HOCHSCHULE
MITTWEIDA
University of Applied Sciences

Fakultät Angewandte Computer- und Biowissenschaften

Professur Medieninformatik

Bachelorarbeit

Vergleichende Anwendung verschiedener Data-To-Text-Ansätze für die Generierung von Questtexten in Videospielen

Alex Prezewowsky

Mittweida, den 28. März 2023

Erstprüfer: Prof. Dr. rer. nat. Marc Ritter
Zweitprüfer: Manuel Heinzig, M. Sc.

Prezewowsky, Alex
Vergleichende Anwendung verschiedener Data-To-Text-Ansätze für die Generierung von Questtexten in Videospielen
Bachelorarbeit, Fakultät Angewandte Computer- und Biowissenschaften
Hochschule Mittweida — University of Applied Sciences, März 2023

Referat

Die immer größer werdenden virtuellen Welten von Computerspielen mit spannendem und glaubhaftem Inhalt zu füllen, ohne die Entwicklungszeit enorm in die Höhe zu treiben, ist eine der großen Herausforderungen für Spielentwickler heutzutage. Eine Möglichkeit dieses Problem anzugehen ist der Einsatz computergestützter Generierungsalgorithmen um manuellen Aufwand zu verringern. Die vorliegende Arbeit befasst sich mit der Umsetzung dreier Data-To-Text-Ansätze zum Zweck der automatischen Generierung von Questtexten aus einer Datenstruktur in der Spieleentwicklungsumgebung Unity. Die entstehenden Implementierungen werden im Anschluss evaluiert auf Eignung für den Anwendungsfall. Folgende Methoden zur Realisierung der Texte werden angewandt: Templating, Templating mit Template-Generierung aus einer kontextfreien Grammatik, sowie die Oberflächenrealisierungsbibliothek SimpleNLG.

Name: Prezewowsky, Alex
Studiengang: Medieninformatik
Seminargruppe: MI18w1-B
English Title: Comparative application of different data-to-text approaches for the generation of quest texts in video games

Inhaltsverzeichnis

1. Einführung

Die Forschung an Textgeneratoren hat in den letzten Jahren große Fortschritte gesehen, nicht zuletzt dank stetig neuer Erkenntnisse im Bereich des maschinellen Lernens. Deep Learning Modelle wie ChatGPT[1] oder Jasper[2] demonstrieren eindrucksvoll den Stand, auf dem sich die Wissenschaft in diesem Feld befindet. Übersetzungswerkzeuge wie Google's "Übersetzer" und Microsoft's "DeepL" bieten KI-getriebene Sprachübersetzungen aus über 100 Sprachen an [Fou16] [3]. In sozialen Netzwerken gibt es eine Vielzahl an Text-"Bots" [4,5], welche von Einzelpersonen entwickelt wurden, um zu unterhalten oder zu informieren. Für Journalismus, im Kundenservice, sowie für Content Marketing wird Textgenerierung erfolgreich kommerziell eingesetzt [Wie21] [6].

Die vorliegende Arbeit erkundet das Feld der algorithmischen Textgenerierung für Videospiele anhand des Anwendungsfalls der Generierung von "Quest Objectives". Abschnitt 1.1 gibt einen Überblick über die Problemstellung. In Abschnitt 1.2 wird das Forschungsziel formuliert und die Struktur der Arbeit vorgestellt.

1.1. Motivation

Videospiele sind interaktive Medien, deren Kernziel die Unterhaltung des Nutzers ist. Sie sind eine beliebte Freizeitbeschäftigung, und die Menge an sogenannten "Gamern" wächst stetig[7]. Texte waren von Beginn an ein Teil von Videospielen,

[1] https://openai.com/blog/chatgpt (04.03.2023)
[2] https://www.jasper.ai (04.03.2023)
[3] https://www.deepl.com/de/whydeepl (23.03.2023)
[4] https://nostalgebraist-autoresponder.tumblr.com/about (04.03.2023)
[5] https://www.sciencefriday.com/segments/i-twitter-bot (04.03.2023)
[6] https://www.ibm.com/watson (23.03.2023)
[7] https://www.wepc.com/news/video-game-statistics/ (24.03.2023)

so zum Beispiel in Text-Adventures, welche das Genre der Interactive Fiction hervorbrachten[8]. Man findet sie in Gegenstandsbeschreibungen, in Dialog-Untertiteln, als Hinweis in der Nutzeroberfläche, für Text-Assets wie Briefe oder Notizen, in Anleitungen und in Nutzer-Menüs.

Textgenerierung bietet das Potential, die Erstellung von Texten schnell und in großem Umfang vorzunehmen [vS22, S. 1]. Hierdurch könnte der Aufwand repetetiver Aufgaben verringert werden, wie beispielsweise das Schreiben von Informationstexten oder das Ausschmücken einer großen Spielwelt durch Autoren. Spielwiederholungen können abwechslungsreicher gestaltet werden, indem Charakternamen oder ihre Dialoge in jedem Spieldurchlauf anders sind. Letztlich kann Textgenerierung einzigartige Mechaniken ermöglichen, wie zum Beispiel dynamische Dialoge mit Spielwelt-Charakteren.

Linguistisch korrekte Texte zu generieren ist jedoch eine komplexe Aufgabe. Es existieren eine Vielzahl Methoden hierfür, von welchen nicht alle für eine Anwendung in Videospielen geeignet sind. Häufig existieren Implementierungen von Textgeneratoren nur in einem akademischen Kontext, womit eine Adaption für Videospiele schwierig ist. Erschwert wird dies auch aus dem Grund, dass Wissen um die Methoden, Vorteile und Risiken von Textgenerierung uneinheitlich in der Videospiel-Industrie verbreitet ist. [vS22, S. 9ff]

Ein mögliches Einsatzgebiet für Textgenerierung in Videospielen sind Questtexte. Eine “Quest“, zu deutsch “Suchmission“ ist eine Reise oder ein Auftrag, die, beziehungsweise der, von dem Spieler eines Videospieles angetreten wird [Sal]. Diese Mechanik ist in nahezu jedem Videospiel vorhanden. Informationen über eine Quest werden Spielern häufig textlich mitgeteilt. Diese Texte werden gemeinhin von einem Autor manuell verfasst und im Programm mit den restlichen Questdaten assoziiert. Statt Questtexte manuell zu schreiben, könnte diese Arbeit von einem Textgenerator übernommen werden. Die eingesparte Zeit und das Honorar kommt dem Entwicklungsaufwand zugute. Je nach Umsetzung könnten die Texte sogar ungeahnt abwechslungsreich sein, was ein Qualitätssprung wäre.

[8] http://brasslantern.org/community/history/timeline-c.html 23.03.2023)

1.2. Zielstellung und Aufbau der Arbeit

Das Ziel der Arbeit ist die Überprüfung der Eignung von Textgenerierungsansätzen für die Verwendung in Videospielen, speziell für die Generierung von Questtexten aus einer unterliegenden Datenstruktur. Das Unterfeld der Computerlinguistik, welches sich mit derartigen Prozessen auseinandersetzt, trägt die Bezeichnung "Data-To-Text".

- **Forschungsfrage**: Welcher Ansatz der Data-To-Text-Generierung ist für das Generieren von Questtexten in Videospielen am Besten geeignet?

Als geeignet gilt eine Technologie, wenn der Nutzen der Technik die Risiken und Nachteile aufwiegt. Der Nutzen der Textgenerierung für diesen Anwendungsfall besteht aus der Verringerung des Aufwandes bei gleicher oder sogar höherer Textqualität gegenüber der menschlicher Autoren. Die Risiken sind beispielsweise fehlerhafte Ausgaben oder unerwartete Schwierigkeiten bei der Generator-Entwicklung.

Kapitel 2 "Grundlagen" steigt tiefer in diese Fragen ein. Hier wird geklärt, was eine Quest in Videospielen ist, wozu sie dient, und anhand welcher Eigenschaften Quests beurteilt werden können. Darauf folgend wird der aktuelle Stand der Textgenerierung in Videospielen präsentiert, und auf die für den Entwicklungsprozess entscheidenden Probleme der Textgenerierung eingegangen. Zuletzt folgt ein tieferer Einstieg in die akademische Forschung zur Data-to-Text Generierung. Es werden Architekturen und Methoden vorgestellt und Vorgehensweisen zur Evaluation von Textgeneratoren erkundet.

Aus den Erkenntnissen des Grundlagenkapitels heraus werden in Kapitel 3 die Anforderungen an das Ergebnis ermittelt. Drei Textgenerierungsansätze werden für die Implementierung ausgewählt. Das Ziel ist die Erstellung von für diesen Anwendungsfall wiederverwendbaren Programm-Klassen. Kapitel 4 dokumentiert die Implementierung der Ansätze.

Kapitel 5 beschäftigt sich mit der Evaluation der drei Ansätze. Zunächst wird die Evaluationsmethodik vorgestellt. Danach werden die Ergebnisse der Evaluation präsentiert und die Eignung der Ansätze für den Anwendungsfall festgestellt. Zudem wird der am besten geeignete Ansatz bestimmt.

Das Fazit in Kapitel 6 fasst die Arbeit zusammen und gibt einen Ausblick auf weitere Forschungsmöglichkeiten in diesem Feld.

2. Grundlagen

Videospiele als Medium, und damit auch deren Texte, müssen spezifischen Anforderungen genügen, die nicht notwendigerweise auch in anderen Medien vorherrschen, in denen Textgenerierung eingesetzt wird. Das folgende Kapitel dient dazu, das theoretische Fundament für die spätere Konzeption und die Implementierung zu legen. Das Ziel der Implementierung ist, einen Textgenerator zu erhalten. Dieser soll in der Lage sein, eine nur in Datenform vorliegende Quest in menschenlesbarer Form als Text auszudrücken. Zwei Wissenschaftsfelder sind demnach immanent wichtig: Questdesign und Textgenerierung, speziell Data-to-Text-Generierung. Um zu verstehen, welche Versuche bereits unternommen wurden und auf diesen aufzubauen, wird außerdem ein Blick auf existierende Forschung im Bereich der Textgenerierung für Videospiele geworfen.

Zunächst soll in Abschnitt 2.1 geklärt werden, was eine Quest ist. Es wird zudem betrachtet, welche Bedeutung Quests in Videospielen zukommt und welche Eigenschaften eine gute Quest ausmachen. Im nächsten Schritt, in Abschnitt 2.2, wird die in der Industrie existierende Arbeit mit Textgenerierung in Videospielen vorgestellt. Es werden Probleme erläutert, die sich einer Nutzung in der Praxis entgegenstellen. Anhand von Beispielen wird gezeigt, wo die Technologie dennoch Eingang in die Spielentwicklung gefunden hat, und welche Werkzeuge Entwicklern dazu zur Verfügung stehen. Zuletzt wird der Stand der Textgenerierung in der theoretischen Forschung betrachtet. Abschnitt 2.3 ordnet die Textgenerierung ein in das Forschungsfeld der Computerlinguistik, und präsentiert bekannte Ansätze und Methoden zur Textgenerierung. Außerdem wird gezeigt, welche Arten der Programmevaluation es für diese Art der Software gibt.

Abbildung 2.1.: Die Übersicht der aktiven Quests im Spiel Genshin Impact. Links ist die Liste zu sehen. Eine Quest ist angewählt, und ihre Details werden rechts angezeigt.[1]

2.1. Quests in Videospielen

Der englische Begriff “Quest“ hat im deutschen die Entsprechung “Suchmission“. In mittelalterlichen Erzählungen bezeichnete das Wort die Heldenreise eines Ritters, an deren Ende Ruhm oder die Erfüllung eines Ziels steht. Videospiele verwenden den Begriff in ähnlicher Bedeutung.

> “Questing [...] bezieht sich meist auf eine Reise, die der Hauptcharakter aus eigenem Willen unternimmt, oder eine spezifische Aufgabe, die ihm gegeben wurde.“ [Sal, S. 2]

Jedes Videospiel gibt dem Spieler ein Ziel, welches dieser zu erreichen hat. Zwischen dem Erreichen des Ziels und dem Spieler steht immer eine Herausforderung, seien es Kämpfe, Rätsel oder Fähigkeitsprüfungen. Es kommt also zum Konflikt. Am Ende des Spiels wird das Ziel, oder ein äquivalentes Ergebnis erreicht. Dies ist die Reise des Spielers, seine Quest. Wird über Quests gesprochen, so sind jedoch auch alle Teilaufgaben innerhalb des Spiels, die dieser Form folgen, gemeint.

Eine klassische Videospiel-Quest besteht aus drei Schritten: Aufgabe, Konflikt und Belohnung. Der initiale Auftrag kann aus unterschiedlichen Quellen kommen. Meist wird der Spieler von einem NPC[2]) beauftragt, Etwas zu tun. Manchmal gibt das Spiel selbst diesen Auftrag. Häufig wird der Beginn einer Quest durch eine Einblendung in der Nutzeroberfläche begleitet, um den Spieler darüber zu informieren, dass eine neue Quest gestartet wurde. Die "aktiven" Quests sind dann in der Regel in einem eigenen Menüfenster aufgelistet, zusammen mit der Beschreibung des Ziels und des bisherigen Verlaufs. Abbildung 2.1 zeigt ein solches Menüfenster.

Der Spieler versucht nun, die ihm gestellte Aufgabe zu erledigen. Typischerweise fallen die Aufgaben, die von Quests gestellt werden, in eine der folgenden Kategorien [Sal, S. 3]:

- Töte Etwas oder Jemanden!
- Finde Etwas oder Jemanden!
- Überbringe einen Gegenstand!
- Sammle Zutaten oder Gegenstände!
- Eskortiere Jemanden!

Solche Aufgaben finden sich in fast jedem Videospiel. Eine Quest muss jedoch nicht zwingend so linear sein. Das Ziel kann sich im Verlauf ändern, oder es erwachsen zusätzliche Hindernisse, die der Spieler ebenfalls überwinden muss [Sal, S. 7].

Wurde die Quest erfolgreich beendet, erhält der Spieler eine Belohnung. Diese kann aus Elementen der Spielwelt bestehen, wie Ausrüstung, Spielwährung, oder dem Gewinn von Verbündeten. Sie kann aber auch eine Videosequenz oder ein Fortschritt in der Spielhandlung sein. Letzteres ist eine emotionale Belohnung für den Spieler. [Sal, S. 13f]

[1]https://genshin.hoyoverse.com (11.03.2023)
[2]non player character - Charaktere im Spiel, die nicht vom Spieler gesteuert werden

Quests als Werkzeuge im Videospieldesign

Die grundlegenden Problemstellungen der Aufgabentypen begreift ein Spieler schnell, und bald hat er Übung darin, sie zu lösen. Um keine Langeweile aufkommen zu lassen, müssen die Grundkonzepte daher kombiniert oder auf neue Weise präsentiert werden. [Sal, S. 7] Es entstehen komplexe Queststrukturen, da jede einzelne Quest aus mehreren kleineren Quests bestehen kann. Oftmals kann hierbei eine Unterteilung in Hauptquests und Nebenquests beobachtet werden.

Hauptquests, auch Main Quests genannt, stellen die höchste Eben der Queststruktur dar. Ihre Erledigung treibt die zentrale Handlung des Spiels voran. Nebenquests, oder Side Quests, sind optional und können separat verfolgt werden. Ihre Belohnung ist meist kleiner und hat die Form von Aufstufung der Charakterwerte oder Fähigkeiten.

Videospielentwickler nutzen solche Queststrukturen, um Spieler durch das Spiel zu leiten, ihre Aufmerksamkeit auf interessante Inhalte zu lenken und sie für ihr Engagement zu belohnen [Sal, S. 5]. All dies trägt dazu bei, dass der Spieler motiviert bleibt und eine positive Spielerfahrung hat. Quests sind zudem hervorragend dazu geeignet, die einzelnen Teile eines Spiels zu koordinieren, indem sie diese in einen vorgegebenen Ablauf einbetten [Sal, S. 24]. Manche Entwicklerteams haben genau aus diesem Grund dedizierte Spezialisten für diese Aufgaben, sogenannte Questdesigner [3] [4].

Gutes Questdesign

Die Qualität eines Designs wird an seiner Fähigkeit gemessen, dem Einsatzzweck zu dienen. In Videospielen besteht das letztendliche Ziel darin, dem Spieler ein unterhaltsames, “spaßiges“ Spielerlebnis zu bieten, um ihn am Spielen zu halten. Folglich muss auch das Questdesign danach bewertet werden, ob es der Spielerfahrung dient, oder sie sogar aktiv stört.

Zunächst braucht die Quest einen Grund zu existieren. Sie muss in der Welt verwurzelt sein und eine Handlung haben. Dem Spieler soll das Gefühl gegeben werden, etwas Sinnvolles zu tun. Er wird zusätzlich motiviert durch die Aussicht, mehr über

[3] https://jobs.smartrecruiters.com/CDPROJEKTRED/743999857102501-quest-designer (23.03.2023)

[4] https://gamejobs.co/Quest-Designer-at-Ubisoft-1222 (23.03.2023)

die Spielwelt und die Charaktere zu erfahren. [Sal, S. 6] Wenn es um die Bewertung einer Quest geht, ist die von ihr erzählte Geschichte oftmals der ausschlaggebende Faktor [Tha18].

