Hochschule Düsseldorf
University of Applied Sciences

Fachbereich Medien
Faculty of Media

How to Design for Augmented and Virtual Reality

Entwicklung eines Stealth VR Games

Technische Dokumentation

Sommersemester 2021
Hochschule Düsseldorf

Projektleitung:
Jochen Feitsch und Charlotte Triebus

1. Einleitung

1.1 Projektbeschreibung

Spätestens seit dem Release von *Half Life: Alyx* durch *Valve* ist man sich in Entertainmentkreisen dem enormen Potenzial von VR bewusst. Aber auch in anderen Bereichen, wie Bildung und Medizin, gewinnen *Mixed Reality* Anwendungen immer mehr an Bedeutung. Somit gewinnt ein Verständnis für deren Design und immersionssteigernde Techniken ebenfalls an Relevanz.

Als praktischen Teil einer Projektarbeit zum Thema "*How to Design for Augmented and Virtual Reality"* an der Hochschule Düsseldorf haben wir ein Stealth VR Game entwickelt. Dabei lag der Fokus auf designrelevanten Aspekten wie *Affordance* und *Agency*, welche die Anwendererfahrung stark beeinflussen.
Auf die spezifische Umsetzung dieser wird in der Konzeptbeschreibung eingegangen.

1.2 Motivation

Heutzutage hat wohl schon so ziemlich jede Person Erfahrungen mit fragwürdigen Bedienungselementen gemacht, sei es im Online- und Gamingbereich oder bei dem Versuch noch schnell ein Ticket aus dem Automaten zu ziehen, bevor die Bahn abfährt. Unübersichtliche, verschachtelte Menüs führen zu Verzögerungen in der Bedienung und führen zu unnötigem Frust. Gerade VR und AR Anwendungen können einen mit zu kleinen Schaltflächen und einer mühseligen Auswahl per Laser Pointer in den Wahnsinn treiben.

Im praktischen Teil dieses Projektes hatten wir die Möglichkeit unsere neu gewonnen Kenntnisse anzuwenden, selbst in den Designprozess einzutauchen und uns mit den damit verbundenen Problemen auseinanderzusetzen.
Aufgrund unserer persönlichen Vorliebe für Gaming-Inhalte haben wir uns aus einer Vorauswahl an Anwendungsideen für die Umsetzung eines Stealth VR Games entschieden. Als Inspiration dienten uns vorwiegend Videospiele wie *Aragami* und die älteren Ableger der bekannten *Assassin's Creed* Reihe.

2. Umsetzung

2.1 Technologien

2.1.1 Hardware

Zur Entwicklung standen uns die VR Systeme der *Oculus Quest* und *Oculus Quest 2* zur Verfügung.

2.1.2 Software

Game Engine

Unity 2020.3.12f1 LTS

Relevante Packages	**Beschreibung**
Oculus XR Plugin 1.9.1	Bildschirm- und Eingabe-Unterstützung für Oculus Geräte
XR Interaction Toolkit 1.0.0-pre.5 (Preview)	Komponenten-basiertes Interaktionssystem für VR und AR
XR Plugin Management 4.0.6	Verwaltung von XR plug-ins
Animation Rigging 1.0.3	Unterstützung zur Erstellung von Animationen
Visual Studio Code Editor 1.2.3	Editor Integration für Visual Studio Code
Shadow Detect 2.0.0	Tool zur Erkennung von Schatten

2.2 Teilaufgaben

2.2.1 Versionsverwaltung und Entwicklungsumgebung

Um es zu ermöglichen, auf sämtliche Versionen und Änderungen zuzugreifen, und sicherzustellen, dass alle Beteiligten jederzeit Zugriff auf die aktuelle Version der Anwendung haben, entschieden wir uns dazu, ein GitLab Repository für unser Projekt anzulegen.

