Bachelorthesis

Konzeption und Umsetzung von Software-Architekturen in Multiplayer-Spielen

zur Erlangung des akademischen Grades

Bachelor of Science

eingereicht im Fachbereich Mathematik, Naturwissenschaften und Informatik an der Technischen Hochschule Mittelhessen

von

Leon Schäfer

1. März 2022

Referent: Dr. Dennis Priefer

Korreferent: Prof. Dr. Peter Kneisel

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Gießen, den 1. März 2022 Leon Schäfer

Abstract:

Die Entwicklung von Multiplayer-Spielen gestaltet sich für angehende Entwickler schwierig. Das erste eigene Projekt umzusetzen erfordert viel Einarbeitung und es gibt viele Hürden zu bewältigen.

Das Ziel dieser Arbeit ist, eine Einstiegshilfe für die Entwicklung von Multiplayer-Spiele-Projekten zu geben. Hierzu wird eine abstrakte Vorlage konstruiert, welche auf möglichst viele Multiplayer Use Cases anwendbar ist.

Dazu wurden abstrahierte Konzepte erstellt, welche auf möglichst viele Spielideen übertragbar sein sollen. Diese Konzepte bestehenden aus einzelnen Software-Komponenten, welche selbst noch keine konkrete Implementierungsvorgabe sind, sondern lediglich das Konzept der Komponente abstrakt beschreiben.

Als 'Proof of Concept' wurden die beschriebenen Konzepte anhand einer Implementierung in einem Multiplayer-Spiel in der Unity Engine, mithilfe des Mirror Frameworks implementiert.

Inhaltsverzeichnis

1 Einführung

Die Spieleindustrie ist im Wandel. Reine Singleplayer-Spiele, die ohne andere menschliche Mitspieler auskommen, werden immer seltener entwickelt. Die Erfolgsaussichten erscheinen um ein Vielfaches schlechter. Die drei bestplatzierten Titel der meistverkauften PC- und Konsolenspiele 2019 (FIFA 2020, Call of Duty: Modern Warfare und Mario Kart 8 Deluxe) in Deutschland enthalten einen großen Multiplayer-Anteil [gam20]. Deutlicher wird der Trend, wenn die Statistik der meistgespielten Titel auf der Internet Vertiebsplattform für Computerspiele Steam betrachtet wird. Das mit Abstand meistgespielte Spiel 2020 ist das Multiplayer-First-Person-Shooter Spiel 'Counter-Strike: Global Offensive' gefolgt von 'DOTA 2' auf Platz 2. Beide Spiele sind primär Multiplayer-Titel [Git21].

Einer internationalen Umfrage zufolge geben 60 % der Befragten an, seit Beginn der Corona-Pandemie mehr Multiplayer-Videospiele zu spielen [Sim20].

Wegen der Marktentwicklung ist es für angehende Entwickler, welche in der Spieleindustrie Fuß fassen möchten, auf lange Sicht eine gute Entscheidung, sich mit der Entwicklung von Multiplayer-Spielen auseinanderzusetzen.

1.1 Problembeschreibung und Motivation

Konzeption und Entwicklung von Online-Multiplayer-Spielen stellt angehende Entwickler vor große Herausforderungen. Einer der Hauptunterschiede zwischen einem Multiplayer- und einem Singleplayer-Spiel ist, dass eine Spielumgebung unabhängig von einem Spieler existiert. Das bedeutet, dass sich in der Regel bei Singleplayer-Spielen alle Software-Prozesse bei Beendigung des Spiels auch schließen, bei Multiplayer-Spielen jedoch laufen Client- und Serverprozess unabhängig voneinander. So ist es möglich, dass Serverprozesse, und somit auch Spiellogik unabhängig von anwesenden Clients (Spieler) weiterläuft und sich Zustände einer Spiel-Session verändern.

Außerdem benötigt das Versenden von Nachrichten über das Internet eine gewisse Zeit. Der Entwickler muss also herausfinden, wie man mit asynchroner Kommunikation,

Clients und Servern umgeht. Ebenso stoßen Entwickler vor Probleme wie Hosting, Matchmaking und Datenverwaltung. [Pay19]

Einheitliche Vorgehensmodelle und Best Practises gibt es nicht oder beziehen sich stets auf eine spezielle Art von Spieltyp. Das Thema Hosting ist ebenfalls ein komplexer Aspekt eines solchen Projekts, welcher von Anfängern oft nicht sofort durchschaut wird. Durch erste Projekterfahrung gewinnen Entwickler nach und nach das Knowhow. Ohne ein erfahrenes Team sind die Chancen auf Misserfolg jedoch hoch. [Pay19]

Die Motivation für diese Arbeit war es, angehenden Entwicklern eine Einstiegshilfe in die Entwicklung von Multiplayer-Spielen zu geben. Hierzu wurden Konzepte erarbeitet, die sie in ihrem ersten Praxisprojekt ausprobieren und idealerweise davon profitieren können.

1.2 Ziele dieser Arbeit

Das primäre Ziel dieser Arbeit ist es, Anfängern den Einstieg in die Entwicklung von Multiplayer-Spielen zu erleichtern.

Ein Entwickler soll außerdem nach dem Lesen dieser Arbeit eine konkrete Vorstellung davon besitzen, welche Voraussetzungen ein Framework oder eine Game-Engine erfüllen muss, damit die Konzepte möglichst leicht umzusetzen sind.

Die Arbeit soll als eine 'Blaupause' dienen, welche andere Entwickler nutzen können, um einen leichteren Einstieg in ein Multiplayer-Projekt zu gewährleisten. Diese Blaupause soll generisch sein, sodass sie unabhängig von einem konkreten Implementierungskontext funktioniert. Die beschriebenen Konzepte sollen auf möglichst viele Szenarien anwendbar sein.

Durch die Konzepte sollen verschiedene Use cases in der Multiplayer-Spieleprogrammierung abgedeckt werden. Beispiele sind das Aufbauen und Trennen von Client/Server Verbindungen, Matchmaking [Wik21g] und Spielerverwaltung.

Als 'Proof of Concept' soll ein Prototyp eines Hide-and-Seek Multiplayer-Spiels dienen, welches sich an den abstrahierten Konzepten orientiert, bzw. diese implementiert.

Folgende Forschungsfragen soll die Arbeit beantworten:

F1: Können durch die Untersuchung der beschriebenen Probleme Konzepte abgeleitet werden, um diese Probleme zu lösen beziehungsweise zu vereinfachen?

F2: Wie könnte ein Studienkonzept aussehen, welches bei Durchführung beweist, dass die in dieser Arbeit beschriebenen Konzepte einen Mehrwert bei der Entwicklung eines Multiplayer-Spiels für angehende Entwickler liefern?

1.3 Vorgehensweise

F1 soll wie folgt beantwortet werden:

Zunächst wird die Problemstellung anhand von Literatur und Internetquellen aufgezeigt und erläutert.

Aus der bisher gesammelten Praxiserfahrung und gefundenen Literatur werden Konzepte abstrahiert, welche jeweils einen bestimmten Einsatzzweck erfüllen sollen. Hierbei wurde auf Implementierungsdetails verzichtet und lediglich ein generelles Konzept erarbeitet, an welchem der Entwickler sich während des Implementierungsprozesses orientieren kann.

Jedes dieser Konzepte wurde in einem Prototyp umgesetzt, welche im Kapitel Realisierung näher vorgestellt wird.

F2 soll wie folgt beantwortet werden:

Es wird ein Studienkonzept erarbeitet, welches beschreibt, wie durch wissenschaftliche Methoden und Forschung untersucht werden kann, ob die Konzepte einen tatsächlichen Mehrwert für angehende Entwickler bietet. Das Studienkonzept findet sich im Kapitel 'Konzepte' in der Sektion 'Studienkonzept'.

1.4 Abgrenzung

Die Arbeit ist keine Schritt-für-Schritt-Anleitung für Multiplayer-Spielprojekte. Implementierungsdetails müssen von einem Entwickler, welcher diese Arbeit als Hilfsmittel nutzt, selbstständig konzipiert und umgesetzt werden. Lediglich wird auf Spielkonzepte Rücksicht genommen, welche in der heutigen Industrie gängig sind [Wik21g]. Es ist durchaus denkbar, dass zukünftige Multiplayer Architekturen nicht mehr mit den hier beschriebenen Konzepten umsetzbar sind.

Es ist zudem nicht final bewiesen, dass die beschriebenen Methoden tatsächlich einen Mehrwert bei unterschiedlichen Projekten bieten. Die Konzepte wurden lediglich an einem Prototyp getestet und validiert. Um zu testen, ob die Konzepte auch praxistauglich

für verschiedene Arten von Projekten sind, muss eine weitergehende Untersuchung erfolgen. Diese kann anhand des in dieser Arbeit beschriebenen Studienkonzept stattfinden.

Die erarbeiteten Konzepte sind unabhängig von einer Game-Engine oder einem Framework. Es aber möglich, dass ein bestehendes Framework oder eine Game-Engine Hilfestellung bei der Implementierung der Konzepte bietet, oder diese bereits als Feature bereitstellt.

1.5 Struktur der Arbeit

Diese Arbeit besteht aus der Einführung, in welcher die generelle Problemstellung erläutert, und die Forschungsfragen gestellt wurden. Ebenfalls wurde die Relevanz des Themengebiets verdeutlicht und eingeordnet.

Im Anschluss wird der Hintergrund erklärt, dort sind alle Informationen zu finden, um die späteren Konzepte zu verstehen. Hier werden ebenfalls Begrifflichkeiten und Grundlagen erläutert.

Das Kapitel 'Konzepte' behandelt nun die erarbeiteten Konzepte, welche in Summe die oben beschriebene Blaupause abbilden. Es wird auf jedes einzelne Konzept im Detail eingegangen und genauer erklärt.

Anschließend werden die beschriebenen Konzepte in einem Prototyp umgesetzt, dieser wird im Kapitel 'Realisierung' näher erklärt. In diesem Kapitel befindet sich kommentierter Beispielcode aus dem umgesetzten Prototyp, um einen besseren Eindruck der gelösten Probleme zu bekommen.

Im Kapitel 'Abschluss' werden die Ergebnisse der Arbeit beleuchtet, und ein Studienkonzept aufgeführt, welches weiterführende Forschung ermöglicht.

2 Hintergrund

In diesem Kapitel werden die Grundlagen erläutert, welche nötig sind, um die Konzepte und Realisierung genauer zu verstehen. Unter anderem wird auf die Grundlagen des Networkings, die verschiedenen Arten des Hostings, Datenspeicherung und Transfer, Informationskontrolle sowie Interest Management, Matchmaking und Synchronisation eingegangen.

2.1 Grundlagen des Networkings in Online-Videospielen

Die Grundlagen des Networkings in Videospielen lassen sich in zwei große Bereiche kategorisieren. Die physische und die logische Plattform.

Die physische Plattform setzt sich aus den physikalischen Komponenten zusammen, die vereint eine Infrastruktur bilden. Hierzu zählt Hardware, welche in Rechenzentren eingesetzt wird, das lokale Endgerät wie z. B. ein Smartphone oder Personal Computer. Ebenfalls zählen Kabelleitungen und drahtlose Übertragungswege dazu.

Online-Videospiele sind aus technischer Sicht auch nichts anderes als Anwendungen, welche miteinander kommunizieren. Die physischen Restriktionen wie Bandweite und Latenz können also ebenfalls auf den Kontext der Multiplayer-Spiele angewendet werden. Die Menge an Informationen, welche über ein Netzwerk versendet werden, kann je nach Spieltyp sehr hoch skalieren, weshalb sich die Entwickler eines Multiplayer-Spiels intensiv mit der logischen Plattform beschäftigen müssen.

Die logische Plattform baut auf der physischen Plattform auf und nutzt ihre Ressourcen. Sie kann unterteilt werden in Kommunikation zwischen Clients und Server, Datenspeicherung und Transfer sowie Kontrolle des Spielflusses. Auf diese drei Punkte wird in den folgenden Sektionen näher eingegangen.

[Sme02b]

2.2 Verschiedene Arten des Hostings

Die Kommunikation zwischen Clients und Server kann in zwei verschiedenen Varianten vorkommen:

Client Host:

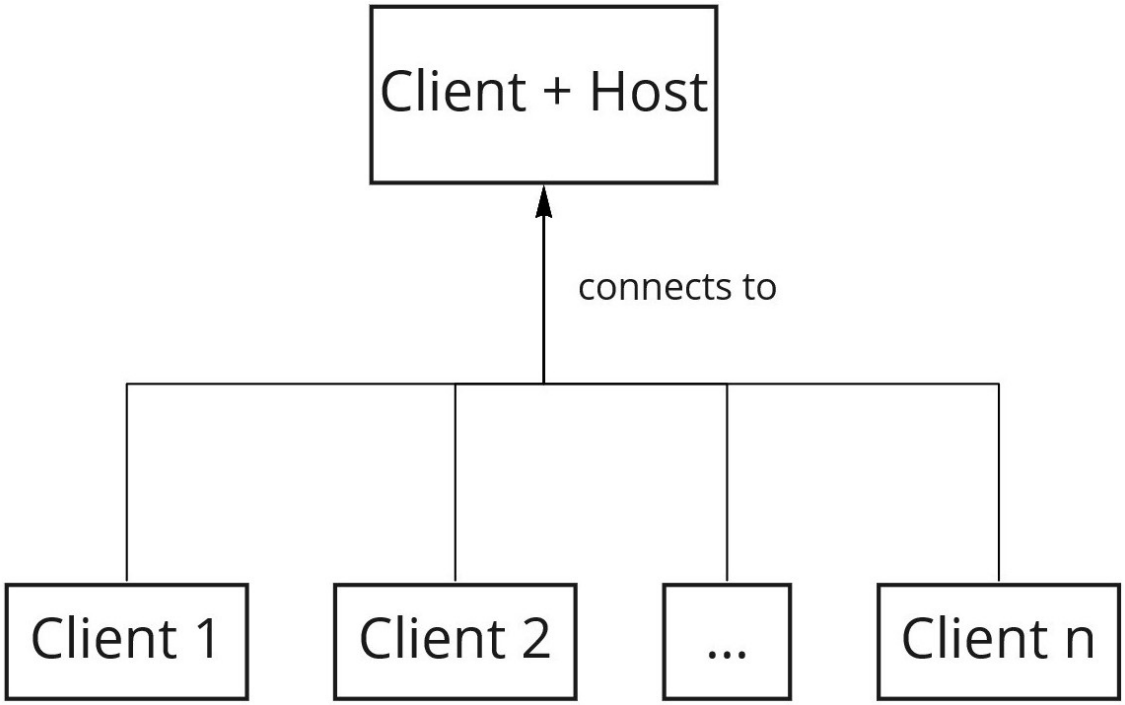

Abbildung 2.1: Das Client-Host-Modell einfach veranschaulicht

Bei dieser Variante wird der Serverprozess auf demselben Gerät ausgeführt, auf dem auch eine Client-Instanz gestartet wurde. Ein Spieler übernimmt also selbst das Hosting. Andere Clients haben die Information, welcher Client einen Serverprozess besitzt und wie sie sich dorthin verbinden können.

Vorteile: Die Unabhängigkeit von Hardwareressourcen für Entwickler Dieses 'Problem' wird an die Spieler ausgelagert.

Nachteile: Da der Serverprozess nun ebenfalls auf einem Gerät läuft, auf welches Spieler Zugriff haben, gibt es ebenfalls mehr Möglichkeiten des Hacking (Spielmanipulation) [Wik21d]. Die Entwickler haben keinerlei Einfluss auf die Serverprozesse, welche ein Spieler startet. [Sme02a]

Client Server:

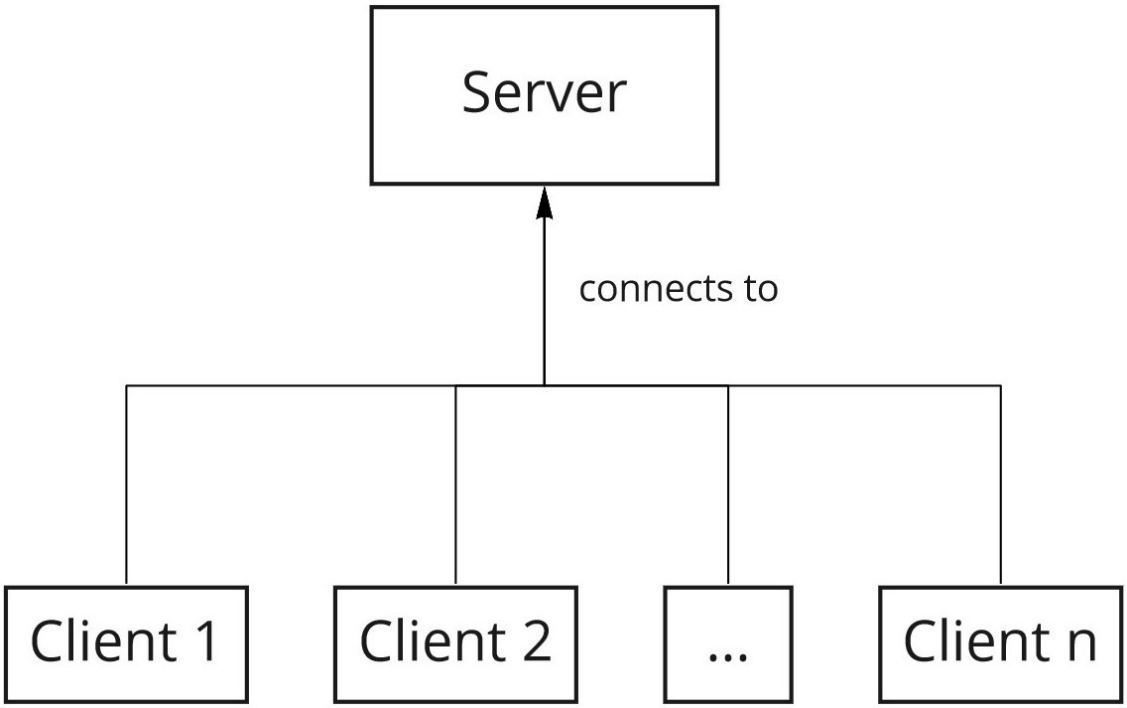

Abbildung 2.2: Das Client-Server-Modell einfach veranschaulicht

Diese Variante trennt Client und Server physikalisch voneinander. Serverprozesse werden außerhalb einer Client-Umgebung gestartet und verwaltet. Clients haben einen (oder mehrere) zentrale Zielsysteme, zu welchen sie sich verbinden können.

Vorteile:

Mehr Kontrolle auf Seiten der Entwickler, Hacking ist deutlich erschwert. Die Software-Architektur kann verhindern, dass spielentscheidende Daten nicht in der Hand der Spieler liegen, und somit ein sicheres und faires Spiel gewährleistet werden kann. [Sme02a]

Nachteile:

Die Kosten für Hardware und Bandbreite skalieren mit den Spielerzahlen. Diese Tatsache könnte sehr schnell hohe Kosten verursachen. [Den18]

Bei einer simplen, monolithischen Architektur kann dieses Modell dazu führen, dass die Hardware, welche als Server fungiert zum 'Single Point of Failure' wird. Das bedeutet, dass ein Ausfall des zentralen Spielservers dazu führen kann, dass ein Großteil der Spielerfahrung nicht mehr spielbar ist.

2.3 Datenspeicherung und Transfer

Um das bestmögliche Spielerlebnis zu ermöglichen, sollte innerhalb der Entwicklung von Multiplayer-Spielen darauf geachtet werden, dass spielentscheidende Daten schnellstmöglich zwischen den Clients synchronisiert werden. Wie auch bei anderen Echtzeitsystemen [Wik21b] kommt es also auch bei der Entwicklung von dieser Art von Software darauf an, Daten zu serialisieren (Konvertierung der Daten für die Übertragung über das Netzwerk) [Wik19b].