Quests, die offensichtlich versuchen dem Spieler ein Vorgehen aufzudrängen, werden schnell als störend empfunden. Ein ähnlicher Fall tritt ein, wenn die Auswahl an Questtypen zu eintönig ist. [Sal, S. 5ff] Spiele sind ein *interaktives* Medium. Spieler erwarten daher, selbständig Entscheidungen treffen zu dürfen. Wird ihnen diese Möglichkeit genommen, sinkt ihre Freude am Spiel enorm.

Dies hängt auch damit zusammen, dass Spieler es genießen, während des Spielens in eine Rolle zu schlüpfen. Mit einer Entscheidung konfrontiert, bevorzugt es der Eine möglicherweise, einen Konflikt diplomatisch zu lösen. Ein Anderer greift direkt zur Waffe. Gibt man dem Spieler den Raum, diese Rolle auszuleben, taucht er in die Welt ein und fühlt sich ihr stärker verbunden. Questdesigner wissen dies, und kreieren darum Situationen ohne objektiv richtige Antwort, wo dennoch jede Entscheidung spürbare Folgen hat. [Sal, S. 4f]

Hierbei wird darauf geachtet, dass dem Spieler immer klar ist, welche Aufgabe ihm gerade gestellt wird. Konsequenzen bestimmter Entscheidungen werden im Vorhinein bekanntgegeben. Denn verwirrende oder nicht lösbare Quests frustrieren den Spieler ebenfalls.[Sal, S. 5,21]

Eine weitere Herausforderung ist es, Abwechslung und Konsistenz im Questdesign zu vereinen. Jedes Spiel hat ein oder mehrere Kernkonzepte, die eine kohärente Botschaft vermitteln sollen. Einerseits sollte jede Quest diese Grundnarrative spiegeln. Andererseits dürfen sich Quests nicht zu häufig ähneln, da ihre Erledigung sonst schnell zu lästiger Routine wird. [Sal, S. 7ff] Quests, welche es erfordern, bekannte Spielmechaniken auf neue Weise zu verwenden, oder anderweitig Einzigartig sind, helfen bei dieser Aufgabe [Sal, S. 10][Tha18].

Letztlich ist es wichtig, ein Spiel und seine Zielgruppe genau zu kennen [Sal, S. 21]. Spieler hegen bestimmte Erwartungen an Genres: Einzelspieler legen den Fokus auf eine gute Geschichte. In einem Rollenspiel steht die individuelle Anpassung des eigenen Charakters im Vordergrund. In Mehrspieler-Umgebungen möchten die Spieler hingegen zusammenarbeiten. Gutes Questdesign unterstützt die Spieler darin, diese Erfahrungen zu finden.

2.2. Textgenerierung in Videospielen

Textgenerierung kann ein nützliches Werkzeug für die Videospielentwicklung sein. Mit ihrer Hilfe kann der Umfang an Text-Assets für ein Spiel signifikant erhöht werden [vS22, S. 1], wodurch es möglich ist, Spielwelten mit stetig mehr Inhalt zu füllen. Zudem können generierte Texte "live" auf Spielzustandsveränderungen eingehen, indem bei ihrer Generierung Daten aus der zugrundeliegenden Spielwelt berücksichtigt werden.

Bisher findet Textgenerierung jedoch nur in wenigen spezifischen Fällen Eingang in Videospiele. Noch gibt es zu viele praktische Probleme, die einer weitreichenden Nutzung im Wege stehen [vS22, S. 9]. In Unterabschnitt 2.2.1 werden diese erläutert. Wird sie dennoch verwendet, liegt dies meist daran, dass das entsprechende Spiel explizit von Textgenerierung profitiert, oder ein akademisches Interesse besteht. Spiele des "Interactive Fiction" Genres, wie Textadventures, gehören dazu. Andere Anwendungsfälle sind das Generieren von NPC-Dialogen oder Item-Beschreibungen. Mittlerweile existieren Entwicklungswerkzeuge, welche das Implementieren von Textgeneratoren für Spiele vereinfachen.

2.2.1. Praktische Probleme

Videospielentwicklung ist ein aufwändiger und oft kostspieliger Prozess. Meist investiert ein Entwicklerstudio mehrere Jahre in ein einziges Spiel. Fehlentscheidungen zu Beginn können Verzögerungen im späteren Verlauf bewirken, indem sie das Team dazu zwingen, grundlegende Funktionen neu zu implementieren oder sogar zu verwerfen. Entwickler wägen daher genau ab, ob der Nutzen einer einzusetzenden Technik den durch sie entstehenden Aufwand rechtfertigt. Im Fall der Textgenerierung sprechen mehrere praktische Probleme gegen einen Einsatz in Videospielen.

Textgeneratoren sind inhärent komplex. Eine passende Ausgabe zu erhalten ist nicht garantiert [vS22, S. 11], und es besteht daher das Risiko, dass sporadisch fehlerhafte Texte produziert werden. Geschieht dies zur Laufzeit, können solche Fehler die Immersion des Spielers ruinieren, oder ihm notwendige Informationen vorenthalten. Andersherum, ist der Generator klein und besitzt nur wenige Produktionsregeln, sind die Ausgaben möglicherweise ungeeignet für ein Spiel, da sie langweilig und

vorhersagbar sind [vS22, S. 11]. Dies ist für ein Entwicklungsteam ein möglicherweise nicht akzeptables Risiko.

Theoretisch, hat eine Firma einen Generator entwickelt, können für einen Bruchteil des Aufwands Texte erstellt werden. Die Entwicklung eines Generators benötigt jedoch ebenfalls Zeit und Expertenwissen. Der Aufwand für die Texterstellung wird nicht notwendigerweise geringer, sondern nur an eine andere Stelle verschoben. [vS22, S. 11f] Zusätzlicher Aufwand entsteht, möchte man das Risiko fehlerbehafteter Texte verringern. Generatoren können in diesem Fall genutzt werden, um im Vorhinein Texte zu generieren. Diese werden dann von einem Menschen überprüft und gegebenenfalls korrigiert. [vS22, S. 11] Die Hoffnung, Textgeneratoren würden sich rentieren, indem sie Kosten und Aufwand in der Entwicklung einsparen, kann daher täuschen.

Erkenntnisse, welche die Risiken oder den Entwicklungsaufwand von Textgeneratoren verringern würden, finden kaum Verbreitung in der Videospielindustrie. Stegeren spricht von einer "Kluft zwischen Forschung und Industrie". Es fehle an Austausch zwischen den Interessenten, und auch innerhalb der Industrie. [vS22, S. 12] Dies führt dazu, dass Experten für Textgenerierung rar gesät sind. Der Wissenstand zu dem Thema ist uneinheitlich.

2.2.2. Existierende Ansätze

Textgenerierung erfreut sich großer Beliebtheit bei der Erstellung von Textadventures, sogenannter interactive Fiction (IF). Dieses Genre siedelt am Übergang zwischen klassischer Geschichtenerzählung und interaktivem Videospiel [vS22, S. 31]. Im Stile eines Gamebooks[5] findet die Geschichte pur in geschriebener Form Ausdruck, und der Spieler interagiert mit der so erlebten Spielwelt durch das Anwählen von Optionen oder eigenen Texteingaben.

Das Spiel "Knights of San Francisco" ist ein Vertreter dieses Genres. Statt einer vorgeschriebenen Geschichte zu folgen, wird für den Spieler im Hintergrund eine Spielwelt simuliert, welche durch einen Textgenerator auf in englischer Schrift erlebbar gemacht wird. Der Spieler besteht dynamische Kämpfe und trifft folgenreiche

[5]Spielbuch - in den 80er Jahren beliebte Buchform, in welcher der Leser ein individuelles Abenteuer erleben konnte.

Entscheidungen, alles im Kontext einer Texterzählung. [HW] Das Spiel wurde in einer eigenen Entwicklungsumgebung namens “Egamebook“ umgesetzt. Diese stammt von den selben Entwicklern, und steht jedem frei zur Verfügung, um damit eigene Egamebook-Geschichten zu skripten. [6]

Einen anderen Ansatz wählt “epitaph“. In diesem Spiel werden automatisch Sprachen für Alien-Zivilisationen generiert, aus welchen daraufhin die Bezeichnungen dieser Völker für Pflanzen, Tiere, Planeten und sich selbst abgeleitet werden. [Kre] Die Textgenerierung ist hier qualitativer Natur [vS22, S. 33], sie hinterlegt das Spiel mit einer Sprachstruktur, die Bezeichnungen innerhalb Zivilisation gleichen sich, und sind somit glaubhafter.

Ein weiterer zu beobachtender Ansatz ist jener, Textgenerierung zu nutzen um abwechslungsreiche Dialoge mit NPC's zu ermöglichen. Es existiert Forschung mit dem Ziel, diese Technologie ergänzend in “Open-World“[7] Spielen einzusetzen, um die erlebten Gegenden und Charaktere glaubhafter zu machen.[vS22, S. 35] Andere Spiele setzen dynamische Dialoge als ihre Kernmechanik ein, und gestalten den Rest des Spiel darum herum.

Das Adventure-Game “Bot Colony“ ist ein Beispiel für diese Kategorie. Die Aufgabe des Spielers ist es, im Dialog mit Robotern Tathergänge zu rekonstruieren und ihnen Anweisungen zu geben. Diese Roboter haben jedoch keine vorgeschriebenen Dialog-Pfade, und sind stattdessen in der Lage, dynamische Gespräche zu führen, und sogar Bezug auf die Spielwelt um sie herum zu nehmen. [Eug10]

Textgenerierung wird zudem in Verbindung mit anderen Formen prozeduraler Generierung verwendet, um vollständig generierte Spielwelten zu erschaffen. Dies ist bisher jedoch nur eine rein akademische Forschung. [FUR+20]

2.2.3. Entwicklungswerkzeuge

Ein Entwicklungswerkzeug ist eine Software, welche dazu gedacht ist, bei der Software-Entwicklung zu *helfen*, und ist häufig nicht Teil der entstehenden Software. Im Fall von Videospielen wären dies zum Beispiel Entwicklungsumgebungen wie Unity[8] oder

[6] https://egamebook.com (23.03.2023)
[7] weitläufige, für den Spieler frei erkundbare Spielwelten
[8] https://unity.com (10.03.2023)

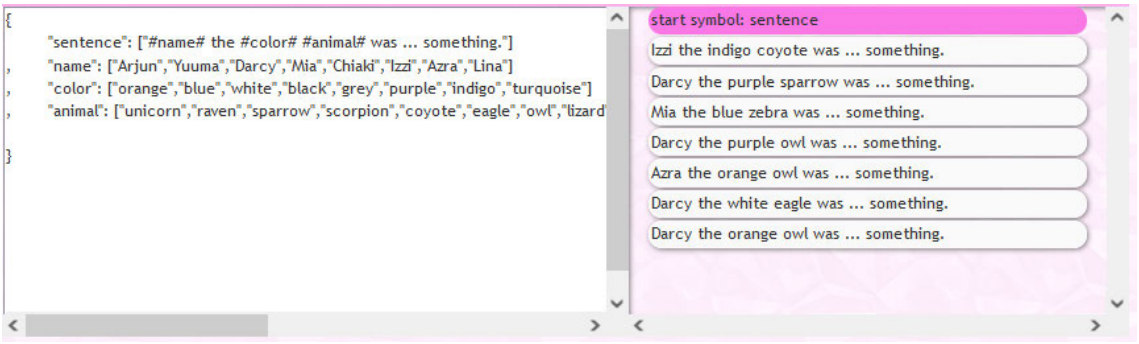

Abbildung 2.2.: Eine Beispielimplementierung von Tracery. Links ist die Grammatik zu sehen, rechts einige mögliche Ergebnisse.[12]

die Unreal Engine[9], welche Skripting, Building und Rendering-Technologien für die Videospielentwicklung bereitsstellen. Für Textgenerierung findet man sogenannte Autorenwerkzeuge (engl. authoring tool), welche den Entwickler bei der Erstellung von Textgeneratoren unterstützen [vS22, S. 29f].

Das Open-Source[10] Tool "Tracery" erlaubt es, textgenerierende Grammatiken in Form einer kontextfreien Grammatik[11] zu schreiben. Die Grammatik selbst wird in JSON geschrieben und dann anhand der in dem Programm definierten Regeln entfaltet (engl. expansion). Der Fokus liegt hierbei auf Nutzbarkeit, auch für Laien, und schnellen Ergebnissen. [CKM15] Aus diesem Grund ist Tracery ein beliebtes Programm für Spiele, interactive Fiction und Twitter-Bots geworden [vS22, S. 29]. Tracery ist allein nicht in der Lage, gezielt Informationen auszudrücken. Die Ergebnisse der Realisierung einer Grammatik sind zufällig.

Das Tool Expressionist [RSMW16] hingegen ist speziell für Laufzeitumgebungen wie Videospiele entwickelt worden. Es nutzt, ähnlich wie Tracery, eine Syntax für kontextfreie Grammatiken, erlaubt es aber, beliebige Marker (engl. tag) an Elemente der Grammatik zu heften. Anfragen der Laufzeitumgebung geschehen in der Form eines Sets an Tags, woraufhin ein Ausdruck generiert werden kann, der genau diese Tags beinhaltet. Im Unterschied zu Tracery muss ein Entwickler eine eigene "expansion enginge" für die Realisierung der Ausgabetexte implementieren. [RSMW16]

[9] https://www.unrealengine.com (10.03.2023)
[10] Open Source - Software, deren Quellcode veröffentlicht wurde
[11] Siehe Unterabschnitt 2.3.2
[12] http://www.crystalcodepalace.com/traceryTut.html (10.03.2023)

2.3. Data-To-Text Forschungsansätze

Das Feld der Computerwissenschaft, welches sich mit dem Verstehen und dem Erzeugen "natürlicher", das heißt menschlicher, Sprache mithilfe von Computern beschäftigt, nennt sich Computerlinguistik (CL) oder auch linguistische Datenverarbeitung (LDV). Im Englischen existieren hierfür die Begriffe "Natural Language Processing (NLP)" und "Computational Lingustics (CL)" [RD00, S. 1f]. Es ist ein interdisziplinäres Forschungsfeld mit Verbindungen zu Linguistik, Computerwissenschaft und künstlicher Intelligenz und baut stark auf den Erkenntnissen in diesen Forschungsfeldern auf. [13]

Seinerseits ist Computerlinguistik unterteilt in "Natural Language Understanding (NLU)", das Verstehen natürlicher Sprache, sowie "Natural Language Generation (NLG)", der Textgenerierung, [RD00, S. 2f] wobei bei Letzterer noch zwei weitere Aufgabenkategorien unterschieden werden: "Data-To-Text" und "Text-to-Text". Text-to-Text befasst sich mit mit der automatischen Zusammenfassung von bereits vorliegenden Artikeln, Überschriftenerstellung oder automatisierter Textübersetzung. Die Aufgabe von Data-to-Text-Anwendungen ist die Generierung von Text aus unterliegenden, non-linguistischen Daten. [GK18, S. 4f.] Wichtig zu erwähnen ist, dass diese Anwendungsgebiete ineinander übergehen, und oftmals werden erprobte Techniken in mehr als einem Gebiet eingesetzt. [GK18, S. 6 1.1]

Die als Text darzustellenden Daten können unterschiedlichste Formen haben, von Wetterdaten, Daten medizinischer Instrumente und Umfragedaten bis hin zu Finanzdaten oder Sportergebnisse. Bekannte Anwendungsbeispiele in der Industrie sind "Chat-Roboter" im Kundenservice [14] oder Newsticker auf Sportwebsites [Wie21]. Je nach Form der Daten und den Qualitätsanforderungen an den zu generierenden Text kommen unterschiedliche Techniken zum Einsatz, von einfacher Text-Substituierung bis hin zu anspruchsvollen Deep-Learning Modellen.

Es existieren mehrere Möglichkeiten, Textgenerierungssoftware zu klassifizieren. In ihrer Studie "Survey of the State of the Art in Natural Language Generation: Core tasks, applications and evaluation" [GK18] präsentieren Gatt und Kramer zwei

[13] https://uni-tuebingen.de/fakultaeten/philosophische-fakultaet/fachbereiche/neuphilologie/seminar-fuer-sprachwissenschaft/studium-lehre/studiengaenge/faq/was-ist-computerlinguistik (23.03.2023)

[14] https://www.ibm.com/watson (23.03.2023)

verbreitete Optionen: Klassifizierung anhand der *Architektur* einer Software oder anhand der verwendeten *Methoden* [GK18, S. 23]. Wie Textgenerierungssoftware evaluiert wird, zeigt Unterabschnitt 2.3.3.

