Rückmeldung an einen oder mehrere Sinne geben, das Gefühl vermittelt, in die Simulation einzutauchen oder darin präsent zu sein “ [CSW09].

Dies bedeutet, dass die virtuelle Realität ein Mittel darstellt, welches es Personen ermöglicht, sich physisch in einer simulierten Umgebung zu bewegen und mit dieser zu interagieren, die sich jedoch von ihrer physischen Realität unterscheidet [CSW09].

Nach Dörner et al. [DBJ+19] setzt sich ein VR-System aus drei Teilsystemen zusammen. Diese sind Eingabegeräte, Ausgabegeräte und Weltsimulation. In Abbildung 2.1 ist ein Überblick über die Teilsysteme und deren Zusammenspiel dargestellt. Die im Folgenden beschriebenen Inhalte basieren alle auf dem Buch von Dörner et al. [DBJ+19].

Mithilfe von Sensoren (z.B. Mikrofon oder Kamera) erfassen Eingabegeräte Daten über Nutzeraktionen, Objekte und die virtuelle Umgebung. Dazu gehört beispielsweise die Erkennung der Position und Blickrichtung des Nutzers, so dass die richtige Perspektive für ihn in der virtuellen Welt berechnen werden kann. Diese gewonnenen Daten werden zusammengefasst und an die Weltsimulation weitergeleitet. Oftmals werden mehrere Eingabegeräte gleichzeitig genutzt, um z.B. die Erkennung der Position und Blickrichtung des Nutzers zu verbessern. Die Informationen aus den verschiedenen Eingabegeräten müssen korrekt zusammengefasst werden. Dies wird *Sensorfusion* genannt. Eingabegeräte können anhand verschiedener Kriterien klassifiziert werden. So lassen sie sich anhand der Genauigkeit (fein und grob) und Reichweite (z.B. eingeschränkter Bewegungsbereich) bzw. als diskret (z.B. drücken der Maustaste) oder kontinuierlich (kontinuierliche Übermittlung von Positionsdaten- Tracking) klassifizieren.

Ausgabegeräte dienen der Darstellung der virtuellen Welt für den Nutzer, in diese er „eintaucht“. Dabei Dabei werden die Sinne des Nutzers hauptsächlich über visuelle, akustische und haptische Sinne angesprochen. Die Umwandlung des Modells von der virtuellen Welt im Computer zu den

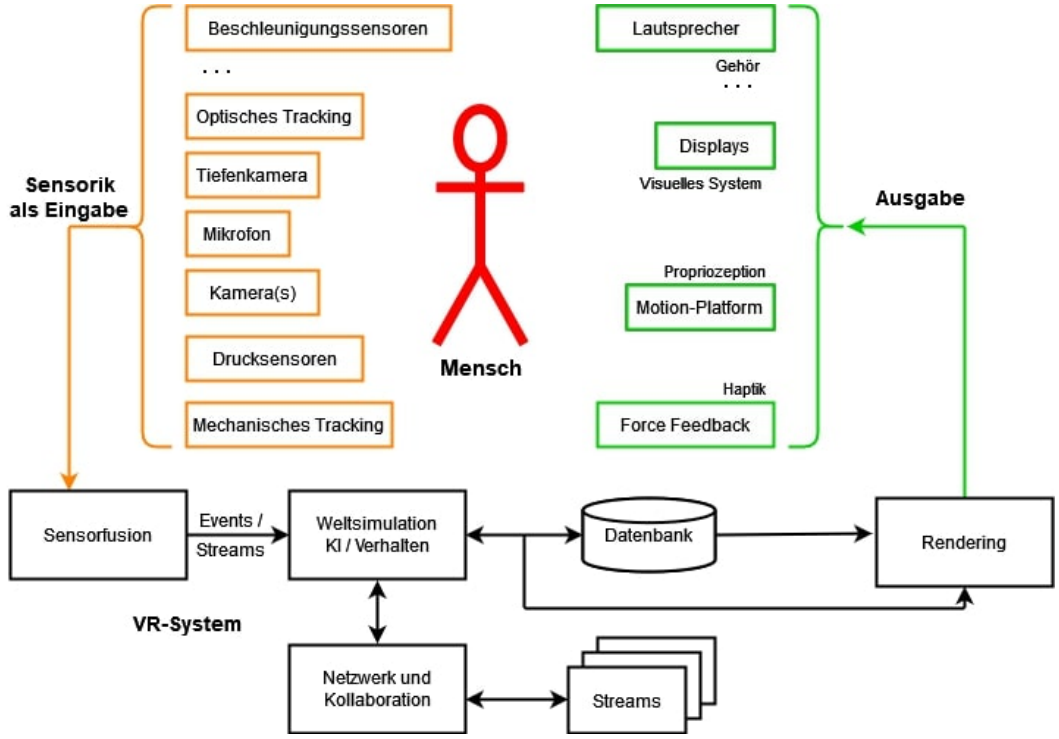


Abbildung 2.1: Sensoren für die Eingabegeräte (orange), Ausgabegeräte für die Adressierung der einzelnen Wahrnehmungskanäle des Menschen (grün), restlichen Teilsysteme eines VR-Systems inklusive der Weltsimulation (blau) [DBJ+19]

Sinnesreizen für den Nutzer wird *Rendering* genannt. Ein Beispiel für ein visuelles Ausgabegerät ist ein Head-Mounted-Display (HMD). Dies ist ein Display, welches der Nutzer direkt vor seinen Augen trägt, in Form einer Brille.

Das dritte Teilsystem einer VR-Anwendung ist die Weltsimulation. Diese Komponente verfügt über ein passendes Modell der virtuellen Umgebung. Dies können z.B. Simulationsmodelle oder Modelle, die auf Künstlicher Intelligenz (KI) basieren, sein. Die Modelle legen außerdem fest, wie sich die virtuelle Welt verhält. Das Verhalten kann mit den erhaltenen Informationen der Eingabegeräte beeinflusst werden.

Ein wichtiger Aspekt bei VR-Systemen ist der Begriff *Echtzeit*. Das bedeutet, dass das VR-System Ergebnisse in einer möglichst kurzen Zeitspanne berechnet und an die entsprechende Komponente liefert. Das Ziel besteht darin, die Zeitspanne zu minimieren. Dies kann in verschiedenen Teilen der in Abbildung 2.1 beschriebenen Architektur stattfinden. Man unterscheidet dazu die Trackinglatenz, Transportlatenz, Simulationslatenz, Generierungslatenz und die Darstellungslatenz. Die Reihenfolge dieser Latenzen entspricht den einzelnen Arbeitsschritten der in Abbildung 2.1 abgebildeten Architektur. Unterstützend kann auf vorberechnete Simulationsdaten zurückgegriffen werden, anstelle der Berechnung in Echtzeit.

Die Entwicklung einer VR-Anwendung ist sehr komplex und fragmentiert [NS18]. Entwickler müssen eine Vielzahl an unterschiedlichen Tools lernen. Dazu gehören Programmierbibliotheken, Programmierwerkzeuge (Software Development Kit, SDK) oder Programmierschnittstellen (Application Programming Interface, API). Darüber hinaus ist es ebenfalls notwendig, dass der

Entwickler Know-how über weitere, nicht VR- bzw. AR-spezifische Tools besitzt. Beispiele hierfür sind Bildbearbeitungsprogramme wie Photoshop für die Texturgenerierung oder Werzeuge für die Animierung wie Blender. Eine zentrale Rolle spielen mittlerweile Entwicklungsumgebungen, die viele der oben genannten Funktionen in einem einzigen Tool integrieren. Beispiele für solche Entwicklungsumgebungen, die häufig bei der Spieleentwicklung und auch für VR-/AR-Anwendungen zum Einsatz kommen, sind Unity und Unreal. In diesem Zusammenhang werden die Entwicklungsumgebungen auch Game Engines genannt.

2.2 Fuzzing

Oehlert definiert Fuzzing als „eine automatisierte Testmethode, die zahlreiche Grenzfälle mit ungültigen Daten (aus Dateien, Netzwerkprotokollen, API-Aufrufen und anderen Zielen) als Anwendungseingabe abdeckt, um sicherzustellen, dass keine Schwachstellen vorhanden sind“ [Oeh05].

Die Idee beim Fuzzing besteht darin, das zu testende Zielprogramm mit einer Vielzahl von Testfällen zu füttern, fast schon zu überhäufen, und das entsprechende Programmverhalten zu beobachten, um festzustellen, ob es Fehler oder Schwachstellen in der Software aufweist. Fuzzing kann bei jeder Art von Systemen mit Eingabeschnittstellen eingesetzt werden, z.B. bei Netzwerkprotokollen [BCF+06], Dateiformaten [KCL11] und vielen weiteren.

Der allgemeine Arbeitsablauf von Fuzzing wird mithilfe einer gewöhnlichen Fuzzing-Architektur in Abbildung 2.2 erläutert und basiert auf der Arbeit von [LPJ+18]:

- *Zielprogramm*: Das zu testende Programm, welches in Form von Binärcode oder dem Quellcode vorliegen kann. Da in den überwiegenden Fällen der Zugang zum Quellcode nicht möglich ist, zielen die meisten Fuzzer auf Binärcode ab.
- *Monitoring*: Eine Monitoring-Komponente liefert Laufzeitinformationen (z.B. Code-Abdeckung oder Taint-Datenfluss) des Zielprogramms mithilfe von Code-Instrumentierung oder Taint-Analyse. Normalerweise ist diese Komponente in einem White-Box oder Gray-Box Fuzzer integriert. In einem Black-Box Fuzzer ist dies nicht notwendig.
- *Testfallgenerator*: Diese Komponente sorgt für die Generierung von Testfällen. Dazu existieren zwei Methoden: *mutation-based*- und *grammar-based*-Methoden. Die erstgenannte Methode generiert Eingabedaten aus validen Seeds. Dazu werden die validen Seeds zufällig verändert oder nach einer bestimmten Mutations-Strategie. Die *grammar-based*-Methode hingegen benötigt keine Seeds. Hier werden Eingabedaten mithilfe der Spezifikation generiert.
- *Fehlerdetektor*: Der Fehlerdetektor unterstützt den Nutzer bei der Auffindung von Fehlern. Bei einem Absturz oder einer Fehlermeldung des Zielprogramms sammelt der Fehlerdetektor dazugehörige Informationen (z.B. stack traces [WCGB13]), anhand derer bewertet werden kann, ob ein Fehler (Bug) existiert. Anstelle des Fehlerdetektors kann auch manuell ein Debugger eingesetzt werden, um Informationen von Ausnahmen (Exceptions) aufzuzeichnen und zu speichern [BB09; BBGM12; WZLY13].

- *Fehlerfilter*: Diese Komponente filtert die für den Nutzer relevanten Fehler heraus. Das größte Interesse besteht oftmals an sicherheitsrelevanten Fehlern und an der Feststellung, ob es sich tatsächlich um einen Fehler handelt. Dieser zeitaufwendige Prozess wird größtenteils manuell durchgeführt [BBGM12], jedoch existieren auch automatisierte Ansätze zu der Problematik [CGZ+13].

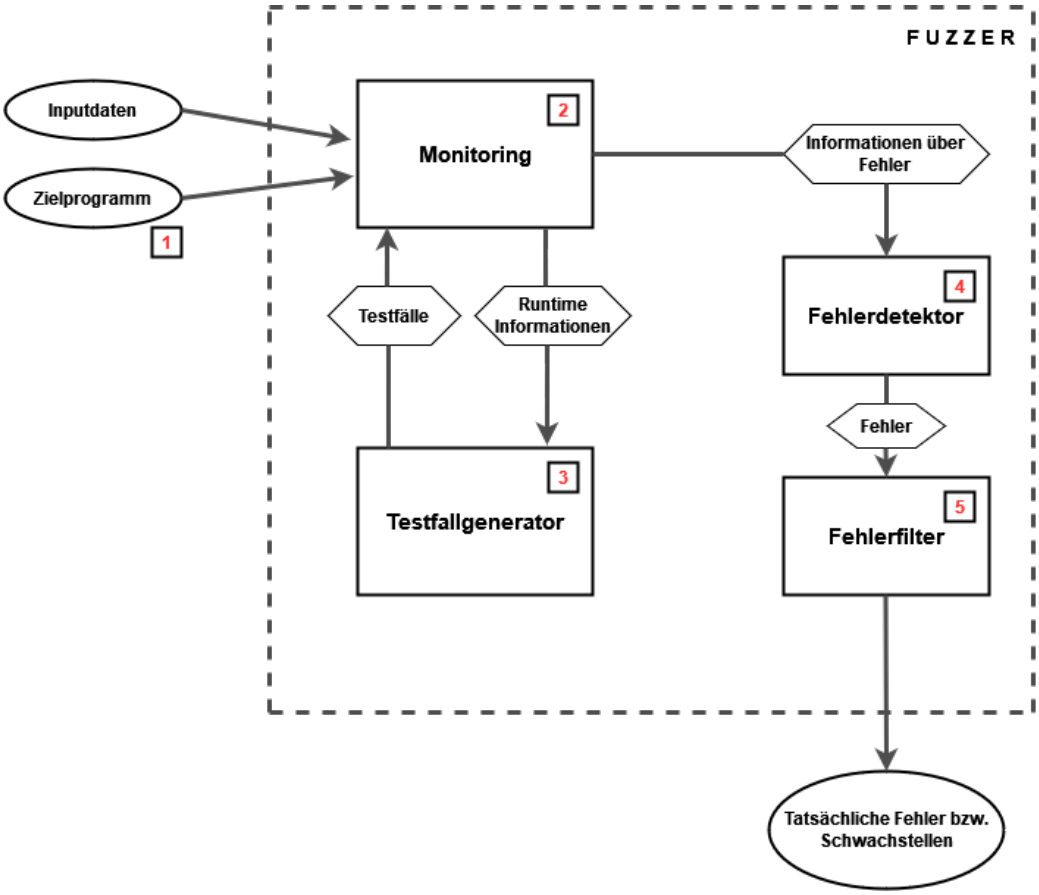


Abbildung 2.2: Allgemeiner Ablauf von Fuzzing [LPJ+18]

Fuzzer können auf unterschiedliche Art und Weise klassifiziert werden. Li et. al [LZZ18] geben dazu einen Überblick:

Zum einen können Fuzzer in *generation-based* und *mutation-based* eingeteilt werden [Van05]. *Generation-based* Fuzzer benötigen das Wissen über die Eingabe des Zielprogramms. Mithilfe dieses Wissens, das oftmals in Konfigurationsdateien abgelegt wird, werden Testfälle erstellt. Somit können diese Testfälle die Validierung des Zielprogramms leichter durchlaufen. Dies erhöht die Chance den tieferen Code des Zielprogramms zu erreichen. Ist dieses Wissen nicht vorhanden, muss zuerst das Format der Eingabe analysiert werden. Dies stellt jedoch eine große Herausforderung dar. *Mutation-based* Fuzzer benötigen eine geringe Menge an validen Eingaben des Zielprogramms. Diese Eingaben werden dann mutiert und somit weitere Testfälle generiert.

Zum anderen können Fuzzer auf Basis des vorhandenen Informationsgehalts des Zielprogramms zur Laufzeit klassifiziert werden (z.B. Code-Abdeckung, Speichernutzung des Programms oder CPU-Auslastung) [Jää16].

Hier werden *Black-Box*, *Grey-Box* und *White-Box* Fuzzer unterschieden. *Black-Box* Fuzzer besitzen kein Wissen über den Quellcode des Zielprogramms. Demnach werden die Testfälle zufällig generiert. *Grey-Box* Fuzzer besitzen ebenso wie *Black-Box* Fuzzer kein Wissen über interne Strukturen des Zielprogramms. Jedoch sind sie in der Lage durch Programmanalysen des Zielprogramms diese internen Informationen zu erhalten. Im Gegensatz dazu ist der Quellcode bei *White-Box* Fuzzern vorhanden, so dass durch Programmanalysen weitaus mehr Informationen des Zielprogramms gewonnen werden können.

