HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Christian Tschub

Technische Konzeption und Implementierung eines adaptiven Missionssystems für den Spielprototyp Urban Legend

2013

Fakultät **Mathematik/Naturwissenschaften/Informatik**

BACHELORARBEIT

Technische Konzeption und Implementierung eines adaptiven Missionssystems für den Spielprototyp Urban Legend

Autor:
Christian Tschub

Studiengang:
Informatik

Seminargruppe:
IF10w1-B

Erstprüfer:
Prof. Dr.-Ing. Mario Geißler

Zweitprüfer:
M.Sc. Enrico Pisko

Mittweida, September 2013

Bibliografische Angaben

Tschub, Christian: Technische Konzeption und Implementierung eines adaptiven Missionssystems für den Spielprototyp Urban Legend, 63 Seiten, 8 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Mathematik/Naturwissenschaften/Informatik

Bachelorarbeit, 2013

Referat

Die vorliegende Arbeit beschäftigt sich mit der Konzeption und Implementierung eines adaptiven Missionssystems für den Spielprototyp Urban Legend. Das Ziel ist es, eine Umgebung zu schaffen, in der Missionen erstellt werden können und diese an Spieler eines bestimmten Typs verteilt werden. Dabei sollen nicht nur einzelne Missionen erstellt werden können, sondern auch Missionsstränge, die eine Handlung zwischen mehreren Spielern darstellen.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

1 Einleitung

In interaktiven Medien, wie z.B. Videospielen, ist das Spielerlebnis oft unabhängig davon, welcher Nutzer gerade dieses Medium konsumiert[1]. Der Spieler muss die Elemente im Spiel durchführen, welche ihm vom Designer vorgegeben wurden. Dabei wird nicht berücksichtigt, ob es dem Spieler gerade gefällt, welche Aufgabe er als nächstes zu erfüllen hat. Aufgrund dieser Tatsache muss das Videospiel schon bei der Entwicklung auf eine bestimmte Zielgruppe zugeschnitten werden[2]. Die Einschränkung der Zielgruppe wirkt sich natürlich auch auf die Wirtschaftlichkeit des Videospiels aus. Dieses Problem könnte dadurch entschärft werden, indem das Konzept des Spiels adaptiv umgesetzt wird. Das bedeutet, dass sich das Spiel an den Spieler anpasst und ihm Inhalte liefert, welche dem Spieler Vergnügen bereiten. Das Fehlen von passenden Game-Engines, also auch Tools, um diese Adaptivität zu realisieren, stellt laut Hudlicka ein Problem dar[3].

Die vorliegende Arbeit beschäftigt sich deshalb damit, einen Prototyp eines adaptiven Systems für Missionen zu konzipieren und zu implementieren. Ebenfalls wird eine Skriptsprache entwickelt, mit welcher es möglich ist verschiedene Arten von Missionen zu erstellen. Dieses System soll am Beispiel von **Urban Legend**, einem von der Forschungsgruppe Gamecast entwickelten Videospiels, demonstriert werden.

Die Zielgruppe dieser Arbeit sind Personen mit Programmierkenntnissen, die noch keine Erfahrungen mit der Skriptsprache UnrealScript haben.

1.1 Motivation

Die Motivation besteht darin, ein System zu entwickeln, welches sich an den Spieler anpasst und nicht der Spieler sich an das Spiel. Besonders die Implementierung dieses Systems für ein Multiplayer-Spiel stellt sich als Herausforderung dar. Denn im Gegensatz zum Multiplayer, ist die Konzeption von Missionen für den Singleplayer von geringerem Aufwand, da weniger interaktive Möglichkeiten geschaffen werden müssen. Dies ist der Fall, weil im Singleplayer vorprogrammierte NPCs meißtens das gewünschte Verhalten besitzen, während im Multiplayer die Spieler nicht kontrolliert werden können, dass sie wirklich das machen, was ihnen die Mission vorgibt. Deshalb ist hier die Motivation das Missionssystem nicht nur zum Funktionieren zu bekommen, sondern dieses auch so robust wie möglich zu implementieren.

[1] vgl. Hudlicka 2009
[2] vgl. Spiele entwickeln 2009, 122
[3] vgl. Hudlicka 2009, 300

1.2 Zielsetzung

Um das adaptive Missionssystem zu entwickeln, müssen folgende Schwerpunkte bearbeitet werden:

- Entwicklung einer eigenen Skriptsprache, damit Autoren ohne Programmcodemodifikation Missionen erstellen können.
- Erfassung und Auswertung von Spieler-Aktionen, um diesen einen bestimmten Spielertyp zuzuordnen und nach diesem Typ die Missionen zu verteilen.
- Entwicklung eines Missions-Managers, welcher automatisch laufende Missionen verwaltet, passende Missionen an Spieler verteilt und Missionen zwischen den Spielern synchronisiert, um eine sinnvolle Handlung aufzubauen.
- Implementierung einer geeigneten Oberfläche für den Spieler, um seine laufenden Missionen zu betrachten und den Status der Missionen zu erkennen.

1.3 Abgrenzungen

Diese Arbeit zeigt lediglich einen prototypischen Ansatz, wie man ein adaptives Missionssystem implementieren könnte. So werden nur bestimmte Informationen über den Spieler gesammelt, um zu zeigen, wie diese Informationen für die automatische Zuweisung von Missionen genutzt werden können. Es findet keine komplexe Analyse des Spielers statt, so werden z.B. keine Emotionen berücksichtigt.

Das System wird keine sinnvolle Handlung automatisch aus einzelnen Missionen bilden, d.h. der Autor muss sich vorher selbst überlegen, wie er die Missionen sinnvoll zu einem Missionsstrang zusammenführt.

2 Vorbetrachtung

2.1 Gamecast

Gamecast ist eine Forschungsgruppe an der Hochschule Mittweida, welche sich zum Ziel gesetzt hat, die Produktion von Animationsfilmen zu vereinfachen und auf diese Weise die Effizienz der Produktion zu steigern. Dies soll durch die Verbindung von Videospiel und Animationsfilm realisiert werden. Die Spielpartie soll aufgezeichnet werden und anschließend daraus ein Animationsfilm entstehen.
Ein weiteres Ziel ist es, die Emotionen des Spielers während des Interagierens in der Videospielwelt mit in das Spielerlebnis zu integrieren. So sollen bestimmte Events ausgelöst werden, wenn der Spieler eine bestimmte Mimik aufzeigt. Das Spiel soll sich also an die Stimmung des Spielers anpassen und so das Spielerlebnis individuell verstärken.

Studenten aus verschiedenen Studienrichtungen wie Medien, Informatik, Multimediatechnik, 3D-Gamedesign lassen ihre Kompetenzen in die Bearbeitung der Aufgaben einfließen. Somit ist gewährleistet, dass die Ziele unter verschiedenen Ansichten betrachtet werden und somit unter Umständen verschiedene Lösungen für Probleme gefunden werden können.

Die Technologien, welche für die Umsetzung der Ziele benötigt werden, wurden über mehrere Jahre in unterschiedlichen Forschungsprojekten realisiert. Im Projekt **GAMECAST**, welches im Zeitraum vom 01.03.2009 bis zum 31.12.2010 stattfand, wurden verschiedene Schnittstellen entwickelt, welche die Produktion von Animationssequenzen mit Hilfe einer Game-Engine ermöglichen. Damit wurde der erste Punkt der Zielsetzung umgesetzt. Ebenfalls im Projekt wurde eine Mimikerkennung implementiert, welche auf der SHORE-Bibliothek des Frauenhofer Institus für integrierte Schaltungen in Erlangen basiert. Damit wurde es möglich, mit einer handelsüblichen Webcam Emotionen des Spielers aufzuzeichnen.

In der Zeitspanne vom 01.02.2011 bis zum 31.12.2011 wurden im Forschungsprojekt **ADAM** Schnittstellen umgesetzt, mit denen die erfassten Emotionsdaten verarbeitet werden konnten. Dadurch wurde es möglich die Spielwelt an die Emotionen des Spielers anzupassen und so ein intensiveres Spielerlebnis zu gewährleisten.

Im Projekt **VRAPS** (Virtual Research and Production Studio) wurde ein Modul entwickelt, mit dessen Hilfe die Bewegungen einer Person erkannt und auf eine 3D-Figur übertragen werden konnten. Mit diesem Modul sollte das Animieren von Figuren beschleunigt und vereinfacht werden. Das Projekt wurde im Zeitraum vom 01.01.2012 bis zum 32.12.2012 bearbeitet.

Die in den Forschungsprojekten **GAMECAST** und **ADAM** entwickelten Technologien, sollte anhand des Prototyp-Videospiel **Urban Legend** demonstriert werden. **Urban Legend** wird seit dem Jahr 2011 entwickelt.

2.2 Urban Legend

Urban Legend ist ein, mit dem UDK (Unreal Developer's Kit) entwickeltes, Videospiel. Es dient als Prototyp für die Demonstration von verschiedenen entwickelten Technologien und Schnittstellen. Dabei ist es nicht nur eine Tech-Demo, sondern ein Spiel mit einem vollständigem Gameplay. Da sich das adaptive Missionssystem auf dieses Spiel bezieht, wird das Gameplay im Folgenden näher erläutert.

2.2.1 Story

Bei Urban Legend handelt es sich um ein Online-Multiplayer-Spiel. Die Story beruht auf dem Konflikt zwischen Staat und Rebellen. Der Staat versucht alles daran zu setzen, die Bevölkerung zu kontrollieren und so eine Gesellschaft nach ihren Vorstellungen zu formen. Die Rebellen versuchen sich dieser Unterdrückung zu widersetzen, jedoch ohne dem Wirken von Gewalt, sondern mit eher subtilen Mitteln, wie dem Lahmlegen von Kommunikationsnetzen. Die Staatsgewalt wird durch 2 Fraktionen im Spiel repräsentiert, den Polizisten und den Technikern. Die Rebellen sind ihre eigene Fraktion. Bestimmte Fähigkeiten können von den Polizisten, Technikern und Rebellen eingesetzt werden, um ihre eigenen Ziele durchzusetzen.

2.2.2 Spielablauf

Sobald das Spiel beginnt, hat der Spieler die Auswahl, welcher Fraktion er sich anschließen möchte. Nach der Auswahl der Fraktion wählt der Spieler das Aussehen seiner Spielfigur und die gewünschten Fähigkeiten (Abbildung 2.1), welche er während der Partie verwenden möchte. Mehrere Fähigkeiten stehen dabei zur Auswahl, jedoch können nur maximal 3 ausgewählt werden.

Während der Spielrunde müssen nun die Spieler bestimmte Aktionen durchführen, um Punkte zu sammeln und so ihrer Fraktion am Ende der Spielrunde zum Sieg zu verhelfen. So haben Polizisten die Möglichkeit, Rebellen mit einem Emotionsscanner zu enttarnen und anschließend zu betäuben. Rebellen besitzen die Fähigkeit Hotspots im Spiel zu hacken und dafür Punkte zu bekommen. Der Techniker kann wiederum diese gehackten Hotspots reparieren. Am Ende der Runde werden alle Punkte der Spieler berechnet und daraus der Sieger ermittelt, die Punkte der Polizisten und Techniker werden dabei summiert. Es ist möglich, dass Spieler negative Punkte bekommen, so wird z.B. der Polizist dafür bestraft, dass er wahllos NPCs betäubt.

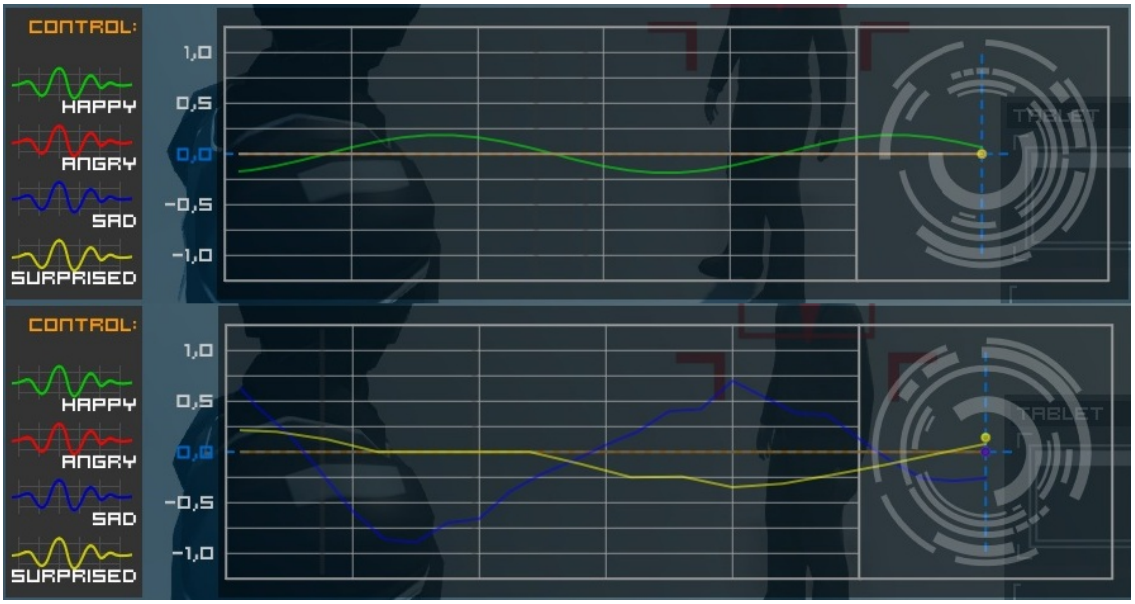
Abbildung 2.1: Auswahlmenü der Spielfigur

2.2.3 Eingesetzte Technologien

Die in Kapitel 1.2 beschriebenen Technologien werden auf folgende Weise durch Urban Legend demonstriert:

- Die Erkennung der Emotionen des Spielers wird für eine Fähigkeit der Polizisten verwendet. Die Polizisten besitzen einen Emotionsscanner, mit dessen Hilfe sie erkennen können, ob das gescannte Gegenüber gerade traurig, fröhlich, sauer oder überrascht ist. Am Anfang des Scanvorgangs hat jede Figur im Spiel, unabhängig ob Spieler oder NPC, den selben Verlauf der Emotionskurven. Nach einigen Sekunden würden die Kurven jedoch ausschlagen, falls die gescannte Person ein Spieler ist und gerade nicht lächelt. Anhand der Abbildung 2.2 kann man dies erkennen. Beim Lächeln würde die Kurve grün und sinusförmig dargestellt werden, während sie bei anderen Emotionen die Farbe ändert und ausschlägt. Der gerade gescannte Spieler muss also in die Webcam lächeln, damit er nicht als Rebell enttarnt wird und somit nicht betäubt wird.
- Die Verbindung zwischen Videospiel und Animationsfilm wird dadurch aufgebaut, dass sämtliche Aktionen und Bewegungen des Spielers im Level aufgezeichnet werden. Alle diese Daten werden in ein Log-File geschrieben und können anschließend verwertet werden. So lässt sich die gesamte Szene durch die Game-Engine wieder abspielen. Es ist zudem möglich die Szene zu modifizieren, so lassen sich andere Animationen abspielen oder bestimmte Kameraeinstellungen manipulieren.

Abbildung 2.2: Vergleich von Emotionskurven



2.3 UDK

Das UDK ist eine kostenlose Entwicklungsumgebung für Videospiele, welche von EPIC Games bereitgestellt wird. Es basiert auf der Unreal Engine 3 und bietet alle Technologien und Tools, welche auch von der kostenpflichtigen Unreal Engine 3 angeboten werden [4]. Der einzige Unterschied zwischen dem UDK und der Unreal Engine 3 beruht darauf, dass der Entwickler keinen Zugang auf den C++ Quellcode hat. Für die Entwicklung im UDK wird UnrealScript verwendet. Die Verwendung des UDK ist komplett kostenfrei, jedoch müssen Gebühren entrichtet werden, sobald das entwickelte Produkt verkauft werden soll. Zum einen muss eine einmalige Lizenzgebühr von $99 gezahlt werden, sowie bei Einnahmen von mehr als $50,000 müssen 25% der weiteren Einnahmen an EPIC entrichtet werden.

2.3.1 UnrealScript

UnrealScript ist eine Objektorientierte Programmiersprache und kommt einer Mischung aus C++ und Java gleich. Der Syntax ähnelt sehr C++, während Zeiger und Speicher-Reservierungen und Freigaben, dank eines Garbage Collectors, komplett wegfallen. Besonders viel Gebrauch wird von der Vererbung von anderen Klassen gemacht. So erben z.B. jegliche Gegenstände, welche im Spiel platziert werden können, von der Klasse `Actor`.

[4] http://www.unrealengine.com/udk/licensing/licensing-faqs/

Klassen

Klassen in UnrealScript können beim Deklarieren, von einer anderen Klasse sämtliche Funktionen und Variablen erben. Mit Hilfe der Anweisung `extends` wird dies umgesetzt, z.B. `class PlayerController extends Controller`. Außerdem kann der Klasse eine Konfigurationsdatei zugewiesen werden. Die Variablen in dieser Datei können dann von der Klasse eingelesen und verwendet werden. Auf die Konfigurationsdateien wird später in der Arbeit näher eingegangen.

Variablen

Sollen Globale Variablen vereinbart werden, dann müssen diese immer mit `var` am Anfang der Zeile beginnen. Anschließend folgt der Datentyp und die Bezeichnung der Variable, z.B. `var int a`. Lokale Variablen müssen mit dem Schlüsselwort `local` angegeben werden, z.B. `local int a`. Variablen können nicht in der selben Anweisung deklariert und initialisiert werden. Globale Variablen müssen in den sogenannten `defaultproperties` initialisiert werden. Die `defaultproperties` befinden sich immer am Ende einer Klasse.

`byte` - einzelner Byte-Wert, von 0 bis 255
`bool` - Wahrheitswert, true/false
`int` - 32 bit Ganzzahl
`float` - 32 bit Gleitkommazahl
`string` - Zeichenkette
`class` - Klassenreferenz
`vector` - Vector, besteht aus X, Y und Z, vom Datentyp `float`
`rotator` - Rotator, besteht aus Pitch, Yaw und Roll, jeweils vom Datentyp `int`

Arrays

Es existieren sowohl statische, als auch dynamische Arrays.
Statische Arrays: `var|local array` Datentyp arrayName[arraySize]
Dynamische Arrays: `var|local array`<Datenentyp> arrayName

Wichtige Array-Funktionen:

`addItem(Element)`	- Fügt dem dynamischen Array ein Element hinzu.
`remove(index, count)`	- Gibt an ab welchem `index` im Array gelöscht werden soll und wie oft der Löschvorgang stattfinden soll, z.B. myArr.Remove(1,3) würde bedeuten, dass die Elemente vom Index 1 bis einschließlich Index 3 gelöscht werden.

	Die Indices der anderen übrigen Elemente würden verändert werden, weil das Array verkleinert wurde.
`find(Element)`	- Durchsucht das Array nach einem Element und gibt ggf. den Index zurück, an dem das Element gefunden wurde, sonst -1.

Structs

Structs lassen sich mit der selben Syntax wie in C++ realisieren, möchte man jedoch Standard-Werte für die Variablen im Struct festlegen, dann muss dies in den `structdefaultproperties` getan werden.

Funktionen

[Funktionsmodifikator] function [Rückgabewert]Funktionsname([optional] Parameter)

Es existieren verschiedene Funktionmodifikatoren, welche angeben, wie die Funktion ausgeführt werden soll:

`static`	- Die Funktion kann aufgerufen werden, ohne das sie eine Referenz zu einem Objekt der Klasse besitzt.
`client`	- Die Funktion soll auf dem Client ausgeführt werden und nicht auf dem Server.
`server`	- Die Funktion soll auf dem Server ausgeführt werden.
`exec`	- Die Funktion soll durch einen Tastaturbefehl ausführbar sein.
`simulated`	- Die Funktion soll sowohl auf dem Server, als auch auf dem Client laufen.
`reliable`	- Muss bei der Verwendung von `client` bzw. `server` angegeben werden. Dadurch wird garantiert, dass die Funktion aufgerufen wird.
`unrealiable`	- Wie reliable, bloß dass nicht garantiert wird, dass die Funktion aufgerufen wird.

Falls eine geerbte Funktion überschrieben wurde, kann die Funktion der übergeordneten Klasse, mit Hilfe von `super.Funktionsname`, aufgerufen werden.

Es existiert eine besondere Art von Funktionen, die Events. Dabei wird bei der Benennung einer solchen Funktion, anstelle von `function`, `event` geschrieben. Events werden direkt von der Engine aus aufgerufen und müssen nicht direkt ausgeführt werden. Aus diesem Grund existiert auch keine main()-Funktion im UnrealScript, eigene Funktionen könnten z.B. über Events ausgeführt werden.

Objekterzeugung

In Unreal gibt es 2 verschiedene Arten um Objekte zu erzeugen. Gehört das Objekt einer Klasse an, welche von `Object` erbt, dann muss das Objekt mit `Variablenname = new class'Klassenname'`, erzeugt werden. Erbt die Klasse jedoch von `Actor`, dann muss mit `Variablenname = Spawn(class'Klassenname')` das Objekt erzeugt werden. Ein Objekt kann sich nur im Level bewegen und mit anderen Actors, als auch der Umgebung interagieren, wenn es von `Actor` erbt. Deswegen ist es notwendig, dass solche Objekte mit `Spawn()` erstellt werden.

PlayerController

Der PlayerController ist eine Klasse in UnrealScript. Dieser enthält sämtliche Informationen über den Spieler, wie z.B. die gerade kontrollierte Spielfigur. Der PlayerController dient ebenfalls als Bindeglied zwischen Client und Server. So kann der Server durch die Referenz auf den PlayerController, Funktionen auf dem Client des Spielers ausführen. Ebenfalls kann der PlayerController Funktionen auf dem Server aufrufen und so Informationen von diesem holen.

Pawn

Der Pawn kann als die Hülle des Spielers betrachtet werden. So besitzt jeder Pawn eine Referenz auf seinen PlayerController und vice versa. Wichtige Funktionen werden durch den Pawn bereitgestellt, z.B. ob dieser getroffen wurde oder andere auf die Spielfigur bezogenen Zustände.

2.3.2 Konfigurationsdateien

Konfigurationsdateien können direkt von der Unreal Engine eingelesen werden und alle beinhaltenden Variablen sofort verarbeiten. Die Werte der Variablen in der Datei können ebenfalls während des Spiels modifiziert und abgespeichert werden. Das bietet den Vorteil, wichtige Informationen nach einer Spielpartie wieder parat zu haben und so z.B. Spielstände zu sichern oder Informationen über das Spielverhalten zu sammeln.

Anbindung an UnrealScript

Sämtliche Konfigurationsdateien, welche von der Unreal Engine benutzt werden, befinden sich im Ordner \UDKGame\Config. Möchte man eine neue Konfigurationsdatei anlegen, dann sollte zuerst eine *Default*NameDerDatei angelegt werden. In diese sollten alle Standardwerte reingeschrieben werden. Beim Start des Spiels, überprüft die Engine alle Konfigurationsdateien und versucht eine *UDK*NameDerDatei an-

zulegen. Falls diese schon vorhanden ist, wird sie nicht nochmal angelegt. Aus diesem Grund muss, sobald etwas in der *Default*NameDerDatei modifiziert wurde, die *UDK*NameDerDatei gelöscht werden. Wird per UnrealScript etwas an der Datei verändert, dann geschieht dies immer in der *UDK*NameDerDatei, damit immer die Standardwerte in der *Default*NameDerDatei gesichert sind.

Damit eine Klasse in UnrealScript weiß, dass sie die Werte aus einer Kondigurationsdatei entnehmen muss, wird in der Konfigurationsdatei der Name dieser Klasse angegeben. In der Klasse selbst muss ebenfalls angegeben werden, welche Konfigurationsdatei verwendet werden soll. So wird z.B. im UnrealScript die Klasse `GC_Mission_Manager` mit folgender Anweisung erstellt: `class GC_Mission_Manager extends Actor config (NameDerDatei)`. In der Konfigurationsdatei muss ebenfalls die Klasse, in welcher die Variablen verwendet werden, angegeben werden, z.B. [Gamecast.GC_Mission_Manager]. Dadurch ist es möglich, mehrere Klassen in der Konfigurationsdatei anzugeben und so eine Konfigurationsdatei in mehreren Klassen zu verwenden. Eine Klasse kann jedoch nur eine Konfigurationsdatei verwenden. Alle Variablen, welche aus einer Konfigurationsdatei entnommen werden, müssen im UnrealScript mit dem Schlüsselwort `config` gekennzeichnet werden, z.B. `var config int intFromConfig`.

Mögliche Datentypen

Eine Konfigurationsdatei kann sämtliche Datentypen, welche vom UnrealScript bereitgestellt werden, auch an UnrealScript weiterleiten. Möchte man z.B. einen Vector aus der Konfigurationsdatei laden, dann muss lediglich der Vector angegeben werden, z.B. myVect=(X=0,Y=20,Z=40). Im UnrealScript muss nun noch dieser Vector deklariert werden: `var config myVect`.

Es können auch Arrays und Structs weitergeleitet werden. Arrays können wiederum Arrays enthalten und in jedem Array können Structs vorhanden sein. Structs können ebenfalls Arrays enthalten. Arrays werden automatisch erzeugt, sobald in der Konfigurationsdatei der selbe Variablenname verwendet wird, z.B.
`Missions=(startText="1. Mission")`
`Missions=(startText="2. Mission").`
Durch diese Anweisung wird ein Array Missions angelegt, welches aus 2 Elementen besteht.
Die Zuweisung eines Structs in der Konfigurationsdatei geschieht folgendermaßen:
`myStruct=(playerFaction=FACTION_Police, missionActive=5)`
Structs müßen dementsprechend auch im UnrealScript, mit den entsprechenden Variablen, deklariert werden.

2.4 Überblick zu Adobe Flash Professional

Wie in der Zielstellung genannt, muss eine grafische Oberfläche für das Missionssystem entwickelt werden. Dies wird mit Hilfe von Adobe Flash Professional CS5 realisiert. Es eignet sich besonders gut dafür, weil mit Flash schnell grafische Elemente erstellt werden können. Außerdem bietet das UDK die Möglichkeit Flash-Dateien mit Hilfe von Scaleform[5] in das Videospiel zu integrieren und dort abzuspielen. Scaleform ist eine Schnittstelle, um eine Kommunikation zwischen der Unreal Engine und Flash herzustellen.

In Flash lassen sich sowohl statische, als auch dynamische Oberflächen erstellen. Soll eine dynamische Oberfläche erstellt werden, dann ist der Einsatz der Flash-eigenen Programmiersprache ActionScript von Nöten. Dynamische Oberflächen sind besonders dann wichtig, wenn die Oberfläche auf bestimmte Events reagieren soll, oder z.B. bei einem Tastendruck ihr Aussehen verändern soll. So lassen sich bestimmte Texte dynamisch hinzufügen oder entfernen, analog dazu auch Grafiken.

Kommunkation zwischen ActionScript und UnrealScript

Durch Scaleform ist es möglich, im Flash UnrealScript Funktionen aufzurufen und diesen ggf. Parameter zu übergeben. Dafür muss lediglich im ActionScript mit dem Aufruf `ExternalInterface.call(„myUScriptFunc“,param1)` der Name der Funktion im UnrealScript angegeben werden. Damit die Funktion aufgerufen wird, muss sie ebenfalls im UnrealScript unter dem selben Namen, mit den Parametern vom selben Typ, vereinbart werden.

Listing 2.1: Von ActionScript aufrufbare UnrealScript-Funktion

```
function myUScriptFunc(string param1)
{
    `log("param1 received");
}
```

Es ist ebenfalls möglich ActionScript-Funktionen vom UnrealScript aus aufzurufen. Dies wird mit Hilfe der Funktion `ActionScriptVoid` durchgeführt, welche als Parameter denn Namen der Funktion im ActionScript enthalten muss. Parameter, welche an ActionScript übergeben werden sollen, werden durch die UnrealScript-Funktion weitergeleitet. Sie müssen nicht explizit an die `ActionScriptVoid`-Funktion übergeben werden.

[5] http://udn.epicgames.com/Three/Scaleform.html

Listing 2.2: Aufruf von ActionScript-Funktionen

```
function callMyActionScriptFunc(string param1)
{
     ActionScriptVoid("MyActionScriptFunc");
}
```

So würde z.B. der Parameter param1 automatisch an die Flash-Funktion `MyActionScriptFunc` übergeben werden.

2.5 Notwendigkeit einer Skriptsprache für die Missionserstellung

Eine eigene Skriptsprache zur Erstellung von Missionen ist erforderlich, damit die Missionen von Autoren ohne Programmierkenntnisse erstellt werden können. Dadurch müssen sich Autoren nicht erst mit Quellcode auseinander setzen, sondern können sofort mit dem Erstellen der Missionen beginnen. Natürlich ist trotzdem eine Einarbeitungszeit in das System erforderlich, um die gegebenen Möglichkeiten kennenzulernen. Durch die Zusammenarbeit zwischen Autor und Entwickler lässt sich die Skriptsprache komplett auf die Bedürfnisse des Autors anpassen. Dies kann vom Benennen der Schlüsselwörter, bis zum Implementieren neuer Features reichen.

Mit der Entwicklung der Skriptsprache geht auch die Auslagerung der Missionen aus dem Quellcode einher. Demzufolge muss der Quellcode nicht neu kompiliert werden, falls irgendetwas modifiziert wird. Daraus erschließt sich der Vorteil, dass der Autor völlig getrennt vom Programmierer die Missionen in das Videospiel einbringen und so das Gameplay erweitern kann.

2.6 Adaptiver Ansatz im Missionssystem

Die adaptive Komponente des Missionssystems wird realisiert, indem die Spieler in verschiedene Spielertypen eingeteilt werden. Folgende Spielertypen sind möglich:

Tabelle 2.1: Spielertypen

Typ	**Beschreibung**
Shooter	Das Verhalten des Spielers gilt als besonders aggressiv. D.h. er benutzt übermäßig oft seinen Taser um Personen zu betäuben.
Hacker	Der Spieler hat besonders viele Hotspots gehackt und repräsentiert so dem zurückhaltenderen Typ, welcher das Spiel ohne direkte Konfrontation lösen möchte.

Tabelle 2.1: Fortsetzung Spielertypen

Schlüsselwort (Datentyp)	Beschreibung
Animator	Animationen werden vom Spieler oft eingesetzt, um sich auszudrücken, z.B. Winken oder Tanzen. Dadurch entspricht der Spieler einem schauspielerischen Typ.
Blinder	Der Spieler blendet oft Personen, damit diese für eine kurze Zeit nichts mehr erkennen können. Er entspricht deshalb einem Spielertyp, welcher gerne in Konfrontation mit anderen Spielern gerät.

Das Verhalten des Spielers wird während einer Spielpartie ständig überwacht. Wenn er z.B. eine Animation ausführt, dann wird dies mitgeloggt und eine entsprechende Variable hochgezählt. Jeder Spielertyp gibt an, wie oft eine bestimmte Aktion durch den Spieler ausgeführt wurde. Die adaptive Verteilung des Missionen greift auf diese Spielertypen zurück. Der Autor der Missionen muss angeben, welchen Spielertyp die Mission benötigt. Er muss außerdem angeben, ab welcher Anzahl an Ausführungen der bestimmte Spielertyp gilt, z.B. dass der Spieler bei 20 ausgeführten Animationen als Animator gilt. Wenn ein Spieler nun den benötigten Spielertyp erfüllt, dann erhält er auch die Mission.

2.7 Vorgehen bei der Entwicklung

Für die Entwicklung des adaptiven Missionssystems wurden Ansätze aus dem Vorgehensmodell Scrum verwendet, besonders die Prinzipien „Zerlegung“, „Überprüfung“ und „Anpassung“ wurden beachtet [6]. Zuerst mussten die Anforderungen an das Missionssystem gestellt werden, z.B. welche Arten von Missionen es geben soll. Außerdem musste definiert werden, welche Spielertypen es geben soll, damit die Missionen an einen bestimmten Spielertyp adaptiv verteilt werden können.

Als nächstes musste eine Architektur konzipiert werden, welche sowohl die Erfassung der Spielerdaten, als auch die Überwachung und Verteilung der Missionen an die Spieler beschreiben konnte. Besonders das Verhalten zwischen dem Spiel-Server und den Clients musste bei der Planung der Architektur beachtet werden.

Nach der Konzeption der Architektur, wurden die benötigten Funktionalitäten für das Missionssystem nach einer Priorität sortiert. Als aller erstes musste eine Methode gefunden werden, wie die Spielerdaten erfasst und gespeichert werden konnten. Danach wurde diese Erfassung implementiert. Die benötigten Spielertypen wurden ebenfalls festgelegt.

[6] http://de.wikipedia.org/wiki/Scrum

Danach musste ein geeigneter Weg gefunden werden, damit der Autor selbständig Missionen erstellen konnte. Anschließend wurden die Missionstypen entwickelt. Die Missionstypen wurden aufgeteilt und jede einzeln implementiert. Parallel dazu wurde auch die Skriptsprache zur Erstellung dieser Missionen implementiert. Jeden Tag wurde eine kurze Absprache mit dem zukünftigen Autor der Missionen gehalten, ob der Missionstyp die benötigte Funktionalität erfüllt. Nach der Implementierung galt der Missionstyp jedoch nicht als endgültig abgeschlossen, obwohl die Funktionalität gegeben war. Es kam vor, dass dieser erweitert werden musste. So musste eine Mission vom Typ „MEET_AT_LOCATION“, welche dem Spieler einen bestimmten Ort zum Aufsuchen gibt, erweitert werden. In diesem Missionstyp konnte man zu Beginn nur den Endort angeben, jedoch wurde vom Autor gewünscht, dass sich der Spieler anhand von Wegpunkten zu einem Ziel hinführen lassen muss. Deswegen musste dieser Missionstyp um diese Funktionalität erweitert werden.

Jede Implementierung eines Missionstyp wurde ausgiebig getestet, damit so wenige Seiteneffekte wie möglich auftreten. Da eine Mission, unabhängig vom Verhalten des Spielers, immer abgeschlossen werden muss, mussten hier bestimmte Sicherheiten implementiert werden.

Während der Implementierung der Missionen wurde als Ausgabe für den Erfolg oder Misserfolg, eine von Unreal bereitgestellte Funktion verwendet. Diese gibt lediglich einen Text auf dem Bildschirm des Spielers aus, welcher nach wenigen Sekunden wieder verschwindet. Deswegen wurde als letzter Schritt die grafische Oberfläche für die Missionen mit Adobe Flash Professional entwickelt. Auch hier wurde mit dem Autor zusammengearbeitet, um ein möglichst leicht zu bedienende Oberfläche zu implementieren.

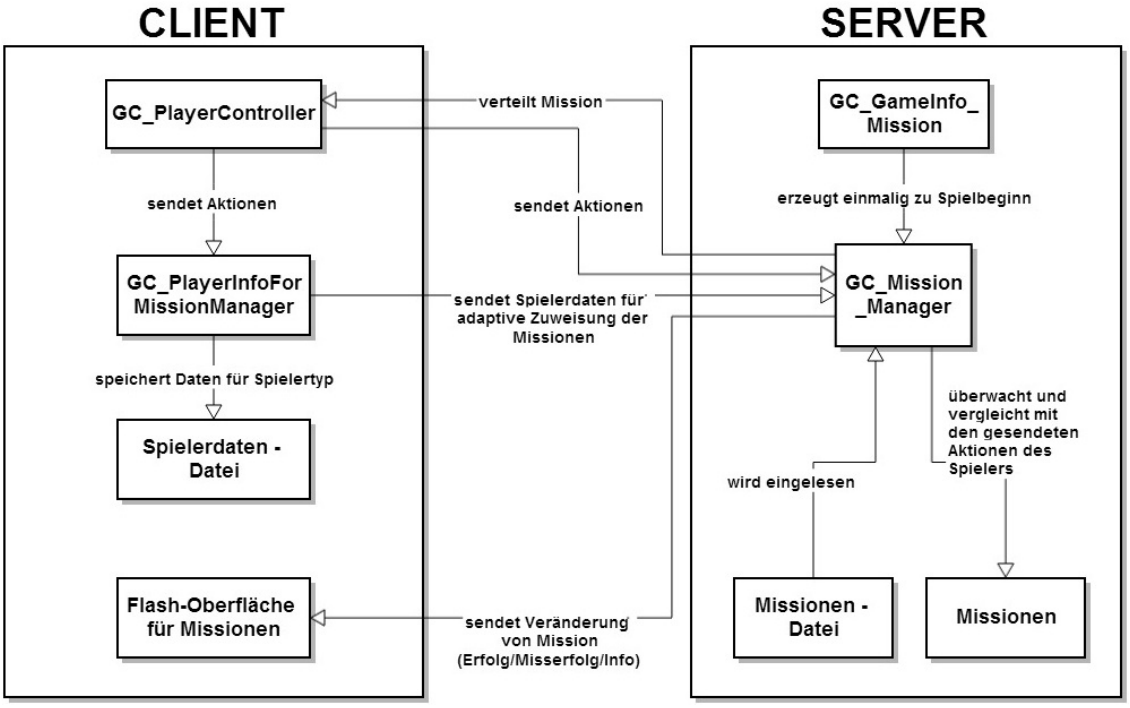
2.8 Architektur des Systems

Da es sich bei Urban Legend um ein Multiplayer Spiel handelt, muss das Missionssystem auf einer Client-Server-Struktur aufgebaut werden. Der Missionsmanager wird durch die Klasse `GC_Mission_Manager` repräsentiert. Der Manager wird beim ersten Start in der Klasse `GC_GameInfo_Mission` als ein neuer Actor initialisiert. Alle Objekte, welche auf dem Server laufen, müssen in einer GameInfo initialisiert werden.

Der Missionsmanager verteilt Missionen an einen Spieler, also an den Client. Zum Client gehören mehrere Klassen, welche für das Missionssystem von Bedeutung sind. Die Klasse `GC_PlayerController` dient als Bindeglied zwischen Client und Server, um Informationen untereinander auszutauschen. Sämtliche Daten über den Spielertyp werden von der Klasse `GC_PlayerInfoForMissionManager` bereitgestellt, dabei ist ein Objekt dieser Klasse dem `GC_PlayerController` zugeteilt. Dadurch hat auch der Server Zugriff auf diese Daten und kann sie zur adaptiven Verteilung der Missionen verwenden.

Die Datei mit den erstellten Missionen wird vom Missionsmanager eingelesen, um verteilt werden zu können. Sobald ein Spieler eine Aktion durchführt, z.B. Schießen, wird diese dem `GC_PlayerInfoForMissionManager` mitgeteilt. Dadurch wird der Spielertyp weiter bestimmt. Außerdem erhält der Missionsmanager ebenfalls eine Benachrichtigung über die Aktion und überprüft, ob eine Mission durch das Eintreten dieses Ereignisses abgeschlossen werden kann. Wenn das der Fall ist, sendet der Missionsmanager an die Flash-Oberfläche des Spielers die entsprechende Nachricht.

Abbildung 2.3: Übersicht über das Missionssystem



3 Lösungskonzeption

3.1 Entwicklung einer geeigneten Skriptsprache

Durch die Skriptsprache soll eine Art Baukasten geschaffen werden, mit dem es möglich ist, bestimmte Missionen und Missionstypen zu definieren. Die Sprache soll so gestaltet werden, dass der Autor mit geringem Aufwand, eine Menge an grundlegenden Missionen erstellen kann. Es soll die Möglichkeit bestehen, diese Missionen dann weiter auszubauen, um diesen mehr Komplexität zu verleihen. Der Autor soll außerdem die Möglichkeit haben, Missionsstränge zu definieren, um so eine Handlung aufzubauen.

Wie in Abschnitt 2.5 genannt, muss die Erstellung der Missionen außerhalb des Quellcodes stattfinden. Das bedeutet, dass die Missionen mit Hilfe der eigenen Skriptsprache, in einer eigenen Datei definiert werden müssen. Der Inhalt dieser Datei muss anschließend, beim Starten des Videospiels, wieder zurück in die Unreal Engine eingelesen werden. Diese Missionsdaten müssen dann von Funktionen, welche in UnrealScript realisiert werden, verarbeitet werden. Im folgenden werden 2 unterschiedliche Arten vorgestellt, wie man das Einlesen der Daten ermöglichen könnte.

3.1.1 1. Variante: Einlesen der Daten aus einer XML Datei

Die Missionsdaten könnten in eine XML-Datei eingegeben werden. Damit würden sich die Daten sehr übersichtlich darstellen lassen, was einen großen Vorteil beim Erstellen der Missionen bieten würde. So könnten beispielsweise Missions-Variablen (Schlüsselwörter) in Kategorien eingeteilt werden. Der größte Nachteil wäre allerdings der Implementierungsaufwand. Unreal unterstützt keine direkte Schnittstelle, um XML-Dateien einlesen zu können. Um dieses Problem zu lösen, müsste mit Hilfe von C++ eine DLL geschrieben werden, welche die XML-Daten ausliest. Diese Daten müssten dann an UnrealScript gesendet und mit Hilfe eines selbstgeschriebenen Parsers in die missionsspezifischen Variablen übertragen werden.

3.1.2 2. Variante: Einlesen der Daten aus einer Konfigurationsdatei

Wie in Abschnitt 2.3.2 beschrieben, besitzt die Unreal Engine die Möglichkeit, Konfigurationsdateien einzulesen. Das hätte den erheblichen Vorteil, dass für die Verarbeitung der Missionsdaten kein extra Parser benötigt wird. Die Daten könnten sofort an UnrealScript weitergeleitet werden. Falls der Autor beim Erstellen der Missionen ein Schlüsselwort falsch benennen würde, würde die Konsole sofort eine Fehlermeldung ausgeben, dass das Schlüsselwort unbekannt ist. Falls beim Schreiben der Missionen syntaktische Feh-

ler auftreten, würde somit ein einfacher Weg existieren, um den Fehler zu finden. Ein weiterer Vorteil ist, dass die Schlüsselwörter in keiner bestimmten Reihenfolge, beim Definieren der Missionen, angegeben werden müssen. Nicht benötigte Schlüsselwörter können auch komplett weggelassen werden, weil diese immer mit einem Standardwert initialisiert sind.

Ein Nachteil bei der Verwendung von Konfigurationsdateien ist die teilweise schlechte Übersichtlichkeit. So muss eine Mission komplett auf einer Zeile geschrieben werden, dabei leidet die Übersichtlichkeit besonders, wenn die Missionstexte sehr lang sind und dadurch die Zeile besonders lang wird. Dieses Problem könnte teilweise behoben werden, wenn ein Editor mit Zeilenumbruch verwendet wird.

3.1.3 Entscheidung für eine Variante

Der Verwendung von Konfigurationsdateien wird der Vorzug gegeben, weil diese sehr einfach benutzt werden können und eine schnelle Fehlersuche bei syntaktischen Fehlern bieten. Das Implementieren eines eigenen Parsers und das Einbinden einer DLL entfällt ebenfalls komplett.

3.2 Verwaltung von Missionen

Der Spiele-Server muss ständig überwachen, welche Missionen gerade aktiv sind und ob Missionen verteilt werden können. Er muss Spieler registrieren und überprüfen, ob sie berechtigt sind eine Mission zu erhalten. Es ist ebenfalls seine Aufgabe, zu prüfen, ob Missionen erfolgreich abgeschlossen wurden oder gescheitert sind. Er muss Nachrichten an Spieler senden, dass sie eine neue Mission erhalten haben oder eine abgeschlossen haben. Neben der Verteilung der Missionen an die Spieler, muss ebenfalls eine Synchronisation von Missionen untereinander stattfinden. Dadurch kann eine sinnvolle Handlung eingehalten werden, welche vorher vom Autor festgelegt wurde. Es müssen bestimmte Voraussetzungen geschaffen werden, wann Missionen gestartet oder abgeschlossen werden.

Der Server soll außerdem die Fähigkeit besitzen, bestimmte Missionen zu pausieren, um einen Missionsstrang zu starten, welcher in Zusammenhang mit anderen Spielern steht. Sobald dieser Missionsstrang abgeschlossen ist, soll die letzte geführte Mission wieder aktiviert werden. Dadurch wird dem Autor ermöglicht, Single-Player Missionen zu schreiben, welche unabhängig von anderen Spielern sind. Der Spieler hätte so immer Missionen, die er erledigen könnte und dadurch würde kein Leerlauf im Spielerlebnis entstehen. Sobald der Server erkennt, dass eine Multiplayer Mission gestartet werden kann, wird die Singleplayer Mission pausiert. Sobald alle Multiplayer Missionen erledigt wurden, wird die Singleplayer Mission fortgesetzt.

3.3 Erfassung von Spieler-Daten

Missionen können nur adaptiv und sinnvoll an einen Spieler verteilt werden, wenn bestimmte Informationen über diesen Spieler vorhanden sind. Der Spieler muss also einem bestimmten Typ zugeordnet werden, damit eine sinnvolle Verteilung stattfinden kann. Es müssen also gewisse Spieler-Daten erfasst werden. Aus diesem Grund muss der Spiele-Server ständig beobachten, ob und welche Aktionen der Spieler ausführt. Diese Werte muss der Server in eine Datei abspeichern, damit die Spieler-Daten auch nach dem Ende einer Partie wieder verwendbar sind. Somit lässt sich ein Profil über einen längeren Zeitraum bilden. Die Datei muss außerdem über die ganze Partie hinweg, vom Server aktualisiert werden. Dadurch wird sichergestellt, dass sich der Spieler während einer Spiel-Partie in seinem Typ weiterentwickeln kann. Somit wird es möglich dem Spieler Missionen zu geben, welche noch am Anfang der Partie nicht in Frage kamen.

3.4 Singleplayer Missionen und Adaptivität

Durch Singleplayer Missionen kann ein Spieler beschäftigt werden, solange eine Multiplayer Mission nicht gestartet werden kann. Es kann durchaus vorkommen, dass ein Spieler durch die Singleplayer Missionen in einen bestimmten Spielertyp gedrängt wird. Dies kann besonders schnell passieren, wenn ein Spieler zum ersten mal in das Spiel einsteigt und somit keine Spieler-Daten über diesen vorhanden sind. Bei Spielern, welche schon ein Profil besitzen, ist das weniger ein Problem, da die Verteilung der Singleplayer Missionen hier adaptiv geschehen kann. Deshalb ist es Aufgabe des Autors, Missionen zu verfassen, welche den Spieler nicht sofort in einen bestimmten Spielertyp drängen. Dies kann dadurch realisiert werden, indem dem Spieler bestimmte Freiheiten bei der Durchführung der Missionen gelassen werden. So könnte eine Mission lauten: „Bewege dich zum Zielort und hacke Hotspots oder scanne nach verdächtigen Personen und betäube diese“. Dadurch steht es dem Spieler frei, ob er auf dem Weg zum Zielort hackt und so seinen Hacker-Wert steigert oder Personen betäubt und seinen Shooter-Wert steigert.

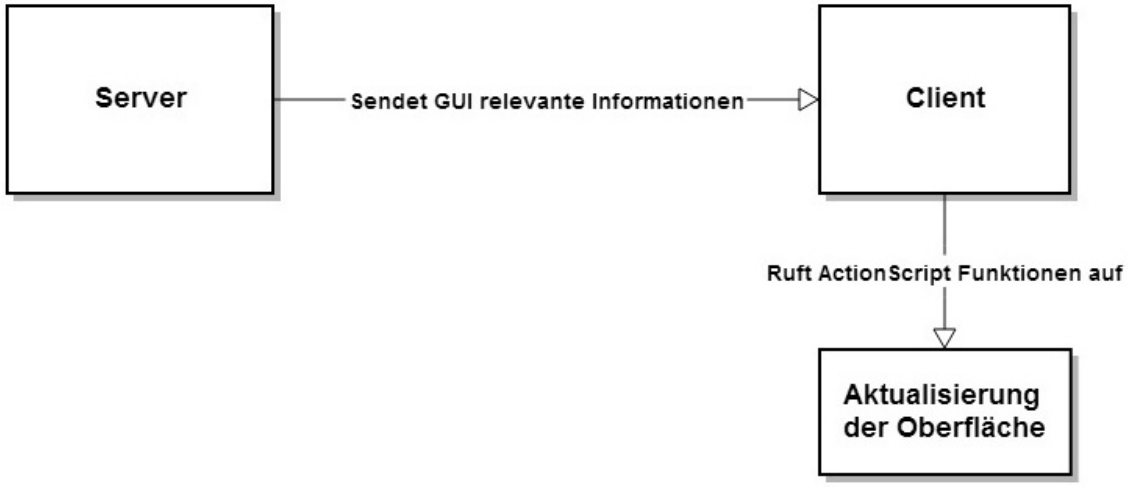
3.5 Anforderungen an die Benutzeroberfläche

Eine Oberfläche für die Missionen muss erstellt werden, damit der Spieler einen Überblick über diese hat. Der Spieler soll erkennen können, ob eine Mission bereits abgeschlossen ist. Eine gewisse Anzahl an Missionen soll auf einen Blick sichtbar sein. Details über eine spezifische Missionen, soll mit einem Klick in die Oberfläche, angezeigt werden.

Um die oberen Anforderungen zu realisieren, muss der Spiele-Server missionsspe-

zifische Informationen an die Clients der Spieler senden. Diese Informationen müssen dann an die Flash-GUI gesendet und von ActionScript verarbeitet werden, um die Missions-Elemente in der GUI aufzubauen. Der Server muss also wissen, welchem Spieler er welche Informationen zusenden muss. Er muss ebenfalls ActionScript mitteilen, ob eine Mission gerade aktualisiert wurde, abgeschlossen oder gescheitert ist. Eine eindeutige Zuweisung, um welche Mission es sich überhaupt handelt, muss ebenfalls stattfinden.

Abbildung 3.1: Kommunikation zwischen Server und GUI



4 Technische Realisierung

4.1 Erfassung und Auswertung der Spieler-Daten

In dieser Arbeit werden nur beispielhaft einige Werte erfasst, um daraus einen Typ für den Spieler zu bilden. Der erstellte Spielertyp dient lediglich der Demonstration der adaptiven Komponente des Missionssystems. So wird z.B. nicht berücksichtigt, wie viele Spielpartien der Spieler durchgeführt hat. Es wird nur aufgezeichnet, wie oft der Spieler eine bestimmte Aktion durchgeführt hat.

Die Spieler-Daten werden in eine Konfigurationsdatei gespeichert, dies bietet die bereits in Kapitel 3.1.2 genannten Vorteile. Allerdings existiert ein Nachteil bei der Verwendung dieser Methode. Die Konfigurationsdatei wird lokal im Verzeichnis der Urban Legend Installation angelegt. Somit wäre es möglich, dass der Spieler die Datei modifiziert und so seinen ermittelten Spielertyp an seine Wünsche anpasst. Außerdem wird in der Konfigurationsdatei nicht berücksichtigt, welcher Spieler sich gerade am PC befindet. Das liegt vor allem daran, dass in Urban Legend noch kein Login System mit Spieler-Accounts integriert wurde. Sobald jemand eine Spielpartie beginnt, wird die Konfigurationsdatei mit den Spieler-Daten ständig modifiziert.

Anlegen der Konfigurationsdatei

Es muss eine Konfigurationsdatei angelegt werden, welche den Typ eines Spielers repräsentiert. Sie muss bestimmte Variablen beinhalten, durch welche sich ein Spielertyp aufbauen lässt.

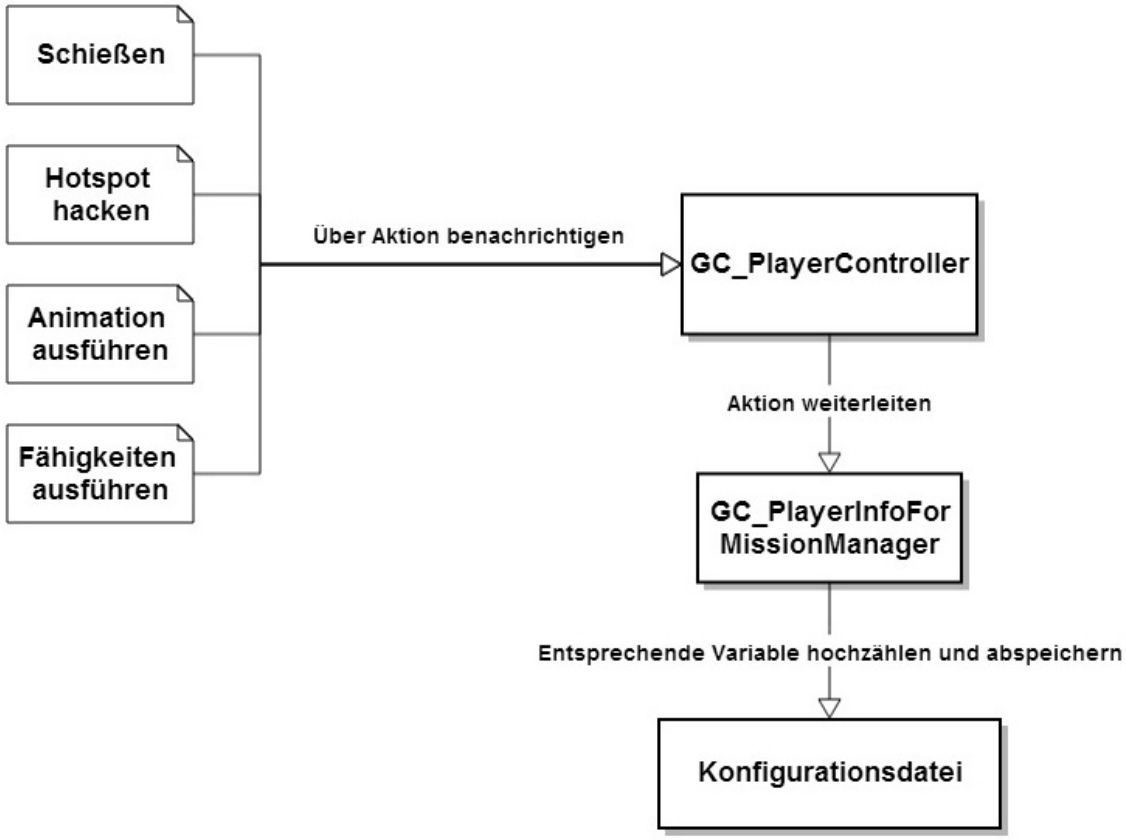
Listing 4.1: Konfigurationsdatei mit Spieler-Daten

```
InfoForMissions=(hittedWithTaser=0,
                 hackedHotspots=0,
                 playedAnimations=(animShrug=0,
                                   animScratchHead=0,
                                   animWave=0,
                                   animClapHand=0,
                                   animDance=0,
                                   animListen=0,
                                   animTalk=0,
                                   animIdle=0),
                 usedTechnologies=(techAllTwin=0,
                                   techBlind=0,
                                   techSelfDisguise=0,
                                   techDisguisePolice=0,
                                   techAllRun=0,
                                   techEmoScan=0,
```

```
                techTagging=0,
                techBeacon=0,
                techCallDrone=0)
        )
```

Wie an der Konfigurationsdatei zu sehen ist, besitzt sie mehrere Variablen, welche die Anzahl der Ausführungen von bestimmten Aktionen darstellen. Sobald diese Aktion ausgeführt wird, wird die Variable hochgezählt. Eine Klasse `GC_PlayerInforForMissionManager` wird geschrieben, um die Aktionen zu empfangen und dementsprechend die Variablen hochzuzählen. Sie ist ebenfalls dafür zuständig diese Daten bereitzustellen, damit sie vom Server ausgelesen werden können und für die Missionen verwendet werden können. Da jeder Spieler seine eigenen Daten hat, wird ein Objekt der Klasse `GC_PlayerInforForMissionManager` dem PlayerController zugewiesen.

Abbildung 4.1: Erfassung der Spielerdaten



Anzahl von getroffenen Personen

Ein Pawn besitzt eine Funktion, mit welcher ein Treffer auf diesen registriert wird. Es wird ebenfalls der Verursacher dieses Treffers übermittelt. Sobald nun also eine Figur mit dem Taser getroffen wurde, wird die Variable `hittedWithTaser` des Verursachers hochgezählt, und somit bestimmt, wie oft der Spieler eine Figur getroffen hat. Dadurch könnte man dem Spieler Missionen geben, welche eine hohe Anzahl an Schüssen erfordert.

Anzahl an gehackten Hotspots

Ein Hotspot besitzt eine Funktion, mit welcher ermittelt wird, ob er gerade gehackt wurde. Der Hacker des Hotspots wird ebenfalls übermittelt. Durch den PlayerController des Hackers kann wiederum die Variable `hackedHotspots` hochgezählt werden. Daraus lässt sich wieder bestimmen, wie viele Hotspots der Spieler gehackt hat.

ausgeführte Animationen

Der Spieler kann bestimmte Animationen ausführen, wie z.B. Winken oder Tanzen. Diese können, mit Hilfe des NumPads auf der Tastatur, ausgelöst werden. Die Funktion zum Auslösen der Tatstatur-Befehle befindet sich im PlayerController. Sobald eine bestimmte Taste für die Animation gedrückt wurde, wird diese ausgewertet und der entsprechende Wert in `playedAnimations` erhöht. Durch die Trennung der Animationen nach ihrer Art, könnte z.B. bestimmt werden, ob der Spieler gern tanzt und diesem so Missionen zugewiesen werden, die dieses Verhalten erfordern.

ausgeführte Fähigkeiten

Durch den Einsatz von Fähigkeiten können Spieler bestimmte Effekte bewirken. So können Rebellen z.B. Polizisten blenden, sodass deren Bildschirm für einige Sekunden mit einer weißen Farbe überblendet wird. Die Funktionen für diese Fähigkeiten befinden sich ebenfalls im PlayerController. Sobald eine Fähigkeit ausgelöst wird, wird diese ebenfalls mitgeloggt und dementsprechend in `usedTechnologies` ausgewertet. Da die Fähigkeiten hier ebenfalls nach ihrer Art getrennt werden, kann auch hier bestimmt werden, welche Fähigkeiten der Spieler besonders gern ausführt.

Auswertung der Daten für die Missionen

In der Konfigurationsdatei der Missionen werden Variablen festgelegt, welche angeben, ab welcher Anzahl der Spieler als bestimmter Typ gilt. Diese lauten:

```
SHOOTER_AT=20       //Spieler gilt als Shooter, wird mit dem Wert
                    //hittedWithTaser verglichen
ANIMATOR_AT=5       //Spieler gilt als Animator, wird mit der Summe
                    //aller Animationen von playedAnimations
                    //verglichen
HACKER_AT=5         //Spieler gilt als Hacker, wird mit hackedHotspots
                    //verglichen
BLINDER_AT=10       //Spieler gilt als Blinder, wird mit techBlind
                    //verglichen
```

In der Klasse `GC_PlayerInforForMissionManager` werden nun diese Variablen mit den entsprechenden Variablen in der Konfigurationsdatei mit den Spieler-Daten verglichen. Anschließend gibt die Funktion zurück, ob der Spieler zu diesem bestimmten Typ gehört oder nicht. Bei der Definition von Missionen kann außerdem festgelegt werden, zu wie viel Prozent der Spieler einem bestimmten Typ entsprechen soll. So kann z.B. angegeben werden, dass die Variable `SHOOTER_AT` bereits schon ab 50% als erfüllt gelten soll. So würde der Spieler unter Einbeziehung des obigen Beispiels als Shooter gelten, wenn er 10 Personen getroffen hätte. Es ist ebenfalls möglich mehrere Typen miteinander zu vermischen, so wäre es z.B. möglich zu definieren, dass der Spieler zu 80% Shooter sein soll und zu 20% ein Hacker.

4.2 Aufbau der Skriptsprache zur Erstellung von Missionen

Für die Erstellung von Missionen werden, wie in Abschnitt 3.1.3 genannt, Konfigurationsdateien verwendet. Mit Hilfe bestimmter Schlüsselwörter werden Missionen aufgebaut.

4.2.1 Missionsstränge

Der Autor kann Missionsstränge angeben, um dem Server mitzuteilen, welche Missionen zu einer größeren Handlung zusammengehören. Außerdem kann der Autor festlegen, wie viele Spieler für den Missionsstrang benötigt werden, damit dieser ausgelöst wird. Für das Starten des Missionsstrangs, kann ebenfalls der Typ des Spielers hinzugezogen werden. So kann z.B. angegeben werden, dass sich 3 Spieler von der Fraktion Rebellen im Spiel befinden müssen. Diese Spieler müssen außerdem schon mindestens 10 Hotspots gehackt haben. Erst wenn all diese Voraussetzungen erfüllt sind, kann der Missionsstrang gestartet werden.

Listing 4.2: Beispiel für einen Missionsstrang

```
MissionThreads=(id=1,
                playerType[0]=(playerFaction=FACTION_Police),
                playerType[1]=(playerFaction=FACTION_Rebels),
                playerType[2]=(playerFaction=FACTION_Rebels))
```

Folgende Schlüsselwörter kann ein MissionThread enthalten:

Tabelle 4.1: Schlüsselwörter für einen Missionsstrang

Schlüsselwort (Datentyp)	**Beschreibung**
`id` (int)	ID des Missionsstranges, muss beim Definieren von Missionen ebenfalls angegeben werden.
`playerType[i]` (struct)	Typ des Spielers. Die Angabe von mehreren Spielertypen, teilt dem Server mit, dass für den Missionsstrang mehrere Spieler benötigt werden. Erst wenn alle diese Spieler vorhanden sind und die Spielertypen übereinstimmen, wird der Missionsstrang gestartet (siehe Beispiel in Listing 4.2).
`bCanBeAborted` (bool)	Gibt an, ob die Missionen dieses Strangs als Singleplayer Missionen angesehen werden sollen.

Da das Schlüsselwort `playerType[i]` durch ein Struct repräsentiert wird, werden im folgenden die Möglichen Variablen aufgezeigt:

Tabelle 4.2: Schlüsselwörter für den Spielertyp

Schlüsselwort (Datentyp)	**Beschreibung**
`playerFaction` (Enum)	Repräsentiert die Fraktion des Spielers. Folgende Werte sind zulässig: FACTION_Police, FACTION_Rebels, FACTION_Neutral (Techniker).
`shooter` (int (0-100))	Gibt an, zu wieviel Prozent der Spieler, den Typ Shooter, mindestens erfüllen soll (Anzahl der Treffer mit dem Taser auf Personen).
`hacker` (int (0-100))	Analog zu `shooter`, allerdings vom Typ Hacker (Anzahl der gehackten Hotspots).
`animator` (int (0-100))	Analog zu `shooter`, allerdings vom Typ Animator (Anzahl der ausgeführten Animationen).
`blinder` (int (0-100))	Analog zu `shooter`, allerdings vom Typ Blinder (Anzahl der Nutzungen der Fähigkeit „Blenden“).
`character` (string)	Gibt an, welches Aussehen die Figur des Spielers haben muss.

4.2.2 Missionen

Missionen können erstellt werden, um dem Spieler ein Aufgabe zu geben. Es ist möglich verschiedene Typen von Missionen zu definieren, um so unterschiedlich Ziele zu verfolgen. Die Synchronisierung von Missionen kann ebenfalls vorgenommen werden, so könnte z.B. eine Mission erst gestartet werden, wenn eine andere bereits aktiv ist. Aus der Menge der möglichen Schlüsselwörter zur Definition von Missionen, existieren Schlüsselwörter, welche für alle Missionen gültig sind und Variablen, welche nur für einen bestimmten Missionstyp gültig sind.

allgemeingültige Schlüsselwörter

Tabelle 4.3: Allgemeingültige Schlüsselwörter für Missionen

Schlüsselwort (Datentyp)	**Beschreibung**
`thread` (int)	ID des Missionsstrangs, damit der Server weiß, welche Missionen zu einer Handlung gehören.
`id` (int)	ID der Mission, jede Mission muss eine einzigartige ID haben, unabhängig vom Missionsstrang.
`bHasPrevMission` (bool)	Die Mission hat eine andere Mission als Voraussetzung, welche vorher erfüllt werden muss.
`bIsDynamic` (bool)	Die Mission wird dynamisch an eine Spieler verteilt, wenn dieser bestimmte Voraussetzungen erfüllt.
`count` (int)	Gibt an, wie oft die Mission verteilt werden soll. Wird an Spieler verteilt, welche die gleichen Voraussetzungen erfüllen.
`conditions` (struct)	Beschreibt die Voraussetzungen, die benötigt werden, damit die Mission gestartet werden kann. Es können die selben Schlüsselwörter, wie in Tabelle 4.2 gezeigt, verwendet werden. Damit wird sichergestellt, dass die Mission an einen bestimmten Spielertyp übergeben wird. Hier kann ein weiteres Schlüsselword verwendet werden: `missionActive` (int). Dieses gibt an, dass eine andere Mission gerade aktiv sein muss, damit die aktuelle Mission vergeben werden kann.
`missionType` (string)	Gibt an, um welchen Missionstyp es sich handelt.
`startText` (string)	Text, welcher in der grafischen Oberfläche angezeigt wird, wenn der Spieler eine neue Mission erhält.
`successText` (string)	Text, nachdem eine Mission erfolgreich abgeschlossen wurde.
`failText` (string)	Text, nachdem eine Mission gescheitert ist.

Tabelle 4.3: Fortsetzung Allgemeingültige Schlüsselwörter für Missionen

Schlüsselwort (Datentyp)	**Beschreibung**
`relevantCharFromMission` (int)	Ein Spieler aus einer anderen Mission kann als relevant angegeben werden. Das ist wichtig, wenn ein Spieler z.B. eine Mission hat, in welcher er einen Spieler aus einer anderen Mission blenden soll. Der übergebene Wert muss dabei die ID der entsprechenden Mission sein, welche diesen relevanten Spieler beinhaltet.
`showPicFromPlayer` (int)	Gibt an, ob ein Bild eines Spielers aus einer anderen Mission in der Oberfläche für die Missionen angezeigt werden soll. Der Wert ist die ID der entsprechenden Mission.
`playAnimAfterSuccess` / `playAnimAfterFail` (string)	Eine Animation kann nach dem Erfolg/Misserfolg einer Mission vom Spieler automatisch abgespielt werden. Mögliche Werte für den String: „shrug“, „scratchHead“, „wave“, „clapHand“, „dance“, „listen“, „talk“ und „idle“.
`bInvisOnCompass` (bool)	Es existiert ein Missionstyp, bei dem z.B. Spieler A Ausschau nach Personen halten soll. Sobald ein Spieler B in der Nähe ist, wird Spiele A benachrichtigt und Spieler B wird diesem als Ziel angezeigt. Mit diesem Schlüsselwort ist es möglich, dass der Spieler B nicht als Ziel angezeigt wird.
`timer` (float)	Ein Timer kann für eine Mission festgelegt werden, wenn der Spieler die Mission nicht in der angegebenen Zeit erfüllt, dann gilt sie als gescheitert.
`nextMission` (int)	Nachdem eine Mission erfolgreich erfüllt wurde, kann mit diesem Schlüsselwort angegeben werden, welche Mission im Anschluss gestartet werden soll. Dadurch lassen sich Missionsketten bilden. Der Wert entspricht der `id` der nächsten Mission.
`nextMissionAfterFail` (int)	Analog zu `nextMission`, allerdings nachdem eine Mission gescheitert ist.

Es ist möglich sogenannte optionale Nachrichten in den Missionen zu definieren. Diese Nachrichten werden angezeigt, sobald der Spieler z.B. an einem bestimmten Ort vorbeigekommen ist oder wenn ihm etwas zugestoßen ist. Sie müssen nicht unbedingt während der Mission ausgelöst werden, um die Mission erfolgreich abzuschließen. Sie dienen lediglich dem Zweck, die Missionen etwas lebendiger darzustellen. Dies kann mit dem Schlüsselwort `optMsg[i]` umgesetzt werden. Der Syntax ist ähnlich der in den Missionssträngen beschriebenen `playerDetails[i]`, d.h. es können mehrere Nachrichten

für eine Mission definiert werden, indem der Index i erhöht wird. Folgende Schlüsselwörter kann eine optionale Nachricht enthalten:

Tabelle 4.4: Schlüsselwörter für optionale Nachrichten

Schlüsselwort (Datentyp)	**Beschreibung**
`msgCond` (struct)	Beschreibt die Voraussetzungen für das Eintreten der Nachricht.
`msgType` (string)	Gibt an, um welche Art von Nachricht es sich handelt, z.B. nur eine Aktualisierung der Mission. Mögliche Werte: „info“.
`msg` (string)	Repräsentiert den eigentlichen Text, welcher bei der Nachricht eingeblendet wird.

Für das Schlüsselwort `msgCond` existieren ebenfalls mehrere Variablen, welche verschiedene Zustände zum auslösen der Nachricht repräsentieren:

Tabelle 4.5: Schlüsselwörter für msgCond

Schlüsselwort (Datentyp)	**Beschreibung**
`bPlayerBlinded` (bool)	Nachricht wird ausgelöst, wenn der Spieler der aktuellen Mission geblendet wurde.
`minDistanceToLocation` (int) & `nearLocation` (vector)	Wenn der Spieler in einem bestimmten Abstand in Metern (`minDistanceToLocation`) zu einem bestimmten Ort (`nearLocation`) ist, wird die Nachricht ausgelöst. Beide Schlüsselwörter müssen angegeben werden, falls diese Voraussetzung angegeben wird.
`minDistanceToPlayer` (int) & `nearPlayerFromMission` (int)	Wenn der Spieler in einem bestimmten Abstand in Metern (minDistanceToPlayer) zu einem Spieler aus einer anderen Mission ist, wird die Nachricht ausgelöst. Für `nearPlayerFromMission` muss die id der Mission übergeben werden.

Es besteht ebenfalls die Möglichkeit, den Spieler mit einem Gegenstand zu belohnen, wenn dieser eine Mission abgeschlossen hat. Dafür muss das Schlüsselwort `reward` verwendet werden, welches folgende Variablen enthalten kann:

Tabelle 4.6: Schlüsselwörter für reward

Schlüsselwort (Datentyp)	**Beschreibung**
`rewardType` (string)	Gibt an, um welche Belohnung es sich handelt. Mögliche Belohnungen sind: „pistol“ und „flashlight“.

Tabelle 4.6: Fortsetzung Schlüsselwörter für reward

Schlüsselwort (Datentyp)	Beschreibung
`bInstantStun` (bool)	Falls der Taser („pistol“) als Belohnung ausgewählt wurde, kann festgelegt werden, ob dieser mit nur einem Schuss eine Person betäuben soll. Standardmäßig wird eine Person nach 4 Schüssen betäubt.
`ammoCount` (int)	Für die Pistole kann angegeben werden, wieviel Munition diese enthalten soll. Standardmäßig wird die Munition langsam aufgefüllt. Durch die Begrenzung der Munition lässt sich eine überlegte Handhabung mit dem Taser sicherstellen.

missionsspezifische Schlüsselwörter

Wie in Tabelle 4.3 gezeigt, muss beim Erstellen von Missionen der Missionstyp (missionType) angegeben werden. Abhängig vom Missionstyp müssen auch verschiedene Schlüsselwörter benutzt werden.

MEET_AT_LOCATION

Mit diesem Missionstyp erhält der Spieler die Anweisung, zu einem bestimmten Ort im Spiel zu gehen. Die Mission gilt als erfüllt, wenn er diesen Ort erreicht hat. Es ist ebenfalls möglich, eine Menge an Wegpunkten anzugeben. Dies bietet den Vorteil, dass der Spieler gezielt auf einer Route geführt werden kann. Somit wäre es z.B. möglich die Wegpunkte so zu verteilen, dass der Spieler auf andere Spieler trifft oder an wichtigen Orten vorbei kommt. Standardmäßig wird dem Spieler auf einem Kompass angezeigt, wie weit dieser Ort entfernt ist und in welcher Richtung er liegt.

Abbildung 4.2: Richtungsanzeige für Missionen

Tabelle 4.7: spezifische Schlüsselwörter für MEET_AT_LOCATION

Schlüsselwort (Datentyp)	Beschreibung
`targetLocation` (vector)	Gibt den Zielort an, zu dem sich der Spieler bewegen muss. Das Schlüsselwort muss verwendet werden, wenn keine Wegpunkte gewünscht sind, sondern ein direkter Ort.

Tabelle 4.7: Fortsetzung spezifische Schlüsselwörter für MEET_AT_LOCATION

Schlüsselwort (Datentyp)	Beschreibung
`waypoints[i]` (vector)	Wegpunkte werden angegeben, indem der Index i immer hochgezählt wird, falls ein neuer Wegpunkt hinzukommt. Der erste Index muss immer mit 0 anfangen. Der letzte Wegpunkt wird automatisch als `targetLocation` angesehen.
`bHideMapMarker` (bool)	Gibt an, ob die Markierung des Ortes auf dem Kompass angezeigt werden sollen oder nicht. So könnte z.B. eine Mission den Zweck haben, dass der Spieler das Level erkundet, ohne mit dem Kompass dorthin geführt zu werden.

ELIMINATION

Das Ziel dieses Missionstyps ist es, einen Spieler aus einer anderen Mission oder mehrere NPCs zu betäuben.

Tabelle 4.8: spezifische Schlüsselwörter für ELIMINATION

Schlüsselwort (Datentyp)	Beschreibung
`charsToStunCount` (int)	Gibt an, wieviele Charaktere betäubt werden soll, damit die Mission erfolgreich abgeschlossen wird.
`targetChars` (string)	Es können 2 unterschiedliche Arten von Zielen angegeben werden. Sollen Spieler, welche eine Mission laufen haben, betäubt werden, dann muss `targetChars` der Wert „players“ zugewiesen werden. Sollen NPCs betäubt werden, dann muss der Wert „NPCs“ übergeben werden.

PLAY_ANIMATION

Eine Animation muss vom Spieler abgespielt werden, damit diese Mission erfolgreich ist. Dieser Missionstyp kann komplexer gestaltet werden, indem festgelegt wird, dass der Spieler die Animation in der Nähe eines Spielers ausführen soll.

Tabelle 4.9: spezifische Schlüsselwörter für PLAY_ANIMATION

Schlüsselwort (Datentyp)	Beschreibung
`toDoAnim` (string)	Animation, welche vom Spieler ausgeführt werden soll. Mögliche Animationen: „shrug“, „scratchHead“, „wave“, „clapHand“, „dance“, „listen“, „talk“ und „idle“.

Tabelle 4.9: Fortsetzung spezifische Schlüsselwörter für PLAY_ANIMATION

Schlüsselwort (Datentyp)	Beschreibung
`minDistance` (int)	Es kann eine Mindestentfernung zu einem Spieler angegeben werden, in der die Animation als erfolgreich ausgeführt gilt. Bei der Verwendung dieses Schlüsselwortes muss zwangsläufig die Variable `relevantCharFromMission` angegeben werden, damit das System weiß, zu welchem Spieler diese Entfernung gilt.

EMOTION

Ein Spieler muss eine bestimmte Emotion, über einen gewissen Zeitraum hinweg, zeigen. Für diesen Missionstyp ist zwingend eine Webcam erforderlich, welche die Emotionsdaten erfassen kann. Durch die Variablen im Schlüsselwort `targetEmotion` können die benötigten Voraussetzungen definiert werden.

Tabelle 4.10: Schlüsselwörter für targetEmotion

Schlüsselwort (Datentyp)	Beschreibung
`eType` (string)	Gibt an, welche Art von Emotion der Spieler zeigen soll. Mögliche Werte sind: „happy“, „angry“, „sad“ und „surprised“.
`minValue` (int(0-100))	Die Sträke der Emotionen werden von 0 bis 100 gemessen. 0 steht dabei für Emotion nicht erreicht und 100 für Emotion sehr gut erreicht. In diesem Schlüsselwort kann angegeben werden, wie stark die Emotion vom Spieler mindestens gezeigt werden soll.
`duration` (int)	Gibt die Dauer in Sekunden an, wie lange die Emotion aufrechterhalten werden soll.

USE_TECHNOLOGY

In diesem Missionstyp muss der Spieler eine bestimmte Fähigkeit einsetzten, um die Mission abzuschließen. Es können dabei Fähigkeiten angegeben werden, welche auf Personen gerichtet sind, z.B. das Blenden eines Spielers. Fähigkeiten, welche auf sich selbst anwendbar sind, z.B. das Verändern des Aussehens der Spielfigur, lassen sich ebenfalls als Mission realisieren.

Tabelle 4.11: spezifische Schlüsselwörter für USE_TECHNOLOGY

Schlüsselwort (Datentyp)	Beschreibung
techToUse (string)	Fähigkeit, welche eingesetzt werden soll. Mögliche Werte sind: „blind“, „allRun“, „allTwin“, „callDrone“, „selfDisguise“, „disguisePolice“, „emoScan“, „tagging“, und „beacon“. Soll eine Fähigkeit auf eine Person aus einer anderen Mission angewendet werden, dann muss wieder die Variable relevantCharFromMission angegeben werden. Die Beschreibung der Fähigkeiten kann in der Tabelle A.1 eingesehen werden.
scanningCount (int)	Falls als techToUse „emoScan“ ausgewählt wurde, kann durch dieses Schlüsselwort angegeben werden, wie oft der Spieler Personen scannen soll.

FOLLOWING

Eine Mission kann definiert werden, in welcher der Spieler einer anderen Person folgen muss. Dabei muss der Spieler in einem bestimmten Abstand zu der zu verfolgenden Person sein, wenn das Erfolgsereignis für die Mission eingetreten ist.

Tabelle 4.12: spezifische Schlüsselwörter für FOLLOWING

Schlüsselwort (Datentyp)	Beschreibung
minDistance (int)	Gibt die Mindestentfernung zu einem Spieler einer anderen Mission an. Bei diesem Missionstyp muss ebenfalls die Variable relevantCharFromMission angegeben werden. Somit erkennt das System, zu welchem Spieler der Abstand gehalten werden muss.
successAfterMissionEnd (int)	Neben der Mindestentfernung zur Person wird eine weitere Angabe benötigt, wann die Mission erfolgreich ist. Mit diesem Schlüsselwort kann angegeben werden, dass die Mission als erfolgreich gilt, wenn eine andere Mission abgeschlossen wurde. Damit lässt sich beispielsweise folgendes Szenario realisieren: Spieler A muss Spieler B solange beobachten, bis Spieler B seine Mission abgeschlossen hat.

HACKING

Das Hacken von Hotspots kann als Missionstyp vereinbart werden. Als Zielhotspot lassen sich sowohl ganz bestimmte Hotspots definieren oder auch beliebige.

Tabelle 4.13: spezifische Schlüsselwörter für HACKING

Schlüsselwort (Datentyp)	Beschreibung
`hotspotLocation` (vector)	Als Hackziel kann ein bestimmter Hotspot angegeben werden. Dabei muss die Position des Hotspots im Level angegeben werden.
`bEndlessHacking` (bool)	Wenn mehrere Personen den Hotspot hacken sollen, dann kann durch dieses Schlüsselwort festgelegt werden, dass der Hotspot nach dem Hacken nicht deaktiviert wird und somit immer gehackt werden kann.
`hackingCount` (int)	Gibt an, wie viele Hotspots gehackt werden sollen.

4.3 Verwaltung der Missionen in UnrealScript

Die komplette Steuerung der Missionen wird durch die Klasse `GC_Mission_Manager` realisiert. Diese enthält Funktionen, um die Missionen aus der Konfigurationsdatei auszulesen und auszuwerten. Funktionen zur Vergabe der Missionen an Spieler werden ebenfalls in dieser Klasse implementiert.

4.3.1 Anbindung des Missionssystems an den Server

Damit Funktionen des Missions-Managers aufgerufen werden können, muss eine Referenz auf ein Objekt der Klasse `GC_Mission_Manager` auf dem Server vorhanden sein. Dadurch wird also beispielsweise die Überprüfung von bestimmten Aktionen, welche vom Spieler durchgeführt wurden, ermöglicht. Auf diese Weise kann der Manager feststellen, ob die Aktion eine Relevanz zu den gerade laufenden Missionen hat.

In der Klasse `GC_GameInfo` werden sämtliche Objekte erzeugt, welche auf dem Server laufen sollen und auf verschiedene Weise die Spielwelt beeinflussen oder bestimmte Aktionen überwachen sollen. In einer Klasse `GC_GameInfo_Mission`, welche von `GC_GameInfo` erbt, muss auch ein Objekt der Klasse `GC_Mission_Manager` erzeugt werden. Dadurch hat beispielsweise der PlayerController Zugriff auf dieses Objekt und kann sämtliche Aktionen an den Missions-Manager weiterleiten.

Sobald ein Spieler in die Spielpartie einsteigt, wird er vom Missions-Manager erfasst und in ein Array geschrieben. Mit Hilfe dieses Array versucht der Missions-Manager herauszufinden, welche Missionen an welchen Spieler verteilt werden können. Aus diesem Grund muss der PlayerController des Spielers bestimmte Variablen besitzen, welche seinen aktuellen Zustand bezüglich von Missionen darstellen. So muss gespeichert werden, ob der Spieler bereits eine aktive Mission am Laufen hat und ob es sich um eine Singleplayer oder Multiplayer Mission handelt. Falls er bereits eine Mission am laufen

hat, muss ebenfalls die ID dieser Mission gespeichert werden.

4.3.2 Verteilung der Missionen

Alle Missionen, welche dynamisch an Spieler verteilt werden können und keine vorherige Mission als Voraussetzung haben, werden in ein eigenes Array `dynamicMissions` geschrieben. Anschließend wird ein Timer gesetzt, welcher in einem bestimmten Intervall versucht Missionen an Spieler zu verteilen.

Als erstes versucht der Manager zu prüfen, ob ein Missionsstrang gestartet werden kann. Dabei werden alle Spieler, welche sich beim Missions-Manager registriert haben, durchgesucht. Für jeden einzelnen Spieler wird geprüft, ob er die benötigten Voraussetzungen für den Start des Missionsstranges erfüllt. Bei dieser Überprüfung wird bestimmt, ob der Spielertyp der Person übereinstimmt oder ob der gewählte Charakter zur richtigen Fraktion gehört. Wie in Kapitel 4.1 beschrieben, besitzt der PlayerController sämtliche Informationen über den Spielertyp. Über eine Funktion werden nun sämtliche Schlüsselwörter [7] nacheinander überprüft. Sobald eine erste Unstimmigkeit zwischen dem benötigten Wert und dem vom Spieler gelieferten Wert auftritt, wird der Spieler als ungeeignet gekennzeichnet. Wenn sämtliche Voraussetzungen erfüllt sind, wird der Missionsstrang markiert, dass er gestartet werden kann. Die Überprüfung, ob ein Missionsstrang gestartet werden kann ist wichtig, damit auch wirklich eine Handlung aufgebaut werden kann. Dadurch wird gewährleistet, dass auch alle zum Missionsstrang gehörenden Missionen von den Spielern durchgeführt werden können.

Da der Missions-Manager weiß, welche Missionsstränge gestartet werden können, kann er gezielt Missionen an die Spieler verteilen. Der Manager versucht nun zu jeder Mission, welche zum bereiten Missionsstrang gehört, einen passenden Spieler zu finden. Es werden allerdings nicht alle Missionen durchgesucht, sondern nur die, welche in `dynamicMissions` enthalten sind. Hier wird ebenfalls nach dem Spielertyp verglichen, ähnlich wie es beim Vergleich in den Missionssträngen geschah. Eine Mission wird jedoch nicht direkt vergeben, wenn der Spielertyp übereinstimmt. Wie in Tabelle 4.3 genannt existiert die Variable `missionActive`, durch welche bestimmt wird, ob eine andere Mission aktiv ist. Die Überprüfung dieser Variable, bevor eine Mission gestartet werden kann, ist besonders wichtig, damit eine gewisse Ordnung/Reihenfolge unter den Missionen aufgebaut werden kann.

Bevor die Mission endgültig vergeben werden kann, muss noch bestimmt werden, ob es sich um eine Singleplayer oder Multiplayer Mission handelt. Dabei wird der PlayerController des Spielers markiert, dass er den bestimmten Typ der Mission hat. Singleplayer Missionen können wie bereits erwähnt pausiert werden, um eine Multiplayer Mission zu starten. Aus diesem Grund muss auch der PlayerController markiert werden. Dadurch

[7] siehe Tabelle 4.2

kann das System nämlich erkennen, ob die gerade laufende Mission des Spielers unterbrochen werden kann und er eine neue erhalten kann oder ob er bereits eine Multiplayer Mission durchführt.

Bei der Vergabe der Mission kommt diese in ein neues Array `currentMissions`. Alle Missionen, welche sich in diesem Array befinden, gelten als gestartet und werden nun vom Manager ständig überwacht. Dabei werden sämtliche Ereignisse, die mit diesen Missionen zu tun haben, überwacht. Beispielsweise wenn ein Spieler eine Person blendet, wird mit Hilfe dieses Array geprüft, ob die Aktion in irgendeiner Weise mit einer Mission in Zusammenhang steht.

Als nächstes wird geprüft ob der Autor bei der Definition der Mission angegeben hat, dass ein Charakter aus einer anderen Mission relevant ist. Wenn dies der Fall ist, wird nach dem PlayerController dieses Spielers gesucht und dieser wird in der gerade vergebenen Mission gespeichert. Dadurch wird es ermöglicht, sämtliche Informationen dieses Spielers abzurufen, z.B. seine Position im Level oder zu welcher Fraktion er gehört. Ebenfalls wird der PlayerController des Spielers der gerade vergebenen Mission, in diese gespeichert.

Sobald die Mission an den Spieler vergeben wurde, wird sie aus dem Array `dynamicMissions` entfernt, damit sie nicht noch einmal an einen Spieler vergeben werden kann. Wurde allerdings angegeben, dass die Mission mehrmals vergeben werden kann, dann wird lediglich die Variable `count`[8] dekrementiert.

4.3.3 Überwachung der Missionen

Wenn eine Mission gestartet wurde, wird sie ständig überwacht, ob irgendwelche Events eingetreten sind, die den Status der Mission ändern. Bei jedem Missionstyp werden jedoch unterschiedliche Funktionen und Variablen betrachtet, die diesen Status verändern könnten. Es gibt aber auch Ereignisse, welche unabhängig vom Missionstyp, immer überwacht werden. Der Manager iteriert dabei durch jedes Element in `currentMissions` und überprüft ob diese Ereignisse bei einer Mission auftreten.

allgemeine Ereignisse

Timer
Wenn der Autor bei der Definition von einer Mission einen Timer angegeben hat, dann hat der Spieler nur einen begrenzten Zeitraum, indem er die Mission erfolgreich abschließen kann. Sobald eine Mission an den Spieler vergeben wurde, wird in dieser die Zeit gespeichert, wann die Mission gestartet wurde. Dadurch kann der Manager bestim-

[8] siehe Tabelle 4.3

men, ob die angegebene Zeit für die Mission überschritten wurde. Falls dies der Fall ist, wird an den Spieler eine Nachricht gesendet, dass die Mission gescheitert ist.

optionale Nachrichten
Wie in Tabelle 4.5 beschrieben, existieren 3 Arten, wie optionale Nachrichten ausgelöst werden können. Wurde angegeben, dass der Spieler eine Nachricht bekommt, wenn er in der Nähe eines bestimmten Ortes ist, dann überprüft der Manager zyklisch die Entfernung zwischen dem Spieler und dem angegebenen Ort. Sobald diese Entfernung kleiner als die angegebene Mindestentfernung ist, wird die Nachricht an den Spieler gesendet.
Sobald ein Spieler geblendet wird, wird in der Pawn-Klasse eine Variable gesetzt, welche angibt, dass der Spieler geblendet wurde. Der Missions-Manager überprüft zyklisch ob diese Variable gesetzt ist. Wenn dies eintritt, wird dem Spieler die entsprechende Nachricht gesendet.
Soll eine Nachricht gesendet werden, wenn der Spieler in einem bestimmten Abstand zu einer anderen Person ist, dann wird zyklisch der Abstand zwischen dem Spieler der aktuellen Mission und der anderen Person ermittelt. Wenn der Abstand klein genug ist, wird die Nachricht verschickt.

Betäubung des Spielers
Wenn ein Spieler während einer Mission durch einen anderen betäubt wird, dann wird ebenfalls in der Pawn-Klasse eine Variable gesetzt. Falls dieses Ereignis eintritt, wird die aktuelle Mission des Spielers als gescheitert verbucht und die entsprechende Nachricht geschickt.

Missionsspezifische Ereignisse

MEET_AT_LOCATION
Bei diesem Missionstyp wird immer die Entfernung zwischen dem Spieler und dem Zielort berechnet, wenn der Abstand klein genug ist, wird die Mission als erfolgreich abgeschlossen. Hat der Autor eine Anzahl an Wegpunkten angegeben, dann werden diese Wegpunkte in ein Array gespeichert. Als temporärer Zielort wird immer das erste Element des Arrays betrachtet, mit dem Index 0. Wenn dieser Zielort erreicht wurde, wird das erste Element aus dem Array gelöscht und der nächste Wegpunkt befindet sich somit wieder am Index 0. Wenn im Array immer noch Elemente vorhanden sind, wird der nächste temporäre Zielort ermittelt. Dieser Vorgang wird solange wiederholt, bis sich keine Elemente mehr im Array befinden und somit der letzte Wegpunkt erreicht wurde. Am Anfang wird ebenfalls, wenn vom Autor erwünscht, eine Markierung auf dem Kompass angezeigt, mit Entfernung und Richtung des Zielortes.

ELIMINATION
Jede Mission vom Typ ELIMINATION besitzt eine Variable `bTargetCharStuned`, die an-

gibt ob die Zielperson erfolgreich betäubt wurde. Sobald ein Spieler eine Person betäubt, wird dies an den Missions-Manager weitergeleitet. Es wird überprüft ob der Schütze gerade eine Mission am laufen hat und ob er die richtige Person betäubt hat. Wenn dies der Fall ist wird die Variable gesetzt. Außerdem wird hochgezählt, dass der Spieler eine Person betäubt hat. Dies ist nötig, falls der Autor angegeben hat, dass mehrere Personen betäubt werden müssen. Der Manager überprüft ob die Anzahl der betäubten Personen kleiner als die Vorgabe ist, falls ja wird `bTargetCharStuned` wieder auf den Ausgangszustand gesetzt. Der Spieler muss nun solange Personen betäuben, bis die Anzahl übereinstimmt und die Mission als erfolgreich verbucht wird. Wenn nur NPCs betäubt werden müssen, dann ist keine weitere Bestimmung notwendig ob auch die richtige Person betäubt wurde.
Sobald eine Person aus einer anderen Mission als Ziel gilt, wird das Verfahren deutlich komplexer. Es kann immer nur eine Person dem Spieler als Ziel angezeigt werden, jedoch kann es vorkommen, dass mehrere Spieler als Ziel in Frage kommen. Aus diesem Grund werden alle Spieler, welche als Ziel in Frage kommen, in eine Art Puffer aufgenommen und nacheinander abgearbeitet. Mit Hilfe einer Funktion im Manager werden alle Personen in der Umgebung des Spielers durchgesucht. Dabei wird geprüft, ob diese eine Mission am laufen haben und deshalb als Ziel in die Auswahl kommen. Wenn das der Fall ist, wird dieser Spieler in das Array der Zielpersonen hinzugefügt. Die erste hinzugefügte Person wird dem Spieler der Mission als Ziel angezeigt. Es werden genau so viele Personen in das Array aufgenommen, wie die vom Autor angegebene Anzahl beschreibt. Wenn nun wieder eine Person betäubt wurde, wird in dem Array der Zielpersonen überprüft, ob die gerade betäubte zu den Zielpersonen gehört. Wenn dies der Fall ist, wird diese Person aus dem Array gelöscht und die nächste Person wird dem Spieler als Ziel angezeigt. Dieser Vorgang wird solange durchgeführt, bis sich keine Zielpersonen in dem Array mehr befinden.

PLAY_ANIMATION
Die Missionen vom Typ PLAY_ANIMATION bestitzen eine Variable `bAnimationDone`, welche angibt ob die benötigte Animation ausgeführt wurde. Der PlayerController sendet jedesmal, wenn eine Animation vom Spieler ausgeführt wurde, die Art der Animation und den Spieler an den Missions-Manager. Wenn beides mit der Mission übereinstimmen, wird `bAnimationDone` gesetzt und der Missions-Manager sendet eine Nachricht an den Spieler, dass die Mission erfolgreich ausgeführt wurde. Wenn allerdings die Animation im Umkreis eines bestimmten Spielers ausgeführt werden soll, wird `bAnimationDone` auf den Standardwert gesetzt. Beim nächsten Ausführen der Animation wird der Vorgang solange wiederholt, bis die Entfernung zum Spieler dem gewünschten Wert entspricht.

EMOTION
Die von der Webcam aufgezeichneten Emotionen werden an den Missions-Manager gesendet. Anschließend wird verglichen, ob die Emotion des Spielers den beschriebenen Mindestwert in der Mission erreicht. Ist dieser Mindestwert erreicht, fängt der Manager

an eine Variable hochzuzählen, welche die Zeit repräsentiert. Wenn diese Zeit gleich der in der Mission beschriebenen Mindestzeit ist, gilt die Mission als erfolgreich. Falls jedoch die Emotion während der Zeiterfassung nicht mehr den Mindestwert erreicht, wird die Zeit gestoppt und fängt wieder bei 0 an.

USE_TECHNOLOGY
Der PlayerController des Spielers sendet die Art der benutzen Fähigkeit, sowie gegebenenfalls die Person, welche von der Fähigkeit getroffen wurde, an den Missions-Manager. Es wird wieder überprüft, ob eine Mission existiert, welche das Ausführen dieser Fähigkeit zum Erfolg benötigt. Wenn das der Fall ist, wird eine Nachricht mit dem Erfolg an den Spieler gesendet.

FOLLOWING
Zunächst wird eine Markierung auf dem Kompass erstellt, damit der Spieler weiß, welcher Person er zu folgen hat. Wie in Tabelle 4.12 angegeben, muss der Autor festlegen, wann die aktuelle Mission als abgeschlossen gilt. Dafür wird die Variable `successAfterMissionEnd` angegeben. Aus diesem Grund überprüft der Missions-Manager, ob die genannte Mission bereits abgeschlossen wurde. Erst wenn die Mission abgeschlossen wurde, wird überprüft, ob der Spieler sich mindestens in dem angegebenen Abstand zur Person befindet. Wenn der Mindestabstand zu dem Zeitpunkt eingehalten wurde, wird eine Erfolgsnachricht gesendet, andernfalls gilt die Misssion als gescheitert.

HACKING
Der Zustand eines Hotspot zu Beginn einer Spielpartie basiert nach dem Zufall. Das heißt, ein Hotspot welcher in der einen Partie noch zu Beginn aktiviert war, könnte in der nächsten Partie deaktiviert sein. Das spielt keine Rolle, wenn der Spieler die Mission erhält einen beliebigen Hotspot zu hacken. Wenn jedoch ein Hotspot an einer bestimmten Stelle gehackt werden soll, dann muss sicherheitshalber der Mission-Manager prüfen, ob der in der Mission beschriebene Hotspot überhaupt gehackt werden kann. Sollte das nicht der Fall sein, dann wird der Hotspot automatisch aktiviert, wenn der Spieler die Mission erhält.
Sobald der Hotspot gehackt wurde, sendet er dieses Ereignis an den Missions-Manager. Dieser überprüft ob der Hacker des Hotspots gerade eine Mission laufen hat, welche als Ziel das Hacken eines Hotspots voraussetzt. Wenn das der Fall ist, wird in der Mission eine Variable gesetzt, dass der Hotspot erfolgreich gehackt wurde. Sollen mehrere Hotspots gehackt werden, dann wird zuerst verglichen, ob die Anzahl der gehackten Hotspots mit der Anzahl, welche in der Mission verlangt wird, übereinstimmen. Soll ein und der selbe Hotspot mehrmals gehackt werden, dann wird dieser nach dem Hacken wieder aktiviert. Damit ist er auch für andere Spieler wieder für das Hacken offen.

4.3.4 Abschluss einer Mission

Nachdem eine Mission abgeschlossen wurde, überprüft der Missions-Manager, ob eine Animation ausgeführt oder eine Belohnung vergeben werden muss. Ebenfalls werden sämtliche Markierungen auf dem Kompass wieder versteckt. Nachdem die Erfolgs/Misserfolgs Nachricht an den Spieler gesendet wurde, überprüft der Manager, ob eine weitere Mission an der gerade abgeschlossenen hängt. D.h. ob der Autor die Schlüsselwörter `nextMissionAfterSuccess` bzw. `nextMissionAfterFail` definiert hat. Wenn das der Fall ist, wird überprüft, ob die Mission gescheitert ist oder nicht.

Es wird also nach der nächsten Mission gesucht und diese ersetzt die gerade abgeschlossene Mission. Dabei wird der PlayerController aus der abgeschlossenen Mission in die neue Mission übertragen. Dadurch wird sichergestellt, dass auch der richtige Spieler die entsprechende Mission erhält. Variablen wie der Timer oder ein relevanter Charakter aus einer anderen Mission müssen ebenfalls der neuen Mission zugewiesen werden.

Der Manager überprüft außerdem, ob eine Singleplayer Mission wieder aufgenommen werden kann, welche vorher durch die Multiplayer Mission pausiert worden ist. Sobald eine Singleplayer Mission pausiert wird, wird eine Variable `bPaused` gesetzt, welche diesen Zustand der Mission repräsentiert. Es wird geprüft, ob der Spieler aus der gerade abgeschlossenen Mission eine pausierte Mission besitzt. Falls das zutrifft, wird die Mission wieder aktiviert und kann fortgesetzt werden. Außerdem werden Markierungen auf dem Kompass, welche aufgrund der neuen Multiplayer Mission ausgeblendet werden mussten, wieder eingeblendet.

4.3.5 Fehlerrobustheit

Während dem Ablauf der Missionen kann es zu Ereignissen kommen, welche nicht vorhersehbar sind. Deswegen muss das Missionssystem gegen solche Ereignisse geschützt sein, damit es dem Spieler weiterhin möglich ist seine Missionen zu erfüllen. Das in dieser Arbeit beschriebene System, bietet einige Absicherungen, jedoch kann es trotzdem zu Fällen kommen, in denen die Missionen nicht mehr zu erfüllen sind, weil bestimmte Voraussetzungen fehlen.

Wenn z.B. der Spieler einen bestimmten Hotspot hacken soll, kann der Autor durch das Setzen der Variable `bEndlessHacking` dafür sorgen, dass der Hotspot durch mehrere Spieler gehackt werden kann. So kann es nicht vorkommen, dass der Hotspot nicht mehr gehackt werden kann, weil ein anderer Spieler diesen schon vorher gehackt hat.

Falls ein Spieler während dem Lauf einer Mission betäubt wird, gilt die Mission als Fehlschlag und wird abgeschlossen. Dadurch wird immer sichergestellt, dass eine Mission

abgeschlossen wird. Angenommen die Mission von Spieler B kann erst abgeschlossen werden, wenn Spieler A seine Mission abgeschlossen hat. Spieler A wird betäubt und kann somit theoretisch seine Mission nicht mehr erfüllen. Da aber seine Mission automatisch als Fehlschlag verbucht und abgeschlossen wird, kann Spieler B seine Mission trotzdem abschließen.

Das Missionssystem bietet allerdings keine Sicherheit, wenn ein Spieler während einer Mission das Spiel verlässt. Bauen noch weitere Mission auf der nicht abgeschlossenen Mission auf, dann kann es zu einer Sackgasse für die ganzen Spieler des Missionsstrangs kommen. Dadurch ist es nicht mehr möglich den Missionsstrang weiter abzuarbeiten, da eine Teilkomponente (der Spieler) fehlt.

4.4 Implementierung der Flash-Oberfläche

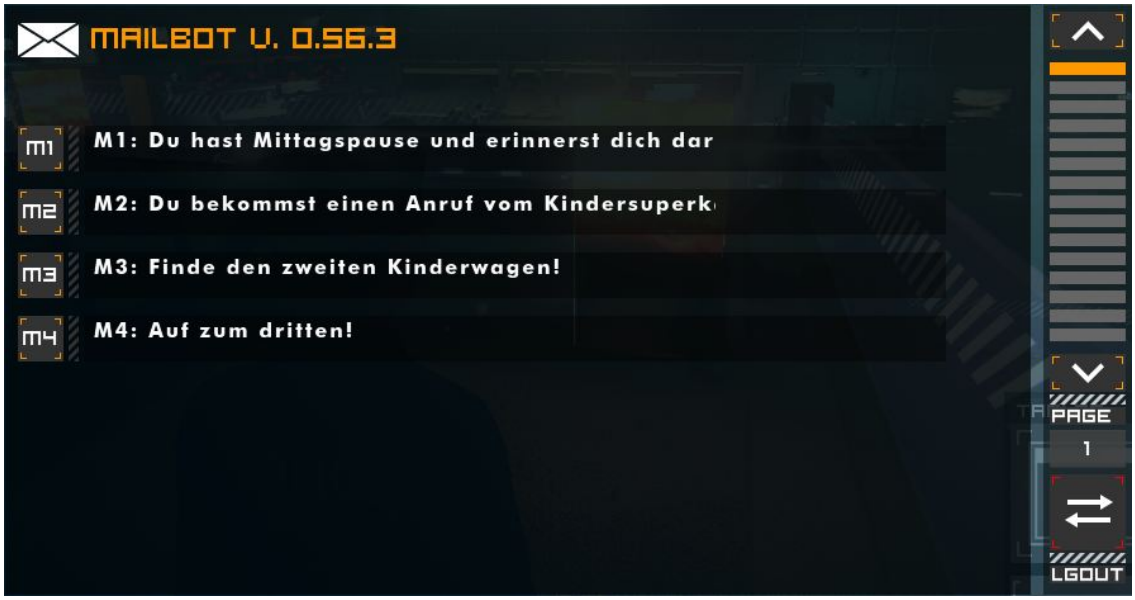
Es gibt 4 unterschiedliche Nachrichtentypen, die durch den Server an die GUI des Spielers geschickt werden können. Dadurch kann im ActionScript unterschieden werden, welche Art von Nachricht in die Missionen eingetragen werden müssen, z.B. eine Nachricht über den Erfolg der Mission oder Misserfolg. Dies dient lediglich der besseren Übersicht für den Spieler, so werden unterschiedliche Nachrichtentypen mit einer anderen Farbe dargestellt.

Tabelle 4.14: Nachrichtentypen für ActionScript

Nachrichtentyp	**Beschreibung**
`start`	Sobald dieser Typ von Nachricht im ActionScript ankommt, wird eine neue Mission angelegt. Es handelt sich dabei um den Ausgangstext der Mission.
`info`	Gibt an, dass es sich lediglich um eine Information in der Mission handelt, z.B. wenn ein besonderer Ort entdeckt wurde.
`success`	Der Text wird als erfolgreich markiert.
`fail`	Der Text wird als gescheitert markiert.

Sobald die Mission durch den Nachrichtentyp `start` erstellt wurde, wird automatisch zu der Mission ein Knopf in der Oberfläche erstellt, welcher die Mission ein und ausklappen kann. Dadurch wird sowohl ein Überblick über alle Missionen gewährleistet, als auch eine detailliertere Ansicht zu jeder einzelnen Mission.

Abbildung 4.3: Übersicht über alle Missionen



Wenn der Spieler eine Mission erhält, wird in seinem PlayerController eine ID gespeichert, welche zur eindeutigen Bestimmung der Mission in Flash dient. Falls beispielsweise eine Information zu einer Mission kommt, kann durch die ID bestimmt werden, zu welcher Mission dieser Text hinzugefügt werden soll. Es findet außerdem eine Trennung statt zwischen der ID für Singleplayer Missionen und der ID für Multiplayer Missionen. Würde z.B. der Spieler eine Singleplayer Mission mit der ID 2 laufen haben und bekommt anschließend eine neue Multiplayer Mission, dann bekommt diese die ID 3. Nun muss aber der Missions-Manager wissen, bei welcher Mission er fortlaufen muss, wenn die Multiplayer Mission abgeschlossen wurde. Durch diese Trennung wird dies auch ermöglicht.

Während der Übersicht über alle Missionen, wird lediglich ein Ausschnitt des Starttextes angezeigt. Infotexte der Mission und Erfolgs/Misserfolgstexte sind dagegen komplett unsichtbar. Sobald der Spieler den Knopf für die detaillierte Ansicht der Mission auswählt, wird die ID der Mission erfasst und sämtliche Textfelder, welche nicht der ID entsprechen, ausgeblendet. Danach werden alle zur Mission gehörenden Texte angezeigt. Um die Details einer anderen Mission anzusehen, reicht es den entsprechenden Knopf zu drücken. Die aktuelle Detailansicht wird dabei mit der neuen Mission ausgetauscht.

Da nach einer bestimmten Anzahl an Missionen kein Platz mehr in der Übersicht vorhanden ist, werden die weiteren Missionen auf einer neuen Seite angelegt. Der Spieler kann dabei zwischen den Seiten wechseln, indem der Knopf in der rechten Ecke gedrückt wird. Außerdem wird die Seite automatisch umgeblättert, sobald eine neue Mission auf eine neue Seite kommt.

Abbildung 4.4: Detailansicht einer Mission

5 Test Missionsstrang

Im Folgenden wird ein Missionsstrang demonstriert. Dieser basiert auf Multiplayer Missionen und soll eine Handlung zwischen mehreren Spielern aufbauen. Die Missionen sollen auch an Spieler eines bestimmten Typs verteilt werden. Sobald ein Spieler sich in das Spiel einloggt, soll er eine Singleplayer Mission erhalten, diese soll er so lange ausführen, bis die benötigte Anzahl an Spielern sich im Level befinden. Erst danach soll die Singleplayer Mission pausiert werden und die Multiplayer Mission gestartet werden.

5.1 Handlung des Missionsstrangs

Die Rebellen wollen versuchen die Polizei bloßzustellen, indem sie einen Polizisten solange provozieren, bis dieser die Provokateure mit seinem Taser betäubt. Der Polizist bekommt dabei Nachrichten, dass er sich um einige unruhestiftende Rebellen kümmern muss. Wenn der Polizist dieser Aufgabe nachgeht, muss ein Rebell den Polizisten verfolgen und die ganze Angelegenheit mitfilmen. Dadurch wollen die Rebellen zeigen, wie wahllos und aggressiv die Polizei vorgeht und somit Sympathie in der Bevölkerung gewinnen.

5.2 Aufbau der Missionen

Als erstes müssen die Werte für die Spielertypen definiert werden, z.B. ab welcher Anzahl von Schüssen auf eine Person, der Spieler als „shooter“ gilt. Da für diese Mission nur die beiden Spielertypen „hacker“ und „shooter“ relevant sind, reicht es aus nur diese anzugeben.

Listing 5.1: Definition der Spielertypen

```
SHOOTER_AT=10
HACKER_AT=5
```

Zu aller erst werden Singleplayer Missionen für die Spieler definiert, falls nicht genug andere Spieler im Level vorhanden sind, um die Multiplayer Mission zu starten. Dazu muss in der Konfigurationsdatei der Missionsstrang definiert werden.

Listing 5.2: Missionsstränge für Singleplayer Missionen

```
MissionThreads=(id=2,
                playerType[0]=(playerFaction=FACTION_Police),
```

```
                bCanBeAborted=true
               )

MissionThreads=(id=3,
                playerType[0]=(playerFaction=FACTION_Rebels),
                bCanBeAborted=true
               )
```

Da in der Mission Spieler von der Fraktion Polizei und Rebellen benötigt werden, müssen 2 Missionsstränge für die Singleplayer Missionen angegeben werden.

Die Rebellen bekommen als Singleplayer Mission die Aufgabe, 2 beliebige Hotspots zu hacken.

Listing 5.3: Singleplayer Mission für Rebellen

```
Missions=(thread=3,
          id=9,
          bHasPrevMission=false,
          bIsDynamic=true,
          count=10,
          conditions=(playerFaction=FACTION_Rebels),
          startText="Hacke 2 beliebige Hotspots",
          missionType="HACKING",
          hackingCount=2,
          successText="Du hast alle benötigten Hotspots gehackt.",
         )
```

Es werden keine speziellen Voraussetzungen für die Singleplayer Missionen, wie beispielsweise hacker=80 festgelegt. Denn es sollen alle Spieler im Level eine Singleplayer Mission erhalten dürfen, unabhängig ob diese ein bestimmter Spielertyp sind. Wenn eine große Menge an Missionen geschrieben wird, dann ist es sinnvoll diese nach dem Spielertyp einzugrenzen.

Der Polizist erhält die Mission, 10 Personen mit dem Emotionsscanner zu erfassen.

Listing 5.4: Singleplayer Mission für Polizisten

```
Missions=(thread=2,
          id=8,
          bHasPrevMission=false,
          bIsDynamic=true,
          count=10,
          conditions=(playerFaction=FACTION_Police),
          startText="Scanne 10 Personen mit deinem Emotionsscanner.",
          missionType="USE_TECHNOLOGY",
          techToUse="emoScan",
          scanningCount=10,
          successText="Du hast erfolgreich die 10 Personen gescannt.",
         )
```

Die Spieler können nun Singleplayer Missionen erhalten. Jetzt muss ein Multiplayer Missionsstrang vereinbart werden. Da für den Missionsstrang 4 Spieler benötigt werden, muss dies auch in der Definition angegeben. Außerdem müssen die Spieler einem bestimmten Typen entsprechen. So muss z.B. der Polizist den Wert shooter mindestens zu 50% erfüllen.

Listing 5.5: Missionsstrang für die Multiplayer Mission

```
MissionThreads=(id=1,
                playerType[0]=(playerFaction=FACTION_Police,
                               shooter=50),
                playerType[1]=(playerFaction=FACTION_Rebels,
                               hacker=70),
                playerType[2]=(playerFaction=FACTION_Rebels,
                               hacker=70),
                playerType[3]=(playerFaction=FACTION_Rebels,
                               hacker=70),
               )
```

Sobald alle Spieler im Level vorhanden sind, um den Missionsstrang zu starten, werden die zum Missionsstrang gehörenden Missionen an die Spieler verteilt. Der Polizist und die Rebellen erhalten dabei gleichzeitig eine Mission. Die Aufgabe des Polizisten ist es, seinen Taser bei einer Straßensperre abzuholen.

Listing 5.6: Multiplayer Mission MEET_AT_LOCATION des Polizisten

```
Missions=(thread=1,
          id=2,
          bHasPrevMission=false,
          bIsDynamic=true,
          conditions=(playerFaction=FACTION_Police,
                      shooter=50),
          startText="Du bist derzeit unbewaffnet, hol dir den Taser
                     von der Stra\ss ensperre.",
          missionType="MEET_AT_LOCATION",
          targetLocation=(X=-619,Y=1089,Z=67),
          successText="Sehr gut, du hast nun die Waffe.",
          reward=(rewardType="pistol",bInstantStun=true,ammoCount=6),
          timer=20,
          failText="Beeil dich und hol die Waffe",
          nextMissionAfterFail=2,
          nextMission=5
          )
```

Der Polizist erhält einen Taser, welcher eine Munitionskapazität von 6 Schuss hat und eine Person mit einem Schuss betäuben kann. Das hat den Zweck, dass der Spieler nicht wahllos Personen betäubt, sondern sich das genau überlegt und sich sicher ist, dass er auch die richtige Person betäubt. Außerdem hat die Mission eine Timer von 20 Sekunden, indem sie ausgeführt werden soll. Da aber dem Schlüsselwort `nextMissionAfterFail` die selbe ID zugewiesen wurde, die die Mission hat, gilt der Timer nicht als

Fehlschlag der Mission, sondern als Erinnerung an die Mission.

Während der Polizist seine Waffe holt, bekommen die Rebellen gleichzeitig die Mission, einen Hotspot an einem bestimmten Ort zu hacken. Der Ort des Hotspots wird dabei auf dem Kompass der Spieler angezeigt.

Listing 5.7: Hacking Mission für die Rebellen

```
Missions=(thread=1,
          id=1,
          count=3,
          bHasPrevMission=false,
          bIsDynamic=true,
          conditions=(playerFaction=FACTION_Rebels,
                      hacker=70),
          startText="Hacke den auf deinem Kompass markierten Hotspot",
          missionType="HACKING",
          hotspotLocation=(X=-284,Y=-469,Z=40),
          successText="In der Stadt soll sich ein emotional instabiler
                       Polizist befinden, den nutzen wird für unser
                       nächstes Projekt. Warte ab, bis weitere
                       Informationen über ihn bekannt sind.",
          bEndlessHacking=true
          )
```

Die Mission wird 3 mal verteilt, weil 3 Rebellen an ihr beteiligt sein sollen. Außerdem wird das Schlüsselwort `bEndlessHacking` gesetzt, weil die 3 Rebellen alle den selben Hotspot hacken sollen. Somit wird verhindert, dass nur der schnellste Hacker den Hotspot hacken kann.

Wie in Listing 5.6 zu sehen ist, besitzt die Mission eine Folgemission `nextMission=5`. Sobald also der Polizist seine Waffe geholt hat, bekommt er gleich danach eine neue Mission.

Listing 5.8: Multiplayer Mission ELIMINATION des Polizisten

```
Missions=(thread=1,
          id=1,
          count=3,
          bHasPrevMission=true,
          startText="In der Nähe sollen sich Rebellen befinden, die
                     Unruhe stiften. Halte Ausschau nach diesen.
                     Vergiss nicht, du hast nur 6 Schuss.",
          relevantCharFromMission=4,
          missionType="ELIMINATION",
          charsToStunCount=2,
          targetChars="players",
          optMsg[0]=(msgCond=(bPlayerBlinded=true),msgType="info",
                     msg="Lass dir das nicht gefallen, betäube ihn!"),
          successText="Sehr gut, du hast dich um die Unruhestifter
                       gekümmert."
```

```
        )
```

Der Polizist erhält also die Aufgabe 2 Personen zu betäuben. Die Zielpersonen sollen dabei nur Spieler sein und keine NPCs. Sobald der Polizist diese Mission erhält, erhalten auch die Rebellen eine neue Missionen. Das Schlüsselwort `relevantCharFromMission=4` wird benutzt, damit dem Polizisten ein Hinweis auf seinem Kompass gegeben wird, wo sich ein Rebell aufhält. Dadurch wird verhindert, dass der Polizist durch die Stadt umherirrt und keine Rebellen findet. Dem Polizisten wird außerdem eine Nachricht angezeigt, falls dieser geblendet wird. Wenn der Polizist sich einem Rebellen nähert, wird ab einer bestimmten Entfernung dem Polizisten der Rebell als Ziel angezeigt. Sollte der Polizist mit seinen 6 Schuss nicht treffen oder diese anderweitig verschossen haben, dann gilt seine Mission als gescheitert.

Wenn die 3 Rebellen ihre Hacking Mission abgeschlossen haben und der Polizist seine ELIMINATION Mission erhalten hat, bekommen die 3 Rebellen jeweils eine neue Mission. Der erste Rebell bekommt die Aufgabe, den Polizisten zu blenden und so zu provozieren, damit dieser den Rebellen betäubt.

Listing 5.9: Blenden Mission des 1. Rebellen

```
Missions=(thread=1,
          id=4,
          bHasPrevMission=false,
          bIsDynamic=true,
          conditions=(playerFaction=FACTION_Rebels, missionActive=5),
          startText="Da ist unser Polizist, blende ihn und lass
                     dich danach betäuben, während ein anderer das
                     Ganze filmt.",
          relevantCharFromMission=5,
          showPicFromPlayer=5,
          missionType="USE_TECHNOLOGY",
          techToUse="blind",
          successText="Super, dass war fast filmreif.",
          failText="Naja hoffentlich hat dein Kollege wenigstens das
                    gefilmt.",
          )
```

Falls der Rebell betäubt wird, bevor er den Polizisten blenden konnte, wird der `failText` übermittelt.

Der 2. Rebell bekommt ebenfalls eine Mission, in der er den Polizisten provozieren soll. Dies soll aber nicht durch das Einsetzen einer Fähigkeit, sondern einfach durch sein Verhalten geschehen. Dafür muss der Spieler in der Nähe des Polizisten die Animation „dance“ durchführen.

Listing 5.10: PLAY_ANIMTAION Mission des 2. Rebellen

```
Missions=(thread=1,
```

```
          id=6,
          bHasPrevMission=false,
          bIsDynamic=true,
          conditions=(playerFaction=FACTION_Rebels, missionActive=5),
          startText="Da ist unser Polizist, laufe zu ihm hin und tanze
                     vor ihm, anschlie\ss end lässt du dich betäuben.
                     Diese Respektlosigkeit wird ihn bestimmt
                     aufregen.",
          relevantCharFromMission=5,
          showPicFromPlayer=5,
          missionType="PLAY_ANIMATION",
          toDoAnim="dance",
          minDistance=10,
          successText="Das war gut, du solltest Tänzer werden.",
          failText="Naja hoffentlich hat dein Kollege wenigstens das
                    gefilmt.",
          )
```

Auch hier gilt die Mission gescheitert, wenn der Rebell betäubt wird, bevor er getanzt hat.

Der letzte Rebell bekommt die Mission, den Polizisten zu verfolgen und so sein Verhalten zu filmen. Das Filmen ist nur symbolischer Natur, der Spieler muss sich lediglich in der Nähe des Polizisten befinden, damit seine Mission als erfolgreich verbucht wird.

Listing 5.11: FOLLOWING Mission des 3. Rebellen

```
Missions=(thread=1,
          id=7,
          bHasPrevMission=false,
          bIsDynamic=true,
          conditions=(playerFaction=FACTION_Rebels, missionActive=5),
          startText="Deine Kollegen werden einen Polizisten
                     provozieren. Du sollst das ganze Geschehen filmen
                     und so den Polizisten später blo\ss stellen.",
          relevantCharFromMission=5,
          showPicFromPlayer=5,
          missionType="FOLLOWING",
          minDistance=10,
          successAfterMissionEnd=5,
          bInvisOnCompass=true,
          successText="Das war gut, du solltest mal einen Film
                       drehen.",
          failText="Danke, du hast die komplette Aktion in den Sand
                    gesetzt.",
          )
```

Es ist wichtig, dass der Spieler dieser Mission dem Polizisten nicht als Ziel angezeigt wird, wenn er sich in der Nähe des Polizisten befindet. Sonst wäre die Mission nicht durchführbar. Somit wird mit dem Schlüsselwort `bInvisOnCompass` festgesetzt, dass

der Rebell dem Polizisten nicht als Ziel angezeigt wird. Es muss außerdem für den Rebell ein Ereignis geben, ab wann seine Mission als erfolgreich gelten kann. In dieser Verfolgungsmission wird die Mission als erfolgreich verbucht, wenn der Rebell sich mit einem Mindestabstand von 10 Metern zum Polizisten befindet und der Polizist zum selben Zeitpunkt seine ELIMINATION Mission abgeschlossen hat. Das heißt wenn der Polizist den 2. Rebellen betäubt hat, ist seine Mission erfolgreich. Als Hilfestellung für die Entfernung zwischen dem Polizisten und dem Verfolger, wird dem Verfolger ein Indikator auf dem Kompass angezeigt. Dieser stellt sowohl die Richtung des Polizisten, als auch die Entfernung dar. Sollte der Spieler der Verfolgungsmission sich weiter als 10 Meter beim Abschluss der Mission des Polizisten befinden, wird die Mission als gescheitert markiert.

Mit der in diesem Kapitel beschriebenen Missionen lässt sich also eine Handlung zwischen mehreren Spielern aufbauen. Es lassen sich sowohl Szenarien erstellen, in denen die Spieler miteinander oder gegeneinander interagieren. Die Missionen können dabei, je nach Spielertyp, dynamisch zugewiesen werden.

Dieser Missionsstrang wurde mit 4 Personen aus der Forschungsgruppe GAMECAST getestet. Alles verlief nach Plan, außer dass der Spieler des Polizisten seine 6 Schuss vorzeitig verschossen hatte und somit seine Mission fehlgeschlagen ist.

6 Zusammenfassung

6.1 Resultat der Arbeit

Diese Arbeit zeigt am Beispiel eines Missionssystems für den Spielprototyp Urban Legend, wie ein adaptives Missionssystem prototypisch realisiert werden kann. Dabei wurde gezeigt, wie Missionen an einen bestimmten Spielertyp verteilt werden können. Es wurden Spielerdaten erfasst und in eine eigenständige Datei gespeichert. Somit wurde gewährleistet, dass sich der Spieler ständig in der Entwicklung befindet. Aus diesen erfassten Daten konnte ein Spielertyp gebildet werden, welcher vom Missionssystem benutzt werden konnte. Es wurde ebenfalls eine eigene Skriptsprache entwickelt, um Autoren ohne Programmierkenntnisse eine Möglichkeit zu bieten, Missionen ohne großen Aufwand zu erstellen und ohne der Hilfe von Programmierern in das Spiel zu integrieren. Ebenfalls wurde ein Oberfläche für die Missionen entwickelt, mit welcher der Spieler eine Übersicht über seine laufenden Missionen hat.

6.2 Ausblick

Das in dieser Arbeit beschriebene System stellt lediglich einen Prototypen dar. Damit der adaptive Aspekt des Systems besser funktioniert, muss ein viel genaueres Profil über den Spieler erstellt werden. So wäre es sinnvoll die aufgezeichneten Aktionen über einen bestimmten Zeitraum zu betrachten, z.B. nicht nur wie viele Personen der Spieler betäubt hat, sondern wie viele pro Partie oder pro Stunde. Damit ließe sich ermitteln, ob der Spieler die Personen betäuben musste, oder ob er das macht, weil es ihm Freude bereitet. Damit ein Spieler beispielsweise wirklich als Hacker dargestellt werden kann, müsste z.B. ermittelt werden, wie schnell er einen Hotspot hackt. Dadurch könnte man schlussfolgern, dass er dies besonders gut kann. So wäre es denkbar das Missionssystem so zu gestalten, dass es den Schwierigkeitsgrad an die Geschwindigkeit des Hackers anpasst. Dadurch würde der Spieler nicht gelangweilt werden und immer eine Herausforderung haben oder nicht frustriert werden, weil es zu schwer ist.

Die Spielerdaten befinden sich noch lokal auf dem PC des Spielers und könnten durch diesen manipuliert werden. Deshalb wäre es wichtig, diese Daten auszulagern und vom Spieler fernzuhalten. Am besten würde sich eine Datenbank mit einem Login-System anbieten. So würde zu jedem Spieler, extern auf einem Server, sein komplettes Profil gespeichert werden können.

Aktuell werden die Missionen in einer Konfigurationsdatei erstellt. Auf längere Sicht wäre es besser, ein anderes Format zu verwenden, welches eine bessere Übersicht bietet. Wie in Kapitel 3.1 beschrieben, würde sich hierfür die Verwendung von XML-Dateien

eignen.

Es müssten ebenfalls noch Wege gefunden werden, wie man die Synchronisation unter den Missionen besser durchsetzen könnte. Vorallem für ein Multiplayer Spiel ist es besonders wichtig, dass die Verknüpfungen zwischen den Missionen abgesichert sind. So darf es unter keinen Umständen vorkommen, dass ein Spieler seine Mission nicht abschließen kann, weil z.B. ein anderer Spieler vom Netzwerk getrennt wurde. Damit würde der komplette Missionsstrang zusammenbrechen und kein Spieler könnte die Mission abschließen. Ein Lösungsansatz dafür wäre, jedem Spieler eine Singleplayer Mission zu geben und die Multiplayer Mission anzuhalten, ggf. solange, bis der getrennte Spieler sich wieder verbindet.

Im aktuellen System, muss ein Missionsstrang während einer Spielpartie abgeschlossen werden. Damit wäre es nicht möglich sehr lange Missionsstränge zu implementieren, da man nicht voraussetzen kann, dass alle Spieler genügend Zeit haben, den kompletten Missionsstrang abzuarbeiten. Deswegen wäre es notwendig, für jeden Spieler den aktuellen Stand im Missionsstrang zu speichern, sobald der Spieler das Spiel verlässt. Beim nächsten einloggen in das Spiel müsste der Spieler sich wieder am selben Standpunkt im Missionsstrang befinden, um so die Handlung weiter anzutreiben. Das wäre für eine Singleplayer Mission auch ohne größere Probleme machbar, jedoch stellt es für eine Multiplayer Mission eine große Herausforderung dar. So müsste neben dem Spiele-Server ein weiterer Server implementiert werden, welche alle Spiele-Server überwacht und den richtigen Server für den Spieler aussucht. Dabei muss er beachten, in welchem Missionsstrang sich der Spieler befindet und an welcher Stelle er in diesem Strang gerade ist. Dann müsste er weitere Spieler finden, welche sich ebenfalls an der selben Stelle im Strang befinden, damit die Handlung fortgesetzt werden kann.

Anhang A: Einsetzbare Fähigkeiten für Missionen

Tabelle A.1: Beschreibung der Fähigkeiten

Schlüsselwort	**Beschreibung**
`blind`	Ein Spieler wird geblendet, dabei wird der Bildschirm des Spielers in einer weißen Farbe dargestellt.
`allRun`	Alle NPCs um den Spieler herum fangen an wegzulaufen.
`allTwin`	Alle NPCs um den Spieler herum werden verkleidet und sehen aus wie der Spieler.
`callDrone`	Eine Drohne wird gerufen und scannt die Umgebung nach Spielern, wenn der Scanvorgang erfolgreich war, wird der Spieler markiert. Dabei wird ein rotes Viereck um den Kopf des Spielers gezeichnet.
`selfDisguise`	Der Spieler verkleidet sich und nimmt so ein anderes Aussehen ein.
`disguisePolice`	Der Spieler kann eine Polizisten verkleiden lassen, sodass dieser das Aussehen eines normalen Bürgers einnimmt. Der Polizist selbst, sieht nicht das er verkleidet ist, nur andere Polizisten sehen ihn als normalen Bürger.
`emoScan`	Ein Polizist kann die Emotionen eines Spielers scannen.
`tagging`	Ein anderer Spieler kann markiert werden, es wird ein rotes Viereck um dessen Kopf gezeichnet.
`beacon`	Eine Markierung kann auf der Karte gesetzt werden, somit können der selben Fraktion erkennen, wo sich gerade etwas besonderes abspielt.

Anhang B: Inhalt der mitgelieferten CD-ROM

- die vorliegende Arbeit als PDF-Datei
- Quellcode der Klasse GC_GameInfo_Mission.uc
- Quellcode der Klasse GC_Mission_Manager.uc
- Quellcode der Klasse GC_PlayerInfoForMissionsManager.uc
- Quellcode der Klasse GC_PlayerController.uc
- Beispielmissionen DefaultMissions.ini
- Besipiel Spielertyp UDKLoggedPlayerInfo.ini

Der Quellcode der Klassen dient zur Veranschaulichung der wichtigsten Funktionen für das Missionssystem. Alle Klassen von Urban Legend werden jedoch nicht mit aufgeführt.

Anhang C: Verwendete Tools

Folgende Tools wurden im Rahmen dieser Arbeit verwendet:

- Microsoft Visual Studio 2010 Ultimate
- Pixel Mine nFringe Plugin für Visual Studio - Syntax Highlightning und Code-Vervollständigung für UnrealScript
- UDK(Unreal Development Kit) von Epic Games in der November 2012 Version
- Adobe Flash Professional CS5

Literaturverzeichnis

[1] Gamecast Website
URL:http://www.gamecast-tv.com (30.08.2013)

[2] Epic Games: Übersicht über UnrealScript
URL:http://udn.epicgames.com/Three/UnrealScriptReference.html
(30.08.2013)

[3] Cordone, Rachel: Unreal Development Kit Game Programming with UnrealScript: Beginner's Guide, Packt Publishing 2011

[4] UnrealScript-QuellCode von Urban Legend

[5] Epic Games: Übersicht über Client-Server-Verhalten in Unreal
URL:http://udn.epicgames.com/Three/ReplicationHome.html (30.08.2013)

[6] Epic Games: Grundlagen zur Unreal Engine 3
URL:http://udn.epicgames.com/Three/UE3Basics.html (30.08.2013)

[7] Epic Games: Übersicht über Konfigurationsdateien
URL:http://udn.epicgames.com/Three/ConfigurationFiles.html
(30.08.2013)

[8] Weiterführende Informationen über Konfigurationsdateien
URL:http://romerounrealscript.blogspot.de/2012/03/
using-configuration-files-in.html (30.08.2013)
URL:http://www.moug-portfolio.info/udk-config-files/ (30.08.2013)

[9] Wikipedia „Multiplayer"
URL:http://de.wikipedia.org/wiki/Mehrspieler (30.08.2013)

[10] Wikipedia „Singleplayer"
URL:http://de.wikipedia.org/wiki/Einzelspieler (30.08.2013)

[11] ActionScript 3 Referenz
URL:http://help.adobe.com/en_US/FlashPlatform/reference/
actionscript/3/index.html (30.08.2013)

[12] Schmieder, Thomas: Adaptive Medienwelten: Konzeption eines emotional-adaptiven Game- und Mediensystems, Masterarbeit, HS-Mittweida 2012

[13] Hauser, Tobias; Kappler, Armin; Wenz, Christian: Das Praxisbuch ActionScript 3, Galileo Design 2009

[14] Meyer, Andrea; Poehl, Hennig; Wetter, Andreas: Spiele entwickeln 2009, Weilburg/Hessen 2009

[15] Hudlicka, Eva: Affective game engines: motivation and requirements. In: Proceedings of the 4th International Conference on Foundations of Digital Games (FDG 09), New York 2009

Glossar

Game-Engine Software zur Steuerung eines Videospiels. Komponenten wie die Grafik oder der Ton des Spiels werden gesteuert.

Gameplay Ablauf eines Spiels, also Art und Weise die vom Spiel bereitgestellt wird, um ein Spielerlebnis aufzubauen.

Hotspot Ist ein öffentlicher drahtlose Internetzugriffspunkt.

Konfigurationsdatei Spezielle Datei, die von der Unreal Engine beim Initialisieren des Videospiels eingelesen werden kann.

Level Spielwelt, in der sich ein Spieler befindet.

Mission Aufforderung an eine Person, eine bestimmte Handlung durchzuführen.

Missionsstrang Menge an Missionen, welche zusammen eine Handlung ergeben.

Multiplayer Modus in einem Videospiel, bei dem der Spieler mit oder gegen andere menschliche Spieler interagiert.

NPC 'non-player character', ist eine Figur in einem Videospiel, die von einer künstlichen Intelligenz gesteuert wird.

Pawn Ist eine Klasse im UnrealScript, beinhaltet alle Funktionen für die Spielfigur selbst. Ist die 'Hülle', welche von einem Controller gesteuert wird.

PlayerController Klasse im UnrealScript, beinhaltet alle Funktionen zum Steuern einer Spielfigur durch den Spieler.

Scaleform Eine von Autodesk entwickelte Schnittstelle, um zwischen einer Flash-Datei und einer Game-Engine zu kommunizieren.

Singleplayer Modus in einem Videospiel, bei dem der Spieler ohne menschliche Mitspieler das Spiel absolviert.

Taser Pistolenähnliche Waffe, mit der Personen durch Elektroschocks außer Gefecht gesetzt werden können.

UDK 'Unreal Developer's Kit', ist eine Entwicklungsumgebung für Videospiele, welche auf der kostenlosen Version der Unreal Engine 3 basieren.

XML 'Extensible Markup Language', ist eine Sprache zur Darstellung von Hierarchien und Strukturen von Daten.

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 23.09.2013