Daten unterscheiden sich im Kontext der Online Spielentwicklung grundsätzlich nicht großartig von anderer Software. Typischerweise werden Programmiersprachen wie C# oder C++ genutzt, um Klassen und Datenstrukturen zu erstellen, welche die eigene Spiellogik abbilden. [Gli08]

Das folgende Klassendiagramm zeigt zwei simple Klassen. Eine Klasse, welche den Spieler abbildet sowie eine Klasse, welche alle Spieler verwaltet.

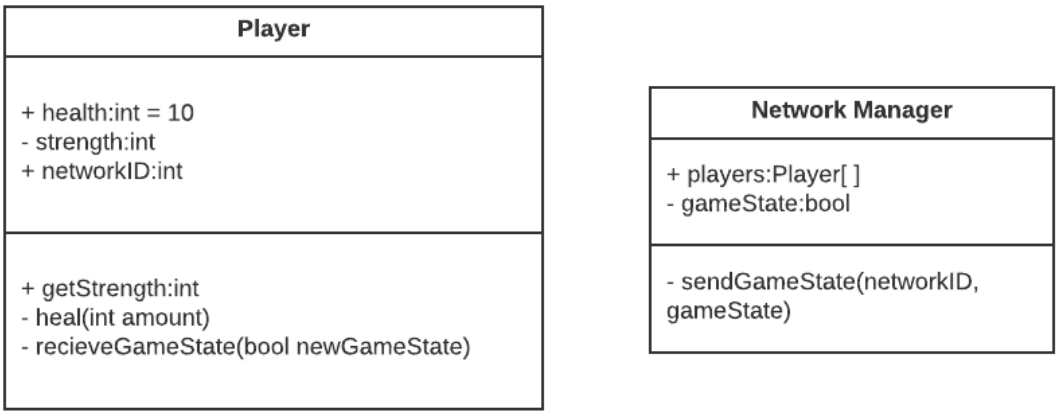

Abbildung 2.3: UML Klasse Player und Network Manager

Das Beispiel ist stark vereinfacht, soll jedoch illustrieren, auf welche Art und Weise Daten typischerweise innerhalb eines Multiplayer-Kontextes verwaltet werden.

Der Entwickler entscheidet hierbei selbst, welche Informationen in welchem Kontext vorhanden ist. Konkret muss sich ein Entwickler stets die Frage stellen, ob eine Information im Server-Kontext oder im Client-Kontext verwaltet bzw. verarbeitet werden soll.

Serialisierte Daten werden über ein Netzwerk transportiert. Je nach Art der Architektur erfolgt ein Umweg über eine Server-Instanz, oder direkt zu einem anderen Client. [Sme02b]

2.4 Informationskontrolle und Interest Management

Informationskontrolle:

'Welche Information soll zu welchem Zeitpunkt in welchem Kontext verfügbar sein?' Diese Frage müssen sich Multiplayer-Spieleentwickler aus verschiedenen Gründen regelmäßig stellen. Die Gründe sind:

Security:
Beispiel: Bei einem Online-Poker Spiel wäre es fatal, wenn ein Mitspieler jederzeit über alle Karten seiner Mitspieler Kenntnis hätte.

Network traffic:
Beispiel: In Mario Kart sammelt ein Spieler ein Item ein. Der Server würfelt für diesen Spieler ein zufälliges Item aus, welches der Spieler im Anschluss verwenden soll. Die Information über das Resultat der Zufallsauswahl sollte an den jeweiligen Spieler gesendet werden, welcher dieses Item erhalten soll. Alle anderen Spieler muss diese Information nicht interessieren.

Interest Management:

Interest Management beschreibt ein Konzept, welches delegiert, welche Informationen in welchem Kontext zu welcher Zeit für welche Personen verfügbar sind. Befindet sich der Spieler in einem bestimmten Bereich einer Spielwelt, kann das Interest Management dafür sorgen, dass dieser Spieler für manche Spieler sichtbar, und für manche Spieler nicht sichtbar ist. Die Sichtbarkeit verändert sich, je nachdem wie nahe sich die Spieler zueinander befinden, oder in welchem Areal der Spielwelt sie sich befinden.

Dieses Konzept wird besonders oft in MMO-Spielen [Wik21f] benutzt, damit ein Server mehrere Tausend Spieler ohne Abstürze verwalten kann. Ebenso wird es benutzt, um Spiellogik umzusetzen, welche es nur ausgewählten Spielern erlaubt, andere Spieler oder Objekte innerhalb der Spielwelt zu sehen und/oder mit ihnen interagieren zu dürfen.

[Sme02b]

2.5 Matchmaking

Matchmaking bei Multiplayer Online Spielen ist ein Dienst, der es Spielern ermöglicht andere Mitspieler oder Gegner zu finden. [.2014]

Die häufigsten Konzepte von Matchmaking Diensten sind:

Playlists:

Playlists sind automatisch verwaltete Ströme von Spielsitzungen, denen die Spieler nach Belieben beitreten und verlassen können. Anhand einer Reihe vordefinierter Regeln wird die Konfiguration der einzelnen Sitzungen festgelegt, ohne dass ein Mensch eingreifen muss.

Die Spiele bieten in der Regel eine Auswahl an thematischen Wiedergabelisten (z. B. Teams oder Singleplayer, ausgefallene Regeln usw.), um verschiedenen Geschmäckern oder Stimmungen gerecht zu werden. Da die Wiedergabelisten von Servern verwaltet werden, die vom Entwickler des Spiels kontrolliert werden, können sie im Laufe der Zeit geändert werden.

Wenn ein Spieler eine Wiedergabeliste auswählt, schließt er sich einem Pool von anderen Spielern an, die die gleiche Wahl getroffen haben. Der Playlist-Server verbindet sie dann entweder mit einer bestehenden Sitzung oder erstellt eine neue.

[Wik21g]

Parties:
Partys sind Gruppen von Spielern, die von Matchmaking-Systemen als eine Einheit behandelt werden. Eine Gruppe kann von Sitzung zu Sitzung wechseln, ohne dass ihre Spieler voneinander getrennt werden. Das Konzept eignet sich besonders gut für Wiedergabelisten, die automatisch die Logistik der Suche oder Erstellung von Spielsitzungen mit genügend Platz für die gesamte Gruppe übernehmen können.

[Wik21g]

Lobbys:
Bei manchen Spielmodellen wird zwischen mehreren Spielszenen hin- und hergewechselt. Eine Spielszene ist eine Sammlung an Spielobjekten, Spielmenüs und Spielcharakteren, die als eine Einheit innerhalb eines Spiels zu verstehen ist. Oft sind Szenenwechsel mit Ladezeiten verbunden. Die Aufgliederung eines Spiels in Spielszenen erleichtert Entwicklern den Umgang mit limitierten Hardwareressourcen. [Wik12]

Lobbys sind Spielszenen, in denen die Spieler die bevorstehende Spielsitzung einsehen, die Ergebnisse der letzten Sitzung prüfen, ihre Einstellungen ändern und miteinander sprechen können.

In vielen Spielen kehren die Spieler am Ende jeder Sitzung in die Lobby zurück. In einigen Spielen werden Spieler, die einer bereits begonnenen Sitzung beitreten, bis zum Beginn der nächsten Sitzung in der Lobby untergebracht. Da Lobbys nur wenige Ressourcen verbrauchen, werden sie manchmal zusätzlich als 'Warteschleife' für Spieler verwendet, bis ein geeigneter Gastgeber für die nächste Sitzung gefunden ist.

Lobbys, die von Playlists erstellt werden, haben oft einen Countdown-Timer, bevor die Sitzung beginnt, während Lobbys, die von einem Spieler erstellt werden, im Allgemeinen nach dessen Ermessen übergehen.

[Wik21g]

Ranking:
Viele Matchmaking-Systeme verfügen über ein Ranglistensystem, mit dem versucht wird, Spieler mit ähnlichen Fähigkeiten zusammenzubringen.

Spiele wie League of Legends oder die FIFA-Reihe verwenden Divisionen und Ränge für ihr Matchmaking-Bewertungssystem. Jeder Spieler tritt in verschiedenen Rängen an und für jeden Sieg gibt es Ligapunkte, für jede Niederlage werden Ligapunkte abgezogen.

Bei Spielen mit Rangliste werden in der Regel nicht gewertete Sitzungen für Spieler angeboten, die nicht wollen, dass ihre Leistung aufgezeichnet und analysiert wird. Diese werden getrennt gehalten, damit sich rangierte und nicht rangierte Spieler nicht vermischen.

[Wik21g]

Server browsers:
Einige Spiele zeigen den Spielern eine Liste aktiver Sitzungen an und ermöglichen diesen manuell eine Sitzung auszuwählen, der sie beitreten möchten. Dieses System kann in Verbindung mit Ranglisten und Lobbys verwendet werden, wird aber durch die On-Demand-Sitzungserstellung von Playlists vereitelt.

Die meisten dieser Server-Browser ermöglichen es den Spielern, die Ergebnisse zu filtern, die sie liefern. Zu den üblichen Filterkriterien gehören Servername, Spielerzahl, Spielmodus und Latenzzeit.

[Wik21g]

Contacts lists:
Eine der gängigsten Formen des Matchmaking besteht darin, den Spielern eine Liste anderer Spieler zur Verfügung zu stellen, die sie bereits kennengelernt haben und mit denen sie vielleicht wieder spielen möchten. Der Status jedes Spielers (offline, online, spielend) wird angezeigt, es besteht die Möglichkeit, einer laufenden Sitzung beizutreten.

In vielen Fällen werden die Kontaktlisten von der Plattform verwaltet, auf der ein Spiel läuft (z. B. Xbox Live, PlayStation Network, Steam), um den Spielern den Aufwand zu ersparen, viele separate Listen für viele einzelne Spiele zu verwalten. [Wik21g]

3 Konzepte

In diesem Kapitel werden abstrakte Konzepte beschrieben. Besonders unerfahrene Entwickler eines Online-Multiplayer Spiels sollen anhand von diesen Konzepten Ideen für eine sinnvolle Herangehensweise erhalten, um einen roten Faden in ihrer Entwicklung verfolgen zu können. Insbesondere wenn der Entwickler noch wenig oder keine Erfahrung mit der Entwicklung von Online-Mulitplayerspielen gesammelt hat, sollen diese Konzepte bei der Erstellung einer soliden Grund-Architektur nützlich sein.

Bevor eines dieser Konzepte angewandt werden kann, müssen folgende Voraussetzungen erfüllt sein:

1. Das grundsätzliche Spielkonzept steht fest.

2. Anhand von dem festgelegten Spielkonzept kann abgeschätzt werden, ob sich für das Spiel eine Client / Server oder eine Client / Host Architektur eignet.

Sollte das Team bzw. der Solo-Entwickler zum Entschluss gekommen sein, dass sich ein Client / Server Modell am besten für den spezifischen Use Case eignet, so müssen Vorkehrungen getroffen werden, um die Hardware-Ressourcen für den späteren Server-Runner bzw. die Matchmaking API sicherzustellen. Die Grundvoraussetzung ist jedoch ein aus dem Internet erreichbarer PC mit einer festen IP-Adresse.

Alternativ kann auch zunächst lokal innerhalb eines LAN [Wik22b] entwickelt werden. Hierbei ist jedoch wichtig, dass eventuell auftretende Seiteneffekte, die durch Latenz [Wik22d] oder Bandbreite [Wik19a] verursacht werden, nicht unter Realbedingungen getestet werden können. Es empfiehlt sich deshalb bereits innerhalb der ersten Entwicklungsiterationen auszutesten, wie Prototypen für Serverprozesse auf unterschiedliche Standorte der Serverhardware reagieren.

Nun gilt es, passende Technologien zu finden. Hier gibt es keine klare Empfehlung, jedoch können die in diesem Kapitel beschriebenen Konzepte bei der Entscheidung helfen. Es gibt bereits Frameworks & Game Engines, die manche dieser Konzepte bereits implementiert haben und als Libraries für Entwickler bereitstellen.

[MFa21]

3.1 Vorwort zu den aufgestellten Konzepten

Wie bereits in der Einführung erläutert, sind die folgenden Konzepte keine konkrete Entwicklungsanleitung. Vielmehr sind sie im Kontext einer bereits erarbeiteten Spielidee eine Hilfestellung zur Erarbeitung einer individuellen technischen Architektur.

Konkret muss beachtet werden, dass einige der Konzepte nicht als einzelne Programm-Klassen verstanden werden müssen, sondern als Hilfestellung für Ableitungen zu konkreten Klassen oder Skripten genutzt werden können. Es ist auch gut denkbar, dass einige Implementierungen deutliche Vorteile genießen, wenn bestimmte Konzepte jeweils für getrennte Spiel-Szenen [Wik12] umgesetzt werden.

Die Konzepte sind als separat voneinander zu betrachtende Softwarekomponenten zu sehen, die miteinander kommunizieren können. Man könnte die einzelnen implementierten Konzepte auch als Microservices [Tho15] bezeichnen, die innerhalb eines Server-Prozesses eigene interne Abläufe besitzen und durchführen. Die Kommunikation zwischen den Komponenten sollte über direkte gegenseitige Funktionsaufrufe oder über ereignisgesteuerte Funktionen [Mic06] geschehen.

3.2 API für Matchmaking & Server Runner

Für Spiele, welche ein Matchmaking-System benötigen, muss eine API entworfen werden, welche unabhängig von einem existierenden Client oder Serverprozess arbeitet. Diese API hat 2 grundsätzliche Aufgaben.

Matchmaking:

Die API muss einen Algorithmus implementieren, welcher mehrere Spieler zu einer Spiel-Session zusammenführt. Mögliche Matchmaking Konzepte sind bereits im Hintergrund-Kapitel beschrieben. Die Matchmaking API verwaltet als eine Liste an aktiven Serverprozessen sowie die Information, wie man sich zu ihnen verbindet. In der Regel wird pro Serverprozess ein Netzwerk-Port an der Server-Maschine reserviert, auf denen sich dann N Spieler verbinden können.

Je nach Spielkonzept können hunderte, tausende Serverprozesse existieren, welche jeweils nur eine vergleichsweise geringe Anzahl (1-20) an Spieler verwalten. Spielkonzepte, welche viele Spieler (200-1000) innerhalb eines einzigen Serverprozesses voraussetzten, erzeugen dagegen zwar quantitativ weniger Serverprozesse, diese neigen aber in der Regel schnell zu Überlastung. Neben der Umsetzung von Interest Management und einer performanten Architektur für möglichst wenig Network-Traffic kann aber auch die Matchmaking API Abhilfe schaffen, bspw. durch 'Umverlegung' von Spielern auf andere oder neue Cluster.

Server Runner:

Neben dem Matchmaking ist die API ebenfalls auch zuständig für das Starten und die Überwachung von Serverprozessen. Ebenfalls können technische Statusinformationen über aktuell laufende Spiel innerhalb eines Serverprozesses von der API verwaltet werden, beispielsweise ob es möglich ist einem Server beizutreten, oder wie viele Spieler bereits auf einem Serverprozess spielen.

Die folgende Grafik visualisiert beide Aufgaben:

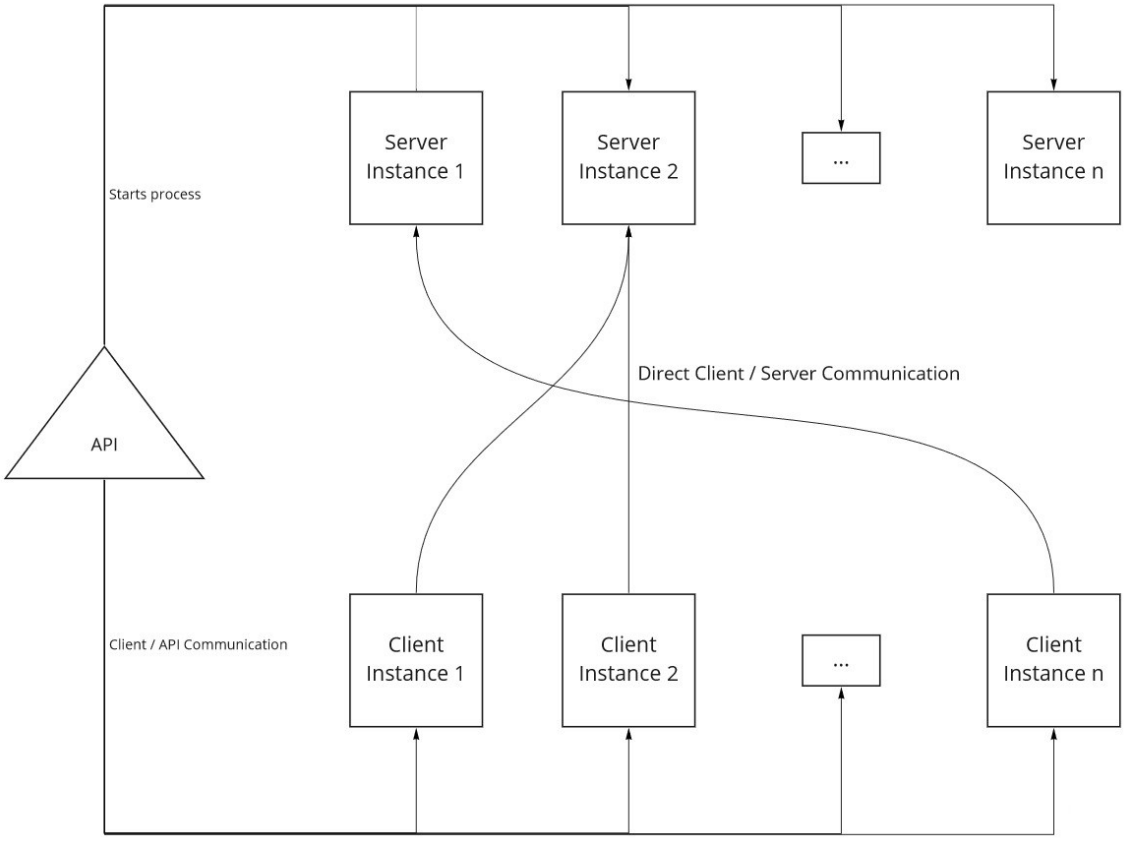

Abbildung 3.1: Veranschaulichung des API-Konzepts

Je nach Spiel kann das Aufgabenfeld der API um weitere Punkte erweitert werden, beispielsweise die Verwaltung einer Datenbank für Authentifizierung bzw. Kommunikation mit externen Authentifizierungs-Service-Providern oder für einen In-Game Shop.

3.3 Client UI & Visual Controller

Der Client UI & Visual Controller beschreibt die Art und Weise, wie die Benutzeroberfläche und visuelle Ebene eines Spielers durch Antworten des Servers oder einzelner Spieleraktivitäten manipuliert wird.