Fuzzer lassen sich auch anhand der Explorationsstrategie des Zielprogramms einteilen,wie beispielsweise die *directed* Fuzzer und die *coverage-based* Fuzzer. Ein *directed* Fuzzer verfolgt das Ziel Testfälle zu generieren, die einen bestimmten Pfad oder einen bestimmten Bereich im Quellcode abdecken. *Coverage-based* Fuzzer hingegen haben das Ziel so viel Quellcode wie möglich zu testen.

Fuzzer können auch als *dumb* Fuzzer und *smart* Fuzzer klassifiziert werden, abhängig davon, ob zwischen der Überwachung des Programmausführungszustands und der Testfallgenerierung ein Feedback gegeben wird oder nicht. Ein *smart* Fuzzer generiert Testfälle mithilfe der gesammelten Informationen der Programmanalyse, aber auch durch die Informationen, wie Testfälle zuvor das Zielprogrammverhalten beeinflusst haben. *Dump* Fuzzer steht diese Information nicht zur Verfügung. Demnach haben sie eine höhere Testgeschwindigkeit, wohingegen *smart* Fuzzer bessere Testfälle generieren und damit eine höhere Effizienz erreichen.

Bei der Entwicklung eines Fuzzing-Tools sollte sich zuvor über bestimmte Fragestellungen Gedanken gemacht werden. Liang et al [LPJ+18] stellen hierdurch auch dar, welche Herausforderungen bei der Entwicklung auftreten können:

1. Wie werden Seeds und andere Testfälle ausgewählt bzw. generiert?
2. Wie können Eingaben anhand der Spezifikation des Zielprogramms validiert werden?
3. Wie wird mit Testfällen umgegangen, die einen Absturz verursacht haben?
4. Wie können gewonnene Laufzeitinformationen verwendet werden?
5. Wie kann die Skalierbarkeit des Fuzzers verbessert werden?

3 Relevante Arbeiten

Bei der Entwicklung von AR- und VR-Anwendungen können nicht nur die üblichen Compilerfehler auftreten, sondern auch inhaltliche, konzeptionelle und operative Herausforderungen entstehen. Dies macht das Testen und Debuggen für Entwickler anspruchsvoll. Im Folgenden werden Ideen und Ansätze für das Testen und Debuggen, sowohl von VR- als auch AR-Anwendungen, vorgestellt. Im praktischen Teil der Arbeit findet jedoch ausschließlich die Testmethodik Fuzzing Anwendung.

Winterbottom et al. [WBG06] stellen in ihrer Forschungsarbeit ein System vor, das mithilfe von Visualisierungen den Entwurf und das Debugging von Interaktionen in VR-Anwendungen unterstützt. Das Verwenden und die Kombination von Grundrissen des Raums, Zeitleisten und Sequenzdiagrammen bilden die Schlüsselaspekte für den Ablauf von Interaktionen im VR System. Die Programmierung von Interaktionen basiert auf dem *event-action* Paradigma. Dazu wurden Tripel von *trigger-condition-action* (Triggersets) verwendet. Der Entwickler kann Objekte und die Umgebung spezifizieren. Das System erstellt daraufhin einen 2D-Grundriss. Zeitliche Abfolgen von Aktionen können über die Zeitleisten eingegeben werden. Sobald der Entwickler eine Menge von Triggersets eingegeben hat, wird ein Sequenzdiagramm erzeugt, das die Interaktion beschreibt. Somit werden Änderungen im Code direkt auf das System übertragen und können mithilfe der Visualisierungen überprüft und nachverfolgt werden.

Die Arbeit von Hoppenstedt et al. [PGD+15] beschäftigt sich mit der Visualisierung von Daten eines Quadrocoptersystems in Mixed Reality. Dazu wurde eine HoloLens und das Robot Operating System (ROS) [1] verwendet und miteinander verbunden. Die Nutzung der HoloLens ermöglicht es ein virtuelles Modell der realen Welt zu erstellen und damit Interaktionen von Hologrammen und Objekten aus der realen Welt zu gewährleisten. ROS ist für die Steuerung eines Roboters und für die Visualisierung der aufgenommen Daten durch die Hololens verantwortlich. Durch das Verwenden von *source origin tracing*, welches z.B. in der Arbeit von Breckel et al. [BT16] verwendet wird, ist es durch die Visualisierung möglich direkt im Quellcode Veränderungen vorzunehmen.

Das Analysieren von verteilten VR-Anwendungen ist sehr aufwändig und komplex. Einen Ansatz für das Debuggen von verteilten VR-Anwendungen stellen Guimarães et at. [PGD+15] vor. Da mehrere Prozesse auf unterschiedlichen Rechnern ausgeführt werden, besteht die Notwendigkeit, die Berechnungsergebnisse zu synchronisieren. Das Auftreten eines Fehlers hängt demnach von den Zuständen der einzelnen Prozesse ab. Abhilfe verschaffen sogenannte *functional behavior models*, welche z.B. *shared variables* oder *synchronization barriers* verwenden. Der Ansatz von Guimarães et at. besteht darin, Nachrichten während der Kommunikation der verteilten Systeme zu extrahieren und diese zu analysieren. Dazu wurde das Tool GTracer entwickelt, das abschließend einen Fehlerbericht liefert und die Nachrichten visuell darstellt.

[1] https://www.ros.org/

Ein nutzerfokussiertes Framework für das Testen von AR-Anwendungen stellen Lehman et al. [LLT20] in ihrem veröffentlichten Artikel (2020) auf. Das entwickelte Framework ARCHIE sammelt Feedback des Benutzers und Zustandsdaten des Systems zur Laufzeit. Diese Informationen können Aufschluss darüber geben, welche Fehler bei schlechtem Benutzer-Feedback verantwortlich sind. Das Framework ermöglicht außerdem das Testen mehrerer Versionen einer Anwendung („profiles “), so dass diese miteinander verglichen werden können.

Einen Ansatz zum automatisierten Testen von VR-Anwendungen liefern Bierbaum et al. [GFR+20]. Dazu werden die Unit-Tests in Python generiert und durch Verwendung des Plugins Youkai die Verbindung zur Unity-Anwendung bzw. sogar zum UnityEditor hergestellt. Bierbaum et al. verglichen ihren Ansatz mit AltUnitTester [Con], einem Testing-Tool aus dem Unity Asset Store. Die Vorteile sind die einfache Erweiterung des Frameworks und die Möglichkeit, angebotene Methoden zur Interaktion mit Unity zu verwenden. Dies führt zu verbesserten Unit-Tests.

Willans und Harrison [WH00] präsentieren einen „Plug and Play “-Ansatz zum Testen von Interaktionstechniken in einer VR-Anwendung. Dazu entwickelten sie das Toolset *Mariegold*, welches aus zwei Tools besteht: Dem *interaction technique builder* (ITB) und dem *prototype builder* (PB). Das erstgenannte Tool unterstützt die visuelle Spezifikation einer Interaktionstechnik. Der ITB erzeugt einen Stub [2], der die Interaktionstechnik beschreibt. Das zweite Tool bietet dem Benutzer eine visuelle Methode an, welche die generierten Stubs in die virtuelle Welt integriert. Der PB generiert automatisch den Code für die virtuelle Umgebung.

Bierbaum et al. [BHC03] stellen in ihrer Arbeit einen Ansatz für das automatisierte Testen von Interfaces bei VR-Anwendungen vor. Zuerst erklären sie, warum übliche Unit-Tests aus der Softwareentwicklung für das Testen von Interfaces in VR-Anwendungen nicht geeignet sind. Danach stellen sie ihre Idee vor. Der High-Level-Ansatz besteht aus einer Komponente, die eine Liste von generierten Testfällen verwaltet. Jeder einzelne Testfall übernimmt die Überwachung des kompletten Systemzustands der VR-Anwendung selbst, während die VR-Anwendung durch die zuvor erstellten Eingaben gesteuert wird. Erreicht die Anwendung nun einen Zustand, der einen Testfall erfordert, so aktiviert sich der Testfall und überprüft die Anwendung auf den korrekten Zustand. Ist der Zustand der Anwendung nicht korrekt, so wird ein Fehler signalisiert. Für das Entwickeln des Systems haben Bierbaum et al. das plattformübergreifende Framework VRJuggler [3] genutzt.

[2] https://de.wikipedia.org/wiki/Stub_(Programmierung)
[3] http://www.vrjuggler.org/

4 Interviews

In diesem Kapitel werden die Inhalte, Ergebnisse und Erkenntnisse aus den Interviews präsentiert. Aufgrund der Corona-Regelungen wurden die meisten Interviews online via Webex und Skype durchgeführt. Die sieben teilnehmenden Personen vom Visualisierungsinstitut der Universität Stuttgart (VISUS), davon sechs wissenschaftliche Mitarbeiter und ein technischer Mitarbeiter, sollten über ihre eigenen Erfahrungen bzw. ihr Wissen zu auftretenden Problemen und typischen Fehlern bei der Entwicklung und Nutzung von VR-Anwendungen berichten. Die Antworten der Befragten werden im Abschnitt 4.2 aufgeführt. Abschnitt 4.3 stellt im Anschluss die gewonnenen Ergebnisse aus den Interviews dar.

4.1 Zielsetzung

Ziel der Interviews war es, Informationen zu typischen Problemen und Fehlern bei der Nutzung und Entwicklung von VR-Anwendungen zu gewinnen, daraus Muster abzuleiten und aus den aufgetretenen Fehler- und Problemarten Cluster zu bilden. Die Idee war es, aus diesen Erkenntnissen bestimmte Fehler auszuwählen und synthetisch in eigene Anwendungen einzubauen. Daraufhin könnte ein fuzzing-basierendes Tool implementiert werden, das im ersten Schritt die eingebauten Fehler erkennt und später dieselbe Art von Fehlern in neuen Anwendungen erkennt.

4.2 Inhalte der Interviews

Im ersten Teil dieses Abschnitts werden die Antworten der Befragten über auftretende Probleme bei der Nutzung von VR-Anwendungen dargestellt, sowohl aus Sicht eines Endanwenders, als auch aus Sicht eines Entwicklers.

Der zweite Teil befasst sich mit Fehlern und Problemen, die während der Entwicklung von VR-Anwendungen bei den Befragten schon einmal selbst aufgetreten sind oder sie auf eine andere Art und Weise erfahren haben. Viele Probleme, die bei der Nutzung von VR-Anwendungen auftreten, können ebenso bei der Entwicklung bzw. beim Testen der Anwendungen auftreten. Demzufolge werden diese Punkte im zweiten Abschnitt nicht mehr erneut erwähnt.

Die Interviewteilnehmer schilderten die unterschiedlichsten Probleme bei der Nutzung von VR-Anwendungen. Zunächst machte ein Teilnehmer darauf aufmerksam, dass es für Menschen mit fehlenden Gliedmaßen, wie Armen oder Beinen, sehr schwer oder gar unmöglich ist, eine VR-Anwendung zu nutzen. Für Interaktionen in der virtuellen Welt, wie bspw. das Teleportieren, werden mit der Hand gesteuerte Controller verwendet.

Auch für Mensch mit eingeschränktem Sehvermögen, wie zum Beispiel Farbenblindheit oder Rot-Grün-Schwäche, kann es zur Herausforderung werden eine Anwendung zu nutzen. Diese Personen können dann evtl. verwendete Farben nicht unterscheiden, was für die Nutzung jedoch notwendig wäre. Benutzer, die eine Sehhilfe in Form einer Brille tragen, können nur eine begrenzte Anzahl an VR-Brillen verwenden, die dieses Feature auch unterstützen.

Ein weiterer genannter Punkt in den Interviews betrifft die Störung des vestibulookulären Reflexes [1], der Motion Sickness mit Schwindel oder Kopfschmerzen hervorrufen kann. Dieses Phänomen tritt häufig in VR-Spielen auf, bei denen der Nutzer still in einem Stuhl sitzt, während das Auge jedoch gleichzeitig wilde Bewegungen wahrnimmt. In einem Beispiel führte eine Anwendung mit einer zu geringen fps (frames per seconds) Anzahl dazu, dass dem Benutzer bei Rotationen schlecht wurde.

Nahezu alle Befragten sind schon einmal auf das Problem gestoßen, dass der virtuelle Raum, in dem sich bewegt und interagiert wird, größer ist als der physische Raum. Hier ist die Navigation durch die virtuelle Umgebung eine große Herausforderung und zu Beginn ein wenig befremdlich. Zudem kann es zu Komplikationen kommen, wenn die Kabellänge nicht ausreicht um die Welt vollständig zu erkunden.

Ein Interviewter berichtete von einer VR-Brille, die nur fünf Minuten tragbar war, da sie dann zu heiß wurde und eine Gefahr für die Augen darstellte. Das (Nischen-) Produkt war nur für bestimmte Unity-Versionen lauffähig und hat dadurch, nach Angabe der befragten Person, einen minderwertigen Eindruck hinterlassen.

Mit alltäglichen Komplikationen, wie zum Beispiel dem Ausfall der Controller aufgrund von leeren Batterien, war jeder Befragte schon einmal konfrontiert.

Darüber hinaus wurde von mehreren visuellen Anwendungsproblemen berichtet. Ein Interviewter erzählte beispielsweise, dass sein Bild anfing zu flackern, als er in einer Anwendung eine bestimmte Position angeschaut hatte. Zudem hat sich die Ausrichtung des Koordinatensystems um 180 Grad gedreht und der Turm stand in der virtuellen Welt plötzlich auf dem Kopf. Diese Erfahrung stammte aus einem AR (Augmented Reality) Projekt. Zwei Personen beschrieben eine Änderung der Sichthöhe in ihrem Projekt, sowohl beim Spawnen, als auch beim Teleportieren durch die Welt. So waren zum Beispiel die Augen auf der gewollten, eingestellten Höhe von $Y = 1.80$, jedoch war die Höhe nach dem Teleportieren plötzlich bei $Y = 0.4$. Dies hatte den Anschein, man krieche auf dem Boden, was so nicht gewollt war.

Eine weitere befragte Person befasste sich hauptsächlich mit haptischem Feedback in VR-Anwendungen. Manchmal kam es in Situationen zu haptischen Feedbacks, in denen es nicht gewollt war.

Bei der Entwicklung von VR Anwendungen traten bei den Befragten auch diverse Probleme auch.

Bei drei Personen traten zum Beispiel Komplikationen in Verbindung mit dem Netzwerk auf. Bei einer Netzwerkkomponente wurde bei der Verwendung einer Library aus dem Unity-Store [2] kontinuierlich die Netzwerksession gestartet, auch während des Debug-Vorgangs, als es nicht erwünscht war. Bei

[1] https://de.wikipedia.org/wiki/Vestibulookulärer_Reflex
[2] https://assetstore.unity.com/

einem weiteren Szenario war ein Netzwerkport belegt, so dass eine Demonstration der Anwendung vorerst nicht möglich war. In einem dritten Fall brachte eine inkorrekte Stringübertragung zwischen einem Hololens-Tablett und einer HTC VIVE einen Fehler hervor.

Es wurden viele negative Erfahrungen zum Zusammenspiel von VR-Hardware und entsprechender Software berichtet. So kam es häufig bei der Ausführung von Anwendungen zu Fehlern, da genutzte Libraries oder Software Development Kits (SDK) nicht mit jeder Hardware kompatibel waren. Zudem waren auch nicht alle SDKs und VR-Hardwares kompatibel mit allen existierenden Unity-Versionen. Sofern ein Projekt eine andere Version eines SDKs oder anderer Software benötigte, blieb daher kaum eine andere Wahl, das Projekt selbstständig durchzuarbeiten, anzupassen oder Änderungen an der API vorzunehmen. Dies wurde als sehr mühsam und fehleranfällig empfunden.

