Hochschule Karlsruhe
University of Applied Science

Fakultät für Informatik und Wirtschaftsinformatik
Wirtschaftsinformatik

BACHELORTHESIS

Netcode in Unity

Von	Alexander Seitz
Matrikelnr.	77642
Arbeitsplatz	Hochschule Karlsruhe
Erstbetreuer	Prof. Dr. Udo Müller
Zweitbetreuer	Prof. Dr. Jan Stöß
Abgabetermin	31.08.2025

Karslruhe, DATUM

Netcode in Unity

Alexander Seitz

30. Juli 2025

Inhaltsverzeichnis

Kapitel 1

Grundlagen

Die Grundlagen sollen dem gründlichen Verständnis komplexerer Thematiken helfen, die im Laufe dieser Arbeit aufgegriffen werden. Es bezieht sich hierbei auf fundamentale Aspekte betreffend:

- Unity spezifische Terminologie
- Grundlagen in Spieldesign
- Netcode spezifische Begriffe und Konzepte
- Mathematische Konzepte

Viele der Begrifflichkeiten existieren mehrfach und werden somit voneinander getrennt und dessen Unterschiede betont.

Kapitel 2

Einleitung

Moderne Echtzeitanwendungen, insbesondere im Bereich der Computerspiele, stellen hohe Anforderungen an die zugrunde liegende Netzwerkarchitektur. Die Synchronisation von Spielzuständen, die Minimierung von Latenzzeiten sowie ein robuster Umgang mit Paketverlust und Jitter sind entscheidend für eine positive Nutzererfahrung. Multiplayer-Systeme, wie sie etwa mit Unity entwickelt werden, müssen deshalb nicht nur funktional, sondern auch leistungsfähig und fehlertolerant sein.

Die vorliegende Arbeit beschäftigt sich mit den theoretischen und technischen Grundlagen der Netzwerkkommunikation in Mehrspielerumgebungen. Anhand eines prototypischen Projekts in Unity werden exemplarisch zentrale Mechanismen wie Reconciliation, Interpolation sowie die Kommunikation im Client-Server-Modell umgesetzt und analysiert.

Im Folgenden werden zunächst die Motivation und Zielsetzung der Arbeit erläutert. Anschließend wird der strukturelle Aufbau der Arbeit vorgestellt.

2.1 Motivation

Multiplayer-Komponenten sind aus modernen digitalen Spielen kaum mehr wegzudenken. Sowohl kompetitive als auch kooperative Mehrspieler-Modi sind in einer Vielzahl von Genres – von taktischen First-Person-Shootern über Survival-Games bis hin zu Rennspielen – zum Standard geworden. Mit dieser Entwicklung steigen auch die Anforderungen an die zugrunde liegenden Netcode-Systeme, insbesondere hinsichtlich Präzision, Konsistenz und Latenzkompensation.

Während große Entwicklerstudios in der Lage sind, umfangreiche Netzwerkinfrastrukturen zu realisieren, stellt der zuverlässige und performante Betrieb eines Netcodes für kleinere oder unabhängige Entwicklerstudios eine erhebliche Herausforderung dar. Die Komplexität entsprechender Systeme führt häufig dazu, dass Mehrspielerfunktionen nur rudimentär implementiert oder zugunsten anderer Spielmechaniken zurückgestellt werden.

Auch bei großbudgetierten Produktionen bleibt die Qualität der Netzwerkmechanismen regelmäßig Gegenstand öffentlicher Kritik. Spielende bemängeln unter anderem sogenannte *Desynchronisationen* (Desyncs), bei denen Spielerinteraktionen nicht konsistent übermittelt werden – etwa in Form von „Kill Trades“ oder dem sprichwörtlichen „Sterben um die Ecke“. Solche Phänomene sind Ausdruck komplexer zeitlicher und logischer Inkonsistenzen in der Server-Client-Kommunikation.

Ein aktuelles Beispiel für einen alternativen technologischen Ansatz ist die Einführung eines sogenannten Subtick-Systems im Titel *Counter-Strike 2*. Anstelle fester Tickintervalle sollen Eingaben auf Basis präziser Zeitstempel verarbeitet werden. Dieses Konzept verspricht eine höhere Eingabegenauigkeit und ein konsistenteres Spielerlebnis, wirft aber gleichzeitig Fragen zur tatsächlichen Wirksamkeit und Praxistauglichkeit solcher Systeme auf.

Die hohe technische Komplexität sowie die Vielzahl konkurrierender Anforderungen machen deutlich, dass es sich bei Netcode-Systemen nicht um triviale Umsetzungen handelt. Vielmehr stellt sich die Frage, inwiefern es überhaupt möglich ist, eine universell optimale Lösung zu entwickeln – oder ob es sich um ein inhärent trade-off-basiertes System handelt, in dem stets zwischen Genauigkeit, Latenz, Stabilität und Ressourcenverbrauch abgewogen werden muss.

2.2 Zielsetzung

In dieser Arbeit sollen Netcode und darauf aufbauende Netcode Mechanismen auf theoretischer und praktischer Grundlage beleuchtet werden und in einem prototypischen Spiel integriert werden. Da die reine Umsetzung keinen großen akademischen Mehrwert bietet, sollen Strategien entworfen werden, die zur Visualisierung und damit der Verbesserung des Verständnisses der Mechaniken dienen. Es sollen des weiteren verschiedene Szenarien untersucht werden, die auch von Idealbedingungen abweichen können.

2.3 Redraft

Multiplayer-Funktionalität ist in modernen Videospielen nahezu allgegenwärtig – sei es in Form kompetitiver Mehrspieler-Modi, kooperativer Spielvarianten oder persistenter Online-Welten. Titel aus unterschiedlichsten Genres – darunter strategische First-Person-Shooter, Survival-Spiele oder Rennsimulationen – stellen hohe Anforderungen an die Netzwerksynchronisation zwischen den Spielteilnehmern. Im Zentrum dieser Anforderungen steht ein performantes und präzises Netcode-System, das eine konsistente Spielerfahrung gewährleistet.

Die technische Umsetzung eines solchen Systems bringt erhebliche Herausforderungen mit sich. Besonders kleinere Entwicklerstudios und Indie-Teams sehen sich oftmals gezwungen, auf Mehrspieler-Komponenten gänzlich zu verzichten oder diese stark zu reduzieren, da die Implementierung eines zuverlässigen Netcodes umfassende Kenntnisse in Bereichen wie Netzwerktechnologien, Latenzkompensation und Serverarchitektur erfordert.

Auch bei großen Produktionen (AAA-Titeln) ist die Kritik an unzureichender Netcode-Qualität regelmäßig Gegenstand von Diskussionen in der Fachpresse und Community. Häufig wird dabei der Vorwurf laut, dass zur Kostenreduktion auf leistungsschwächere Infrastruktur – beispielsweise Server mit niedriger Tickrate – zurückgegriffen wird. Die daraus resultierenden Probleme wie asynchrone Spielerinteraktionen, sogenannte *Desynchronisationen* (Desyncs), oder Phänomene wie das „Sterben um Ecken“ und *Kill Trades* wirken sich spürbar negativ auf das Spielerlebnis aus.

Ein prominentes Beispiel für die aktuelle Weiterentwicklung von Netcode-Systemen bietet *Counter-Strike 2*, der Nachfolger von *CS:GO*. Anstelle einer Modernisierung des klassischen Tick-basierten Modells wurde ein Subtick-System eingeführt, das Eingaben nicht mehr an feste Tickintervalle bindet, sondern mit präzisem Zeitstempel verarbeitet. Obwohl die Grundidee eines solchen Systems nicht neu ist, stellt die konsequente Umsetzung in einem Mainstream-Titel einen signifikanten technologischen Schritt dar. Ziel dieser Architektur ist eine verbesserte Präzision der Eingabeverarbeitung sowie eine höhere Synchronität zwischen Server und Client.

Angesichts dieser Entwicklungen stellt sich die Frage, ob die oftmals kritisierte Qualität des Netcodes primär auf Entwicklungsentscheidungen zurückzuführen ist – oder ob es sich vielmehr um ein inhärent komplexes Problemfeld handelt, das durch eine Vielzahl

technischer, ökonomischer und gestalterischer Faktoren beeinflusst wird und kaum in seiner Gesamtheit zu „perfektionieren“ ist.

Kapitel 3

Auswahl der Engine

Um die bestmöglichste und vor allem effizienteste Entwicklung des Spiels und der geplanten Netcode Mechanismen zu integrieren, ist also eine durchdachte Wahl der Spiel Engine wichtig. Da vor allem Zeit ein Engpass in einem ambitionierten Projekt, wäre es fatal, wenn auf Grund der Wahl der Engine kein Fortschritt gemacht werden kann. Die wichtigsten Kriterien werden im Folgenden aufgelistet.

3.1 Lernkurve

Eine der wichtigeren Aspekte bezieht sich auf die Lernkurve der Game Engine für den Entwickler. Wenn also eine Engine mit so viel Lernaufwand in Verbindung steht um überhaupt eine triviale Szene zum Laufen zu bekommen, dann ist diese wohlmöglich nicht die richtige Engine. Im Anbetracht dieser Arbeit ist Zeit die wichtigste Ressource und darf somit nicht unterschätzt werden. Wenn somit Zeit gespart werden kann, indem man eine einsteigerfreundliche Engine lernen kann, ist das für die anstehende Entwicklungsarbeit ein Erfolg . Wenn jedoch gewisse Vorkenntnisse im Bereich Spieleentwicklung bestehen sollten, dann wird das Erlernen einer weiteren Engine, was mit viel duplikativem Wissen im Zusammenspiel steht, für wesentlich geringern Lernaufwand sorgen. Unter dieser Bedingun ist der Lernaufwand wiederum keine

3.2 Community und Ressourcen

Die Zugänglichkeit von Dokumentation, Foren und Tutorials ist in diesem Kontext extrem wertvoll, da vor allem die Kombination daraus ein tiefes Verständnis einbringen kann. Wenn also die Dokumentation nicht verständlich sein sollte oder es nicht genug Anwendungsbeispiele im Internet geben sollte, kann es den Lernaufwand zusätzlich erhöhen oder zu Missverständnissen kommen.

3.3 Netcode-Fähigkeiten

Wenn also öut of the box"Features gut integriert und dokumentiert sind, die die Weiterentwicklung geplanter Netcode-Mechaniken erleichtern, ist dies der Idealzustand. Dennoch ist für diese Arbeit eine Abgrenzung zu machen, da die Eigenimplementierung im Mittelpunkt steht. Wenn also Mechanismen von der Engine bereits zum Großteil gestellt sind und der Quellcode hiervon nicht anpassbar ist, wirkt sich dies eher negativ auf die Bewertung aus.

3.4 Performance

Wie gut das Spiel im Einsatz von mehreren Clients läuft und wie viele Einbuße in diesem Fall der Multiplayer-Aspekt mit sich bringt. Sollte dies jedoch garnicht mehr möglich sein, dass ein Editor und ein oder mehrere Clients verbunden sind - was im Grunde das Laufen von drei Spielinstanzen darstellt - nicht funktionieren sollte, weil keine Ressourcen mehr auf dem Computer übrig bleiben, ist dies wohl keine geeignete Engine oder man muss optimieren. Allerdings kann eine graphische Grundvoraussetzung zum Problem werden. Wenn also die Engine durch die Natur der Renderpipeline zu sehr die GPU belastet, kann das schon das Prototyping beeinflussen, da in diesem Use Case eine hohe Framerate bevorzug ist.

3.5 Kosten und Lizenzmodell

Viele der bekannten Engines sind nicht Open Source sondern erwarten i.d.R. eine jährliche Zahlung oder eine Art Pacht . In manchen Fällen ist eine jährliche Hochschullizenz möglich, dennoch fällt das somit immernoch unter die Kategrie der Zahlungspflicht. Wenn

eine Engine somit einen hohen monetären Einsatz erfordert ohne der Möglichkeit auch nur auf eine bspw. Testversion zuzugreifen, müssen die anderen Aspekte sehr viel gutmachen.

3.6 Programmiersprache

In vielen Fällen steht für den Entwickler ein hoher Lernaufwand mit einer noch zu erlernenden Programmiersprache, bei der neue Aspekte in den Mittelpunkt gerückt werden, die u.U. noch nicht zuvor behandelt wurden. In den meisten Fällen sind dies eher hardwarenähere Programmiersprachen wie bspw. C++. Hierbei müssen neue Konzepte wie der Umgang mit Memory erlernt und gemeistert werden, was im Umgang mit neueren Programmiersprachen wie C# schon bereits gestellt ist. Im Grunde stellt das nicht nur ein Risiko dar, sondern ist auf der anderen Seite auch eine Chance zu lernen, wie man richtig Spiele entwickelt und vor allem auch Quellcode optimieren kann. Nicht ohne Grund bauen die meisten Spiele-Engines auf C++ auf und werden in der Zukunft auch eher nicht ersetzt.

3.7 Marktrelevanz

Die Marktrelevanz ist wie in Programmiersprache schon angesprochen, stark abhängig von der in der Engine genutzten Programmiersprache. Abgesehen davon gibt es unterschiedliche Einsatzbereiche verschiedener Engines und somit lässt sich die Marktrelevanz nicht nach absoluten Zahlen der Nutzter und oder bestimmter Spiele klären. Dennoch muss man die Relevanz der Engine im Auge behalten, wenn es also kaum Spiele auf dem Markt gibt, die mit einer bestimmten Engine entwickelt wurden, dann ist die potenziell investierte Zeit in einer beliebteren Engine wahrscheinlich angebrachter.

Ein direkter Vergleich der bekanntesten Engines zeigt folgende Eigenschaften:

Tabelle 3.1: Vergleich populärer Spiel-Engines mit Gewichtung

Kriterium	**Gewichtung**	**Unity**	**Unreal Engine**	**Godot**
Programmiersprache	3	C#	C++ / Blueprints	GDScript / C#
Lernkurve	5	Mittel	Hoch	Niedrig
Community	4	Sehr groß	Groß	Wächst schnell
Netcode	5	NGO, FishNet u.a.	Eigenes Replication-System	Drittanbieter, rudimentär
Performance	4	Gut (abhängig von Optimierung)	Sehr hoch	Mittel
Lizenz / Kosten	5	Kostenlos (eingeschränkt), Runtime Fee seit 2024 [1]	Royalty-basiert [2]	Open Source [3]
Verbreitung in Industrie	3	Sehr hoch	Hoch	Gering
Gesamtbewertung	–	**–**	**–**	**–**

Die Quantifizierung wurde auf Grundlage einer Skala von 1-5 bewerkstelligt, wobei 5 das Maximum ist und somit am besten. Diese Werte dienen als Multiplikatoren mit den Werten (ebenfalls 1-5) der einzelnen Spalten. Beide Skalen sind subjektiv und sind somit auch stark von Vorkenntnissen abhängig. Die einzelnen Spalten jedoch sind keine Gewichtungen sondern eher die tatsächlichen Werte, die vergeben wurden.

Basierend auf den genannten Faktoren fiel die Wahl auf **Unity**, da diese Engine eine ausgewogene Kombination aus Zugänglichkeit, Flexibilität und Netcode-Erweiterbarkeit bietet. Besonders die Integration von C# als Programmiersprache, die umfangreiche Dokumentation sowie die breite Community-Unterstützung waren ausschlaggebend. Demnach ist Unity vor allem auch am einsteigerfreundlichsten, was im Endeffekt Komplexität reduziert.

Unreal Engine hingegen spielt vor allem in der Entwicklung moderner, grafisch anspruchsvoller Spiele eine zentrale Rolle. Während Unity und Godot insbesondere im Indie-Bereich weit verbreitet sind, wird Unreal Engine regelmäßig für sogenannte AAA-Titel eingesetzt – etwa bei *Fortnite*, *Valorant* oder der *Gears of War*-Reihe. In Addition kommt noch eines der am meist antizipiertesten Spiele überhaupt dazu nämlich The Witcher 4 [4].