Konkret sorgt ein Server-Ereignis, der Input eines Spielers (beispielsweise durch Klicken eines Buttons oder einsammeln eines Gegenstands) oder ein anderes Ereignis dafür, dass Funktionen innerhalb des Client UI & Visual Controllers aufgerufen werden. Innerhalb dieser Funktionen werden Komponenten der Benutzeroberfläche wie z. B. Anzeigetexte, Bilder etc. oder Objekte innerhalb der Spielwelt selbst manipuliert. In der folgenden Grafik zeigt die beschriebene Folge an Ereignissen:

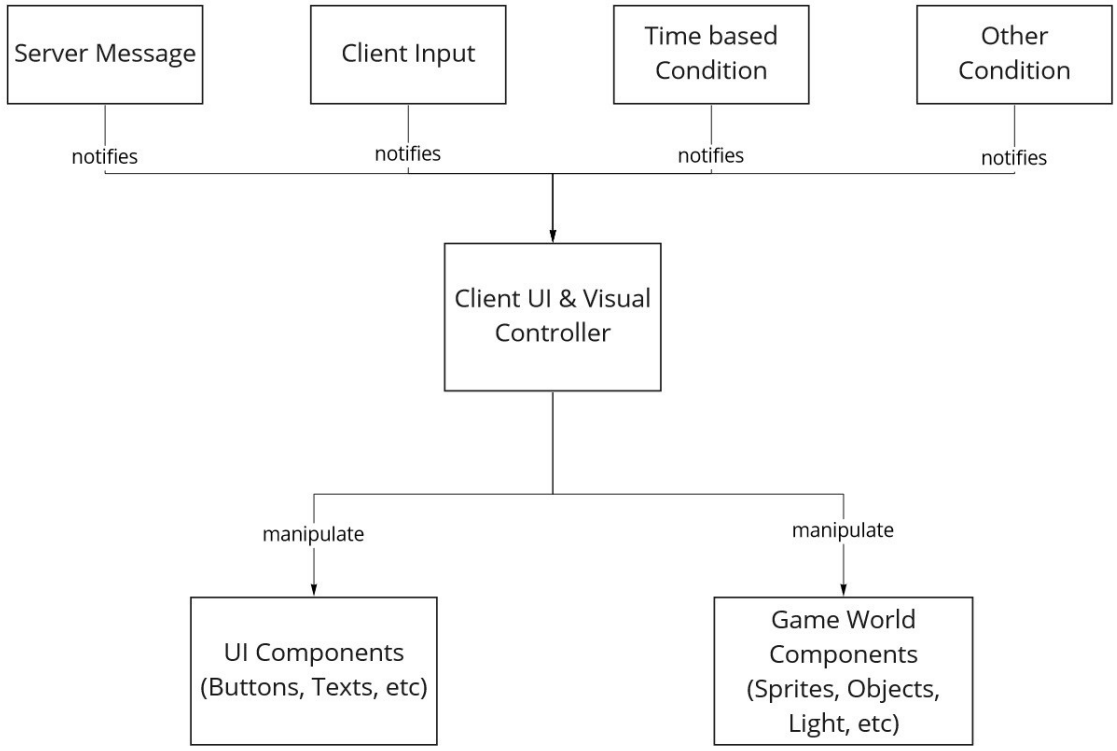

Abbildung 3.2: Veranschaulichung des Client UI & Visual Controller Konzepts

Wichtig ist, dass Funktionen des Client UI & Visual Controller erst nach server- oder clientseitigen Überprüfungen ausgeführt werden. Der Controller selbst sollte nur überprüfen, ob fluktuierende Komponenten noch existieren oder sich in einem konsistenten Zustand befinden. Spiel-Relevante Logik, beispielsweise, ob ein Spieler berechtigt ist, bestimmte Aktionen durchzuführen, ist nicht Teil des Client UI & Visual Controllers.

3.4 Server Network Manager

Der Server Network Manager ist für die korrekte Abarbeitung von serverseitigen Aufgaben zuständig, welche aufkommen, sobald ein Netzwerk-spezifisches Ereignis stattfindet.

Beispiel:
Die Verbindung eines Clients bricht mitten in der laufenden Spielszene ab. Der Ser-

ver Network Manager stößt infolgedessen Funktionen anderer serverseitigen Manager-Komponenten an, welche dafür sorgen, dass alle Spieler über den neuen Zustand des Spiels informiert werden. Hierfür müssen zunächst serverinterne Variablen aktualisiert werden.

Beispiel:
In einem Online-Ableger des Spiels 'Mensch ärger dich nicht' verliert ein Spieler seine Netzwerkverbindung. Als Reaktion kümmert sich der Server Network Manager darum, dass das Spiel trotzdem zu Ende gespielt werden kann. Die Spielfiguren des ausgeschiedenen Spielers müssen entfernt, und der Spiel-Fortschritt aktualisiert werden. Diese Funktionen stellt möglicherweise der Runtime Spawn Manager, Progress / Game State Manager zur Verfügung. Außerdem muss dem Serverside Client Manager mitgeteilt werden, dass dieser Spieler nun nicht mehr Teil der Spiel-Session ist.

Die folgende Grafik illustriert die Aufgaben des Server Network Managers:

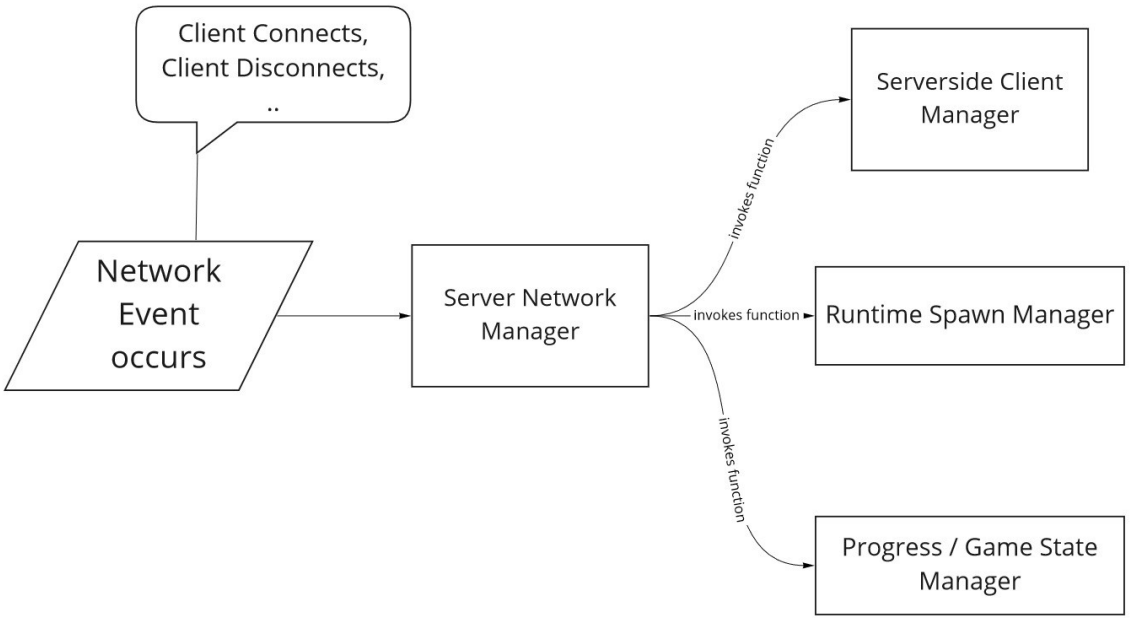

Abbildung 3.3: Veranschaulichung des Server Network Manager Konzepts

3.5 Lobby / Multi Scene Manager

Der Lobby/ Multi Scene Manager ist eine serverseitige Komponente, welche es erlaubt, Spielszenen-übergreifende Informationen zu speichern und zu verwalten. Beispielsweise könnte bei einem Lobby-basierten Matchmaking die Information des Leiters einer Lobby über Spielszenen hinweg gespeichert werden. Ebenso wäre es denkbar, dass der Bereitschafts-Status vor dem tatsächlichen Spiel innerhalb des Lobby / Multi Scene Managers gespeichert wird.

Außerdem stellt der Lobby / Multi Scene Manager Funktionen bereit, welche

- alle Clients informiert, sobald ein globaler Szenenwechsel angestoßen werden soll

- alle Clients über den aktuellen Status einer Lobby informiert.

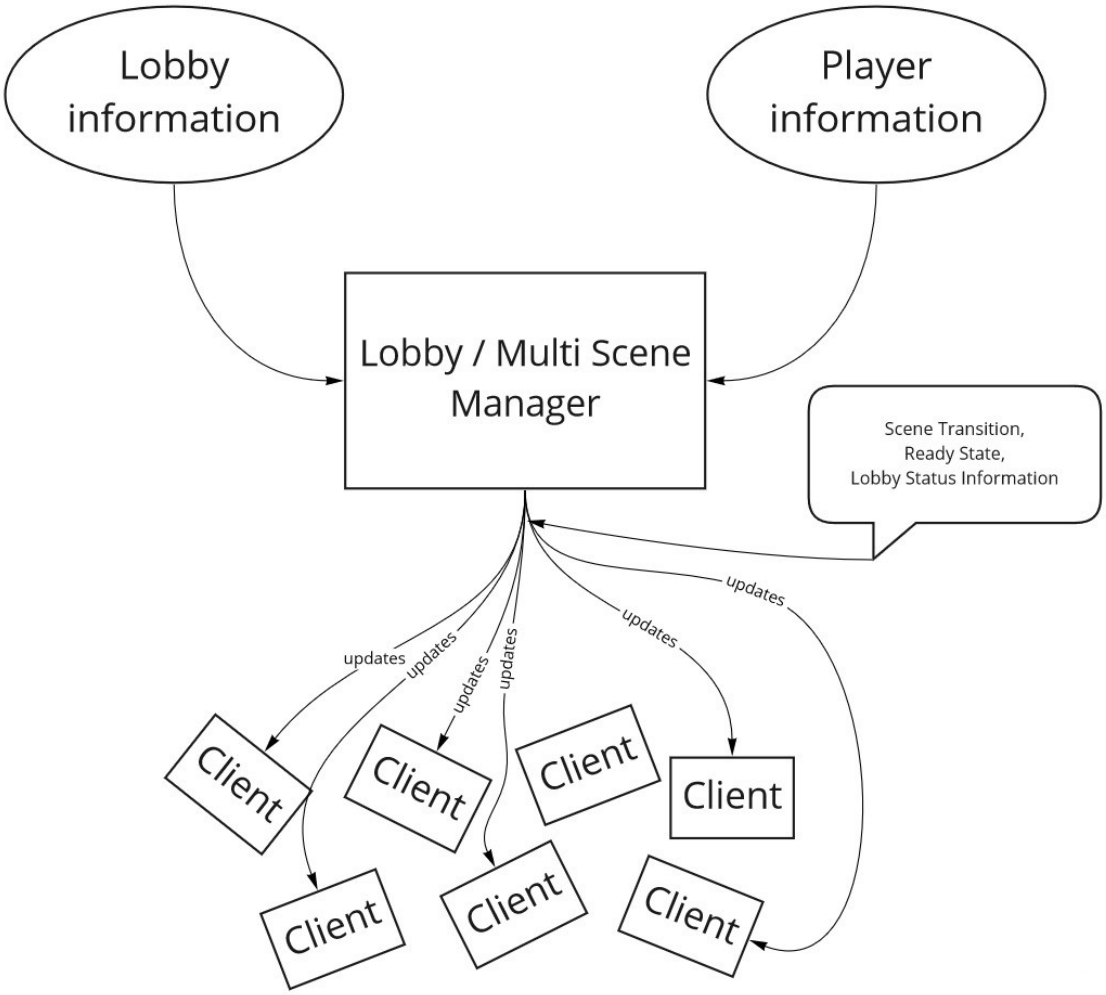

Abbildung 3.4: Veranschaulichung des Lobby / Mutli Scene Manager Konzepts

3.6 Client Connection Manager

Der Client Connection Manager beschreibt eine Softwarekomponente, welche folgende Aufgaben umsetzen muss:

- Implementierung von clientseitigen Methoden zum Informationsaustausch mit Matchmaking API. Z. B. stellt die Matchmaking API dem Client über HTTP-Kommunikation Informationen bereit, mit denen sich der Client auf einen speziellen Server verbinden kann (IP + Port). [Wik21e] [Wik21i]

- Verarbeitung der von Matchmaking API bereitgestellten Informationen.

Z. B. könnte die Matchmaking API einem Client die Information bereitstellen, dass aus einem bestimmten Grund dem angeforderten Serverbeitritt verweigert wird. Diese Information muss der Client Connection Manager zwischenspeichern und verwalten.

- Aufruf von Methoden eines Client UI Controllers.

Z. B. Die soeben erlangte Information über den verweigerten Serverbeitritt muss nun dem Nutzer dargestellt werden, hierfür ruft der Client Connection Manager Funktionen auf, welche ein Client UI Controller implementiert hat.

- Bereitstellung von Funktionen, welche den Beitritt zu einem Game-Server sicherstellen.

- Ausführen von Handler-Funktionen und Auslösen von Events für clientseitige Konsequenzen bei Netzwerkabbruch, manuellem Verlassen einer Server-Session oder sonstigen Abbruchgründen, welche dazu führen, dass eine Online-Spielsession verlassen wird.

Z. B. wird die notwendige Verbindung zum Internet während des Spiels unterbrochen. Der Client Connection Manager löst Callbacks aus, welche im Client UI & Visual Controller implementiert sind. Diese sorgen dafür, dass der Spieler zurück ins Hauptmenü geleitet wird, in dem eine Fehlermeldung dargestellt wird, welche erklärt, warum das Spiel abgebrochen ist.

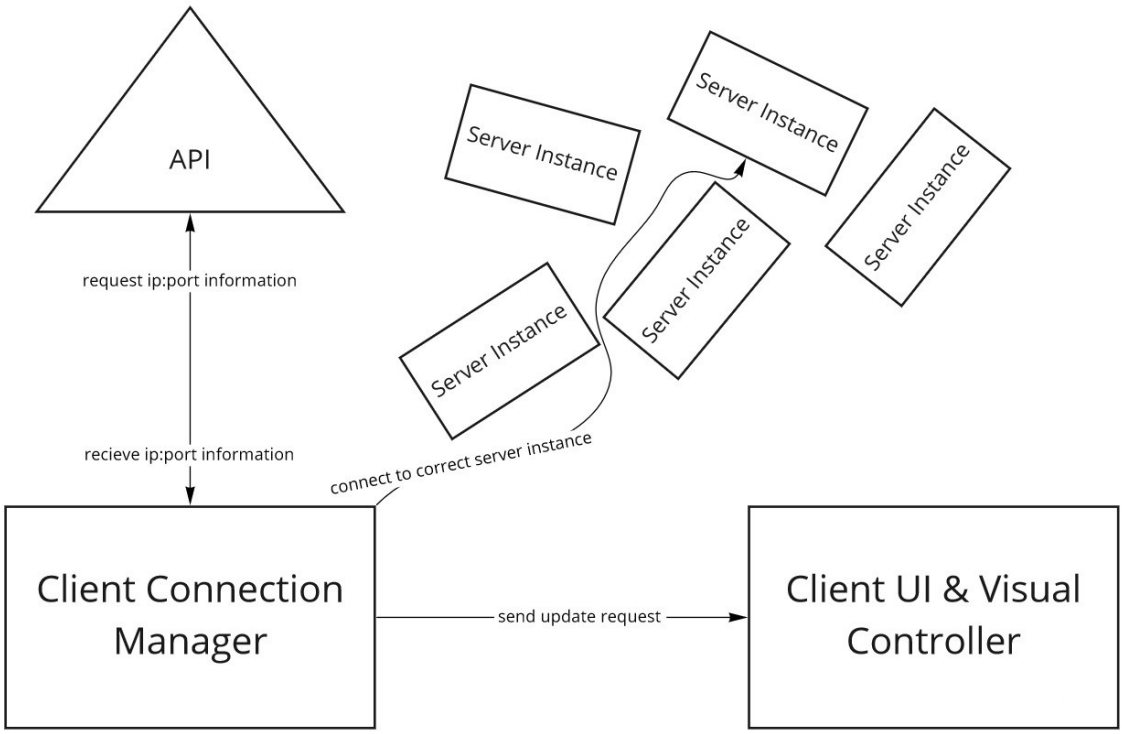

Abbildung 3.5: Veranschaulichung des Client Connection Manager Konzepts

3.7 Serverside Client Manager

Das Konzept des serverseitigen Client Managers beinhaltet die Verwaltung aller verbundener Clients innerhalb eines aktiven Serverprozesses. Insbesondere speichert der serverseitige Client Manager 2 Arten von Information über jeden Client:

Einerseits werden netzwerkbezogene Informationen über einen Client gespeichert, im einfachsten Fall ist dies lediglich die IP-Adresse des Hosts sowie der Port, auf welchem der Server seinen Kommunikationskanal geöffnet hat. Bei Spielkonzepten, welche eine weitergehende Authentifizierung des Spielers voraussetzen, können hier auch Benutzernamen, Benutzertags, E-Mail-Adressen o. ä. gespeichert und weiterverarbeitet werden.

Zum Anderen speichert und verwaltet der serverseitige Client Manager auch spielespezifische Informationen wie bspw. den aktuellen Anzeigenamen, konfigurierte Individualisierung des Spielcharakters, Spielerrollen oder andere Eigenschaften, über die der Server während einer Spiel-Session Kenntnis haben sollte.

Die folgende Grafik visualisiert diesen Informationsfluss:

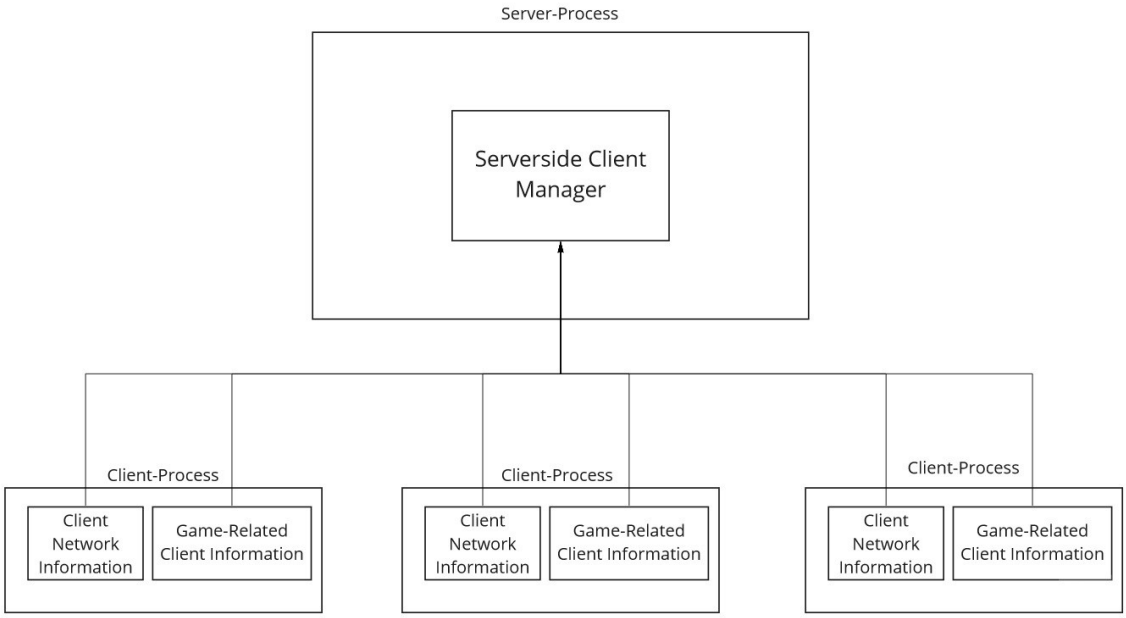

Abbildung 3.6: Veranschaulichung des serverseitigen Client Manager Konzepts

3.8 Prepare-Game-Manager

Der Prepare-Game-Manager ist Teil des Serverprozesses und beinhaltet alle Funktionen, welche nötig sind, um eine Spielszene aufzubauen, bevor die Spieler ihr beitreten dürfen. Die Funktionen sollten konkret umsetzen:

- Spawning [Wik20] von NPCs [Wik21h] oder Spielgegenständen, die in Abhängigkeit zur Anzahl der beitretenden Spieler, einer Spielkonfiguration oder sonstigen Parametern stehen.

- Anpassung der Eigenschaften von bereits gespawnten NPCs oder Spielgegenständen, die in Abhängigkeit zur Anzahl der beitretenden Spieler, einer Spielkonfiguration oder sonstigen Parametern stehen.

- Initialisierung der Spawnpunkte für Spieler anhand von Spawnalgorithmen oder festgelegten Punkten in der Spielwelt.