2.2.2 Oculus Integration

SDK Installationen

Im ersten Schritt werden Pakete, die relevant für die VR Entwicklung in Unity sind, installiert. Unter *Edit > Project Settings…* finden sich unter dem Reiter *XR Plugin Management* Einstellungen zum entsprechenden Paket, das sich, wenn nicht vorhanden über einen Button installieren lässt.

Nach der Installation kann unter diesen Einstellungen, Oculus als Entwicklungsplattform ausgewählt werden. Für die Installation eines weiteren relevanten Paketes aktiviert man zunächst unter dem Reiter *Package Manager > Advanced Settings* die Option *Enable Preview Packages* um das Paket *XR Interaction Toolkit 1.0.0-pre.5 (Preview)* unter *Window > Package Manager > Packages: Unity Registry* installieren zu können. Dazu werden auch zugehörige Samples des Pakets wie die *Default Input Actions* in das Projekt importiert.

Qualitätseinstellungen

Um die Qualität für VR Endgeräte zu verbessern werden unter *Edit > Project Settings… > Quality* folgende Einstellungen vorgenommen:

Pixel Light Count	1
Anisotropic Textures	Per Texture
Anti-Aliasing	4x Multi Sampling
Soft Particles	disabled

Als letzte Einstellung werden unter *Edit > Project Settings… > Preset Manager* Presets aus dem zuvor importierten *Sample*-Assets des *XR Interaction Toolkit 1.0.0-pre.5 (Preview)* Paketes hinzugefügt. Diese sich in der *Project*-Ansicht unter *Assets > Samples > XR Interaction Toolkit > 1.0.0-pre.5 > Default Input Actions*, befindenden fünf Presets *XRI Default Continuous Move*, *XRI Default Continuous Turn*, *XRI Default Left Controller*, *XRI Default Right Controller* und *XRI Default Snap Turn* können nach Auswahl im Inspector über einen Button hinzugefügt werden.

Zurück im *Edit > Project Settings… > Preset Manager* können nun bei den Presets für den *ActionBasedController* zur Unterscheidung des linken und rechten Controllers entsprechende Filter für *Links* und *Rechts* eingetragen werden.

XR Rig Set-up

Für die Repräsentation des Spielenden in der virtuellen Umgebung wird über *GameObject > XR* ein *XR Rig* erstellt. Dieses beinhaltet bereits eine stereoskopische Kamera und zwei Controller-Objekte die über das HMD und zugehörigen Controllern getrackt werden. Auf die gleiche Weise werden nun noch zusätzlich ein *Interaction Manager* und *Locomotion System* erzeugt. Beide Objekte lassen sich gut mit dem *XR Rig* zusammenfassen, so dass folgende Hierarchie entsteht:

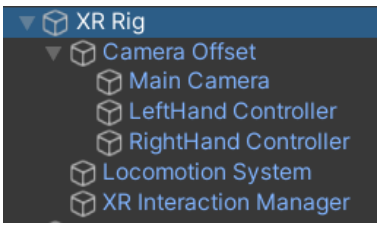

Abbildung: Hierarchie des XR Rigs

Auf die genaue Funktion und Implementierung des Locomotion Systems und der Controller wird im späteren Verlauf der Dokumentation eingegangen.

2.2.3 Full Body VR Rigging

Für die Umsetzung eines Ganzkörper Avatars für die virtueller Umgebung, wird zunächst ein geeignetes Modell benötigt. Wichtig hierbei ist, dass es sich um ein humanoid geriggtes Modell handelt, damit es sich auf den humanoiden Körper des Spielenden abbilden lassen kann.

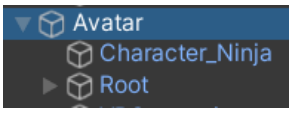

Abbildung: Hierarchie unseres Avatar Rigs

Im Root-Objekt unseres Avatar Rigs befinden sich die durch erfolgreiches Rigging importierten Objekte, die alle relevanten Körperteile einer humanoiden Figur in ihre Position im Welt- und Lokalen Koordinatensystem zu einer Hierarchie von *Transform*-Komponenten zusammenfasst und voneinander abhängig macht.