Ein besonderes Beispiel für die Weiterentwicklung der Engine durch Eigengebrauch ist das Prinzip des „Dogfooding“. Epic Games nutzt die Unreal Engine intern intensiv zur Entwicklung eigener Spiele wie *Fortnite*, wodurch neue Features – etwa das Partikelsystem *Niagara* – direkt unter realen Produktionsbedingungen erprobt und optimiert werden [5].

Insbesondere mit der Einführung von Unreal Engine 5 (UE5) hat sich die Engine als Standardlösung für kleinere bis mittlere Studios etabliert, die auf fotorealistische Darstellung und moderne Grafiktechnologien wie *Lumen* oder *Path Tracing* setzen. Unity kann in diesen Bereichen zwar nicht vollständig mithalten, bietet jedoch im Kontext dieser Arbeit – etwa bei der prototypischen Umsetzung netzwerkbasierter Mechaniken – eine effizientere und zugänglichere Plattform. Beispiele wie *BattleBit Remastered* (2023) jedoch zeigen, dass auch technisch reduzierte, aber netzwerkseitig ausgereifte Multiplayer-Titel mit Unity erfolgreich realisiert werden können.

3.8 Eigenimplementierung und Auswahl des Frameworks

Im Bereich Netcode existieren für Unity verschiedene Frameworks. Zunächst wurde Unitys offizielles **Netcode for GameObjects (NGO)** in Betracht gezogen. NGO ist mit Netcode for Entities der Standard und das als eigenes Paket, welches man über den Package Manager installieren kann. In der praktischen Erprobung zeigten sich jedoch Schwächen im zugrunde liegenden *Tickmodell*. NGO nutzt ein festes Zeitraster zur Synchronisation zwischen Server und Clients, wobei alle Netzwerkaktionen strikt an die sogenannte *NetworkTick*-Rate gebunden sind.

Dieses Modell ist zwar grundsätzlich stabil, führt jedoch bei Spielen mit hoher Eingabefrequenz oder schnellen Bewegungsabläufen zu Einschränkungen: Eingaben, die zwischen zwei Ticks liegen, werden verzögert verarbeitet, was sich in Form von spürbarer Latenz oder schwankender Reaktionsgeschwindigkeit äußern kann. Besonders bei deterministisch kritischen Anwendungen erschwert dies eine präzise Umsetzung von Mechaniken wie Client-Side Prediction oder Server Reconciliation. Hierfür müsste ein Ticksystem selbst implementiert werden, was je nach Architektur zu Kollisionen führen kann.

Als Alternative wurde das Framework **FishNet** in Betracht gezogen, das in der offiziellen Dokumentation ausdrücklich für Anwendungsfälle mit hohen Anforderungen an Präzision und deterministische Logik empfohlen wird [6]. FishNet nutzt ein eigenes Tick-basiertes Netzwerkmodell und bietet unter anderem:

- Vollständige Kontrolle über Netzwerklogik
- Support für Host-Modus, Dedicated Server und Peer-to-Peer
- Integration mit Prediction, Interpolation und Authentifizierung

Trotz dieser technischen Vorteile wurde für diese Arbeit letztlich **NGO verwendet**. Hauptgrund war zum einen der geringere Integrationsaufwand und die direkte Unterstützung durch Unity. Eine vollständige Migration auf FishNet hätte umfangreiche Anpassungen erfordert, die im gegebenen Zeitrahmen nicht realisierbar gewesen wären. Fishnet nutzt somit oft auch eine andere Syntax, die nicht so weit verbreitet ist wie der Standard von Unity's NGO. Dies macht die Fehlerbehebung komplizierter und führt vor allem bei Netcode-Unerfahrenen zu Komplikationen. In der Dokumentation ist ebenfalls nichts zur Portierung von NGO auf Fishnet enthalten, was die Umstellung deutlich aufwändiger und zeitlich bedingt riskanter machen würde. Hierfür wäre sogar ein kompletter Umbau nötig und selbst damit könnte Erfolg nicht garantiert sein. Abgesehen davon, ist Fishnet sehr reich an Features, die über die Mindestanforderungen hinausgehen. In jedem anderen Use Case wäre Fishnet wohl die beste Entscheidung, weil es die sinnvollste und beste Vorablösung bereitstellt, dessen Umbau jedoch in keinster Weise trivial ist.

Zum anderen liegt der Schwerpunkt dieser Arbeit explizit auf dem **Verständnis und der eigenständigen Implementierung** zentraler Netcode-Mechaniken wie *Client-Side Prediction*, *Server Reconciliation* und *Interpolation*. Statt auf bereits vollständig implementierte Framework-Funktionalitäten zurückzugreifen, wird ein eigenes, leichtgewichtiges System entwickelt, um die Funktionsweise dieser Konzepte praktisch und nachvollziehbar umzusetzen.

NGO wird hierbei vor allem als *Basisschicht* für die Netzwerkkommunikation genutzt, während zentrale Mechanismen unabhängig davon realisiert werden. Dieser Ansatz erlaubt eine tiefere Auseinandersetzung mit den zugrunde liegenden Prinzipien und Herausforderungen im Netcode-Design.

Kapitel 4

Beschreibung des Unity Prototypen

Der entwickelte Prototyp bildet eine einfache First-Person-Shooter-Sandbox ab und dient als Testumgebung für die Untersuchung netzwerkbezogener Mechaniken wie Interpolation, Client-Side Prediction mit Reconciliation und Lag Compensation. Der Prototyp wird auch zum Abschluss der Arbeit bewusst schlicht gehalten, da der Fokus auf der Netzwerkschicht liegt – nicht auf grafischen oder animationsbasierten Aspekten. Texturen, Beleuchtung und Animationen wurden daher nur in minimalem Umfang berücksichtigt. Im Verlauf der Arbeit existieren prinzipiell zwei Versionen, bei denen es sich zum einen um die Grundstruktur des Spiels handelt, was demnach den Offline-Prototypen darstellt. Diese Version besteht aus fundamentalen aber auch sehr schlicht gehaltenen Bestandteilen für ein FPS-Spiel. Diese bestehen aus einem Character Controller, mit dem man springen und laufen kann und einer Waffe in First-Person-Ansicht, die über eine Schuss- und Nachlademechanik verfügt. Ein Schadensmodell, das als Metadaten den einzelnen Waffen zugewiesen werden kann und das alles auf einer minimalistischen Prototyp-Map. Auf diesem Prototypen anknüpfend, sollen und wurden besagte Netcode-Mechaniken inklusive Synchronisation integriert werden.

4.1 Szenario und Spiellogik

Die Umgebung erinnert an Trainingsszenarien aus Spielen wie *Valorant - The Range* [7] oder Tools wie *Aim Lab* [8]. Der Spieler wird durch einen FPS-Controller dargestellt, der als `Prefab` implementiert ist. In der Szene bewegen sich ein oder mehrere Zielobjekte (Targets), die sich zufällig und unvorhersehbar über das Spielfeld bewegen. Diese Targets können wiederholt erscheinen (respawnable) und dienen im späteren Verlauf zur Untersuchung netzwerkbedingter Bewegungsartefakte. Zusätzlich können sich mehrere Clients auf den Server verbinden und sollen synchronisiert werden. Hierbei geht es darum Inputlatenz auch bei hohem Ping minimal zu halten und ein flüssiges und direktes Spielgefühl zu erlauben. Dies ist der Kern von modernen Netcodeverfahren und somit auch die Kernproblematik, da man hier den Kompromiss wählt, den Client in seiner Eingabe zu priorisieren und somit alles was dadurch an Nebenwirkungen entsteht im Nachhinein glätten zu müssen.

Ein Game Loop im klassischen Sinne existiert nicht, es ist also kein Spiel das einen Abschluss findet. Der Kern ist die Visualisierung der Netcode-Mechaniken und macht einen Game Loop also den strukturierten Ablauf eines Spiels hinfällig. Das Spiel kann gestartet werden indem ein Build über den Unity Editor gestartet wird und es kann unter anderem über die Konsole mit dem Befehl: disconnect verlassen werden.

4.2 Waffenmechanik und Spielgefühl

Für das Waffenmodell wurde ein Asset aus dem Unity Asset Store verwendet[1], da die Eigenmodellierung in Blender aus zeitlichen Gründen verworfen wurde. Dennoch wurden grundlegende FPS-typische Elemente integriert:

- Rückstoß (Recoil)
- Bewegungsbasiertes Waffenbobbing und Sway
- Mündungsfeuer (Muzzle Flash)
- Einschusslöcher (Decals)
- Treffermarkierung (visuell und auditiv)

[1] Platzhalter: *Name des Assets einfügen*

Diese Features verbessern nicht nur das Spielgefühl, sondern sind auch eine wichtige Grundlage für die spätere visuelle Analyse der Netzwerkmechaniken. Die Netzwerksynchronisation solcher Effekte gilt als nicht trivial und ist auch in professionellen Produktionen fehleranfällig und wird demnach ebenfalls ausgelassen. Spielerobjekte werden in Unity meistens über ein eigenes Network-Transform synchronisiert, welches in der Regel nur Für die Position des jeweiligen Transforms zuständig ist. Wenn also Einschusslöcher synchronisiert werden, müssen Animationen bzgl. der Waffen auf anderer Ebene hierarchischer Ebene entstehen was auch zu Kollisionen mit dem bereits verwendetem Network Transform führen kann.

Im Bezug auf der Trefferregistrierung, die ein essentieller Bestandteil von Multiplayer Spielen sind. In allen Fällen wird dies unter der Haube in Kommunikation von Clients und Server mittels Remote Procedure Calls getätigt, sodass hier im Idealfall eine Trefferbestätigung an den Client zurückgeliefert wird. Bei hohem Ping ist das jedoch problematisch und führt oft zu extremen Verzögerungen, die nicht nur sehr auffallend sondern auch behindernd sein können. In Vielen Fällen kommt hierbei noch visuelles und auditives Feedback für den Spieler also den betrfoffenen Client in Form von Hitmarkern dazu. Hitmarker oder Treffermarkierungen kommen häufig in eher im Bereich von Gelegenheitsspielen vor um den Spieler ein besseres Feedback zu gewährleisten, ob denn überhaupt ein Schuss registriert wurde. In den meisten Fällen ist dieses Feature nicht mit Rückkopplung an den Server gebunden sondern ausschließlich vom Client abhängig um nicht gegen das Prinzip der direkten Eingabe zu verstoßen (Client Side Prediction). Das kann wiederum je nach Umsetzung auch zum Verhängnis werden, wenn nämlich fälschlich erkannte Treffererkennung stattfindet. Dieses besagte Phänomen lässt sich gut in Spielen wie CS2 beobachten, in dem man dieses Feature: "Damage Prediction"variabel aktivieren kann.

Das vorgestellte Szenario verwendet jedoch keinerlei Damage Prediction beim Treffer, sondern wartet gegensetzlich der Norm auf die Trefferbestätigung vom Server um entscheidende Mechanismen besser zu demonstrieren. (vielleicht nicht so smart)

4.3 Technische Herausforderung: Raycasting

Ein zentrales technisches Problem liegt bei der Definition des Ursprungs der Schusslinie (Raycast). Diese Thematik betrifft nicht nur die Handhabung im Zusammenspiel mit Netcode sondern auch aktiv den Game Loop, sollten mehrere Waffenmodelle im Spiel sein.

Zwei gängige Ansätze wurden gegenübergestellt:

1. Raycast vom Lauf der Waffe (Mündung)
2. Raycast aus dem Zentrum der Kamera (Fadenkreuz)

Für den Prototyp wurde Variante 2 gewählt, da diese Methode eine konsistentere Treffergenauigkeit gewährleistet und im Hinblick auf Debugging und Netzwerkanalyse einfacher zu handhaben ist. Aus diesen Gründen wird dieser Ansatz auch bei der Entwicklung größerer Spiele - vor allem Spiele, die Hit Scan implementieren - präferiert.

4.4 Netzwerkarchitektur des Prototypen

Der Prototyp basiert auf dem *Netcode for GameObjects*-Framework von Unity und nutzt ein Server-Client-Modell. Clients werden über den Unity-Build gestartet und mit einem lokalen Server verbunden. Netzwerkkomponenten wie Zustandsübertragung und Objektregistrierung erfolgen über Unity's `NetworkTransform`-Komponente.

Diese Komponente erlaubt eine schnelle Prototypenerstellung, kann jedoch automatisch Funktionen wie Interpolation und Zustandssynchronisation übernehmen. Im Kontext dieser Arbeit ist das nicht ideal, da viel Eigenkontrolle über alle Mechaniken herrschen sollen und somit eine Architur benutzt werden muss, die stark vomn Network Transform abhängt. In Abwägung im Bezug auf Architektur und Mehraufwand ist der Network Transfrom zumindest im Bereich der Spielerbewegung sinnvoller, da sonst noch mehr unvorhersehbare Variablen und Komplexität beim Entwicklen und Debugging entstehen.

Die verwendete Architektur ist jedoch zu unterteilen in zwei Bereiche, die bewusst voneinander getrennt wurden. Zum einen handelt es sich bei First Person Shootern um Spielerbewegungsmechaniken, die sehr entscheidend sind beim Synchronisieren und Kompensieren von Netzwerkzuständen. Dies ist eng verbunden mit der Schussmechanik und dessen Schussregistierung, da vor allem bei sich bewegenden Zielen insbesondere Spielern, sehr viele Abhängigkeiten entsehen und viel auf geteilte Payloads zugegriffen wird und werden muss. Da in diesem Szenario keine PvP (Player versus Player) vorgesehen war und ist, wird auf diese zusätzliche Komplexität verzichtet und eine getrennte, etwas schlichtere Architektur verwendet. Hierbei sollen jediglich bei der Schussregistrierung, bei den vom Server gesteuertem Target, Netzwerkmechanismen repliziert werden. Aufgrund des grundlegende Unterschiedes, dass es sich hier um ein Serverobjekt handelt und kein Clientobjekt, verfällt hierbei die Komplexität die durch die Trivialisierung in der Netzwerkschicht Zustande kommt. In diesem Fall müssen somit Pakete nur vom Client zum Server

und vom Server zum Client gesendet werden, was im Grunde zu einer Halbierung der theoretischen Latenz führt. Somit handelt es sich hierbei um kein System, dass hohe Einsatzrelevanz in echten PvP Spielen hat, kann aber im Rahmen dieser Arbeit die Mechaniken veranschaulichen. (hier nochmal schauen, ob man das nicht besser hinbekommt).

4.5 Analysewerkzeuge für Netcode und Debugging

Zur Untersuchung netzwerkbasierter Spielmechaniken wie Client-Side Prediction, Lag Compensation oder Interpolation ist der Einsatz geeigneter Analysewerkzeuge essenziell. Ziel ist es, sowohl das Verhalten dieser Mechaniken unter verschiedenen Bedingungen zu verstehen, als auch deren Einfluss auf das Spielgefühl sichtbar und messbar zu machen.

Im Rahmen dieser Arbeit soll ein System verwendet werden, bestehend aus einem Debug-Overlay und einer integrierten Ingame-Konsole. Dieses Werkzeug ermöglicht es, netzwerkrelevante Parameter – wie beispielsweise Interpolationszeiten, künstliche Latenz oder Glättungsalgorithmen – zur Laufzeit zu verändern. Alternativ wird dies nach herkömmlichem Ablauf über Netzwerkvariablen im Unity-Editor
So lassen sich spezifische Szenarien gezielt nachstellen und deren Auswirkungen visuell nachvollziehen.