- Abarbeitung sonstiger serverseitigen Abläufe, welche Voraussetzung für den Start des Spiels sind

Das folgende Diagramm zeigt, wann der Prepare-Game-Manager bei einem lobbybasierten Spielkonzept zum Einsatz kommt:

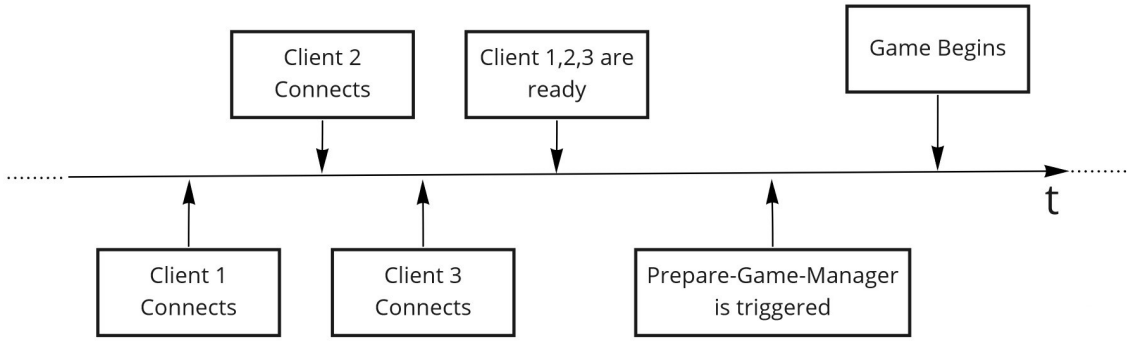

Abbildung 3.7: Veranschaulichung des Prepare Game Manager Konzepts

3.9 Progress / Game-State Manager

Die Aufgabe eines Progress bzw. Game-State Managers ist es, Gewinn- bzw. Verlustbedingungen einer Spielsession zu speichern und zu verwalten.

Beispiele hierfür sind:
- Verwaltung und Synchronisierung von globalen Timern - Verwaltung und Synchronisierung von Spiel-Kennzahlen bzw. Variablen (z. B. der Spielstand bei einem Fußball-Match)

Außerdem ist der Progress Manager / Game-State Manager dafür verantwortlich, alle Clients über spielentscheidende Ereignisse zu informieren.

Die Abfolge dieser Ereignisse werden im folgenden Diagramm aufgezeigt:

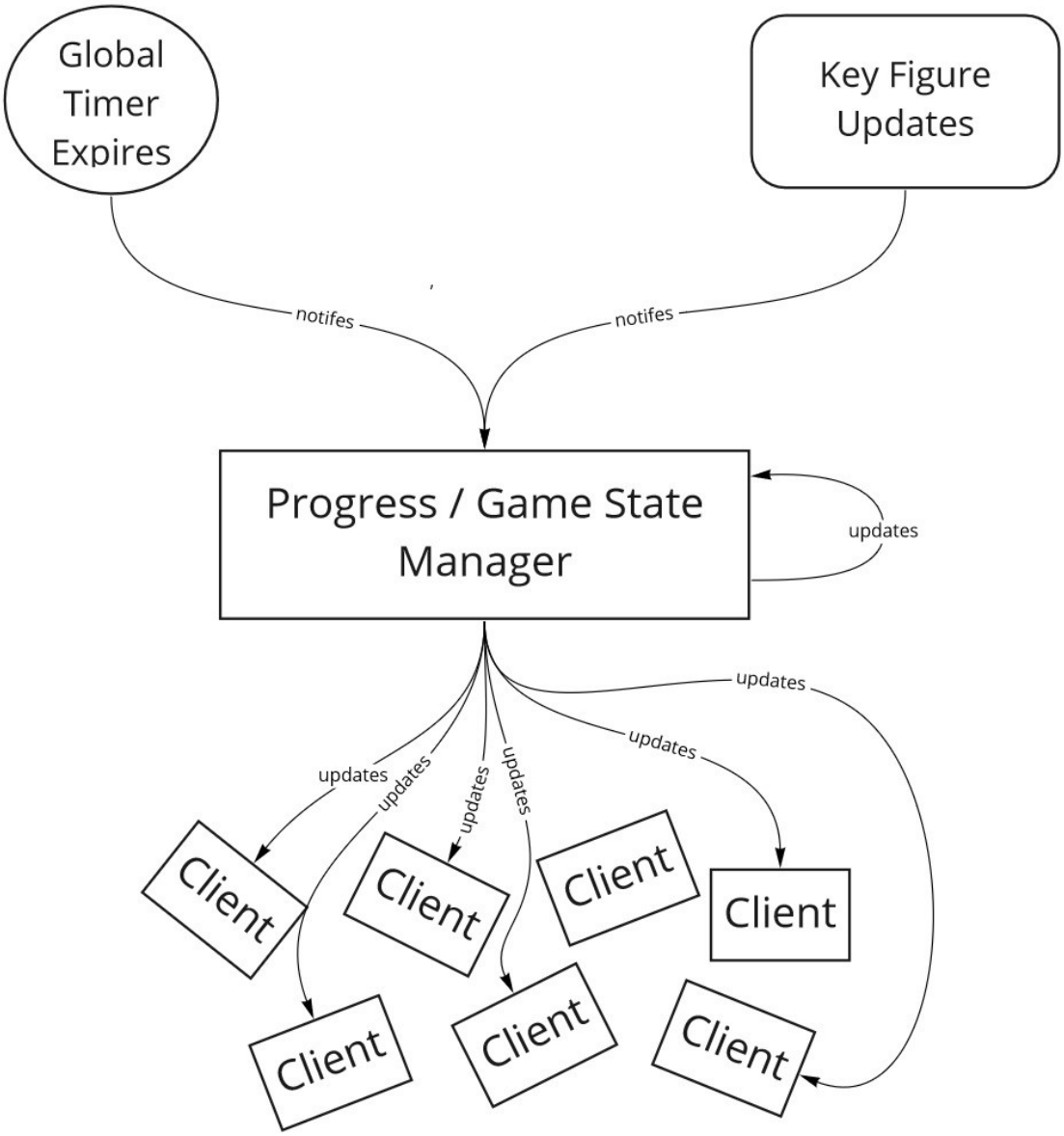

Abbildung 3.8: Veranschaulichung des Progress / Game-State Manager Konzepts

3.10 Runtime Spawn Manager

Der Runtime Spawn Manager verwaltet die zur Laufzeit zu erzeugenden Spieler- und Nicht-Spieler-Objekte. Konkret werden Funktionen des Runtime Spawn Managers ausgeführt, wenn ein spezifisches Event innerhalb einer Spiel-Session ausgelöst wird oder ein Spieler- bzw. Nicht-Spieler-Charakter stirbt, und diese erneut an anderer Position spawnen sollen ('Re-Spawning'). Die Logik, wo genau ein Spieler oder Nicht-Spieler Objekt seine neue Einstiegsposition erhält, regelt ebenfalls der Runtime Spawn-Manager.

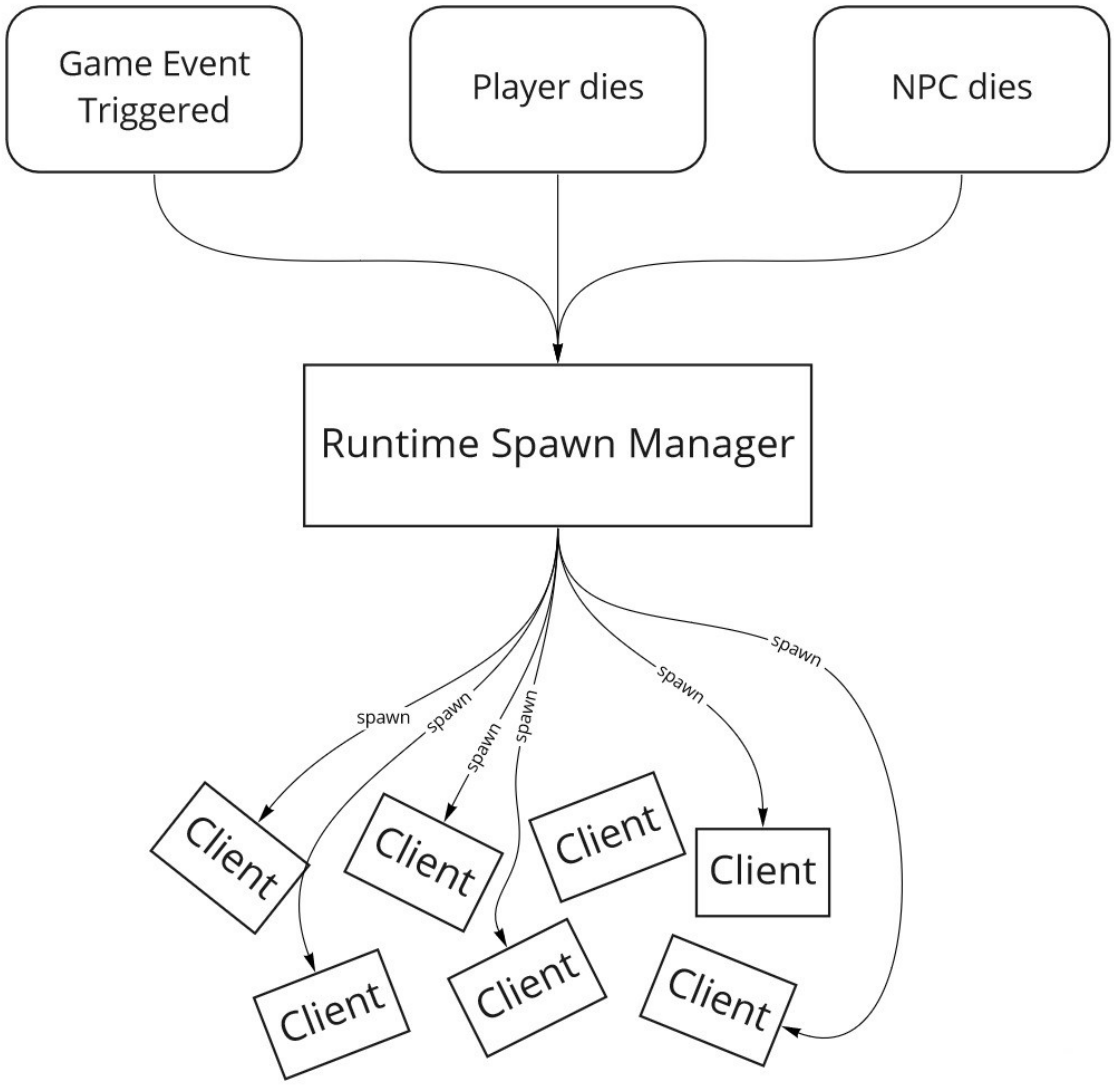

Abbildung 3.9: Veranschaulichung des Runtime Spawn Manager Konzepts

Beispiel: In einem Online Multiplayer Ego-Shooter [Wik21c] stirbt ein Spieler durch Schüsse anderer Spieler. Der Runtime Spawn Manager sorgt dafür, dass anhand von bestimmten, zur Laufzeit ausgerechneten Bedingungen eine neue Spawn-Position für den gestorbenen Spieler gefunden wird.

3.11 Interest Manager

Der Interest Manager kümmert sich um die Sichtbarkeit von Objekten. Wie bereits in der Sektion über Interest Management erklärt, ist es je nach Spielkonzept notwendig, dass der Spieler stets nur über den für ihn relevanten Teil der Spielwelt Kenntnis hat. Aus diesem Grund sollten sich angehende Spieler-Entwickler fragen, ob sie eine

solche Software-Komponente in die Architektur ihres Projekts integrieren möchten. Die folgende Grafik veranschaulicht diese Aufgabe:

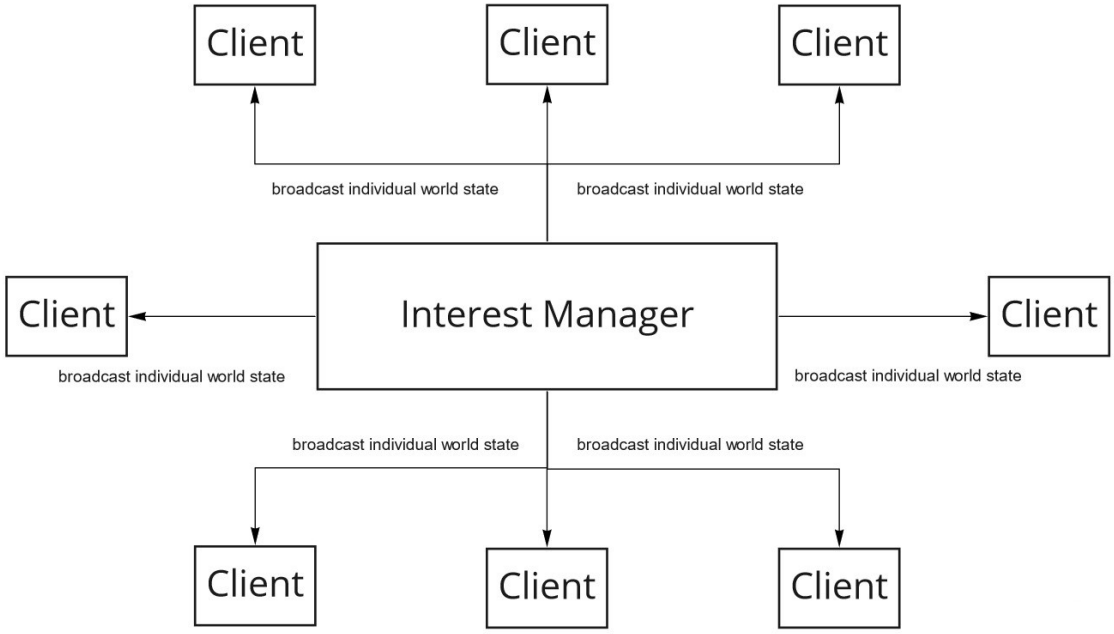

Abbildung 3.10: Veranschaulichung des Interest Manager Konzepts

Mögliche Faktoren, die der Interest Manager benutzen kann, um den Spielern die Objekte anzeigen zu lassen, die sie sehen dürfen sind:

- Distanz zwischen Spieler und anderen Objekten

- Teams bzw. Gruppen. Die registrierten (Spieler)-Objekte innerhalb eines Teams/ einer Gruppe sind für Spieler, welche ebenfalls innerhalb des Teams/Gruppe registriert sind sichtbar.

- Spatial Hashing / Grid Checker: Die Spielwelt und ihre Objekte werden in quadratische Zellen eingeteilt. Je nachdem in welcher Zelle sich ein Spieler befindet, werden nur diejenigen Objekte der Spielwelt mit dem Spieler synchronisiert, welche sich innerhalb der 8 'Nachbarzellen' befinden:

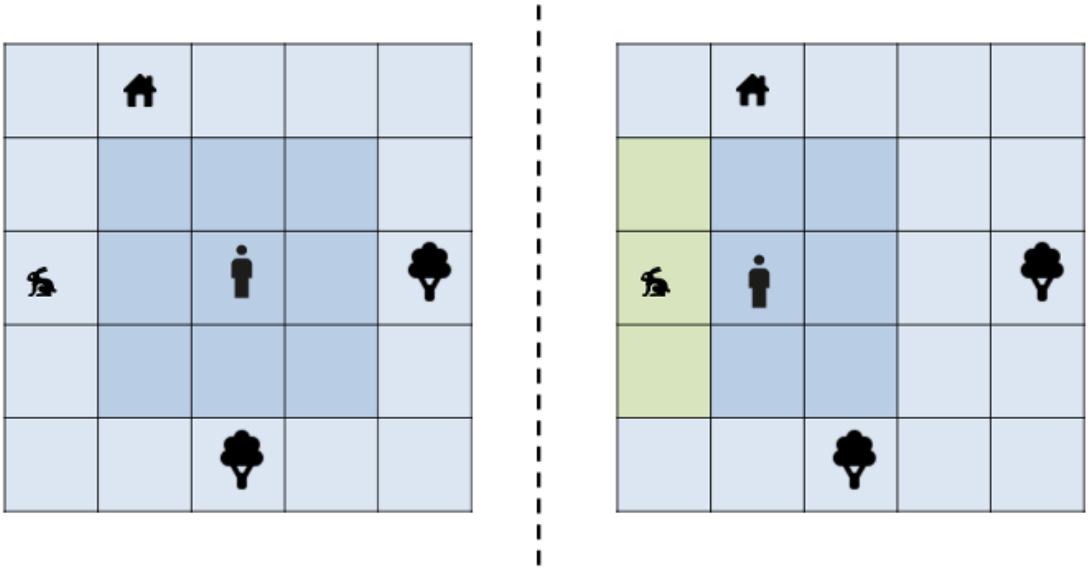

Abbildung 3.11: Illustration des Interest Management Konzepts Spatial Hashing. [Jer17]

Sollte keine der hier beschriebenen Beispiele auf das gewünschte Spielkonzept passen, so ist es notwendig, eine komplett eigene Logik für das Interest Management zu implementieren.

4 Realisierung

In diesem Kapitel werden die im vorherigen Kapitel beschriebenen Konzepte anhand von einem Prototyp implementiert. Die konkrete Logik wird bei jedem implementierten Konzept anhand von Code erläutert.

4.1 Eingesetzte Technologien

Für die Entwicklung des Prototyps kamen folgende Technologien zum Einsatz:

Als Game-Engine wurde Unity [Tec22] benutzt. Die Entscheidung hierfür begründet sich durch die gute Dokumentation und den Long-Time-Support der verschiedenen Unity-Versionen. Außerdem gibt es eine große Anzahl an Benutzern. Unity existiert bereits seit dem Jahr 2005 [Wik22c], und hat sich seitdem in Entwickler-Kreisen einen nachhaltigen Ruf erarbeitet.

C# kommt als Programmiersprache zum Einsatz. Unity bietet eine sehr gute Unterstützung hierfür.

Zusätzlich zu Unity wurde Mirror[.0322] als weiteres Framework benutzt. Mirror ist eine kostenloses Open Source Networking Bibliothek, welche die Entwicklung von Netzwerk-Code deutlich erleichtert.

4.2 Spielidee des Prototyps

Das Grundprinzip des Prototyps basiert auf dem Kinderspiel 'Fangen und Verstecken'. Hinzu kommen weitere Spiel-Elemente, die in dieser Sektion erläutert werden. Innerhalb der Spielwelt existieren 2 verschiedene Teams, welche ausschließlich aus Spielern bestehen.

Spieler mit der Rolle 'Hider' und 'Ghost' gehören zu einem Team. Spieler mit der Rolle 'Seeker' bilden das zweite Team. Anfangs wird per Zufall entschieden, welche Spieler zum Team 'Hider/Ghost' und welche zum Team 'Seeker' gehören. Alle Teammitglieder des

Teams 'Hider/Ghost' beginnen das Spiel initial als 'Hider'. Der Leiter einer Lobby kann festlegen, ob ein oder zwei Seeker ausgewürfelt werden sollen. Zum Leiter einer Lobby wird derjenige Spieler, der als Erstes einer Lobby beitritt. Sollte der Leiter einer Lobby diese verlassen, so wird der Spieler zum Leiter, der am längsten Mitglied derselben Lobby war.

Die Aufgaben der Spieler sind je nach Rolle unterschiedlich:

Hider:
Spieler mit der Rolle 'Hider' haben die Aufgabe, bestimmte Objekte in der Spielwelt zu finden, und diese zu einem Zielort zu bringen. Sind alle Objekte zu ihrem Zielort gebracht, so hat das Team 'Hider/Ghost' das Spiel gewonnen. Hider müssen zudem vor den Spielern mit der Rolle 'Seeker' flüchten, bzw. sich vor diesen verstecken.

Seeker:
Spieler mit der Rolle 'Seeker' haben lediglich eine Aufgabe. Sie müssen alle Spieler mit der Rolle 'Hider' fangen. Dies können sie tun, indem sie in die unmittelbare Reichweite eines Hiders laufen, und die Taste 'E' bzw. den in der Benutzeroberfäche festgelegten Knopf 'Catch' drücken. Sind alle Hider gefangen, so gewinnen die Seeker das Spiel.