Nach den Erfahrungen der Interviewten erschienen auch Fehler, nachdem Support-Updates von SDKs durchgeführt wurden. Demnach konnte zum Beispiel eine Szene in Unity nicht mehr komplett importiert werden. Auch Performance-Einbußen waren beispielweise ein Resultat eines Softwareupdates. Ein SDK für eine Netzwerkkomponente und ein Computervision Toolkit benötigten unterschiedliche .NET Versionen. Daraufhin konnte die angebotene Funktionalität nicht genutzt werden und musste selbst implementiert werden.

Zwei Personen benutzten neben Unity noch ein weiteres SDK für das Eye-Tracking einer VIVE Pro in ihrem Projekt, jedoch wurden dazu Adminrechte benötigt. Das ließ für Studenten an der Universität kein komfortables Arbeiten zu. Auch bei der Kombination aus Unity, SteamVR [3] und eines VR-Headsets trat ein Fehler beim Tracking auf. Dieser konnte nur temporär mit neuer Kalibrierung des Setups behoben werden. Bei der Verwendung von SteamVR und Unity verschwanden teilweise die Mapping-Profile der Controller, nachdem SteamVR neugestartet werden musste. Daraufhin musste sich der Benutzer bei einem SteamVR-Account erneut einloggen, um das Problem zu beheben. Von einem ähnlichen Szenario erzählte ein anderer Befragter. Dabei verschwand jedoch nicht das komplette Mapping-Profile, sondern es wurde nur die Belegung der Buttons ständig verändert. Nur durch einen Restart konnte er dieses Problem beheben. In einem Projekt wurde ZeroMQ [4] integriert und verwendet, jedoch ließ sich ein Kommunikationsthread nicht schließen. Resultat waren eine fehlende Kontrolle und die Ursache konnte nicht identifiziert werden. In einem weiteren Projekt entdeckte der Entwickler eine eingeschränkte Funktionalität beim Tracking. Sobald der Endanwender in den Himmel schaute, funktionierte das Tracking nicht mehr.

Objekte für VR-Anwendungen werden oftmals gesondert erstellt. Dazu eignen sich Programme wie Blender und Maya. Sofern mehrere Entwickler an einem Projekt arbeiten, benötigen alle dieselbe Version von Blender, damit die Meshes korrekt geladen werden können. Ein weiteres Problem stellte das Importieren von Meshes in Unity dar, wenn andere Entwickler ein Projekt auschecken und kein Maya benutzen. Danach werden die Meshes falsch oder gar nicht angezeigt. Auf Codeebene kann das falsche Iterieren über Gameobjects in Unity die Performance einschränken und die gezielten 90 Hz bis 120 Hz für eine flüssige Anwendung werden dann nicht erreicht. Die Folge sind Verzögerungen (Lags) für den Benutzer.

[3] https://store.steampowered.com/app/250820/SteamVR
[4] https://zeromq.org/

In einem Szenario trat ein Fehler bei der Interaktion von Gameobjects in Unity auf. Es wurde an ein Gameobject ein Skript mit einem Collider als RequireComponent [5] hinzugefügt. Diese wurde vom Entwickler vergessen, sodass Unity diese Komponente nachträglich hinzufügte. Dabei wurde ein Collider eines komplexen Objekts aus einfachen Colliders, wie Boxcollider oder Spherecollider, nachgebaut. Dies kann jedoch dazu führen, dass der Nutzer plötzlich ungewollt innerhalb eines Objekts steht und dabei hinter das Objekt schauen kann.

4.3 Ergebnisse

Meine Stichprobe an interviewten Personen war nicht im statistischen Sinne repräsentativ gewählt, so dass keine allgemeingültigen Rückschlüsse gezogen werden können. Dennoch liefern die Antworten einen guten ersten Eindruck zu möglichen Fehlern, was unter anderem damit zusammenhängt, dass die Befragten ganz unterschiedliche Erfahrungshintergründe einbringen konnten.

Im nächsten Schritt wurden die Antworten zu auftretenden Fehlern und Problemen bei der Nutzung und Entwicklung von VR-Anwendungen in folgende Cluster eingeteilt: Probleme bezüglich der Software, Probleme bezüglich der Hardware, Probleme bei körperlichen Einschränkungen und Performance-Probleme.

Durch die Interviews wurde zunächst deutlich, dass die Software und Hardware sowie deren Zusammenspiel eine enorm große Herausforderung bei VR-Anwendungen darstellt. Alle Teilnehmer hatten schon einmal die Erfahrung gemacht, dass genutzte Software mit ausgewählter Hardware Inkompatibilitäten hervorbrachte. Dies kann von Beginn an der Fall sein. Aber auch Updates der Software, zum Beispiel von SDKs oder Libraries, oder neuen Versionen der Hardware können Inkompatibilitäten hervorrufen und damit Projekte nutzlos machen, so dass die Funktionalitäten durch manuelle Änderungen wieder hergestellt werden müssen.

Die Interviews zeigten zudem auch, dass Anwendungen für Menschen mit Handicap noch nicht ausgereift sind. Sowohl für Menschen mit körperlicher, als auch für Menschen mit kognitiver Einschränkung, zeigen VR-Systeme noch große Unzulänglichkeiten.

Von enormer Wichtigkeit ist zuletzt die Performance der VR-Systeme. Neben den grundsätzlich notwendigen hohen Rechnerleistungen sind gerade bei hochqualitativen Trackingsystemen geringe Latenzen und eine Aktualisierungsfrequenz von mindestens 90 Hz notwendig, um eine flüssige Ausführung der Anwendungen zu gewährleisten [DBJ+19].

Im weiteren Verlauf der Arbeit wurde ein Problem herausgegriffen: sogenannte *Wall-Glitches*. Diese Fehler treten auf, wenn die HitBox bzw. der Collider von Objekten in der virtuellen Welt nicht korrekt oder nicht genau genug platziert werden. Diese Art von Fehler ist im Rahmen der Entwicklung einer Anwendung enorm wichtig. Warum wurde dieser Fehler gewählt? Zum einen kann der Fehler in den unterschiedlichsten Arten von Anwendungen auftreten, und ist somit universell relevant. Zum anderen ist der Fehler möglicherweise gut geeignet um mit Fuzzing erkannt zu werden.

[5] https://docs.unity3d.com/ScriptReference/RequireComponent

5 Eigener Ansatz

Dieses Kapitel befasst sich mit der grundlegenden Idee eine fuzzing-basierte Anwendung zu entwickeln, um den in Abschnitt 4.3 ausgewählten Fehler in VR-Anwendungen zu entdecken. Die simulierte Spielfigur, die die virtuelle Welt erkundet, wird im weiteren Verlauf der Arbeit als *Non-Player Character (NPC)* bezeichnet, da es den Benutzer mit einem VR-Headset simulieren soll.

5.1 Zielsetzung und Idee

Die Interviews boten eine gute Übersicht, mit welchen Fehlern Entwickler von VR-Systemen konfrontiert werden. So tauchen neben den herkömmlichen Compilerfehlern auch weitere VR-spezifische Fehler (z.B. die oben genannten Wall-Glitches) auf, die bei der Entwicklung entdeckt werden müssen. Da es sehr mühsam ist, diese Fehler durch manuelle Tests zu entdecken, besteht das Ziel dieser Arbeit darin, eine automatisierte Testanwendung auf Basis von Fuzzing zu entwickeln. Diese soll Wall-Glitches, wie im letzten Absatz in Abschnitt 4.3 beschrieben, erkennen und dem Entwickler entsprechendes Feedback dazu geben. Würde das Tool solche selbst eingebauten Fehler erkennen, wäre es interessant dies auch in anderen Anwendungen vorzunehmen.

Die grundlegende Idee der Umsetzung besteht darin, den NPC die Welt erkunden zu lassen und Screenshots bei auftretenden Kollisionen mit der Umwelt zu machen. Diese sollen dann analysiert werden, um zu erfahren, ob der Collider eines Gameobjects korrekt angebracht wurde.

5.2 Annahmen

Für die Entwicklung des Prototyps wurden einige Annahmen getroffen. Diese werden im Folgenden aufgezählt und erläutert.

Aufgrund der Vielfalt an verfügbarer Hardware und Software für die Entwicklung von VR-Anwendungen, wurde dieser Prototyp mit Unity entwickelt und konzentriert sich ebenso ausschließlich auf Anwendungen, die in Unity implementiert wurden. Unity ist eine sehr populäre und weitverbreitete Spiele-Engine , die für die Entwicklung von 3D-Spielen, aber auch VR-Anwendungen genutzt wird. Bei der Hardware lag der Fokus dieser Arbeit auf den Headsets, die kompatibel zur Plattform von Windows Mixed Reality (WMR) sind. Dazu wurde das XR Management Toolkit [1] in Unity verwendet.

[1] https://docs.unity3d.com/Packages/com.unity.xr.management@4.0/manual/index.html

Im Anfangsstadium der Entwicklung wurde der NPC durch ein Capsule 3D-Objekt mit einem Rigidbody dargestellt. Die Bewegung wurde mit *Vector3.MoveTowards* [2] umgesetzt. Eine Kollision wurde an der kompletten Kapsel erkannt.

Im weiteren Verlauf der Entwicklung zeigte sich, dass zum Beispiel eine Treppe ein unerwartetes Hindernis darstellte, welches nicht überwunden werden konnte. Es sollte jedoch keine Kollision mit der Treppenstufe detektiert werden. Nach langer Recherche war klar, dass dieser Ansatz zu viel Aufwand darstellen würde, da die Umsetzung sehr komplex ist.

Ein zusätzliches Problem kam im Zusammenhang mit der Physik-Engine auf. Soll der NPC eine Schräge hinauflaufen oder nach einer Kollision das Gleichgewicht halten, gibt es sehr viel zu beachten bzw. kann sehr viel schieflaufen.

Nach erfolgreicher Recherche kam ich zu dem Entschluss, dass stattdessen die Komponente Character Controller genutzt werden kann um diesen Problemen entgegenzuwirken. Eine realistische Bewegung des NPCs ist zum Beispiel in dieser Anwendung nicht nötig, wodurch die Physik vernachlässigt werden kann.

Der NPC wurde durch eine statische Kapsel simuliert, wodurch es beispielsweise nicht möglich ist, sich zu ducken. Die Kapsel ist 2 Einheiten (in Meter) groß und hat einen Capsulate Collider angefügt. Die Höhe der Kamera liegt bei 1.75 Einheiten (in Meter). Eine Blickänderung ist momentan nur um die Y-Achse möglich - dies entspricht einer Kopfdrehung nach links oder rechts.

Der entwickelte Prototyp ist in erster Linie auf Entwickler bzw. Tester ausgerichtet. Die Architektur wurde so gewählt, dass er in ein Unity-Projekt und somit in das Zielprogramm integriert werden kann. Ein gewöhnliches Fuzzing-Tool hingegen bekommt oftmals das Zielprogramm als Eingabe.

Da immer nur eine Instanz von Unity mit demselben Projekt aktiv sein kann, wurde eine Kommandozeilenanwendung nicht weiterverfolgt. Somit wurde die Nutzung des Prototyps für den Entwickler auf den UnityEditor beschränkt.

5.3 Architektur

Der Arbeitsablauf der Anwendung wird anhand der in Abbildung 5.1 gezeigten Architektur im Folgenden vorgestellt. Zuallererst findet die Datengenerierung der Inputs für die VR-Anwendung statt Abbildung 5.1(1). Anhand dieser Daten bewegt sich der NPC in die jeweilige Blickrichtung geradeaus, bis eine Kollision mit einem anderen Gameobject stattfindet oder die über den Parameter *stepsize* gewählte Schrittweite erreicht wurde Abbildung 5.1(2). Tritt eines dieser beiden Szenarien auf, nimmt der NPC durch das nächste Inputdatum eine entsprechende Richtungsänderung vor. Auf diese Art und Weise kann der NPC die virtuelle Welt erkunden. Während der NPC sich durch die Szene bewegt, wird in jedem Frame (Ausführung von *MonoBehaviour.Update()* [3]) eine frühzeitige Kollisionsabfrage durchgeführt Abbildung 5.1(3), um zu erfahren, ob der NPC sich einem anderen Gameobject nähert und mit diesem in absehbarer Zeit kollidiert. Sowohl bei der frühzeitigen Kollisionserkennung Abbildung 5.1(4.1), als auch bei der Kollision selbst Abbildung 5.1(4.2), wird eine Bildschirmaufnahme bzw. ein Screenshot von diesen jeweiligen

[2] https://docs.unity3d.com/ScriptReference/Vector3.MoveTowards.html
[3] https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

Momenten gemacht. Die aufeinanderfolgenden Screenshots werden daraufhin auf deren Ähnlichkeit überprüft Abbildung 5.1(4.3). Dies geschieht auf Basis einer Computervision-Library und wird in Abschnitt 5.5 genauer erläutert. Unterscheiden sich die beiden Bilder sehr, so wird der Screenshot der Kollision mit dem Ground Truth-Bild verglichen Abbildung 5.1(4.4). Ähneln sich nun das Bild der Kollision und das der Ground Truth, so erhält der Nutzer ein Feedback, dass es sich bei dem angefügten Collider um einen Fehler handelt, z.B. dass der Collider an der falschen Position bzgl dem dazugehörigen Gameobject angebracht wurde. Besteht keine Ähnlichkeit zwischen den Bildern, so erhält der Nutzer eine Nachricht, dass kein Fehler besteht und alles korrekt ist. Das Feedback Abbildung 5.1(5) an den Benutzer wird zum einen in eine Datei geschrieben, zum anderen aber auch im UnityEditor mitgeteilt.

Anhand der in Abschnitt 2.2 vorgestellten Klassifizierungen von Fuzzern, wird nun erläutert, wie der eigene entwickelte Prototyp eingeordnet werden kann. Zum einen könnte man den Fuzzer als generation-based Fuzzer betrachten, da das Eingabeformat bekannt ist (Vektor, der eine Rotation beschreibt). Gleichzeitig ist es auch ein Black-Box Fuzzer, da die zu testenden Eingaben zufällig generiert werden. Aufgrund der Abzielung auf die Kollisionsdetektion, und damit auf einen bestimmten Bereich im Quellcode, könnte der Prototyp als directed Fuzzer bezeichnet werden. Da keine Informationen über vorherige Fehler in die zukünftig generierten Testfälle miteinfließt, handelt es sich ebenso um einen Dumb Fuzzer. Der in Abbildung 2.2 beschrieben Fehlerfilter wurde nicht implementiert. Die Umsetzung der Funktionalität wurde manuell durchgeführt.

5.3.1 Komponenten - Gameobjects

In diesem Abschnitt wird die in Abbildung 5.1 vorgestellte Architektur in ihre Logikbausteine aufgespalten und deren Realisierung mithilfe von Gameobjects genauer erläutert.

Abbildung 5.2 zeigt alle implementierten und genutzten Gameobjects. Das XR Plug-in Management stellt ein Prefab [4] der Kamera mit entsprechender Funktionalität für die Nutzung von VR-Anwendungen zur Verfügung. Dieses Prefab, bestehend aus XRRig, Camera Offset und Main Camera wurde als Basis verwendet, diese erweitert und weitere Gameobjects mit notwendigen Funktionalitäten hinzugefügt. Diese sind RecognizeHit, Screenshot und AnalyseScreenshot. Alle Gameobjects zusammen mit deren Eigenschaften und Funktionalitäten bilden das selbst entworfene Prefab *CameraEnd*, das in andere Unity-Anwendungen importiert und daraufhin verwendet werden kann.