2.3.1. Architekturen

Robert Dale und Ehud Reiter formulierten in ihrem Buch "Building Natural Language Generation Systems" von 2000 eine modulare Pipeline, welche die Strukturen von existierender Textgenerierungssoftware verallgemeinerte [RD00] und in einzelne Aufgabenschritte herunterbrach. Die eingegebenen Daten werden in mehreren abgetrennten Abschnitten analysiert, umstrukturiert und schließlich als Text ausgegeben. Dies geschieht zumeist mithilfe einer internen Datenbank, welche für das Programm als Lexikon fungiert [GR09]. Diese Architektur wird als "modular" bezeichnet und gilt als die klassische "Pipeline" [15]. [GK18, S. 22f]

Folgende Aufgabenbereiche sind innerhalb der Pipeline definiert:

- Inhaltsbestimmung: Sortieren und Filtern der Eingabedaten, Bestimmung der im Text auszudrückenden Inhalte
- Dokument-Strukturierung: Organisation der soeben gewählten Inhalte
- Aggregation: Überführung der organisierten Inhalte in eine linguistische Struktur, wie beispielsweise Sätze oder Phrasen.
- Wortwahl: Wahl der lexikalischen Begriffe für die Inhalte mithilfe des Lexikons
- Referenzgenerierung: Generierung von Fachwörtern oder Referenzwörtern für bestimmte wiederholt auftauchende Inhalte, sogenannte Koreferenzen
- Realisierung: Wahl von Morphologie, Syntax, Orthographie und schließlich Generierung des Ausgabetextes

Ein weiterer Vorteil der Unterteilung war, dass sich Forscher auf bestimmte, ihrerseits sehr komplexe Unterprobleme, wie die Referenzgenerierung (engl. referring expression generation), konzentrieren können. Entworfene Module könnten mit bereits existierenden Modulen getestet werden, und der Aufwand eine eigene Testsoftware

[15] englisch für: Leitung. Bezeichnet einen Datenstrom zwischen zwei Prozessen. Die Ausgabe des einen Prozesses wird als Eingabe des folgenden Prozesses verwendet.

zu schreiben fiele weg. [RD00] Folglich entstanden eine Reihe von eigenständigen Softwareprojekten, wie “SimpleNLG“, die sich nur auf den Schritt der Realisierung konzentrierten. [GR09, S. 22 3]

Eine Alternative zu dem modularen Softwareansatz sind globale oder “end-to-end“ Ansätze, in welchen keine klare Linie zwischen Eingabe, Verarbeitung und Ausgabe gezogen wird. Ein weiterer Begriff für diese Kategorie ist “integriertes“ System. Solche Architekturen bauen meist auf statistischen Verfahren wie Maschinellem Lernen auf, wobei das Modell eine Beziehung zwischen bestimmten Eingaben und gewünschten Ausgaben “lernt“. [GK18, S. 22 3]

Zuletzt existiert nach dieser Klassifizierung noch der “planning“ Ansatz. Dieser sieht das Problem in der Tradition von automatisiertem Planen, einem Teilbereich der KI-Forschung, welche sich mit dem Problem beschäftigt, Computern die Fähigkeit zu geben, selbstständig ihre Handlungen zu planen und anzupassen, um bestimmte Ziele zu erreichen.[16] [GK18, S. 22 3]

2.3.2. Generierungsmethoden

Statt von einem Standpunkt der Systemarchitektur kann NLG-Software aufgrund der verwendeten Generierungsmethoden unterteilt werden. Es existieren regelbasierte Ansätze, statistische Ansätze und, innerhalb von Letzterem, Machine Learning und Deep Learning. [vS22, S. 45]

Unter den regelbasierten Ansätzen sind Templates die Simpelsten. Templates, frei auf deutsch übersetzt “Schablonen“, funktionieren nach dem Prinzip der Textersetzung. In einem vordefinierten Text werden Markierungen gesetzt, welche im Folgenden mit den Ausdrücken für die tatsächlichen Daten ersetzt werden. Diese Methode wird durch den manuellen Aufwand der Template-Erstellung limitiert, bietet jedoch volle Kontrolle über das Ergebnis und verringert das Auftreten grammatikalisch falsch geformter Sätze. [GK18, S. 19 2.6.1] Eine andere Methodik macht sich die mathematische Theorie von formalen Grammatiken zunutze. Diese ist in der Lage, anhand von wenigen Grundregeln einen Ergebnisraum grammatikalisch korrekter Wörter oder Sätze einer Sprache zu definieren. [GK18, S. 20] Für die Textgenerierung kann aus diesen Regeln ein Graph aufgebaut werden, das System durchläuft diesen

[16] https://ai-leaders.de/portfolio/was-ist-automatisiertes-planen/

und trifft anhand der angetroffenen Informationen Entscheidungen über die Platzierung der Wörter. Eine große Anzahl Realisierungsmodule basieren auf formalen Grammatiken, so auch SimpleNLG. [GK18, S. 20 2.6.2]

Statistische Ansätze umfassen ein weites Feld, und können sowohl als einzelne Module, als auch als komplette globale Architekturen angetroffen werden. Für Realisierungsprozesse wurden erfolgreich stochastische Modelle angewandt, meist in Kombination mit Grammatiken. [GK18, S. 20f] Neuronale Netzwerke lernen aus vorgegebenen Korpora, Verknüpfungen zwischen Eingabedaten und Ausgabesätzen zu bilden. Somit erlangen sie theoretisch die Fähigkeit, unbekannte Daten ohne menschliche Hilfe in sinnvollen Sätzen auszudrücken. Hierfür muss jedoch ein solcher Korpus vorliegen. Für Videospielanwendungen existieren bisher deutlich weniger Korpora als für klassische Anwendungsfälle [vS22, S. 64f], wie zum Beispiel in der Medizin.

2.3.3. Evaluationsmethoden

Das Ziel eines Textgenerators kann unterschiedlich gewählt werden, aber fast immer soll menschenähnlicher, "natürlicher" Text entstehen. Um die entstandenen Texte zu überprüfen, eine gleichbleibende Qualität zu gewährleisten und, im Falle von statistischen Methoden, das Modell zu trainieren, benötigt es möglichst objektive Metriken. Textgeneratoren werden daher zumeist anhand ihrer Ausgabetexte evaluiert. [RD00, S. 37] Unterschieden wird hierbei zwischen zwei Test-Kategorien, mehreren Qualitätsmerkmalen und verschiedenen Vorgehensweisen zur Evaluation.

Extrinsische Tests bewerten ein System nach seiner Fähigkeit, dem letztendlichen Ziel in der Umgebung zu dienen, in der es eingesetzt wird (fitness for purpose). [vS22, S. 65 3.5] [S. 66ff][GK18] Die sehr viel diverseren "intrinsischen" Evaluationen fokussieren sich auf die Bewertung des Systems ohne seinen Kontext. [vS22, S. 65 3.5] Wie gut ist das System in der Lage, seine Aufgabe zu erfüllen (Daten in Text auszudrücken)? Entspricht der generierte Text bestimmten Qualitätsanforderungen, wie beispielsweise Menschenähnlichkeit (humanlikeness), Lesbarkeit (readability), Übersichtlichkeit (clarity) und Relevanz (relevance) und gibt er die gewünschten Inhalte wieder (correctness)? Wie gut passt der entstehende Text zu dem Stil des Kontextes (genre compatibility)? [GK18, S. 66ff] Ein Textgenerator für ein Video-

spiel mit einem Mittelalter-Setting sollte zum Beispiel keine komplizierten Sätze mit modernen Wörtern generieren.

Je nach gewähltem Evaluationsziel kommen unterschiedliche Methoden zum Einsatz, manchmal allein, oftmals jedoch in Kombination miteinander. Werden nur die generierten Texte betrachtet und unabhängig von der restlichen Funktionalität der Software untersucht, dann handelt es sich um eine "black box evaluation". Wenn im Gegensatz dazu das Programm auseinandergenommen und einzelne Abschnitte evaluiert werden, so ist dies eine "glass box evaluation". [GK18, S. 74 7.3]

Die einfachste Methode, um Textgeneratoren zu evaluieren, ist, die generierten Texte einer ausgewählten Gruppe von menschlichen Probanden vorzulegen. Diese Methode basiert also auf subjektiver Beurteilung. Sie wird zumeist eingesetzt um die Lesbarkeit, Relevanz und Genauigkeit der Texte zu überprüfen. Auch Stil und, für extrinsische Evaluationen, Effektivität des Textes, werden auf diese Weise beurteilt. [GK18, S. 67 7.1.1] Da jedoch die Urteile von Menschen zwischen Evaluationen stark schwanken, ist es schwierig hieraus belastbare Ergebnisse zu erhalten, anhand derer unterschiedliche Systeme miteinander verglichen werden können. Zudem ist das Erhöhen der Genauigkeit von subjektiven Methoden mit erheblichem Aufwand verbunden. [GK18, S.74f 7.4.1]

Um vergleichbare Ergebnisse zu erhalten, und außerdem den bei Umfragen notwendigen Organisationsaufwand zu vermeiden, kann der entstandene Text mit automatischen Metriken evaluiert werden. Ein Algorithmus vergleicht hierfür die entstandenen Texte mit einem Korpus an Ergebnistexten, welche bereits die gewünschte Form haben. Es handelt sich also um eine Form objektiver Beurteilung unter Zuhilfenahme eines Korpus. Das unter Beobachtung stehende Qualitätsmerkmal ist die Menschenähnlichkeit (humanlikeness). [GK18, S. 70f]

Um den Problemen der subjektiven Evaluation entgegenzuwirken können anstelle dessen Experimente durchgeführt werden. Die Probanden geben dabei keine Beurteilung ab, sondern nutzen das System, während ihre Aktivitäten dabei erfasst werden. Mit diesem Vorgehen können also ebenfalls objektive Daten erhoben werden, für Qualitätsmerkmale, die ansonsten nur subjektiv bewertbar sind. Dies ist zum Beispiel die Eignung des Systems für sein Einsatzgebiet. [GK18, S. 71f 7.1.3]

3. Theoretische Überlegungen und Konzeption

Das folgende Kapitel beschäftigt sich mit der Konzeption des zu implementierenden Systems. Abschnitt 3.1 begründet die Entscheidung, das Design der Textgeneratoren anhand eines Korpus zu gestalten. Es werden verschiedene Korpora-Kandidaten vorgestellt und die Wahl des zu verwendenden Korpus erläutert. In Abschnitt 3.2 werden die Anforderungen an das entstehende System geklärt. Zur Ermittlung dieser wird die Form der Korpustexte und das Wissen aus Kapitel 2 verwendet.

Aufbauend auf den Theorien der Data-to-Text-Forschungsansätze in Abschnitt 2.3, und unter Beachtung der entwickelten Anforderungen, wird in Abschnitt 3.3 das Programmkonzept entwickelt. Die gewählte Entwicklungsumgebung wird vorgestellt und die Wahl der Textgenerierungsansätze begründet. Zudem wird die theoretische Eignung der Ansätze betrachtet.

3.1. Wahl eines geeigneten Korpus

Es gibt mehrere Vorgehensweisen, um das Design eines Textgenerators zu planen. Ein erprobter Weg ist, die gewünschten Eingabe- und Ausgabedaten (Input und Output) des Systems als Ausgangspunkt zu nehmen, und daraus die Anforderungen zu ermitteln. Gemeinhin wird eine solche Datensammlung als Korpus bezeichnet. Die generierten Texte des Systems werden dann zumeist mit den Wunsch-Ausgabedaten verglichen, um die Leistung des Generators zu bewerten. [RD00, S. 30ff] Um möglichst nahe an Industrie-Standards für Questtexte zu arbeiten und sowohl die Implementierung, als auch die Evaluierung zu erleichtern, wird der korpusbasierte Design-Ansatz gewählt.

Das Erlangen geeigneter Korpora für Textgenerierungsanwendungen in Videospielen bringt eigene Schwierigkeiten mit sich. Eine Quelle für Videospieltexte sind natürlich

die Spieledateien selbst. Jedoch liegen diese meist in proprietären, verschlüsselten Formaten vor, wodurch der Zugriff auf diese verwehrt bleibt. [vST21] Das Abschreiben von Texten während des Spielens ist aufwändig, und beim Abschreiben können Fehler entstehen oder Texte übersehen werden. Eine dritte Möglichkeit ist, Texte aus Fan-Websites zu extrahieren. Diese Methode ist weniger aufwändig als manuelles Abschreiben, leidet aber an den gleichen Fehlerrisiken. Auf diese Weise gewonnene Korpora bedürfen zusätzlicher Bearbeitung, bevor ihre Qualität jenen Korpora gleicht, die aus Spieldateien gewonnen wurden. [vST21]

Im Fall dieser Arbeit soll der Input aus einer Datenstruktur bestehen, die einen Questauftrag beschreibt. Der Output soll formulierter Text sein, der diesen Auftrag ausdrückt. Bei der Auswahl soll darauf geachtet werden, dass die Texte tatsächlich Quest-Texte sind und aus einem kommerziellen Spiel stammen. Da die Korpora nicht dem Zweck dienen sollen, ein Machine-Learning-Modell zu trainieren, ist ihr Umfang von minderer Bedeutung. Optimalerweise paart der Korpus bereits Inputdaten und Output. Zudem muss der Korpus so gewählt werden, dass ein System zu seiner Umsetzung im Rahmen dieser Arbeit möglich ist.

3.1.1. Korpus-Kandidaten

Für die vorliegende Arbeit wurden 3 Textsammlungen in die engere Auswahl genommen: Pinnwand-Anschläge aus dem Spiel “The Witcher 3“ [1], ein “World of Warcraft“ [2]-Quest-Datenset [Mys20] [Mys] und eine Sammlung aus Questtexten des Spiels “Torchlight II“ [3,4].

Die Recherche nach Questtext-Korpora hat ergeben, dass der Auftrag einer Quest in sehr unterschiedlichen Formen formuliert sein kann. Grob können 3 Kategorien unterschieden werden:

- Einleitungstext: Hier wird der Auftrag das erste Mal formuliert, zum Beispiel durch einen NPC oder einen Anschlag auf einer Pinnwand. Manchmal darf der Spieler hier wählen, ob er die Quest beginnt oder nicht.

[1] https://witcher.fandom.com/wiki/Category:The_Witcher_3_notice_board_postings (13.03.2023)
[2] https://worldofwarcraft.blizzard.com (13.3.2023)
[3] https://www.torchlight2.com (13.03.2023)
[4] https://github.com/hmi-utwente/video-game-text-corpora (13.03.2023)

Titel	Objective	Einleitungstext
Message from the West	Read the Letter from Saurfang and then destroy it.	Just play along. It is good to see you here, soldier! Lord Hellscream has undoubtedly sent his strongest warriors to Lord Agmar! Read it and then destroy it in the fire pot next to me.
A Leg Up	Recover 10 Stolen Tallstrider Legs for Narasi Snowdawn at the Argent Tournament Grounds.	The Argent Crusade dispatched a fast ship carrying tallstrider meat and other supplies for us here at the tournament grounds, but it never arrived. Scouts reported seeing it attacked by the Kvaldir, fearsome vrykul raiders who roam the seas. They've slain the crew and taken the cargo to an island called Hrothgar's Landing. If you look to the north, you can see it shrouded in the mist. Would you help us in recovering the stolen supplies? Without those tallstrider legs, we'll soon be out of food.
Return to the Exodar	Speak with Caedmos at the Exodar.	Your master Caedmos of the Exodar has requested to see you. It concerns your training, but I don't really know anything beyond that.

Tabelle 3.1.: Beispiele aus dem "World of Warcraft"-Dataset

- Beschreibungstext: Eine detaillierte Zusammenfassung des Auftrags. Manchmal wird hier der gesamte bisherige Verlauf der Quest dokumentiert. Während der Recherche konnten nur Einzelbeispiele, jedoch kein Korpus dieser Sorte gefunden werden.
- Zieltext: Sehr kurze Zusammenfassung des Auftrags in Form einer Anweisung. Auf englisch wird hierfür der Begriff "Objective" verwendet.

In den gewählten Korpora sind Einleitungstexte dominant. Die "Witcher 3"-Textsammlung besteht nur aus Einleitungstexten. Im "World of Warcraft"-Datenset findet man neben Questtiteln ebenfalls Einleitungstexte, aber auch Objective-Texte. Der "Torchlight II"-Korpus beinhaltet ebenfalls Einleitungstexte und Questtitel, aber führt als Einziger zusätzliche Metadaten zu jedem Eintrag auf. Diese geben Auskunft über interne Repräsentationen der Quest, wie Questname, Auftraggeber und Speicherort, sind jedoch keine Input-Daten im Sinne unseres gewünschten Korpus.

3.1.2. Wahl des Korpus

Keiner der Korpus-Kandidaten ist direkt für das Vorhaben in dieser Arbeit geeignet. Erstens fehlt bei allen Korpora die gewünschte Input-zu-Output-Paarung. Auch bei anderen Korpora, die nicht in die engere Auswahl genommen wurden, war dies der Fall. Zweitens: In den gewählten Korpora sind Einleitungstexte dominant. Es ist festzustellen, dass all diese Texte eine Erzählung beinhalten. Es werden Personen und Ereignisse referenziert, die offensichtlich Teil der Spielwelt sind. Der Stil ist einheitlich, aber die Länge und Formulierung der Texte, ebenso ihr Inhalt, variiert extrem. Ein Generator, der derartige Texte erstellen kann, ist aufwändig zu entwerfen und zu implementieren, und das würde den Rahmen dieser Arbeit sprengen.

Das Erstellen eines eigenen Korpus scheint aus diesen Gründen vielversprechend. Jedoch sprechen zwei Argumente dagegen: der Aufwand, einen eigenen Korpus zu entwerfen, ist ebenfalls sehr hoch und die enthaltenen Texte würden nicht repräsentativ für Industrie-Praktiken sein.