Ghost:
Spieler, welche initial mit der Rolle 'Hider' gespawnt sind, und von einem Seeker gefangen wurden, wechseln ihre Rolle zu 'Ghost'. Ghosts dürfen nicht mehr mit Objekten der Spielwelt interagieren, sie können sich lediglich noch über das Spielfeld bewegen. Außerdem bekommt ein Ghost eine Fähigkeit, welche er einsetzen kann, um seinem Team zu helfen. Diese Fähigkeit bemächtigt den Spieler einen Seeker für 3 Sekunden bewegungsunfähig zu machen, sollte der Spieler in der unmittelbaren Nähe eines Seekers sein, und die Taste 'E' bzw. den dafür vorgesehen Knopf in der Benutzeroberfläche drücken. Diese Fähigkeit kann lediglich alle 30 Sekunden eingesetzt werden.

Das Spielkonzept wird in folgendem Spielflussdiagramm zusammengefasst:

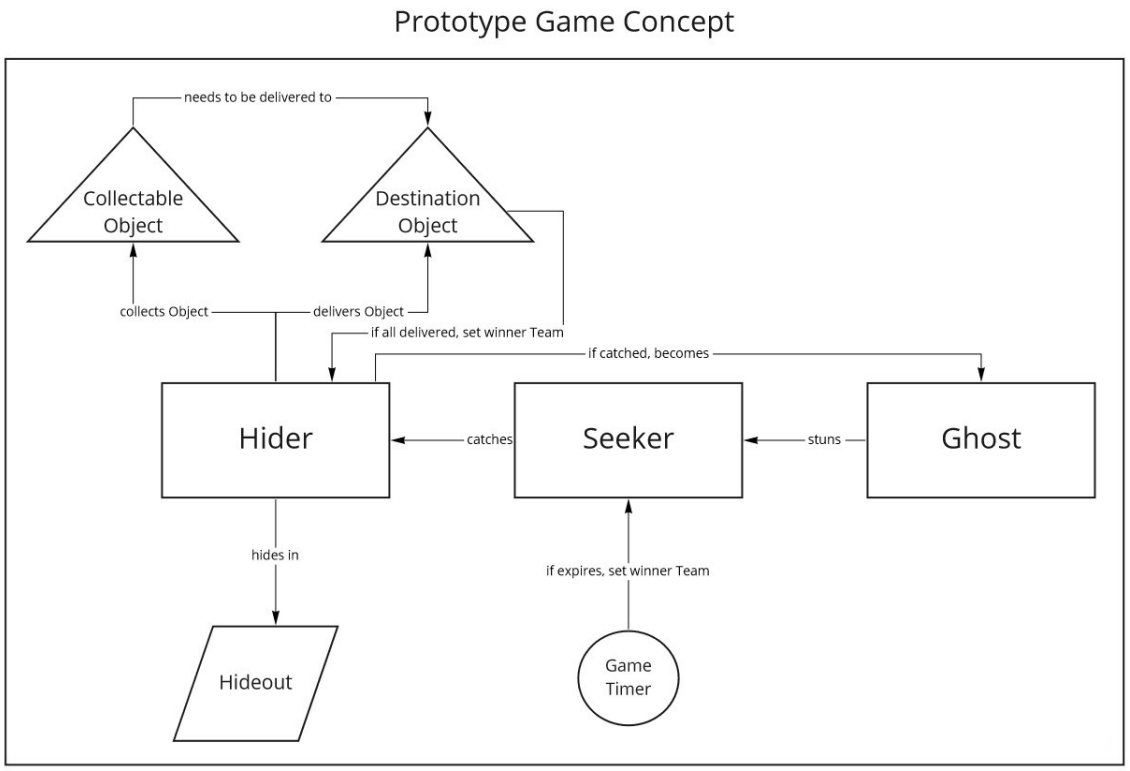

Abbildung 4.1: Veranschaulichung des Spielkonzepts

Spiel-Ablauf:

Zunächst muss ein Spieler eine Lobby erstellen. Dies tut er, indem er auf den Knopf 'Create Lobby' im Hauptmenü drückt. Nun erhält dieser Spieler einen 'Code', welcher eindeutig diese Lobby identifiziert. Diesen 'Code' kann der Spieler jetzt an seine Mitspieler weitergeben. Diese können den Code im Hauptmenü in ein Eingabefeld eingeben, und auf 'Join Lobby' drücken. Der Spieler, welcher die Lobby erstellt hat, ist inzwischen auch Lobby-Leiter und kann einstellen, ob die Lobby als 'public' markiert werden soll. In diesem Fall können andere Spieler durch den Button 'Join Random Lobby' ebenfalls der Lobby beitreten, ohne den Code zu kennen.

Das folgende Aktivitätsdiagramm zeigt die möglichen Abfolgen des Beitritts einer Lobby:

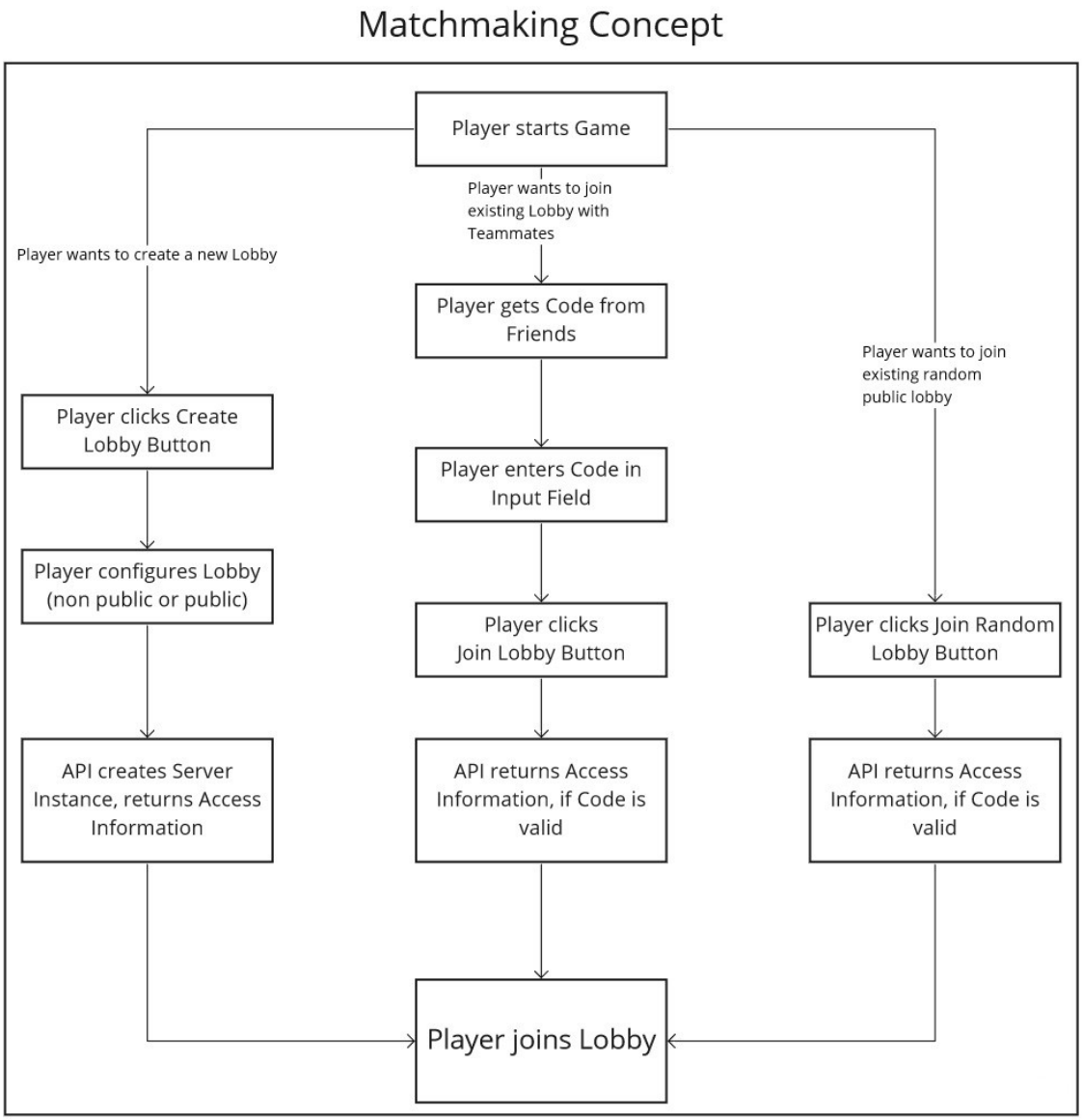

Abbildung 4.2: Veranschaulichung des Prototyp-Matchmaking

Ist nun eine Lobby erstellt, und es sind mindestens 2 und maximal 10 Spieler der Lobby beigetreten, welche ihren Bereitschaftsstatus durch das Drücken des Knopfes 'Ready' kommuniziert haben, so hat der Lobby-Leiter die Möglichkeit durch Drücken des Buttons 'Start Game' das Spiel zu starten.

Aus insgesamt 10 möglichen Spawn-Punkten würfelt der Prepare-Game-Manager Spawn-Punkte für die beigetretenen Spieler aus. Sobald alle Spieler die Spiel-Szene geladen haben, und auf ihrem Startpunken stehen, beginnt das Spiel. Ab diesem Zeitpunkt läuft eine Stoppuhr ab, erreicht diese den Wert '0' bevor alle Objekte an ihren Zielort gebracht wurden, so hat das Team 'Seeker' gewonnen.

4.3 Architektur des Prototyps

Matchmaking: Die Matchmaking Architektur basiert auf einem Client Server Modell, sowie dem Einsatz einer REST API, welche als Matchmaking- und Server-Runner Schnittstelle dient. Der oben beschriebene Ablauf zum Beitreten einer Lobby ist im folgenden Diagramm noch einmal aus technischer Sicht visualisiert:

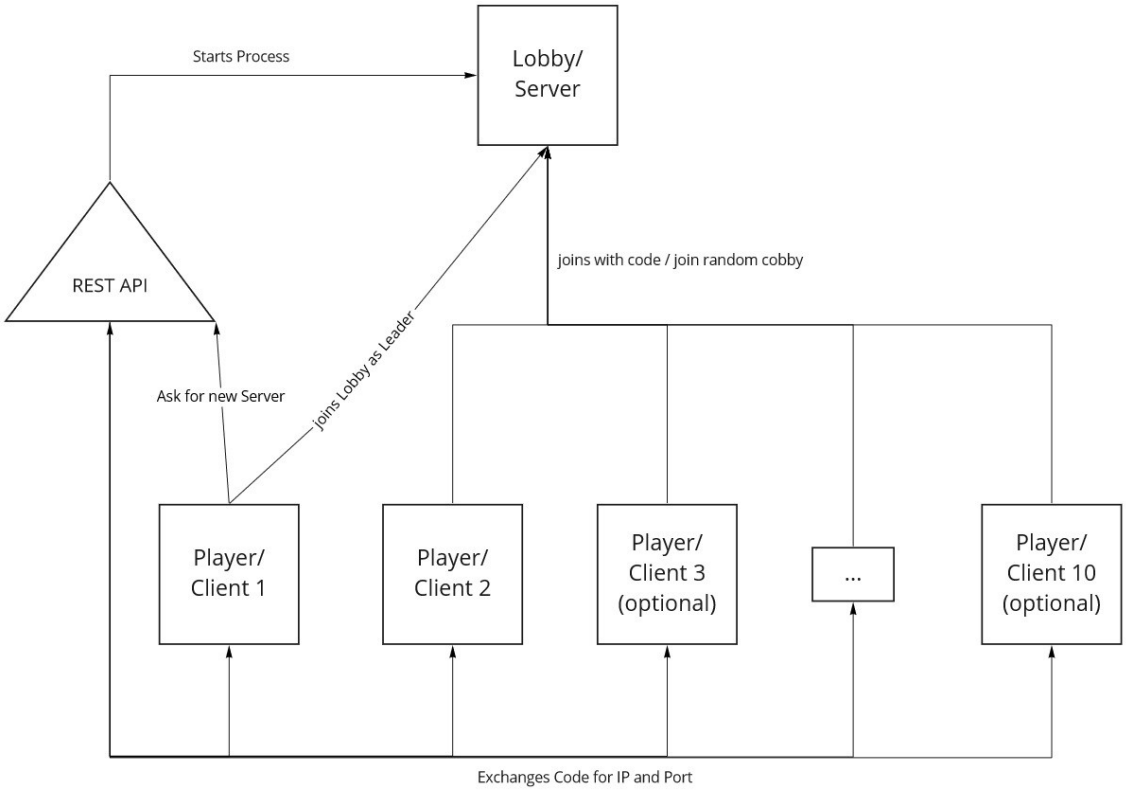

Abbildung 4.3: Veranschaulichung der Architektur des Matchmakings im Prototyp

Szenen:

Im Prototypen gibt es lediglich 2 Szenen: Die Menü-Szene (Menu Scene) und die Spielszene (Game Scene). Die Menu Scene gliedert sich in 2 verschiedene Sub-Menüs. Das 'Main Menu' wird zu Spielstart angezeigt, sowie wenn ein Spieler die Verbindung zu einem Server abbricht oder verliert. Das 'Lobby Menu' wird angezeigt, sobald ein Spieler einer Lobby beigetreten ist.

Die Game Scene wird geladen, sobald ein Spiel aus dem 'Lobby Menu' heraus gestartet wird. Das ist dann der Fall, sobald alle Mitglieder einer Lobby auf den Button 'Ready' gedrückt haben, und der Leiter dieser Lobby auf den Button 'Start Game' gedrückt hat.

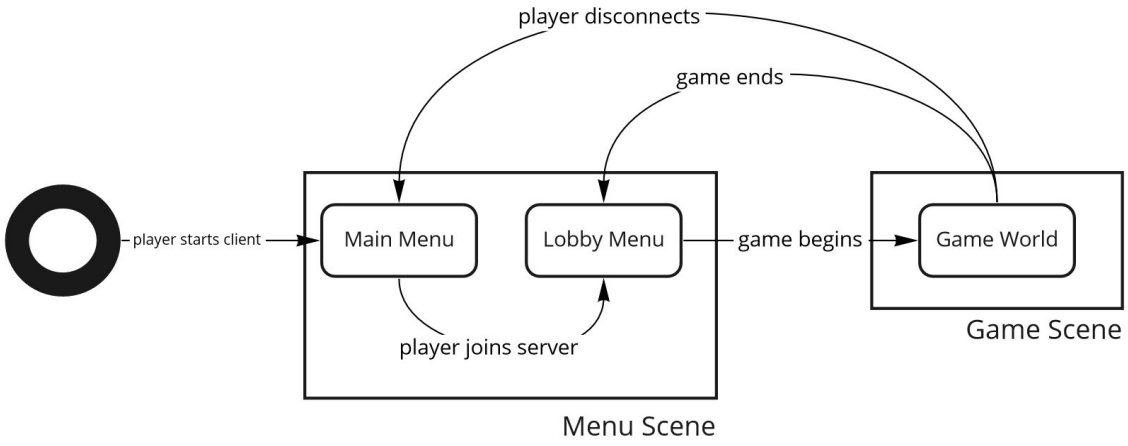

Abbildung 4.4: Veranschaulichung der Szenen-Architektur im Prototyp

Auf die Implementierungen der einzelnen Konzepte wird in den kommenden Sektionen im Detail eingegangen. Die folgende Grafik zeigt die Relationen zwischen den implementierten Konzepten untereinander:

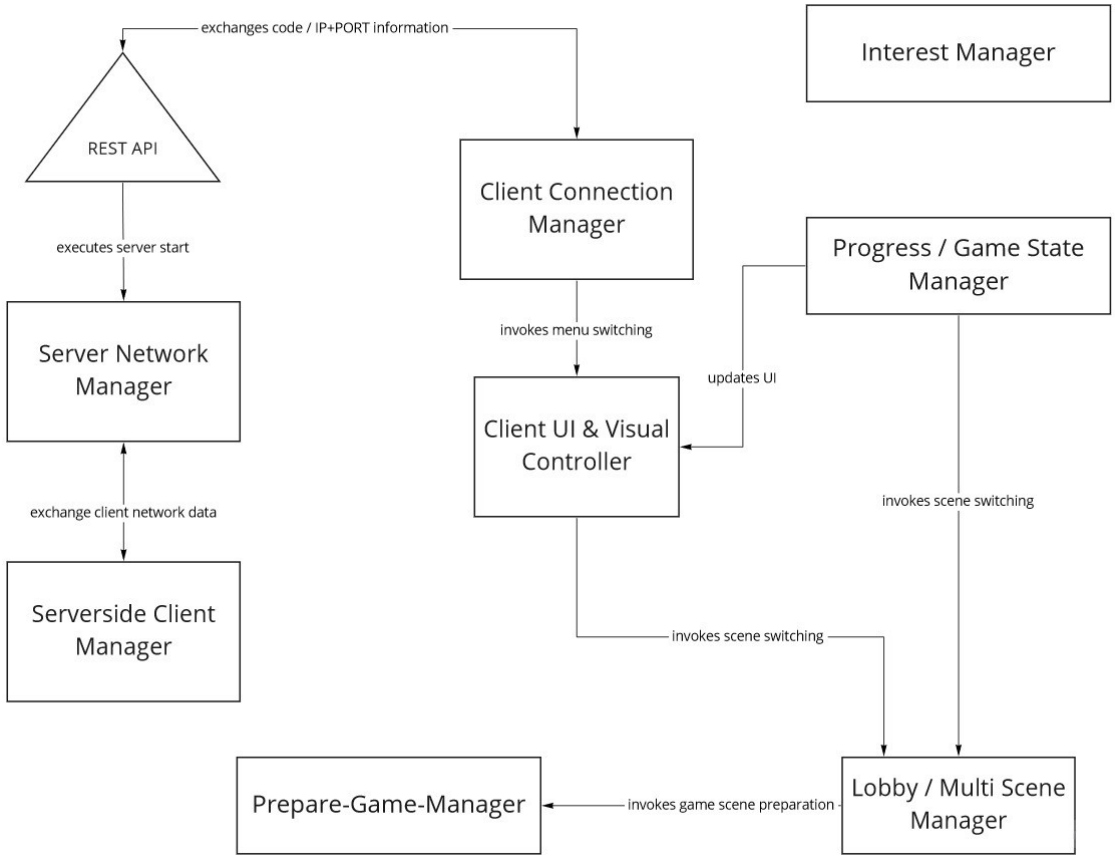

Abbildung 4.5: Veranschaulichung der Konzeptrelationen im Prototyp

4.4 Implementierung: API für Matchmaking & Server Runner

Für die Matchmaking & Server Runner API wurde auf eine externe Lösung zurückgegriffen, da eine eigene Entwicklung aus zeitlichen Gründen nicht möglich war. Genutzt wurde eine Lösung von Felix Brübach [Fel22]. Diese beinhaltet eine REST API sowie eine Verwaltung von beliebig vielen Serverthreads. Diese werden repräsentiert durch eine Aneinanderreihung von einem Adjektiv und Tiernamen als 'Identifier'(ID). Dieser Identifier wird in unserem Spielkonzept als 'Code' verwendet. Außerdem kann die REST API so konfiguiert werden, dass ein eigenes Datenmodell zusätzlich zu den IDs verwaltet werden. Die REST API erwartet ebenfalls eine Datei, welche sie zur Laufzeit als Server-Thread ausführen kann. Im Falle des Prototyps ist das eine Mirror Server-Instanz.

Die REST API akzeptiert die HTTP-Verben GET, POST, PUT und DELETE, mit denen server-relevante Daten abgefragt und Serverprozesse erzeugt, manipuliert, oder gelöscht werden können. Die Erzeugung der Serverprozesse erfolgt durch Ausführung der Mirror-Serverdatei und Übergabe der folgenden 2 Programm-Argumente:

'–port': Die nächste freie Port-Nummer auf der Hardware, auf welcher auch die REST API läuft

'–id': Die ID, welche die REST API für diesen Serverprozess erzeugt hat. Im Falle des Prototyps wird diese als 'Code' behandelt.