Neben dieser Eigenentwicklung existieren auch externe Lösungen, wie etwa der Unity Profiler mit Netcode-Unterstützung oder Werkzeuge von Drittanbietern wie Photon Fusion Analyzer [**<empty citation>**] (könnte sein, dass das hier kompletter bs ist). Diese bieten detaillierte technische Einblicke, sind jedoch oft nicht für eine interaktive Analyse innerhalb des Spiels ausgelegt.

Für die Konsole, die auch in Spielen wie Counter Strike oder anderen Spielen existiert um ohne Benutzeroberfläche Einstellungen zu ändern, wird hier aus dem Unity Asset Store bezogen. [**<empty citation>**] Mit diesem Asset lassen sich Konsolen-Kommandos im Quellcode registrieren, die den Spielfluss manipulieren können. Außerdem ist auch das Auslesen der Debugging-Konsole möglich wenn man im Build ist, welcher keine integrierte Konsole besitzt wie der normale Editor.

Das entwickelte Debug-System hingegen erlaubt eine tiefere Integration in den Entwicklungsprozess: Es zeigt Zustände wie die aktuelle Netzwerkverzögerung, Tick-Synchronisation

oder Prediction-Fehler direkt im Spiel an. Ergänzend dazu erlaubt die Konsole das Aktivieren und Deaktivieren einzelner Netcode-Komponenten oder das dynamische Nachjustieren von Parametern – ohne das Spiel neu starten zu müssen.

Durch diese Flexibilität ist das System besonders geeignet für die iterative Entwicklung und Bewertung von Netcode-Strategien und stellt daher das zentrale Analysewerkzeug dieser Arbeit dar.

4.5.1 Ausblick:

In den kommenden Entwicklungsschritten soll die
`NetworkTransform`-Komponente teilweise durch eine eigene Implementierung ersetzt werden, um die Funktionsweise und Auswirkungen von Interpolation, Prediction, Reconciliation und Lag Compensation explizit untersuchen und visualisieren zu können. Hierbei werden die einzelnen Arbeitschritte und Methoden zur Umsetzung dieser Visualisierung Anhand von Quellcode und nötigen Schritten innherhalb des Unity Editors gezeigt.

Kapitel 5

Aufbau der Spielmechaniken

Für das Verständnis der später eingesetzten Netcode-Mechaniken ist eine vorherige Erläuterung der implementierten Spielmechaniken unerlässlich. Diese Spielmechaniken sind von besonderer Bedeutung, da sie in engem Zusammenhang mit den Netzwerkfunktionen stehen, die im weiteren Verlauf behandelt werden. Das Spiel verfügt über die grundlegenden Funktionalitäten eines klassischen FPS-Titels; so kann der Spieler beispielsweise:

- Laufen und springen
- Mit einer Waffe schießen und diese nachladen
- Ein beweglich Ziel zerstören

Die erwähnten Bestandteile sind somit erweiterbar und stellen ein Grundgerüst von Funktionalitäten dar. Es können modular Waffenmodelle hinugefügt und mit weiteren Spielmechaniken ergänzt werden.

5.1 Bewegung

Die Art und Weise, wie sich ein Spieler im Spiel bewegen kann, ist von grundlegender Bedeutung für die Wahrnehmung und das Spielerlebnis. Wenn bspw. eine zu hohe Bewegungsgeschwindigkeit implementiert ist, fühlt sich das Spiel unter anderem nicht realistisch an. Eine zu statische Kamera kann bei Spielerbewegung ebenfalls zu einem unechten Spielgefühl beitragen. Wenn also bei Spielerbewegung keinerlei Bewegung in der

Waffenführung erkenntlich ist, wird auch je nach Spielfeld keine konkrete Bewegung wargenommen. Um das zu unterbinden ist für den Spieler clientseitge Waffenbewegung implementiert, die sich bei Rotation durch Mausbewegung und Laufbewegung erkenntlich macht. Hierbei handelt es sich um Gun Bobbing, was in leichter Form beim Stehen auftritt und in stärker Form beim Laufen. Außerdem gibt es noch die Waffenrotation, die beim Zielen (5.1.2) erkenntlich ist.

Maus- und Tastatureingaben werden in Unity separat ausgewertet, erfüllen jedoch gemeinsam den Zweck, die Bewegung des Spielers zu steuern. Die horizontale Rotation der Spielfigur über die Maus ermöglicht eine präzise Bestimmung der Blick- und Bewegungsrichtung und kann – ähnlich wie beim Autofahren das Lenkrad in Kombination mit dem Gaspedal – als zentrales Steuerelement für die Navigation betrachtet werden.

Die Tastatureingaben werden nicht durch hartcodierte Tastenabfragen realisiert, sondern über einen von Unity bereitgestellten *Input Reader* verarbeitet. Dieses abstrahierte Input-Handling-System erlaubt es, Tastenbelegungen flexibel im Unity-Editor festzulegen und im Code darauf zuzugreifen. Dies bringt insbesondere Vorteile im Zusammenhang mit Netcode-Mechanismen, da Eingaben so leichter serialisiert und für den Netzwerktransfer vorbereitet werden können. Darüber hinaus erleichtert es die Erweiterung auf alternative Eingabegeräte wie Controller oder Touch-Eingaben, ohne die interne Logik anpassen zu müssen.

(hier Abbildung zum Input Reader vielleicht).

5.1.1 Funktionsweise der Bewegung

Die reine Bewegung durch die Richtungstasten "WASD"werden mit einer einzelnen Funktion verarbeitet, die einen Eingabevektor als Parameter nimmt. Abgesehen von den Richtungseingaben setzt die Funktion auch Gravitationslogik um, hierbei also jediglich das Springen und das Fallen des Spielers. Dazu sind in Unity bestimmte Game-Objects also Teile des Spielfelds als Ground Layer deklariert, was für den Spieler als betretbare Fläche gilt. Hierzu bietet sich an in der Hierarchie den gesamten Spielbereich als Ground-Layer zu deklarieren, was nicht in jedem Fall gewollt ist, da man in manchen Fällen nicht die Fähigkeit besitzen sollte, auf bestimmten Objekten zu stehen. Hierzu wird eine neue Layer erstellt und der gesamte oder nur Teile des Environment-Containers mit dem *Ground*-Tag markiert.

(Screenshot hierzu, Text u.U. auch ausbaufähig)

Diese Schicht oder Ground-Layer ist im Code explizit als SerialzeField annotiert und kann somit im Inspector per drag and drop, als Referenz im jeweiligen Skript zugewiesen werden. Zuzüglich gibt es eine weitere Variable, die eine konkrete Fläche abbildet und als Transform verwendet wird. Diese Fläche ist nötig, um den Abstand vom Spieler zum Boden zu berechnen und den Grounded-State zu beurteilen und zu verwalten. Dieser State ist ein Zustand den der Speieler hat, wenn er sich nicht in der Luft befindet und somit Kontakt zur als Boden markierten Schicht hat. Dieses leere Game Object ist unterhalb des Spielers gesetzt und analog zur Layer Mask im Editor gesetzt und im Code referenziert. Auf diese Weise lässt sich mit der Funktion CheckSphere aus dem Modul: Unity.Physics diesen Sachverhalt bearbeiten. Diese Funktion gibt einen boolschen Wert zurück der innerhalb eines eigenen Feldes gespeichert wird und somit verwendet werden kann.

```
1  void Move(Vector3 inputVector)
2  {
3      if (DebugLogManager.IsConsoleOpen) return;
4
5      _isGrounded =
6          Physics.CheckSphere(groundCheck.position,
7                              groundDistance,
8                              groundMask);
9
10      if (_isGrounded && _velocity.y < 0f)
11          _velocity.y = -2f;
12
13          Vector3 move =
14              transform.right *
15              inputVector.x +
16              transform.forward *
17              inputVector.z;
18
19          controller.Move(move * (speed * Time.fixedDeltaTime));
20
21          if (playerInput.JumpPressed && _isGrounded)
22              _velocity.y = Mathf.Sqrt(jumpHeight * -2f * gravity);
23
24          _velocity.y += gravity * Time.fixedDeltaTime;
25          controller.Move(_velocity * Time.fixedDeltaTime);
26 }
```

Listing 5.1: Methode `Move()`

Beim Prüfen auf ein geerdetes Spielerobjekt und ob die aktuelle Geschwindigkeit in Y-Richtung negativ ist, wird grundsätzlich diese Geschwindigkeit auf -2 gesetz. Das hat den Grund, dass in vielen Fällen ein unsauberers Verhalten auftreten kann und der Spieler in manchen Fällen ungewollt durch das Spielfeld fallen könnte. (2f wegen Wurf-Formel?)

Anschließend wird das eigentliche Spielerobjekt – in diesem Fall der Character Controller – bewegt. Dazu wird dessen *Move*-Methode aufgerufen, welche von Unitys CharacterController-Klasse bereitgestellt wird. Die übergebene Verschiebung ergibt sich aus dem Produkt des gewünschten Bewegungsvektors, der aktuellen Geschwindigkeit und Time.fixedDeltaTime. Die Einbeziehung von **fixedDeltaTime** ist entscheidend, um Framerate-Abhängigkeiten zu vermeiden, die andernfalls zu inkonsistenten Bewegungsgeschwindigkeiten führen würden. Durch die Verwendung dieser zeitlichen Konstante, welche im physikbasierten FixedUpdate-Hook von Unity mit einer festen Frequenz (standardmäßig 50 Hz) aufgerufen wird, bleibt die Bewegung des Spielers unabhängig von der tatsächlichen Bildwiederholrate konsistent.

Berechnung der Anfangsgeschwindigkeit für einen Sprung (senkrechter Wurf nach oben)

Um eine gewünschte Sprunghöhe h zu erreichen, muss eine entsprechende Anfangsgeschwindigkeit v_0 gegen die konstante Schwerkraft g aufgebracht werden. Die Berechnung erfolgt nach den Gesetzen des senkrechten Wurfs nach oben:

$$v^2 = v_0^2 + 2as$$

Am höchsten Punkt des Sprungs ist die Geschwindigkeit $v = 0$, die Beschleunigung a entspricht der Schwerkraft g (meist negativ, z. B. $g = -9{,}81\,\text{m/s}^2$) und der zurückgelegte Weg s entspricht der gewünschten Höhe h. Umgestellt nach v_0 ergibt sich:

$$0 = v_0^2 + 2gh$$

$$v_0^2 = -2gh$$

$$v_0 = \sqrt{-2gh}$$

- v_0: Anfangsgeschwindigkeit beim Absprung (in m/s)
- g: Erdbeschleunigung / Schwerkraft (in m/s^2, typischerweise negativ)

- h: gewünschte maximale Sprunghöhe (in m)
- v: Geschwindigkeit am höchsten Punkt (hier: 0)

Die Berechnung stellt sicher, dass der Charakter unabhängig von der gewählten Schwerkraft und Sprunghöhe immer genau auf die gewünschte Höhe springt.

Beispiel

Soll eine Sprunghöhe von $h = 1{,}5\,\text{m}$ bei $g = -9{,}81\,\text{m/s}^2$ erreicht werden, so berechnet sich die notwendige Anfangsgeschwindigkeit zu:

$$v_0 = \sqrt{-2 \cdot (-9{,}81) \cdot 1{,}5} = \sqrt{29{,}43} \approx 5{,}43\,\text{m/s}$$

In den restlichen Fällen ist der Spieler bereits in der Luft oder fällt von einem erhöhtem Objekt. Demnach wird eine negativer Wert für die Geschwindigkeit in Y-Richtung vorausgesetzt.

5.1.2 Sicht und Mausbewegung

In den allen Unity Projekten ist der zentrale visuelle Bestandteil die Kamera, welche einem Spielerobjekt hierarchisch untergeordnet ist. Die Platzierung der Kamera ist aus Sicht des Spieldesigns entscheidend, weil die Platzierung den Kern des Spiels grundlegend ändert. Wenn die Kamera also hinter dem Spieler platziert wird, handelt es sich um ein Third Person (Vogelperspektive) Spiel in diesem Fall ein Third Person Shooter. In anderen Fällen wird die Kamera auch auf physikalischer Kopfebene des Spielers platziert. Hierbei handelt es sich dann um ein First Person Spiel oder First Person Shooter (FPS). In einigen Fällen kann zwischen diesen Ansichten auch gewechselt werden, was in einer einfachen Implementierung dem Ändern des Transforms der Kamera gleicht.

Die Maus oder der Joystick sind nicht nur zur Feinjustierung der Bewegung wichtig, sondern auch der Kern des Zielens. Hierfür müssen Bewegungen auf X- und Y-Achse aufgenommen und umgesetzt werden. Üblicherweise werden diese beiden Achsen getrennt und in manchen Fällen auch als separate Mausempfindlichkeiten integriert, welche zur Feinjustierung beim Zielen dienen soll. Diese ist frei wählbar und auch so implementiert, dass man sie über einen Konsolenbefehl frei ändern kann.

Die Berechnung und Einstellung der Mausempfindlichkeit erfolgt durch Multiplikation der rohen Eingabewerte (`Input.GetAxis("Mouse X")` bzw. `"Mouse Y"`) mit dem einstellbaren Empfindlichkeitsfaktor (`mouseSensitivity`) sowie `Time.deltaTime`. Die Skalierung mit `Time.deltaTime` stellt sicher, dass die Steuerung unabhängig von der Framerate konsistent bleibt.

In Unity oder in der Spielentwicklung werden im Umgang mit Rotationen meistens Quaternionen verwendet während bei Positionen 2-Dimensionale oder 3-Dimensionale Vektoren verwendet werden. Diese Abgrenzung ist zum einen für die Entwickler gut zur schnellen Abgrenzung beim Lesen von Quellcode und zum anderen vermeidet man konsequent kardanische Blockaden, da diese bei Quaternionen nicht existieren. Wenn für die Maussteuerung demnach keine Quaternionen verwendet werden, wie es hierbei der Fall ist, muss dafür kompensiert werden um Fehlverhalten zu vermeiden.

Um das sogenannte Gimbal Lock-Problem oder Kardanische Blockade zu vermeiden, wird die Rotation der Kamera um die lokale x-Achse (`_xRotation`) mittels `Mathf.Clamp` auf den Bereich zwischen $-90°$ und $+90°$ begrenzt. Gimbal Lock bezeichnet den Zustand, bei dem durch Rotationen um mehrere Achsen eine Freiheitsgrad verloren geht und unerwünschte Effekte wie das Umklappen oder Überschlagen der Kamera auftreten können. Die Begrenzung sorgt dafür, dass der Spieler maximal nach oben oder unten blicken kann, jedoch nie die Grenze überschreitet, an der die Steuerung unnatürlich oder fehlerhaft wird.

Die tatsächliche Kamerarotation erfolgt durch Setzen der lokalen Rotation des Kameraobjekts mittels `Quaternion.Euler(_xRotation, 0f, 0f)` für die vertikale Achse sowie durch Rotation des Spielerobjekts (hier `playerBody`) um die Welt-y-Achse (`Vector3.up`) für horizontale Mausbewegungen. Durch diese Trennung der Achsen wird eine intuitive und fehlerfreie First-Person-Steuerung realisiert.

Die Methode `SetSensitivity(float val)` ermöglicht es, die Empfindlichkeit der Mausbewegung zur Laufzeit anzupassen, wodurch die Steuerung individuell auf die Präferenzen des Nutzers abgestimmt werden kann.

5.2 Schießen

Beim Schießen wird primär eine Mechanik verwendet die aus Spielen wie Valorant oder Counter Strike üblich ist. Es wird somit auf eine primäre Zielfunktionalität verzichtet (Aiming Down Sights). Da es sich hierbei vor allem auch um Animationen und Waffenmodelle bzw.