Die Probleme sind also die Komplexität der Ausgabetexte und das Fehlen von Eingabedaten. Das Problem der Text-Komplexität löst sich, wenn der Fokus auf die Quest-Objective-Texte des "World of Warcraft"-Datensets gerichtet wird. Diese

Id	14
Task	Collect
ContractorName	-
ContractorHome	-
TargetName	Narasi Snowdawn
TargetHome	the Argent Tournament Grounds
ItemName:amount:Home	Stolen Tallstrider Legs:10:
Enemy:amount:Home	-

Tabelle 3.2.: Beispiel für den Input des Korpus

ähneln einander stark, und es sind direkt einige Regeln in ihrer Formulierung erkennbar. Diese Eigenschaften begünstigen die Generierung durch einen Textgenerator. Auch ist der Aufwand, dieses Unter-Set mit einem eigenen Set an Eingabedaten zu vervollständigen vergleichsweise gering. Da sich die Texte sehr ähneln, ist es leicht veränderliche Inhalte zu identifizieren.

Der Korpus, auf dessen Grundlage die Textgenerierungsansätze entworfen werden, wird demnach folgendermaßen gewählt: Als Output-Part des Korpus wird eine Sammlung von Quest-Objective-Texten gewählt, welche eine Untermenge des “World of Warcraft“-Datensets sind. Der Input-Part wird manuell erstellt, indem aus den Inhalten des Outputs auf Eingabedaten geschlossen wird.

3.2. Ergebnisanforderungen

Data-To-Text-Generatoren sind Programme oder Programm-teile, die nach der Eingabe von non-linguistischen Daten, Texte in natürlicher Sprache ausgeben. Eine grundsätzliche Anforderung ist daher, dass die implementierten Ansätze *menschenähnliche* Texte generieren. Für den Anwendungsfall der Questtextgenerierung ergeben sich jedoch noch weitere Anforderungen. Diese werden aus den Erkenntnissen des Grundlagenkapitels und den Inhalten des Korpus ermittelt.

3.2.1. Anforderungen an die Textgenerierung

Abschnitt 2.1 definiert die qualitativen Anforderungen an eine Quest. Diese sind wie folgt:

- Es wird eine Geschichte erzählt.
- Dem Spieler wird Entscheidungsfreiheit gegeben.
- Die Anforderungen sind klar.
- Die Quest ist einzigartig.
- Die Spielerfahrung wird unterstützt.

Da Questtexte Ausdruck der unterliegenden Quest sind, ist zu vermuten, dass einige dieser Anforderungen auch für sie gelten. Anhand der in Abschnitt 3.1 vorgestellten Korpora wird nun ermittelt, welche der Eigenschaften einer Quest auch von Questtexten getragen werden.

In der "Witcher 3"-Textsammlung findet man Elemente der Quest-Erzählung und die Anforderungen der Quest an den Spieler. Jeder Text ist einzigartig formuliert und unterstützt gleichzeitig die Fantasie von der Mittelalterumgebung. Dies kommt der Einzigartigkeit der Quest zugute und unterstützt die Spielererfahrung des Spiels. Das Gleiche gilt für die Einleitungstexte im "World of Warcraft"-Datenset. Die Objective-Texte hingegen legen einen immensen Fokus auf die Klärung der Anforderungen, geben jedoch nur einen kleinen Hinweis auf die Erzählung durch die Nennung der Namen von Orten und Personen. Die "Torchlight II"-extsammlung hingegen gleicht in ihren Eigenschaften der "Witcher 3"-Textsammlung.

Ob die unterliegenden Quests Entscheidungsfreiheit bieten oder nicht, ist aus den Texten schwer erkennbar. Dies mag daran liegen, dass die betrachteten Texte möglicherweise nur den initialen Schritt des Spielers leiten wollen. Folgende qualitative Eigenschaften einer Quest werden aber durchaus durch die Questtexte getragen: der Erzählungsaspekt, klare Anforderungen, Einzigartigkeit und die Unterstützung der Spielerfahrung.

Anhand dieser Erkenntnisse ist es möglich, die Anforderungen an die zu generierenden Questtexte zu formalisieren:

- **Konsistenz**: Die Texte müssen zu einander und zum Spiel passen. Dies umfasst die korrekte Verwendung der englischen Sprache sowie die Einhaltung des Stils. Es muss sichergestellt werden, dass der Questtext die Spielerfahrung unterstützt.
- **Varianz**: Die Texte müssen abwechslungsreich sein. Abweichungen in der individuellen Formulierung unterstützen die Einzigartigkeit einer Quest.
- **Wiedergabetreue**: Die Texte nutzen eingegebene Daten, und geben diese in ausreichendem Umfang wieder. Als ausreichend gilt, wenn alle notwendigen Daten im Ergebnis enthalten sind. Um Raum für Varianz zu lassen, müssen nicht alle vorhandenen Daten ausgedrückt werden. Notwendige Daten umfassen alle Informationen, die der Spieler zur Erledigung der Quest benötigt, sowie mindestens ein Name eines Ortes oder ein Person. Auf diese Weise sind Anforderungen klar, und ein Bezug zur Erzählung der Quest wird hergestellt.

Konsistenz und Varianz schließen sich nicht aus, jedoch führen Maßnahmen zur Erhöhung der Konsistenz leicht zu einer verringerten Varianz. Die Herausforderung ist also, ein Optimum zu finden zwischen möglichst geringen Inkonsistenzen und einem möglichst großen Ergebnisraum. Dies wird umso wichtiger, je größere Mengen an Texten generiert werden.

Aus Unterabschnitt 2.2.1 ist bekannt, dass das Fehlerrisiko von Textgeneratoren einen wesentlichen Faktor bei der Entscheidung spielt, Textgenerierung zu verwenden. Das Auslassen wichtiger Daten, ebenso wie sprachliche Fehler im Text, vermindern die Qualität der Spielerfahrung. Sind die Generierungsregeln zu strikt, entstehen langweilige Texte, und es geht die Einzigartigkeit verloren. Auch aus diesem Grund sind die Anforderungen der Konsistenz und Varianz wichtig.

3.2.2. Anforderungen an die Software

Unterabschnitt 2.2.1 führt aus, dass Textgeneratoren inherent aufwändig zu entwickeln sind, und zudem das Wissen um ihre Entwicklung wenig Verbreitung erfahren hat. Die Entwicklungswerkzeuge Tracery [CKM15] und Expressionist [RSMW16] wurden entwickelt, um diese Probleme zu lösen. Sie sind so gestaltet, dass sie wiederverwendbar und einfach zu bedienen sind. Das in dieser Arbeit entstehende Programm soll danach streben, diese Anforderungen ebenfalls zu erfüllen.

Die Wiederverwendbarkeit eines Textgenerators für Spiele wird zumeist durch den Aufwand gehemmt, ihn für einen neuen Anwendungsfall anzupassen. Ist der Generator zu spezifisch für eine bestimmte Aufgabe designt, oder existieren Anwendungsbeispiele nur in rein akademischen Umgebungen, steigt der Aufwand für eine Übernahme. Gleiches gilt, wenn der Generator so komplex ist, dass eine intensive Beschäftigung mit den Theorien der Textgenerierung notwendig ist. Fehlende Dokumentation oder häufige Programmabstürze tragen ebenfalls zum Aufwand bei.

Die Anforderung an die Textgeneratoren an sich ist also folgende:

- **Wiederverwendbarkeit**: Die entstehenden Textgeneratoren müssen für die Generierung von Questtexten in Spielen wiederverwendbar sein. Dies umschließt die einfache Anpassbarkeit des Programmes, eine unkomplizierte Bedienung und Stabilität.

3.3. Konzipierung der Software

Das Ziel der vorliegenden Arbeit ist, mindestens einen geeigneten Data-to-Text-Ansatz für die Generierung von Questtexten in Videospielen zu finden. In Abschnitt 3.1 wurde die Form der Eingabedaten und Ausgabetexte geklärt. Abschnitt 3.2 definiert die Anforderungen an das Programm. Das folgende Kapitel dient der Konzipierung von drei Ansätzen zur Textgenerierung unter Beachtung der festgestellten Anforderungen.

Jeder der drei Ansätze muss bestimmte Eigenschaften der generierten Texte sicherstellen: Wiedergabetreue der Eingabedaten, Konsistenz mit den Texten des Korpus und Varianz in der Formulierung. Zudem sollen die entstehenden Textgeneratoren wiederverwendbar sein.

Aufgrund ihrer Stabilität und Vorhersagbarkeit sind regelbasierte Textgenerierungsansätze eine populäre Wahl für Videospiele [vS22, S. 9]. Ihre vergleichsweise einfache Funktionsweise begünstigt zudem das Verständnis und die Anpassung des Generators. Für diese Arbeit werden darum ebenfalls regelbasierte Ansätze gewählt, da diese am Besten geeignet scheinen, die Anforderungen der Wiedergabetreue und Wiederverwendbarkeit zu erfüllen.

Modul	Aufgaben
Dokumentplanung	Daten einlesen, Inhalte bestimmen, Dokumentstrukturierung
Mikroplanung	Lexikalisierung, Aggregation
Realisierung	Realisierung

Tabelle 3.3.: Die Module des Programms und ihre Aufgaben, angelehnt an die klassische NLG-Pipeline

Implementierungsumgebung und Programmarchitektur werden so gewählt, dass das Nachvollziehen der Implementierung, und Wiederverwendung des Programmcodes in weiteren Videospielen, möglichst einfach ist.

Als Entwicklungsplattform wird "Unity" [5] genutzt. Diese wird häufig für kommerzielle Spiele eingesetzt, und die Betreiber melden 1.5 Millionen aktive monatliche Nutzer [6]. Eine Implementierung in dieser Umgebung ist demnach exemplarisch für Videospielentwicklung.

Die Architektur der Textgeneratoren wird der modularen NLG-Pipeline nachempfunden (siehe Unterabschnitt 2.3.1). Auf diese Weise wird die Wiederverwendbarkeit der einzelnen Klassen, und damit der Generatoren, erhöht. Funktionen, welche sich mehrere Ansätze teilen, müssen zudem nicht mehrmals implementiert werden. Tabelle 3.3 dokumentiert die Aufteilung des Programms.

3.3.1. Einfaches Templating

Der erste der zu vergleichenden Ansätze dieser Arbeit ist Templating (siehe Unterabschnitt 2.3.2). Das Prinzip hinter diesem Ansatz ist simpel: zunächst werden manuell Texte geschrieben und mit Ersetzungsmarkierungen versehen. Das Programm ersetzt dann nach bestimmten Regeln diese Marker durch Wörter, die die eingegebenen Daten repräsentieren. Die Syntax der Sätze, sowie die Orthographie, werden bereits durch die Templates vorgegeben. Dies bedeutet zum Einen, die Mikroplanung muss keine komplizierten Aggregation vornehmen, und kann sich auf die Lexikalisierung

[5] https://unity.com (15.03.2023)
[6] Siehe Fußnote 1

konzentrieren. Die Realisierung kann sich zudem die Schritte der Kapitalisierung und der Zeichensetzung sparen.

Dieser Ansatz wird hohen Schreib-Aufwand, aber auch große Kontrolle mit sich bringen. Die Konsistenz der Ausgabetexte mit dem Spiel und mit einander hängt stark am Autor der Template-Texte. Die Chancen für fehlerhafte Ausgaben sind dafür jedoch minimal. Ein weiterer Vorteil gegenüber pur autorengefertigten Text-Assets wird sein, dass einmal geschriebene Templates wiederverwendet werden können. Im Vergleich wird daher vorraussichtlich ein Gewinn an Varianz zu verzeichnen sein.

Verfeinertes Templating

Simples Templating bringt einen hohen manuellen Aufwand mit sich. Im zweiten Vergleichsansatz dieser Arbeit wird versucht, diesen zu verringern und gleichzeitig die Varianz weiter zu steigern. Es werden manuell Satz*elemente* mit Ersetzungsmarkern geschrieben und eine kontextfreie Grammatik mit diesen gebildet. Das Programm generiert dann eine große Vielfalt an Templates, welche wie gewohnt verwendet werden. Da die Position der Satzelemente im Vorhinein nicht klar ist, ist hier eine orthographische Behandlung in der Realisierung notwendig.

Der menschliche Autor wird hier gute Kontrolle über die Konsistenz des Textes mit dem Grundmaterial haben, wenn auch weniger als bei Ansatz 1. Das Risiko fehlerhafter Texte wird moderat sein, kann jedoch verringert werden, wenn im Vorhinein der Ergebnisraum der Template-Generierung überprüft wird. Die Varianz der Ergebnisse wächst überproportional mit dem manuellen Aufwand.

3.3.2. Oberflächenrealisierungsbibliothek SimpleNLG

Die Bibliothek “SimpleNLG“ ist eine sogenannte “realisation engine“. Sie fokussiert sich auf das Erstellen des Satzes, nachdem Entscheidungen bezüglich Wortwahl und Satzstruktur bereits getroffen wurden. Dies beinhaltet morphologische und orthographische Bearbeitung der Wörter und des Satzes sowie die Umsetzung der abstrakten Repräsentation als Text. [GR09]

Die Nutzung dieser Bibliothek für die Realisierung der Questaufträge ist der dritte Ansatz der in dieser Arbeit verglichen wird. Die Bibliothek setzt ihren Fokus

ebenfalls auf Stabilität und Wiederverwendbarkeit. Zudem bietet sie den Freiraum, Worte und Satzstruktur frei zu wählen [GR09], wodurch sie für die Anwendung in einem unkonventionellen Anwendungsfall wie einem Videospiel geeignet erscheint.

Die Bibliothek bietet drei Hauptkomponenten: ein Lexikon, verschiedene Klassen zur Erstellung einer syntaktischen Struktur eines Satzes, und den "Realizer". Um den Realizer zu verwenden, muss zunächst ein Lexikon instanziiert werden. Dieses beinhaltet die zu verwendenden Wörter und ihre Eigenschaften, zum Beispiel Beugungen für unterschiedliche Zeitformen. Im Folgenden wird dann aus Satzgliedern und Satzverbindungen eine abstrakte Definition des gewünschten Satzes erstellt. Das Lexikon und die Satzdefinition werden dem Realizer übergeben, welcher dann den realisierten Satz zurückgibt.

Die Kontrolle über die Inhalte und Form der Ausgabetexte wird bei diesem Ansatz ebenfalls hoch sein. Orthographische Fehler werden dank der Bibliothek vermieden, Fehler in der Satzstruktur können jedoch auftreten. Die Varianz der Ausgabetexte hängt von der Größe und Komplexität der vorgelagerten Verarbeitung ab. Die Konsistenz kann mit weit weniger Aufwand sichergestellt werden. Das zu implementierende Verarbeitungsmodul wird sich daher auf die Sicherstellung von Konsistenz konzentrieren.

4. Implementierung

Das folgende Kapitel dokumentiert die Implementierung der Data-To-Text-Ansätze in der Videospiel-Entwicklungsumgebung “Unity“. Abschnitt 4.1 gibt eine Übersicht über die Textgenerierungsmodule, die notwendigen Datentypen und den Ablauf der Textgenerierung. Abschnitt 4.2 beschreibt die letztendliche Umsetzung der Textgenerierungsmodule als Programmklassen im Detail.

Um die Textgeneratoren zu testen wurde eine Nutzeroberfläche angelegt. Dort getätigte Eingaben werden an eine Steuerungs-Klasse (englisch: Controller) weitergegeben, die den Generierungsprozess steuert. Ergebnisse werden von diesem Controller wiederum an die Nutzeroberfläche zurückgegeben. Dies folgt dem Vorbild der “Model-View-Controller“-Architektur[1]. Die “View“ ist die Nutzeroberfläche und das “Model“ ist die Textgenerierungslogik. Als Controller fungiert die “RealizerController“-Klasse.

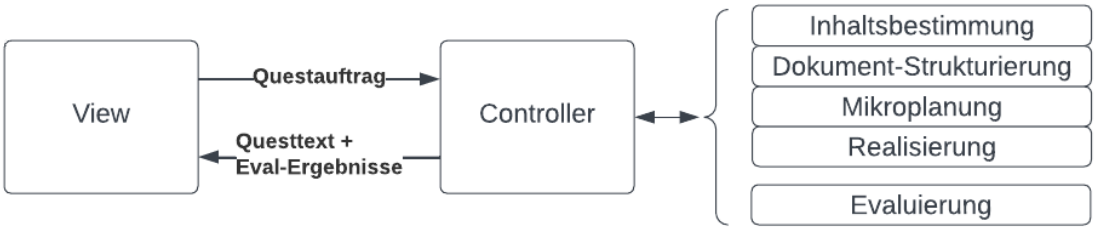

Abbildung 4.1.: Die Architektur des Programmes

[1] https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained (22.03.2023)

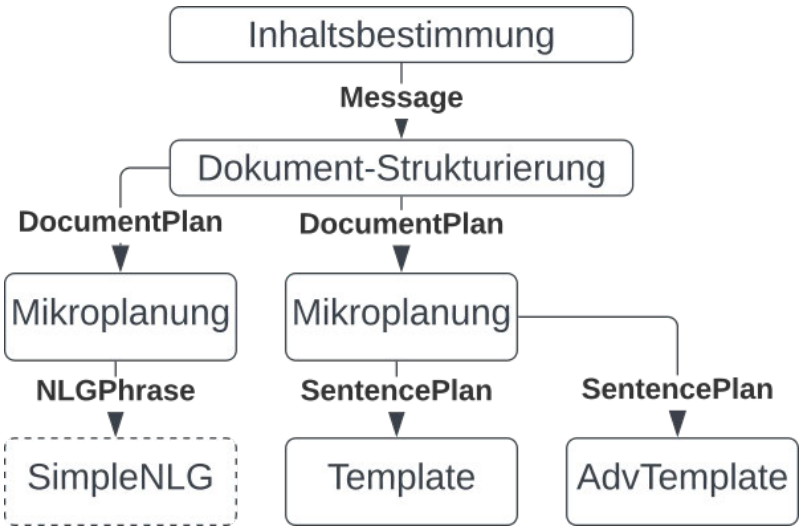

Abbildung 4.2.: Der Ablauf der Textgenerierung. Die Namen auf den Pfeilen benennen den Typ der Übergabedaten.