Der folgende Code zeigt die Methode 'ServerStart', welche innerhalb des Mirror-Serverprozesses aufgerufen wird, sobald die REST API den Prozess startet:

Listing 4.1: OwnKcpTransport.cs ServerStart()

```
1 public override void ServerStart()
2 {
3   ushort port = 7777;
4
5   string lobbyCode = 'No Code specified';
6
7   int portIndex = Array.FindIndex(System.Environment.GetCommandLineArgs()
        , elem => elem == '--port') + 1;
8
9   int codeIndex = Array.FindIndex(System.Environment.GetCommandLineArgs()
        , elem => elem == '--id') + 1;
10
11  if (portIndex > 1 && System.Environment.GetCommandLineArgs().Length >
        portIndex)
12  {
13    try
14    {
```

```
15        port = Convert.ToUInt16(System.Environment.GetCommandLineArgs()[
              portIndex]);
16      } catch
17      {
18        Debug.LogError('Invalid port specified: ' + System.Environment.
              GetCommandLineArgs()[portIndex]);
19      }
20    }
21    if(codeIndex > 1 && System.Environment.GetCommandLineArgs().Length >
          codeIndex)
22    {
23      try
24      {
25        lobbyCode = System.Environment.GetCommandLineArgs()[codeIndex];
26        GameNetworkManager.singleton.curLobbyCode = lobbyCode;
27      }
28      catch
29      {
30        Debug.LogError('Invalid code specified: ' + System.Environment.
              GetCommandLineArgs()[codeIndex]);
31      }
32    }
33
34    base.server.Start(port);
35    Debug.Log('Server started on Port ' + port + 'and with code ' +
          lobbyCode);
36  }
```

4.5 Implementierung: Client UI & Visual Controller

Der Client UI & Visual Controller spielt im Prototyp eine wesentliche Rolle. Er kommt in der Game Scene als Singleton [M. 09] zum Einsatz, wo er alle nötigen Methoden implementiert, um das lokale UI des Spielers manipulieren zu können. Hier sind nun 3 Beispiele aufgeführt:

Die folgenden 'Update'-Methode ist eine Unity bereitgestellte Schnittstelle. Der Rumpf der Update Methode ist nie vorgegeben, und muss vom Entwickler selbst implementiert werden. Unity führt diese Methode jedes 'Frame' (Einzelbild) [Wik21a] aus. Dies ermöglicht eine nahezu vollständige Echtzeitabfrage der Inputs (Tastatur / Maus / andere Eingangsgeräte) des Spielers.

Es wird dauerhaft abgefragt, ob ein Spieler die Taste 'E' oder 'Q' auf seiner Tastatur drückt. Wenn er das tut, und ebenfalls ein paar andere Randbedingungen erfüllt sind, so wird ein 'Linksklick' auf den im UI des Spielers sichtbaren Button simuliert.

Listing 4.2: InGameUiControllerScript.cs Update Method

```
1 private void Update()
2 {
3   if (Input.GetKeyDown(KeyCode.E) && hotkeyImage != null && hotkeyImage.
        sprite == HOTKEY_KEYBOARD_E && interactButton.GetComponent<Button
        >().interactable )
4   {
5     interactButton.GetComponent<Button>().onClick.Invoke();
6   }
7   if (Input.GetKeyDown(KeyCode.Q) && lightButton.GetComponent<Button>().
        interactable)
8   {
9     lightButton.GetComponent<Button>().onClick.Invoke();
10  }
11 }
```

Doch was heißt das genau? Im folgenden Code wird ein Event definiert, welches durch andere Klassen 'abonniert' werden kann. Wird dann das Event durch '.Invoke()' ausgelöst, so werden alle Funktionen in externen Skripten ausgeführt, die dieses Event abonniert haben. Hier zunächst die Definition des Events im InGameUiControllerScript und die Funktion 'clickInteractButton', die ausgeführt wird, wenn beispielsweise wie im obrigen Code die Taste 'E' auf der Tastatur gedrückt wird.

Listing 4.3: InGameUiControllerScript.cs OnInteractButtonClick Event

```
1 public event Action OnInteractButtonClick;
2
3 public void clickInteractButton()
4 {
5   OnInteractButtonClick?.Invoke();
6 }
```

Um den gesamten Aufrufstapel nachvollziehen zu können, folgt nun ein Beispiel aus einem anderen Script. Die Klasse 'HiderScript.cs' abonniert in seiner Methode 'RpcEnableHideButton' das Event des InGameUiControllerScript Singletons 'OnInteractButtonClick' mit seiner eigenen Methode 'OnHiderHideButtonClick'. Die Methode 'RpcEnableHideButton' wird dann ausgeführt, wenn der Server einem Spieler mit der Roller 'Hider' erlaubt, ein Versteck betreten zu dürfen.

Listing 4.4: HiderScript.cs Subscribe to InGameUiControllerScript Event

```
1 [TargetRpc]
2 private void RpcEnableHideButton()
3 {
4   InGameUiControllerScript.singleton.resetInteractButtonClickEvents();
5   InGameUiControllerScript.singleton.OnInteractButtonClick +=
        OnHiderHideButtonClick;
```

```
6   InGameUiControllerScript.singleton.setInteractButtonEnabled(true);
7   InGameUiControllerScript.singleton.setInteractButtonTextAndHotkeyImage
        ('Hide');
8 }
```

Sollte der Hider nun auf seinen 'Interact'-Button drücken, so wird die folgende Methode 'OnHiderHideButtonClick' aufgerufen. Diese ruft eine weitere Methode 'CmdHideHider' auf, welche einen Remote Procedure Call[.0522] zum Server darstellt.

Listing 4.5: HiderScript.csOnHiderHideButtonClick() Method

```
1
2 [Client]
3 private void OnHiderHideButtonClick()
4 {
5   if (enabled)
6   {
7     CmdHideHider();
8   }
9 }
```

Der Server führt nun innerhalb des Methodenrumpfs von 'CmdHideHider' einige Überprüfungen durch, ändert Server-interne Variablen und sorgt somit dafür, dass alle Spieler darüber Bescheid wissen, dass sich der Spieler nun in einem Versteck befindet.

Listing 4.6: HiderScript.cs Subscribe to InGameUiControllerScript Event

```
1
2   [Command]
3   private void CmdHideHider()
4   {
5     if(vicinityScript.getHideoutObjectScript() != null)
6     {
7       HideableObjectScript curHideoutScript = vicinityScript.
            getHideoutObjectScript();
8       if (!vicinityScript.getHideoutObjectScript().getIsTaken())
9       {
10        GetComponent<PlayerBaseScript>().setCurHideout(curHideoutScript.
              gameObject);
11        curHideoutScript.setIsTaken(true);
12        curHideoutScript.setCurrentHider(gameObject);
13        isHiding = true;
14        GetComponent<PlayerBaseScript>().playerLightEnabled = false;
15        GameNetworkManager.FindObject(gameObject, 'FeetTrigger').
              SetActive(!isHiding);
16        RpcNotifyHideState(true, GameNetworkManager.singleton.
              getConfiguredPlayerSpeed());
17        RpcEnableLeaveHideoutButton();
```

```
18          GetComponent<PlayerBaseScript>().TargetEnableLightButton(false);
19        }
20        else
21        {
22          RpcDisableInteractButton();
23        }
24      }
25    }
```

Die nächste Beispielmethode implementiert die Logik, den Interact Button selbst aktiv- und inaktiv zu setzen. Der Interact Button selbst verfügt ebenfalls über ein 'HotkeyImage', welches einem visuellen Indikator gleich kommt, der die aktuell zu drückende Taste auf der Tastatur darstellt. Dieser Indikator wird bei jedem Aufruf von 'setInteractButtonEnabled' ebenfalls an - oder ausgeschaltet.

Listing 4.7: InGameUiControllerScript.cs setInteractButtonEnabled

```
1 public void setInteractButtonEnabled(bool enabled)
2   {
3     interactButton.GetComponent<Button>().interactable = enabled;
4     if(hotkeyImage != null)
5     hotkeyImage.gameObject.SetActive(enabled);
6   }
```

Das letzte Beispiel ist eine Implementierung eines simplen 'Flip-Mechanismus' beim Betätigen des 'Light-Buttons' (An- und ausschalten der Taschenlampe eines Spielers). Diese Logik tauscht lediglich das Bild (Sprite) aus, welches auf dem Button liegt.

Listing 4.8: InGameUiControllerScript.cs flipLightButtonImage

```
1 public void flipLightButtonImage()
2 {
3   if (!lightOn)
4   {
5     lightButton.GetComponent<Image>().sprite = lightOnSprite;
6   }
7   else
8   {
9     lightButton.GetComponent<Image>().sprite = lightOffSprite;
10  }
11  lightOn = !lightOn;
12 }
```

4.6 Implementierung: Server Network Manager

In der Implementierung des Prototyps ist der Server Network Manager ein Teil der 'GameNetworkManager' Klasse. Diese implementiert sowohl client- als auch serverseitige Netzwerkfunktionen. Die nicht vorhandene Separierung von Client- und Serverfunktionen ist dem Mirror Framework geschuldet. Die Klasse 'GameNetworkManager' ist ebenfalls ein Singleton und erbt die Klasse 'NetworkManager', welche von Mirror bereitgestellt wird. Durch die Vererbung ist es möglich, virtuelle Methoden[Bil22] der NetworkManager Klasse zu überschreiben, netzwerkseitige Events abzufangen und mit eigener Logik auf diese zu reagieren.

Im folgenden Beispiel wird die virtuelle Methode 'OnServerConnect' überschrieben, um zu überprüfen, ob ein Spieler einer Spiel-Session beitreten darf. Sollte sich diese bereits in der Spiel-Szene befinden, so wird eine neue Message erzeugt, welche an den Client geschickt wird, welcher aktuell versucht sich zu verbinden. Das struct 'Message' nutzt ein von Unity bereitgestelltes Interface 'NetworkMessage', welches eine leichtgewichtige Art und Weise der Netzwerk-Kommunikation ermöglicht. Das gleiche Schema wird im zweiten 'if' Block benutzt. Hier wird überprüft, ob die maximale Anzahl an Spielern innerhalb einer Lobby bereits erreicht ist.

Listing 4.9: GameNetworkManager.cs OnServerConnect() und Message struct

```
1 public struct Message : NetworkMessage
2 {
3   public string message;
4   public MessageType messageType;
5 }
6
7 public override void OnServerConnect(NetworkConnection conn)
8 {
9   // If the game server is already in gameScene, forbid access to game
10  if (singleton.getNetworkSceneName() == 'Assets/Scenes/GameScene.unity')
11  {
12    Message msg = new Message()
13    {
14      message = 'Game has already started',
15      messageType = MessageType.gameAlreadyStartedError
16    };
17    conn.Send(msg);
18    conn.Disconnect();
19  }
20
21  // Forbid access to a lobby, if the lobby is already at maxPlayers (
        configurable inside GameNetworkManager Component)
22  if(lobbyPlayers.Count == maxPlayers)
23  {
```

```
24      Message msg = new Message()
25      {
26        message = 'Maximum Players reached for this lobby',
27        messageType = MessageType.tooManyPlayersError
28      };
29      conn.Send(msg);
30      conn.Disconnect();
31    }
32  }
```

Als Gegenstück zu 'OnServerConnect' gibt es ebenfalls die virtuelle Methode 'OnServer-Disconnect', welche dann aufgerufen wird, wenn ein Spieler den Server gewollt, oder ungewollt (bspw. durch Netzwerkabbruch) verlässt. Im Prototyp wurde diese Methode ebenfalls überschrieben und im GameNetworkManager selbst implementiert.

Zunächst wird überprüft, ob die Anzahl aller Spieler, welche sich auf dem Server befinden, nach dem Beenden der aktuell abgefangenen und beendeten Client-Verbindung '0' beträgt. Falls ja, wird der Server-Prozess beendet.

Sollten sich noch Spieler auf dem Server-Prozess befinden, werden je nach Spielszene verschiedene Szenarien durchlaufen. Wenn die Spielszene aktuell läuft, so wird überprüft, ob die Anzahl an Spielern mit der Rolle 'Hider' oder 'Seeker' aktuell '0' beträgt. Falls eine der beiden Fälle zutrifft, so wird das Spiel zugunsten des konkurrierenden Teams beendet.

Wenn sich die Spiel-Session aktuell in der Menü-Szene befindet, so werden Lobby-Relevante Variablen aktualisiert und zunächst überprüft, ob der Spieler, welche gerade den Server verlassen hat, den Leader-Status hatte. Sollte dieser Fall zutreffen, wird der Leader-Status weiter gegeben. Zuletzt werden alle Spieler über den neuen Zustand der Lobby informiert.

Listing 4.10: GameNetworkManager.cs OnServerDisconnect()

```
1 public override void OnServerDisconnect(NetworkConnection conn)
2 {
3   if ((NetworkServer.connections.Count == 0)
4   {
5     Debug.Log('Server shutdown because all Players left');
6     Application.Quit();
7   }
8
9   if (SceneManager.GetActiveScene().path == gameScene)
10  {
11    // if Seeker disconnected and no seeker left, let hiders win
12    if(inGameProgressManager.getTotalSeekerCount() == 0)
13    {
14      inGameProgressManager.notifyHidersWinEvent();
```

```
15      }
16
17      // If Hider disconnected and no hiders left, let seekers win
18      if(inGameProgressManager.getTotalHidersCount() == 0)
19      {
20        inGameProgressManager.notifySeekerWinEvent();
21      }
22    }
23    else if(conn.identity != null)
24    {
25      lobbyPlayers.Remove(conn);
26      LobbyPlayer curPlayerScript = conn.identity.GetComponent<LobbyPlayer
            >();
27
28      if (curPlayerScript.isLeader)
29      {
30        // Pass Leader State to next Player
31        var newLeader = lobbyPlayers.First().Value;
32        newLeader.isLeader = true;
33        leader = newLeader;
34        leader.TargetNotifyLobbyReady(isLobbyReady);
35      }
36
37      var index = 0;
38      // Pass the index
39      foreach (var lobbyPlayer in lobbyPlayers.Values)
40      {
41        lobbyPlayer.index = index;
42        index++;
43      }
44    }
45
46    UpdateLobbyReadyState();
47  }
```

Weitere virtuelle Methoden, welche der Prototyp implementiert hat, sind:

'OnServerChangeScene' - Diese Methode wird ausgeführt, sobald ein Szenenwechsel bevorsteht, aber noch nicht vollzogen ist.

'OnServerSceneChanged' - Diese Methode wird ausgeführt, sobald ein Szenenwechsel vollständig vollzogen ist.

4.7 Implementierung: Lobby / Multi Scene Manager

Das Spielkonzept beinhaltet wie im Teil der Spielidee und Architektur des Prototyps beschrieben auch eine Lobbymechanik. Die Ideen aus dem Konzept des Lobby / Multi Scene Managers wurden aufgrund der geringen Anforderungen des Prototyps ebenfalls in der Klasse 'GameNetworkManager.cs' untergebracht. Dies hat den Vorteil, dass man die Lobby-spezifischen Variablen direkt mit den überschriebenen virutellen Methoden der Mirror Klasse 'NetworkManager' kombinieren kann.

Im folgenden Beispiel werden die server-internen Variablen aufgeführt, welche im Prototyp für die Szenen-Übergreifenden Lobby-Verwaltung verantwortlich sind:

1. 'List<GameRule>' wird verwendet, um die für diesen Server-Prozess konfigurierten Spielregeln zu verwalten. Diese sind durch den Leader einer Lobby konfigurierbar. Beispiele für konfigurierbare Spielregeln sind:
- Total Game Time - Nach wieviel Sekunden wird das Spiel automatisch zu Gunsten der Spieler entschieden, welche zum Team 'Seeker' gehören.
- Player Base Speed - Die Grundgeschwindigkeit der Spieler-Avatare.
- Daytime - Einstellung für den Wechsel zwischen Tag - und Nacht.

2. 'LobbyPlayer leader' beinhaltet das Spieler-Objekt, welches der Server als Leader einer Lobby ansieht.

3. 'Dictionary<NetworkConnection, LobbyPlayer> lobbyPlayers' in diesem Dictionary wird pro Spieler die dazugehörige 'NetworkConnection' Objekt und das 'LobbyPlayer' Spieler-Objekt verwaltet. 'NetworkConnection' ist eine Mirror-spezifische Klasse, welche alle Netzwerk-relevanten Informationen zu einem Spieler verwaltet (Bspw. die IP-Adresse oder eine für diesen Spieler einzigartige Verbindungs-ID).

4. 'bool isLobbyReady' ist ein boolscher Wert, welcher nur dann den Wert 'true' annimmt, sobald alle Spieler durch das Drücken auf einen 'Ready'-Button signalisiert haben, dass sie bereit für den Wechsel in die Spiel-Szene sind.

5. 'uint maxPlayers' ist ein Wert, welcher bestimmt wieviele Spieler maximal einer Lobby beitreten dürfen.

6. 'uint keepLobbyAliveTimeInSeconds' ist ein Wert, welcher festlegt nach wie viel Sekunden Inaktivität der Spieler ein Server-Prozess beendet wird.

Listing 4.11: GameNetworkManager.cs Lobby Variables

```
1   public List<GameRule> gameRules = new List<GameRule>();
2
3   private LobbyPlayer leader = null;
4
5   private Dictionary<NetworkConnection, LobbyPlayer> lobbyPlayers =
6   new Dictionary<NetworkConnection, LobbyPlayer>();
7
8   private bool isLobbyReady = false;
9
10  private uint maxPlayers = 10;
11  private uint keepLobbyAliveTimeInSeconds = 600;
```

Die folgenden Methoden sorgen dafür, dass vor dem Übergang in die Spiel-Szene die nun nicht mehr benötigten Lobby-Spielerobjekte zerstört werden und im Anschluss der gewünschte Übergang in die Spiel-Szene erfolgt. Hierfür stellt Mirror die Methode 'ServerChangeScene' bereit, welche dafür sorgt, dass der Szenenwechsel sowohl beim Server, als auch bei allen verbundenen Clients vollzogen wird.

Listing 4.12: GameNetworkManager.cs StartGame

```
1 public void StartGame()
2 {
3   clearLobbyData();
4   ServerChangeScene(gameScene);
5 }
6
7 private void clearLobbyData()
8 {
9   foreach(LobbyPlayer lobbyPlayer in lobbyPlayers.Values)
10  {
11    NetworkServer.Destroy(lobbyPlayer.gameObject);
12  }
13 }
```

Der Aufruf der Initialisierung des Szenenwechsels zurück in die Menü-Szene geschieht aus Komplexitätsgründen aus dem InGameProgressManager heraus. Die nähere Erläuterung folgt in der Sektion Implementierung: Progress / Game-State Manager

4.8 Implementierung: Client Connection Manager

Die Prinzipien des Client Connection Manager Konzepts wurden im Prototyp innerhalb der Klasse 'ConnectionScript.cs' realisiert. Die Hauptaufgabe ist der Kommunikationsaustausch mit der Matchmaking API sowie der Verbingdungsaufbau zu einer Mirror-

Serverinstanz. Für diesen Kommunikationsaustausch wurde das Open Source Package 'Rest Client' von Juan Nicholls (proyecto26) benutzt [Git22].

Im folgenden Code Beispiel wird zunächst das Datenmodell einer Lobby aus der Sicht der Matchmaking API und dem 'ConnectionScript' aufgezeigt. Eine kurze Erläuterung der einzelnen Felder:

- 'bool isPublic' sagt aus, ob eine Lobby öffentlich bereitgestellt wurde. Falls ja, können andere Spieler auch ohne Zugangs-Code über die 'Join Random Lobby' Funktion des Prototyps dieser Lobby beitreten.

- 'string adress' gibt die IP-Adresse des Servers an, auf dem die der Mirror-Serverprozess zu dieser Lobby gestartet wurde.