Im folgenden wird für jedes Gameobjects dessen Aufgabe erklärt, sowie die dazugehörigen Komponenten und Skripte beschrieben.

[4] https://docs.unity3d.com/Manual/Prefabs.html

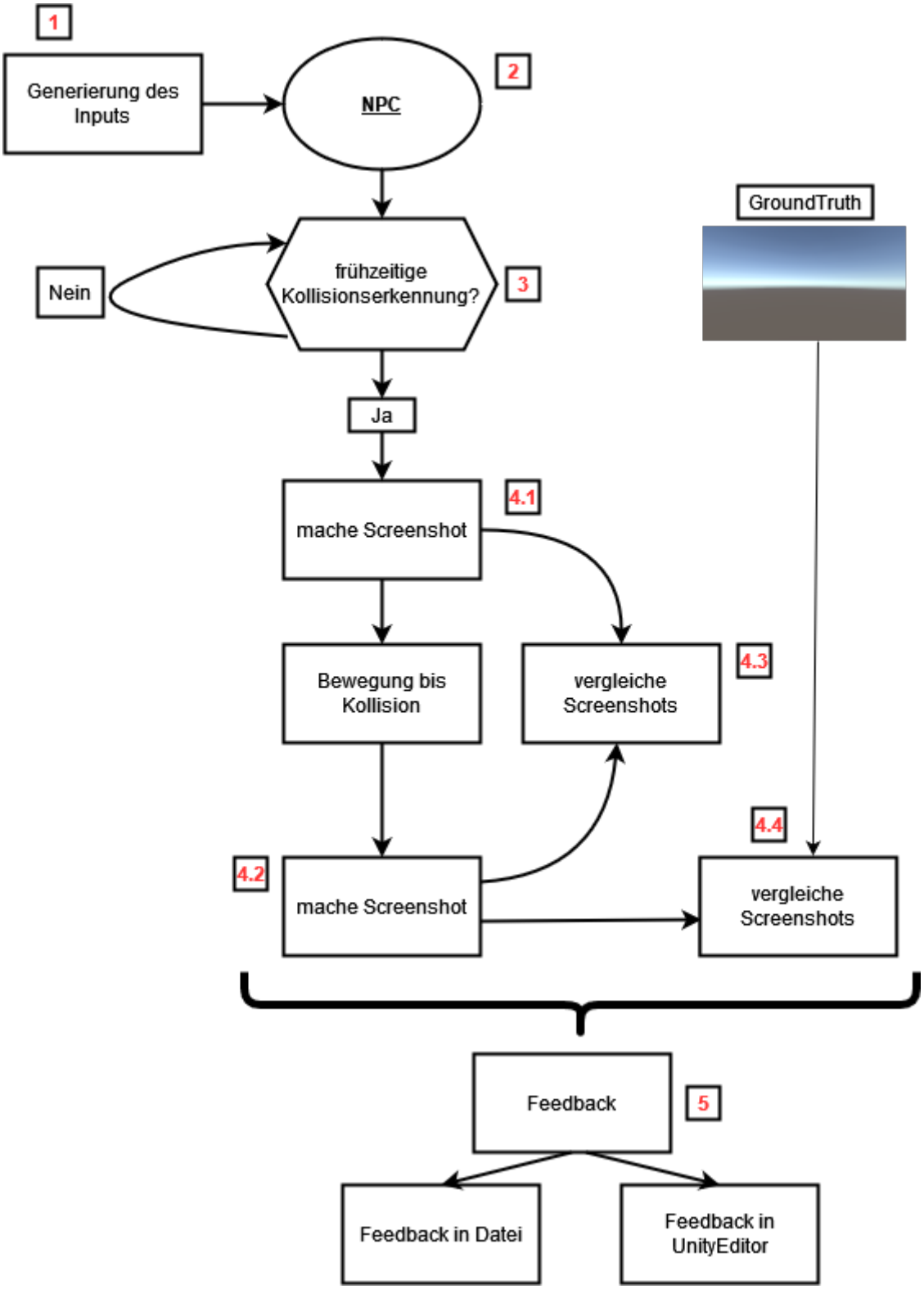


Abbildung 5.1: Arbeitsablauf des entwickelten Tools

Abbildung 5.2: Hierarchie der verwendeten Gameobjects, welche die komplette Logik der Anwendung beinhaltet

Das Gameobject CameraEnd bildet gemeinsam mit dem Child-Gameobject Camera Offset und dessen Child-Gameobject Main Camera die Steuerzentrale und gleichzeitig den Mittelpunkt in dieser Anwendung. Einzig das Gameobject CameraEnd wurde von den drei Gameobjects um diverse Komponenten und Skripte erweitert. Dazu gehört die Komponente Character Controller. Dies ist eine Kapsel der Größe zwei und soll einen Menschen mit aufgesetzter VR-Brille in einer virtuellen Welt simulieren. Die Kamera ist in der Höhe $Y = 1.75$ angebracht.

Die Abbildung 5.3 zeigt die Kapsel von der Character Collider Komponente mit einer integrierten Kamera. Das Blickfeld aus der Ego-Perspektive der Kapsel zeigt doe Abbildung 5.4. Das Slope Limit ist auf den Wert 45 und der stepOffset auf den Wert 0.5 gesetzt. Damit kann der NPC zum einen eine Rampe mit der Steigung von 45 Grad passieren, zum anderen können Hindernisse ab einer Höhe von 0.5 überlaufen werden. Ein Step Offset Wert von 0 bedeuten, dass zum Beispiel keine Treppe genutzt werden kann und schon die erste Stufe ein blockierendes Hindernis darstellt. Da ein Character Controller keinen Einfluss durch physikalische Kräfte erfährt, kann keine Kollision erkannt werden.

Abhilfe verschafft das Hinzufügen von einem Rigidbody und das Anbringen eines Capsulate Colliders, sodass bei einem Kontakt dieses Colliders mit einem andere Gameobject eine Kollision detektiert wird. Die Bewegung und demnach die Kollision wird von der Unity-Physics-Engine gesteuert.

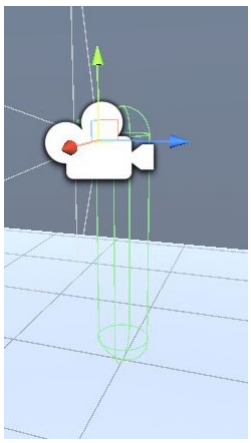


Abbildung 5.3: Character Controller Komponente mit der integrierten Kamera auf Augenhöhe

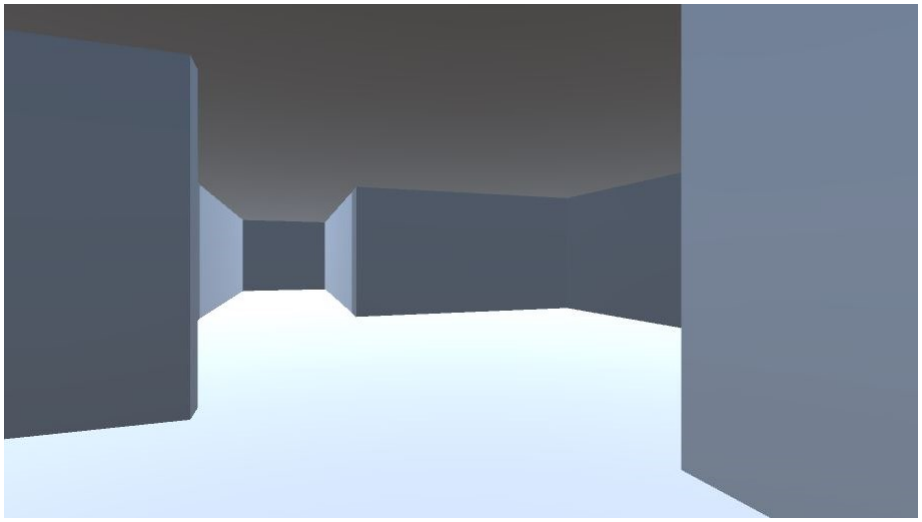


Abbildung 5.4: Exploration der Welt aus Sicht der Ego-Perspektive

Die beiden Skripte *InputRotationWalking.cs* und *CharacterControllerTry.cs* sind an das Gameobject CameraEnd angefügt. Das erstgenannte dient der Generierung der Inputdaten für den NPC. Die Anzahl der Inputdaten wird durch die Variable *numberRotations* bestimmt. Bei jedem einzelnen Inputdatum handelt es sich um Vektoren, die eine Rotationen um die Y-Achse beschreiben und die Form $(0, [0, 360], 0)$ haben. Das entspricht der horizontalen Drehung des Kopfes nach rechts und links. Der Y-Wert einer Rotation wird zufällig mit der Funktion *Random* der Klasse UnityEngine generiert. Das zentrale Skript mit der meist enthaltenen Logik ist *CharacterControllerTry.cs*. Darin werden andere Skripte, wie zum Beispiel *InputRotationWalking.cs* aufgerufen, die Kollisionserkennung durchgeführt, das Feedback an den Nutzer in Form einer Datei erstellt und der Bewegungsablauf des NPCs koordiniert. Außerdem ist es möglich, die wichtigsten Parameter, auch von anderen Skripten, wie zum Beispiel die Anzahl der Inputdaten, im Inspector zu ändern.

Die komplette Kollisionserkennung wird mithilfe der Methoden *OnCollisionEnter*, *OnCollisionStay* und *OnCollisionExit* umgesetzt. Beim Auftreten einer Kollision wird zunächst in *OnCollisionEnter* überprüft, ob ein Screenshot von einer zuvor möglichen Kollision gemacht wurde und dem Array *screenshotComparison* hinzugefügt wurde. Ist dies der Fall, so wird beim Eintreten der Kollision ein Screenshot gemacht und ebenfalls im Array eingefügt. Somit ergeben sich zwei aufeinander folgende Screenshots, die daraufhin der Methode *getComparisonResults* im Skript *AnalyseTwoScreenshots.cs* zur Analyse übergeben werden. Das daraus erhaltene Ergebnis wird über die Methode *writeResultsIntoFile.cs* in eine Datei geschrieben und beinhaltet , wie in Abbildung 5.5 zu sehen, fünf Werte der Kollision. Dazu gehört die ID, welche fortlaufend jeder Kollision einen eindeutigen Wert zuordnet. Die nächsten beiden Werte sind die Namen und die Koordinaten des Colliders eines Gameobjects, mit dem der NPC kollidiert ist. Die vierte Spalte beschreibt die Einstufung der Kollision - handelt es sich um einen Fehler oder ist der Collider korrekt angebracht worden. In der letzten Spalte stehen die Koordinaten des NPCs beim Eintreten der Kollision.

ID	ColliderName	CoordinatesCollider	Status	PositionPlayer
1	Wall	(0,0, 3,5, 10,0)	1	(6,0, 0,1, 9,7)
2	Wall3	(10,0, 3,5, 0,0)	2	(9,2, 0,1, 8,2)
3	Wall3	(10,0, 3,5, 0,0)	2	(9,2, 0,1, 9,0)
4	Wall3	(10,0, 3,5, 0,0)	2	(9,3, 0,1, 1,6)
5	Wall	(0,0, 3,5, 10,0)	1	(6,3, 0,1, 9,7)

Abbildung 5.5: Das Feedback an den Benutzer in Form einer Textdatei

Neben der angelegten Datei mit dem Feedback zu den auftretenden Kollisionen, wird dem Benutzer ebenso im UnityEditor ein Hinweis eines auftretenden Fehlers gegeben. In der Konsole wird bei einer Kollision der Name des Gameobjects ausgegeben. Klickt der Benutzer nun, wie die Abbildung 5.6 zeigt, auf die entsprechende Zeile (hier blau markiert) in der Konsole, so rückt das betroffene Gameobject mit der gelben Umrandung in den Vordergrund. Dies wird in Abbildung 5.7 dargestellt.

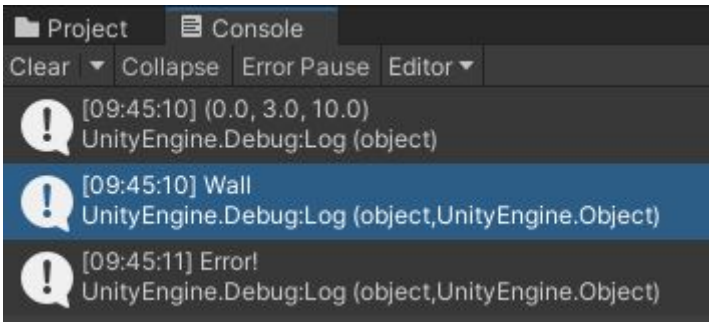


Abbildung 5.6: Das Feedback an den Benutzer über die Konsole, an welchem Gameobject ein möglicher Fehler aufgetreten ist. In diesem Fall an dem Gameobject *Wall*

Abbildung 5.7: Das gelb umrandete Gameobject *Wall* weist den Benutzer im Hierarchiefenster darauf hin, an welchem Gameobject ein möglicher Fehler aufgetreten ist

Nach dem Auftreten der Kollision wird in der Methode *OnCollisionStay* dafür gesorgt, dass der NPC eine Richtungsänderung seiner Bewegung vornimmt. Dazu wird zuerst der NPC um den Wert von *resetDistance* von der Kollisionposition zurückgesetzt und in eine zufällige Richtung innerhalb

des angegebenen Intervalls von *rangeMIN* und *rangeMAX* gesetzt. In der Methode *OnCollisionExit* wird die boolesche Variable *collided* auf true gesetzt. Diese Variable hat die Aufgabe, dass beim Eintreten der Kollision genau ein Screenshot gemacht wird. Wird diese Variable weggelassen, so würden viele Screenshots gemacht werden, da die Kollisionsabfrage jeden Frame durchgeführt wird.

Das Gameobject RecognizeHit hat die Aufgabe, eine frühzeitige Kollisionsabfrage zu machen. Ziel ist es frühzeitig zu erfahren, ob eine Kollision mit einem anderen Gameobject auftreten wird. Dazu wird die in dem angehängten Skript *EarlyRecognizeHit.cs* die *SweepTest*-Methode in der *Update*-Methode des Skripts verwendet. Dies trifft allerdings nur auf primitive Colliders (Sphere, Cube, Capsule) oder konvexe meshes zu. Bei Konkaven meshes wird diese Methode nicht funktionieren [5]. Der Zeitpunkt der Abfrage kann mit der Variablen *distanceRayCastHit* festgelegt werden. Dieser gibt die Entfernung an, ab der eine Abfrage für eine mögliche Kollision durchgeführt werden soll. Je größer dieser Wert ist, desto früher wird eine auftretende Kollision detektiert. *collisionRecognize* übernimmt dieselbe Aufgabe bei der Aufnahme eines Screenshots, wie es die Variable *collided* in dem darüberliegenden Abschnitt übernommen hat.

Das Gameobjects Screenshot ist für das Aufnehmen der Screenshots verantwortlich. Das hinzugefügte Skript *CaptureScreenshots.cs* umfasst die Koordination der Dateistruktur für das Abspeichern der Aufnahmen. Das Erstellen des Screenshots wird in einem separaten Thread ausgeführt. Sowohl die Größe, über die Länge und Breite, des Screenshots, als auch das Format des abzuspeichernden Bildes können im Inspector verändert werden. Die unterstützten Bildformate sind JPG und PNG.

An das letzte Gameobject AnalyseScreenshots ist das Skript *AnalyseTwoScreenshots.cs* angefügt, welches die Verbindung zu dem Python-Skript *sender.py* mithilfe von System.Diagnostics aufbaut. Dabei werden die beiden zu vergleichenden Screenshots dem Python-Skript übergeben und dort anhand der Library OpenCV analysiert. Als Rückgabe bekommt das Skript *AnalyseTwoScreenshots.cs* das Feedback, ob ein Fehler aufgetreten ist oder nicht. Die Vergleiche selbst werden über Histogramme der Bilder durchgeführt.