4.1. Übersicht

Vorbild für die Textgenerierungskomponenten ist die "Natural Language Generation Pipeline" [RD00]. Das vorherrschende Prinzip hier ist Modularität. Austauschbarkeit der einzelnen Komponenten und kleine Aufgabengebiete sind das Ziel.

Dem Vorbild der von Reiter beschriebenen Architektur folgend enthält das implementierte Programm Dokumentplanung, Mikroplanung und Realisierung. Diese werden sequentiell hintereinander ausgeführt, wobei zwischen den Modulen spezialisierte Datentypen ausgetauscht werden. Der initiale Datentyp, welcher die Eingabedaten beinhaltet, wird als "Message" (zu deutsch: "Nachricht") bezeichnet. Der Dokumentplaner extrahiert aus dieser Nachricht die nutzbaren Inhalte und ordnet sie. Der entstehende Datentyp ist ein Dokumentplan und wird dem Mikroplaner übergeben. Dessen Aufgabe ist es, neben Anderem, diese Anordnung in Satzstrukturen zu überführen und Wörter zu wählen. Der sich daraus ergebende Satzplan wird dem Realisierer übergeben, welcher den Text anhand der Regeln der Ausgabesprache generiert. Der Ausgabetyp ist in den meisten Fällen ein "String".

Bei Data-To-Text-Generierung wird der Beginn der Kette, die "Message", mit den Eingabedaten gefüllt. Dies ist Teil der Dokumentplanung, wird im vorliegenden Programm jedoch von einem eigenen Modul, der Inhaltsbestimmung, übernommen. Im

Falle dieses Projektes bestehen die Eingabedaten aus Worten oder Wortgruppen. Es soll möglich sein, beliebige Daten einzugeben und Eingabedaten für spätere Vergleiche zu speichern. Als Dateiformat für die Eingabedaten wird daher “csv“ (“comma separated value“)[2] gewählt.

Der komplette Vorgang einer Questauftrags-Realisierung ist in Abbildung 4.1 zu sehen. Zunächst werden die Eingabedaten eingelesen und die brauchbaren Inhalte bestimmt. Die instanziierte Message wird an den Dokument-Strukturierer gegeben. Dessen Aufgabe wäre, mehrere Messages in ihrer rethorischen Beziehung zueinander zu ordnen. Der Zielkorpus enthält jedoch nur Einzelsätze, welche der Ausdruck einer einzelnen Questmessage sind. Der DokumentStrukturierer wird daher nur angelegt, auch im Hinblick auf zukünftige Erweiterung, dient jedoch im vorliegenden Programm keinem Zweck.

Am Punkt der Mikroplanung divergiert das Programm. Da der SimpleNLG Realisierer eine spezialisierte Datenstruktur entgegennimmt, erhält er einen eigenen Mikroplaner. Für Templating und erweitertes Templating kann der selbe Mikroplaner verwendet werden. Die entstandene Datenstruktur wird dem jeweiligen Realisierer übergeben, welcher sie in einen englischen Satz wandelt. Dieser gibt einen String zurück, welcher den realisierten Text enthält.

Während der Implementierung hat diese Architektur die Fehlersuche enorm vereinfacht.

4.2. Implementierungsdetails

Das folgende Kapitel dokumentiert die implementierten Module und dafür verwendeten Bibliotheken und Methoden. Die verwendete Version des Unity-Editors ist 2022.2.4f1. Als Scripting Backend wurde Mono[3], mit einer .NET Kompatibilität von DotNet Standard 2.1[4] verwendet. Die Programmiersprache ist C#. Es sind keine zusätzlichen Unity-Packages notwendig.

[2] Ein nicht standardisiertes Format, in welchem Textdaten durch Kommata und Zeilenumbrüche sortiert werden

[3] https://docs.unity3d.com/Manual/Mono.html (21.03.2023)

[4] https://learn.microsoft.com/de-de/dotnet/core/introduction (21.03.2023)

Die zentrale Klasse, der “RealizerController“, ist verantwortlich für das Aufrufen der Textgenerierungsmodule mit dem korrekten Datentyp und das Entgegennehmen ihres Ergebnisses.

Dateneingabe und Inhaltsbestimmung

Tabelle 3.2 listet die Datenpunkte eines einzelnen Quest-Auftrags auf. Die Dateneingabe erfolgt im csv-Format. Für diese Implementierung werden die Daten-Felder durch Semikolons getrennt. Für “Item“ und “Enemy“ sind beliebig viele Einträge möglich, und mit jedem Eintrag sind bestimmte Eigenschaften assoziiert. Daher werden zusätzliche Teiler verwendet: das Komma trennt Einträge, und der Doppelpunkt trennt Eigenschaften des Eintrages. Der Beispielauftrag aus Tabelle 3.2 sieht in Eingabeform folgendermaßen aus:

> 14;Collect;;;Narasi Snowdawn;the Argent Tournament Grounds;Stolen Tallstrider Legs:10:;

Diese Daten liegen in Textform vor und werden als “String“-Datentyp[5] eingelesen. Das Einlesen der Rohdaten und deren Umwandlung in eine “Message“ wurde in zwei Schritte unterteilt, um die Aufgaben klarer zu gliedern. Beide Schritte werden von einer Instanz des “QuestContentDeterminator“ übernommen, dem Inhaltsbestimmer für Quests. Der erste Schritt teilt den eingegeben String anhand der Teiler[6] und assoziiert die so erhaltenen Datenfragmente mit ihrem Kontext, indem er sie den Variablen eines “QuestData“-Objektes zuweist. Der zweite Schritt nimmt ein QuestData-Objekt entgegen und erstellt daraus eine Instanz der “QuestMessage“-Klasse.

Die QuestMessage-Klasse enthält ebenfalls alle möglichen Datenpunkte der Eingabedaten, die Datentypen ihrer Felder sind jedoch auf einem höheren Abstraktionsniveau. Für die Architektur wurde eine Message-Schnittstellenklasse (“interface“) angelegt, von welcher QuestMessage erbt. Abbildung 4.3 zeigt die Repräsentation dieser Klasse, die für sie angelegten Datentypen und ihre Eigenschaften.

[5] https://learn.microsoft.com/en-us/dotnet/api/system.string?view=netstandard-2.1 (22.03.2023)

[6] https://learn.microsoft.com/en-us/dotnet/csharp/how-to/parse-strings-using-split (22.03.2023)

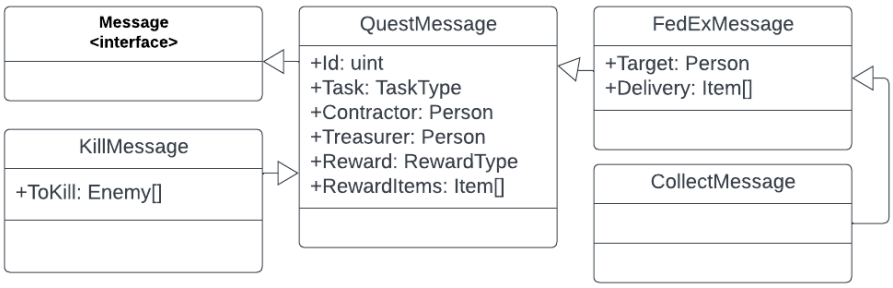

Abbildung 4.3.: UML Diagramm der Questmessage Datenstruktur[7]

Die QuestMessage wird danach einer Instanz der DokumentPlanung ("DocumentStructuring"-Klasse) übergeben. In einer vollständigen NLG-Pipeline würde hier nun eine Organisation der Message-Inhalte stattfinden. Abschnitt 4.1 erklärt, warum diese Funktionalität im vorliegenden Programm ausgelassen wird. Stattdessen wird, im Hinblick auf zukünftige Erweiterung, ein DokumentPlan mit einer Message-Liste erstellt, jedoch wird diese immer die Länge 1 haben. Der effektive Inhalt des Dokumentplans ist also die einzelne QuestMessage-Instanz.

Template-Mikroplaner

Der Template-Mikroplaner nimmt effektiv eine einzelne QuestMessage entgegen und gibt einen Satzplan zurück. Der Rückgabetyp ist eine Text-Spezifizierung ("TextSpec"), welche im Fall dieser Klasse für die Template-Generierung nutzbar sein muss. Die Aufgabe der Template-Realisierung durch Ersetzung der Template-Marker liegt beim Realisierer. Die Aufgabe der Wortwahl liegt im Mikroplaner. Der Template-Mikroplaner assoziiert also die gewählten Worte mit einem Marker in einer Datenstruktur.

In der klassischen Pipeline werden die einzelnen Satzelemente als Phrasen-Spezifizierungen ("PhraseSpec") bezeichnet. Diese Bezeichnung wurde übernommen und verschiedene PhraseSpec Klassen angelegt. Alle diese Klassen erben vom PhraseSpec Interface, und ihre Felder gewähren nur Lesezugriff. Abbildung 4.4 zeigt das UML-Diagramm dieser Datenstruktur.

"PSCannedText" Klassen enthalten eine zwar geordnete, jedoch noch zu bearbeitende Wortgruppe, während "PSOrthographicString" Klassen einen fertig formatierten

[7] Abbildung A.1.2 zeigt eine vollständigere Repräsentation

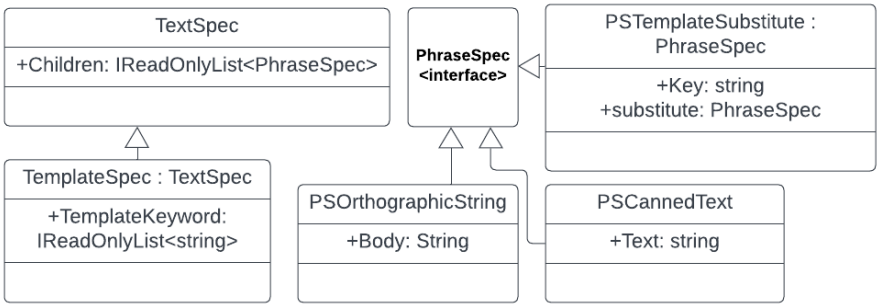

Abbildung 4.4.: Die Klassen der Phrasen-Spezifizierung

und geordneten String enthalten. Für den Zweck der Marker-Assoziation existiert die Klasse “PSTemplateSubstitute“. Diese enthält eine PhraseSpec-Instanz und einen Marker, der dem Realisierer anzeigt, an welcher Stelle diese Phrase im Template eingefügt werden muss.

Die Template-Spezifizierung enthält die Liste der zu verwendenden Phrasen sowie eine Liste an Template Kategorie-Schlüsseln. Diese sollen dem Realisierer eine Vorauswahl der Templates ermöglichen, um Suchzeiten zu verringern.

Der Code des Template-Mikroplaners definiert folgenden Ablauf:

1. Das Anlegen zweier Listen für Template-Kategorien und PhraseSpecs. Hierfür wird die .Net Datenstruktur “List<T>“ verwendet.[8]
2. Die QuestMessage wird aus dem DokumentPlan extrahiert.
3. Abhängig vom Auftragstyp werden die Felder der Questmessage auf Inhalte überprüft. Falls benötigt, wird den Template-Kategorien ein weiterer Schlüssel hinzugefügt. Der PhraseSpec-Liste werden PSTemplateSubstitute-Instanzen mit dem Inhalt der Felder und ihrem korrespondierenden Marker hinzugefügt.
4. Ein TemplateSpec-Object wird mit den Listen instanziiert und an den Aufrufer zurückggeben.

[8] https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=netstandard-2.1 (22.03.2023)

Templating - Realisierer

Der Realisierer bekommt ein "TextSpec"-Objekt übergeben, welches vom Typ "TemplateSpec" sein muss. Die Realisierer-Klasse hält zudem ein Dictionary[9], in welchem Templates nach Kategorie-Schlüsseln geordnet sind.

Zunächst werden anhand der im TemplateSpec angegebenen Kategorien aus dem Template-Dictionary alle möglicherweise geeigneten Templates ausgewählt. Unter diesen soll nun das geeignetste gefunden werden. Hierfür wird durch die Liste der geeigneten Templates iteriert. Jedes Template wird von vorne her durchschritten, und seine enthaltenen Marker darauf überprüft, ob sie bei dem aktuellen Input ersetzt werden könnten. Ist dies der Fall, wird intern eine Punktwertung ("score") hochgezählt. Unter den Templates mit dem höchsten Score wird zufällig eines ausgewählt.

Die letztendliche Markerersetzung ist dann trivial. Die System-Bibliothek von .Net bietet eine Klasse namens "StringBuilder"[10], welche neben anderen String-Manipulationen auch eine Methode zur automatischen Ersetzung von Text durch anderen Text in einem Text vornimmt. Die Liste der PSTemplateSpec's im TemplateSpec wird nun iteriert, die Ersetzung auf das Template angewandt.

Eine Erkenntnis aus der Entwicklung ist, dass spätere Änderungen an der Marker-Syntax eine äußerst aufwendige Korrektur der vorhandenen Templates nach sich ziehen. Die Marker-Systax sollte daher im Vorhinein recht ausführlich konzipiert werden.

Erweitertes Templating - Realisierer

Die Erweiterung dieser Methode gegenüber dem simplen Templating besteht in der Generierung einer großen Anzahl Template-Varianten. Der Vorgang der Marker-Substitution und TextSpec-Realisierung ist also der Gleiche wie in Abschnitt 4.2. Im folgenden soll die Generierung der Template-Varianten erklärt werden.

[9] Eine Datenstruktur, die Inhalte nach Schlüsseln sortiert - https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=netstandard-2.1 (22.03.2023)

[10] https://learn.microsoft.com/en-us/dotnet/api/system.text.stringbuilder?view=netstandard-2.1 (22.03.2023)

Dictionary<sting,string>
- <General>:
  - <Contractor> wants you to <Action>
  - <Action> Return to <Contractor> after completing this task
  - <Action>
- <Contractor>
  - <Contractor.Name>
  - <Contractor.Name> of <Contractor.Home.Name>

string[]
- speak to <Target>
- seek out <Target>
- take to <Target>

string[]
- kill <Enemy>
- slay <Enemy>

Abbildung 4.5.: Ausschnitt aus den Daten, aus welchen die Templates generiert werden.

Die freie Grammatik für die Generierung wird in Form eines Basis-Dictionary und mehrerer String-Arrays definiert, welche bei der Generierung die "<Action>"-Marker ersetzen. Der Vorteil ist, dass somit wieder Kategorien von Templates gebildet werden können. Die aus dem Basis-Dictionary und dem gewählten Action-Array generierten Templates werden in ein Template-Dictionary eingeordnet, mit einem Schlüssel, der ein weiteres String-Array ist. In diesem werden alle für die generierten Templates passenden Template-Kategorien aufgelistet.

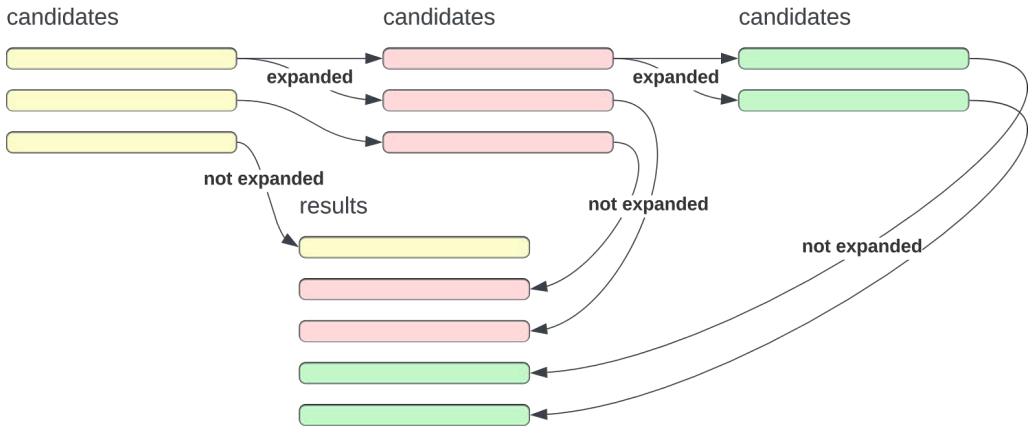

Abbildung 4.6.: Schematische Darstellung des Vorgangs der Template-Generierung beim erweiterten Templating

Abbildung 4.6 zeigt den Generierungsvorgang. Zu Beginn werden die Strings unter dem Schlüssel "General" als Kandidaten angenommen. Diese Kandidaten werden erweitert (englisch "expanded") und die Ergebnisse dieser Erweiterung in die Liste der Kandidaten aufgenommen. Kann ein Kandidat nicht weiter erweitert werden,

weil keiner der Marker des Basis-Dictionary mehr in ihm vorhanden ist, zählt er als Ergebnis ("result"). Dieser Vorgang wird rekursiv wiederholt, bis keine Kandidaten mehr vorhanden sind.