Assets handelt, ist das kein komplexer Mehraufwand sondern eher eine Frage des Spieldesigns. Somit wäre ein ADS-Mechanismus eine mögliche Erweiterung, ist allerdings nicht zwingend notwendig oder gewollt. Abgesehen davon führt es zu keiner besseren Visualisierung der vorzuführenden Netcode-Effekte, was es somit zu einer vor allem auf die Zeit bezogen, unnötigen Ergänzung macht.

Die gewählte Mechanik ist im Grunde kein realistisches Schussverhalten, da die Waffe eher an der Hüfte geführt wird und von dieser Position treffsicher geschossen werden kann. Hierfür wird ein Crosshair oder Fadenkreuz verwendet, welches als Teil des Heads Up Displays(HUD) des Spielers existiert. Dieses HUD ist in Unity explizit ein Prefab, welches als eine im Code referenzierte Variable verwendet werden kann. Hiermit können Anpassungen auf Sichtbarkeit und weitere designspezifische Aspekte im Kontext des Spielflusses geändert werden. Dementsprechend gibt der Lauf der Waffe nicht das entscheidende Ziel des Schusses vor, sondern das Fadenkreuz in der Mitte des Bildschirms des Spielers. In vielen Spielen ist das jedoch eine sehr triviale und somit für den Spieler zu einfache Mechanik. Somit werden dementsprechend auch im Sinne des Realismus erschwerende Mechaniken wie wie mechanischer und prozeduraler Rückstoß integriert. Mechanischer Rückstoß kann in den meisten Fällen, abhängig von der Waffe entweder einem fixen oder eher freierem Muster folgen. Im Kontext auf den Verzicht auf die ADS-Mechanik, ist es somit üblich, dass Schüsse bei erhöhter Feuerrate nicht zwingen dort landen wo auch das Fadenkreuz zum Zeitpunkt eines Schusses liegt. In dieser Arbeit ist jedoch die Rückstoßberechnung prozedural gestaltet, indem jeder Schuss für eine visuelle Verschiebung und Rotation der Waffe sorgt. Prozedural bedeutet in diesem Kontext, dass der Rückstoß oder zumindest dessen visuelle Erscheinung, algorithmisch entsteht.

```
1  class GunSway
2  {
3      public void ApplyRecoil(Vector3 positionKickback,
4                               Vector3 rotationKickback)
5      {
6          _recoilOffset += positionKickback;
7          _recoilRotation *= Quaternion.Euler(rotationKickback);
8      }
9
10 }
11
12 class Gun
13 {
14     void Recoil()
15     {
```

```
16          gunSway?.ApplyRecoil(new Vector3(0, 0, -0.05f),
17                               new Vector3(-2f, 1f, 0f));
18      }
19  }
```

Listing 5.2: Prozedural-visueller Rückstoß

Diese Implementation hat keine Einflüsse auf die Spielmechanik und ist somit von Netcode spezifischer Komplexität befreit. Das hat den Vorteil im Kontext der visuellen Interpretierbarkeit Abschnitt 6(genauer), dass es keine Varianzen im Schussverhalten gibt.

Der grobe Ablauf beim Waffe abfeuern ist eventbasiert implementiert. Es sind von zwei Events die Rede, nämlich das Schießen zum einen und das Nachladen zum anderen. Wie bereits erwähnt ist ein Skript, welches für die Mausbewegung zuständig ist ausschlaggebend für die Positionierung des Fadenkreuzes und somit der Kameramitte. Wenn somit ein Schuss abgefeuert wird, soll genau dort wo die Kameramitte liegt, ein Raycast projeziert werden. Ein Raycast hat somit eine Endposition und an dieser Position soll ein Einschussloch gerendert werden. Ein Einschussloch dient somit der Validierung der Schussmechanik und zur besseren Immersion des Spiels.

Im Bezug auf das Schießen auf Ziele, soll eine Waffe Metadaten beinhalten wie:

```
1  using UnityEngine;
2
3  [CreateAssetMenu(fileName="Gun", menuName = "Weapon/Gun")]
4  public class GunData : ScriptableObject
5  {
6      [Header("Info")]
7      public new string name;
8      [Header("Shooting")]
9      public float damage;
10     public float maxDistance;
11
12     [Header("Reloading")]
13     public int currentAmmo;
14     public int magazineSize;
15     public float fireRate;
16     public float reloadTime;
17     [HideInInspector]
18     public bool isReloading;
19 }
```

Listing 5.3: Scriptable Object GunData

Hier wird unter anderem das Attribut **CreateAssetMenu** verwendet um ein dediziertes Menü in Unity zu erstellen, das die Wertezuweisung möglich macht.

Das Attribut ermöglicht überhaupt ein Anlegen einer solchen Asset Datei, wodurch dann erst ein Menü sichtbar wird. Spezifikationen werden mit den Parametern **fileName** und **menuName** übergeben und festgelegt.

Diese Metadaten können pro Waffe sehr unterschiedlich ausfallen und werden über eine Asset-Datei, welche als Konfigurationsdatei im YAML-Format geschrieben ist, gespeichert. Ein weiteres Attribut: **[HideInspector]** kann verwendet werden um bestimmte Felder im Unity-Editor *unsichtbar* zu machen. Das ist üblich, wenn man Felder nur innerhalb der Spiellogik dynamisch ändern will. Hierbei ist dies der Fall mit dem Status des Nachladevorgangs. Wenn sich nämlich dieser Status ungewollt ändert, kann somit das Schießen gesperrt werden. Durch das Attribut kann man dadurch theoretisch anfallenden Debugging Aufwand reduzieren oder vermeiden, da die Kontrolle auf den Nachladestatus beschränkt ist. Ansonsten ist beim Starten des Spiels eine Wiederherstellung des Standardwertes nötig, um dieses Fehlverhalten bei allen Spielern zu vermeiden.

Wichtige Felder in diesem Kontext sind die Metadaten, die das Schießen direkt oder transitiv beeinflussen. Wenn sich aus einem der Felder eine abgleitete Eigenschaft bilden lässt, ist sie demnach ebenso wichtig.

Die grundlegende Funktionalität basiert auf zwei Events, die beim Start des Spiels jeweils entsprechende Methoden abonnieren und beim Beenden wieder deabonnieren. Je nach Situation kann entweder nachgeladen oder geschossen werden; beide Aktionen schließen sich gegenseitig aus. Befindet sich der Spieler beispielsweise im Nachladevorgang, ist das Schießen für die verbleibende, in den Metadaten festgelegte Nachladezeit nicht möglich.

Jedes Mal wenn also versucht wird zu schießen wird diese Prüfung vorgenommen:

```
public class Gun
{
    private void Update()
    {
        if (!IsOwner) return;
        if (InputState.InputLocked) return;

        _timeSinceLastShot += Time.deltaTime;
        UpdateAmmoUI();
        UpdateHitmarker();
    }


    public bool CanShoot() =>
            !gunData.isReloading && _timeSinceLastShot >
            1f / (gunData.fireRate / 60f);
}
```

Listing 5.4: Methode CanShoot

Die Methode `CanShoot()` prüft, ob die Waffe aktuell nicht nachgeladen wird und ob seit dem letzten Schuss ausreichend Zeit vergangen ist. Letzteres wird anhand der akkumulierten Zeit seit dem letzten Schuss (`_timeSinceLastShot`) und der Feuerrate berechnet. Nur wenn beide Bedingungen erfüllt sind, kann ein weiterer Schuss abgegeben werden. Dadurch wird sichergestellt, dass zwischen zwei Schüssen stets ein Mindestabstand gemäß der festgelegten Feuerrate eingehalten wird.

5.2.1 Nachladen

Zum Nachladen wird eine Coroutine verwendet, die eine asynchrone Verarbeitung des Nachladevorgangs ermöglicht. Coroutinen sind ein besonderes Feature von Unity, das auf dem IEnumerator-Interface aus C# basiert. Während im allgemeinen .NET-Umfeld für asynchrone Operationen üblicherweise das async/await-Pattern mit Task verwendet wird, nutzt Unity Coroutinen, um nicht-blockierende Abläufe innerhalb des Game Loops zu realisieren. Das Konzept von Coroutinen existiert auch in anderen Programmiersprachen, etwa in Python oder Kotlin, wird jedoch jeweils unterschiedlich implementiert [9].

Wenn ein Spieler nachlädt, ist es demnach sinnvoll wenn der in den Metadaten festgelegte Nachladetimer nicht den Mainthread blockiert. Die Methode oder Coroutine gibt also ein Objekt zurück, das das Interface IEnumerator implementiert und wird wenn man sich nicht aktuell im Nachladeprozess befindet, mit Hilfe einer weiteren Methode aufgerufen.

```
1  public class Gun
2  {
3      private void StartReloading()
4      {
5          if (!gunData.isReloading)
6          {
7              StartCoroutine(Reload());
8          }
9      }
10
11     private IEnumerator Reload()
12     {
13         gunData.isReloading = true;
14         yield return new WaitForSeconds(gunData.reloadTime);
15         gunData.currentAmmo = gunData.magazineSize;
16         gunData.isReloading = false;
17     }
18 }
```

Listing 5.5: Methoden zum Nachladen

Das eigentliche Nachladen beschränkt sich auf das Zurücksetzen der Urpsrungswerte wie sie in den Metadaten gespeichert sind. Diese Aktion findet demnach erst statt, wenn die Zeit zum Nachladen abgelaufen ist und wird dann in der Munitionsanzeige im HUD gerendert.

5.2.2 Zielobjekt und Spawner

Das Zielobjekt ist ein vom Server gesteuertes 3D-Objekt, das sich zufällig entlang der X-Achse bewegt. Sobald das Spiel gestartet wird, wird dieses Objekt innerhalb eines festgelegten Bereichs über das Netzwerk erzeugt und platziert. Ein sogenanntes Spawnfeld ist dabei ein zentraler Bestandteil, um Objekte oder Spieler im Spiel korrekt erscheinen zu lassen. Das gezielte Spawnen von Objekten im Multiplayer-Kontext ist eine grundlegende Mechanik, die in den meisten Mehrspieler-Spielen zum Einsatz kommt. In der Regel existieren mehrere Spawnpunkte, die als Array von `Transform`-Objekten bzw. Vektoren definiert werden, um eine flexible und gleichmäßige Verteilung der Spielobjekte zu gewährleisten. Mehrere Spawn-Felder sind jedoch eher in größeren Multiplayer-Spielen entscheidend, wobei ein durchdachtes Spawnsystem implementiert werden muss und auf Spawnpunkte je nach Kontext des Spielzustandes zugegriffen wird. In diesem einfachen Szenario gibt es nur eine Fläche in der das Zielobjekt erscheint und nach Abschuss zerstört und nach einer Verzögerung neu gesetzt wird. Für diese Mechanik ist wiederum ein Prefab dieses Zielobjektes nötig, von dem es Klone geben kann. Diese Klone haben alle unterschiedliche Network-IDs und werden somit klar verwaltet.

Für das Zielobjekt existiert ein dediziertes Spawner-Skript, das die Verantwortung für das Spawnen des Zielobjekts innerhalb des definierten Spawn-Feldes übernimmt. Damit ein korrektes Spawn-Verhalten gewährleistet ist, müssen sowohl das Target- als auch das Spawner-Skript als `NetworkObject` ausgelegt sein. Für eine sinnvolle Integration der beiden Skripte werden gegenseitige Abhängigkeiten benötigt, die in der aktuellen Implementierung durch eine Dependency-Injection-ähnliche Methodik erlangt werden. Im Unterschied zu einer klassischen Dependency Injection erfolgt die Zuweisung der Abhängigkeiten jedoch nicht von außen, sondern wird im Skript selbst zur Laufzeit ermittelt.

```
1 void Start()
2 {
3     _spawner = FindFirstObjectByType<Spawner>();
4
5 }
```

Listing 5.6: spawn

Durch diesen Funktionsaufruf kann somit generisch auf das erste Objekt dieses Typs zugegriffen werden. In diesem Szenario ist das noch möglich, da sich niemals mehr als ein Spawner-Objekt im Spiel befinden können.

Die eigentliche Spawn-Logik ist was die Bereitstellung der Abhängigkeiten angeht, ähnlich aufgebaut und nutzt ebenfalls einen generischen Zugriff auf das Objekt der anderen Klasse um es dann zu setzen. Konkret ist für das Spawnen ein referenziertes Prefab, was in dem Fall, dass Prefab des Zielobjekts ist, eine zufällige Position innerhalb des des Spawnfeldes und eine Rotation nötig. Hiermit lässt sich eine Instanz erstellen, welche dann in erster Linie über das Netzwerk gesetzt werden kann.

```
class Spawner
{
    private void Respawn()
    {
        Vector3 randomPosition = GetRandomPositionInArea();

        var targetInstance =
                Instantiate(targetPrefab,
                            randomPosition,
                            Quaternion.identity);
        targetInstance.GetComponent<NetworkObject>().Spawn();
    }

    private Vector3 GetRandomPositionInArea()
    {
        float x = Random.Range(-spawnAreaSize.x / 2, spawnAreaSize.x / 2);
        float z = Random.Range(-spawnAreaSize.z / 2, spawnAreaSize.z / 2);
        return spawnAreaCenter + new Vector3(x, 0f, z);
    }

    public IEnumerator RespawnDelayed()
    {
        yield return new WaitForSeconds(2f);
        Respawn();
    }

    public override void OnNetworkSpawn()
    {
        if (IsServer)
        {
            Respawn();
        }
    }
}
```

Listing 5.7: Spawn

5.2.3 Kernlogik

Wenn eine Eingabe für einen Schuss registriert wird, soll also geprüft werden können was man getroffen hat. Ist es nämlich ein zufälliger Teil des Spielfeldes, hat dieser Treffer nur wenig Bedeutung. Wenn allerdings das vorgesehene Ziel getroffen wird, soll diesem Zielobjekt Schaden zugeführt werden. Hierbei muss beim Schuss kontrolliert werden ob dem getroffenen Objekt überhaupt Schaden zufügbar ist. Das wird mit Hilfe eines Interfaces getan, welches hiermit prüft, ob die Komponente dieses Interface implementiert und wenn das der Fall ist, kann auf dieser Komponente bzw. dessen Objekt die Methode zum Schaden zuführen aufgerufen werden.

```
1 if (Vector3.Distance(closestPoint, origin) < gunData.maxDistance)
2 {
3     var damageable = targetObj.GetComponent<IDamageable>();
4     damageable?.Damage(gunData.damage);
5 }
```

Listing 5.8: Damage

Um auszuwerten was beim Schuss getroffen wurde, ist im Multiplayer-Szenario die Hauptmethode für die Schussmechanik mit einem ServerRpc verbunden. In Kombination wird ein Raycast von Unity verwendet der prüft, ob ein Collider getroffen wurde innerhalb der Einschränkungen von z.B. der maximalen Schussdistanz. Nur in diesem Fall gibt die Funktion **Physics.Raycast()** überhaupt *true* zurück und es lassen sich Einschusslöcher am Ziel spawnen und sogar Schaden am Zielobjekt zufügen, falls dieses getroffen ist.

Ein Collider wäre in diesem Fall eine Wand, der Boden oder eben das Zielobjekt. Beim Beispiel des Zielobjektes was intern eine Kugel ist oder eine Sphere wird beim Erstellen dieses Objektes Initiieren dieses Prefabs automatisch ein analoger Collider mit angelegt. Passend zu dem Objekttyp handelt es sich hier um einen sogenannten Sphere Collider, welcher aus diesem Objekt ein greifbaren Gegenstand macht an dem Spieler oder Projektile kollidieren können.