5.3.2 Parameter

Im Folgenden werden die im Prefab enthaltenen Parameter aufgezählt und erläutert. Diese können manuell in Unity im Inspector eingestellt werden.

Über den Parameter *AmountOfInputs* kann die Anzahl der vorab generierten Inputs festgelegt werden. Die beiden Parameter *stepSize* und *speed* regulieren die Fortbewegung der NPCs in der Welt. *Stepsize* legt fest, wie lange sich der NPC geradeaus bewegt, bevor eine Richtungsänderung und somit der nächste Input, ausgeübt wird. Je größer dieser Wert, desto länger bewegt sich der NPC geradeaus. *Speed* beschreibt die Bewegungsgeschwindigkeit, mit der sich der NPC durch die virtuelle Welt bewegt.

Die vier Parameter *resetDistance*, *rangeMIN* und *rangeMax*, sowie *distanceRayCastHit* beziehen sich auf die Kollisionserkennung und die Kollision selbst. *ResetDistance* entspricht der Strecke, um welche der NPC, ausgehend vom Kollisionspunkt, zurückgesetzt wird. Das Zurücksetzen erfolgt in Richtung der negativen Blickrichtung. Die Parameter *rangeMIN* und *rangeMAX* geben die

[5] https://docs.unity3d.com/ScriptReference/Rigidbody.SweepTest.html

untere und obere Grenze der möglichen Blickrichtung an, nachdem der NPC mit einem anderen GameObject kollidiert ist. Aus diesem Intervall wird zufällig ein Wert bestimmt, in dessen Richtung die Kamera des NPCs nach der Kollision dann zeigt. Dabei entspricht ein Wert von 0 genau der Richtung in welche die Kollision aufgetreten ist. Entsprechend beschreibt ein Wert von 90 den Blick nach rechts, ausgehend von der Kollision. Der Wert 180 entspricht genau der entgegengesetzten Blickrichtung beim Auftreten der Kollision. Mit dem Parameter *distanceRayCastHit* kann die Distanz eingestellt werden, ab der eine Abfrage für eine Kollisionserkennung durchgeführt wird. Ein großer Wert entspricht der Kollisionsabfrage aus einer großen Entfernung. Wird ein Wert nahe 0 gewählt, so wird die Kollisionsabfrage kurz vor der Kollision durchgeführt.

Die Größe der Screenshots kann über die Parameter *ImageWidth* und *ImageHeight* eingestellt werden.

5.4 Integration in Unity-Anwendungen

Für die Benutzung des entwickelten Prototyps empfiehlt es sich Unity 2020.3.3f [6], Python 3.9 [7] und OpenCV 4.5.2 [8] zu verwenden. Bei anderen Softwareversionen kann ein korrekt funktionierendes Verhalten nicht garantiert werden. Da die vollständige Funktionalität des Prototyps in ein Prefab verpackt ist, kann dieses Prefab direkt in ein Unity-Projekt importiert werden. Das Prefab mit dem Namen CameraEnd kann in Unity über Assets → Import Package → Custom Package ausgewählt werden. Daraufhin lässt es sich mittels Drag and Drop in das Hierarchiefenster der gewünschten Szene einbinden. Zusätzlich wird das XR PlugIn Management benötigt. Dies kann über den Package Manager in Unity installiert werden. In diesem Projekt wurde die Version 4.0.7 verwendet. Daraufhin kann über den Play Button das Projekt gestartet werden. Ergänzungen oder Modifizierungen können jederzeit am Prefab vorgenommen werden, indem der Open Prefab Button im Inspectorfenster betätigt wird und die Änderungen isoliert durchgeführt werden.

5.5 Metriken für die Histogramm-Vergleiche

Mit Hilfe der Open-Source-Bibliothek OpenCV 4.5.2 werden aus den gewonnenen Screenshots Histogramme berechnet. Diese werden mit Hilfe der zur Verfügung gestellten Metriken auf Ähnlichkeit überprüft. OpenCV bietet sechs verschiedene Metriken [9] für den Vergleich von zwei Histogrammen H1 und H2 an.: Correlation, Chi-Square, Intersection, Bhattacharyya distance, Alternative Chi-Square und Kullback-Leibler divergence.

Bei Correlation liegt das berechnete Ergebnis zwischen den Werten -1 und 1. Je näher das Ergebnis bei -1 liegt, desto unterschiedlicher sind die beiden Histogramme. Ist der Wert genau 1, so handelt es sich um identische Histogramme.

[6] https://docs.unity3d.com/Manual/index.html
[7] https://docs.python.org/3.9/reference/
[8] https://docs.opencv.org/4.5.2/d6/dc7/group__imgproc__hist.html#ga994f53817d621e2e4228fc646342d386
[9] https://docs.opencv.org/4.5.2/d6/dc7/group__imgproc__hist.html#ga994f53817d621e2e4228fc646342d386

Bei Chi-Square treten nur Werte größer oder gleich 0 auf. Je näher das berechnete Ergebnis an dem Wert 0 liegt, desto ähnlicher sind die beiden Histogramme.

Intersection benutzt den Minimalwert an derselben Position in den beiden Histogrammen und summiert diese Werte auf. Dabei existiert jedoch im Gegensatz zu den zwei zuvor vorgestellten Methoden kein Bezugswert.

Der Ergebnisbereich von Bhattacharyya distance liegt zwischen 0 und 1. Je näher sich das Ergebnis dem Wert 0 nähert, desto ähnlicher sind die zwei Histogramme. Nimmt das Ergebnis einen Wert nahe der 1 an, so ähneln sich die Histogramme nur sehr gering.

In dieser Arbeit kamen lediglich die drei Methoden Correlation, Chi-Square und Bhattacharyya distance zum Einsatz. Sowohl Kullback-Leiber divergence, als auch Intersection besitzen keinen Bezugswert und sind somit für diese Art der Anwendung nicht geeignet. Die Alternative Chi-Square-Methode wird hauptsächlich für Texturvergleiche verwendet und findet in dieser Arbeit daher auch keine Berücksichtigung.

6 Evaluierung

Dieses Kapitel präsentiert den Aufbau, die Durchführung und Ergebnisse der Evaluierung des in Kapitel 5 vorgestellten Systems. Dazu wurde der Prototyp in vier verschiedene Szenarien integriert, welche im nächsten Abschnitt genauer beschrieben werden, sowie getestet. In Abschnitt 6.3 werden die Ergebnisse der Evaluierung dargestellt. Abschließend werden im letzten Unterkapiteln die Ergebnisse diskutiert.

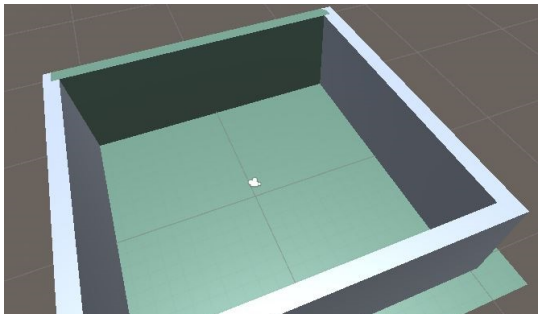
6.1 Szenarien

Für die Evaluierung des entwickelten Ansatzes wurden vier unterschiedliche Beispielszenen in Unity entwickelt. Drei Szenen basieren auf einzelnen Räumlichkeiten, die sich in ihrer Gestaltung, ihrer Komplexität, sowie in ihrer Größe unterscheiden. Es wurden an verschiedenen Stellen bewusst die in Abschnitt 5.1 beschriebenen Fehler synthetisch eingebaut, um zu untersuchen, ob der entwickelte Ansatz diese Fehler entdeckt. Für das vierte Szenario wurde ein fertiges Haus aus dem Unity AssetStore verwendet. In dieses wurde ebenso der NPC integriert und geprüft, ob die Fehler auch in realen Daten auftauchen und erkannt werden.

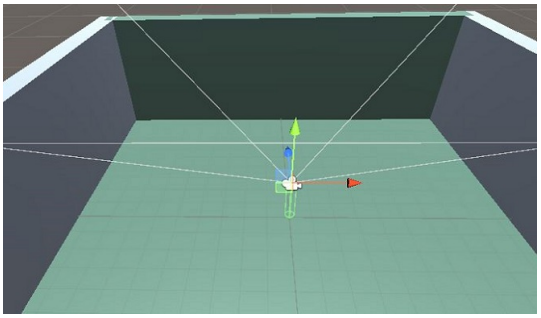
Für das vierte Szenario wurde ein fertiges Haus aus dem Unity AssetStore verwendet. In dieses wurde ebenso der NPC integriert und geschaut, ob auch in realen Daten diese Fehler auftauchen und erkannt werden.

6.1.1 Synthetisch erstellte Szenarien

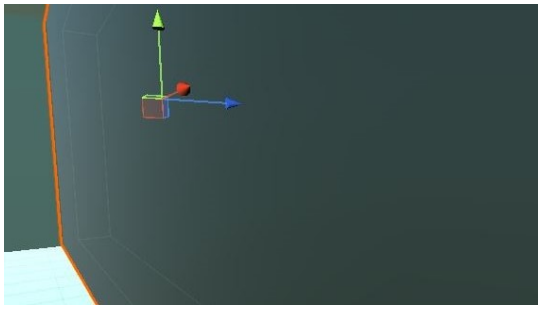
Das erste Szenario besteht aus einem quadratischen Raum. An den drei grauen Wänden sind die Boxcollider korrekt angebracht worden. An der grünen Wand wurde der Boxcollider um 0.5 Einheiten (in Meter) nach hinten verschoben, was in Abbildung 6.1 Bild (c) zu sehen ist. Die orangefarbenen Linien entsprechen der Umrandung des Gameobjects der Wand. Die grünen Linien rechts neben den orangefarbenen Linien beschreiben die Kanten des Colliders der Wand.



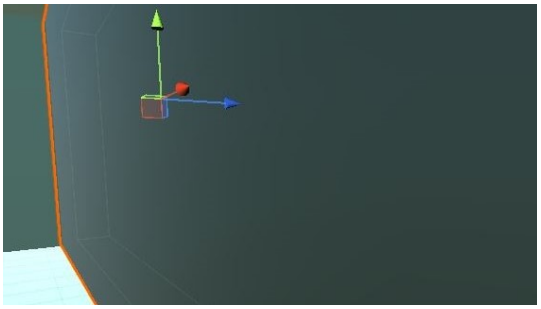
(a) Szenario 1 aus der Sicht von oben



(b) Der in das erste Szenario intergrierte NPC mit Blickrichtung auf die grüne Wand, in welche der Fehler 1 eingebaut wurde



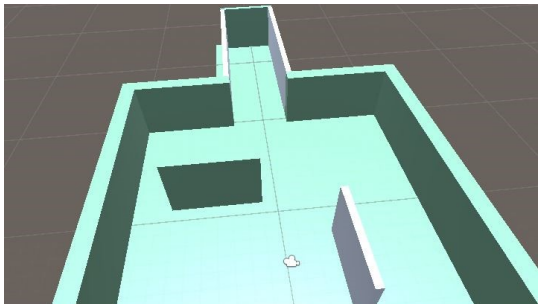
(c) Der Blick auf die grüne Wand mit dem Fehler 1: Die Umrandung des Gameobjects (orange) und der nach hinten versetzte Collider (grün)



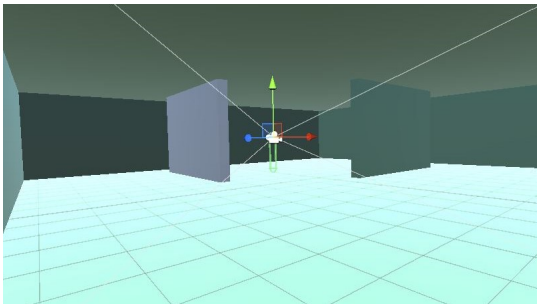
(d) Der Blick auf die grüne Wand mit dem Fehler 1: Die Umrandung des Gameobjects (orange) und der nach hinten versetzte Collider (grün)

Abbildung 6.1: Szenario 1 mit dem integrierten NPC wie in (b) zu sehen und dem einzigen eingebauten Fehler in der grünen Wand (c)

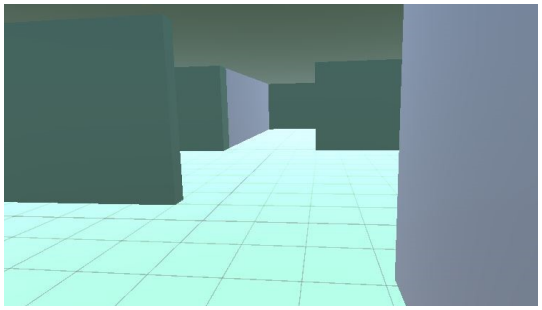
Das zweite Szenario ist eine Erweiterung des ersten Szenarios. Dazu wurde ein schmaler, langer Gang hinzugefügt und zwei weitere Wände in dem Raum platziert. Diese sollen als Hindernisse fungieren. Drei Fehler wurden in die Szene integriert. Der Fehler aus Szenario eins, ein weiterer in einem Teilbereich der Wand am Ende des langen Ganges und ein dritter Fehler in einer Ecke des Raumes.