Kopf und Hände

Mit Hilfe des *Animation Rigging* Paketes, das sich über den Package Manager installieren lässt, lassen sich bestimmte Verbindungen des Kopfes und der Hände auf den Körper, sowie der Arme und Schultern des Avatar-Rigs auf entsprechende VR Objekte abbilden. Die Komponente *Multi-Parent Constraint* stellt hierbei eine Beschränkung auf den geriggten Kopf des Avatar her, während die Komponente *Two Bone IK Constraint* eine Beschränkung auf die Arme des Avatar legen. Diese Beschränkungen (Constraints) des Kopfes und der Arme des Avatars werden dann im nächsten Schritt über das Script *VRRig* auf Kopf und Arme des XR Rigs gemappt. Dieses Mapping bewirkt, dass sich Kopf und Arme des Avatars mit dem Kopf und Controllern des XR Rigs bewegen. Die Beschränkung (Constraint) auf dem Kopf des Avatars lässt dabei den Körper hals abwärts mitdrehen, wenn der Kopf der XR Rigs gedreht wird. Gleichzeitig wird über die IK (Inverse Kinematic) Beschränkung auf den Armen eine natürliche Bewegung von Gelenken prognostiziert, auch wenn nur Positions- und Rotationswerte der Hände über die Controller des XR Rigs verfügbar sind.

Abbildung: Hierarchie der VR Constraints

Beine

Damit Beine in VR simulierbar sind, auch wenn kein Ganzkörper-Tracking vorhanden ist, können diese - genau wie bereits die Arme - über IK, sowie entsprechenden Animationen bei Vor- und Rückwärtsbewegung, sowie der Drehung des Kopfes realisiert werden. Für die Animation der Beine verwenden wir humanoid geriggte Animationen von Mixamo. Die Animationen werden über den Animator gesteuert. Der Animator bietet die Möglichkeit für einzelne Animationen auf einem Objekt Ebenen (Layer) anzulegen. Für die Animation der Beine wird dazu die Ebene *Walk* erstellt und mit den entsprechenden Lauf-Animationen belegt. Da es sich bei den Animationen um Ganzkörper-Animationen handeln kann, die nicht nur auf die Beine beschränkt sind wird auf dieser Ebene eine Maske angewendet, so dass Laufanimationen über die Beine hinaus nicht auf dem Oberkörper des Avatars angewendet werden. Auf die Ebenen des Animators lässt sich auch IK anwenden, was dann natürlich gebeugte Knie zur Folge hat sobald der Kopf des Avatars sich nach unten bewegt. Somit kann der Körper des Avatars auch auf den Spielenden abbilden

Hand Bewegungen

Als Herausforderung stellte sich das Animieren der Hände zur Visualisierung der Interaktionen auf die *XR Rig* Controller dar, da auch hier wieder humanoid geriggte Animationen benötigt wurden. Als Lösung hat sich das Handy Poses Asset aus dem Unity Asset Store ergeben, das 20 humanoid geriggte Hand posen enthält. Diese werden über das Script *HandController* gesteuert und je nach verwendetem Input über die Controller abgespielt.

2.2.4 Interaktionsmethoden

Interactors

XR Direct Interactor

Die direkte Interaktion mit Gegenständen (sog. Interactibles) in der virtuellen Umgebung lässt sich über die *XR Direct Interactor*-Komponente in Kombination mit einem Trigger-Collider auf beiden Controllern des XR Rigs implementieren.

Als weiterer Interaktor wurden mehrere Instanzen der *XR Socket Interactor*-Komponente als Ausrüstungssysteme am Körper des Avatars verwendet.

Interactibles

XR Grab Interactable

Objekte die zur Interaktion mit den o.g. Interactors fähig sein sollen, benötigen die Komponente XR Grab Interactable, sowie einen *Collider* zur Kollisionserkennung und ein Rigidbody, so dass simulierte physikalische Gesetze auf das virtuelle Objekt wirken können.

Des Weiteren werden diese Objekte durch das Script *Interactable Item* erweitert. Diese Komponente implementiert die Verwaltung der Transform-Beziehung mit dem jeweiligen Interactor, sowie den derzeitigen *Physics Layer*, so dass die Collider der Interactibles während ihrer Interaktion mit dem Interaktor nicht dessen Collidern kollidieren.

Zusätzlich verwaltet diese Klasse noch das haptische Feedback, abhängig davon wann und mit welchem Interactor interagiert wird.

Interactable Items

Die Klasse *Interactable Item* dient als Superklasse für die Klasse *Weapon, die* weitere Eigenschaften wie Schaden als auch eine eigene Kollisionserkennung, die speziell auf Gegner und zerstörbare Objekte abzielt, implementiert.

Die Berechnung des Schadens einer Waffe (*Weapon*) auf einen Gegner (*Enemy*) berechnet sich aus der Höhe der Velocity Magnitude am Zeitpunkt der Kollision. Zunächst wurde auch ein *Cutter* Mechanismus für das längere Schwert (*Ninjato*) implementiert, mit dem sich durch Objekte schneiden lässt, dieser wurde aber wegen der Fehleranfälligkeit und der schlechten Limitierung auf das Spielgeschehen wieder verworfen.

Ein weiteres Feature, das in der *Interactable Items* Klasse implementiert wurde, sind Wurfwaffen. Eine Wurf-Interaktion, setzt voraus, dass man eine werfbare Waffe in der Hand hält. Erst dann wird über das Script *PinPointer* beim Formen einer Luftpistole (Grab Button gedrückt, Daumen über A/X) mit der freien Hand ein Raycast über den Zeigefinger gesendet. Dieser Raycast beschränkt sich auf die *Physics Layer* Masken *Enemy* und *Light*, da nur diese Objekte über Wurfwaffen angezielt werden können. Bei einer Erkennung würde sich der Raycast über die Komponente *Line Renderer* als rote Linie sichtbar machen. Die rote Linie des Raycasts ermöglicht es nun über das Drücken des Primary Buttons(A/X auf Oculus) der zielenden Hand einen HitPoint zu platzieren, visuell dargestellt als grüner Punkt. Nach der Markierung des Enemy/Light Objektes, kann die Luftpistole wieder fallengelassen werden. Der Grüne Punkt bleibt bestehen, und ein Wurf mit der Wurfwaffe kann ausgeführt werden. Die Waffe wird dabei automatisch auf den *HitPoint* geworfen, sobald die *acceleration magnitude* des Wurfes mindestens 30 übersteigt.

Interaktion mit Gegner (*Enemy*) / Schadensberechnung

Bei der Kollision einer Waffe (*Weapon*) mit dem Collider eines Gegners (*Enemy*) wird Schaden in Abhängigkeit von der Klasse *WeakPoint* hinzugefügt. WeakPoints werden zusammen mit Collidern auf bestimmten Gelenken des geriggten Gegner-Modells implementiert. Je nach Schwachstelle ist eine WeakPoint Instanz anders konfiguriert, bspw. für eine Kollision der Waffe dem Halsbereich des Gegners mehr Schaden zu, als eine Kollision an den Armen oder Beinen. Entstehender Schaden wird dann von der getroffenen WeakPoint Klasse an das Enemy Script weitergeleitet, wo dann die Schadensberechnung durchgeführt wird. Der Gegner besitzt Standardmäßig eine Höhe von 100 Lebenspunkten (*health points*) und Waffentypen unterscheiden sich in der Höhe ihres individuellen Waffenschadens. Die Formel bei der Schadensberechnung wird davon Abhängig gemacht, ob der Schaden von einer normalen oder einer Wurfwaffe verursacht wurde.

- Schaden durch normale Waffe:
  - Waffenschaden x velocity magnitude x Schwachstellen Multiplikator
- Schaden durch Wurfwaffe:
  - Waffenschaden x Schwachstellen Multiplikator

Der Hintergrund dazu ist, dass für Wurfwaffen eine automatische Wurfmechanik entwickelt wurde, die kaum oder gar keine *velocity magnitude* besitzt, da die Rigidbody Komponente während des automatischen Wurfprozesses ausgeschaltet wird.

Mit Wurfwaffen besteht auch die Möglichkeit Laternen zu zerstören, dafür wurde das LightController Script als so angepasst, dass die Lichtquelle bei einer Kollision mit einer Waffe zerstört wird.

UI Interaktion

Bei der Entwicklung einer Interaktionsmöglichkeit mit UI Objekten ergab sich folgendes Problem: Die Interaktion mit UI über das XR Interaction Toolkit benötigt als Interaktor die *XR Ray Interactor* Komponente, die man jedoch nicht zusammen Nutzen kann mit dem *XR Direct Interactor* nutzen kann, da beide von derselben Klasse erben. Deswegen mussten alternative Wege zur Interaktion mit UI gefunden werden. Eine mögliche Lösung wäre es, einen XR Direct Interactor auf dem einen und einen XR Ray Interactor auf der anderen Hand zu implementieren. Eine andere Lösung wäre die Kopie und Abänderung der XR Ray Interactor Klasse, weil die enthaltene UI Interaktions Implementierung nicht abhängig ist von der Basisklasse.

Die Entscheidung fiel auf die letztere Lösung, damit beide Controller weiterhin direkt mit der virtuellen Umgebung interagieren können.

2.2.5 Locomotion

Continuous movement

Eine kontinuierliche Fortbewegung wurde mit Hilfe des Locomotion Systems aus dem XR Interaction Toolkit umgesetzt. Die entsprechende Komponente *Continuous Move Provider* wurde dabei auf der Standardgeschwindigkeit belassen, da diese relativ langsam ist und somit eine Motion Sickness vermindern kann.

Snap turn movement

Um nicht auf Kopf- oder Körperdrehungen in RL angewiesen zu sein besteht die Möglichkeit einer Drehung über den Input des Controllers durch die *Snap Turn Provider* Komponente. Auch diese Locomotion wurde mit 45 Grad möglichst gering gehalten um eine Motion Sickness zu vermindern.

Teleportation

Nach der Konzeption war es geplant eine Teleportation zu implementieren, zum einen um der Heldenfigur die gewisse Superkraft zu verleihen durch Schatten springen zu können, zum Anderen um weniger Distanz über die *Continuous movement* Locomotion zurücklegen zu müssen. Aber auch hier stellten sich die beiden inkompatiblen Interaktoren als Problem dar. Anders bei der UI Interaktion, war die Teleportation jedoch von der Interaktoren Basisklasse abhängig, so dass eine andere Möglichkeit der Teleportationsimplementation recherchiert werden musste und entwickelt werden musste, für deren Umsetzung am Ende nicht genug Zeit blieb.

2.2.6 Stealth Mechanics

Antagonisten in der Spielwelt können die Spieler*innen wahrnehmen, wenn sich diese innerhalb ihres Sichtfeldes und nicht in Deckung befindet. Das Sichtfeld wird dabei jeweils durch den Strahlungswinkel und die Reichweite eines *Spotlights* definiert. Die Verdeckung durch Umgebungsobjekte, und damit das Verstecken hinter diesen, wird dabei durch den Einsatz einer *Layer Mask* und einer Sichtlinienüberprüfung via *Linecast* in der *Guard* Klasse ermöglicht.

Darüber hinaus sind die Spieler*innen schwerer zu erkennen, wenn sich diese in einem Schatten aufhalten. Dazu wird die Sichtweite alle Gegner reduziert, falls festgestellt wird, dass sich die Spielfigur in einem von einer Schatten befindet. Diese Feststellung erfolgt über das zuvor aufgelistete *Shadow Detect* Plugin mithilfe von *Raycasts,* welche von allen Lichtquellen ausgehend zu Zielen an Kopf und Füßen des Protagonisten erfolgen. Sollte ein Ziel dabei verdeckt sein, liegt es im Schatten (Schatten != Dunkelheit).
Somit gelten Anwender*innen als im Schatten befindlich, solange sich deren Füße und Kopf gleichzeitig in einem solchen befinden.

2.2.7 Gegnerverhalten

Wachen sind stationär oder patrouillieren eine vorbestimmte, durch Wegpunkten festgelegte Route. Von dieser weichen sie nur ab, wenn sie den Protagonisten, wie zuvor beschrieben, wahrnehmen oder auf ein Geräusch, durch das Abspielen von *Audio Sources* bei einer Kollision der Spielfigur mit bestimmten Objekten, aufmerksam werden.
In einem solchen Fall wird entweder die Position der Spieler*inen oder die der Geräuschquelle als nächster Wegpunkt in der *Coroutine FollowPath* innerhalb des *Guard Scripts* gesetzt. Dies ermöglicht zugleich, mittels wiederholter Wegpunktupdates, eine Verfolgung der Anwender*innen durch deren Widersacher.

2.2.8 Dialoge/UI

Es werden verschiedene *Canvases* mit Text- und Button-Elementen verwendet. Die Canvases sind auf “World Space” eingestellt. Dies ermöglicht, dass die Größe der Canvases festgelegt werden kann und dass diese sich an einer festen Position in der Spielwelt befindet.
Wenn Spieler*innen einen Button anklicken, wird das Skript *Dialog Manager* genutzt, welches die Spieler*innen zu einer weiteren Canvas wechseln lässt, die einen anderen Dialog enthält. So können sich Spieler*innen durch die Dialoge fortbewegen.
Außerdem haben diese bei den Dialogen auch mehrere Optionen zur Auswahl. Je nachdem welche Option die Spieler*innen ausgewählt haben, beeinflussen sie, welcher Dialog ihnen angezeigt wird.

Für den Game Manager ist ein Hauptmenü implementiert worden, bei dem das Skript *Menu Manager* ausgeführt wird, mit welchem die Spieler*innen das Spiel starten, unter “Settings” die Lautstärke des Spiels mit Hilfe eines Sliders des Toggle-Buttons ändern und mit dem Exit-Button das Spiel verlassen können.
Wenn die Spieler*innen das Spiel gestartet haben, wird eine Introsequenz eingeleitet.

2.2.9 Intro-Sequenz

Die Intro-Sequenz dient dazu, die Spieler*innen in die Story und die Welt des Stealth Games hinein zu führen.
Dafür werden unterschiedliche Szenen gezeigt, die teilweise auch einen zeitlichen Prozess darstellen und Kontraste schaffen sollen, wie beispielsweise bei den Dorfszenen. Um die Gegensätze stärker hervorzuheben, wird auch mit Farbe “gespielt”.
Hierbei werden kräftige und warme Farben für eine fröhliche Atmosphäre gegenüber trüben, dunklen und kalten Farben für eine düstere Atmosphäre genutzt. Die Endszene des Intros ist dieselbe wie die, die in der Polygon Welt verwendet wird, sodass die Spieler*innen mit dem/der Held*in zusammen aufwachen.
Als Software wurde die Zeichen-App Procreate verwendet, mit der auch Zeitraffern beziehungsweise Animationen in Form eines Daumenkinos erstellt werden können.

Für das Editing der Bilder wurde die Anwendung Premiere Pro genutzt. Die einzelnen Bilder wurden zu einer Slideshow mit Transitions und weiteren visuellen Effekten zusammen-geschnitten und mit verschiedenen Voice Lines versehen.
Das Intro wird in Unity über einen Video Player als Skybox in 180° Panoramaansicht abgespielt.