Der beschriebene Ablauf verläuft wie folgt:

```
void Shoot()
{
    if (gunData.currentAmmo <= 0
        || !CanShoot()
        || _playerCamera == null) return;

    var camOrigin =
        _playerCamera.ViewportToWorldPoint(new Vector3(0.5f, 0.5f, 0f));
    var camDirection = _playerCamera.transform.forward;

    if (Physics.Raycast(camOrigin,
                        camDirection,
                        out RaycastHit camHit,
                        gunData.maxDistance,
                        hitMask))
    {
        ulong targetId =
            camHit
            .transform
            .GetComponent<NetworkObject>()?
            .NetworkObjectId ?? 0;

        ShootServerRpc(camOrigin,
                       camDirection,
                       targetId,
                       NetworkManager
                       .ServerTime
                       .Time - NetworkManager.LocalTime.Time);

        bulletManager.SpawnBulletHole(camHit,
                                      new Ray(camOrigin, camDirection));
    }

    gunData.currentAmmo--;
    _timeSinceLastShot = 0f;
    OnGunShot();
}
```

Listing 5.9: Methode Shoot

Um die Verarbeitung des Schusses sinngemäß mit dem Raycast zu tätigen braucht es nötige Parameter in der Funktion, die vor allem für die Ausrichtung und die Blickrichtung nötig

sind. Für den Ursprung der Kamera wird die Bildschirmmitte gewählt bzw. der Punkt in dem auch das Fadenkreuz liegt. Die Blickrichtung geht über die Z-Achse nach vorne und gibt somit die Schussrichtung an. Der Parameter **camHit** gibt Informationen über das getroffene Objekt an, beispielsweise die genaue Trefferposition oder den getroffenen Collider. In C# werden Parameter standardmäßig als Kopie („by value“) übergeben, sodass Methoden keine Änderungen an der ursprünglichen Variablen des Aufrufers vornehmen können. Durch das Schlüsselwort `out` wird dieses Verhalten jedoch geändert: Die Methode erhält einen Verweis auf die übergebene Variable und kann diese innerhalb ihres Gültigkeitsbereichs mit den relevanten Trefferinformationen befüllen. Die so befüllte Variable steht nach dem Methodenaufruf auch im aufrufenden Kontext mit den neuen Werten zur Verfügung [10].

Als LayerMask wird die bereits deklarierte Variable hitMask verwendet, die für den möglichen Trefferbereich zuständig ist. Eine Skybox wäre zum Beispiel kein Gültiger Treffer. In diesem Fall würde die Methode *false* zurückgeben. Wenn der Schuss außer Reichweite der Waffe sein sollte, es aber ein laut hitMask ungültiges Objekt trifft, wird ebenfalls *false* zurückgegeben. (hier unter Umständen unvollständig weil lost)

Wenn die Schussregistrierung damit abgeschlossen ist, wird letztlich die Munitionsazeige aktualisiert indem der aktuelle Munitionsstand im Magazin um eine Kugel abgezogen wird. Die Zeit seit dem letzten Schuss wird zurückgesetzt und die waffenspezifischen Animationen und das Schussgeräusch werden abgespielt.

Kapitel 6

Netcode Mechaniken

Im folgenden Abschnitt werden die im Projekt umgesetzten und verwendeten Mechaniken beschrieben. Hierbei liegt der Fokus darauf, ein durchdachtes System zu verwenden, das den Kern der Problematik einfach im Spiel erkenntlich machen kann. Viele der Mechaniken ergänzen sich gegenseitig und sind somit auch eng miteinander verbunden in der Implementierung als auch von der theoretischen Definition. (Mögliche Referenz zum Grundlagenteil)

In der Fachliteratur und technischen Dokumentation finden sich unterschiedliche Auffassungen zur Einordnung zentraler Netcode-Mechanismen wie Client-Side Prediction, Reconciliation, Lag Compensation und Interpolation. Zwei grundlegende Perspektiven lassen sich dabei unterscheiden:

Funktionale Trennung: In dieser Sichtweise werden die Mechanismen entlang ihrer funktionalen und systemarchitektonischen Zuständigkeit voneinander abgegrenzt. Lag Compensation wird dabei als rein serverseitige Rückrechnung von Spielzuständen verstanden, während Prediction und Reconciliation auf Client-Seite zur Vorhersage und Korrektur genutzt werden. Interpolation dient primär der visuellen Glättung gegnerischer Bewegungen und wird als unabhängige Darstellungsmechanik betrachtet. Diese Perspektive wird unter anderem von Glenn Fiedler [11] vertreten.

Systemische Integration: In der erweiterten Lesart – wie sie beispielsweise von Valve in der Dokumentation zur Source-Engine [12] vertreten wird – wird Lag Compensation als Sammelbegriff für alle Verfahren verstanden, die zur Kompensation von Netzwerklatenz und Synchronisationsabweichungen beitragen. Hier sind Prediction, Reconciliation und Interpolation funktional stark miteinander verknüpft und technisch teils gemeinsam implementiert, etwa durch geteilte Zustands- und Eingabepuffer.

Beide Sichtweisen bieten wertvolle Konzepte für den Entwurf und das Verständnis verteilter Spielsysteme. Im Folgenden wird gezeigt, wie diese Ansätze in der vorliegenden Arbeit kombiniert und auf ein praktisches Prototypsystem übertragen werden. Im Rahmen dieses Projektes wird ein System verwendet, das die Bewegungs und Schussmechanik trennt, wobei innerhalb der Bewegungsmechanik Lag Compensation und Client Side Prediction eng miteinander verbunden sind. Dies hat wie in Abschnitt 4.4 bereits angerissen, den Vorteil, dass das für das Target verwendete System, somit keinen systematischen Austausch zwischen Client(s) und Server benötigt. Dies spart semantische als auch syntaktische Komplexität, die bei einer kompletten systematischen Integration nur schwer verständlich wäre.

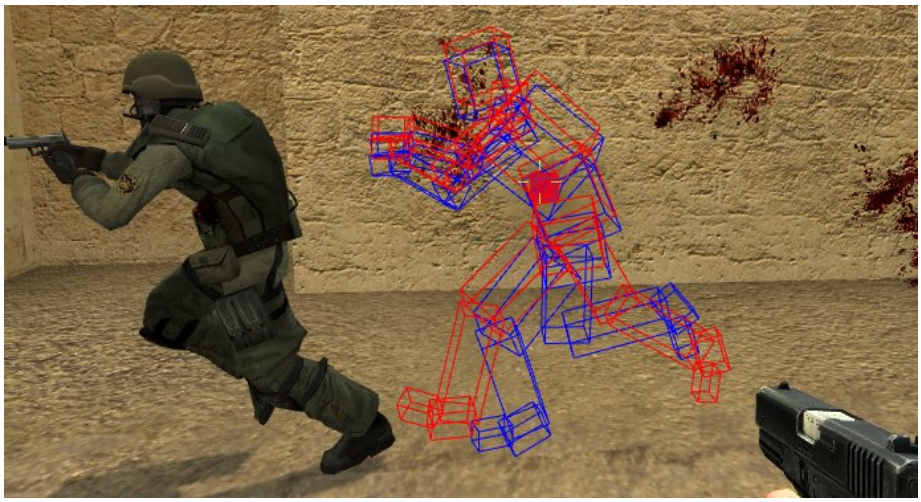

Abbildung 6.1: lag comp source engine

6.1 Client-side Prediction beim Bewegen und Schießen

In einem serverautoritativen Netzwerkmodell ist es essenziell, ein faires und sicheres Spielumfeld zu gewährleisten. Gleichzeitig darf die Spielbarkeit aus Sicht des Spielers nicht unter einer hohen Latenz leiden. Besonders bei reaktionsintensiven Genres wie First-Person-Shootern oder Rennspielen ist eine direkte Rückmeldung auf Eingaben entscheidend für das Spielerlebnis.

Ein vollständiger Verzicht auf Client-Authorität würde zwar maximale Sicherheit bieten, jedoch zu spürbarer Verzögerung bei der Bewegung oder Aktion führen. Ziel ist daher ein Kompromiss: Der Server behält die Entscheidungsgewalt, während der Client Eingaben sofort lokal umsetzt und visuelles Feedback liefert.

Bei der Visualiersierung sind in diesem Bereich keine ausschlaggebenden Erkenntnisse zu ziehen. Es handelt sich hierbei am ehesten um eine reine Feedback-Mechanik, die höchstens kumulativ in die Diskrepanz der eigentlichen Position des Clients und der vom Server gehaltene Position des Clients mit einfließt.

In der vorliegenden Implementierung wird dies durch ein Überschreiben der Unity-eigenen `NetworkTransform`-Komponente realisiert. Die Autoritätslogik wird über eine einfache Enumeration gesteuert:

```
using Unity.Netcode.Components;
using UnityEngine;

namespace Player {
    public enum AuthorityMode {
        Server,
        Client
    }

    [DisallowMultipleComponent]
    public class ClientNetworkTransform : NetworkTransform {
        public AuthorityMode authorityMode =
                Player.AuthorityMode.Client;

        protected override bool OnIsServerAuthoritative() =>
                authorityMode == Player.AuthorityMode.Server;
    }
}
```

Listing 6.1: Überschreiben der NetworkTransform-Komponente mit AuthorityMode

Die Methode `OnIsServerAuthoritative()` gibt in der Regel `false` zurück, sodass der Client die Kontrolle über Bewegung und Eingabe erhält, ohne dass ein Roundtrip zum Server erforderlich ist. Dadurch wird die Latenzwahrnehmung(Ping), beim Spieler erheblich reduziert.

6.2 Reconciliation – Zustandskorrektur bei Abweichungen

Trotz lokaler Vorhersage durch den Client bleibt der Server die Referenzinstanz für den „wahren“ Spielzustand. Sollte es zu größeren Abweichungen zwischen Client- und Serverposition kommen, wird ein Reconciliation-Prozess ausgelöst. Dabei wird der Client zurück auf den zuletzt vom Server bestätigten Zustand gesetzt und die eigenen Eingaben ab diesem Zeitpunkt erneut simuliert.

In Spielen mit präzisem Bewegungsfeedback (z.B. FPS) ist die Toleranzschwelle für Positionsfehler oft sehr gering. In diesem Projekt wurde ein Grenzwert: (`reconciliationThreshold`) definiert, ab dem eine Zustandskorrektur erfolgt. Dieser Wert kann abhängig von variabler Netzwerklatenz angepasst werden.

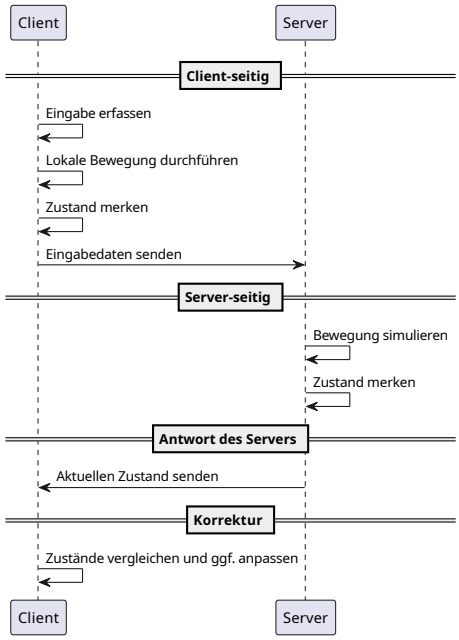

Abbildung 6.2: High Level Ablauf der Reconciliation

6.3 Testmechanismus zur Validierung

Zur gezielten Auslösung der Reconciliation wurde ein einfacher Testmechanismus implementiert: Ein Client-Cheat teleportiert den Spieler um 20 Einheiten nach vorne – weit über die Fehlertoleranz hinaus. Der Server erkennt die Abweichung und triggert die Korrektur, wodurch der Spieler sichtbar in den korrekten Zustand „zurückspringt“.

Zur Visualisierung wurden farblich unterschiedliche Proxies über dem Spielerobjekt angezeigt, die die vom Client bzw. Server berechneten Zustände darstellen. Wenn somit künstlich hinzugefügte oder native Latenz dazukommt, soll sich bei Bewegung der Abstand der Proxies vergrößern. Diese Proxies sind intern als einfache 3D-Objekte im Unity Editor als Prefabs angelegt worden. Sie eignen sich hierbei als Sphären oder Kugeln, da man so einfachste Positionsunterschiede leicht erkennen kann.

In der Praxis wird dies mit einem kleinen Cheat implementiert, welcher sich in Form eines Teleport-Taste äußert. Wenn diese Taste gedrückt wird springt der Spieler inklusive des Client Proxy nach vorne also um 20 Einheiten in Z-Richtung. Wenn somit der Client inklusive Proxy die Schwelle überschritten haben signalisiert das dem Server, dass eine ungültige Aktion geschehen ist und setzt den Spieler auf seine letzte gültige Position, Geschwindigkeit und Rotation zurück. Da dies in Echtzeit ziemlich schnell passiert kann man mit einem `Debug.Break()` den Frame einfrieren an dem die Schwellenüberschreitung geschehen ist, was in folgender Abbildung erkenntlich ist. Man sieht hier also genau diese Teleportierung die unter umständen von Spielern ausgenutzt werden könnte, sollte keine sinnvolle Reconciliation exisitieren. Je nach Spiel ist somit eine sinnvolle Schwelle zu wählen, die unter umständen auch variabel ist und sich an Netzwerkkonditionen anpasst. In Taktischen Shootern bswp. wäre ein grundsätzlicher Ansatz sinnvoll, die Schwelle etwas niedriger zu halten und bei hohem Ping des Spielers diese Schwelle variabel zu erhöhen. Dieser Ansatz ist vor allem dann sinnvoll wenn vom Spielentwickler viel Wert auf Präzision gelegt wird. Ein sinnvoller Grundwert würde hier zwischen 5-10 Einheiten liegen. In diesem Szenario ist alles legitim was unter dem Wert der Teleport-Distanz liegt und wird dann im späteren Ablauf auf die schlechter werdenden Netzwerkkondition angepasst beziehungsweise erhöht.

6.4 Datenstrukturen und Ticksystem

Als Datenstruktur wurde für das Ticksystem auf einen Ringpuffer gesetzt. Im Netcode-Bereich ist dies defacto Standard <Quelle maybe> um die mit den einzelnen Ticks verbundenen Informationen oder genauer Positionen, Rotationen und Geschwindigkeiten umzugehen und Overheads zu negieren. Demnach eignen sich bspw. Arrays oder Dictionaries deutlich schlechter, da diese die gespeicherten Informationen nicht verwerfen und somit unnötige Daten halten, die für bspw. die Reconciliation außer Reichweite liegen und somit die Performance beeinträchtigen. In Kombination mit State und Input Payloads die jeweils als Structs implementiert wurden, sollen die Positionen zum passenden Tick verglichen werden können. Die Eigenimplementierung des Ringpuffers ist somit generisch und basiert auf den Input oder State Payloads. Mit der Methode Add(), die ein generisches Item also in diesem Fall das State oder Input Payload und einen Index als Parameter nimmt, kann somit der Puffer befüllt werden. Das Befüllen folgt den algorithmischen Grundlagen eines solchen Buffers und das geschieht bis zu der statischen Buffergröße und fängt dann wieder von null an zu indezieren. Somit werden in diesem Fall intern bis 1024 Ticks verwaltet, was bei einer Standard-Tickrate von 60Hz ungefähr 17 Sekunden wären, die man zurückspulen könnte. Eine solche Puffergröße ist großzügig, sollte man jedoch auf eine Tickbasis aufsteigen von etwa 128Hz, ist die mögliche Rückspulzeit somit schon sinnvoller. In Anwendungsfällen ist ein Rückspulen was über eine Sekunde her ist spielhindernd und würde die Spielererfahrung deutlich verschlechtern.