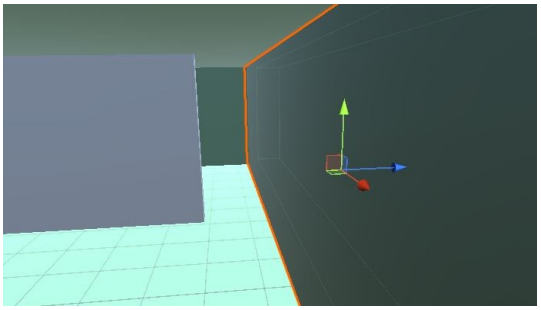
(a) Szenario 2 aus Sicht von oben



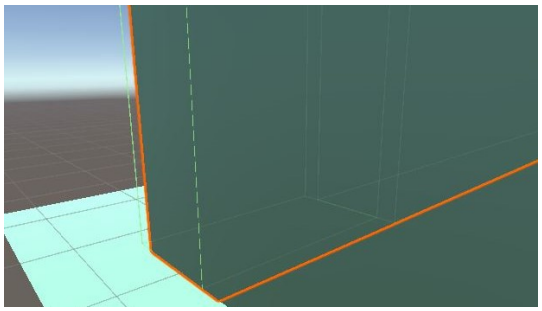
(b) Szenario 2 mit dem integrierten NPC



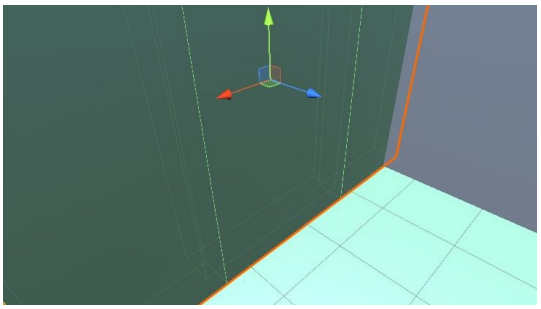
(c) Szenario 2 aus der Sicht des NPCs in Blickrichtung des langen Ganges



(d) Der Blick auf die Wand mit dem eingebauten Fehler 1: Die Umrandung des Gameobjects (orange) und der nach hinten versetzte Collider (grün) der kompletten Wand



(e) Der Blick auf die Wand mit dem eingebauten Fehler 2: Die Umrandung des Gameobjects (orange) und der teilweise nach hinten versetzte Collider (grün) der Wand

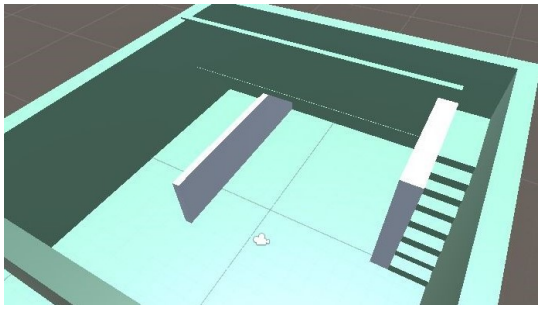


(f) Der Blick auf die Wand mit dem eingebauten Fehler 3 in der Ecke des Szenarios

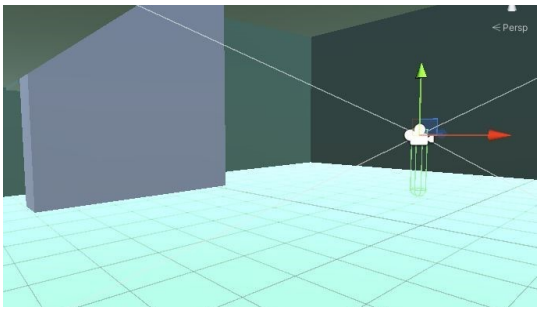
Abbildung 6.2: Das 2.Szenario aus verschiedenen Perspektiven (a)-(c). Bild (d)-(f) zeigt die eingebauten Fehler anhand der versetzten Collider (grün) bezüglich der Gameobjects (orange)

Das dritte Szenario ist ebenfalls ein quadratischer Raum. Im Gegensatz zum ersten Szenario wurde eine Treppe am Rand des Raumes hinzugefügt, welche in den Gang im zweiten Stock führt. Außerdem wurde eine zusätzliche Wand eingefügt, die den Raum unterteilt. Wie im zweiten Szenario, wurden in diesem Szenario auch drei Fehler eingebaut. Wie im ersten Szenario, ist auch in diesem der Boxcollider einer kompletten Wand um 0.5 Einheiten (in Meter) nach hinten verschoben

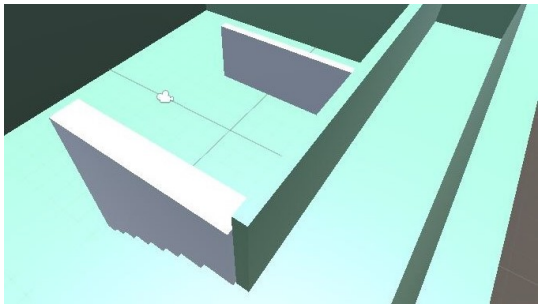
worden. Bei dem zweiten Fehler wurde nur einer bestimmter Ausschnitt eines Colliders einer Wand nach hinten verschoben. Dazu wurden mehrere Boxcollider verwendet und zusammengesetzt. Der dritte Fehler befindet sich im zweiten Stock am Ende des Ganges. Ähnlich zum zweiten Fehler wurde der Boxcollider der kompletten Wand in mehrere kleinere Boxcollider unterteilt und der entsprechende Ausschnitt für den Fehler nach hinten versetzt. Dies ist in Abbildung 6.2 (f) zu sehen.



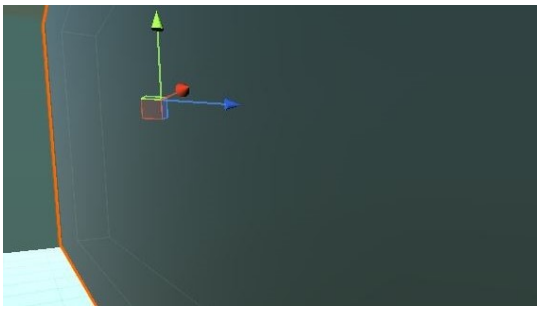
(a) Szenario 3 aus Sicht von oben auf den kompletten Raum



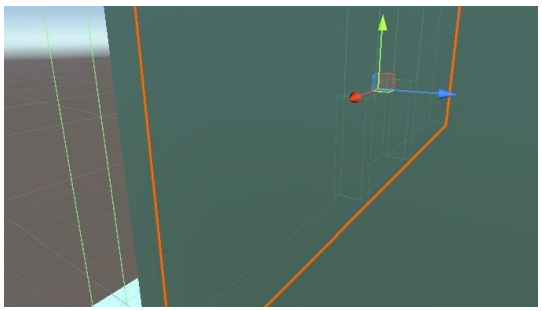
(b) Szenario 3 mit dem integrierten NPC mit Blick auf die Treppe



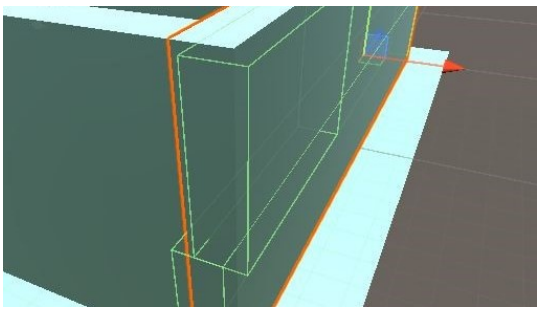
(c) Szenario 3 aus der Sicht von oben auf den Gang im zweiten Stock



(d) Der Blick auf die Wand mit dem eingebauten Fehler 1: Die Umrandung des Gameobjects (orange) und der nach hinten versetzte Collider (grün) der kompletten Wand



(e) Der Blick auf die Wand unterhalb des zweiten Stocks mit dem eingebauten Fehler 2: Die Umrandung des Gameobjects (orange) und der teilweise nach hinten versetzte Collider (grün) der Wand



(f) Der eingebaute Fehler 3 an der Wand am Ende des Ganges im zweiten Stock

Abbildung 6.3: Das 3. Szenario aus verschiedenen Perspektiven (a)-(c) und die drei eingebauten Fehler (d)-(f)

6.1.2 Real Data Szenario

(a) Das gesamte Inventar an Gameobjects in diesem Haus

(b) Das Haus aus der Sicht von außen

(c) Die Küche mit Blick aus dem Fenster

(d) Das Wohnzimmer mit einem Sofa, einem Kamin und einem TV

(e) Das Schlafzimmer mit einem Bett

(f) Das Badezimmer mit einem Waschbecken, einem Fenster und einer Dusche

Abbildung 6.4: Das Haus für das Real Data Szenario, bestehend aus drei Räumen mit all der Einrichtung. Der größte Raum besteht aus der Küche und dem Wohnzimmer, welcher durch einen Holztisch geteilt wird. Dieser Raum führt sowohl in das Schlafzimmer, als auch in das Badezimmer. Einen Durchgang vom Schlafzimmer in das Badezimmer gibt es nicht.

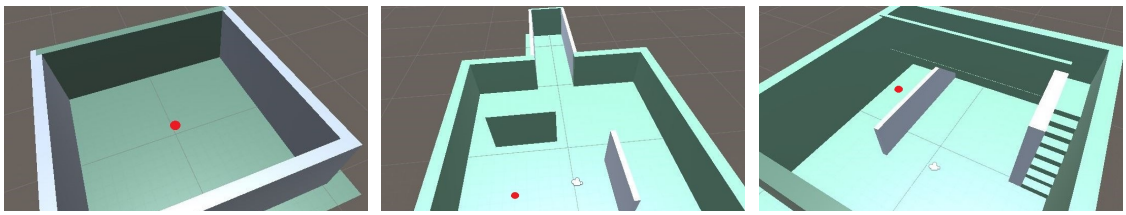
Für das Real Data Szenario wurden einige Assets aus dem Unity Asset Store heruntergeladen und in eine leere Szene integriert. Beim Versuch, den NPC zu importieren und die Anwendung auszuführen, wurde ersichtlich, dass bei drei von vier Assets keine Collider an die Gameobjects

angebracht worden waren. Somit bewegte sich der NPC durch jegliche Gameobjects, zum Beispiel durch Wände oder Stühle. Deshalb konnten diese Assets nicht in die Evaluierung einbezogen werden.

Folglich wurde das in Abbildung 6.4 Bild (b) dargestellte Haus [1] aus dem Asset Store genutzt. Dieses besteht aus einer Etage, umfasst für alle verwendeten Gameobjects Collider und besteht aus drei Räumen - ein Badezimmmer, ein Schlafzimmer und ein dritter Raum, der ein Wohnzimmer und eine Küche enthält. In jedem Raum sind Fenster eingebaut. Das gesamte Inventar an verwendeten Gameobjects ist in Abbildung 6.4 Bild (a) dargestellt.

6.2 Aufbau und Durchführung

In jedes der vier Szenarien wurde der NPC importiert und an die jeweilige Position innerhalb des Raumes bzw des Hauses gesetzt (siehe Abbildung 6.5), um ihn das unbekannte Gebiet erkunden zu lassen. Für die Evaluierung wurden mehrere Durchläufe mit unterschiedlichen Parametereinstellungen vorgenommen. Folgende Parameter wurden im Laufe der Testdurchläufe verändert: Die Anzahl der Inputdaten (*AmountOfInputs*), der Winkel für die Richtungsänderung nach einer Kollision (*rangeMIN* und *rangeMAX*), die Strecke, um die NPC nach einer Kollision zurückgesetzt wird (*resetDistance*) und die Schrittweite (*stepSize*).



(a) Der Startpunkt des NPCs in Szenario 1

(b) Der Startpunkt des NPCs in Szenario 2

(c) Der Startpunkt des NPCs in Szenario 3

Abbildung 6.5: Die roten Punkte stellen den Startpunkt des NPCs in allen drei Szenarien bei der Durchführung der Evaluierung dar

Die Testdurchläufe wurden mit 30, 50, 100 und 300 Inputdaten durchgeführt. Der Winkel wurde einmal auf das Intervall [0,360] gesetzt und beim anderen Mal auf [120,260]. Der Parameter *resetDistance* blieb in all den Testdurchläufen 0.5. Die Schrittweite wurde jeweils auf die Werte 500, 1000 und 2000 gesetzt.

Die Wahl der beiden Parameter für die Histogramm-Vergleiche wurde vorab anhand einer ausgewählten Stichprobe experimentell bestimmt. Dazu wurden 30 mögliche Paare von Bildern verwendet, die einen Fehler darstellten. Parameter 1, der für die Ähnlichkeit der beiden aufeinanderfolgenden Screenshots verantwortlich ist, ergab die besten Ergebnisse für den Wert 0.4. Der zweite Parameter,

[1] https://assetstore.unity.com/packages/3d/environments/urban/furnished-cabin-71426

der den zweiten Screenshot mit dem Bild der Ground Truth vergleicht, ergab die besten Ergebnisse für den Wert 0.5. Als Vergleichsmethode der Histogramme wurde Correlation verwendet. Diese ergab im Vergleich zu den anderen Methoden die für diese Stichprobe zuverlässigsten Ergebnisse.

Für jeden Durchlauf einer Parametereinstellung wurde zum einen die Anzahl der Screenshot-Paare gemessen, zum anderen wurde erfasst, ob alle Fehler gefunden wurden. Außerdem wurde manuell überprüft, ob False Positives oder False Negatives in einem Durchlauf auftraten.

6.3 Ergebnisse

Im ersten Szenario wurde der eingebaute Fehler 1 mit jeder Parametereinstellung erkannt. Gleichzeitig traten in diesem Szenario allerdings 21 False Negatives während der Ausführung bei 156 durch den NPC entdeckte Fehler auf. Damit sind 12,18% der erkannten Fehler als False Negatives einzuordnen, d.h. in etwa jeder achte Fehler ist ein False Negative. Einzig bei der Einstellung mit 30 Inputs wurden keine False Negatives entdeckt. Das geringwertigste Ergebnis erzielte der Durchlauf mit einer Schrittweite von 1000, einem Blickwinkel von 100−260 und der Inputgröße von 100. Hier wurden fünf False Negatives entdeckt. Diese Ergebnisse sind in Tabelle 6.1 abgebildet.

		# Images			# entdeckter Fehler			# FN		
Schrittweite	# Inputs / Blickwinkel	30	50	100	30	50	100	30	50	100
500	0−360	18	20	42	1	1	1	0	1	1
	100−260	16	24	46	1	1	1	0	1	1
1000	0−360	19	37	53	1	1	1	0	1	0
	100−260	22	35	58	1	1	1	0	2	5
2000	0−360	19	29	68	1	1	1	0	1	1
	100−260	27	43	76	1	1	1	0	1	2

Tabelle 6.1: Szenario 1 – Ergebnisse der Evaluierung

Wie Tabelle 6.2 zeigt, beträgt die gefundene Anzahl an Fehlern im zweiten Szenario bei 30 Inputdaten durchgehend eins. Es war in jedem Durchlauf der am nächsten liegende eingebaute Fehler 1, der ausgehend vom Startpunkt direkt im Blickfeld des NPCs lag. Bei 100 Inputdaten wurden durchgehend zwei Fehler gefunden. Häufig wurden die Fehler 1 und 2 in einem Durchgang gefunden, aber auch die Kombination aus dem Fehler 1 und 3 trat auf. Alle drei Fehler wurden mit folgender Parametereinstellung gefunden: Die Schrittweite betrug 1000, der Blickwinkel wurde durch die Werte 100 und 260 beschrieben und die Anzahl der Inputs war auf 50 gesetzt. Insgesamt hat der NPC in allen Durchläufen 12 False Negatives gefunden. Bzgl der Gesamtheit der erkannten Fehler von 162, ergibt dies einen Wert von 7,40% und damit jeder dreizehnte erkannte Fehler entspricht durchschnittlich einem False Negative. Von den oben genannten 12 False Negatives ergaben sich fünf durch die Parametereinstellung mit der Schrittweite von 2000, dem Blickwinkel von 100−260 und die Anzahl der Inputs von 50 bzw. 100.

Der Tabelle 6.3 ist zu entnehmen, dass im dritten Szenario, ebenso wie im zweiten Szenario, einer von drei möglichen Fehlern bei 30 Inputdaten entdeckt wurde. Auch hier war es Fehler 1. Bei 100 Inputdaten wurden immer die zwei Fehler 1 und 2 gefunden. Eine Ausnahme trat jedoch bei der

		# Images			# entdeckter Fehler			# FN		
Schrittweite	# Inputs / Blickwinkel	30	50	100	30	50	100	30	50	100
500	0–360	19	28	52	1	1	2	0	0	2
	100–260	23	31	69	1	1	2	0	0	1
1000	0–360	19	26	57	1	1	2	0	2	1
	100–260	24	40	78	1	3	2	0	1	0
2000	0–360	21	36	62	1	2	2	0	0	0
	100–260	24	44	85	1	2	2	0	2	3

Tabelle 6.2: Szenario 2 – Ergebnisse der Evaluierung

Schrittweite von 2000 und dem Blickwinkel von 0−360 auf. Auch in diesem Szenario wurden bei einer bestimmten Parametereinstellung alle drei eingebauten Fehler vom NPC gefunden. Dabei war die Schrittweite auf 500 gesetzt, der Blickwinkel betrug 0−360 und die Anzahl der Inputdaten lag bei 50. In all den Testdurchläufen wurden 16 False Negatives bei 191 entdeckten Fehlern gefunden. Dies entspricht einem Wert von 8,47% und somit ist in diesem Szenario im Durchschnitt etwa jeder zwölfte Fehler ein False Negative.