- 'int port' gibt den Port des Mirror-Serverprozesses an, den der Server für diesen Prozess reserviert hat.

- 'string code' enthält einen Zugangsschlüssel, den Clients nutzen können, um im Austausch mit der Matchmaking API das passende Lobby-Objekt zu erhalten, mit dem sie sich dann (mithilfe des adress + port Felds) ebenfalls auf den gleichen Mirror-Serverprozess verbinden können.

Listing 4.13: ConnectionScript.cs Matchmaking Data

```
1 // The Data which is necessary for describing a lobby. This Information
      is synchronized between the REST API and the Mirror Server & Client
2 private class Lobby
3 {
4   public bool isPublic;
5   public string address;
6   public int port;
7   public string code;
8 }
9
10 private class PublicLobbys
11 {
12   public Lobby[] lobbys;
13 }
```

Sollte ein Spieler im Hauptmenü auf den Button 'Create Lobby' drücken, so wird eine POST Anfrage an die Matchmaking API geschickt, diese liefert ein neu erzeugtes Lobby-Objekt zurück, mit welchem sich der Spieler auf den neu erzeugten Mirror-Serverprozess verbindet. Folgender Code zeigt die Implementierung:

Listing 4.14: ConnectionScript.cs createLobby()

```
1 private void createLobby()
2 {
3   RestClient.Post<Lobby>(apiBaseUrl + '/v0/lobbies', '{\'isPublic\': true
        }').Then(lobby =>
4   {
5     string[] splitAddress = lobby.address.Split(new string[] { '://' },
          StringSplitOptions.None);
6
7     GameNetworkManager.singleton.StartClient(new UriBuilder(splitAddress
          [0], splitAddress[1], lobby.port).Uri);
8   }).Catch(error =>
9   {
10    onFailure();
11    Debug.LogError('Error when creating lobby: ' + error);
12  });
13 }
```

Im Fall, dass ein Spieler auf den Button 'Join Lobby' drückt, wird die Zeichenkette, welche der Nutzer in das entsprechende 'Code-Input-Feld' eingegeben hat, der Methode 'joinWithCode' übergeben. Diese überprüft zunächst, ob der Spieler keinen Code eingegeben hat. Falls die übergebene Zeichenkette nicht leer ist, führt das 'ConnectionScript' eine GET Anfrage an die Matchmaking API aus, welche intern überprüft, ob eine Lobby mit diesem Zugangsschlüssel existiert. Falls dies der Fall ist, verbindet sich der Spieler automatisch mit der entsprechenden Lobby.

Listing 4.15: ConnectionScript.cs joinWithCode()

```
1 private void joinWithCode(string code)
2 {
3   registerCallbacks();
4   if (code == '')
5   {
6     noCodeEnteredFailure();
7     return;
8   }
9
10  RestClient.Get<Lobby>(GameNetworkManager.singleton.frapiBaseUrl + '/v0/
        lobbies/' + code).Then(lobby =>
11  {
12    string[] splitAddress = lobby.address.Split(new string[] { '://' },
          StringSplitOptions.None);
13    GameNetworkManager.singleton.StartClient(new UriBuilder(splitAddress
          [0], splitAddress[1], lobby.port).Uri);
14  }).Catch(error =>
15  {
16    Debug.LogError('Error when joining lobby: ' + error);
17    onFailure();
```

```
18     }
19 }
```

In den beiden obigen Beispielen wird beim Auslösen des Catch-Falls die Methode 'onFailure' aufgerufen. Dieser Fall tritt bei verschiedenen Szenarien ein. Beispielsweise wenn die Komminikation mit der REST API nicht möglich ist, oder diese einen 4XX oder 5XX HTTP-Status-Code liefert. Die Methode sorgt dafür, dass der Nutzer zurück ins Main-Menü geschickt wird, und ihm eine Nachricht angezeigt wird, dass die Verbindungsversuche gescheitert ist.

Listing 4.16: ConnectionScript.cs onFailure()

```
1 private void onFailure()
2 {
3   unregisterCallbacks();
4   GameNetworkManager.singleton.curNetworkMessage.message = 'Network Error
        : Could not Connect to Server';
5   changeToMainMenu(true);
6 }
```

Die Implementierung einer 'onSuccess' Funktion (das Gegenstück zu 'onFaliure') innerhalb des Connection-Managers ist nicht nötig, da bei erfolgreicher Verbindungsherstellung Mirror-spezifische Events und Handler-Funktionen für die weitere Logik sorgen. Diese sind nicht mehr Teil des 'ConnectionScript'.

4.9 Implementierung: Serverside Client Manager

Das Konzept des Serverside Client Managers wird nahezu vollständig durch mirrorinterne Prozesse gelöst. Die Verwaltung aller Clients geschieht durch die 'GameNetworkManager' Klasse, welche von der Mirror Klasse 'NetworkManager' erbt. Die Superklasse 'NetworkManager' wiederum nutzt eine andere Mirror-Klasse 'NetworkServer', diese beinhaltet ein Dictionary, welches die Verbindungen zu allen Clients verwaltet. Im folgenden Code ist das Dictionary, sowieso die dazu gehörenden Methoden 'AddConnection' und 'RemoveConnection' zu sehen.

Listing 4.17: Mirror Class NetworkServer.cs Connection Handling

```
1 /// <summary>
2 /// A list of local connections on the server.
3 /// </summary>
4 public static Dictionary<int, NetworkConnectionToClient> connections =
      new Dictionary<int, NetworkConnectionToClient>();
5
```

```
6 /// <summary>
7 /// <para>This accepts a network connection and adds it to the server.</
      para>
8 /// <para>This connection will use the callbacks registered with the
      server.</para>
9 /// </summary>
10 /// <param name='conn'>Network connection to add.</param>
11 /// <returns>True if added.</returns>
12 public static bool AddConnection(NetworkConnectionToClient conn)
13 {
14   if (!connections.ContainsKey(conn.connectionId))
15   {
16     // connection cannot be null here or conn.connectionId
17     // would throw NRE
18     connections[conn.connectionId] = conn;
19     conn.SetHandlers(handlers);
20     return true;
21   }
22   // already a connection with this id
23   return false;
24 }
25
26
27 /// <summary>
28 /// This removes an external connection added with AddExternalConnection
      ().
29 /// </summary>
30 /// <param name='connectionId'>The id of the connection to remove.</param
      >
31 /// <returns>True if the removal succeeded</returns>
32 public static bool RemoveConnection(int connectionId)
33 {
34   return connections.Remove(connectionId);
35 }
```

In der aktuellen Version des Prototyps werden spielerbezogene 'Sonderinformationen' wie Spielername und Rolle im 'GameNetworkManager' mitverwaltet. Die Aufgabe des Serverside Client Managers wird in der Komponente, welche in die der Sektion Implementierung: Lobby / Multi Scene Manager beschrieben wird, mitverwaltet.

Diese Tatsache ist allerdings an die Randbedingungen geknüpft, dass für das den Prototypen das Mirror Framework genutzt wurde, welches viele Funktionen auf die überschriebene 'NetworkManager' Klasse abwälzt, sowie dass der Prototyp keine große Anzahl an 'Sonderinformationen' (wie Charakterindividualisierung, Account-Informationen / In-Game-Shop Daten etc.) speichern muss. Da der Komplexitätsaufwand für eine gesonderte Handhabung dieser Informationen bei steigender Anforderungen enorm zunimmt, ist es ratsam, die Komponente 'Serverside Client Manager' gesondert zu implementieren.

4.10 Implementierung: Prepare-Game-Manager

Die Klasse 'PrepareGameManager' sorgt innerhalb des Prototyps dafür, dass anhand der vom Lobby-Leiter konfigurierten Anzahl an Objekten eine entsprechende Menge an zufällig ausgewählten 'Collectable'-Objekten im Spielfeld platziert werden. Anschließend wählt der 'PrepareGameManager' zufällig ein von drei, für dieses spezifische 'Collectable'-Objekt mögliches 'Destination'-Objekt aus, welches als Zielpunkt dient. Die folgende Methode 'spawnObjects' führt diese Logik aus. Zunächst eine kurze Erläuterung der Variablen und aufgerufenen Methoden innerhalb von 'spawnObjects':

1. Das Feld 'spawnableCollectableObjects' enthält alle 'Collectable'-Objekte, 'spawnableDestinationObjects' alle 'Destination' Objekte, welche gespawnt werden können.

2. 'possibleCollectableSpawnPoints' und 'possibleDestinationSpawnPoints' beinhalten alle möglichen Spawnpunkte für 'Collectable' und 'Destination' Objekte.

3. 'amountOfCollectablesSetting' ist die vom Lobby-Leiter konfigurierte Anzahl an 'Collectable' Objekten, die in der Spielwelt platziert werden sollen.

4. 'popCollectableSpawnpointAt' gibt einen Spawnpunkt für ein 'Collectable'-Objekt zurück für einen gegebenen Index.

5. 'popCollectableObjectAt' gibt ein 'Collectable' Objekt zurück für einen gegebenen Index.

6. 'popRelatedDestinationSpawnPoints' gibt drei Spawnpunkte zurück für einen gegebenen Index.

7. 'popRelatedDestinationObjects' gibt drei 'Destination' Objekte zurück für einen gegebenen Index.

Listing 4.18: PrepareGameManager.cs Variablen und spawnObjects()

```
1
2 private List<GameObject> spawnableCollectableObjects;
3 private List<GameObject> spawnableDestinationObjects;
4
5 private List<Transform> possibleCollectableSpawnPoints;
6 private List<Transform> possibleDestinationSpawnPoints;
7
8 private void spawnObjects()
9 {
10   int amountOfCollectablesSetting = GameNetworkManager.singleton.
         getConfiguredCollectableItemsSetting();
11   for (int i = 0; i < amountOfCollectablesSetting; i++)
12   {
```

```
13      // take one random index from collectable object spawnpoint, use this
            index to instantiate collectable object [index] inside
            collectable object spawnpoint [index],
14      // remove the index in both lists
15      // find corresponding 3 destination objects, pick a random one of
            these 3, make relation collectable <--> Destination
16
17      int randomIndex = new Random().Next(possibleCollectableSpawnPoints.
            Count - 1);
18      Transform curCollectableSpawnPoint = popCollectableSpawnpointAt(
            randomIndex);
19
20      GameObject curCollectableGameObject = popCollectableObjectAt(
            randomIndex);
21      GameObject collectInstance = Instantiate(curCollectableGameObject,
            curCollectableSpawnPoint.position, curCollectableSpawnPoint.
            rotation);
22      NetworkServer.Spawn(collectInstance);
23
24      // Destroy the the spawnpoint that is used to place current
            collectable Object
25      NetworkServer.Destroy(curCollectableSpawnPoint.gameObject);
26
27      List<Transform> relatedDestinationSpawnPoints =
            popRelatedDestinationSpawnPoints(randomIndex);
28      List<GameObject> relatedDestinationObjects =
            popRelatedDestinationObjects(randomIndex);
29
30
31      // pick a random value between e.g. 0-2 or 3-5 or 6-8 .. etc. (based
            on random Collectable index)
32      int randomRelatedDestinationObjectPickIndex = new Random().Next(
            relatedDestinationObjects.Count);
33      Transform curDestinationSpawnpoint = relatedDestinationSpawnPoints[
            randomRelatedDestinationObjectPickIndex];
34      GameObject curDestinationObject = relatedDestinationObjects[
            randomRelatedDestinationObjectPickIndex];
35
36      // Spawn Destination Object, link it with the Collect Object
37      GameObject destinationInstance = Instantiate(curDestinationObject,
            curDestinationSpawnpoint.position, curDestinationSpawnpoint.
            rotation);
38      destinationInstance.GetComponent<DestinationObjectScript>().
            expectedObject = collectInstance;
39      NetworkServer.Spawn(destinationInstance);
40    }
41    destroyUnusedCollectableSpawnPoints();
42  }
```

Der 'PrepareGameManager' sorgt außerdem für die Verwaltung der Spawn-Punkte aller Spieler. Innerhalb der Variable 'possiblePlayerSpawnPoints' werden diese gespeichert. Die Methode 'fillPossiblePlayerSpawnPoints' sorgt dafür, dass die Variable mit Werten gefüllt wird.

Listing 4.19: PrepareGameManager.cs fillPossiblePlayerSpawnPoints()

```
1 private List<Transform> possiblePlayerSpawnPoints;
2
3 public void fillPossiblePlayerSpawnPoints()
4 {
5   possiblePlayerSpawnPoints = new List<Transform>();
6   GameObject[] portals = GameObject.FindGameObjectsWithTag('Portal');
7   foreach (GameObject portal in portals)
8   {
9     possiblePlayerSpawnPoints.Add(portal.transform);
10  }
11 }
```

Mit der Methode 'popRandomPlayerSpawnPoint' gibt einen zufälligen Spieler-Spawnpunkt zurück, und entfernt ihn von der Liste der noch möglichen Spieler-Spawnpunkte.

Listing 4.20: PrepareGameManager.cs popRandomPlayerSpawnPoint()

```
1 public Transform popRandomPlayerSpawnPoint()
2 {
3   int randomIndex = new Random().Next(possiblePlayerSpawnPoints.Count);
4   Transform result = possiblePlayerSpawnPoints[randomIndex];
5   possiblePlayerSpawnPoints.Remove(possiblePlayerSpawnPoints[randomIndex
        ]);
6   return result;
7 }
```

Aufgerufen wird die Methode 'popRandomPlayerSpawnPoint' innerhalb der Lobby Manager Implementierung in der Methode 'replaceLobbyPlayersWithGamePlayers'. In dieser Methode werden alle Lobby-Spielerobjekte mit InGame-Spielerobjekten (Charakteren) ersetzt. Unter anderem wird in diesem Ersetzungsprozess der neue, zufällige Spawn-Punkt benötigt, welcher 'popRandomPlayerSpawnPoint' liefert.

Listing 4.21: GameNetworkManager.cs replaceLobbyPlayersWithGamePlayers()

```
1 private void replaceLobbyPlayersWithGamePlayers()
2 {
3   int firstSeekerIndex = new Random().Next(NetworkServer.connections.
        Count);
4   int secondSeekerIndex = -1;
5
```

```
6   // Only set secondSeekerIndex if Leader has Configured 2 Seekers & we
        are at least 3 Players
7
8   if(getConfiguredSeekerAmount() == 2 && getLobbyPlayerCount() >= 3)
9   {
10    while(secondSeekerIndex == -1 || secondSeekerIndex ==
          firstSeekerIndex)
11    {
12      secondSeekerIndex = new Random().Next(NetworkServer.connections.
            Count);
13    }
14  }
15
16  int currentIndex = 0;
17  bool isDayTime = getConfiguredDayTimeSetting());
18
19  foreach (var pair in lobbyPlayers)
20  {
21    var conn = pair.Key;
22    var lobbyPlayer = pair.Value;
23
24    Transform spawnPoint = prepareGameManager.popRandomPlayerSpawnPoint()
          ;
25    Vector3 spawnPosition = new Vector3(spawnPoint.position.x, 0.88f,
          spawnPoint.position.z);
26    GameObject gamePlayerInstance = Instantiate(gamePlayerPrefab,
          spawnPosition, spawnPoint.rotation);
27
28    Role curRole = (currentIndex == firstSeekerIndex || currentIndex ==
          secondSeekerIndex) ? Role.Seeker : Role.Hider;
29    gamePlayerInstance.GetComponent<PlayerBaseScript>().role = curRole;
30    gamePlayerInstance.GetComponent<PlayerBaseScript>().playerName =
          lobbyPlayer.playerName;
31    gamePlayerInstance.GetComponent<PlayerBaseScript>().
          globalLightEnabled = isDayTime;
32    gamePlayerInstance.GetComponent<PlayerBaseScript>().
          playerLightEnabled = !isDayTime;
33    gamePlayerInstance.GetComponent<PlayerMovementScript>().
          setPlayerSpeed(getConfiguredPlayerSpeed());
34
35    NetworkServer.ReplacePlayerForConnection(conn, gamePlayerInstance,
          true);
36    currentIndex++;
37  }
38  lobbyPlayers.Clear();
39 }
```

4.11 Implementierung: Progress / Game-State Manager

Das Konzept des Progress / Game State Managers wurde innerhalb des Prototyps als Klasse 'InGameProgressManager' realisiert. Dieser verwaltet ein globaler Timer (Stoppuhr), welche bei dem Erreichen des Wertes '0' das Spiel automatisch für die Gruppe der Spieler mit der Rolle 'Seeker' beendet.

Der folgende Code zeigt, wie der globale Timer gestartet wird. Die Methode 'startGameTimer' wird serverseitig ausgeführt, sobald ein Szenenwechsel in die Spiel-Szene erfolgt. Außerdem wird beim Starten des Timers eine Netzwerk-Nachricht initialisiert, die bei jedem 'Tick' des Timers an alle Clients gesendet wird. Diese Nachricht enthält die Information, wie viel Sekunden noch übrig sind. Ein 'Tick' entspricht dem Ablauf einer Sekunde.

Listing 4.22: InGameProgressManager.cs global Game Time Handling

```
1 private Timer gameTimer;
2 private Message curGameTimeLeftMsg;
3
4 public void startGameTimer(uint totalGameTime)
5 {
6   gameStarted = true;
7
8   gameTimer = new Timer(1f, totalGameTime, handleTick);
9   TimersManager.SetTimer(this, gameTimer);
10
11   curGameTimeLeftMsg = new Message()
12   {
13     message = 'Game Time Left: ' + Math.Round(gameTimer.RemainingTime()),
14     messageType = MessageType.gameTimeLeftNotification
15   };
16 }
17
18 private void handleTick()
19 {
20   if (gameTimer.RemainingTime() > 0)
21   {
22     countDownGameTime();
23   }
24
25   else
26   {
27     notifySeekerWinEvent();
28   }
29 }
30
31 private void countDownGameTime()
32 {
```

```
33   curGameTimeLeftMsg.message = 'Game Time Left: ' + Math.Round(gameTimer.
         RemainingTime());
34
35   // Notify each client about the server sided game time left
36   foreach (var conn in NetworkServer.connections)
37   {
38     conn.Value.Send(curGameTimeLeftMsg);
39   }
40 }
```

Darüber hinaus speichert der 'InGameProgressManager' die Information, wie viele 'Collectable' Objekte die Spielergruppe mit der Rolle 'Hider' bereits zu ihrem Ziel gebracht haben. Erreicht die Variable 'totalDeliveredItems' den Wert, welcher der Lobby-Leiter innerhalb der Lobby Szene konfiguriert hat, so hat die Spielergruppe mit der Rolle 'Hider' die Spielrunde gewonnen.

Listing 4.23: InGameProgressManager.cs Item Devlivery Handling

```
1 private ushort totalDeliveredItems = 0;
2
3 public void incrementTotalDeliveredItems()
4 {
5   totalDeliveredItems++;
6   if(totalDeliveredItems >= (GameNetworkManager.singleton.
        getConfiguredCollectableItemsSetting()))
7   {
8     notifyHidersWinEvent();
9   }
10 }
```

Ist das Spiel für eine Partei gewonnen, so gibt der 'InGameProgressManager' die Information an alle Clients per Netzwerk-Nachricht weiter, und leitet die serverseitigen Konsequenzen ein. Der folgende Code zeigt den Code eines Sieges für die Spielergruppe 'Hider'.

Der Aufruf von 'StartCoroutine(endGameCoroutine())' ermöglicht innerhalb der 'IEnumerator endGameCoroutine' Funktion ein aktives Warten beim Ausführen von 'yield return new WaitForSeconds(2f);'. Die Übergabe des Parameters '2f' sorgt für eine Wartezeit von exakt 2 Sekunden. Der Grund für das aktive Warten ist die clientseitige Verarbeitung der Netzwerk-Nachricht, welche in der Funktion 'notifyHidersWinEvent' geschickt wird. Alle Clients bekommen einen visuellen Indikator im User-Interface angezeigt, der die Information beinhaltet, welcher Spielergruppe die aktuelle Runde gewonnen hat. Ohne das aktive Warten auf dem Server würden alle Spieler unmittelbar nach Spielende sofort in die Menü-Szene zurückgeworfen werden.