SimpleNLG

SimpleNLG ist eine Java API, welche simple Wortbeugung, Aggregation und Realisierung beherrscht. Wortwahl und Satzbau bleiben dem implementierenden Programm überlassen. Die originale Bibliothek wurde für die englische Sprache entworfen, es existieren jedoch Weiterentwicklungen für andere Sprachen, darunter Deutsch, Französisch und Spanisch. Diese sind ebenfalls in Java implementiert. Zusätzlich existieren zwei Ports der Originalbibliothek für die Programmiersprache C#.[11] Für die Realisierung bei diesem Ansatz wird die Bibliothek verwendet, für die Mikroplanung eine eigens implementierte Klasse.

Externe Bibliotheken werden in Unity als "Plug-Ins" eingebunden[12]. Dies können kompilierte Software-Pakete aller Art sein, angefangen bei C#-Dll's bis hin zu Java oder c++ Programmen[13].

Um die SimpleNLG-Java-Bibliothek in Unity zu nutzen, kann also entweder einer der C#-Ports verwendet werden oder die kompilierte Java-Bibliothek im ".jar"-Format mit einem C-basierten Wrapper. Eine dritte Möglichkeit wäre das "IKVM"-Tool[14]. Hiermit kann kompilierter Java-Bytecode in eine DotNet-Assembly konvertiert werden, um deren Funktion in einem DotNet-Projekt zu nutzen. Für dieses Projekt wurde sich für den C#-Port SharpSimple[15] entschieden. Da dieser direkt als Nuget-Package vorliegt, schien dies der einfachste und sicherste Weg zu sein.

SharpSimple ist angewiesen auf das Xml.XmlDocument-Paket[16]. Beide liegen als Nuget-Paket vor. Um sie in Unity einzubinden, werden die Pakete heruntergeladen und entpackt. In ihrer Ordnerstruktur findet sich unter "lib/netstandard2.0" eine

[11] https://github.com/simplenlg/simplenlg (23.03.2023)
[12] https://docs.unity3d.com/Manual/Plugins.html (23.03.2023)
[13] https://docs.unity3d.com/Manual/NativePlugins.html (23.03.2023)
[14] https://github.com/ikvm-revived/ikvm (23.03.2023)
[15] https://github.com/nickhodge/SharpSimpleNLG (23.03.2023)
[16] https://learn.microsoft.com/en-us/dotnet/api/system.xml.xmldocument?view=netstandard-2.1 (23.03.2023)

“.dll“-Datei. Diese Dateien werden in den Unity-Projektordner nach “Assets/Plugins/“ kopiert und können nun vom Projektcode verwendet werden.[17]

Im zuständigen Mikroplaner wird der Satzaufbau festgelegt. Während des Ablaufes wird an einem Satz-Objekt gearbeitet. Hinzugefügt werden das Subjekt (“you“), das Objekt (“Targetname“, “Itemname“, “Enemyname“) mit optionalem Post-Modifizierer (“of Placename“), ein zufällig aus einer Liste passender Worte gewähltes Verb (“gather“, “kill“) und optional ein Komplement um für Liefermissionen die Zielperson zu spezifizieren (“to Targetname“). Der Satz wird in den Imperativ-Modus gesetzt. Das Ergebnis ist dann beispielsweise: “Kill Enemy of Placename.“. Falls der Auftraggeber angegeben wurde, wird dieser gesamte Satz zum Komplement mit der Präposition “to“ eines neuen Hauptsatzes. Dieser lautet “Contractorname (Subjekt) wants (Verb) you (Objekt)“, wodurch das Ergebnis lautet: “Contractorname wants you to kill Enemyname of Placename.“.

Der aufwändigste Teil der Implementierung ist das Erstellen von Satzkonstruktionen. Sind diese erst einmal festgelegt, können Satzkonstituenten, wie Subjekt oder Verb, unproblematisch ausgetauscht werden. Möchte man jedoch an der Syntax etwas ändern, ist dies meist mit erneutem Aufwand verbunden.

[17] https://learn.microsoft.com/en-us/visualstudio/gamedev/unity/unity-scripting-upgrade (23.03.2023)

5. Evaluation

Diese Arbeit versucht die Frage zu beantworten, welcher Ansatz der Data-To-Text-Generierung für den Anwendungsfall der Questtextgenerierung in Videospielen am geeignetsten ist. Hierfür wurden Kriterien ermittelt[1], anhand derer eine solche Eignung festgestellt werden kann. Es wurde geschlussfolgert, dass regelbasierte Ansätze mit hoher Wahrscheinlichkeit geeignet sein würden, den Anforderungen an Wiederverwendbarkeit und Wiedergabetreue gerecht zu werden.

Eine Evaluation wird durchgeführt, um festzustellen, ob die Generatoren die gewünschten Eigenschaften besitzen. Zudem soll eine Aussage über den Grad der Kriterien-Erfüllung möglich sein, um zu ermitteln, welcher Ansatz am besten für den Anwendungsfall geeignet ist. Abschnitt 5.1 dokumentiert die verwendeten Evaluationsmethoden und Abschnitt 5.2 präsentiert die Ergebnisse der Evalauation.

5.1. Evaluationsmethodik

5.1.1. Methoden

Aus der Computerlinguistik sind drei Evaluationsmethoden bekannt (siehe Unterabschnitt 2.3.3): subjektive Bewertung durch menschliche Probanden, objektive Bewertung durch Experimente mit menschlichen Probanden, sowie objektive Bewertung durch Metriken [GK18, S. 66ff]. Der Nachteil der ersten beiden Methoden ist der Aufwand. Genügend menschliche Probanden zu finden, und eine möglichst neutrale Bewertung zu erhalten, benötigt Organisation, die zusätzlich zur Implementierung geschehen muss. Der Nachteil der automatischen Messungen ist, dass bestimmte Kriterien schwer oder gar nicht ermittelt werden können. So zum Beispiel Relevanz

[1] Abschnitt 3.2

des generierten Textes für den Spieler. Doch ist erst einmal eine passende Metrik gefunden, so können neue Iterationen des Programms schneller getestet werden.

Die qualitativen Eigenschaften, welche in dieser Arbeit für die Textgenerierung des Programmes gefordert sind, können zu großen Teilen automatisch durch eine Metrik ermittelt werden. Die Vorgehensweise ist zumeist, dass Überlappungen zwischen Wortgruppen der Ausgabetexte und den Texten eines Vergleichskorpus gesucht werden. Das Ergebnis trifft eine Aussage über die Ähnlichkeit der Ausgabe zu den Referenztexten. Hiermit kann effektiv die Konsistenz der generierten Texte gemessen werden. Zusätzlich kann an den Fluktuationen der Messung die Häufigkeit der Abweichung von einer einheitlichen Form beobachtet werden. Somit können Rückschlüsse auf die Varianz der generierten Texte getroffen werden. Für die Evaluation in dieser Arbeit wird die Metrik "BLEU" verwendet werden.

Da die BLEU-Werte ("scores") aufgrund von Fehlern und Varianz in der Ausgabe höchstwahrscheinlich schwanken, wird aus mehreren Durchläufen das arithmetische Mittel der BLEU-Bewertung für jeden Generator ermittelt: $\overline{BLEU} = \frac{1}{n} * \sum BLEU_n$. Ein erwartbares Ergebnis sind Werte zwischen 0,6 und 0,8. Dies zeigt an, das menschenlesbarer und adäquater Text produziert wurde. Im Optimalfall liegt der Durchschnitt sogar höher.

Mithilfe des BLEU-Mittelwertes kann die mittlere absolute Abweichung über alle Durchläufe errechnet werden: $D = \frac{1}{n} * \sum |BLEU_n - \overline{BLEU}|$. Diese gibt Auskunft über die Stärke und Häufigkeit der Fluktuationen. Ein niedriger Wert impliziert eine niedrige Varianz, und somit fehlende Abwechslung in der Formulierung der Ausgabe. Ein sehr hoher Wert deutet auf inkonsistente Ausgaben, und somit Fehlerhaftigkeit hin. Werte zwischen 0,1 und 0,2 werden als gut angesehen.

Die Eigenschaft der Wiedergabetreue entspricht der "correctness"-Qualität in der Computerlinguistik. Metriken für semantische Inhalte existieren, werden aber selten eingesetzt. Es bestehen legitime Zweifel an ihrer Effektivität [GK18, S. 70]. Die gewählten Ansätze beugen Datenverlusten während des Generierungsprozesses bereits vor. Um dies zu verifizieren, werden manuell Stichprobentests durchgeführt. Dies schließt ein Versagen bezüglich der Wiedergabetreue nicht aus, stellt sie aber ausreichend sicher.

Eine inhärente Metrik von Softwaresystemen ist die Verarbeitungsgeschwindigkeit. Es ist zu erwarten, dass diese beim zu evaluierenden Programm niedrig sein wir.

Anforderung	entsprechendes Kriterium	Überprüfungsmethode
Konsistenz	Menschenähnlichkeit, Lesbarkeit, Genre Kompatibilität	Metrik, Stichproben
Varianz	Genre Kompatibilität	Metrik
Wiedergabetreue	Korrektheit	Stichproben

Tabelle 5.1.: Auflistung der Textqualitäts-Anforderungen, Zuordnung zu ihren entsprechenden Evaluationskriterien und ihrer Überprüfungsmethode in dieser Arbeit

Es sind jedoch Differenzen zwischen den Ansätzen erwartbar. Besonders bei "live"-Generierung größerer Mengen an Texten kann die Verarbeitungsgeschwindigkeit ein wichtiger Faktor sein. Für jeden Ansatz wird die Zeit ermittelt, die er zum Generieren eines Textes benötigt. Hiermit wird keine Anforderung überprüft, sondern ein weiterer Vergleichspunkt zwischen den Ansätzen geschaffen.

Die Metrik BLEU

"BLEU" ist ein Akronym und steht für "Bilingual Evaluation Understudy". Die Metrik entworfen, um die Beurteilung von Machine Translation Modellen zu beschleunigen, und somit schnellere Entwicklungszyklen zu erlauben [PRWZ02]. Von großem Vorteil ist dabei, dass diese Metrik schnell und zudem unkompliziert umzusetzen ist. Seit ihrer Vorstellung hat sie weite Verbreitung erfahren. Es ist somit einfach, Richtwerte zu finden, mit welchen das eigene Modell verglichen werden kann. [Tat19]

Die Schnelligkeit der Metrik kommt dem Vorhaben dieser Arbeit zugute, einige Faktoren müssen dabei jedoch beachtet werden. Zunächst ist BLEU eine Metrik, die dafür geeignet ist, Lesbarkeit und Eignung eines Textes zu untersuchen. Sie ist nicht in der Lage semantische Vergleiche zu ziehen. Wie bereits im vorherigen Kapitel erläutert, wird die Metrik daher nur dazu genutzt, die Konsistenz der Ausgabetexte zu überprüfen.

Desweiteren hat sich herausgestellt, dass die Ergebnisse von BLEU schlechter als andere Metriken einer menschlichen Beurteilung entsprechen. Dies bedeutet, dass

die Werte zwar belastbar sind, aber höhere Werte keine Garantie für Konsistenz sind. Die Ergebnisse sollten daher als Trend aufgefasst werden und für eine weitere Entwicklung eine belastbarere Metrik gewählt werden.

Die Funktionsweise von BLEU entspringt einer verbreiteten Vorgehensweise in der Textanalyse. Hierbei wird ein Text in einzelne Worte zerlegt, und n aufeinanderfolgende Worte werden in Gruppen zusammengefasst. Diese Wortgruppen werden als N-Gramm bezeichnet [2]. Für Gruppen von zwei Wörten existiert die Bezeichnung Bigramm, bei drei Wörten ist es ein Trigramm.

BLEU ermittelt, wie oft ein N-Gramm eines generierten Satzes ("candidate") in Referenzsätzen auftaucht ("covered count") und beschneidet die Anzahl um die maximale Wiederholung des Gramms in den Referenzen ("clipped count"). Die Summe der "clipped counts" wird duch die Gesamtanzahl der N-Gramms des Satzes geteilt, wodurch ein vorläufiger Präzisionswert entsteht. [PRWZ02]

$$p_n = \frac{\sum_{n-gram \in candidate} ClippedCount(n-gram)}{\sum_{n-gram \in candidate} Count(n-gram)} \tag{5.1}$$

BLEU kombiniert mehrere N-Gramm-Präzisions-Wertungen mithilfe des gewichteten geometrischen Mittels. Die empfohlenen N-Gramms sind 1 bis 4, und die Wichtungen ergeben in Summe 1: $w_n = \frac{1}{n}$. Zusätzlich wird dieses Ergebnis mit einer Knappheits-Strafe ("brevity penalty") versehen, da zu kurze Ergebnistexte sonst eine unverdient hohe Wertung hätten. [PRWZ02]

$$BLEU = BP * \exp(\sum_{n=1}^{n} w_n \log p_n) \tag{5.2}$$

Stichproben

Stichproben werden manuell durchgeführt und ergeben eine binäre Wertung für einen Satz: Erfolg oder Misserfolg. Hierfür werden die Eingabedaten mit dem Ergebnissatz verglichen und es wird überprüft, ob alle notwendigen Informationen wiedergegeben wurden.

[2] https://web.stanford.edu/ jurafsky/slp3/ (16.03.2023)

Geschwindigkeitsmessung

Abgesehen von Mikroplaner und Realisierer sind die Programmschritte der Generierung für alle Textgenerierungsansätze geich. Zur Messung der Verarbeitungsgeschwindigkeit wird daher folgendermaßen vorgegangen: Vor dem Start eines Mikroplaners wird eine Zeitmarke festgehalten. Nach der Beendigung seines Realisierers wird die Differenz dieser Marke mit der aktuellen Zeit errechnet. Dies ist die vergleichbare Verarbeitungsgeschwindigkeit des Ansatzes ΔT. Sie wird in Sekunden angegeben.

5.1.2. Durchführung

Dieses Unterkapitel formalisiert den Verlauf eines Bewertungsdurchlaufs für alle 3 Texgenerierungsansätze. Die Ausgabe der Textgenerierung hängt stark von den Eingabedaten ab, erzeugt aber auch bei gleicher Eingabe variable Ausgaben. Um beide Phänomene betrachten und bewerten zu können, werden die Testeingabedaten ein Set von Questaufträgen beinhalten, und jeder Textgenerator wird für jeden individuellen Questauftrag mehrmals ausgeführt.

Ein Testdurchlauf besteht also aus k Datensätzen und n Ausführungen je Textgenerator $tmp, advTmp, SimpNLG \in Gen$. Für jeden Datensatz wird die Realisierungsmethode $Gen.realize(K)$ genau n mal ausgeführt. Für jedes so entstehende Ergebnis wird der BLEU-Wert $BLEU$ berechnet und die Ausführungsgeschwindigkeit ΔT erfasst. Zudem werden pro Datensatz 2 Stichproben in den Ergebnissen jedes Generators vorgenommen.

Für jeden Generator werden dann der durchschnittliche BLEU-Wert $\overline{BLEU}$, die mittlere absolute Abweichung D und die durchschnittliche Ausführungsgeschwindigkeit $\overline{\Delta T}$ berechnet.

$$\overline{BLEU}_{Gen} = \frac{1}{n * k} * \sum_{i=0}^{n*k} BLEU_i \tag{5.3}$$

$$D_{Gen} = \frac{1}{n * k} * \sum_{i=0}^{n*k} |BLEU_i - \overline{BLEU}| \tag{5.4}$$

$$\overline{\Delta T}_{Gen} = \frac{1}{n * k} * \sum_{i=0}^{n*k} \Delta T_i \tag{5.5}$$

5.2. Ergebnisse

5.2.1. Ermittelte Werte

In diesem Kapitel werden die Ergebnisse eines individuellen Testlaufs präsentiert. Dies ist nicht der einzige durchgeführte Testlauf, die Ergebnisse sind jedoch repräsentativ für in anderen Testläufen erhobenen Daten.

Die Konfiguration des Durchlaufs bestand aus einem Datensatz mit $k = 3$ Aufträgen, $n = 20$ Wiederholungen je Auftrag, somit insgesamt 60 Einzelergebnissen. Im Datensatz waren alle 3 Questauftrags-Typen vertreten. Die einzelnen Questaufträge sind gekennzeichnet mit den Indizes 0, 1 und 2. Die entstandenen Werte zeigt Tabelle 5.2.

Generator	$\overline{BLEU}$	Deviation D				$\overline{\Delta T}$ in s
		0	1	2	$\overline{D}$	
tmp	0,80027	5,55E-16	0,0268141	3,33E-16	0,11901	0,00005
advTmp	0,80329	0,156746	0,0240633	0,0364819	0,07663	0,00528
simpNLG	0,86195	5,55E-16	0,014826	0	0,06902	0,00158

Tabelle 5.2.: Mittelwerte des Testdurchlaufes. Die mittlere absolute Abweichung innerhalb eines Questauftrags wird unter seinem entsprechenden Index angezeigt.

Die durchschnittlichen BLEU-Scores sind allesamt hoch, was auf eine gelungene Ähnlichkeit zu den Zielausgaben hinweist. Die Varianz ist niedrig, aber unerwarteterweise beim Templating am höchsten. Eine Berechnung der Abweichungswerte innerhalb von Ergebnissen mit gleichem Auftrag bringt mehrere gegen 0 tendierende Werte hervor. Eine Betrachtung der BLEU-Werte ebendieser Ergebnisse (siehe Abbildung 5.1) zeigt enorme Abweichungen zwischen unterschiedlichen Questauftrags-Ausgaben. Die mittlere Abweichung aller Ergebnisse gibt somit eher einen Einblick

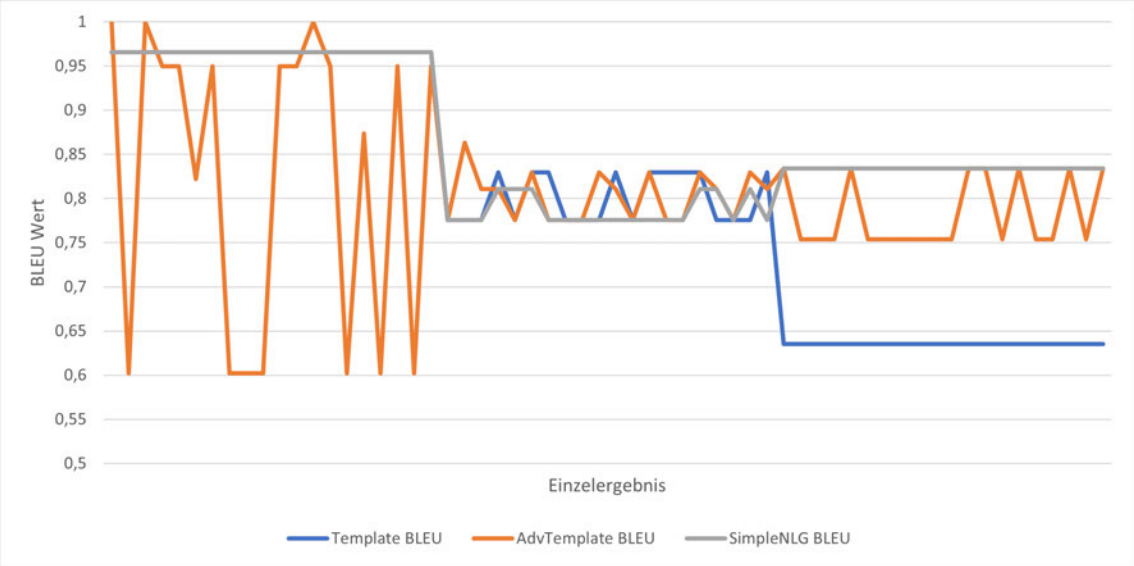

Abbildung 5.1.: Visualisierung der BLEU-Werte des Testdurchlaufes. Tabelle B.1.1 zeigt einen Ausschnitt der rohen Werte.

darauf, wie groß Unterschiede zwischen Questaufträgen sind. Für die Bewertung der Varianz wird daher stattdessen die Abweichung der Ergebnisse eines Auftrags herangezogen.

Tabelle 5.3 zeigt die Stichproben für den Questauftrag 0. Es ist gut zu erkennen, dass das Ziel des Auftrages und die notwendigen Informationen zur Erledigung (“Was“, “Wie viele“), sowie Namen vorhanden sind. In den Ausgaben zu Auftrag 2 sind jedoch Fehler aufgetreten. Der folgende Satz ist eine Ausgabe des Template Generators für Eingabe 2:

Bring 0 Vek’lor’s Diadem,2 Idols of Life and 5 Clay Scarabs to Andorgos.

In den Eingabedaten fehlt eine Nummerierung von “Vek’lor’s Diadem“, wodurch in der Ausgabe eine unklare Anweisung entsteht. Der Generator besitzt keine Regel, die beim Fehlen einer Anzahl automatisch annimmt, dass genau 1 Exemplar des Gegenstandes gemeint ist. Die Ausgaben aller drei Generatoren beinhalten diesen Fehler. Die Anforderung der Wiedergabetreue wurde erfüllt, doch der Satz ist fehlerhaft. Ohne eine menschliche Überprüfung wäre dies nicht aufgefallen.

Eingeabedaten	0;Kill;Astrid Bjornrittar;;;;;Ravenous Jormungar:8:the Hibernal Cavern east of Brunnhildar Village
Templating	Kill 8 Ravenous Jormungar in the Hibernal Cavern east of Brunnhildar Village.
	Kill 8 Ravenous Jormungar. Return to Astrid Bjornrittar after completing this task.
Verfeinertes Templating	Kill 8 Ravenous Jormungar. Return to Astrid Bjornrittar after completing this task.
	Astrid Bjornrittar wants you to kill 8 Ravenous Jormungar.
SimpleNLG	Slay 8 Ravenous Jormungar in the Hibernal Cavern east of Brunnhildar Village.
	Kill 8 Ravenous Jormungar in the Hibernal Cavern east of Brunnhildar Village.

Tabelle 5.3.: Stichproben aus den Resultaten des Testdurchlaufs

Das Templating hat die niedrigste BLEU-Wertung erhalten. Bezüglich Varianz liegt es in der der Mitte, hat aber die größte Abweichung zwischen Questaufträgen. Die Verarbeitungsgeschwindigkeit ist bei Weitem höher als die der anderen Ansätze.

Platz zwei in der BLEU-Wertung ist das erweiterte Templating, jedoch nur mit einem Vorsprung von 0,003. Es hat die höchste Gesamtabweichung und liegt bei der durchschnittlichen Abweichung zwischen den anderen Ansätzen. Die Verarbeitungsgeschwindigkeit ist mit signifikantem Vorsprung die niedrigste. Der Grund hierfür ist vermutlich die rechenaufwändige Iteration der generierten Templates.

Der dritte Ansatz, die Nutzung der “realisation engine“ SimpleNLG, erzielt die höchste BLEU-Wertung, mit einem Abstand von 0,06 vor den anderen Ansätzen. Die Varianz ist die Niedrigste der 3 Textgeneratoren. Gleichzeitig sind aber auch die Unterschiede zwischen Questaufträgen am geringsten. Die durchschnittliche Verarbeitungsgeschwindigkeit liegt bei 1,5 Millisekunden, womit der Ansatz der Zweitschnellste ist.

5.2.2. Schlussfolgerungen und Probleme

Für den Anwendungsfall der Questtexterstellung ist ein Textgenerator genau dann geeignet, wenn die damit Texte in gleicher oder höherer Qualität als handgeschriebene Texte generiert werden können und der für den Generator eingesetzte Aufwand geringer ist als beim manuellen Schreiben.

Die qualitativen Anforderungen erwachsen aus den inhärenten Zielen von Questtexten. Sie sollen das Spielerlebnis unterstützen, dem Spieler die Anforderungen klar präsentieren, Hinweise auf die Erzählung beinhalten und abwechslungsreich sein. In dieser Arbeit wurden die Anforderungen als Konsistenz, Varianz und Wiedergabetreue formuliert.

Da die Ansätze nun bereits implementiert sind, fällt für eine erneute Nutzung der Schritt der Konzeption weg. Der eingesetzte Arbeitsaufwand bezieht sich also nur auf die Anpassung des vorliegenden Programmes auf eine neue Anwendung. Dies beinhaltet, zusätzlich zu den individuellen Anpassungen die ein Ansatz benötigt, das Schreiben eines Inhaltsbestimmers, um die nativen Daten des Spiels in eine für das Textgenerierungssystem nutzbare "Message" umzuwandeln.

Im Folgenden wird jeder Generierungsansatz kurz betrachtet. Alle Ansätze werden anhand der Evaluationsergebnisse auf die Anforderungen überprüft und es wird eine auf die Erkenntnisse der Implementierung gestützte Vermutung über die Höhe des Anpassungsaufwands geäußert. Annahmen aus der Konzeption werden bestätigt oder widerlegt. Letztlich wird aus diesen Faktoren auf die Eignung jedes Textgenerators geschlussfolgert.[3]

Templating

Die Stabilität und Vorhersagbarkeit des Templating hat sich bewiesen. Die Anforderung an die Wiedergabetreue wurde erfüllt. Die Nähe zum Stil der Vergleichstexte ist in Ordnung. Bestimmte Eingaben führen jedoch zu fehlerhaften Ausgaben und der Abwechslungsreichtum der Texte ist gering. Diese letzten beiden Faktoren werden durch die geringe Auswahl an an Templates und Regeln bedingt. Die Leistung könnte

[3] Diese Erkenntnisse beziehen sich auf die Implementierungen dieser Arbeit und sind daher nicht uneingeschränkt generalisierbar.

mit einem moderaten Arbeitsaufwand verbessert werden, indem der Mikroplaner abgeändert und weitere Templates angelegt werden würden.

Der Aufwand, den Ansatz in einer neuen Umgebung einzusetzen, ist vorraussichtlich niedrig. Die dafür einfachste Methode, das Anlegen neuer Templates, ist ein unkomplizierter, intuitiver Vorgang. Kleine Änderungen am Regelsatz sind auch ohne großen Aufwand realisierbar. Mit steigenden Ansprüchen an Funktionsumfang und Qualität steigt der Aufwand jedoch überproportional an, da jede Kombination von Eingabedaten ein passendes Template benötigt.

Insgesamt kann eine Eignung des Ansatzes für Quest-"Objectives" oder ähnlich geartete Questtexte festgestellt werden. Es steht jedoch zu vermuten, dass es sehr aufwändig wäre komplexere Questtexte mit zuätzlicher Verknüpfung zu einer Erzählung mit diesem Ansatz auszudrücken.

Erweitertes Templating

Die Kontrolle über das Ergebnis ist bei diesem Ansatz wie erwartet gut. Die entstehenden Texte sind wiedergabetreu. Die Konsistenz mit dem angestrebten Stil ist sehr gut, genau wie auch die Varianz der Texte. Wie beim Templating treten auch hier Fehler auf, die die Klarheit der Anforderungen der Quest an den Spieler unklar machen.

Der Anpassungsaufwand dieses Ansatzes ist vorraussichtlich niedrig. Der einzige notwendige Schritt ist das Anpassen der Template-Grammatik. Da dieses Vorgehen ein *indirektes* Formulieren der Ausgabetexte ist, benötigt es etwas mehr initialen Aufwand als manuelles Schreiben, um den richtigen Ergebnisraum zu finden. Für eine längere Entwicklungszeit ist es jedoch um einiges Wartbarer und Anpassbarer, und holt somit den Aufwand wieder herein. Das Erweitern des Mikroplaners um neue Regeln ist einfach, und auch die Anpassung an speziellere Fälle ist ohne Probleme möglich. Für eine größere Skalierung des Ansatzes ist jedoch zusätzlicher Aufwand nötig, möchte man die Verarbeitungsgeschwindigkeit niedrig halten.

Der Ansatz des Templatings mit per Grammatik generierten Templates ist ausgezeichnet für Quest-"Objectives" und ähnliche Texte geeignet. Dank der unaufwendigen Erweiterbarkeit ist vorstellbar, dass auch Systeme mit unterliegender Erzählung ihren Ausdruck durch diesen Ansatz finden können.

SimpleNLG

Die "realization engine" SimpleNLG erlaubt ebenfalls, wie erwartet, gute manuelle Kontrolle über die Ausgabe. Zudem führen Fehler in ihrer Bedienung nicht zu Abstürzen des Programms, sondern nur zu fehlerhaften Ausgaben. Diese sind orthographisch oder grammatikalisch inkorrekt, beinhalten jedoch alle notwendigen Daten. Die Anforderung der Wiedergabetreue wird von diesem Ansatz erfüllt. Die Konsistenz in Stil und Formulierung ist ausgezeichnet, jedoch fehlt es den Texten an Abwechslung.

Der Aufwand der Anpassung des Ansatzes ist moderat, aber keinesfalls zu unterschätzen. Das Problem ist, dass zu diesem Zweck der Mikroplaner komplett umgeschrieben werden muss, um die gewünschte, neue Satzstruktur hervorzubringen. Zudem ist die Kenntnis grammatischer Theorie eine Vorraussetzung für diese Aufgabe. Der Aufwand ist für die Questtexte dieser Arbeit nur leicht niedriger als der des manuellen Schreibens. Zwar können einmal geschriebene Regeln mit immer neuen Daten wiederverwendet werden. Neue Formulierungen oder Satzstrukturen müssen jedoch immer wieder neu *programmiert* werden, wodurch ein ähnlicher Aufwand wie für das manuelle Schreiben entsteht.

Die Bibliothek ist für das Ziel, Quest-"Objective" Texte zu generieren zwar geeignet, bringt jedoch einen hohen Anpassungsaufwand mit sich. Dafür besitzt der Ansatz jedoch ein großes Potential. Mithilfe eines ausgefeilten Mikroplaners, der komplexe, narrative Strukturen in mehrzeilige Phrasen umwandeln kann, könnte dieser Ansatz Questtexte mit Narrativen generieren.

Fazit

Die qualitativen Anforderungen werden von allen Generatoren erfüllt. Dies bedeutet, dass das Risiko fehlerhafter oder unpassender Texte minimiert wurde. Bei Kosten-Nutzen-Abwägungen, welche vorher im Gleichgewicht gewesen wären, würde nun der Nutzen überwiegen. Aus diesem Grund wird geschlossen, dass alle drei Ansätze für die Generierung von Questtexten in Videospielen geeignet sind. Tabelle 5.4 zeigt einen zusammenfassenden Vergleich der Ansätze.

Im folgenden Absatz soll die Frage nach dem geeignetsten Data-To-Text-Ansatz, und somit die Forschungsfrage dieser Arbeit, beantwortet werden. Wie sich zeigt, kann

	Tmp	AdvTmp	SimpleNLG
Konsistenz	+	++	+++
Varianz	+	+++	+
Wiedergabetreue	X	X	X
Geschwindigkeit	+++	+	++
Anpassungsaufwand	+++	++	+
Erweiterungspotential	+	++	+++

Tabelle 5.4.: Vergleich der Textgenerierungsansätze

kein Ansatz in einer Mehrheit der Vergleichskriterien die Spitze beanspruchen. Dies ergibt Sinn, da Vorteile in einer Kategorie sich nachteilig auf eine andere Kategorie auswirken können. Eine höhere Konsistenz führt leicht zu einer verminderten Varianz. Die Konsistenz der Texte des erweiterten Templatings ist nur wenig unter dem in dieser Arbeit ermittelten Bestwert, die Varianz ist jedoch ausgezeichnet und weit höher als die der anderen Ansätze. Im Vergleich besitzt das erweiterte Templating die beste Kombination aller beobachteten Merkmale. Der unter den betrachteten Ansätzen am besten geeignete Ansatz für Questtext-Generierung in Videospielen ist Templating mit aus einer kontextfreien Grammatik generierten Templates.

6. Zusammenfassung und Ausblick

Das Potential der Textgenerierung für Videospiele liegt einerseits in der Einzigartigkeit der mit ihrer Hilfe umsetzbaren Mechaniken, aber auch in der Möglichkeit, das repetetive Schreiben von Texten zu verringern. Die Frage, welche diese Arbeit versucht zu beantworten, ist, welcher Ansatz der Data-To-Text-Generierung für das Generieren von Questtexten in Videospielen am Besten geeignet ist. Das Ziel der Implementierung war, für diesen Anwendungsfall wiederverwendbaren Code zu kreieren.

Ausgehend von dem Stand der Textgenerierung in der Videospielentwicklung, den Eigenschaften von Questtexten und der verfügbaren Technologie im Bereich der Data-To-Text-Generierung,[1] wurde zuerst auf wünschenswerte Eigenschaften der Textgeneratoren geschlossen: Konsistenz, Varianz und Wiedergabetreue und Wiederverwendbarkeit.[2] Um den besten Ansatz zu ermitteln, wurden 3 Ansätze der Textgenerierung implementiert, welche aus non-linguistischen Questaufträgen Questtexte in Form kurzer Aufforderungssätze generieren. Diese sind Templating, erweitertes Templating mit via kontextfreier Grammatik generierten Templates und die Nutzung der "realization engine" SimpleNLG.

Das implementierte System ist in der Lage, Questaufträge in Form von csv-Strings einzulesen und dessen Inhalte zu bestimmen. Daraufhin planen die jeweiligen Ansätze die syntaktische Anordnung der Inhalte in einem Mikroplaner und realisieren diese als vollständigen Satz.[3]

Mithilfe der Metrik "BLEU", welche die generierten Texte mit dem Ausgangskorpus vergleichen kann, wurde eine Evaluierung vorgenommen. Die Ergebnisse daraus zeigen, dass alle gewählten Ansätze für diesen Anwendungsfall geeignet sind.

[1] Kapitel 2
[2] Kapitel 3
[3] Kapitel 4

Die Nachteile der jeweiligen Ansätze sind wie folgt: Templating produziert eine geringe Varianz zwischen einzelnen Texten. Die SimpleNLG-Mikroplaner-Klasse müsste komplexer sein um variantenreiche Texte zu generieren, und der Anpassungsaufwand für neue Ausgabetexte ist relativ hoch. Das erweiterte Templating produziert zwar sehr variantenreiche Texte, dies jedoch auf Kosten der Verarbeitungsgeschwindigkeit, welche entsprechend hoch ist.

Letztendlich scheint der Ansatz des Erweiterten Templatings am Besten geeignet für Questtextgenerierung in der Videospielentwicklung. Er bietet eine gute Balance zwischen Konsistenz und Varianz. Der Anpassungsaufwand ist niedrig, und der Ansatz ist für komplexere Anwendungsfälle einsetzbar. Dieses Ergebnis spiegelt die Erkenntnisse der Recherche wieder. Mehrere Spiele und viele Einzelanwendungen, wie "Twitter-Bots", nutzen die Autoren-Werkzeuge "Expressionist" und "Tracery", welche diesen Ansatz verwenden. Besonders für die Generierung einfacher, sich oft wiederholender Text-Schnipsel ist die vorhandene Implementierung vorzüglich geeignet.

Die Ergebnisse der Metrik-Evaluierung sind jedoch kein abschließender Indikator für die Leistung der implementierten Ansätze. Bestimmte Fehler konnten von der Evaluationsmetrik nicht erkannt werden. Diese ist zudem dafür bekannt, nicht immer mit menschlicher Beurteilung von Texten zu korrelieren. Eine regelmäßige menschliche Überprüfung der entstehenden Texte in Form von Stichprobentests ist erforderlich. Zudem schwanken die entstandenen Bewertungen stark zwischen einzelnen Questaufträgen. Dies sollte bei zukünftiger Anwendung berücksichtigt werden, und bedarf möglicherweise weiterer Entwicklungsarbeit.

6.1. Ausblick

Mit den in dieser Arbeit implementierten und dokumentierten Ansätzen ist es ohne großen Aufwand möglich, beliebige repetetive Texte zu generieren. Sie sind nicht auf Quest-"Objectives" beschränkt. Mögliche Anwendungsfälle sind NPC-Beschreibungen oder Status-Updates in einer dynamischen Welt. Keine der vorgestellten Methoden ist abhängig von einem Korpus an Beispieltexten, und sie können unabhängig von existierenden Korpora arbeiten. Daher können, mit Ausnahme von SimpleNLG, auch Texte in anderen Sprachen generiert werden. Für den speziellen Fall der Generierung

deutscher Questtexte könnte die Eignung der deutschen Adaption der SimpleNLG-Java-Bibliothek "SimpleNLG-DE" [BKSM19] für einen Einsatz in Unity getestet werden. Abschnitt 4.2 listet mögliche Methoden der Einbindung auf.

SimpleNLG bietet außerdem das Potential, Teil einer umfangreichen Narrativ - Generierung zu sein. Eine überlegte Architektur mit automatischer Syntax-Wahl könnte möglicherweise anspruchsvolle, mehrparagraphige Datenzusammenhänge ausdrücken. So könnte zu den eigentlichen Quest-Daten noch der Zustand der Spielwelt und die Verbindung der Questinhalte zur Hauptnarrative enthalten sein.

Ein letztes, unbeachtetes Seitengebiet ist die Generierung von Quest-Beschreibungen für die Nutzermenüs. Dies wäre sowohl mithilfe des Erweiterten Templating, als auch der "realization engine" SimpleNLG möglich.

Auch wenn der Trend in der Videospielentwicklung eher auf die Nutzung von Machine-Learning Modeln zusteuert [vS22] [Mys] [4], so liegt dennoch ein großes Potential in den regelbasierten Ansätzen. Vor allem kleine Entwicklerstudios oder Einzelentwickler können von Erkenntnissen aus diesem Feld profitieren.

[4] https://www.videogameschronicle.com/news/ubisoft-is-developing-an-ai-tool-that-aims-to-support-scriptwriters (23.03.2023)

Literaturverzeichnis

[BKSM19] Daniel Braun, Kira Klimt, Daniela Schneider und Florian Matthes: *SimpleNLG-DE: Adapting SimpleNLG 4 to German*, in *Proceedings of the 12th International Conference on Natural Language Generation*, S. 415–420, Association for Computational Linguistics, Tokyo, Japan, Okt. 2019.

[CKM15] Kate Compton, Ben Kybartas und Michael Mateas: *Tracery: An Author-Focused Generative Text Tool*, in *Interactive Storytelling* (herausgegeben von Henrik Schoenau-Fog, Luis Emilio Bruni, Sandy Louchart und Sarune Baceviciute), Lecture Notes in Computer Science, S. 154–161, Springer International Publishing, Cham, 2015, ISBN 978-3-319-27036-4.

[Eug10] Joseph Eugene: *Bot Colony – a Video Game Featuring Intelligent Language-Based Interaction with the Characters*, 2010.

[Fou16] *Found in Translation: More Accurate, Fluent Sentences in Google Translate*, https://blog.google/products/translate/found-translation-more-accurate-fluent-sentences-google-translate/, Nov. 2016.

[FUR+20] Angela Fan, Jack Urbanek, Pratik Ringshia, Emily Dinan, Emma Qian, Siddharth Karamcheti, Shrimai Prabhumoye, Douwe Kiela, Tim Rocktaschel, Arthur Szlam und Jason Weston: *Generating Interactive Worlds with Text*, *Proceedings of the AAAI Conference on Artificial Intelligence*, Bd. 34(02):S. 1693–1700, Apr. 2020, ISSN 2374-3468.

[GK18] Albert Gatt und Emiel Krahmer: *Survey of the State of the Art in Natural Language Generation: Core Tasks, Applications and Evaluation*, Jan. 2018.

[GR09] Albert Gatt und Ehud Reiter: *SimpleNLG: A Realisation Engine for Practical Applications*, in *Proceedings of the 12th European Workshop on Natural Language Generation (ENLG 2009)*, S. 90–93, Association for Computational Linguistics, Athens, Greece, März 2009.

[HW] Filip Hracek und Alec Webb: *Knights of San Francisco — Egamebook*, https://filiph.net/.

[Kre] Max Kreminski: *Epitaph by Max Kreminski*, https://mkremins.itch.io/epitaph.

[Mys] Jakub Mysliwiec: *short RPG side-quest story generator - google collab*, https://jakub.thebias.nl/research/QuestGen/colab/.

[Mys20] Jakub Mysliwiec: *Generating Short RPG Side-Quest Stories with Transformers*, S. 8, Okt. 2020.

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward und Wei-Jing Zhu: *Bleu: A Method for Automatic Evaluation of Machine Translation*, in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, S. 311–318, Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, Juli 2002.

[RD00] Ehud Reiter und Robert Dale: *Building Natural Language Generation Systems*, Studies in Natural Language Processing, Cambridge University Press, Cambridge, 2000, ISBN 978-0-521-62036-9.

[RSMW16] James Ryan, Ethan Seither, Michael Mateas und Noah Wardrip-Fruin: *Expressionist: An Authoring Tool for In-Game Text Generation*, Nov. 2016, ISBN 978-3-319-48278-1.

[Sal] Andres Saladrigas: *Quest-Design*, https://thearzhel.github.io/Quest-Design.

[Tat19] Rachael Tatman: *Evaluating Text Output in NLP: BLEU at Your Own Risk*, https://towardsdatascience.com/evaluating-text-output-in-nlp-bleu-at-your-own-risk-e8609665a213, Juli 2019.

[Tha18] Thalefeather: *What Are the Fundamentals of Quest Design?*, www.reddit.com/r/gamedesign/comments/8ua23p/what_are_the_fundamentals_of_quest_de Juni 2018.

[vS22] Judith van Stegeren: *Flavor Text Generation for Role-Playing Video Games*, März 2022.

[vST21] Judith van Stegeren und Mariet Theune: *Fantastic Strings and Where to Find Them: The Quest for High-Quality Video Game Text Corpora*, in *Proceedings of the 12th Intelligent Narrative Technologies (INT) Workshop*, CEUR, Mai 2021.

[Wie21] *Wie Textautomatisierung BR Sport unterstützt*, https://www.br.de/nachrichten/sport/wie-textautomatisierung-br-sport-unterstuetzt, Febr. 2021.

Anhang

A. Implementierungsdokumente

A.1. Implementierungsdetails

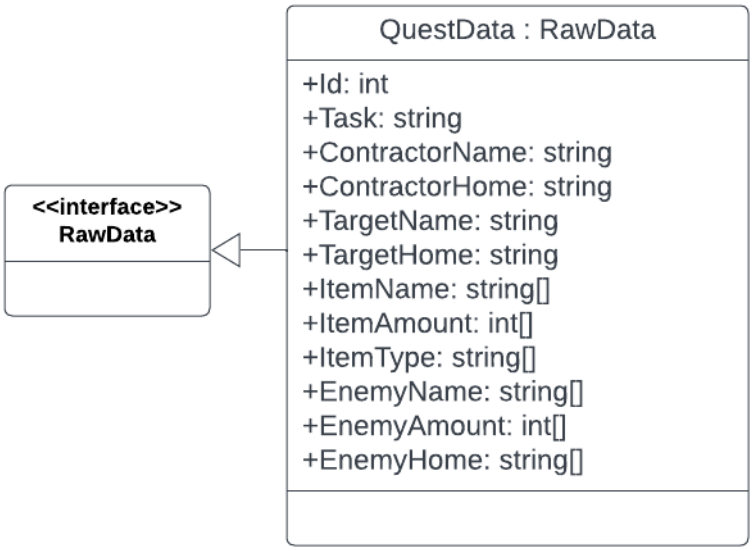

Abbildung A.1.1.: UML Diagramm der QuestData Datenstruktur

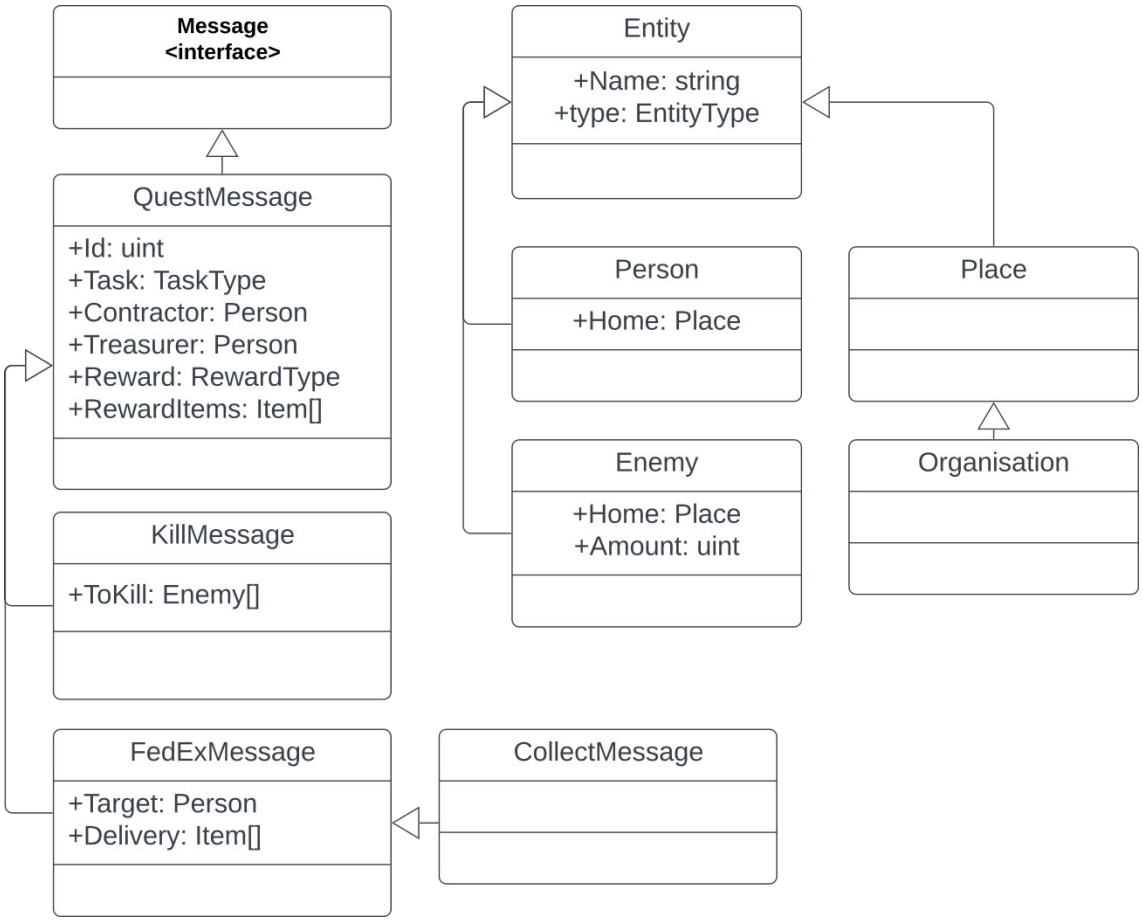

Abbildung A.1.2.: UML Diagramm der Questmessage Datenstruktur

B. Ergebnisdokumente

B.1. Evaluationsdaten

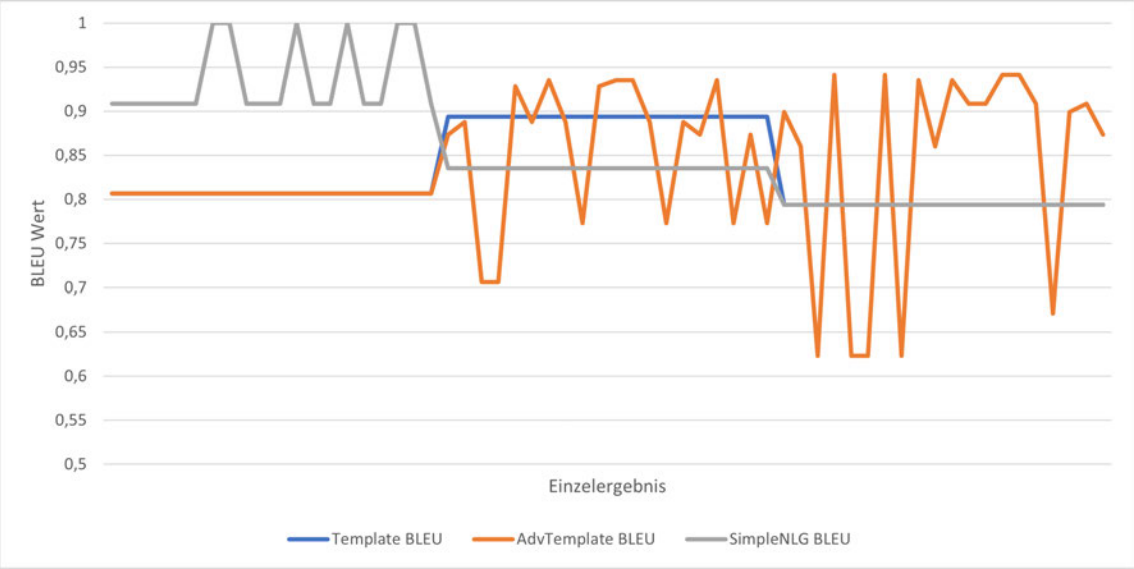

Abbildung B.1.1.: Abbildung der Resultate des Testdurchlaufes 04 vom 20.03.2023

	Template		AdvTemplate		SimpleNLG	
Auftrag	BLEU	DeltaT	BLEU	DeltaT	BLEU	DeltaT
0	0,96549	6,00E-05	1	0,00192	0,96549	0,00102
0	0,96549	3,00E-05	0,60201	0,0019	0,96549	0,00091
0	0,96549	4,00E-05	1	0,00191	0,96549	0,00089
0	0,96549	3,00E-05	0,94991	0,00192	0,96549	0,00089
0	0,96549	3,00E-05	0,94991	0,0019	0,96549	0,00087
0	0,96549	3,00E-05	0,82207	0,00198	0,96549	0,00091
0	0,96549	3,00E-05	0,94991	0,00192	0,96549	0,00089
0	0,96549	3,00E-05	0,60201	0,00194	0,96549	0,0009
0	0,96549	3,00E-05	0,60201	0,00192	0,96549	0,00089
0	0,96549	3,00E-05	0,60201	0,00191	0,96549	0,00089
1	0,77566	7,00E-05	0,77566	0,00738	0,77566	0,00185
1	0,77566	6,00E-05	0,86305	0,00728	0,77566	0,0019
1	0,77566	6,00E-05	0,81096	0,00733	0,77566	0,00189
1	0,82983	6,00E-05	0,81096	0,00724	0,81096	0,00182
1	0,77566	6,00E-05	0,77566	0,00722	0,81096	0,00186
1	0,82983	7,00E-05	0,82983	0,00716	0,81096	0,00186
1	0,82983	6,00E-05	0,77566	0,00737	0,77566	0,00184
1	0,77566	6,00E-05	0,77566	0,00729	0,77566	0,00182
1	0,77566	6,00E-05	0,77566	0,0072	0,77566	0,00186
1	0,77566	6,00E-05	0,82983	0,00752	0,77566	0,00183
2	0,6353	7,00E-05	0,83413	0,00666	0,83413	0,00211
2	0,6353	7,00E-05	0,75395	0,00659	0,83413	0,002
2	0,6353	7,00E-05	0,75395	0,00663	0,83413	0,00198
2	0,6353	7,00E-05	0,75395	0,00701	0,83413	0,00198
2	0,6353	7,00E-05	0,83413	0,00674	0,83413	0,002
2	0,6353	6,00E-05	0,75395	0,00671	0,83413	0,00198
2	0,6353	6,00E-05	0,75395	0,00672	0,83413	0,002
2	0,6353	7,00E-05	0,75395	0,00679	0,83413	0,00196
2	0,6353	7,00E-05	0,75395	0,00673	0,83413	0,00199
2	0,6353	7,00E-05	0,75395	0,0067	0,83413	0,002
Mittel	0,80027	5,00E-05	0,80329	0,00528	0,86195	0,00158

Tabelle B.1.1.: Ausschnitt der Resultate des Testdurchlaufs 03 vom 18.03.2023

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Mittweida, den 28. März 2023

Alex Prezewowsky