		# Images			# entdeckter Fehler			# FN		
Schrittweite	# Inputs / Blickwinkel	30	50	100	30	50	100	30	50	100
500	0–360	18	30	56	1	2	3	2	1	1
	100–260	23	33	65	1	2	2	1	2	2
1000	0–360	14	33	66	1	2	1	0	1	0
	100–260	27	41	76	1	2	1	0	1	0
2000	0–360	21	33	65	1	1	1	0	0	2
	100–260	23	48	74	1	2	2	0	2	1

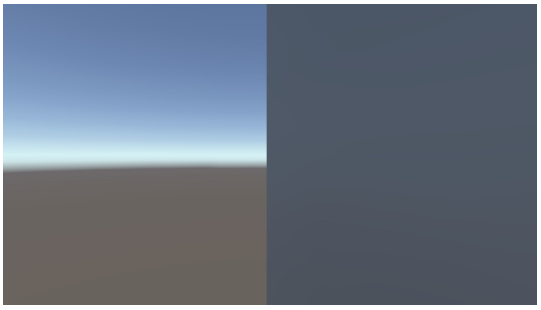
Tabelle 6.3: Szenario 3 – Ergebnisse der Evaluierung

In allen drei Szenarien ist zu erkennen, dass die Anzahl der Screenshot-Paare linear mit der Anzahl der Inputs zunimmt. Zusätzlich werden bei der Blickwinkeleinstellung von 100−260 mehr Screenshot-Paare gemacht, als bei den Werten 0 und 360. In keinem der Testdurchläufe wurden False Positives gefunden. Im Gegensatz dazu wurden in jedem Szenario False Negatives entdeckt. Wie in den Beispielen von Abbildung 6.6 erkennbar ist, sind dies ausschließlich Screenshots, in denen nur Teile des Horizonts zu sehen sind.

(a) Bildpaar 1: Bild 1

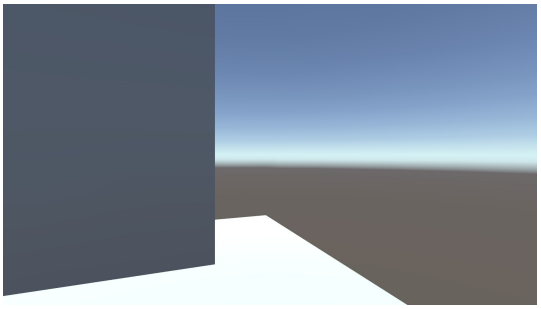
(b) Bildpaar 1: Bild 2

(c) Bildpaar 2: Bild 1

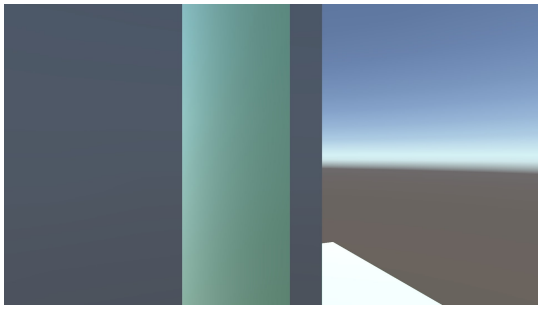


(d) Bildpaar 2: Bild 2

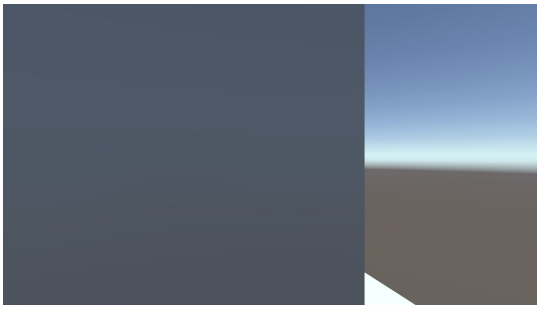
(e) Bildpaar 3: Bild 1



(f) Bildpaar 3: Bild 2



(g) Bildpaar 4: Bild 1

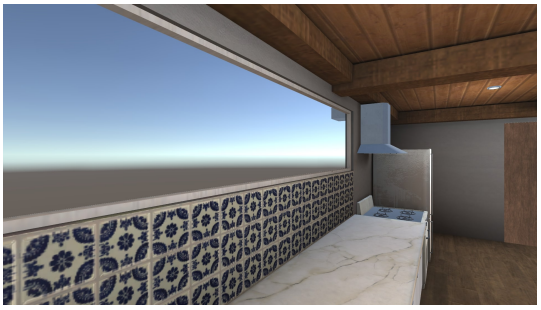


(h) Bildpaar 4: Bild 2

Abbildung 6.6: Auftretende Beispiele von False Negatives während der Durchführung der Evaluierung. Die linke Hälfte der Bilder entsprechen dem ersten Screenshot kurz vor dem Auftreten der Kollision. Die Bilder auf der rechten Seite entsprechen den Screenshots beim Auftreten der Kollision

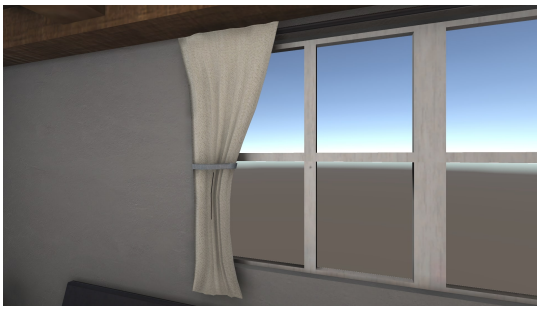
Im Real Data Szenario war der Startpunkt des NPC an der Haustüre innerhalb des Raumes mit dem Blick in Richtung Schlaf- und Badezimmer. Von dort hat er jeden Raum erkundet und bei der Anzahl von 100 Inputdaten insgesamt 57 Screenshot-Paare aufgenommen. Dennoch wurden keine Fehler entdeckt. Allerdings traten erstmals innerhalb der vier Szenarien Beispiele für False-Positives auf, wie in Abbildung 6.7 dargestellt ist. Von den insgesamt 57 gemachten Screenshot-Paaren waren 12 davon False-Positives. Die restlichen 45 Bildpaare wurden richtigerweise als fehlerfrei klassifiziert. Wie schon die Beispiele in Abbildung 6.7 zeigen, entstanden alle False-Positives bei einer Kollision vor einem Fenster mit dem Blick nach draußen.

(a) Bildpaar 1: Bild 1

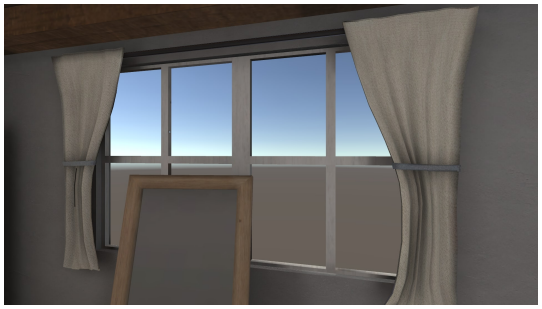


(b) Bildpaar 1: Bild 2

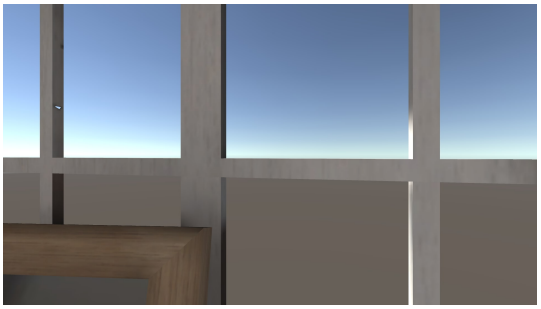
(c) Bildpaar 2: Bild 1



(d) Bildpaar 2: Bild 2



(e) Bildpaar 3: Bild 1



(f) Bildpaar 3: Bild 2

Abbildung 6.7: Auftretende Beispiele von False-Positives während der Durchführung der Evaluierung. Die linke Hälfte der Bilder entsprechen dem ersten Screenshot kurz vor dem Auftreten der Kollision. Die Bilder auf der rechten Seite entsprechen den Screenshots beim Auftreten der Kollision

Im Vergleich dazu ist in Abbildung 6.8 ein Screenshot-Paar dargestellt, welches richtigerweise kein Fehler ist. Im folgenden Abschnitt 6.4 wird näher auf die mögliche Ursache dafür eingegangen.

(a) Bildpaar 1: Bild 1 **(b)** Bildpaar 1: Bild 2

Abbildung 6.8: Ein Beispiel, welches korrekterweise nicht als Fehler entdeckt wurde

6.4 Diskussion

Die Ergebnisse der Evaluierung zeigen, dass die Integration der Fuzzing-Methodik in die Testumgebung von VR-Anwendungen sinnvoll ist um spezielle Fehler, wie in diesem Fall die Auftreten von Glitches an Wänden, zu erkennen. Jedoch wurden bei dem in dieser Arbeit entwickelten Prototyp einige Grenzen deutlich. Diese werden im Folgenden beschrieben.

Voraussetzung für eine Fehlerdetektion ist es, dass der NPC während der Erkundung der Welt Bildschirmaufnahmen macht und diese daraufhin auf Ähnlichkeit geprüft werden. Anhand der Ergebnisse lässt sich erkennen, dass mit der Einstellung des Blickwinkels (*rangeMIN*, *rangeMAX*) zwischen 100 und 260 Grad mehr Screenshots aufgenommen werden, als mit der Parametereinstellung 0 bis 360 Grad. Mehr Screenshots bedeuten zum einen eine höhere Chance der Fehlerdetektion. Jedoch steigt zum anderen auch die Wahrscheinlichkeit Beispiele von False Negative zu erhalten.

Der Grund für die höhere Anzahl an Screenshots bei der Parametereinstellung *rangeMIN* = 100 und *rangeMAX* = 260 liegt darin, dass die Blickrichtung nach der Kollision in die entgegengesetzte Richtung zeigt. Dieselbe Position mit der vorherigen Blickrichtung ist somit nicht mehr möglich. Bei der Einstellung von 0 und 360 Grad kann es passieren, dass der NPC die Blickrichtung nach der Kollision beibehält. Dies kann zur Folge haben, dass an derselben Stelle erneut eine Kollision auftritt, jedoch nicht erkannt wird. Tritt dieses Szenario ein, folgt keine Kollisionsabfrage und damit auch keine Aufnahme eines Screenshots.

Das Ganze tritt auf, wenn der Parameter *resetDistance* kleiner gewählt wird als der Parameter *earlyHitDistance*. *EarlyHitDistance* gibt die Distanz an, zu der geprüft wird, ob eine Kollision auftritt. Wird diese zum Beispiel auf den Wert 10 gesetzt, werden Kollisionen erkannt, die genau 10 Einheiten (in Meter) entfernt sind. *ResetDistance* legt fest, um wie viele Einheiten der NPC nach einer Kollision zurückgesetzt wird, um beispielsweise 7. Befindet sich der NPC nach einer Kollision dann um 7 Einheiten von einem Gameobject entfernt, so wird keine erneute Kollision detektiert, da die Überprüfung nur einer Distanz von exakt 10 Einheiten durchgeführt wird.

Der nächste zu erwähnender Punkt ist die Rate der False Negatives. Diese liegt in den drei selbst erstellen Szenarien zwischen 7,40% und 12,18%. Demnach kann in etwa jeder zehnte Fehler als False Negative betrachtet werden. Grund ist, dass die Methode des Screenshot-Vergleichs sehr einfach gehalten ist. Der reine Vergleich von zwei Histogrammen beinhaltet beispielsweise kein Tracking von Pixelinformationen. Somit können schon leichte Unterschiede bei den Screenshots, beispielsweise bei unterschiedlicher Beleuchtung, gravierende Auswirkungen auf die Ähnlichkeit zweier Histogramme haben. Eine Methodik, die mehr als den reinen Vergleich zwischen zwei unabhängigen Bildern berücksichtigt, nämlich beispielsweise den zeitlichen Verlauf der Pixelinformation miteinbezieht, wäre eventuell genauer. Des Weiteren gibt es keine optimale Parametrisierung der Thresholds bei den Screenshot-Vergleichen für jedes Szenario. Um gute Ergebnisse zu erhalten, sollte dies für jede VR-Anwendung individuell festgelegt werden.

Die Reihenfolge, in der die Fehler entdeckt werden, hängt vom Startpunkt des NPC im jeweiligen Szenario ab. Dies ist ein weiterer Aspekt, den es in diesem Abschnitt zu erwähnen gilt. Es ist kaum vermeidbar, dass in Szenario zwei und drei der Fehler 1 zuerst entdeckt wird. Dieser ist dem Startpunkt des NPCs am nächsten und hat darüber hinaus die größte Fläche: die komplette Wand und nicht nur einen Ausschnitt davon. Somit ist die Wahrscheinlichkeit für das Auftreten einer Kollision am höchsten.

Grundsätzlich wurden mit dem entwickelten Fuzzing-Ansatz alle eingebauten Fehler in den drei selbst erstellten Szenarien entdeckt. Die unterschiedlichen Parametereinstellungen sind jedoch nicht für alle Szenarien geeignet und bringen gewisse Nachteile mit sich, was die räumliche Exploration angeht.

Ein Problem tritt zum Beispiel bei langen, schmalen Gängen auf. Nur, wenn die Blickrichtung auf $0-360$ gesetzt und die Schrittweite genügend groß gewählt wird, besteht die Chance, dass der NPC den Gang erfolgreich bzw. zügig durchschreitet. Bei einer zu klein gewählten Schrittweite bewegt sich der NPC meist im Zickzack durch den Gang, so dass die Wahrscheinlichkeit abnimmt, zeitnah auf der gegenüberliegenden Seite anzukommen. Dies ist ein Grund, weshalb in Szenario 3 die Anzahl der Inputs sehr hoch sein muss, dass der NPC den Fehler 3 überhaupt erreicht und erkennt.

Ein weiterer Punkt ist das Auftreten von False-Positives im Real Data Szenario. Der erste Screenshot weist beim Histogrammvergleich kaum eine Ähnlichkeit zur Ground Truth auf, der zweite zeigt jedoch eine hohe Übereinstimmung. Da die Aufnahme sehr nah am Fenster gemacht wurde, ist auf dem Bild größtenteils der Horizont zu sehen, weshalb ein Fehler detektiert wird. Im Gegensatz dazu zeigt das Beispiel in Abbildung 6.8 eine korrekte Auswertung des Screenshot-Paares. Dies liegt daran, dass die Aufnahme des zweiten Screenshots bei der Kollision aus zu großer Entfernung und einem gewissen Neigungswinkel, gemacht wurde und es keine frontale Aufnahme ist. Das führt dazu, dass der Screenshot und die Ground Truth kaum eine Ähnlichkeit aufweisen.

Alles in allem kann auf Basis der aufgeführten Punkte folgendes Fazit gezogen werden: Sowohl die Explorationsstrategie in der virtuellen Welt als auch die Bildvergleichsmethode haben einen großen Einfluss auf das Endergebnis. Grenzen innerhalb dieser Methoden bedeuten zum einen eine weniger effiziente, zum anderen eine weniger genaue Fehlerdetektion. Ideen zu möglichen Verbesserungen werden in Kapitel 7 vorgestellt.

7 Zusammenfassung und Ausblick

Diese Arbeit liefert einen ersten Versuch die automatisierte Testmethode Fuzzing in die Entwicklung von VR-Anwendungen zu integrieren. Es sollte geprüft werden, ob die Methode auch bei VR Software erfolgreich Schwachstellen identifiziert.

Zu Beginn der Arbeit wurden Interviews mit verschiedenen Personen des Visualisierungsinstitut der Universität Stuttgart (VISUS) geführt, die Erfahrungen in der VR- und AR-Entwicklung aufwiesen. Ziel war es eine Übersicht über Fehler und Probleme zu erhalten, die typischerweise bei der Entwicklung und Nutzung von VR-Anwendungen auftreten. Im Anschluss wurde ein spezifischer Fehler herausgegriffen, bei dem die Hoffnung bestand, dass er mit Fuzzing gut zu entdecken ist: *Wall-Glitches*.