Listing 4.24: InGameProgressManager.cs Win Event

```
1
2 public void notifyHidersWinEvent()
3 {
4   winMsg = new Message()
5   {
6     message = 'Hiders won the Game',
7     messageType = MessageType.hidersWinNotification
8   };
9
10  foreach (var conn in NetworkServer.connections)
11  {
12    conn.Value.Send(winMsg);
13  }
14  StartCoroutine(endGameCoroutine());
15 }
16
17 IEnumerator endGameCoroutine()
18 {
19  resetInGameProgressManager();
20  gameTimer.SetPaused(true);
21  disablePlayerMovementGlobal();
22
23  yield return new WaitForSeconds(2f);
24  GameNetworkManager.singleton.ServerChangeScene(GameNetworkManager.
        singleton.offlineScene);
25 }
```

4.12 Nicht implementierte Konzepte

Runtime Spawn Manager:

Das Spielkonzept hat nicht vorausgesetzt, dass nach Initialisierung der Spiel-Szene weitere Objekte gespawnt werden müssen. Jegliche Objekte, die für eine Spiel-Session benötigt werden, spawnt der 'Prepare-Game-Manager' bereits zum Szenenwechsel zwischen Menü-Szene und Spiel-Szene. Aus diesem Grund wurde für den Prototypen auf den Runtime Spawn Manager verzichtet.

Interest Manager:

Mirror bietet eine Reihe an Komponenten an [Mir22b], welche eine eigene Implementierung eines Interest Managers überflüssig machen. Für den Prototypen wurde die Komponente 'Team Interest Management' benutzt [Mir22c]. Diese sorgt dafür, dass (Spieler)-Objekte nur für die Mitglieder einer Gruppe sichtbar sind.

Im Falle des Prototyps sollen Spieler mit der Rolle 'Ghost' lediglich für Spieler mit der Rolle 'Hider' und 'Ghost' sichtbar sein. Spieler mit der Rolle 'Seeker' sollen Spieler mit der Rolle 'Ghost' nicht sehen können.

Sollte ein Spielkonzept jedoch Voraussetzungen besitzen, über die Möglichkeiten der vordefinierten Interest Manager Komponenten von Mirror hinausgeht, so bietet Mirror ebenfalls die Möglichkeit eigene Interest Manager Implementierungen zu erstellen, indem Mirror eine abstrakte Klasse 'InterestManagement' bereitstellt. Falls eine eigene Interest Manager Klasse von dieser erbt, so kann sie die Methoden 'OnCheckObserver' und 'OnRebuildObservers' überschreiben. [Mir22a]

'OnCheckObserver' wird aufgerufen, wenn ein neuer Spieler spawnt. Gibt 'true' zurück, wenn ein (Spieler)-Objekt von dem neu gespawnten Spieler gesehen werden kann.

'OnRebuildObservers' wird innerhalb einer Update-Methode (diese wurde bereits in der Sektion 'Implementierung: Client UI & Visual Controller' erklärt) aufgerufen. Diese sorgt dafür, dass die Sichtbarkeit aller gespawnten Objekte neu berechnet wird.

5 Studienkonzept & Zusammenfassung

In diesem Kapitel wird zunächst ein Studienkonzept erläutert, welches weitere Forschung ermöglicht. Es folgt ein Fazit und Ausblick.

5.1 Studienkonzept

Um zu beweisen, dass die vorgestellten Konzepte einen tatsächlichen Mehrwert für die Entwicklung von Multiplayer-Spielen bietet, bedarf es einer Studie, welche im folgenden Studienkonzept beschrieben wird.

Quantitative Umfrage:

Diese beinhaltet die Befragung von mehreren Hundert Personen aus unterschiedlichen Ländern und Genres. Sie soll mithilfe der 'Likert-Skala' [Wik22a] einige persönliche Einstellungen der befragten Personen messen. Diese haben starken Bezug zu den Konzepten, welche in dieser Arbeit beschrieben wurden. Um die Einstellungen der befragten Personen zu den einzelnen Entwicklungs-Problemen (Items) zu erfassen, werden stets die gleichen Antwortmöglichkeiten (Grade) angeboten:

trifft zu (1), trifft eher zu (2), teils-teils (3), trifft eher nicht zu (4), trifft nicht zu (5)

Die Umfrage sollte folgende Items beinhalten:

- Die Entscheidung eine passende Technologie für mein erstes Multiplayer-Spiele auszuwählen war mühsam.

- Die Entwicklung des Matchmaking Systems bereitete mir Probleme.

- Die Umsetzung des Server-Runners bereitete mir Probleme.

- Die Implementierung der Benutzeroberfläche bereitete mir Probleme.

- Die Realisierung der Verarbeitung von Netzwerkereignissen bereitete mir Probleme.

- Die Umsetzung der Szenen-Verwaltung bereitete mir Probleme.

- Die Entwicklung des clientseitigen Verbindungsaufbaus zu einem Spiel-Server bereitete mir Probleme.

- Die Entwicklung der serverseitigen Verwaltung aller Spieler bereitete mir Probleme.

- Die Umsetzung der Vorbereitung von Spiel-Szenen bereitete mir Probleme.

- Die Realisierung der Verwaltung von Spielfortschritt bereitete mir Probleme.

- Die Implementierung des Spawnings von Spielobjekten bereitete mir Probleme.

- Die Entwicklung der Sichtbarkeits-Verwaltung von Objekten bereitete mir Probleme.

Die Umfrage soll per E-Mail an ausgewählte Personen an Hochschulen und Unternehmen verschickt werden. Außerdem kann die Umfrage in einschlägige Entwickler-Foren gepostet werden. Beispiele hierfür sind: www.forum.unity.com, www.forums.unrealengine.com und www.gamedev.net/forums/forum/8-networking-and-multiplayer.. Wichtig ist, dass lediglich Personen an der Umfrage teilnehmen, welche aktuell ihr erstes Multiplayer-Spiel entwickeln oder bereits entwickelt haben. Die Teilnehmer sollten darauf hingewiesen werden, dass falls sie noch keine Berührungspunkte mit einzelnen Punkten haben, keine Antwort auf die entsprechende Fragestellung geben sollen.

Die Umfrage kann anonym, oder mit Personenbezug für eine bessere Auswertung der Ergebnisse erfolgen. Ebenso wäre es denkbar, getrennte Umfragen für eine Gruppe an ausgewählten Einzelpersonen sowie die öffentliche Entwickler-Community im Internet zu erstellen.

Feldversuch:

Mithilfe dieses Feldversuchs soll gezeigt werden, dass unerfahrene Entwickler einen klaren Vorteil haben, wenn sie das technische Design eines Neuprojekts nach diesen Konzepten ausrichten.

2 Gruppen von unerfahrenen Entwicklern wird die Aufgabe gegeben, ein Multiplayer-Spiel zu entwickeln. Eine Gruppe bekommt die Konzepte aus dem Kapitel 'Konzepte' als Hilfestellung, die andere nicht.

Die generellen Anforderungen an die Feldstudie sind:

- Sollten die Probanden aus Studenten bestehen, so müssen sich alle im gleichen Fachsemester befinden.
- Die Probanden sollten gleich viel Erfahrung im Bereich der Multiplayer Spieleentwicklung haben.
- Alle Probanden müssen Zugang zu ähnlicher Hardware besitzen.
- Die Gruppen teilen die Aufgaben unter sich auf, und suchen sich selbstständig Frame-

works und Engines für einen Prototyp aus.
- Im Anschluss müssen beide Gruppen die oben erarbeitete Umfrage ausfüllen.
- Das Experiment ist zeitlich begrenzt. Ein Ergebnis muss innerhalb von 7 Tagen eingereicht werden.

Die Anforderungen an das Spiel sind:

- Das Spiel soll ein simpler Multiplayer-Ableger des Spiels 'Snake' werden. Es müssen lediglich die Grundregeln des Spiels umgesetzt werden. Diese sind in der folgenden Quelle genauer beschrieben: [.2222] beschrieben.
- Es soll für die Spieler möglich sein, sich über eine Lobby zu treffen, und von dort aus ins Spiel zu starten.
- Die Grundarchitektur soll dem Client - Server Modell entsprechen.

Während des Experiments sollen folgende Variablen dokumentiert werden:

- Wurden alle Anforderungen an das Spiel erfüllt?
- In welcher Zeit hat welches Team den Prototypen entwickelt?
- Wie sind beide Teams methodisch vorgegangen?

Je mehr Feldstudien dieser Art durchgeführt werden, desto genauer kann später eine finale Aussage getroffen werden.

Auswertung der quantitativen Umfrage:

Die Ergebnisse der Umfrage an ausgewählte Personen und an die Entwickler Community werden ausgewertet. Aus den Ergebnissen sollten Antworten auf folgende Fragestellungen abgeleitet werden können:

Ist ein Trend zu erkennen, dass angehende Spieleentwickler Probleme mit den aufgeführten Implementierungsdetails hatten?

Bei welchen Implementierungsdetails hatten Entwickler am meisten Probleme?

Auswertung der Feldstudie:

Hat die Gruppe, welche sich an die Konzepte gehalten hat, einen messbaren Vorteil gehabt? Konkret sollten die oben beschriebenen Variablen ausgewertet werden:

- Ist die Gruppe, welche den Prototypen mit den Konzepten umgesetzt hat, schneller zum Ziel gekommen, als die Gruppe ohne Hilfestellung?

- Wurden bei beiden Gruppen alle Anforderungen umsetzt?

- Hatte das Team, welches den Prototypen mithilfe der Konzepte umgesetzt hat weniger Probleme die Arbeitslast innerhalb des Teams aufzuteilen?

Anschließend wird die Umfrage betrachtet, welche die Teilnehmer ausgefüllt haben. Aus dieser kann abgeleitet werden, ob die Entwickler der Gruppe, welche eine Hilfestellung erhalten haben, einen spürbaren Vorteil bei der Entwicklung des Prototyps hatten.

Zum Schluss können nun die quantitative Umfrage sowie die Umfrage, welche aus dem Ergebnis der Feldstudie resultiert, miteinander verglichen werden. Sollten die Personen, welche die Feldstudie innerhalb der Gruppe ohne Hilfestellung durchgeführt haben, eine ähnliche Einstellung zu den Punkten der Umfrage haben, wie die Personen, welche an der quantitativen Umfrage teilgenommen haben, so ist davon auszugehen, dass beide Personengruppen eine ähnliche Erfahrung bei der Entwicklung von Multiplayer-Spielen mitbringen, und diese Personengruppen somit als vergleichbar gelten.

Sollte die Gruppe, welche die Konzepte als Hilfestellung genutzt hat, die Umfrage mit einem anderen Ergebnis abschließen, so kann dies ein Hinweis sein, dass die Konzepte, welche in dieser Arbeit beschrieben sind, eine Einstiegshilfe bei der Entwicklung von Multiplayer-Spielen bietet.

Um dies final zu beweisen, müsste die Feldstudie mehrmals durchgeführt werden, und die Anforderungen an das zu entwickelnde Spiel müssen bei jeder neuen Feldstudie geändert werden. Mehr Variation würde dann geschaffen, wenn sich Genre, Matchmaking, Art des Hostings und Spielkonzept bei jeder neu durchgeführten Feldstudie ändert.

5.2 Fazit

Die in dieser Arbeit gestellten Forschungsfragen F1 und F2 konnten teilweise beantwortet werden.
F1:
Durch die Aufstellung der Konzepte, und anschließenden Umsetzung eines Prototyps kann nicht final beurteilt werden, ob die Probleme für eine Vielzahl an angehenden Entwicklern vereinfacht wurden. Allerdings war die Entwicklung des in dieser Arbeit beschriebenen Prototyps durch die Einhaltung der Konzepte deutlich vereinfacht. Dies könnte ein Hinweis darauf sein, dass Entwickler mit anderen Spielideen, welche ähnliche Voraussetzungen besitzen wie die des Prototyps, welcher im Rahmen dieser Arbeit entwickelt wurde, einen Vorteil aus den Konzepten ziehen könnten.

Für Spielideen, welche aus einem anderen Genre kommen, eine andere Form des Matchmakings oder Art des Hostings unterstützen, kann jedoch ohne Ausführung einer weitergehenden Studie keine Aussage getroffen werden.

F2:
Ein Studienkonzept könnte so aussehen, wie es in der Sektion 'Studienkonzept' beschrieben ist. Bestimmte Details können je nach konkreter weitergehender Forschungsfrage abgeändert oder ergänzt werden. Grundsätzlich führt eine korrekte Durchführung des Studienkonzepts zur Beantwortung der Forschungsfrage:
'Bringen die in dieser Arbeit beschriebenen Konzepte einen Mehrwert bei der Entwicklung eines Multiplayer-Spiels für angehende Entwickler?'

5.3 Ausblick

Abschließend ist festzuhalten, dass diese Arbeit einen Ansatz bietet, welcher zur Erleichterung des Einstiegs in die Entwicklung von Multiplayer-Spielen führen könnte. Um zu beweisen, dass die Konzepte praxistauglich und für eine Vielzahl an unterschiedlichen Projekten einsetzbar sind, muss jedoch eine weitergehende Studie durchgeführt werden, welche sich an dem in dieser Arbeit beschriebenen Studienkonzept orientiert. Aus den Ergebnissen der Studie können weitere Schlussfolgerungen gezogen werden, die eine abschließende Bewertung dieser Arbeit ermöglichen.

Literaturverzeichnis

[.0322] Mirror Networking – Open Source Networking for Unity (03.02.2022), URL https://mirror-networking.com/

[.0522] Remote Actions (05.02.2022), URL https://mirror-networking.gitbook.io/docs/guides/communications/remote-actions

[.2014] *Matchmaking in multi-player on-line games: studying user traces to improve the user experience* (2014)

[.2222] Das Onlinespiel Snake - Spielbeschreibung (22.02.2022)

[Bil22] Billwagner: virtual – C#-Referenz (08.02.2022), URL https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/virtual

[Den18] Deng, Yunhua; Li, Yusen; Seet, Ronald; Tang, Xueyan und Cai, Wentong: The Server Allocation Problem for Session-Based Multiplayer Cloud Gaming. *IEEE Transactions on Multimedia* (2018), Bd. 20(5): S. 1233–1245

[Fel22] Felix Bruebach GitHub: felithium - Overview (20.02.2022), URL https://github.com/felithium

[gam20] game - Verband der deutschen Games-Branche e.V.: Jahresreport der deutschen Games-Branche 2020 (2020), URL https://www.game.de/wp-content/uploads/2020/08/game-Jahresreport-2020.pdf

[Git21] GitHyp: Most played games on Steam in 2020, by hourly average number of players (February 2021), URL https://www.statista.com/statistics/656319/steam-most-played-games-average-player-per-hour/

[Git22] GitHub: proyecto26/RestClient: A Promise based REST and HTTP client for Unity (10.02.2022), URL https://github.com/proyecto26/RestClient

[Gli08] Glinka, Frank; Ploss, Alexander; Gorlatch, Sergei und Müller-Iden, Jens: High-Level Development of Multiserver Online Games. *International Journal of Computer Games Technology* (2008), Bd. 2008: S. 1–16, URL https://www.hindawi.com/journals/ijcgt/2008/327387/

[Jer17] Jerome Renaux: Interest management for multiplayer online games (2017), URL https://www.dynetisgames.com/2017/04/05/interest-management-mog/

[M. 09] M. Gatrell; S. Counsell und T. Hall: Design Patterns and Change Proneness: A Replication Using Proprietary C# Software, in: *2009 16th Working Conference on Reverse Engineering*, S. 160–164

[MFa21] MFatihMAR: Game Networking Resources (2021)

[Mic06] Michelson, Brenda: *Event-Driven Architecture Overview*, Patricia Seybold Group, Boston, MA (2006)

[Mir22a] Custom Interest Management (17.02.2022), URL https://mirror-networking.gitbook.io/docs/interest-management/custom

[Mir22b] Mirror Interest Management (17.02.2022), URL https://mirror-networking.gitbook.io/docs/interest-management

[Mir22c] Team Interest Management (17.02.2022), URL https://mirror-networking.gitbook.io/docs/interest-management/team

[Pay19] Payne, John: Making Multiplayer Games Doesn't Have to Be Difficult. *Crayta* (18.09.2019), URL https://medium.com/crayta/making-multiplayer-games-doesnt-have-to-be-difficult-d08d19c83de3

[Sim20] Simon-Kucher & Partners: Impact of COVID-19 on the frequency of playing multiplayer video games worldwide as of June 2020 (2020), URL https://www.statista.com/statistics/1188549/covid-gaming-multiplayer/

[Sme02a] Smed, Jouni; Kaukoranta, Timo und Hakonen, Harri: Aspects of networking in multiplayer computer games. *The Electronic Library* (2002), Bd. 20(2): S. 87–97, URL https://www.emerald.com/insight/content/doi/10.1108/02640470210424392/full/pdf

[Sme02b] Smed, Jouni; Kaukoranta, Timo und Hakonen, Harri: *A review on networking and multiplayer computer games*, Citeseer (2002), URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.2565&rep=rep1&type=pdf

[Tec22] Technologies, Unity: Unity (03.02.2022), URL https://unity.com/de

[Tho15] Thones, Johannes: Microservices. *IEEE Software* (2015), Bd. 32(1): S. 116

[Wik12] Szene (Computergrafik) (2012), URL https://de.wikipedia.org/w/index.php?title=Szene_(Computergrafik)&oldid=99502558

[Wik19a] Bandbreite (2019), URL https://de.wikipedia.org/w/index.php?title=Bandbreite&oldid=189320030

[Wik19b] Serialisierung (2019), URL https://de.wikipedia.org/w/index.php?title=Serialisierung&oldid=190015284

[Wik20] Spawnen (2020), URL https://de.wikipedia.org/w/index.php?title=Spawnen&oldid=200710033

[Wik21a] Bildfrequenz (2021), URL https://de.wikipedia.org/w/index.php?title=Bildfrequenz&oldid=218417275

[Wik21b] Echtzeitsystem (2021), URL https://de.wikipedia.org/w/index.php?title=Echtzeitsystem&oldid=217468472

[Wik21c] Ego-Shooter (2021), URL https://de.wikipedia.org/w/index.php?title=Ego-Shooter&oldid=214880313

[Wik21d] Hack (2021), URL https://de.wikipedia.org/w/index.php?title=Hack&oldid=211775453

[Wik21e] IP-Adresse (2021), URL https://de.wikipedia.org/w/index.php?title=IP-Adresse&oldid=218224804

[Wik21f] Massively Multiplayer Online Game (2021), URL https://de.wikipedia.org/w/index.php?title=Massively_Multiplayer_Online_Game&oldid=213895562

[Wik21g] Matchmaking (video games) (2021), URL https://en.wikipedia.org/w/index.php?title=Matchmaking_(video_games)&oldid=1056891415

[Wik21h] Nicht-Spieler-Charakter (2021), URL https://de.wikipedia.org/w/index.php?title=Nicht-Spieler-Charakter&oldid=217227874

[Wik21i] Port (Protokoll) (2021), URL https://de.wikipedia.org/w/index.php?title=Port_(Protokoll)&oldid=216781784

[Wik22a] Likert-Skala (2022), URL https://de.wikipedia.org/w/index.php?title=Likert-Skala&oldid=219268097

[Wik22b] Local Area Network (2022), URL https://de.wikipedia.org/w/index.php?title=Local_Area_Network&oldid=218925000

[Wik22c] Unity (Spiel-Engine) (2022), URL https://de.wikipedia.org/w/index.php?title=Unity_(Spiel-Engine)&oldid=219813859

[Wik22d] Verzögerungszeit (2022), URL https://de.wikipedia.org/w/index.php?title=Verzögerungszeit&oldid=218922550

Abbildungsverzeichnis

Listings

.