In der Abbildung 6.2 sieht man noch den Aufbau des Circular Buffers und die Strukturen die essentiell sind für den Datentransfer:

```
namespace Player
{
    public class CircularBuffer<T>
    {
        private T[] buffer;
        private int bufferSize;

        public CircularBuffer(int bufferSize)
        {
            this.bufferSize = bufferSize;
            buffer = new T[bufferSize];
        }

        public void Add(T item, int index) =>
            buffer[index % bufferSize] = item;
        public T Get(int index) => buffer[index % bufferSize];
        public void Clear() => buffer = new T[bufferSize];
    }
}
```

LISTING 6.2: Methode `CircularBuffer()`

Auf der anderen Seite muss dies durch die angesprochenen Strukturen gespeichert werden:

```
public struct InputPayload : INetworkSerializable
{
    public int tick;
    public Vector3 inputVector;
    public bool forceTeleport;
    public Vector3 position;

    public void NetworkSerialize<T>(BufferSerializer<T> serializer)
        where T : IReaderWriter
    {
        serializer.SerializeValue(ref tick);
        serializer.SerializeValue(ref inputVector);
        serializer.SerializeValue(ref forceTeleport);
        serializer.SerializeValue(ref position);
    }
}
```

LISTING 6.3: Methode `Payload`

In der InputPayload werden die Eingabedaten des Spielers verwaltet und serialisiert, was über die vom zu implementierenden Interface INetworkSerializeable gestellten Methode NetworkSerialize() getan wird.

6.5 Kernmethoden der Reconciliation-Logik

Im Grunde lässt sich die Logik auf einige Methoden runterbrechen, dies sich alle im PlayerMovement Skript befinden. Diese Methoden sind jedoch verschachtelt in einem komplexeren Ablauf dieses Skriptes, reichen aber fürs Verständnis aus.

1. `ShouldReconcile()`
   Diese Methode prüft, ob neue Serverdaten vorliegen und ob diese sich vom zuletzt verarbeiteten Zustand unterscheiden:

```
bool ShouldReconcile()
{
   bool isNewServerState = !_lastServerState.Equals(default);
   bool isLastStateUndefinedOrDifferent =
        _lastProcessedState.Equals(default)
        || !_lastProcessedState.Equals(_lastServerState);
        return isNewServerState &&
               isLastStateUndefinedOrDifferent;
}
```

LISTING 6.4: Methode `ShouldReconcile()`

2. `ReconcileState(StatePayload rewindState)`
   Hier wird der Client-Zustand auf den vom Server bestätigten Zustand zurückgesetzt. Anschließend werden alle seitdem aufgezeichneten Eingaben erneut angewendet:

```
void ReconcileState(StatePayload rewindState)
{
    controller.enabled = false;
    transform.position = rewindState.position;
    transform.rotation = rewindState.rotation;
    _velocity = rewindState.velocity;
    controller.enabled = true;

    if (rewindState.Equals(_lastServerState)) return;

    _clientStateBuffer.Add(rewindState, rewindState.tick);
    int tickToReplay = _lastServerState.tick + 1;

    while (tickToReplay < _timer.CurrentTick)
    {
        int bufferIndex = tickToReplay % k_bufferSize;
        StatePayload statePayload =
        ProcessMovement(_clientInputBuffer.Get(bufferIndex));
        _clientStateBuffer.Add(statePayload, bufferIndex);
        tickToReplay++;
    }
}
```

LISTING 6.5: Methode `ReconcileState(StatePayload)`

3. `HandleServerReconciliation()`
   Diese Methode koordiniert den gesamten Reconciliation-Prozess und vergleicht die Zustände beider Seiten:

```
void HandleServerReconciliation()
{
    if (_lastServerState.tick <= 1) return;
    if (!ShouldReconcile()) return;

    int bufferIndex = _lastServerState.tick % k_bufferSize;
    if (bufferIndex - 1 < 0) return;

    StatePayload rewindState = IsHost
        ? _serverStateBuffer.Get(bufferIndex - 1)
        : _lastServerState;

    float positionError = Vector3.Distance(
        rewindState.position,
        _clientStateBuffer.Get(bufferIndex).position
    );

    if (positionError > reconciliationThreshold)
    {
        Debug.Break();
        ReconcileState(rewindState);
    }

    _lastProcessedState = _lastServerState;
}
```

LISTING 6.6: Methode `HandleServerReconciliation()`

Wenn diese Prozesskette dann ausgeschlagen ist, wird in einer Methode die sich um die Tickverarbeitung beim Client kümmert, aufgerufen. Diese Methode: `HandleClientTick()` soll also nur aufgerufen werden, wenn es sich bei der aktuellen Instanz um einen Spieler handelt. Hier wird mit den vom NetworkBehaviour geerbten Eigenschaften:

- `!IsClient`
- `!IsOwner`

geprüft ob es sich wirklich um einen Spieler und nicht doch eine Serverinstanz handelt. Sollte dies jedoch nicht der Fall sein, wird die Methode verlassen und die analog aufgebaute Methode zur Server-Tickverarbeitung aufgerufen: `HandleServerTick()`.

Zusätzlich ist wichtig, dass der Spieler bei häufiger Desynchronisation über dem erlaubtem Limit häufig zurückspringt. Das kann man auf verschiedene Weisen unterbinden, allerdings haben diese Herangehensweisen auch oft Nachteile. Hierbei ist ein Cooldown implementiert, der eine wiederholte Reconciliation zeitlich beschränkt. Nach jeder Reconciliation läuft somit ein Cooldown-Timer und erst nach Ablauf davon, kann der Spieler manuell oder durch zu hohen Desync zurückgesetzt werden. Wie bereits angesprochen hat das auch Nachteile und Risiken und somit muss in echten Szenarien ein Missbrauch dieser Mechanik unterbunden werden. Wenn ein Spieler über einen Cheat verfügt, ähnlich wie der in 6.3 Testmechanismus, jedoch extern und ungewollt.

6.6 Lag Compensation für Bewegung

Mit Lag Compensation wird versucht schlechte Netzwerkkonditionen auszugleichen. Wenn also ein Client eine unstabile Verbindung zum Server hat, soll Lag Compensation einen approximativen Ausgleich schaffen im besten Fall ohne, dass einer der Clients im Vorteil ist. Hierbei werden bei einer lokalen Testumgebung in Unity künstliche Latenzen, Paketverlust oder Jitter eingefügt um besagte Netzwerkkonditionen zu replizieren. Das beschriebene System für Lag Compensation bezieht sich auf die Spielerbewegung und ist somit eng mit dem System der zuvor angesprochenen Reconciliation verbunden, demnach ist auch das Validierungsumfeld sehr ähnlich. Technisch gesehen, ist Lag Compensation die Obermenge von allen Werkzeugen, die das Verstecken von Latenz oder ungünstigen Netzwerkkonditionen berwerkstelligen sollen. Dennoch lässt sich Lag Compensation explizit mit einer häufig verwendeten Mechanik, nämlich der Extrapolation implementieren. Extrapolation ist eine gängige Form, Netcode Szenarien, bei fehlenden Serverupdates zu bewältigen. Hierbei ist es wichtig abzuwägen und auch einzuordnen, ob der Client überhaupt noch Updates an den Server sendet oder schon von einem Verbindungsabbruch des Clients die Rede ist. In diesem Zeitraum jedoch soll geschätzt werden, anhand der letzten bekannten Eingabewerte, wo sich der Client als nächstes befinden wird. (Hier der stuff mit Unity am besten...) => wahrscheinlich bessere und ausführlichere Erkläreung...

alternativ: Lag Compensation bezeichnet eine Gruppe von Verfahren zur Abmilderung negativer Effekte instabiler Netzwerkverbindungen, insbesondere erhöhter Latenz, Paketverlust und Jitter. Ziel dieser Mechanismen ist es, trotz solcher Störungen

eine möglichst kohärente und faire Spielerfahrung sicherzustellen – idealerweise ohne spürbare Vor- oder Nachteile für einzelne Clients. Die vorliegende Arbeit fokussiert sich auf Lag Compensation im Kontext der Bewegungssynchronisation. Eine klare Abgrenzung ist dabei zur Lag Compensation bei Treffererkennung (z.B. rückwirkende Hit-Prüfung bei Schusswaffen) erforderlich, wie sie insbesondere in kompetitiven Shootern zum Einsatz kommt. Während dort meist serverseitige Backtracking-Verfahren genutzt werden, steht hier die clientseitige Kompensation von Bewegungslatenz im Vordergrund. Zur Untersuchung der Mechanismen wurde eine lokale Testumgebung in Unity entwickelt, in der sich Netzwerkanomalien wie Latenz, Paketverlust und Jitter gezielt simulieren lassen. Diese bildet die Grundlage für die Validierung verschiedener Kompensationsstrategien. Technisch lässt sich Lag Compensation als Oberbegriff für sämtliche Verfahren verstehen, die Zustandsverzögerungen kaschieren oder korrigieren – darunter Client-Side Prediction, Reconciliation, Interpolation und Extrapolation. Im Kontext der Bewegungssynchronisation kommt insbesondere Extrapolation zum Einsatz, also die Vorwärtsschätzung eines Objektzustands bei verzögerten oder ausbleibenden Serverupdates. Hierbei wird – basierend auf den zuletzt bekannten Zuständen wie Position, Rotation und Geschwindigkeit – eine plausible Fortsetzung der Bewegung berechnet. Dies erlaubt es, Objekte auf dem Client weiterhin konsistent darzustellen, obwohl keine aktuellen Daten vom Server vorliegen. Gleichzeitig muss jedoch berücksichtigt werden, ob tatsächlich nur temporär keine Updates empfangen werden oder ob bereits ein dauerhafter Verbindungsabbruch vorliegt, da sich daraus unterschiedliche Handlungsstrategien ergeben. Das in Unity implementierte System nutzt daher eine Kombination aus Latenzerkennung, Timeout-Steuerung und extrapolierender Bewegungsschätzung, um unter kontrollierten Störbedingungen möglichst flüssige Bewegungsdarstellungen zu ermöglichen.

6.7 Funktionsweise der Extrapolation

Ob überhaupt extrapoliert werden soll, also ob überhaupt eine Rekonstruktion stattfinden soll, wird analog zur Reconciliation in einer Methode: `ShouldExtrapolate()` beurteilt. Die Bedingung ist hierbei, ob die Latenz der aktuellsten Payload kleiner als das definierte Latenzlimit in Millisekunden ist, während sie gleichzeitig größer als das Zeitintervall der physikalischen Aktualisierungsfrequenz (FixedUpdate) ist,

sodass eine Extrapolation notwendig und zugleich noch sinnvoll durchführbar erscheint. Sinnvoll ist eine Extraploation genau dann, wenn die Latenz des Spielers in einem Bereich von ungefähr 3-500ms liegt. In diesem Intervall kann man also von einem Client ausgehen, der unter normalen Konditionen mit dem Server verbunden ist.

hier könnte der code sein:

Ähnlich wie im Fall der Reconciliation wird auch bei der Extrapolation ein Cooldown-Timer verwendet. Dieser sorgt dafür, dass bei ausbleibenden Eingaben des Clients – etwa infolge von Paketverlust oder temporärer Verbindungsinstabilität – weiterhin eine plausible Vorwärtsschätzung der Bewegung vorgenommen wird. Der Zeitraum der Extrapolation ist dabei auf maximal fünf Sekunden begrenzt.

Die Einführung eines Timers bietet mehrere Vorteile: Zum einen kann mit einer konstanten angenommenen Latenz extrapoliert werden, was zu einer stabileren und nachvollziehbareren Bewegungsschätzung führt. Zum anderen wird vermieden, dass bei längerer Unterbrechung der Verbindung unkontrolliert oder unplausibel extrapoliert wird.

Gleichwohl stellt die Wahl der Extrapolationsdauer einen Kompromiss dar. Zu kurze Zeitfenster führen dazu, dass kurzfristige Aussetzer nicht abgefangen werden, während zu lange Zeiträume die Gefahr von Positionsabweichungen erhöhen. Im Idealfall müsste die zulässige Extrapolationsdauer adaptiv an die konkreten Netzwerkverhältnisse des jeweiligen Clients angepasst werden.

Obwohl eine adaptive Anpassung der Extrapolationsdauer an individuelle Netzwerkbedingungen denkbar wäre, hat sich in der Praxis die Verwendung eines festen Grenzwerts als robuster und konsistenter erwiesen. Eine dynamische Lösung würde zusätzlichen Overhead erzeugen, ohne signifikante Vorteile in der Bewegungsgenauigkeit zu garantieren.

6.8 Datenstrukturen

Als Datenstruktur wird der bereits bestehende StatePayload verwendet und jedlglich um ein neues Feld von diesem Typen ergänzt. Zusätzlich wird auf das in 6.1 angepasste (Klasse) ClientNetworkTransform-Modell für die Authoritätenänderung zugegriffen. Diese Komponente wird benötigt um während der Extrapolation variabel die Authoriät zu ändern.

6.8.1 Timer

Spezifisch werden für die Extrapolation CountdownTimer verwendet die inklusive Events, die diesen Ablauf regulieren, in einem ausgelaberten Hilfs-Namespace zu finden. Dieser Ablauf wird in der Unity Hook Awake gesteuert.

6.9 Datenstrukturenrr

Für die Extrapolation wird die bestehende Datenstruktur `StatePayload` verwendet, die um ein zusätzliches Feld zur Speicherung des extrapolierten Zustands erweitert wird. Darüber hinaus greift das System auf die in 6.1 beschriebene Klasse `ClientNetworkTransform` zurück, welche in modifizierter Form eingesetzt wird, um eine dynamische Änderung der Autorität während der Extrapolation zu ermöglichen.

6.9.1 Timerrr

Zur Steuerung der Extrapolation werden spezielle `CountdownTimer`-Instanzen verwendet, die im Rahmen eines Hilfs-Namespaces implementiert sind. Diese Timer unterstützen Events zur gezielten Regelung des Ablaufs. Die Initialisierung erfolgt in der Unity-Hook `Awake`.

6.10 Kernmethoden der Extrapolation

Da Extrapolation eine serverseitige Mechanik ist, muss eine Prüfung stattfinden, welche Instanz gerade die Eingabebearbeitung ausführt. Nur in genau dem Fall, dass die Verabeitung vom Server vom Host durchgeführt wird und der Cooldown für die

Extrapolation gerade am laufen ist, wird also die Methode `Extrapolate()` aufgerufen.

In dieser Methode wird die Position des Transforms des Character Controllers auf die zuvor berechnete Position aus der Extrapolation-Payload gesetzt. Dabei wird die Y-Koordinate explizit auf null gesetzt, um vertikale Bewegungen – etwa durch physikalische Effekte oder fehlerhafte Schätzungen – auszuschließen und eine stabile Bewegung ausschließlich in der XZ-Ebene sicherzustellen. Dieser beschriebene Ablauf wird mit 2 Methoden bewältigt:

```
void Extrapolate()
{
    if (IsServer && _extrapolationCooldown.IsRunning)
    {
        transform.position = _extrapolationState.position.With(y: 0);
    }

void HandleExtrapolation(StatePayload latest, float latency) {
    if (ShouldExtrapolate(latency)) {
        if (_extrapolationState.position != default) {
            latest = _extrapolationState;
        }

        var posAdjustment = latest.velocity *
            (1 + latency * extrapolationMultiplier);
        _extrapolationState.position =
            latest.position + posAdjustment;
        _extrapolationState.rotation = latest.rotation;
        _extrapolationState.velocity = latest.velocity;
        _extrapolationCooldown.Start();
        } else {
            _extrapolationCooldown.Stop();
        }
    }
}
```

Listing 6.7: Extrapolation

Extrapolationsvektor (posAdjustment)

Zur Schätzung einer zukünftigen Position bei fehlenden Netzwerkupdates wird die zuletzt bekannte Geschwindigkeit $\vec{v}$ mit einer gewichteten Zeitkomponente multipliziert. Diese ergibt sich aus der gemessenen Latenz λ und einem anpassbaren Multiplikator α zur Feinsteuerung der Extrapolation:

$$\vec{d} = \vec{v} \cdot (1 + \lambda \cdot \alpha)$$

Dabei ist:

- $\vec{d}$ der extrapolierte Bewegungsvektor (engl. *posAdjustment*),
- $\vec{v}$ die letzte bekannte Geschwindigkeit des Objekts,
- λ die Latenzzeit in Sekunden,
- α ein konfigurierbarer Multiplikator (typischerweise $\alpha = 1.2$).

Die so berechnete Bewegung wird zur letzten bekannten Position $\vec{p}_{\text{alt}}$ addiert, um die extrapolierte Zielposition $\vec{p}_{\text{neu}}$ zu erhalten:

$$\vec{p}_{\text{neu}} = \vec{p}_{\text{alt}} + \vec{d}$$

Beispiel

Gegeben sei eine Geschwindigkeit $\vec{v} = (2, 0, 1)$ in Metern pro Sekunde, eine Latenz von $\lambda = 0{,}3$ Sekunden sowie $\alpha = 1{,}0$. Daraus ergibt sich:

$$\vec{d} = (2, 0, 1) \cdot (1 + 0{,}3 \cdot 1{,}0) = (2, 0, 1) \cdot 1{,}3 = (2{,}6, 0, 1{,}3)$$

Die Position des Objekts kann damit während einer Netzwerkunterbrechung um ca. $2{,}6\,\text{m}$ in x- und $1{,}3\,\text{m}$ in z-Richtung extrapoliert werden.

Mit der gegebenen Latenz wird entweder für die volle Zeitspanne des Cooldown Timers extrapoliert, oder es wird abgebrochen, falls die Latenz über oder unterhalb der Schwellen liegt.

6.11 Lag Compensation beim Schießen

Als Basis für die Implementierung der Lag Compensation im Kontext der Trefferer-kennung wird ein separates System genutzt, das sich von jenem zur Spielerbewegung unterscheidet. Theoretisch wäre es möglich, sowohl die Bewegung als auch die Schussmechanik in einem gemeinsamen Modul — etwa innerhalb eines übergreifenden `PlayerController`-Skripts — zu integrieren, welches weiterhin auf dem bereits etablierten Payload-System basiert. Diese Koppelung wurde jedoch, wie bereits in Abschnitt 6.1 erläutert, bewusst vermieden. Stattdessen wird auf einem dedizierten `Gun`-Skript aufgebaut, das für die grundlegende Waffenlogik, Treffererkennung, sowie die clientseitige Wiedergabe von Waffenanimationen und Audiosignalen zuständig ist. Im Vergleich zur Bewegungsmechanik erscheint die technische Umsetzung hier zunächst einfacher. In einem vollständig einsatzfähigen Mehrspieler FPS-System ist jedoch das Gegenteil der Fall: Die Korrektheit der Trefferregistrierung ist ein zentrales Element für das Spielgefühl. Besonders in Verbindung mit Lag Compensation stellt diese Komponente eine erhebliche Herausforderung dar. In realistischen Netzwerkumgebungen gibt es zahlreiche Randbedingungen und Einflussgrößen, die nicht ohne weiteres in ein allgemeingültiges System integriert werden können, ohne erhebliche Performanceeinbußen zu riskieren.

In älteren Titeln wie *Quake* oder *Doom* war die Entscheidung, ob überhaupt eine Lag Compensation implementiert wird, ein wesentlicher Bestandteil des Spieldesigns — eine Option, die in modernen Spielen kaum mehr denkbar ist. Gerade in Shootern stellt Lag Compensation heute ein zentrales Mittel dar, um ungleiche Netzwerkverhältnisse zwischen Clients bestmöglich auszugleichen. Ohne entsprechende Kompensation wären Spieler mit hoher Latenz insbesondere in Bewegung nur schwer bis gar nicht zuverlässig zu treffen, da die Differenz zwischen dem serverseitigen Zustand und dem, was der Client darstellt, zu groß wird. Dies führt zu einer teils erheblichen Verschiebung der effektiven Hitbox und damit zu unnatürlichen Spielerlebnissen.

Der Grundsätzliche Ablauf lässt sich wie folgt ableiten:

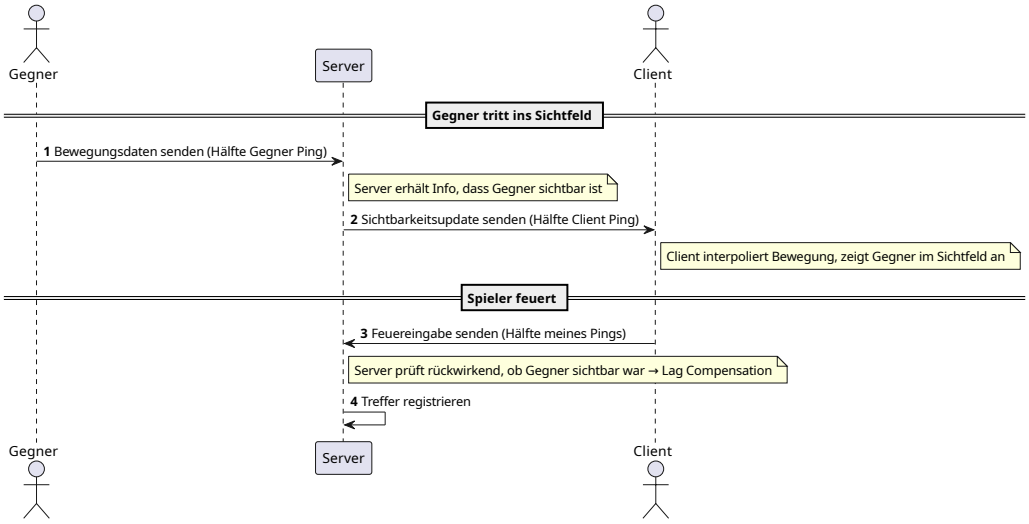

Abbildung 6.3: Visualisierung der Lag Compensation

Mit dieser Mechanik lässt sich zwar kein sauberes Spielgefühl erzeugen und sorgt immernoch nicht für ein faires Spielverhältnis wenn die involvierten Spieler nicht ähnlich hohe Latenzen haben. Jedoch ist dieses Verfahren das weitverbreiteste in der Branche, was bislang noch keinen würdigen Nachffolger erhalten hat, der die durch Lag Compensation entstandenen Probleme löst. Die klassischen Probleme sind hierbei:

- Auf einen Client registrieren Treffer, obwohl er in Deckung ist
- Gegner registriert Schüsse auf einen Client obwohl er für den Client nicht sichbar ist
- Peeker's Advantage
- Killtrades

Die beschriebene Problematik ist nicht immer vollständig auf Lag Compensation zurückzuführen, jedoch bevorzugt sie in den meisten Fällen den Spieler mit dem geringeren Ping. In einem Szenario, in dem beide Spieler bspw. gleichzeitig aufeinander

schießen und treffen würden – was im Kontext von Netcode nahezu paradox ist – würde allerdings die Bestätigung für den Treffer oder Abschuss, den Spieler mit dem geringeren Ping früher erreichen. Das wäre für die meisten Entwickler kein ideales aber dennoch faires Szenario, denn ein niedriger Ping sollte zumindest keinen Nachteil einbingen. In manchen Fällen und Spielen wird hierfür überkompensiert, sodass höherer Ping im Bereich von 60-80ms vorteilhaft ist – vor allem beim Initiieren eines Duels (peeking). In dieser Implementierung, wie es bei Valorant der Fall ist, muss auf bestimmte Ping-Bereiche eingegangen sein und mit variablen Multiplikatoren gearbeitet worden sein, um somit für die meisten Szenarien diesen Vorteil (Peeker's Advantage) zu minimieren [13]. Es gibt somit keinen Idealzustand für eine Integration von beliebten Netcode-Mechaniken, die für direkte Client-Eingabe sorgen und gleichzeitig ein zeitlich faires und korrektes System stellen.

Im Rahmen der Arbeit wird jedlglich versucht diese Problematik zu visualisieren, da die Grundlage eine andere ist. In einem Szenario in dem es nur einen Spieler und ein vom Server gesteuertes Zielobjekt gibt, ist die Komplexität und Signifikanz der Lag Compensation geringer. In diesem Fall kann also keine vollständige Implementierung von einer Treffererkennung und Lag Compensation gezeigt werden wie sie in PvP-Spielen vorkommt, allerdings wird ein Modell gezeigt, dass versucht diesen Sachverhalt unter den genannten Einschränkungen zu veranschaulichen.

6.11.1 Aufbau der Replikation

Um in Unity Lag Compensation beim Schießen darzustellen, ist ein bewegliches Ziel im Einsatz, welches intern wieder ein 3D-Objekt (Sphere) als sogenanntes Enemy-Prefab eingebunden ist. Das Prefab ist außerdem ein Network Objekt, allerdings ohne eines Network Transfomrs für die Synchronisation, welche in diesem Kontext selbstständig implementiert ist. Dieses Ziel soll die horrizontale Bewegung eines echten Spielers replizieren, allerdings über den Server gesteuert. Das bedeutet wiederum, dass das zuständige Target-Skript nur auf dem Server ausgeführt werden darf und wird. (hier noch überlegen, ob das Base-Skript wo anders erklärt werden soll). Somit wird hierbei auschließlich von der Eigenschaft `IsServer` Gebrauch gemacht, um sicher zu stellen, dass wirklich nur der Server oder der Host das Skript ausführen.

Das Skript ist somit in erster Linie für das Bewegen des Zielobjektes zuständig. Außerdem soll das Ziel, wenn es durch die Waffe an ausreichend Schaden genommen hat, an einem zufälligem Punkt in im Spawnfeld erneut erscheinen. Hierbei werden

Klone dieses Prefabs über das Netzwerk gespawned oder despawned. Dieser Vorgang ist vor allem wichtig für die Synchronisation, da ein Zerstören des Objekts zu einer ungewollten Nebenwirkung führt.

Im Grunde sind im high level Ablauf des geschilderten Szenarios drei Skripte involviert:

- Target-Skript: Bewegung und Spawnausführung
- Spawner-Skript: Spawnlogik des Zielobjektes
- Gun-Skript: Schadenverwaltung und Effekte

Es werden in diesen Skripten somit für die Integration der Lag Compensation essentielle Funktionalitäten gestellt. Diese Funktionalitäten sind jeoch nur ein Mindestansatz für FPS-Spiele, werden jedoch durch die Ergänzung der Visualisierung der Lag Compensation dringend benötigt.

6.11.2 Integration der Lag Compensation

wie bereits in 5.2.2 angesprochen ist der Hauptbestandteil hierbei das Zusammenspiel des Waffen- und Zielskriptes. In diesen beiden Skripten ist somit geteilte Logik enthalten und wird mit ähnlichem Aufbau wie bei der Bewegungssemenatik implementiert.

Um die Effekte besser darstellen zu können, kommen auf Client-Seite Feedback-Mechanismen zum Einsatz. Diese wurden teilweise schon in 4.2 erwähnt. Hierbei geht es in erster Linie darum, die Dynamik von Lag Compensationsation beim Schießen mit erhöhter Latenz zu visualisieren.

Wenn ein stehendes Ziel getroffen wird, gibt es keinen auffälligen Effekt, da sich höchstens der Zeitpunkt des vom Server registrierten Treffers vom eigentlichen Treffer auf Client-Seite unterscheidet und somit ein zeitlich versetzter Hitmarker-Effekt und Sound abspielt. Abgesehen davon wird das Ziel farblich für einen Frame geändert um das Server-Feedback nochmals besser zu zeigen.

Bei einem Treffer eines sich bewegenden Ziels jedoch, ist ein zeitlich versetzter Treffer erkenntlich. Hierbei unterscheiden sich die States von Server und Client beim Treffer, sodass der farblich eingeblendete Rewind eine andere Position als das Ziel zum aktuellen Standpunkt wie er beim Client angezeigt wird, wiederspielgelt. . . . (not

too sure about that one). In allen Fällen wird das Zielobjekt zusätzlich farblich markiert (was halt eigentlich nicht sonderlich viel Sinn macht...)

Wird ein Treffer durch Lag Compensation auf dem Server bestätigt, so wird ausschließlich das aktuell sichtbare Zielobjekt (z.B. durch kurzfristiges Einfärben) visuell markiert. Die tatsächliche zurückgerechnete Trefferposition bleibt für den Spieler unsichtbar und dient nur der korrekten, latenzunabhängigen Treffererkennung im Hintergrund.

Fundamental ist eine trivialere Form der Ringpuffer wie in (ref) im Einsatz. Es werden bei der Srtruktur jediglich die aktuelle Position und ein Zeitstempel mitgegeben. Das bietet sich vor allem auf Grund eines vereinfachten Szenarios an, indem zwar grundlegend Ähnlichkeiten zur bereits verwendeten Architektur verwendet werden, allerdings im Bezug auf die Komplexität der Kommunikation nicht nötig ist den gleichen Ansatz zu wählen.

```
1  public class Target : NetworkBehaviour, IDamageable
2  {
3      private struct BufferState
4      {
5          public Vector3 Position;
6          public double Timestamp;
7      }
8      private readonly List<BufferState> _buffer = new();
9      private const float BufferTime = 1.0f;
10
11     private void FixedUpdate()
12     {
13         if (!IsServer) return;
14
15         _buffer.Add(new BufferState
16         {
17             Position = transform.position,
18             Timestamp = NetworkManager.ServerTime.Time
19         });
20
21         _buffer.RemoveAll(state => NetworkManager.ServerTime.Time -
22                                    state.Timestamp > BufferTime);
23     }
24 }
```

LISTING 6.8: Methode GetRewindPosition

Mit diesen kürzeren Payloads wird somit der Informations- Abgleich und Austausch zwischen Client und Server getätigt. Diese Struktur befindet sich somit mit der Logik für Lag Compensation und Synchronisation innerhalb der Klasse **Target**. In dieser Klasse werden somit die benötigten Datenstrukturen angelegt und variabel befüllt oder gelöscht. Diese Operationen geschehen ausschließlich in fixen Zeitschritten in der dafür vorgesehenen Hook: FixedUpdate.

Das Befüllen des Puffers findet zu jedem Zeitsschritt statt und speichert die aktuelle Position des lokalen Transforms und den Zeitstempel der lokalen Serverzeit. Analog wird das Löschen der Daten innerhalb des Puffers mit der Methode **RemoveAll()** getan. Wenn hierbei mehr als eine Sekunde zwischen serverseitigem Zeitstempel und dem aktuellen Zeitstempel liegen, wird diese Methode aufgerufen.

Im Bereich der relevanten Funktionalität der Lag Compensation ist eine Methode bedeutsam, welche die Position des Treffers wiedergibt, an dem der Server diesen registriert hat.

```
public Vector3? GetRewindPosition(double timestamp)
{
    BufferState? older = null, newer = null;

    foreach (var state in _buffer)
    {
        if (state.Timestamp <= timestamp)
            older = state;
        else if (state.Timestamp > timestamp)
        {
            newer = state;
            break;
        }
    }
    if (older.HasValue && newer.HasValue)
    {
        float t = (float)((timestamp - older.Value.Timestamp) /
        (newer.Value.Timestamp - older.Value.Timestamp));

        return Vector3.Lerp(older.Value.Position,
                            newer.Value.Position, t);
    }
    return older?.Position;
}
```

Listing 6.9: Methode GetRewindPosition

Der Aufruf dieser Methode erfolgt innerhalb der Klasse, die für die Waffenlogik zuständig ist. Um die prinzipielle Serverautorität sicherzustellen, wird die Methode explizit als ServerRPC ausgeführt. Die zentrale Aufgabe von `ShootServerRpc` besteht darin, einen vom Client ausgelösten Schuss serverseitig entgegenzunehmen und unter Berücksichtigung möglicher Netzwerklatenz korrekt zu verarbeiten. ServerRPCs sind hierfür essenziell, da sie gezielt ermöglichen, spielrelevante Aktionen wie Schüsse oder Trefferprüfungen nicht lokal auf dem Client, sondern ausschließlich auf dem Server auszuführen.

Nur so kann garantiert werden, dass die Treffererkennung manipulationssicher und für alle Spieler konsistent abläuft. Der Server übernimmt dabei die gesamte Logik der Treffererkennung: Er identifiziert das angezielte Objekt und berechnet - abhängig von der aktivierten Lag Compensation - entweder die zurückgerechnete Position des Ziels zum Schusszeitpunkt oder verwendet die aktuelle Position.

Im Anschluss prüft der Server, ob der Schuss ein gültiges Ziel getroffen hat und löst bei einem Treffer die weiteren Schritte wie Schadensberechnung und visuelles Feedback auf den beteiligten Clients aus.

Im Kontext der Treffererkennung werden ebenfalls RPCs benötigigt um allerdings die Bestätigung der Treffer weiterzuleiten. Somit wird hier ein ClientRpc benötigt der vom Server initiiert wird. Diese ClientRPCs werden allerdings nur ausgeführt, wenn die Lag Compensation aktiviert ist und erlaubt somit die zeitliche Rückrechnung um einen Treffer auch auf Serverseite zu bestätigen.

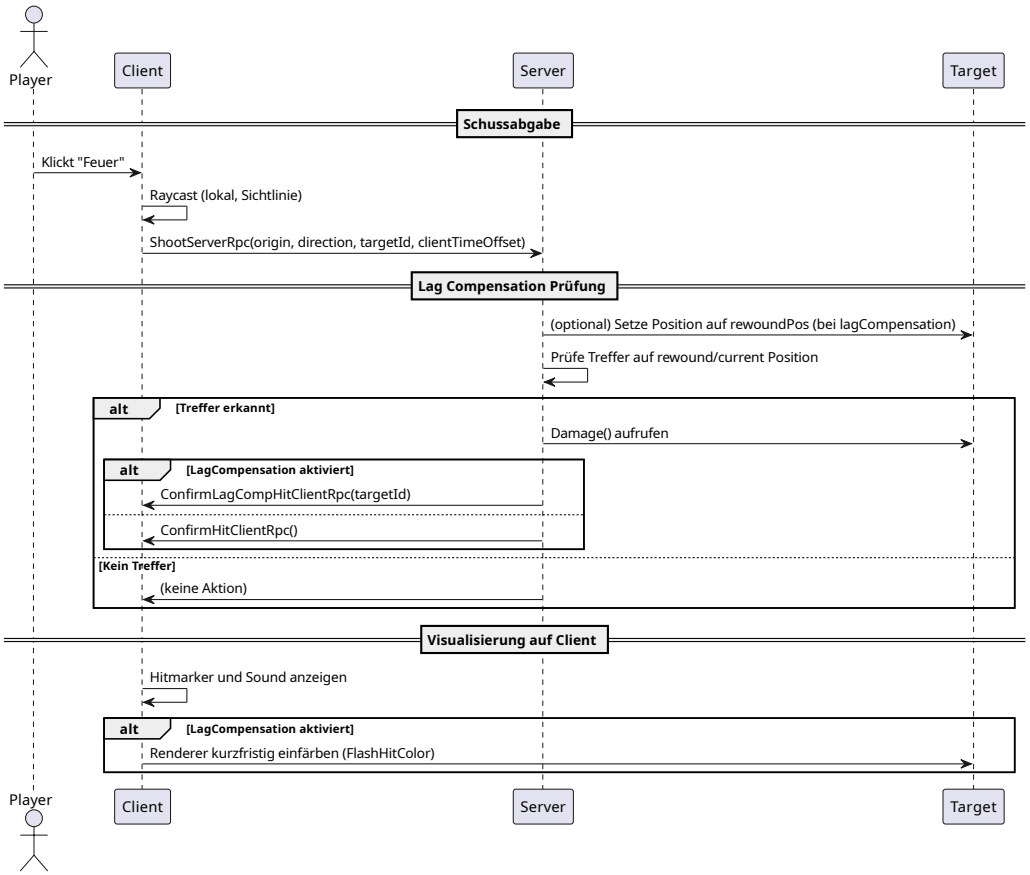

Abbildung 6.4: Visualisierung der Lag Compensation im Projekt

Im Sequenzdiagramm 6.4 ist der Ablauf der Lag-Compensation-Implementierung im Projekt dargestellt. Die Lag Compensation kann dabei gezielt aktiviert oder deaktiviert werden, um die Auswirkungen dieses Mechanismus zu untersuchen. Bei deaktivierter Lag Compensation erfolgt die Treffererkennung ausschließlich auf Basis der aktuellen Zielposition, während bei aktivierter Lag Compensation die Position des Ziels zum Schusszeitpunkt rekonstruiert und für die Trefferprüfung herangezogen wird. Dies wirkt sich nicht nur auf die Visualisierung, sondern insbesondere auf die Genauigkeit und Fairness der Treffererkennung aus. Beispielsweise wird bei aktiver Lag Compensation im Falle eines bestätigten Treffers das Zielobjekt hervorgehoben, um den Unterschied für den Spieler erkennbar zu machen.

6.12 Interpolation

Interpolation ist in den meisten Multiplayer-Spielen unverzichtbar. Ohne sie wäre auch eine funktionierende Implementierung der Lag Compensation weitgehend wirkungslos, da die für den Client sichtbare Position des Zielobjekts keine ausreichende Aussagekraft hätte. Wenn ausschließlich zum jeweiligen Servertick eine Positionsaktualisierung an die Clients übertragen wird, kann es insbesondere bei niedrigen Tickraten zu deutlichen Ungenauigkeiten und ruckelnden Bewegungsabläufen kommen – vergleichbar mit den Effekten, die auch ohne Lag Compensation auftreten. Beispielsweise würde bei einer Tickrate von unter 10 Hz ein Spieler oder Zielobjekt nicht oft genug aktualisiert, um eine flüssige Bewegung darstellen zu können.

In den meisten modernen Spielen liegt die Tickrate allerdings deutlich höher, sodass das Fehlen von Interpolation nicht sofort als gravierendes Problem auffällt. Bei einer im Projekt verwendeten Tickrate von 60 Hz sind Unterschiede in der Bewegungsglättung für den Nutzer meist nur minimal erkennbar. Dennoch bleibt Interpolation ein wesentlicher Baustein für eine konsistente und angenehme Spielerfahrung.

Gerade im Hinblick auf Desynchronisation zwischen Server und Client spielt Interpolation eine entscheidende Rolle: Sie sorgt dafür, dass auch bei kleinen Verzögerungen oder Paketverlusten Bewegungen zwischen den übermittelten Serverpositionen sinnvoll und flüssig fortgeführt werden können. Ohne Interpolation würden solche Desynchronisationen unmittelbar als Sprünge oder Ruckler wahrgenommen werden, wohingegen eine kontinuierliche Interpolation zwischen den bekannten Zuständen die Auswirkungen von Latenz und Datenverlust für den Spieler deutlich abmildert.

Extrapolation und Interpolation ergänzen sich in modernen Multiplayer-Spielen und gelten als Gegenstücke: Während Interpolation die Bewegungen zwischen bekannten Server-Updates für den Client glättet, sorgt Extrapolation dafür, dass bei ausbleibenden Netzwerkpaketen dennoch eine plausible Fortsetzung der Bewegung berechnet wird. Bei einer durchdachten Kombination dieser beiden Verfahren bleibt der Spielfluss auch unter kritischen Netzwerkbedingungen, wie etwa hoher Latenz oder Paketverlusten, für den Spieler möglichst flüssig und konsistent.

In vielen Engines, wie beispielsweise der Source Engine von Valve, ist die Lag Compensation eng mit der Interpolationsdauer verzahnt. Der Wert der Interpolation bestimmt, wie weit die Darstellung des Spielgeschehens auf dem Client hinter den

tatsächlich vom Server berechneten Zuständen zurückliegt. Eine längere Interpolationsdauer bedeutet, dass mehr verlässliche Zustände für die Glättung zur Verfügung stehen, jedoch erkauft man sich dies mit einer erhöhten „künstlichen Latenz“. Letztere bezeichnet den zusätzlichen zeitlichen Versatz, der ausschließlich durch das Interpolieren entsteht und auf den Client wirkt, unabhängig von der eigentlichen Netzwerkverbindung.

Der Interpolationspuffer stellt eine gezielte, zusätzliche Verzögerung („künstliche Latenz“) in der Clientdarstellung dar. Diese kommt zu den eigentlichen Netzwerk- und Verarbeitungsverzögerungen hinzu und wird bewusst gewählt, um durch Interpolation stets eine ausreichend flüssige und konsistente Bewegung zwischen empfangenen Serverzuständen gewährleisten zu können Gerade in latenzkritischen Spielen ist es daher essenziell, die Interpolationsrate so niedrig wie möglich zu halten, um jede vermeidbare Quelle zusätzlicher Verzögerung zu minimieren. Gleichzeitig darf die Interpolationsdauer aber auch nicht zu gering gewählt werden, da sonst abrupte Positionssprünge oder Ruckler auftreten können, falls einzelne Serverpakete verloren gehen.

Eine adaptive Lag Compensation, die sich dynamisch an die gewählte Interpolationsdauer anpasst, kann hier Abhilfe schaffen. Sie gewährleistet, dass trotz unterschiedlich langer Pufferzeiten zwischen Server und Client eine möglichst faire und präzise Treffererkennung möglich bleibt. Insgesamt zeigt sich, dass das geschickte Zusammenspiel von Interpolation und Extrapolation – unter Berücksichtigung der mit ihnen einhergehenden Vor- und Nachteile – ein zentrales Element für eine stabile und flüssige Spielerfahrung im Online-Multiplayer darstellt [14].

6.12.1 Eigenumsetzung

Für die eigene Umsetzung dieser Mechanik ist in erster Linie nicht der Spieler an sich betroffen, wessen Positionen oder Rotationen interpoliert werden. Hierbei geht es vielmehr um das Ziel, auf welches man schießt und hängt demnach teilweise von der in 6.11.2 implementierten Lag Compensation zusammen.

Prinzipiell besteht in Unity die Möglichkeit, das vorhandene `NetworkTransform` entweder direkt zu verwenden oder es durch Vererbung zu erweitern und gezielt anzupassen. In beiden Fällen bleibt jedoch die umfangreiche und generische Grundstruktur der Unity-Implementierung erhalten, was zu einem gewissen Overhead und einer erhöhten Komplexität führt. Auch durch partielle Anpassungen per Vererbung kann die eigentliche Komplexität des Systems selten wesentlich reduziert werden. Für klar umrissene Anforderungen, wie sie im Rahmen dieses Projekts bestehen, ist daher eine speziell zugeschnittene Eigenimplementation oft sinnvoller, da sie sich gezielt auf die benötigten Funktionen konzentriert und unnötigen Funktionsumfang vermeidet.

Mit dieser Umsetzung können vergleichsweise wenige Einstellungen vorgenommen werden, die mit der Interpolation durch das Network Transform von Unity möglich sind. Hier kann dennoch die Senderate, die Verzögerung der Interpolation, das Positionslimit und die Möglichkeit die Interpolationsfunktionalität aus- oder einzuschalten.

Somit lassen sich Feinjustierungen vornehmen oder die Effekte durch Abschalten dieser Mechanik erkenntlicher machen.

6.12.2 Kernfunktion

Die ausschlaggebende Funktion in diesem Szenario, kümmert sich um die clientseitige Interpolation und glättet somit Positions- und Rotationsdarstellung bei sich bewegenden Objekten.

```
private void Interpolate(float renderTime)
{
    while (_stateBuffer.Count >= 2)
    {
        State prev = _stateBuffer.Peek();
        State next = default;

        foreach (var state in _stateBuffer)
        {
            if (state.timestamp > renderTime)
            {
                next = state;
                break;
            }
            prev = state;
        }

        float t = Mathf.InverseLerp(prev.timestamp,
                                    next.timestamp,
                                    renderTime);

        if (Vector3.Distance(transform.position,
                             next.position) > positionThreshold)
            transform.position = Vector3.Lerp(prev.position,
                                              next.position, t);

        transform.rotation = Quaternion.Slerp(prev.rotation,
                                              next.rotation, t);
        return;
    }
    ApplyLatestState();
}
```

LISTING 6.10: Methode `Interpolate()`

Die Methode `Interpolate(float renderTime)` sorgt dafür, dass Positions- und Rotationsänderungen eines Objekts auf dem Client auch bei unregelmäßig eintreffenden Server-Updates flüssig dargestellt werden. Zu diesem Zweck wird in einem lokalen Buffer eine Folge von Bewegungszuständen (*States*) gespeichert, die jeweils einen Zeitstempel, eine Position und eine Rotation enthalten. Die Methode sucht stets die beiden States im Buffer, zwischen deren Zeitstempeln der gewünschte Anzeigemoment (`renderTime`) liegt. Mit Hilfe eines Interpolationsfaktors wird dann eine Position und Rotation berechnet, die zeitlich genau zwischen diesen beiden Punkten liegt. So entsteht für den Spieler der Eindruck einer kontinuierlichen Bewegung, selbst wenn neue Bewegungsdaten nur in größeren Abständen eintreffen. Sollte keine Interpolation möglich sein, etwa weil zu wenige States vorliegen, wird stattdessen einfach der letzte bekannte Zustand übernommen.

...

Kapitel 7

Fazit

Literatur

[1] *Unity Runtime Fee – Unity*. Zugriff am 21. Mai 2025. URL: https://unity.com/pricing-updates.

[2] *Unreal Engine Licensing - Unreal Engine*. Zugriff am 21. Mai 2025. URL: https://www.unrealengine.com/en-US/license.

[3] *Godot Engine – Lizenzmodell*. Zugriff am 21. Mai 2025. URL: https://godotengine.org/license.

[4] CD Project Red. *The Witcher 4 — Unreal Engine 5 Tech Demo*. Letzter Zugriff: 25.06.2025. 2025. URL: https://www.thewitcher.com/us/en/news/51521/the-witcher-4-unreal-engine-5-tech-demo.

[5] Scott Kennedy. *How Epic is integrating Niagara into Fortnite*. https://www.unrealengine.com/de/tech-blog/how-epic-is-integrating-niagara-into-fortnite?. Letzter Zugriff: 21.05.2025. 2019.

[6] *FishNet Documentation*. Zugriff am 21. Mai 2025. URL: https://fish-networking.gitbook.io/docs/.

[7] *Range*. letzter Zugriff: 02.07.2025. URL: https://valorant.fandom.com/wiki/Range.

[8] aimlabs. *master your aim dominate the game*. letzter Zugriff: 02.07.2025. 2025. URL: https://aimlabs.com/.

[9] Unity. *Coroutines*. Zugriff am 24.07.2025. 2025. URL: https://learn.unity.com/tutorial/coroutines.

[10] microsoft. *out (csharp Reference)*. 2025. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out.

[11] Glenn Fiedler. *Fix Your Timestep!* Zugriff am 11.07.2025. 2004. URL: https://gafferongames.com/post/fix_your_timestep/.

[12] Valve Corporation. *Lag Compensation*. Zugriff am 11.07.2025. 2010. URL: https://developer.valvesoftware.com/wiki/Lag_compensation.

[13] Anna Donlon. *On Peeker's Advantage and Ranked*. Zugriff am 21.07.2025. 2020. URL: https://playvalorant.com/en-us/news/game-updates/04-on-peeker-s-advantage-ranked/.

[14] Valve Corporation. *Interpolation*. abgerufen am 28.07.2025. 2024. URL: https://developer.valvesoftware.com/wiki/Interpolation.