Es wurde daraufhin ein Ansatz in Unity entwickelt, der es erlaubt einen NPC in eine VR-Anwendung in Unity zu integrieren. Der NPC soll die virtuelle Welt erkunden mit dem Ziel den oben genannten Fehler zu identifizieren. Es wurden drei unterschiedlich komplexe Szenarien in Unity erstellt, bei denen dieser Fehler mehrfach synthetisch eingebaut wurde. Die Fehlerdetektion erfolgt durch den Vergleich von Screenshots, welche der NPC beim Auftreten einer Kollision mit GameObjects erstellt. Diese werden der Ground Truth gegenübergestellt,

Die Ergebnisse zeigen, dass es in allen drei Szenarien tatsächlich eine hohe Rate an Fehlerentdeckung gab. Jedoch traten zudem False Negatives auf: von den analysierten Screenshots wurden in den drei Szenarien zwischen 7.40% und 12.18% der gefundenen Fehler nicht als solche erkannt und falsch klassifiziert. Die Methode des Screenshot-Vergleichs kam hier an ihre Grenze.

Zum Abschluss wurde der Prototyp in einem Real Data Szenario aus dem Unity Asset Store getestet. Der gesuchte Fehler wurde in diesem Szenario nicht entdeckt. Ebenso traten keine False Negatives auf. Jedoch gab es False Positives, das heißt es wurden fälschlicherweise Fehler gemeldet.

Zusammenfassend lässt sich sagen, dass der entwickelte Ansatz ein erster, vielversprechender Research-Prototyp darstellt, der jedoch durch seine einfach gewählten Methoden an einigen Punkten an seine Grenzen kommt. Im nächsten Abschnitt wird auf diese Herausforderungen eingegangen und Ideen für mögliche Verbesserungen vorgestellt.

Ausblick

Während der gesamten Arbeit wurden einige Herausforderungen deutlich. Im Zusammenhang damit entstanden Ideen für Verbesserungen, die eine gute Ausgangsbasis für künftige Arbeiten bieten.

Zu Beginn der vorliegenden Arbeit wurde die Architektur entworfen. Aufgrund der begrenzten Bearbeitungszeit wurde für jede Komponente zunächst eine einfache Methode gewählt. Dies führte jedoch zu diversen Problemen, die in künftigen Arbeiten mittels komplexerer Ansätze vermieden werden könnten. Im Folgenden sind Ideen dazu beschrieben.

Die Erkundung der virtuellen Welt durch den NPC hängt im entwickelten Prototyp größtenteils vom Zufall ab. Eine effizientere Erkundung, bei der interessante Stellen zuerst angelaufen werden, wäre jedoch besser. Informationen während der Erkundung könnten gespeichert werden, so dass dieses Wissen für künftige Entscheidungen nutzbar ist. Mehrfaches Aufsuchen desselben Ortes könnten so vermieden werden. Zudem könnte der NPC nach Start priorisiert Stellen mit möglichen Fehlern aufsuchen, womit in Summe auch Zeit gespart würde. Im besten Fall gäbe es eine vollständige Exploration des Szenarios um die Wahrscheinlichkeit einen Fehler zu finden zu maximieren. Es wäre beispielsweise möglich einen Reinforcement Learning Ansatz zu wählen. Gordillo et al. [GBTG21]) befassen sich mit dem Problem der automatischen Erkundung und Bewertung eines gegebenen Szenarios unter Verwendung von Agenten mit Reinforcement Learning. Diese sind darauf trainiert, die Abdeckung des Spielzustands zu maximieren.

Ein weiterer Verbesserungsvorschlag betrifft die Erkennung der Collider. Bisher werden nur Collider auf Höhe der Kamera und in Blickrichtung erkannt, so dass keine Reaktion erfolgt, wenn der NPC beispielsweise gegen einen Tisch auf Hüfthöhe läuft. Im schlechtesten Fall muss abgewartet werden, bis der NPC einen Richtungswechsel vornimmt. Besonders schlecht wäre ein Szenario mit vielen Hindernissen, die unter Kamerahöhe liegen, jedoch größer sind als *stepOffset* der Character Controller Komponente (Die Höhe ab der ein Hindernis überlaufen werden kann). In diesem Fall wäre eine schnelle bzw. pragmatische Lösung die Schrittweite (*stepSize*) zu verringern, damit schneller die Richtung gewechselt werden kann.

Im Abschnitt 6.4 wurde schon beschrieben, dass bei einem einfachen Histogrammvergleich der Screenshots teilweise keine guten Ergebnisse bei der Prüfung der Ähnlichkeit erzielt werden. Verbesserungen könnte hier zum Beispiel die Verwendung von Neural Networks (NN) oder Recurrent Neural Networks (RNN) liefern. Dabei könnten, basierend auf den Bildern der Ground Truth, weitere Bilder synthetisch erstellt werden, zum Beispiel dasselbe bzw. ähnliche Bilder mit einer anderen Helligkeit oder Sättigung. Auf diese Weise würde man eine große Anzahl an Bildern erhalten, welche für das Trainieren des Neuronalen Netzes oder des Recurrent Neural Networks verwendet werden könnte.

Eine weitere Möglichkeit den Bilderverlauf einer Bildsequenz zu analysieren wäre ein Ansatz mit der Information des optischen Flusses (optical flow). Dieser beschreibt die zeitliche Verschiebung von Grauwerten bzw. Farbwerten in einem Pixel zwischen zwei oder mehreren Bildern. Dabei sollten bei zwei aufeinanderfolgenden Bildern keine zu großen Unterschiede auftauchen. Ist dies der Fall, so wäre das ein mögliches Indiz dafür, dass ein Fehler aufgetreten ist und der Collider möglicherweise nicht korrekt angebracht wurde. Darüber hinaus könnte der optische Fluss mit Machine Learning kombiniert werden, um zukünftige Bilder (Frames) vorherzusagen [WYL18].

Eine weitere Herausforderung ist es einen Fehler zu detektieren, wenn an ein Gameobject kein Collider angebracht wurde. Der NPC bewegt sich in diesem Fall durch das Gameobject hindurch. Ideen für Verfahren, die solche Gameobjects trotzdem erkennen, sind in künftigen Arbeiten willkommen.

Der letzte zu erwähnende Punkt ist die rein horizontale Bewegung des NPCs im gewählten Ansatz. Damit treten weitere Limitierungen beim Explorieren der Welt auf und das Öffnen von Türen oder das Erreichen eines erhöhten Plateaus sind zum Beispiel nicht möglich. Funktion wie Springen, Kopfbewegungen bzgl. der X- ,Y- und Z-Achse oder Interaktionen mit der virtuellen Welt würde Abhilfe verschaffen. So könnten durch das Springen zum Beispiel Bereiche einer virtuellen Welt erreicht werden, die durch rein horizontale Bewegungen nicht exploriert werden können. Das

Interagieren mit der virtuellen Welt, zum Beispiel das Öffnen einer Tür oder das Aktivieren eines Schalters wären hilfreich. In diesem Fall müsste der NPC auch nicht, wie im vorliegenden Prototyp, in jeden abgeschlossenen Bereich separat gesetzt werden. Auch hierzu könnten die Methoden in der Arbeit von Gordillo et al. [GBTG21] gewinnbringenden Mehrwert leisten.

In den vorherigen Abschnitten wurden diverse Vorschläge zu Verbesserungen gemacht, die künftige Arbeiten aufgreifen und somit den Erfolg der Fehlerdetektion steigern könnten. In einem zweiten Schritt könnte geprüft werden, ob durch den dann verbesserten Prototyp auch weitere Fehlerarten entdeckt werden können. Die generelle Weiterentwicklung des Ansatzes, so dass nicht nur ein spezifischer Fehler erkannt wird, sondern andere Fehler desselben Typs, wäre eine Möglichkeit für zukünftige Forschung.

Literaturverzeichnis

[Ado21] Adobe. *Binspector: Evolving a security tool*. https://blog.adobe.com/en/topics/security.html. 2021 (zitiert auf S. 11).

[AG21] D. AG. *Brandschutzübungen bei der Daimler AG – Fire Sat 4.0*. https://www.youtube.com/watch?v=fooRJEx6myw. 2021 (zitiert auf S. 11).

[BB09] R. Brummayer, A. Biere. „Fuzzing and delta-debugging SMT solvers“. In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. 2009, S. 1–5 (zitiert auf S. 15).

[BBGM12] S. Bekrar, C. Bekrar, R. Groz, L. Mounier. „A Taint Based Approach for Smart Fuzzing“. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, S. 818–825. DOI: 10.1109/ICST.2012.182 (zitiert auf S. 15, 16).

[BCF+06] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, G. Vigna. „SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr“. In: *International conference on information security*. Springer. 2006, S. 343–358 (zitiert auf S. 15).

[BHC03] A. Bierbaum, P. Hartling, C. Cruz-Neira. „Automated Testing of Virtual Reality Application Interfaces“. In: *Proceedings of the Workshop on Virtual Environments 2003*. EGVE '03. Zurich, Switzerland: Association for Computing Machinery, 2003, S. 107–114. ISBN: 1581136862. DOI: 10.1145/769953.769966. URL: https://doi.org/10.1145/769953.769966 (zitiert auf S. 20).

[BT16] A. Breckel, M. Tichy. „Live programming with code portals“. In: *Workshop on Live Programming Systems-LIVE*. Bd. 16. 2016 (zitiert auf S. 19).

[CGZ+13] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, J. Regehr. „Taming compiler fuzzers“. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, S. 197–208 (zitiert auf S. 16).

[Cis21] Cisco. *Cisco secure development lifecycle*. https://www.cisco.com/c/en/us/about/trust-center/technology-built-in-security.html. 2021 (zitiert auf S. 11).

[Con] A. Consulting. „AltUnity Tester“. In: (Zitiert auf S. 20).

[CSW09] A. B. Craig, W. R. Sherman, J. D. Will. *Developing virtual reality applications: Foundations of effective design*. Morgan Kaufmann, 2009 (zitiert auf S. 13).

[DBJ+19] R. Dörner, W. Broll, B. Jung, P. Grimm, M. Göbel. „Einführung in virtual und augmented reality“. In: *Virtual und Augmented Reality (VR/AR)*. Springer, 2019, S. 1–42 (zitiert auf S. 13, 14, 24).

[Fra21] M. Frankfurt. *Ausstellung Museum Frankfurt*. https://museumfrankfurt.senckenberg.de/de/ausstellung/virtual-reality/. 2021 (zitiert auf S. 11).

[GBTG21] C. Gordillo, J. Bergdahl, K. Tollmar, L. Gisslén. „Improving Playtesting Coverage via Curiosity Driven Reinforcement Learning Agents“. In: *CoRR* abs/2103.13798 (2021). arXiv: 2103.13798. URL: https://arxiv.org/abs/2103.13798 (zitiert auf S. 48, 49).

[GFR+20] A. Gil, T. Figueira, E. Ribeiro, A. Costa, P. Quiroga. „Automated Test of VR Applications“. In: *International Conference on Human-Computer Interaction*. Springer. 2020, S. 145–149 (zitiert auf S. 20).

[Goo21] Google. „ClusterFuzz is a scalable fuzzing infrastructure that finds security and stability issues in software“. In: 2021. URL: https://github.com/google/clusterfuzz (zitiert auf S. 11).

[Jää16] E. Jääskelä. „Genetic algorithm in code coverage guided fuzz testing“. In: *University of Oulu* (2016) (zitiert auf S. 16).

[KCL11] H. C. Kim, Y. H. Choi, D. H. Lee. „Efficient file fuzz testing using automated analysis of binary file format“. In: *Journal of Systems Architecture* 57.3 (2011), S. 259–268 (zitiert auf S. 15).

[LLT20] S. M. Lehman, H. Ling, C. C. Tan. „ARCHIE: A User-Focused Framework for Testing Augmented Reality Applications in the Wild“. In: *2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. 2020, S. 903–912. DOI: 10.1109/VR46266.2020.00013 (zitiert auf S. 20).

[LPJ+18] H. Liang, X. Pei, X. Jia, W. Shen, J. Zhang. „Fuzzing: State of the Art“. In: *IEEE Transactions on Reliability* 67.3 (2018), S. 1199–1218. DOI: 10.1109/TR.2018.2834476 (zitiert auf S. 15–17).

[Luf21] Lufthansa. *Flugsimulation bei der Lufthansa – Lufthansa Aviation Training (LAT)*. https://www.3spin.com/de/vfr. 2021 (zitiert auf S. 11).

[LZZ18] J. Li, B. Zhao, C. Zhang. „Fuzzing: a survey“. In: *Cybersecurity* 1 (Dez. 2018). DOI: 10.1186/s42400-018-0002-y (zitiert auf S. 16).

[Med21] Medizin. *Traumatherapie - Ängste bekämpfen*. https://www.soldierstrong.org/strongmind/. 2021 (zitiert auf S. 11).

[Mic21] Microsoft. *Microsoft Security Development Lifecycle, verification phase*. https://www.microsoft.com/en-us/securityengineering/sdl/practices. 2021 (zitiert auf S. 11).

[NS18] M. Nebeling, M. Speicher. „The trouble with augmented reality/virtual reality authoring tools“. In: *2018 IEEE international symposium on mixed and augmented reality adjunct (ISMAR-Adjunct)*. IEEE. 2018, S. 333–337 (zitiert auf S. 14).

[Oeh05] P. Oehlert. „Violating assumptions with fuzzing“. In: *IEEE Security Privacy* 3.2 (2005), S. 58–62. DOI: 10.1109/MSP.2005.55 (zitiert auf S. 15).

[PGD+15] M. de Paiva Guimaraes, B. B. Gnecco, D. R. C. Dias, J. R. F. Brega, L. C. Trevelin. „Graphical high level analysis of communication in distributed virtual reality applications“. In: *Procedia Computer Science* 51 (2015), S. 1373–1382 (zitiert auf S. 19).

[Sol21] Soldamatic. *Schweißerausbildung mit AR – WeldPlus*. https://weldplus.de. 2021 (zitiert auf S. 11).

[Van05] I. Van Sprundel. „Fuzzing: Breaking software in an automated fashion“. In: *Decmember 8th* (2005) (zitiert auf S. 16).

[WBG06] C. Winterbottom, E. Blake, J. Gain. „Using Visualizations to Support Design and Debugging in Virtual Reality“. In: Bd. 4291. Nov. 2006. ISBN: 978-3-540-48628-2. DOI: 10.1007/11919476_47 (zitiert auf S. 19).

[WCGB13] M. Woo, S. K. Cha, S. Gottlieb, D. Brumley. „Scheduling black-box mutational fuzzing“. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, S. 511–522 (zitiert auf S. 15).

[WH00] J. S. Willans, M. D. Harrison. „A ‘plug and play’approach to testing virtual environment interaction techniques“. In: *Virtual Environments 2000*. Springer, 2000, S. 33–42 (zitiert auf S. 20).

[WYL18] H. Wei, X. Yin, P. Lin. *Novel Video Prediction for Large-scale Scene using Optical Flow*. 2018. arXiv: 1805.12243 [cs.CV] (zitiert auf S. 48).

[WZLY13] G. Wen, Y. Zhang, Q. Liu, D. Yang. „Fuzzing the actionscript virtual machine“. In: *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. 2013, S. 457–468 (zitiert auf S. 15).

[ZGB+21] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, C. Holler. *The Fuzzing Book*. Retrieved 2021-03-12 11:41:11+01:00. CISPA Helmholtz Center for Information Security, 2021. URL: https://www.fuzzingbook.org/ (zitiert auf S. 11).

Alle URLs wurden zuletzt am 04. 10. 2021 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift