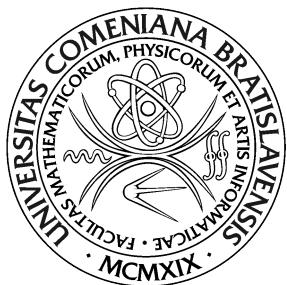


UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



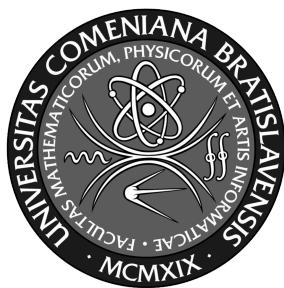
PRIRODZENÝ DIALÓGOVÝ SYSTÉM PRE HRY

Diplomová práca

2022

Bc. Michal Chamula

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



PRIRODZENÝ DIALÓGOVÝ SYSTÉM PRE HRY

Diplomová práca

Študijný program: Aplikovaná informatika

Študijný odbor: 2511 Aplikovaná informatika

Školiace pracovisko: Katedra aplikovanej informatiky

Školiteľ: RNDr. Jozef Šiška

Bratislava, 2022

Bc. Michal Chamula



38537744

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Michal Chamula

Študijný program: aplikovaná informatika (Jednoodborové štúdium,
magisterský II. st., denná forma)

Študijný odbor: informatika

Typ záverečnej práce: diplomová

Jazyk záverečnej práce: slovenský

Sekundárny jazyk: anglický

Názov: Prirodzený dialógový systém pre hry
Natural dialogue system for computer games

Anotácia: Počítačové hry využívajú rôzne spôsoby na reprezentáciu komunikácie hráča s postavami v hre: dialógové systémy. Najčastejšie sú používané systémy vo forme dialógových stromov, kde všetky možné repliky hráča a odpovede postáv tvoria vopred vytvorený / navrhnutý acyklický graf a hráč si vyberá jednu z možností dostupných v danom momente.

Iné dialógové systémy sa snažia emulovať prirodzenú formu dialógu, kde hráč nevyberá z predvolených možností, ale priamo píše svoje odpovede / otázky. Ich zložitosť sa lísi od jednoduchých, založených na rozpoznaní kľúčových slov, po zložitejšie, založené na prístupoch spracovania prirodzeného jazyka. Vo všeobecnosti sú však málo používané v reálnych hrách kvôli ich nedokonalosti / nepredvídateľnosti.

Ciel: Cieľom práce je navrhnúť parser-driven dialógový systém, v ktorom sa repliky postáv vyberajú na základe hráčom napísanej správy / otázky a kontextu (stavu hry, predchádzajúca komunikácia) a implementovať prototyp, bud' ako samostatnú aplikáciu (jednoduchá textová adventúrka alebo pod.) prípadne integrovaný do nejakého herného enginu.
Dialógový systém by mal čo najvernejšie simulovať rozhorov hráča s postavou: mal by hráčovi ukazovať repliky postavy relevantné k aktuálnej situácii v hre a k otázkam hráča / doterajšiemu priebehu dialógu.

Kľúčové slová: Počítačové hry, dialógové systémy

Vedúci: RNDr. Jozef Šiška, PhD.

Katedra: FMFI.KAI - Katedra aplikovanej informatiky

Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 14.10.2019

Dátum schválenia: 15.10.2019

prof. RNDr. Roman Ďuríkovič, PhD.
garant študijného programu




38537744

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

.....
študent

.....
vedúci práce

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením školiteľa a uvedenej literatúry.

.....
Bratislava, 2022

Bc. Michal Chamula

Pod'akovanie

Týmto by som chcel pod'akovať vedúcemu diplomovej práce RNDr. Jozefovi Šiškovi, za jeho poznatky, rady a priponienky, ktoré mi boli nápmocné pri tvorbe tejto práce. Ďalej by som sa chcel pod'akovať všetkým kolegom a ochotným ľud'om, ktorí mi tiež svojimi poznatkami prostredníctvom internetu dopomohli k tvorbe a realizácii práce.

Abstrakt

Chamula, Michal: Prirodzený dialógový systém pre hry. Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra aplikovanej informatiky. Vedúci práce: RNDr. Jozef Šiška, Bratislava: FMFI UK, 2020.

V diplomovej práci sa venujeme tvorbe dialógového systému, bežiacom v prostredí Unreal Engine 4.27. Programový kód je vytváraný ako kombinácia jazykov C++ a Blueprint, ktorý je navrhnutý pre prostredie Unreal Engine.

Výsledkom práce je stochastický dialógový systém zameraný hlavne na použitie v hernom priemysle. Jeho úlohou je vyhľadávanie dialógových replík pomocou algoritmov založených na analýze kl'účových slov z herného vstupu.

Systém sa značí vysokou portabilitou a modulárnosťou, vd'aka čomu ho dokážeme jednoducho integrovať do herných projektov vytvorených v prostredí Unreal Engine. Systémový kód dokážeme jednoducho rozširovať o nové algoritmy, a funkcionality bez potreby zasahovania do jadra programu.

Kl'účové slová: počítačová hra, experimentálny dialógový systém, spracovanie prirodzeného jazyka, analýza kl'účových slov

Abstract

Chamula, Michal: Natural dialogue system for games. Comenius University, Bratislava. Faculty of Mathematics, Physics and Informatics; Department of Applied Informatics. Supervisor: RNDr. Jozef Šiška, Bratislava: FMPH UK, 2020. 84 pages.

In this thesis we present dialogue system for games created in Unreal Engine 4.27. Programming code is represented as combination of C++ and Blueprint programming languages.

The result of this work is a stochastic dialogue system, mainly for use in the gaming industry. Its purpose is to select dialogue replies using algorithms based on a keyword analysis from game input.

The system is highly portable and modular, due to which we are able to easily integrate it into game projects in Unreal Engine environment. We are able to easily expand the system code by implementing new algorithms and functionality, without the need of editing the core of the system.

Keywords: PC game, experimental dialogue system, natural language processing, keyword analysis

Obsah

Úvod	6
1 Teoretické východiská práce	7
1.1 Dialógové systémy	7
1.1.1 Prirodzené dialógové systémy	8
1.1.2 Dialógové stromy	8
1.2 Problematika v herných systémoch	10
1.2.1 Podpora pre viac jazykov	10
1.2.2 Preklad zvuku	11
1.2.3 Štylizovanie dialógu	11
1.2.4 Reakcie na zmeny stavu hry	12
1.2.5 Práca s hernými objektami	12
1.3 Metódy spracovania prirodzeného jazyka	12
1.3.1 Metódy extrakcie klúčových slov	13
1.3.2 Metódy normalizácie textu	15
1.3.3 Levenshteinova vzdialenosť	16
1.4 Použitý software	17
1.4.1 Unreal Engine 4	18
1.4.2 Unreal Rider	18
1.4.3 Digital ocean	18

1.4.4	Helix core	19
1.5	Podobné práce	20
1.5.1	Dialogue Plugin System	20
1.5.2	Dialogue Plugin	21
1.5.3	Zork	22
1.5.4	Chatbot	23
2	Ciele práce	25
3	Návrh	27
3.1	Normalizácia textu	27
3.2	Hierarchia implementovaných tried	28
3.3	Natural Dialog system settings	30
3.4	Data Table	30
3.5	Dictionary Subsystem	33
3.5.1	Reprezentácia slovníka	33
3.6	Natural Dialog system component	34
3.6.1	NPC komponent	35
3.6.2	Player komponent	36
3.7	Funkčné triedy	36
3.7.1	Dictionary Word Picker Function	37
3.7.2	String Distance Function	39
3.7.3	Keyword Picker Function	40
3.7.4	Dialog Reply Function	42
3.7.5	Reply Helper Function	43
3.8	Dialógové úlohy	46
3.9	Replikácia systému	47
3.10	Práca systému	49

4 Použitie systému	53
4.1 Integrovanie pluginu	53
4.2 Implementácia pluginu	57
4.2.1 Integrácia NPC komponentu	57
4.2.2 Integrácia Player komponentu	59
4.3 Nastavenie pluginu	61
4.3.1 Globálne nastavenia	62
4.3.2 Nastavenia komponentov	62
4.4 Vytvorenie a editácia dátových tabuľiek	64
4.5 Vytvorenie a editácia dialógových úloh	65
4.6 Použitie systému	68
5 Výsledky práce	71
5.1 Portabilita a modularita	72
5.2 Testovanie	72
5.3 Chybovost'	76
A Interakcia s NPC	81

Zoznam obrázkov

1.1	Príklad hry s prirodzeným dialógovým systémom	8
1.2	Príklad hry s dialógovým stromom	9
1.3	Príklad normalizácie textu	16
1.4	Linuxový server bežiaci na cloude Digital Ocean	19
1.5	Verzovací systém Helix core	19
1.6	Príklad zo systému: <i>Dialogue Plugin System [1]</i>	20
1.7	Príklad zo systému: <i>Dialogue Plugin [2]</i>	21
1.8	Príklad z hry Zork	22
3.1	Triedna hierarchia jadra systému	29
3.2	Príklad dátovej tabuľky	32
3.3	Algoritmus pre vyhľadávanie slov v slovníku	38
3.4	Príklad výstupu z funkcie <i>Find Ask Options</i>	44
3.5	Workflow diagram systému	52
4.1	Príklad umiestnenia zdrojových súborov	54
4.2	Implementovanie pluginu do projektového súboru	54
4.3	Implementovanie pluginu v súbore Build.cs	55
4.4	Implementovanie pluginu v súbore Target.cs	56
4.5	Implementovanie pluginu v súbore Editor.target.cs	56
4.6	Deklarácia NPC komponentu v .h súbore	57

4.7	Vytvorenie NPC komponentu v .cpp súbore	58
4.8	Vytvorenie NPC komponentu v Blueprintie	58
4.9	Úspešná implementácia NPC komponentu	59
4.10	Deklarácia Player komponentu v .h súbore	60
4.11	Vytvorenie Player komponentu v .cpp súbore	60
4.12	Vytvorenie NPC komponentu v Blueprintie	61
4.13	Úspešná implementácia Player komponentu	61
4.14	Nastavenia systému v projekte	62
4.15	Nastavenia NPC komponentu	62
4.16	Nastavenia Player komponentu	63
4.17	Príklad vytvárania dátovej tabuľky	64
4.18	Editácia dátovej tabuľky	65
4.19	Vytvorenie dialógovej úlohy	66
4.20	Editácia dialógovej úlohy	66
4.21	Príklad widgetu so vstupnými prvkami	68
4.22	Príklad eventov pre vstupné prvky z obrázku 4.21	69
4.23	Príklad implementácie nájdenia dostupných otázok v Blueprintie	69
4.24	Príklad implementácie nájdenia dostupných otázok v C++ .	70
4.25	Príklad implementácie generovania odpovedí v Blueprintie .	70
4.26	Príklad implementácie generovania odpovedí v C++	70
5.1	Výsledky otázky: <i>Ako často hrávaš hry?</i>	73
5.2	Výsledky otázky: <i>Aké herné tituly preferuješ?</i>	73
5.3	Výsledky otázky: <i>Hráš hry založené na dialógových systémoch?</i>	74
5.4	Výsledky otázky: <i>Zaujal t'a typ odskúšaného dialógového systému?</i>	74

5.5 Výsledky otázky: <i>Preferoval by si v budúcnosti viac hier s odskúšaným dialógovým systémom?</i>	75
5.6 Výsledky otázky: <i>Myslíš, že podobný systém je budúcnosť v hernom priemysle?</i>	75
5.7 Výsledky otázky: <i>Ako náročný bol systém na používanie?</i> . .	76
5.8 Výsledky otázky: <i>Ako intuitívny bol pre teba systém?</i>	76
5.9 Výsledky otázky: <i>Ako hodnotíš odskúšaný dialógový systém?</i>	77
5.10 Výsledky otázky: <i>Ako hodnotíš konverzáciu s postavami v hre?</i>	77
5.11 Výsledky otázky: <i>Ako často postava nevedela odpovedať na tvoju otázku?</i>	77
5.12 Výsledky otázky: <i>Ako náročné bolo postupovať príbehom?</i> .	78
A.1 Príklad interakcie s NPC	81
A.2 Príklad vstupu pre dialóg s NPC	81

Zoznam skratiek a značiek

UE4 - Unreal Engine 4

NPC - Non Player Character

BP - Blueprint

UI - User Interface

Slovník

Non Player Character - Nehráčska postava, zvyčajne ovládaná pomocou umelej inteligencie

Blueprint - Vizuálny skriptovací jazyk navrhnutý pre prácu v UE4

User Interface - Používateľné rozhranie

Asset - Projektový súbor použitý na reprezentáciu hernej súčasti (obrázok, dátový súbor, 3D model, Blueprint, atď.).

Úvod

Cieľom tejto diplomovej práce je vytvorenie prototypu experimentálneho systému pre spracovanie prirodzeného jazyka v herných systémoch, ktorý bude navrhnutý, pre projekty bežiace v prostredí Unreal Engine. Práca bude zameraná na experimentálny pokus o implementovanie prirodzeného jazyka do herného sveta.

Posledných niekol'ko rokov sa v počítačovom odvetví úspešne rozvíja umelá inteligencia, ktorá veľmi výrazne prispieva k porozumeniu prirodzeného jazyka ľudí a následne sa ho snaží využiť pre rôzne odvetvia informatickej sféry. Avšak v hernom odvetví sa táto metóda aktívne nerozvíja a používa sa stará metóda dialógových stromov. Napriek tomu, že jej návrh siaha až do 80. rokov, ostáva najpopulárnejšia. Využíva preddefinovaný deterministický dialóg medzi hráčom a NPC postavou. V tejto práci sa pokúsime o vytvorenie systému, ktorý nahradí metódu dialógových stromov novým stochastickým správaním.

Dúfame, že prácou na tomto projekte môžeme vytvoriť novú podobu systému, ktorý by tvoril základ moderných herných dialógov, zlepšil hernú úroveň a zážitok pre hráčov.

1. Teoretické východiská práce

V tejto kapitole sa budeme venovať herným dialógovým systémov z historického a súčastného hľadiska. Rozoberieme problematiku dnešných systémov a možnú problematiku implementácie prirodzeného spracovania jazyka v hernom odvetví. Súčasťou kapitoly bude aj analýza súčasných techník na spracovanie prirodzeného jazyka.

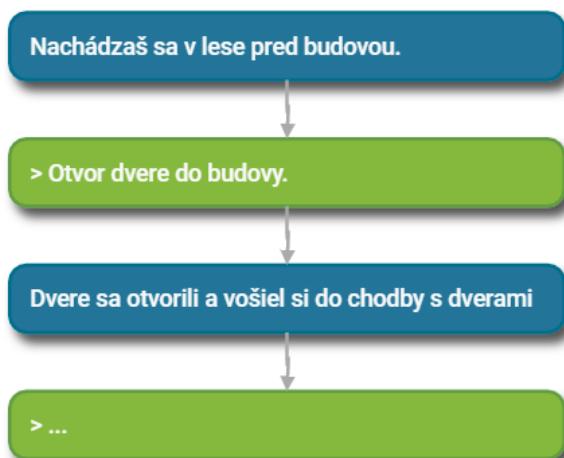
1.1 Dialógové systémy

Dialógové systémy sú aktívne využívané v rôznych odvetviach informatiky, hlavne s použitím neurónových sietí, ktoré dosahujú veľmi dobré výsledky. Avšak v herných systémoch, podobné prvky neboli implementované z niekoľkých rôznych dôvodov, ktoré popíšeme v časti 1.2.

V hrách sa najčastejšie používajú dialógové stromy, ktoré sú najobľúbenejšou variantou. Vďaka ich jednoduchej implementácii je ich časová a pamäťová náročnosť minimálna a pre ich editáciu nie je potrebný často zásah vývojárov.

1.1.1 Prirodzené dialógové systémy

Prirodzené dialógové systémy boli populárne pred príchodom dialógových stromov. Ich určenie bolo obmedzené na konkrétnu hru a neboli prenosné medzi hernými produktami. Boli oblúbené v herných adventúrach, ako napríklad hra Zork [3], ktorú popíšeme v časti 1.5.3. Podobné hry boli prezentované iba textovou formou, kde herný príbeh bol reprezentovaný ako postupnosť otázok a odpovedí.



Obr. 1.1: Príklad hry s prirodzeným dialógovým systémom

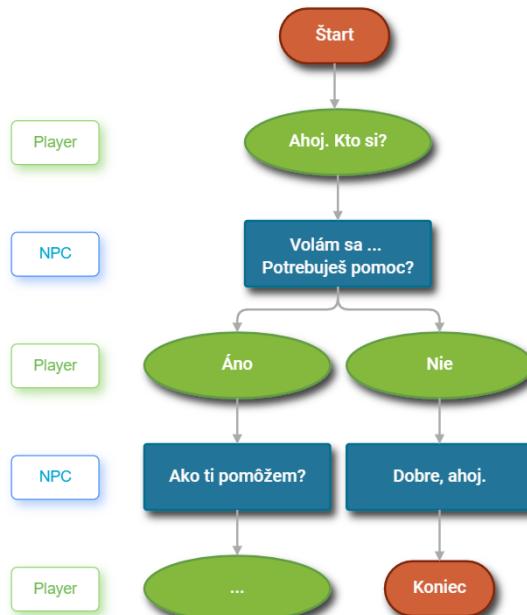
Na obrázku 1.1 je uvedený príklad textovej hry, využívajúcej prirodzený dialógový systém. Modré obdĺžniky sú repliky, ktoré nás posúvajú v príbehu. Zelené obdĺžniky sú hráčske repliky, ktorými sa hráč snaží posúvať herným príbehom. Každá herná replika tiež reprezentuje stav hry.

1.1.2 Dialógové stromy

Od konca 80. rokov sa v hernom odvetví na reprezentáciu a implementovanie používajú najčastejšie dialógové stromy [4]. Ich reprezentácia pri-

pomína graf, obsahujúci štartovný bod *Root*. Každý vrchol grafu prestaruje dialógovú repliku a je uložený v postupnosti vo vетvách grafu. Každá veta predstavuje vlastnú dejovú os dialógu a jej výberom sa herný dej odvíja. Je často závislý od aktuálneho stavu hry, atribútov a stavu dialógového stromu. Hráč vždy pozná nasledovné možnosti, ktoré môže použiť pre pokračovanie v príbehu. V hrách sú repliky vizuálne zobrazené v grafickom rozhraní a hráč si vie vybrať dejovú os, ktorou chce v hernom deji pokračovať.

Narozdiel od prirodzeného dialógového systému vieme, že každý dialógový strom zaručene vyberie správnu odpoveď na akúkol'vek repliku postavy. Príbeh hry sa často odohráva na jednej dejovej osi, ktorú rozvetvujeme a neexistuje žiadna alternatívna os, ktorou by sme príbeh postavy mohli začať.



Obr. 1.2: Príklad hry s dialógovým stromom

Na obrázku 1.2 vidíme jednoduchú reprezentáciu dialógového stromu. Koreň stromu slúži ako počiatočný bod *Root*, z ktorého môže viest' viac replík, ktoré sa môže hráč spýtať. Každú repliku zobrazíme v používateľskom rozhraní a hráč si vyberie vetvu príbehu, ktorou bude pokračovať. Modré obdĺžniky sú repliky patriace NPC a zelené ovály patria hráčovi. Každý dialógový systém, využívajúci dialógové stromy, je unikátny a obsahuje vlastnú funkcionality podľa implementácie a modularity, ktorá zjednodušuje manipuláciu s takýmto systémom.

1.2 Problematika v herných systémoch

V tejto časti rozoberieme problematiku implementácie spracovania prirodzeného jazyka v hernom odvetví. Oproti bežnej informatickej sfére musí byť súčasťou takéhoto systému implementácia súčasti, ktoré sa v bežných komunikačných aplikáciách nevyskytujú.

1.2.1 Podpora pre viac jazykov

V informatických odvetviach, ktoré sa zaobrajú spracovaním prirodzeného jazyka, často používame umelú inteligenciu a neurónové siete na spracovanie vstupov od užívateľa. Tieto vstupy sú hlavne v angličtine a nie sú viacjazyčné. Pri herných systémoch však treba dbať na to, že hra je veľa krát preložená do viacerých jazykov. Ak systém vie pracovať iba v jednom jazyku (napr. v angličtine) hra by sa obmedzila iba na hráčov z krajín, ktorí ovládajú tento jazyk a firmy by takýto systém nepoužívali. Z tohto dôvodu je dôležité pristupovať k vývoju s podporou pre viaceré jazyky.

1.2.2 Preklad zvuku

V predchádzajúcim bode sme sa venovali problému spojenom s textovou formou. V hrách je ale veľmi dôležité, aby každý dialóg medzi hráčom a NPC mal aj zvukovú formu. V dialógových stromoch bolo veľmi jednoduché skombinovať textovú časť odpovede so zvukovou, priradením zvukového assetu. V stochastickom systéme, kde by na otázku existovala nekonečná množina odpovedí, je nemožné nahrať pre každú repliku zvukovú formu.

1.2.3 Štylizovanie dialógu

Dnešnú spoločnosť tvoria skupiny ľudí, ktorí používajú rôzny slang a hovorový spôsob vyjadrovania. Ak človek navštívi webovú stránku, na ktorej použije internetového chatbota pre vyhľadávanie, pristupuje k programu s odborným, prípadne úradným jazykom. V herných systémoch je dôležité mysliť na široké spektrum herných titulov, v ktorých sa spôsoby dialógu môžu rozlišovať.

Predstavme si dva herné tituly, kde dej jedného titulu sa odohráva v prostredí mafie a druhý vo vojnovej prostredí. Dialógy prvého titulu, budú veľmi odlišné od dialógov z druhého titulu. Bude sa to týkať štylizácie textu, slovnej zásoby a rôznych iných vecí. Tak isto budú otázky smerované od hráča pre NPC úplne odlišné. Podobný prípad môže nastat aj v dialógoch vedených s rôznymi NPC v rámci jednej hry, kde jedna postava môže s nami viest arogantný a druhá príjemný dialóg.

1.2.4 Reakcie na zmeny stavu hry

Počas herného dejá môže nastáť situácia, kedy chceme aby NPC poskytvalo nové informácie, na ktoré sa hráč môže spýtať až v nejakom žiadúcom stave hry. Ak hráč splní úlohu, ktorá ho posunie príbehom, môže NPC nadobudnúť nové informácie o stave sveta, ktoré musíme zaregistrovať pre daný charakter. Splnením úlohy si hráč môže vypýtať od NPC novú, ku ktorej pred tým nemal prístup. Podobná situácia môže nastáť, ak si dialóg vyžiada aby sme sa najprv porozprávali v iném NPC, ktoré nám rozšíri dejovú linku a až potom sa môžeme vrátiť a pokračovať v dialógu so starou postavou.

1.2.5 Práca s hernými objektami

V prípade hier je potrebné aby systém vedel spolupracovať s hernými objektami. Hráč sa počas dialógu môže pýtať na herné súčasti a mechaniky, o ktorých sa bude chcieť informovať, alebo spolupracovať s nimi. Súčasťou takejto implementácie je možnosť integrácie vlastnej funkcionality pre potreby hry.

1.3 Metódy spracovania prirodzeného jazyka

Pre spracovanie prirodzeného jazyka (*NLP*) existuje viacero spôsobov, ako k nemu pristupovať. Text môžeme analyzovať pomocou zaužívaných algoritmov, alebo moderných neurónových sietí. Každá implementácia závisí na rozsahu výpočtovej techniky, alebo od typu *NLP*. Texty môžu existovať v rôznej forme (ústne a písomné), alebo môžu byť v rôznych jazykoch. Existuje mnoho metód *NLP* [5], ktoré sú zaužívané.

1.3.1 Metódy extrakcie kl'účových slov

Extrakcia kl'účových slov (tiež známa ako detekcia kl'účových slov, alebo analýza kl'účových slov) je technika analýzy textu, ktorá automaticky extrahuje najpoužívanejšie a najdôležitejšie slová a výrazy. Systém môže používať na vyhľadávanie kl'účových slov rôzne textové formy (bežné dokumenty, online fóra, recenzie, správy atď.).

Metódy používané na analýzu kl'účových slov môžu byť založené napr. na algoritme TextRank [6], Tf-idf [7], RAKE [8] (Rapid automatic keyword extraction), alebo na vyhľadávaní jazykových fráz, a pod. Pre implementáciu nás zaujali algoritmy TextRank a Tf-Idf, ktoré sú založené hlavne na detekcii kl'účových slov pomocou analýzy dokumentov.

TextRank

Algoritmus TextRank [6] je založený na grafovom modeli hodnotenia na spracovanie textu, ktorý možno použiť na nájdenie najrelevantnejších viet v texte a tiež na nájdenie kl'účových slov.

Aby bolo možné nájsť relevantné kl'účové slová, algoritmus vytvára siet' slov. Táto siet' je vytvorená tak, že sa pozrieme, ktoré slová nasledujú za sebou. Prepojenie sa vytvorí medzi dvoma slovami, ak nasledujú za sebou, pričom toto prepojenie získa väčšiu váhu, ak sa tieto dve slová vyskytujú v texte častejšie vedľ'a seba.

Podľ'a [6] môžeme TextRank definovať nasledovne. Nech $G = (V, E)$ je orientovaný graf s množinou vrcholov V a množinou hrán E , kde E je podmnožina $V \times V$. Pre daný vrchol V_i nech $In(V_i)$ je množina vrcholov, ktoré naň ukazujú a nech $Out(V_i)$ je množina vrcholov, na ktoré vrchol V_i

ukazuje. Skóre vrcholu V_i je definované nasledovne:

$$S(V_i) = (1 - d) + d * \sum_{j \in In(V_i)} \left(\frac{1}{|Out(V_j)|} \right) S(V_j)$$

kde d je faktor tlmenia, ktorý možno nastaviť medzi 0 a 1, ktorý má za úlohu integrovať do modelu pravdepodobnosť skoku z daného vrcholu do iného náhodného vrcholu v grafe.

Vo výslednej sieti sa aplikuje algoritmus Pagerank [9], aby sa zistila dôležitosť každého slova. Horná tretina všetkých týchto slov sa ponechá a považuje sa za relevantnú. Potom sa zostaví tabuľka klúčových slov.

Tf-Idf

Algoritmus *Term frequency-inverse document frequency* [7], (skrátene Tf-Idf) je metóda založená na numerickej štatistike, ktorá ohodnotí dôležitosť slova pre dokument z konečnej množiny dokumentov. Využíva sa ako váhový faktor, pri vyhľadávaní informácií. Prieskum z roku 2015 ukázal, že viac ako 80% systémov využívajúcich textové odporúčania, používa algoritmus *Tf-Idf*:

$$Tf - Idf(t, d) = Tf(t, d) \times Idf(t)$$

Hodnota slova priamo-úmerne narastá s počtom výskytov termu t v dokumente d . $Tf(t, d)$ a je kompenzovaná počtom dokumentov, ktoré slovo obsahujú hodnotou $Idf(t)$ pre term t :

Hodnotu $Tf(t, d)$ môžeme zapísat' ako:

$$Tf(t, d) = \left(\frac{n_{t,d}}{m_d} \right)$$

kde hodnota n_{td} zodpovedá počtu výskytov termu t v dokumente d a hodnota m_d predstavuje počet všetkých termov v dokumente d.

Hodnotu $Idf(t)$ môžeme zapísat' ako:

$$Idf(t) = \left(\frac{i}{j_t} \right)$$

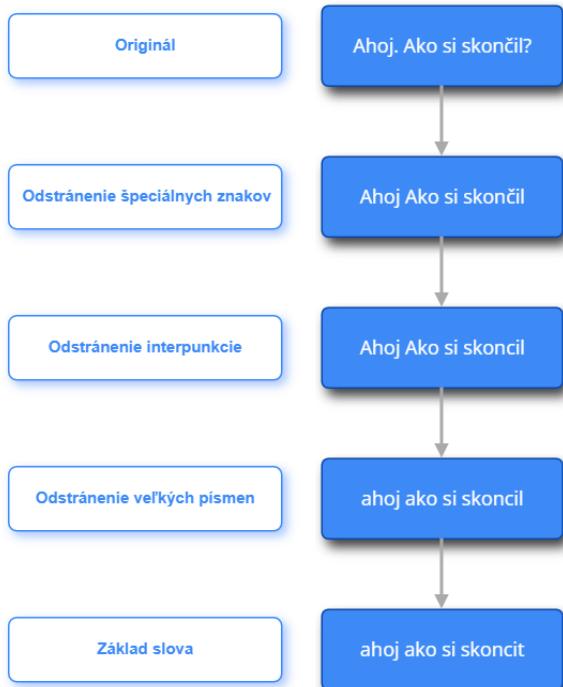
kde hodnota i je počet všetkých dokumentov a hodnota j_t je počet dokumentov, ktoré obsahujú term t.

1.3.2 Metódy normalizácie textu

Úlohou normalizácie textu [10] je zmapovať nekanonický jazyk na štandardizované písmo. Je nevyhnutná na následné využitie nástrojov na spracovanie prirodzeného jazyka. Text by sa mal normalizovať odstránením diakritických znamienok a konvertovať na malé písmená. Môže byť tiež upravený do základného stavu slova (napr. neurčitok), môžeme ho kanonizovať, alebo môžu byť odstránené zastavovacie slová *Stop Words*.

Techniky normalizácie textu môžu byť založené na jednoduchej od kontextu nezávislej normalizácii. Na odstránenie nealfanumerických znakov, alebo diakritických znamienok, by postačovali regulárne výrazy. Na zložitejšiu normalizáciu sú potrebné komplikované algoritmy, vrátane doménových znalostí jazyka a slovnej zásoby.

Na obrázku 1.3 je príklad normalizácie textu, kde v jednotlivých krokoch normalizujeme text. Začíname s originálnym textom, ktorý sa nachádza v hornej časti obrázku a po spracovaní dostaneme normalizovaný text ako sa nachádza v dolnej časti obrázku.



Obr. 1.3: Príklad normalizácie textu

1.3.3 Levenshteinova vzdialenosť

Levenshteinova vzdialenosť [11] počíta najmenší počet operácií potrebných na transformáciu jedného retázca na druhý. Operácie pre úpravu zahŕňajú vkladanie, mazanie a nahradzanie znakov v retázcoch. Patrí medzi najobľúbenejšiu metriku zo skupiny metrík vzdialenosť známych ako *Edit distance*. Tieto metriky sa líšia v súbore základných operácií.

Operáciu na vypočítanie vzdialenosť definujeme vzorcom:

$$Lev_{a,b}(i, j) = \begin{cases} \max(i, j) & \text{Ak } \min(i,j)=0 \\ \min \begin{cases} Lev_{a,b}(i - 1, j) + 1 \\ Lev_{a,b}(i, j - 1) + 1 \\ Lev_{a,b}(i - 1, j - 1) + 1_{(a_i \neq b_j)} \end{cases} & \text{inak} \end{cases}$$

kde a, b sú vstupné retázce a i, j sú indexy znakov týchto retázcov. Prvá časť vzorca označuje počet krokov vloženia, alebo vymazania. Druhý blok je rekurzívny výraz, pričom prvý riadok predstavuje vymazanie znaku a druhý predstavuje vloženie. Posledný riadok je zodpovedný za zmenu znaku.

Vlastnosti algoritmu:

- Funkcia vráti hodnotu 0 vždy ak sa vstupné retázce a, b rovnajú.
- Funkcia je symetrická:

$$Lev_{a,b}(i, j) = Lev_{b,a}(i, j).$$

- Maximálna hodnota funkcie je dĺžka najväčšieho vstupného retázca

1.4 Použitý software

V tejto časti popíšeme software, ktorý bol aktívne používaný na prácu a vývoj systému. Informácie o softvéroch sú čerpané z oficiálnych firemných stránok.

1.4.1 Unreal Engine 4

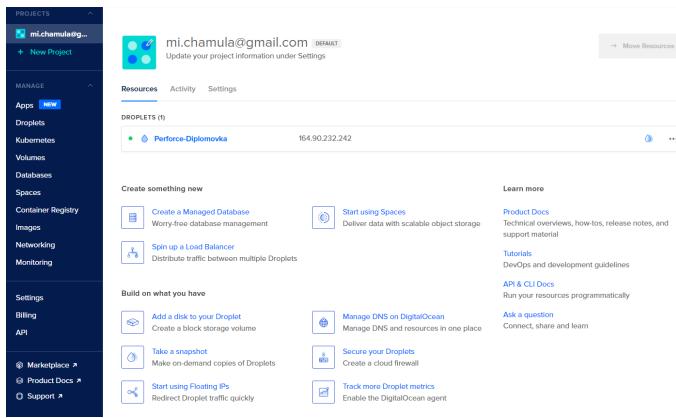
Unreal Engine [12] je open-source herný engine vyvinutý spoločnosťou Epic Games, ktorý bol prvýkrát predstavený v roku 1998. Pôvodne bol vyvinutý pre hry z pohľadu prvej osoby, ale neskôr sa začal používať v rôznych žánroch 3D hier. V neskoršom štádiu sa dočkal prijatia aj v iných odvetviach, najmä vo filmovom a televíznom priemysle. Unreal Engine je napísaný v jazyku C++ a vyznačuje sa vysokým stupňom prenosnosti na rôzne platformy.

1.4.2 Unreal Rider

Unreal Rider [13] je software, ktorý je vytvorený firmou JetBrains s.r.o. a je podporovaný na rôzne platformy. Unreal Engine podporuje iba niekoľko druhov softwaru pre programovanie C++ kódu. V porovnaní s Visual Studiom, ktoré je prezentované a zväčša používané na vývoj hier pre Unreal Engine, má Unreal Rider rozšírenejšiu podporu pre Unreal Engine a dokáže aktívne komunikovať s prostredím Unreal Editor.

1.4.3 Digital ocean

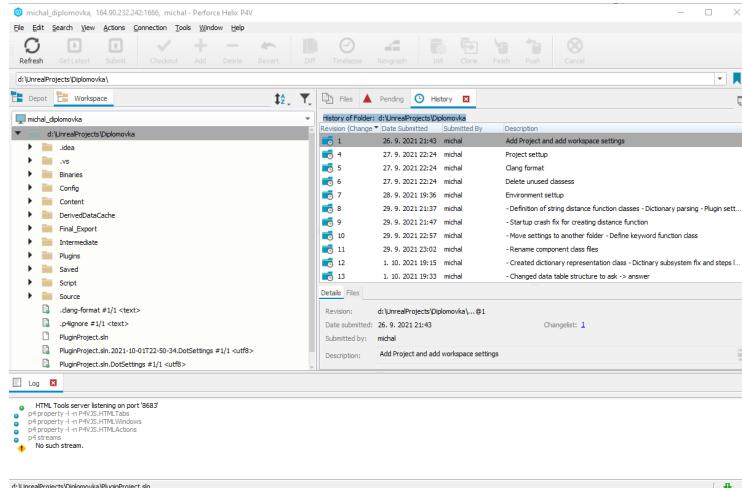
DigitalOcean [14] je americký poskytovateľ cloubovej infraštruktúry so sídlom v New Yorku s dátovými centrami po celom svete. Poskytuje vývojárom cloudové služby, ktoré pomáhajú vyvíjať aplikácie, ktoré bežia súčasne na viacerých počítačoch. Pre vývoj a ľahkej práci na systéme sme si vytvorili úložisko, na ktorom je spustené linuxové prostredie a v ňom verzovací systém Helic core popísaný v časti 1.4.4, so zálohami pre zabezpečenie prípadnej straty súborov aplikácie.



Obr. 1.4: Linuxový server bežiaci na cloude Digital Ocean

1.4.4 Helix core

Helix Core [15] je softvér používaný na vývoj aplikácií a na správu verzií. Je veľmi využívaný pri vývoji v hernom priemysle, kde dokáže veľmi dobre komprimovať a pracovať s binárnymi súbormi. V našej práci je používaný predovšetkým na verzovanie projektových súborov. Je spustený na linuxovom serveri bežiacom na cloude sprostredkovovanom cez systémy Digital ocean popísaný v časti 1.4.3.



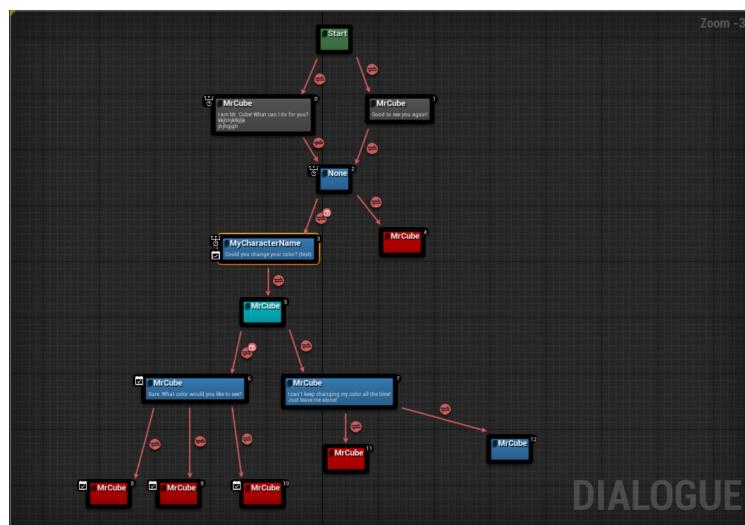
Obr. 1.5: Verzovací systém Helix core

1.5 Podobné práce

V tejto časti sa budeme venovať dialógovým systémom, ktoré sa používajú v hernom, alebo informačnom odvetí. Popíšeme tu staršie systémy, ktoré sa už nepoužívajú a tiež systémy používané v dnešnej dobe. Súčasťou tejto kapitoly budú tiež systémy používané v ostatných informačných odvetviach aby sme ich vedeli porovnať’.

1.5.1 Dialogue Plugin System

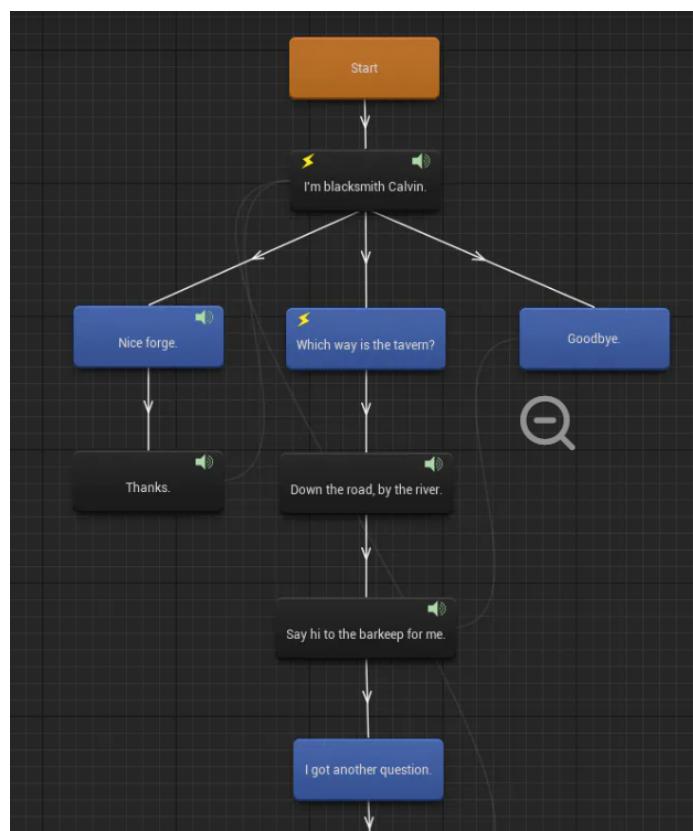
Dialogue Plugin System [1], vytvorený v prostredí Unreal Engine 4 a vydaný ako herný plugin 2. Marca 2018 vývojárom Not Yet. Je založený na koncepte dialógových stromov. Jeho veľkou výhodou je jeho ľahká integrácia do projektu a je založený na grafovom editore, vďaka čomu je vyhľadávaný užívateľmi UE prostredia. Je napísaný v jazyku C++ s plnou podporou pre BP kód. Ponúka komplexnú implementáciu dialógového systému, ktorá ale ovplyvňuje užívateľskú náročnosť’.



Obr. 1.6: Príklad zo systému: *Dialogue Plugin System* [1]

1.5.2 Dialogue Plugin

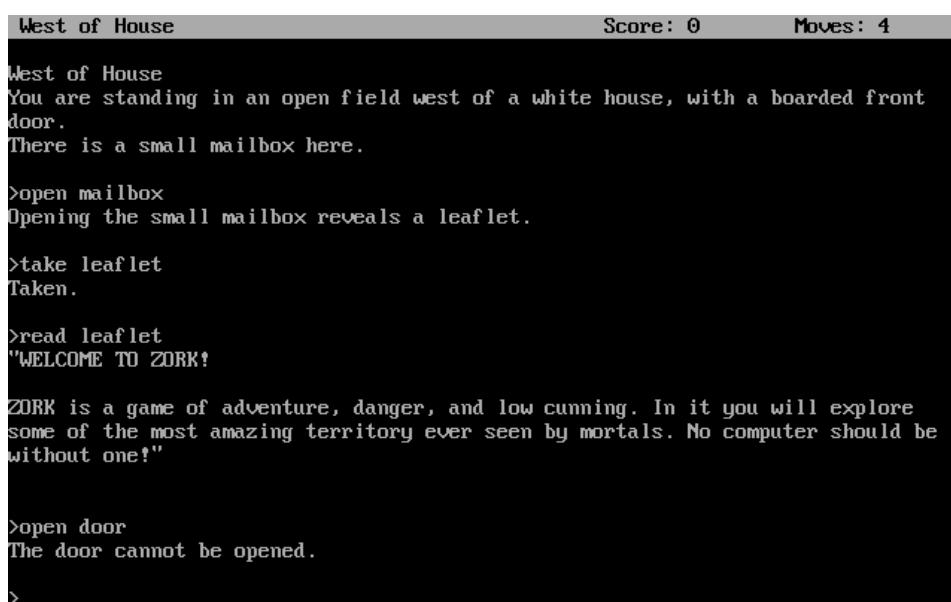
Dialogue Plugin [2] je vytvorený v prostredí Unreal Engine 4 a vydaný ako herný plugin 27. Mája 2016 vývojárom CodeSpartan. Podobne ako predchádzajúci plugin, je založený na koncepte dialógových stromov a na grafovom editore. Je napísaný v jazyku C++ s podporou pre BP kód. Je používaný známymi spoločnosťami ako *Piranha Games*, alebo *Iron Tower Studio*. Oproti predchádzajúcemu pluginu, patrí medzi staršie a menej robustné systémy a preto je ľahší na porozumenie.



Obr. 1.7: Príklad zo systému: *Dialogue Plugin* [2]

1.5.3 Zork

Zork [3] je interaktívna počítačová hra vyvinutá v 70. rokoch 20. storočia. Patrí medzi prvé textové hry, v ktorých užívateľ prechádza príbehom, zadávaním krokov v textovej forme. Hra je prezentovaná ako dialóg medzi hráčom a rozprávačom príbehu, ktorý odpovedá na otázky a kroky užívateľa. Cieľom hry je postupovať príbehom zadávaním krokov do príkazového riadku. Hra sa snaží analyzovať herný vstup a vysvetliť vhodný postup príbehom.



The screenshot shows a terminal window with the title "West of House". At the top right, it displays "Score: 0" and "Moves: 4". The main area contains the following text:

```
West of House
Score: 0      Moves: 4

West of House
You are standing in an open field west of a white house, with a boarded front
door.
There is a small mailbox here.

>open mailbox
Opening the small mailbox reveals a leaflet.

>take leaflet
Taken.

>read leaflet
"WELCOME TO ZORK!"

ZORK is a game of adventure, danger, and low cunning. In it you will explore
some of the most amazing territory ever seen by mortals. No computer should be
without one!

>open door
The door cannot be opened.

>
```

Obr. 1.8: Príklad z hry Zork

Narozdiel od tejto hry sa snažíme systém navrhnúť, aby fungoval modulárne na rôzne hry a rôzne platformy. Veľkým rozdielom je tiež, že hra sa snaží naviest užívateľa k ďalšiemu kroku, čiže systém napodobňuje deterministické správanie. Ak sa hráč nachádza v akomkoľvek bode hry, existuje konečná množina vstupov, ako sa môže posunúť do nového bodu. Hráč má teda možnosť vybrať si ďalší krok sám, ale závisí na ňom akým

spôsobom v dialógu sa k tomuto kroku dostane.

Systémy sú si teda podobné iba tým, že postup príbehom docielime správnym kladením otázok. Ich modularita je ale rozdielna, ako aj spôsob výberu vhodnej odpovedi pre hráča. Veľký rozdiel je tiež v podpore herných jazykov (Zork podporuje iba angličtinu). Ako sme už popísali vyššie, jedným z cieľov práce je pokúsiť sa dosiahnuť, aby hra podporovala viac jazykov. Na internete môžeme nájsť množstvo podobných titulov, podobných hre Zork.

1.5.4 Chatbot

Chatboty [16, 17, 18] sú programy, ktoré napodobňujú konverzáciu ľudí použitím umelej inteligencie. Sú využívaní ako pomocní asistenti, alebo pre zábavné účely. Stali sa populárnymi v biznis oblastiach, kde sú nasadené na podporné centrá, vďaka čomu dokážu redukovať čas so zákazníckou odozvou, cenu podporných centier a podporu viacerých užívateľov naraz. Ich úlohou je jednoducho odpovedať na otázky položené používateľom, formou:

Otázka \Rightarrow Odpoved'.

Pre implementáciu chatbotov sa používajú rôzne metódy umelej inteligencie. Napríklad implementácia pomocou porovnávania vzorov (pattern matching), v ktorom sa rozoznané poradie vety a uložený vzorec reakcie upraví na premenné vety. Nie sú stavané na odpovedanie komplexných otázok, nakoľko je to pomerne nová technológia. Pattern matching je technika vyhľadávania podobností vstupu uloženým v databáze. K tomuto vstupu je definovaný vhodný výstup, ktorým užívateľovi odpovie.

Chatbot je systém, ktorý je k tejto práci implementáciou najviac podobný. V práci budeme používať na spracovanie vstupu užívateľa pattern matching, pomocou ktorého sa budeme snažiť zistit', aká odpoved' by bola pre užívateľa najvhodnejšia. Postavy, ktoré sa nachádzajú vo svete budú tiež obsahovať databázu možných odpovedí, z ktorých budeme vyberať vhodnú odpoved' na otázku. V tejto práci sa zameriavame na vyhľadávanie kľúčových slov zo vstupu od užívateľa, ktoré budú našimi paternami. Vďaka nim budeme vyhľadávať vhodný výstup z databázy uloženej pre každé NPC samostatne. Tento systém sa ale bude líšiť v niektorých bodoch, ktoré sú popísané v časti 1.2.

2. Ciele práce

Cieľom tejto práce je navrhnúť a implementovať experimentálny systém so stochastickým dialógovým správaním pre herné systémy. Bude navrhnutý pre prostredie Unreal Engine 4 ako herný plugin. Jeho súčasťou je podpora pre rôzne platformy (PC, konzoly, mobily, atď.). Jadro aplikácie bude vytvorené v jazyku C++ s podporou použitia v Blueprint kóde. Systém bude navrhnutý modulárne, aby používateľ mohol funkciu jednoducho rozšíriť bez zásahu do jadra systému a ľahko integrovať do herného projektu.

Úlohou systému je vyhľadávať najvhodnejšie dialógové odpovede pre herné vstupy a podporovať herné mechaniky projektu. V práci bude integrovaný modulárny systém, ktorý umožní vývojárovi vybrať si algoritmus pre vyhľadávanie, prípadne generovanie, odpovedí. Jeho súčasťou bude implementovaný predvolený algoritmus, založený na vyhľadávaní pomocou klíčových slov (*Keywords analysis*).

Súčasťou práce bude vytvorené herné prostredie na odskúšanie a testovanie funkciu s jednoduchým používateľským rozhraním. Súčasťou testovacieho balíčka bude dátová množina, ktorá bude obsahovať dialógové repliky pre herný dej. Testovanie systému bude prebiehať v etapách, kde v každej etape použijeme skupinu ľudí, pričom tieto skupiny budú navzájom disjunktné. Aplikácia bude testovaná na platforme

Windows v Unreal Editore, v ktorom bude práca aj prezentovaná.

Výsledkom práce bude jednoducho prenositeľný program využiteľný pre rôzne Unreal Engine projekty, v ktorých po ich skompilovaní bude možné okamžite využiť plugin bez zásahu do kódu programu.

3. Návrh

V tejto kapitole sa budeme venovať návrhu softvérového diela, implementovaným mechanizmom a algoritmom. V nasledujúcom texte sa zameriame na popisanie samostatných herných súčastí a algoritmov, a v poslednej časti 3.10 popíšeme prácu celého systému.

Systém je navrhnutý aby vybral vhodnú dialógovú repliku z množiny replík, ktoré sú súčasťou NPC postáv. Túto množinu replík rozdeľujeme do logických celkov (dátových tabuľiek). Tie sú reprezentované ako samostatné dátové assety a sú súčasťou herného projektu.

Po spustení systému vygenerujeme množinu slov (termov), ktoré získame z množiny replík a pomocou nich vygenerujeme reprezentácia slovníka pre hru. Termy následne použijeme na korekciu herného vstupu, ktorý použijeme na vygenerovanie klúčových slov a s pomocou algoritmov nájdeme najvhodnejšiu odpoveď.

3.1 Normalizácia textu

Pre normalizáciu textu sme implementovali vlastný algoritmus. Pretože systém má podporovať multi-jazyčnosť, nedokážeme text upraviť do dokonalej podoby. Aby sme toho boli schopní museli by sme poznat doménové znalosti každého jazyka, do ktorého by sme hru prekladali. Z toho

dôvodu sme aplikovali iba základné techniky, ktoré upravia text nasledovne:

- Odstráni všetky špeciálne znaky a čísla
- Zmení veľké písmená na malé
- Rozdelí repliku na slová (termy)

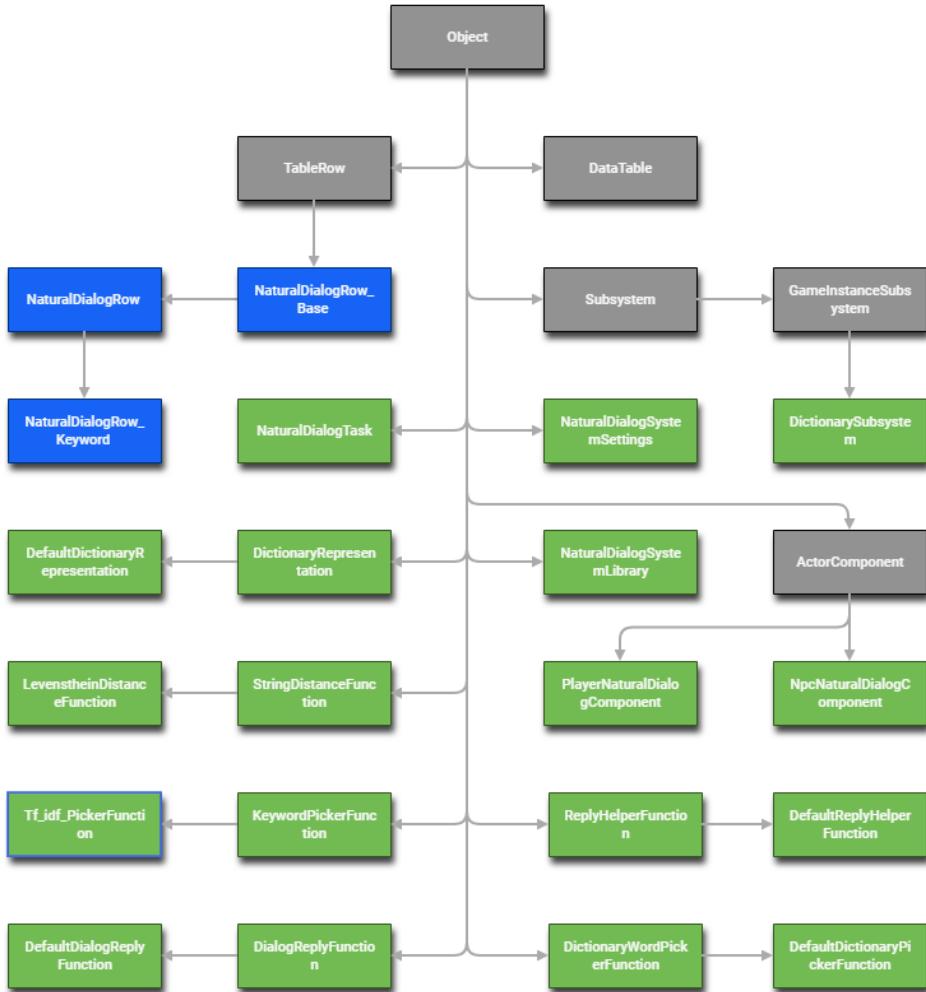
Do projektu sme tiež pridali existujúcu pomocnú knižnicu [19], ktorá dokáže pracovať so znakmi kódovanými v unicode formáte. Vďaka tomu sme schopní normalizovať aj znaky mimo základnej abecedy. Pomocou tejto knižnice transformujeme veľké písmena na malé, čím dosiahneme jeden z bodov normalizácie.

3.2 Hierarchia implementovaných tried

Pre prácu v prostredie Unreal Engine používame rozsiahlu triednu hierarchiu pripravenú pre vývojárov. Systémový kód používa dedičnosť z tried pripravených v tomto prostredí. Pre jednoduchosť integrácie pluginu do prostredia Unreal Engine 4, je nutné aby každá trieda dedila z existujúcich tried, ktoré obsahujú podpornú funkcionality pre manipuláciu v tomto prostredí. Vďaka tomu sa nemusíme v našom systéme staráť o ošetrovanie nízko úrovňových problémov spojených so správou pamäte a optimalizácie.

Obrázok 3.1 znázorňuje hierarchickú implementáciu tried, ktorá je rozdelená do troch kategórii. Triedy zobrazené šedou farbou sú implementované prostredím UE4 a triedy zobrazené zelenou farbou sú súčasťou nášho systému. Modré obdĺžniky predstavujú pripravené štruktúry

použité v dátových tabuľkách, ktorých využite popíšeme v časti 3.4. Šípky na obrázku znázorňujú dedičnosť tried kde koreňom je trieda *Object*.



Obr. 3.1: Triedna hierarchia jadra systému

Triedy v UE sa delia na dve hlavné kategórie *Object* a *Actor*. *Object* je základný typ triedy, ktorý je v hierarchickej architektúre UE koreňom a zároveň s ním Unreal Editor dokáže pracovať. *Actor* je špeciálny typ triedy, ktorá dedí z triedy *Object*. Je viazaný na herný level a v jeho vlastnostiach musí byť definovaná transformácia, ktorá definuje kde v hernom

svete sa inštancia triedy *Actor* nachádza. Súčasťou triedy *Actor* je tiež vstavaná replikácia pre hry s viacerými hráčmi. *Object* tieto súčasti neobsahuje a slúži ako súčasť herného systému bez reprezentácie vo svete. V tomto systéme budeme pracovať predovšetkým s triedami *Object*, ktoré nepotrebuju reprezentáciu v hernom svete, ale budú potrebovať systémovú replikáciu, ktorú musíme doplniť.

3.3 Natural Dialog system settings

Triedy obsahujie nastavenia pluginu, ktoré sú registrované a použité, ak je systém súčasťou UE projektu. Tieto nastavenia sú ukladané do config súboru v projekte a obsahujú globálne nastavenia, pomocou ktorých si vývojár dokáže plugin prispôsobiť. Po registrácii triedy vznikne singleton, ktorý vieme jednoducho editovať a pristupovať k nemu v prostredí Unreal Editor.

Ukážka príkladu implementácie nastavení nájdeme v časti 4.3.1. Na obrázku 4.14 vidíme príklad zobrazenia nastavení pluginu v Unreal Editor, kde môžeme editovať premenné a systém si prispôsobiť. V prípade potreby vieme nastavenia jednoducho rozšíriť.

3.4 Data Table

Trieda *Data Table* je definovaná prostredím UE, ktorá má špeciálnu implementáciu, vďaka ktorej nemusíme dedit' a rozšíriť kód pre náš systém a zároveň vytvára jeho základ. Trieda je používaná ako dátový asset a pristupujeme k nemu, ako k úložisku herných dát pre dialógy, ktoré sú reprezentované v tabuľke. Rozhodli som sa pre tento asset z dôvodu

jeho jednoduchej implementácie a je využívaný ako databáza replík. Dáta dokážeme jednoducho importnúť a exportnúť do Excel tabuľky, ktorú môžeme samostatne formátovať.

Pretože na herných aplikáciách pracuje veľké množstvo ľudí s rôznym zameraním, môže tím, zaoberajúci sa editáciou týchto dát, nastavovať dátá mimo prostredia Unreal Editor. Keďže dátový asset sa kóduje ako binárny súbor, tak pri verzovaní projektu môže dôjsť k prepisovaniu údajov v týchto assetoch a narúšať prípadnú prácu medzi jednotlivými osobami, ktoré pracujú na rovnakom súbore. Samozrejme sa to môže stať aj v tomto prípade, ale v nežiadúcom výsledku máme k dispozícii kópiu dát v exportovanej tabuľke, ktorú uchováme v externom súbore. Ďalšou výhodou je, že členovia tímu zaoberajúci sa editáciou dátových tabuľiek, nepotrebuju detailne poznat' prostredie Unreal Editor.

Pri implementovaní novej tabuľky nemusíme dedit' a rozširovať kód existujúcej triedy. Vytvorili sme štruktúry s premennými, ktoré slúžia na reprezentáciu jedného riadku tabuľky a budú používané vo všetkých dátových tabuľkách určených pre prácu s našim systémom. Tieto štruktúry sú zobrazené na obrázku 3.1 modrou farbou.

Na obrázku 3.2 je príklad tabuľky, ktorá v každom riadku obsahuje tieto hodnoty (číslo riadku, unikátny identifikátor riadku a samotnú štruktúru). Prvé dva parametre sú definované prostredím UE, vďaka ktorým môžeme v kóde riadky iterovať, vyhľadávať a upravovať. Ostatné parametre sú pridané podľa premenných v štruktúre, ktorú sme vytvorili a priradili tabuľke. V tomto prípade používame štruktúru s názvom `FNaturalDialogRow_Keyword`.

Pole klíčových slov *Keywords* používame pre mapovanie k hernému vstupu na výhodnocovanie, či vstup od užívateľa predstavuje otázku *Ask*,

Obr. 3.2: Príklad dátovej tabuľky

ktorú môžeme položiť. Pole odpovedí *Answer* obsahuje rôzne možnosti odpovedania postavy, pričom každá z nich obsahuje textovú aj zvukovú formu. Všetky textové hodnoty sú typu *FText*, ktorý je definovaný prostredíom UE a podporuje lokalizáciu. Vďaka tomu bude náš systém automaticky podporovať multi-jazyčnosť. Vývojár definuje, ktoré jazyky bude hra podporovať a tieto repliky bude musieť do týchto jazykov preložiť. Typ zvukovej formy je tiež definovaný prostredíom UE a tak isto ako aj textová forma, aj zvuková forma podporuje lokalizáciu.

Pole dialógových úloh *Dialog Tasks* popíšeme v časti 3.8. Dátové tabuľky môžu byť použité vo viacerých komponentoch, aby sme sa mohli opýtať otázku viacerých postáv. Pole dialógových tabuliek *Dialog Tables* obsahuje tabuľky, ktoré sa po použití repliky automaticky pridajú, alebo odstraňujú zo systému pre konverzáciu s NPC postavou. Vďaka tomu vieme rozširovať, alebo uzatvárať dialógovú linku pre NPC postavu.

3.5 Dictionary Subsystem

Trieda *Dictionary Subsystem* je dedená z mohutnej rodiny subsystémov, ktoré UE implementuje pre rôzne herné účely. V tomto prípade dedíme zo subsystému *Game Instance*, ktorý je unikátny tým, že je udržiavaný počas celého behu programu. Triedy v Engine majú rozdielnú životnosť, napríklad pri prechode medzi levelmi, kedy väčšinu vytvorených triednych inštancií odstráni Garbage Collector. Aplikácia pre každú vytvorenú triedu, zdedenú z rodiny subsystémov, vytvorí inštanciu, ktorá je singleton a jej životnosť určuje implementácia daného subsystému. V prípade Game Instance subsystému, počas celého behu aplikácie.

Úlohou tejto triedy je automatizovaný proces, ktorý po spustení a inicializácii, vyhľadá všetky dátové tabuľky, ktoré používajú ako štruktúru riadka *NaturalDialogRow*. Po vyhľadaní všetkých vyššie uvedených dátových tabuliek spustí rekonštrukciu slov (termov) z replík, ktoré sa nachádzajú v týchto tabuľkách a vytvorí si objekt pre internú reprezentáciu slov v slovníku, ktorý systém používa počas celého behu programu. Slová nachádzajúce sa v slovníku budú používané pre korekciu vstupných dát od užívateľa, alebo pre vyhľadávanie kl'účových slov.

3.5.1 Reprezentácia slovníka

Slovník je reprezentovaný triedou *Dictionary Representation*. Po rekonštrukcii slov z replík, každý term zaregistrujeme do singleton triedy *Dictionary Representation* pomocou funkcie, ktorá je znázornená vo výpise 3.1.

```
0 virtual void RegisterWord(const FString& Word,  
                           const UDataTable* FromDataTable) = 0;
```

Výpis 3.1: Zápis virtuálnej funkcie *Register Word*

Trieda ukladá vstupný term *Word* do triednej premennej *B*. V poli sa nachádza mapa hodnôt, kde klúčom je term *Word* a hodnota je štruktúra dát, ktorú popíšem v algoritme 3.1.

Algoritmus 3.1 Registrácia termov v slovníku

- Ak term *t* je nový:

1. Vytvorí sa nová štruktúra *D*, ktorá parametrizuje term *t*.
2. Nájdeme dĺžku *l* pre term *t*.
3. Podľa dĺžky *l* vyhodnotíme index *i* pre pole *B*:

$$i = (l - 1).$$

4. Slovo zaregistrujeme do mapy, kde klúčom je term *t* a hodnota je štruktúra *D*. Táto mapa sa nachádza v poli *B* na indexe *i*.

- Ak term *t* už existuje:

1. Nájdeme referenciu na štruktúru *D* pre term *t*.
 2. Upravíme dátu v štruktúre pre nový výskyt termu *t*.
-

3.6 Natural Dialog system component

Komponenty sú triedy, ktoré majú náväznosť na triedu *Actor*. Sú ich súčasťou a ich životný cyklus určuje životnosť inštancie triedy *Actor*. Vďaka ich modularite sú používané na uzavorenie programátorského logického celku a sú použité ako súčasť objektu.

V našom prípade uzatvárame logickú funkciu dialógového systému, ktorá je súčasťou avatara. Obsahuje nastavenia pre každú inšanciu triedy *Actor*, ktoré dokážeme upravovať v editore, ale zároveň aj počas behu programu. Komponent bude používaný ako vstupná brána, kde bude začínať proces spracovania vstupu od užívateľa až po vygenerovanie výstupu.

Týmto dosiahneme, že každá reprezentácia NPC má vlastné nastavenia pre dialógový systém, ktoré využíva na vygenerovanie výstupu pre hráča. Zároveň budú dátu pre NPC načítané v pamäti počas celej existencie inštancie objektu v hernom svete. V systéme budú existovať dva druhy komponentov:

- `NPCNaturalDialogComponent` (ďalej ako *NPC komponent*)
- `PlayerNaturalDialogComponent` (ďalej ako *Player komponent*)

3.6.1 NPC komponent

NPC komponent je navrhnutý ako súčasť nehráčskej postavy. Jeho implementácia umožňuje vývojárovi nastaviť počiatočné vlastnosti pre NPC, ktoré budú použité pri prvotnej komunikácii s hráčom. V jeho nastaveniach nájdeme predvolené dátové tabuľky, ktoré definujú akú komunikáciu môže hráč viest s NPC pri prvom stretnutí. Tieto nastavenia sme popísali v časti 4.3.2. Na začiatku systému bola táto funkciu spojená s kódom, ktorý sa teraz nachádza v Player komponente. Kvôli neskoršej implementácii replikácie do systému, ktorá umožňuje sietovú komunikáciu medzi hernými klientmi, sme túto funkciu museli rozdeliť do dvoch častí, pretože serverové volanie funkcie je možné vykonať iba z objektov, ktoré sú vlastníkom herného klienta. NPC komponent je teda

iba držiteľom dát pre inicializáciu a nevyžadujú žiadnu replikáciu.

3.6.2 Player komponent

Player komponent je implementovaný ako súčasť hráčskej postavy. Riadi celú komunikáciu medzi hráčom a NPC s implementovaným NPC komponentom. Player komponent je replikovaný medzi serverom a herným klientom. Kvôli optimalizácii dát nereplikujeme na ostatných herných klientov, ale iba na vlastníka postavy (dáta komponentu obsahuje iba server a vlastník postavy, nie ostatní herní klienti). Okrem herných dát, obsahuje aj funkcionality na komunikáciu medzi hráčom a NPC. Má teda pripravené funkcie na generovanie odpovedí, registráciu nových dialógových tabuľiek a generovanie pomôcok pre vedenie dialógu s NPC. Nastavenia komponentu a jeho použitie sú popísané v častiach 4.3.2 a 4.6.

3.7 Funkčné triedy

Triedy *String Distance Function*, *Keyword Picker Function*, *Dialog Reply Function*, *Dictionary Word Picker Function*, *Reply Helper Function* (ďalej ako funkčné triedy) sú abstraktné triedy, ktoré sú používané na implementáciu procesu spracovania vstupných dát užívateľa a generovanie výstupu. Ich funkcionality môže byť implementovaná v triedach *Dictionary Subsystem* a herných komponentoch popísaných v časti 3.6, ale z dôvodu vyšej modularity systému sme sa rozhodli túto funkcionality rozdeliť do samostatných tried. Vďaka tejto modularite dokážeme do systému implementovať rôzne spôsoby spracovania dát a rôzne spôsoby generovania výstupu.

3.7.1 Dictionary Word Picker Function

Pri spracovávaní vstupu od užívateľa predpokladáme, že vstup môže obsahovať náhodné chyby. Po rozdelení a normalizovaní vstupu určíme, či sa term t nachádza v našom slovníku. Ak by term t obsahoval chybu v jednom a viac písmen, porovnanie so slovníkom by nám často vracalo neznámy výraz. Z tohto dôvodu sme implementovali abstraktnú triedu *Dictionary Word Picker Function* s virtuálnou funkciou, ktorá je znázorená vo výpise 3.2.

```
0 virtual FString PickWordFromDictionary(const FString& Input) const = 0;
```

Výpis 3.2: Zápis virtuálnej funkcie *Pick Word From Dictionary*

Funkcia používa ako vstupný parameter term t , ktorý sa snažíme vyhľadátať v našom slovníku a výsledkom je opravený term vrátený so slovníka. Pre riešenie sme navrhli vlastný algoritmus, vďaka ktorému nemusíme prehľadávať celý slovník, ale iba jeho časť. Algoritmus je veľmi úzko spätý so *String Distance Function* triedou, ktorá je popísaná v časti 3.7.2.

Na obrázku 3.3 je znázornená práca algoritmu pre vyhľadávanie najpodobnejšieho slova v slovníku. Začíname vstupným termom t dĺžky m . Túto dĺžku používame pre výber slov v slovníku rovnakej dĺžky, kde v každom cykle túto dĺžku rozširujeme o hodnotu 1. Ak vstupný term t má dĺžku $m = 5$, tak t kontrolujeme so slovami dĺžky m a ak stopovacie kritéria nie sú splnené, tak pokračujeme na slovách dĺžky $(m \pm 1)$. Ak stopovacie kritéria nie sú zase splnené, tak pokračujeme na slovách dĺžky $(m \pm 2)$. Výsledok funkcie je term w s najmenšou chybou c . Ak chyba $c \equiv 0$, tak term t existuje v slovníku a vrátíme ho ako výsledok w . Ak neexistuje term t v slovníku ($c \neq 0$), vrátíme term w s najmenšou chybou c .

Ked'že ide iba o pomerne náhodné prípady, kedy užívateľ zadá chybný vstup, tak algoritmus skončí už v prvom cykle a prehľadá iba slová dĺžky m . Ak by sme prehľadávali celý slovník s n slovami a k slovami s dĺžkou m , tak algoritmická náročnosť by bola $O(n)$ namiesto $O(k)$.



Obr. 3.3: Algoritmus pre vyhľadávanie slov v slovníku

Stopovacie kritéria

Aby algoritmus pracoval efektívne, v každom cykle sa skontrolujú stopovacie kritéria, či existuje možnosť, že slová s rozdielnou veľkosťou (napr. $m \pm 1$) môžu obsahovať lepší výsledok ako slová s veľkosťou m .

Napríklad pre vstupný term padla sa nachádzajú v slovníku vhodné

slová **panda** a **spadla**. Ak by sme kontrolovali term iba slovami s veľkosťou m , je dost' pravdepodobné, že dospejeme k chybe pretože hodnota chyby c pre slovo **spadla** je menšia.

Z tohoto dôvodu jedno zo stopovacích kritérií je, že hodnota poslednej najmenšej nájdenej chyby c , nesmie byť väčšia ako rozdiel m a slova w s dĺžkou m_2 :

$$c < |m - m_2|.$$

Ak by sa ale toto stopovacie kritérium nesplnilo v n cykloch, tak sme pre optimalizáciu pridali nové stopovacie kritérium. Ak je chyba príliš vysoká a rozdiel slova tiež, tak pre nás nemá význam hľadať slovo s príliš veľkým rozdielom vo veľkosti. Preto sme si definovali konštantu, ktorá popisuje maximálny rozdiel v dĺžke slova, ktorá bude skontrolovaná a pomenovali sme ju *MAXLENDIFF*. Hodnota l predstavuje dĺžky slov, ktoré budeme kontrolovať:

$$(m - MAXLENDIFF) < l < (m + MAXLENDIFF)$$

Ak by sa aplikovalo toto stopovacie kritérium, ale vstupné slovo by malo chybu $c > MAXLENDIFF$, tak sa považuje za nerelevantné pri vyhľadávaní odpovedi a výstupom je prázdný term w .

3.7.2 String Distance Function

Pre vydelenie chyby v terme t sme definovali triedu *String Distance Function*, ktorá implementuje virtuálnu funkciu, ktorá je znázorená vo výpise 3.3.

```
0 virtual uint32 GetStringDistance(const FString& InputA,  
                                const FString& InputB) const = 0;
```

Výpis 3.3: Zápis virtuálnej funkcie *Get String Distance*

Funkcia obsahuje dva stringové parametre, ktoré má za úlohu porovnať. Výsledkom tejto funkcie je numerická hodnota, ktorá vyjadruje rozdiel vstupu *InputA* od vstupu *InputB*. Čím je výsledná hodnota vyššia, tým je rozdiel medzi vstupmi väčší. Pokial' sú vstupné parametre zhodné, funkcia vráti hodnotu 0.

Pre riešenie problému som vybral implementáciu algoritmu pomocou Levenshteinovej vzdialenosť popísanej v časti 1.3.3. Vďaka modularite, ktorú som popísal v časti 3.7 si vývojár v nastaveniach systému vie vybrať vlastný algoritmus, ktorý si môže implementovať.

3.7.3 Keyword Picker Function

Po spracovaní vstupných slov začneme vyberať, ktoré z nich použijeme na generovanie výstupu. Trieda *Keyword Picker Function* je abstraktná trieda s virtuálnou funkciou, ktorá je znázornená vo výpise 3.4.

```
0 virtual TArray< FString> PickKeywords(  
2                                         const UPlayerNaturalDialogComponent* DialogComponent,  
                                         const TArray< FString>& Input) = 0
```

Výpis 3.4: Zápis virtuálnej funkcie *Pick Keywords*

Tf-Idf

V našom riešení sme pre výber klúčových slov použili ohodnocovací algoritmus *Tf-Idf* popísaný v časti 1.3.1, ktorý sme mierne upravili pre zefek-

tívnenie práce, ale hlavne kvôli návrhu celého systému. Hodnota $Tf(t, d)$ sa v algoritme počíta ako počet termov t v dokumente d :

$$Tf(t, d) = t/n.$$

Algoritmus sme upravili a jeho výsledkom je hodnota počtu termov t v množine dokumentov, ktoré máme priradené pre NPC postavu. Aj keď repliky rozdeľujeme na logické časti (môžu to byť dejové línie, druhy konverzácií, atď.) do rôznych dátových tabuľiek, tak pre prácu algoritmu $Tf-Idf$ k nim pristupujeme ako k jednému veľkému dokumentu s replikami, pretože sa snažíme vybrať vhodný výsledok z celej množiny dátových tabuľiek, ktorú obsahuje NPC postava.

Reprezentácia slov v *Dictionary Subsystem* popísaná v časti 3.5 obsahuje dátá, ktoré parametrizujú term t , vieme veľmi efektívne tieto parametre využiť v tomto algoritme. Súčasť týchto parametrov je počet výskytov slova a zároveň v ktorých dokumentoch sa toto slovo nachádza. Zároveň subsystém počíta počet slov v dokumentoch a hodnota je prezentovaná ako statická premenná n , ktorú vieme prečítať zo subsystému. Vďaka týmto parametrom nemusíme počítať žiadne pomocné hodnoty pre $Tf(t, d)$ funkciu a vieme použiť vypočítané parametre zo subsystému.

Hodnota $Idf(t)$ je nezmenená a teda je počítaná podľa pôvodného algoritmu popísanom v časti 1.3.1. Vo funkcii $Idf(t)$ tiež používame parametre z *Dictionary Subsystem*, kde čítame hodnotu, v ktorých dokumentoch sa nachádza term t .

Výsledkom funkcie $Tf-Idf(t)$ je skalárna hodnota pre term t . Po ohodení všetkých vstupných termov zo vstupnej postupnosti T , určíme množinu klúčových slov K . Túto množinu tvoria termy z postupnosti T s

najvyšším ohodnotením $Tf\text{-}Idf(t)$ a ich počet je závislý od veľkosti množiny T .

3.7.4 Dialog Reply Function

Po definovaní množiny klúčových slov K začíname s procesom vyberania odpovede pre užívateľa. *Dialog reply function* je abstraktná trieda v ktorej máme definovanú virtuálnu funkciu znázornenú vo výpise 3.5.

```
0 virtual bool GenerateReply(  
1     const FString& Sentence,  
2     const UNpcNaturalDialogComponent* NpcNaturalDialogComponent,  
3     FNaturalDialogResult& ResultAnswer) = 0;
```

Výpis 3.5: Zápis virtuálnej funkcie *Generate Reply*

Jej výsledkom je boolean hodnota, ktorá definuje, či funkcia našla odpoveď na vstupnú otázku. Ak funkcia nájde odpoveď, nastaví do parametra `ResultAnswer` nájdenú repliku.

Pre výber najvhodnejšej odpovede sa použije množina klúčových slov K a skombinujeme ju s doplnkovým ohodnocovacím algoritmom. Množinu K použijeme na výber najvhodnejších možných odpovedí z dátových tabuľiek.

Ako prvé si definujeme novú množinu dátových tabuľiek D , ktorá bude najväčšou spoločnou množinou pre každú množinu O_k . O_k predstavuje množinu dátových tabuľiek, v ktorých sa nachádza klúčové slovo k , pričom $k \in K$. Týmto dosiahneme vytriedenie relevantných dátových tabuľiek, ktoré budeme kontrolovať v nasledujúcom kroku.

Po definovaní množiny D začína proces kontrolovania replík r , uložených v dátových tabuľkách v množine D . Výsledkom je štruktúra dát,

ktorá definuje akú veľkú zhodu majú klúčové slová z množiny K s replikami.

Po výbere najrelevantnejších replík sa aplikuje doplnkový ohodnocovací algoritmus, ktorý používa parametrizáciu každej repliky na vybranie tej najvhodnejšej ktorú označíme ako R .

Ešte pred vrátením výsledku repliky R , upravíme jej parametre pre ohodnocovací algoritmus. Algoritmus obsahuje parametre pre každú odpoved' v podobe metriky, ktorú upravujeme v prípade, že replika r je použitá pre výsledok R . Výsledkom algoritmu je pravdepodobnosť $P(r)$, ktorá definuje s akou pravdepodobnosťou replika r bude použitá ako výsledná replika R . Keďže v systéme sa môže nachádzat' viac replík, ktoré odpovedajú na rovnakú otázku rôznym spôsobom, bez ohodnocovacieho algoritmu by výsledok R bol vždy rovnaký. Vďaka ohodnocovaciemu algoritmu dokážeme upravovať výsledok R pomocou rôznych veličín ako je znáhodnenie odpovedi, opakovateľnosť odpovedí a pod.

3.7.5 Reply Helper Function

V herných systémoch je niekedy najdôležitejší príbeh, ktorý je veľa krát tvorený rôznymi dialógmi. V našom systéme sa snažíme aby mal hráč možnosť si tento príbeh prispôsobovať a rozširovať podľa seba. Niekedy sa ale hráč môže dostať do situácie, kedy nevie ako sa pýtať ďalej a tým rozšíriť herný dej, pretože nevie čo všetko hra môže skrývať'. Práve z tohto dôvodu sme sa rozhodli implementovať doplnkovú abstraktnú funkčnú triedu *Reply Helper Function* s virtuálnou funkciou, ktorá je znázornená vo výpise 3.6.

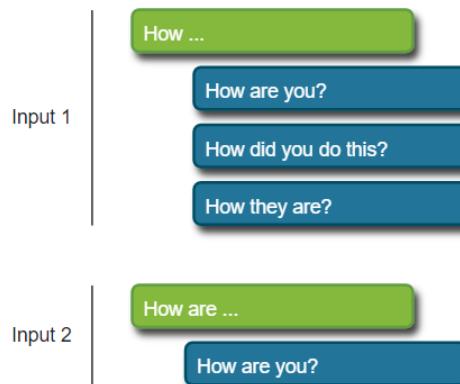
```

0 virtual bool FindAskOptions(
1     const FText& Input,
2     const UNpcNaturalDialogComponent* NpcNaturalDialogComponent,
3     TSet<FString>& OutOptions) = 0;

```

Výpis 3.6: Zápis virtuálnej funkcie *Find Ask Options*

Jej cieľom je implementovať algoritmus, ktorý by našiel všetky relevantné repliky k vstupu, ktorý hráč zadáva.



Obr. 3.4: Príklad výstupu z funkcie *Find Ask Options*

Na obrázku 3.4 vidíme príklad použitia triedy *Reply Helper Function*.

V hornej časti obrázka (Input 1) vidíme vstup od užívateľa v zelenom obdĺžniku, keď napísal slovo How a funkcia vrátila všetky relevantné výsledky pre užívateľov vstup z databázy replík, ktoré sú obsiahnuté v rozhovore s NPC. V druhej časti (Input 2) užívateľ pokračoval v písaní otázky a funkcia vrátila presnejšie výsledky k jeho otázke.

Vďaka tomuto systému dávame užívateľovi pomôcku, ktoré otázky môže položiť NPC. Implementácia triedy vráti všetky relevantné výsledky pre vstup užívateľa a je na strane vývojára, či chce túto funkcionalitu použiť, alebo nie. Tak isto má vývojár možnosť zvoliť si grafické prevedenie

týchto pomocných replík. Pre riešenie funkcionality sme navrhli vlastný algoritmus popísaný v algoritme 3.2.

Algoritmus 3.2 Pomocná dialógová funkcia

1. Vstup rozdelíme na postupnosť otázok $Q = q_0, q_1, \dots, q_n$. Vety zakončené špeciálnym znakom (. ? !) považujeme za uzatvorené a algoritmus spracováva údaje iba o poslednej neuzatvorenej otázke q_n .
2. Otázku q_n rozdelíme na postupnosť termov $T = t_0, t_1, \dots, t_n$. Trieda si tiež pamäta zoznam poslednej spracovanej postupnosti termov, ktorú môžeme označiť ako T_{old} .
3. Pre algoritmus si zadefinujeme postupnosť slov $W = w_0, w_1, \dots, w_n$, kde w_i je slovo na indexe i v replike r , ktorá sa nachádza v dátovej tabuľke.
4. Pre term t_0 vyberieme takú množinu odpovedí, ktorej slovo w_0 je zhodné s termom t_0 . Repliky, ktoré odkontrolujeme, sa nachádzajú v dátových tabuľkách a týkajú sa konverzácie s daným NPC. Výsledkom je množina U_0 .
5. Každý ďalší term t_x z postupnosti T spracujeme týmto spôsobom:
 - (a) Vyberieme množinu odpovedí U_x , ktorá je podmnožinou U_{x-1} .
 - (b) Množinu na nultom indexe sme si zadefinovali zo všetkých replík, ktoré môže hráč použiť. Teraz stačí skontrolovať iba repliky z množiny U_{x-1} , či splňajú podmienku $t_x \equiv w_x$.
 - (c) Ak je podmienka splnená tak replika $r \in U_{x-1} \wedge r \in U_x$. Týmto optimalizujeme, že pri písaní vstupu nemusíme kontrolovať celú množinu replík zo všetkých dátových tabuliek, ale iba podmnožinu U_{x-1} .
6. Ak sa postupnosť termov $t \in T$ zmenila a je rozdielna od množiny T_{old} , skontrolujeme, na ktorom indexe i došlo k zmene a proces spracovania začne od najdeného indexu i .
7. Výsledkom funkcie je množina replík U_n , pre ktorú platí:

$$U_0 \supseteq U_1 \supseteq U_2 \dots \supseteq U_n.$$

3.8 Dialógové úlohy

Aby dialógový systém mohol spolupracovať s ostatnými modulmi projektu, tak sme implementovali triedu *Dialog Task*. V časti 1.2.5 sme spomínali, že systém musí vedieť pracovať s hernými objektami, ktoré si vývojár definuje v projekte. Keďže nevieme o aké objekty ide, snažíme sa pripraviť systém nezávisle na nich. Trieda *Dialog Task* je zdedená z triedy *Object* a jej úlohou bude vykonávať dodatočný herný kód vytvorený vývojárom. Vývojár musí byť schopný herný kód realizovať na strane servera, ale aj na strane klienta. Z toho dôvodu musí byť dialógová úloha spustená na oboch stranách. Dosiahneme to replikáciou, ktorá je podrobnejšie popísaná v časti 3.9.

Vytvorenie inštancie triedy *Dialog Task* vykonáme na strane servera, pričom pomocou vstavanej replikácie v UE sa inštancia zreplikuje na stranu herného klienta. Po vytvorení inštancií zavoláme event *On Task Executed*, ktorý funguje ako virtuálna funkcia, ktorú vývojár definuje podľa svojich potrieb. Keďže dialógová úloha môže existovať neurčitý čas v hernom svete, tak po vykonaní všetkej požadovanej funkcionality, musí vývojár použiť funkciu *Finish Task Execution*, čím v systéme vynúti ukončenie dialógovej úlohy. Po spracovanej požiadavke o ukončení úlohy vývojár môže definovať dodatočnú funkciu pre event *On Task Finished*, ktorý systém zavolá počas procesu zničenia inštancie objektu. Príklad takejto implementácie môžeme vidieť v časti 4.5.

Pre úspešné spustenie dialógovej úlohy, je potrebné priradiť ju do dát v dátovej tabuľke. Po použití repliky v dialógu sa dialógová úloha automaticky spustí. Priradenie dialógovej úlohy pre repliku môžeme vidieť na obrázku 4.18 v žltom rámčeku v poli *Dialog Tasks*.

3.9 Replikácia systému

Pre riešenie hier s viac hráčmi používame vstavanú replikáciu v UE. Hry v tomto prostredí používajú pre komunikácie Client–server model, pričom proces spracovania vstupných dát a generovanie výstupných dát sa vykonáva na strane klienta, ale je vykonaný nad množinou dát, ktoré mu poskytne server. Pre náš systém replikujeme dve triedy Player komponent a dialógové úlohy.

Aby herný klient nemohol spracovávať repliky z dátových tabuľiek ktoré mu nenáležia, musíme tieto hodnoty udržiavať identické so serverom. Player komponent udržuje množinu dátových tabuľiek, ktoré môže používať pri komunikácii s NPC. Preto je potrebné zabezpečiť, aby množina dátových tabuľiek, z ktorých čerpáme údaje, bola totožná na klientskej strane a na strane servera. Preto replikujeme túto množinu medzi serverom a klientom pomocou makra definovanom prostredím UE ako vidíme vo výpise 3.7.

```
0 UPROPERTY(Replicated)
TArray<FDIALOGTABLEHIERARCHY> DialogData;
```

Výpis 3.7: Replikovanie množiny štruktúry s dátovými tabuľkami

Vo výpise 3.7 je znázornená replikácia množiny dátových tabuľiek. Pомocou makra *Replicated* označíme premennú v .h súbore za replikovanú, čím systém automaticky rozozná, že jej hodnota bude posielaná na klientov, čím sa stane identickou ako na serveri. Pre úspešné implementovanie replikácie tejto hodnoty je potrebné pridať túto premennú do funkcie, ktorá zozbiera z triedy množinu premenných, ktoré majú byť replikované, ako to môžeme vidieť na príklad vo výpise 3.8.

```

0 void UPlayerNaturalDialogComponent::GetLifetimeReplicatedProps(
1     TArray<FLifetimeProperty>& OutLifetimeProps) const
2 {
3     Super::GetLifetimeReplicatedProps(OutLifetimeProps);
4
5     // Other clients doesn't need know about data of another player
6     DOREPLIFETIME_CONDITION(UPlayerNaturalDialogComponent, DialogData,
7         COND_OwnerOnly);
8     DOREPLIFETIME_CONDITION(UPlayerNaturalDialogComponent,
9         ActiveExecutionTasks, COND_OwnerOnly);
10 }

```

Výpis 3.8: Replikácia premenných

Vo výpise 3.8 na riadku 5 a 6 vidíme pridanie premenných do replikačnej množiny prostredníctvom makra, pripraveného v UE. Prvý parameter makra je názov triedy a druhý premenná, ktorú budeme replikovať. Tretia hodnota je podmienka, ktorá ak je splnená tak replikácia prebehne. Z dôvodu menšej náročnosti na siet' sme nastavili, aby sa hodnota replikovala iba pre Authoritative klienta, teda ak je vlastník postavy herný klient.

```

0 UFUNCTION(Server, Reliable, WithValidation)
1 void Server_RegisterDialogData(
2     const UNpcNaturalDialogComponent* NpcNaturalDialogComponent,
3     UDataTable* DialogTable);
4
5 UFUNCTION(Server, Reliable, WithValidation)
6 void Server_UnregisterDialogData(
7     const UNpcNaturalDialogComponent* NpcNaturalDialogComponent,
8     UDataTable* DialogTable);

```

Výpis 3.9: Serverové funkcie

Herné dátá sme schopní upravovať pomocou serverových funkcií zoobrazených vo výpise 3.9. Vďaka nim si môže herný klient vyžiadat o ich zmenu. Napríklad ak chceme v hernom bode pridať novu dialógovú tabuľku pre komunikáciu hráča s NPC, použijeme serverovú funkciu, čím vyžiadame na serveri, aby pridal túto tabuľku do množiny tabuliek. Aby sme nemuseli čakať na replikáciu dát zo servera, pridáme túto tabuľku aj na strane herného klienta, čím vzniká predikcia herných dát. V prípade, že server nepovolí pridanie dátovej tabuľky, tak vďaka replikácii sa hodnota na strane klienta opraví.

Okrem dátových tabuliek je pre nás dôležité replikovať aj dialógové úlohy. Dialógová úloha obsahuje funkcionality, ktorá môže závisieť na vykonaní na strane servera, alebo herného klienta. Trieda je dedená z triedy *Object*, ktorá sice podporuje natívnu replikáciu enginu, ale je potrebné upraviť časť implementácie pre jej aktivovanie. Po spustení replikácie v triede *Dialog Task*, sa každá inštancia triedy vytvára zároveň na serverovej strane a na strane herného klienta. Vďaka tomu sme schopný volať pripravený event *On Task Executed* zobrazený na obrázku 4.20. Užívateľ je schopný pomocou dodatočnej funkcionality zistíť, či sa kód vykonáva na strane servera, alebo na strane klienta a podľa toho definovať implementáciu dialógovej úlohy. Bez tejto replikácie by užívateľ nebol schopný vykonávať herný kód na strane servera, čím by sa tento kód stal nerelevantným.

3.10 Práca systému

V tejto časti sa budeme venovať spracovaniu herných dát systémom od herného vstupu až po vyhodnotenie výstupu. V algoritme 3.3 a 3.4 sme

popísali prácu celého systému.

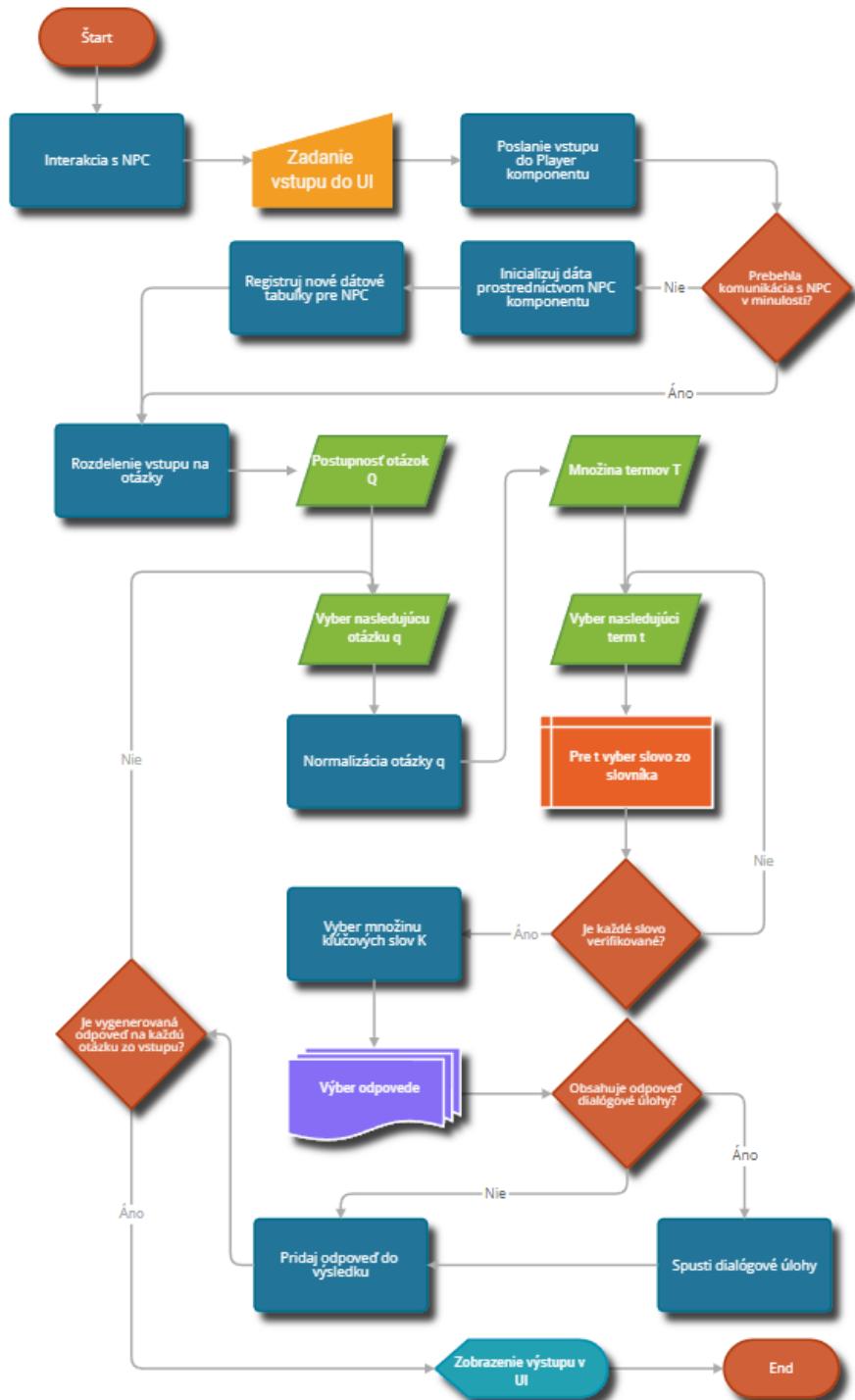
Algoritmus 3.3 Práca systému časť 1.

1. Začíname interakciou hráča s NPC. Táto interakcia je definovaná herným prostredím vývojára. Vývojár môže napr. vytvoriť User Widget s elementom pre vstup z klávesnice. Príklad interakcie môžeme vidieť v prílohe A.1
 2. Použitím klávesnice zadáme požadovanú otázku (vstup IN). Príklad môžeme vidieť v prílohe A.2.
 - V prílohe A.2 vidíme použitie implementovanej funkčnej triedy popísanej v časti 3.7.5. Počas zmeny vstupného reťazca posielame do funkcie požiadavku na vygenerovanie možných otázok. V prílohe tiež vidíme možnosť implementácie tejto funkcionality, ale vývojár si ju môže upraviť. Nemusí všetky vyhľadané repliky zobrazovať v UI ale uváži, ktoré z nich zobrazí a akým spôsobom.
 3. V Player komponente zavoláme funkciu *Generate Reply*, do ktorej ako vstupný parameter pošleme vstup IN a NPC komponent, s ktorým viedieme dialóg:
 - Ak NPC komponent neboli pred tým použitý pre generovanie dialógovej odpovedi, tak Player komponent si vyžiada množinu tabuľiek z NPC komponentu vyznačených na obrázku 4.15. Tieto tabuľky zaregistrovať Player komponent pomocou serverovej funkcie a bude ich používať ako zdroj replík pre odpovede.
 - Ostatné dialógové tabuľky môžeme registrovať nezávisle z herného kódu.
 4. Vstup IN rozdelíme na postupnosť otázok Q, kde $q \in Q$.
-

Algoritmus 3.4 Práca systému časť 2.

5. Pre každú otázku q z postupnosti Q vyberieme najvhodnejšiu repliku R nasledujúcim spôsobom:
 - (a) Otázku q normalizujeme a rozdelíme na množinu termov T , kde term $t \in T$.
 - (b) Pre každý term t množiny T nájdeme prislúchajúcu hodnotu zo slovníka $w \in W$ podľa definície z časti 3.7.1.
 - (c) Z množiny W vyhodnotíme množinu kl'účových slov K podľa definície z časti 3.7.3.
 - (d) Použitím množiny K vyberieme najvhodnejšiu repliku R podľa definície z časti 3.7.4.
 6. Ak replika R obsahuje dialógovú úlohu, systém aktivuje úlohu na strane servera a herného klienta.
 - Dialógová úloha sa ukončuje nezávisle od dialógového systému podľa definície herného kódu v triede *Dialog Task*, implementovanej vývojárom.
 7. Výslednú repliku R pridáme do výstupu OUT.
 8. Výstup funkcie OUT je pole štruktúr replík, ktoré vývojár môže zobrazit' v UI.
-

Na obrázku 3.5 vidíme postupnosť procesov pri práci, ktorý sme popísali v algoritme 3.3 a 3.4.



Obr. 3.5: Workflow diagram systému

4. Použitie systému

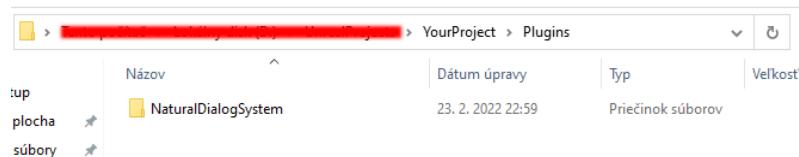
V tejto časti sa budeme venovať zhrnutiu celého systému od implementácie pluginu do herného projektu, nastavenie a použitie systému, a proces vygenerovania výstupných dát zo vstupu od užívateľa. Pri implementácii pluginu do systému, sa zameriame na implementáciu v C++ kóde, ale popíšeme aj možnosť implementácie v Blueprint kóde.

4.1 Integrovanie pluginu

Pre použitie pluginu je potrebné vykonať presný postup pred jeho implementáciou do herného projektu. Všetky kroky pre jeho integráciu sú vynútené prostredím Unreal Engine. Pri vynechaní hociktorého kroku, editor nerozpozná plugin a nebudeme ho vedieť v projekte použiť.

Postup Integrácie:

1. Naše zdrojové súbory nakopírujeme do zložky *Plugins*, ktorá sa nachádza v zložke projektu. Na obrázku 4.1 vidíme príklad takejto zložky, kde zložka *YourProject* predstavuje názov projektu.



Obr. 4.1: Príklad umiestnenia zdrojových súborov

2. Po skopírovaní súborov povolíme projektu používať tento plugin. V textovom editore otvoríme projektový súbor *YourProject.uproject*, ktorý sa nachádza v zložke projektu. V súbore, v časti *Plugins*, doplníme sekciu, ktorá je vyznačená na obrázku 4.2.

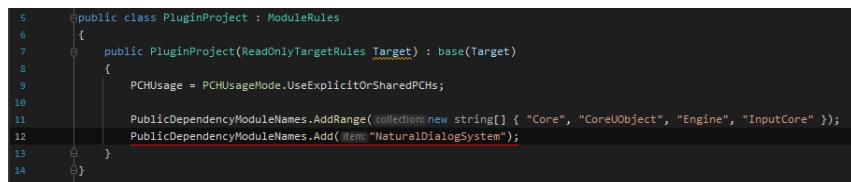
```
1  {
2      "FileVersion": 3,
3      "EngineAssociation": "4.27",
4      "Category": "",
5      "Description": "",
6      "Modules": [
7          {
8              "Name": "PluginProject",
9              "Type": "Runtime",
10             "LoadingPhase": "Default",
11             "AdditionalDependencies": [
12                 "Engine"
13             ]
14         }
15     ],
16     "Plugins": [
17         {
18             "Name": "NaturalDialogSystem",
19             "Enabled": true
20         }
21     ]
22 }
```

Obr. 4.2: Implementovanie pluginu do projektového súboru

Doplnkové kroky pre implementáciu v C++

Pri vytváraní projektu si vývojár môže zadefinovať, či chce projekt tvoriť iba pomocou Blueprint kódu, alebo chce kód rozširovať pomocou C++. Ak je projekt tvorený iba pomocou Blueprint kódu, tieto dva kroky netreba splniť. Ak vývojár pracuje v C++ a chce používať kódy pluginu v tomto prostredí, je potrebné vykonať tieto doplnkové kroky pre funkčnú implementáciu:

1. Doplníme zdrojový kód súboru *Build.cs* o riadok, vyznačený na obrázku 4.3. V projekte každý herný modul obsahuje build súbor, preto je potrebné ho pridať do všetkých modulov, ktoré budú využívať zdrojové súbory pluginu. Pre jednoduchosť ho môžeme zatiaľ implementovať do základného herného modulu, ktorý je automaticky vytvorený s projektom a nachádza sa v zložke *YourProject/Source-YourProject*.



```
5     public class PluginProject : ModuleRules
6     {
7         public PluginProject(ReadOnlyTargetRules Target) : base(Target)
8         {
9             PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;
10            PublicDependencyModuleNames.AddRange(collection: new string[] { "Core", "CoreUObject", "Engine", "InputCore" });
11            PublicDependencyModuleNames.Add(item: "NaturalDialogSystem");
12        }
13    }
14 }
```

Obr. 4.3: Implementovanie pluginu v súbore Build.cs

2. Doplňme zdrojový kód súboru *YourProject.Target.cs* o riadok, vyznačený na obrázku 4.4, ktorý sa nachádza v zložke *YourProject/Source*.

```
6     public class PluginProjectTarget : TargetRules
7     {
8         public PluginProjectTarget( TargetInfo Target ) : base(Target)
9         {
10            Type = TargetType.Game;
11            DefaultBuildSettings = BuildSettingsVersion.V2;
12            ExtraModuleNames.AddRange( collection:new string[] { "PluginProject" } );
13            ExtraModuleNames.Add( item: "NaturalDialogSystem" );
14        }
15    }
16
```

Obr. 4.4: Implementovanie pluginu v súbore *Target.cs*

3. Ako posledný krok je doplnenie zdrojového súboru *YourProjectEditor.Target.cs* o riadok, vyznačený na obrázku 4.5, ktorý sa nachádza v rovnakej zložke ako predchádzajúci súbor.

```
6     public class PluginProjectEditorTarget : TargetRules
7     {
8         public PluginProjectEditorTarget( TargetInfo Target ) : base(Target)
9         {
10            Type = TargetType.Editor;
11            DefaultBuildSettings = BuildSettingsVersion.V2;
12            ExtraModuleNames.AddRange( collection:new string[] { "PluginProject" } );
13            ExtraModuleNames.Add( item: "NaturalDialogSystem" );
14        }
15    }
16
```

Obr. 4.5: Implementovanie pluginu v súbore *Editor.target.cs*

4.2 Implementácia pluginu

Po splnení integračných krokov popísaných v časti 4.1, je plugin pripravený na použitie v našom projekte. Editor ho rozpozná a ponúkne nám funkciu implementácie do hry, popíšeme implementovanie komponentov popísaných v časti 3.6.

4.2.1 Integrácia NPC komponentu

NPC komponent, popísaný v časti 3.6.1, implementujeme ako súčasť NPC postavy. Pre príklad implementácie sme vytvorili triedu NpcCharacter.

Implementáca v C++

```
10     UCLASS()
11     #0 derived blueprint classes
12     class PLUGINPROJECT_API ANpcCharacter : public ACharacter
13     {
14         GENERATED_BODY()
15
16     public:
17     ANpcCharacter();
18
19     /** Natural dialog component definition in NPC character */
20     UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category=Character)
21     UNpcNaturalDialogComponent* NpcComponent;
22 }
```

Obr. 4.6: Deklarácia NPC komponentu v .h súbore

Na obrázku 4.6 vidíme deklaráciu komponentu pre triedu NpcCharacter v .h súbore a na obrázku 4.7 vidíme vytvorenie objektu v konštruktore

```

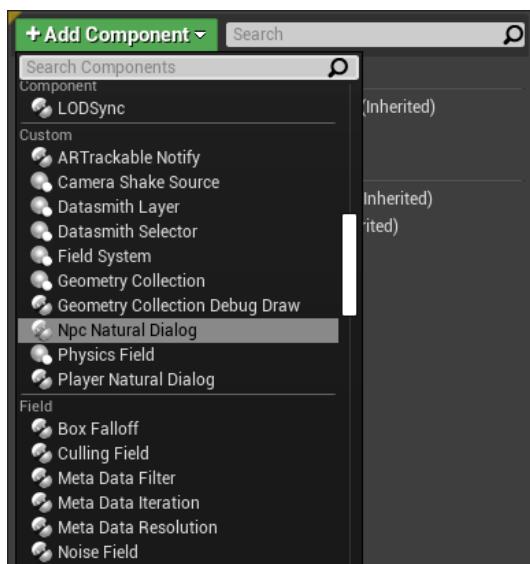
5 ANpcCharacter::ANpcCharacter()
6 {
7     NpcComponent = CreateDefaultSubobject<UNpcNaturalDialogComponent>("Npc Component");
8 }
9

```

Obr. 4.7: Vytvorenie NPC komponentu v .cpp súbore

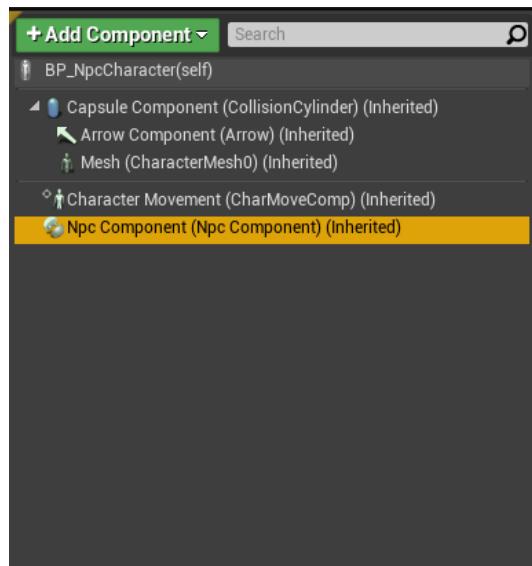
tryed pre túto premennú v .cpp súbore. Komponent existuje v hre počas existencie NPC charakteru v hernom svete.

Implementáca v Blueprintite



Obr. 4.8: Vytvorenie NPC komponentu v Blueprintite

Ak je hra vytváraná len pomocou Blueprint kódu, tak komponent môžeme pridať triede NpcCharacter v Unreal Editore. Na obrázku 4.8 vidíme menu všetkých komponentov, ktoré môžeme pridať do triedy. Nás Npc komponent je vyznačený šedou farbou.



Obr. 4.9: Úspešná implementácia NPC komponentu

Po úspešnej implementácii komponentu bude zobrazený ako súčasť herného objektu NpcCharacter, ako môžeme vidieť na obrázku 4.9. Od teraz môžeme komponent nastavovať v editore a pracovať s ním v hernom kóde.

4.2.2 Integrácia Player komponentu

Player komponent, popísaný v časti 3.6.2, implementujeme ako súčasť triedy hráčskej postavy. Pre príklad implementácie sme vytvorili triedu PlayerCharacter.

Implementáca v C++

```
11  UCLASS()
12  {
13  class PLUGINPROJECT_API APlayerCharacter : public ACharacter
14  {
15  public:
16  APlayerCharacter();
17  GENERATED_BODY()
18
19  /** Natural dialog component definition in player character */
20  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category=Character)
21  UPlayerNaturalDialogComponent* PlayerComponent;
22  };
23 }
```

Obr. 4.10: Deklarácia Player komponentu v .h súbore

```
5 APlayerCharacter::APlayerCharacter()
6 {
7     PlayerComponent = CreateDefaultSubobject<UPlayerNaturalDialogComponent>("Player Component");
8 }
9 }
```

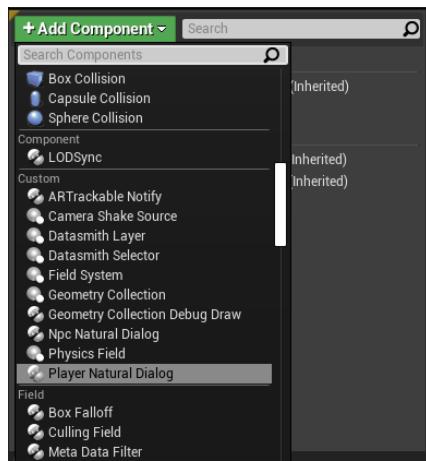
Obr. 4.11: Vytvorenie Player komponentu v .cpp súbore

Na obrázku 4.10 vidíme deklaráciu komponentu pre triedu PlayerCharacter v .h súbore a na obrázku 4.11 vidíme vytvorenie objektu v konštruktore triedy pre túto premennú v .cpp súbore. Ked'že ide o charakter kontrolovaný hráčom, komponent bude existovať počas celej doby hrania hry.

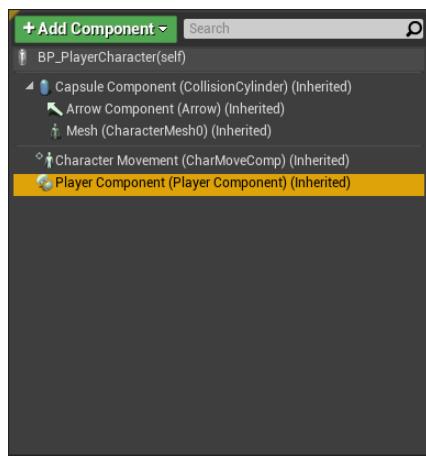
Implementáca v Blueprintite

Ak je hra vytváraná len pomocou Blueprint kódu, tak komponent môžeme pridať triede PlayerCharacter v Unreal Editore. Na obrázku 4.12 vidíme Player komponent vyznačený šedou farbou.

Po úspešnej implementácii komponentu bude zobrazený ako súčasť herného objektu PlayerCharacter, ako môžeme vidieť na obrázku 4.13.



Obr. 4.12: Vytvorenie NPC komponentu v Blueprintite



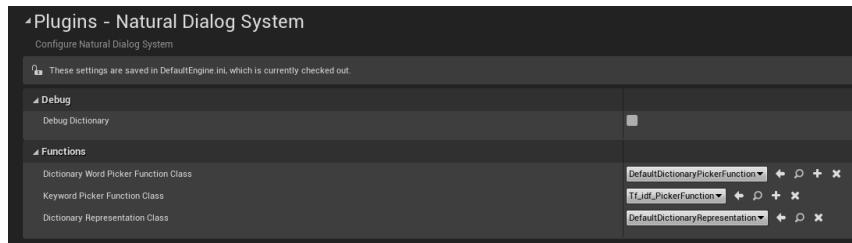
Obr. 4.13: Úspešná implementácia Player komponentu

4.3 Nastavenie pluginu

Nastavenia systému môžeme nájsť na rôznych miestach v editore, pričom ich rozdeľujeme na dve hlavné časti. Globálne nastavenia sa nachádzajú v projektových nastaveniach, ktoré sme popísali v časti 3.3. Ostatné nastavenia sa nachádzajú v komponentoch a sú unikátne pre každý komponent, ktorý sa v hernom svete nachádza.

4.3.1 Globálne nastavenia

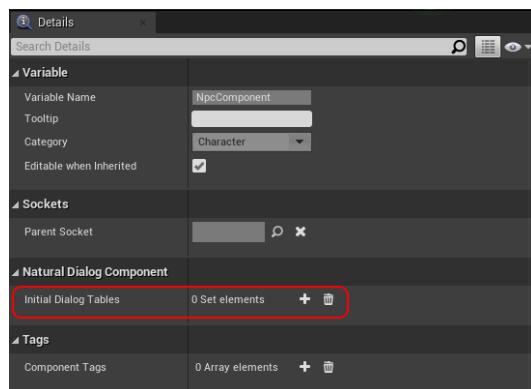
Po integrácii pluginu do projektu z časti 4.1 nám UE umožní meniť nastavenia pluginu v projektových nastaveniach editora. Príklad takýchto nastavení môžeme vidieť na obrázku 4.14. V projektových nastaveniach sa nachádzajú systémové nastavenia, ktoré sa aplikujú na celý systém.



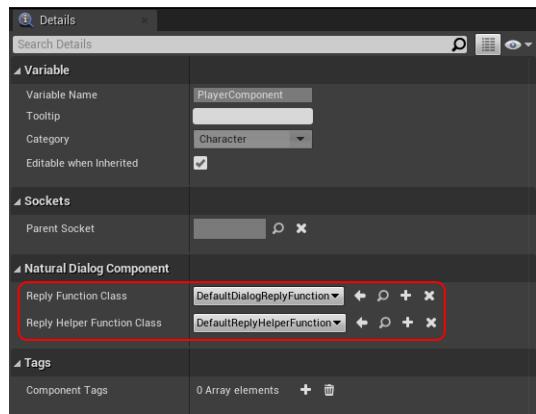
Obr. 4.14: Nastavenia systému v projekte

4.3.2 Nastavenia komponentov

Každý implementovaný komponent obsahuje doplnkové nastavenia, ktoré sa výlučne týkajú postavy. Tu môžeme nastaviť niektoré funkčné triedy, ktoré sme popísali v časti 3.7, a tiež predvolené dátové tabuľky, ktoré sa majú inicializovať pri komunikácii s NPC.



Obr. 4.15: Nastavenia NPC komponentu

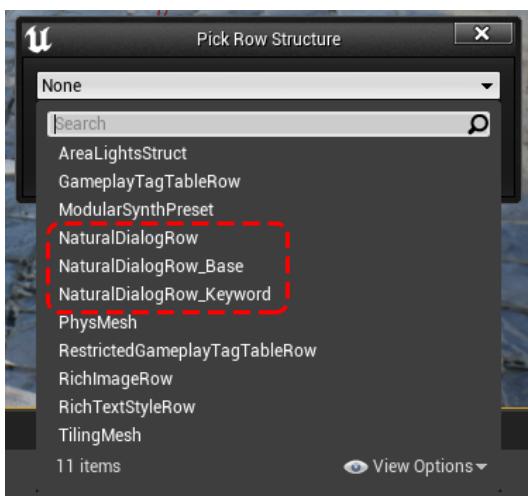


Obr. 4.16: Nastavenia Player komponentu

Na obrázkoch 4.15 a 4.16 vidíme nastavenia komponentov, ktoré definujeme pre každý charakter. Množina Initial Dialog Tables obsahuje dátové tabuľky, ktoré sa použijú na inicializáciu konverzácie medzi hráčom a NPC. Každej postave nastavíme unikátnu množinu dátových tabuľiek, ktorá definuje množinu replík, na ktoré sa môžeme pýtať danej postavy. Parametre Reply Function Class a Reply Helper Function Class sú nastavenia Player komponentu. Definujú, ktorá implementácia funkčnej triedy bude použitá pre daný komponent. Podľa toho bude algoritmus vypočítavať výstupné, alebo pomocné dátá pre hráča. V komponente sú prednastavené predvolené triedy, implementované nami. Ak by hráč chcel implementovať vlastný algoritmus (napr. neurónovú sieť), stačí zmeniť toto nastavenie, aby sa použil jeho vlastný algoritmus.

4.4 Vytvorenie a editácia dátových tabuľiek

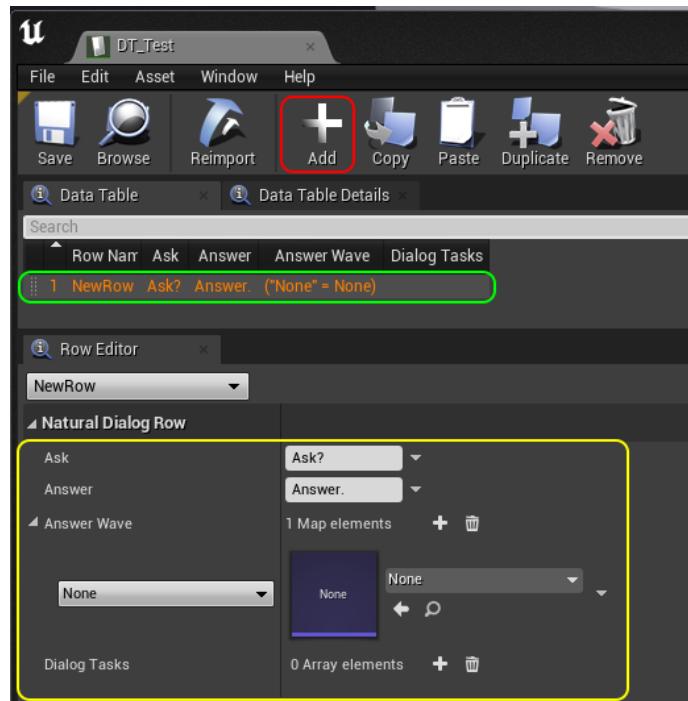
Dátové tabuľky, ktoré použijeme v projekte pre dialógové systémy vytvárame iba v prostredí Unreal Editor. Každá vytvorená dátová tabuľka sa správa ako inštancia triedy *Data Table*. Pri vytváraní definujeme štruktúru riadku, ktorá bude použitá na reprezentáciu tabuľky. Pre plugin máme prichystané štruktúry, ktoré sú zobrazené na obrázku 4.17 a sú tiež popísané v časti 3.4 a 3.5.



Obr. 4.17: Príklad vytvárania dátovej tabuľky

Štruktúra `NaturalDialogRow` tvorí základ nášho systému a jej zadená štruktúra `NaturalDialogRow_Keyword` je zavedená pre prácu s našimi implementovanými algoritmami. Po jej použití vznikne asset, ktorý pomenujeme a uložíme. Po jeho otvorení môžeme začať s jeho editáciou. Na obrázku 4.18 je príklad dátovej tabuľky, ktorú budeme editovať. Tlačidlom, označeným červenou farbou, pridávame nové riadky (nové repliky) do dátovej tabuľky. V zelenom rámčeku vidíme reprezentáciu replík nachádzajúcich sa v tabuľke. Po zvolení repliky, môžeme začať s editáciou. Editácia prebieha pomocou prvkov, označených v žltom rám-

čeku. Po editácii tabuľky, tabuľku uložíme a priradíme do komponentu, kde ju chceme použiť'.



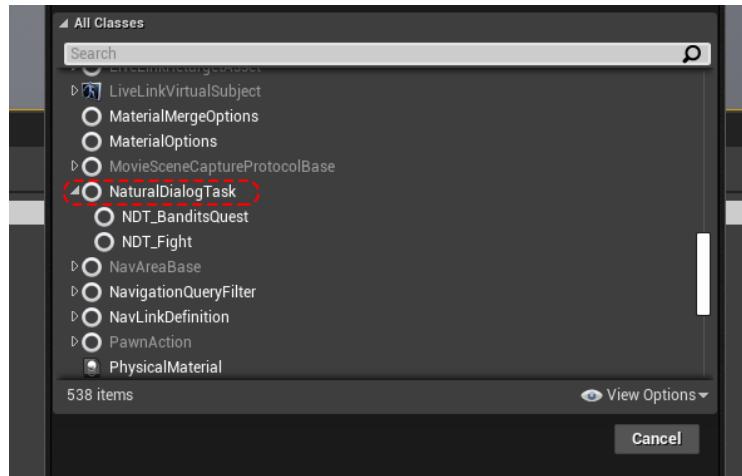
Obr. 4.18: Editácia dátovej tabuľky

Štruktúru `NaturalDialogRow_Base` používame na definovanie unikátnych tabuľiek, v ktorých sa nenachádza položka `Ask`. Je navrhnutá pre zoznam replík pre hráča, kedy systém nerozpozná otázku pre NPC. Ak teda položíme otázku, ktorej systém neporozumie, vyberie sa replika z tejto dátovej tabuľky, ktorú použijeme ako odpoved'.

4.5 Vytvorenie a editácia dialógových úloh

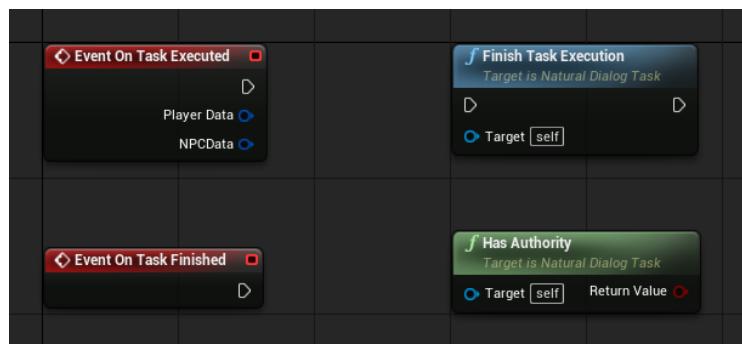
V dialógových tabuľkách môžeme replikám nastaviť doplnkovú funkciunalitu, ktorá sa zavolá v prípade, že sa replika v dialógu použije. Dialógové úlohy môžeme priradiť do pol'a premennej `Dialog Tasks` zobrazenej

na obrázku 4.18 v žltom rámčeku.



Obr. 4.19: Vytvorenie dialógovej úlohy

Dialógovú úlohu môžeme vytvoriť pomocou C++ kódu a pomocou Blueprint kódu. Doporučujeme vytvárať dialógové úlohy pomocou Blueprint kódu, pretože sú predovšetkým záujmom herných dizajnérov a grafikov, ktorí nepracujú v C++. Vytvorením dialógovej úlohy, ako je na obrázku 4.19, vytvoríme Blueprint asset, ktorý uložíme a priradíme do premennej v dialógovej tabuľke.



Obr. 4.20: Editácia dialógovej úlohy

Na obrázku 4.20 vidíme vytvorenú dialógovú úlohu, ktorej priradíme funkcionality. Event `OnTaskExecuted` je zavolaný po vytvorení dialógovej

úlohy a našu funkcionality začneme napájať na tento event. Po ukončení úlohy je potrebné zavolať funkciu `FinishTaskExecution`, ktorá povie systému, že úloha bola ukončená čím vykonáme event `OnTaskFinished`, kde definujeme funkcionality pre ukončenie úlohy, ak to potrebujeme. Po jej vykonaní sa úloha uzatvorí a považuje sa za ukončenú. Funkcia `HasAuthority` poskytuje vývojárovi boolean informáciu, či sa herný kód vykonáva na strane servera.

Príklad použitia:

1. Hráč sa spýta NPC postavy, či má misiu, ktorú by mohol splniť.
2. NPC mu odpovie, že jednu má a povie znenie misie.
3. Ked' mu povie detailly úlohy, aktivuje sa dialógová úloha a vykoná sa táto postupnosť:
 - (a) Event *On Task Executed* vytvorí widget, ktorý bude obsahovať dve tlačidla (*Áno/Nie*), a otázku či hráč chce prijať misiu.
 - (b) Čakáme na odpoved' hráča či príma misiu, alebo nie a vykoná sa:
 - Ak hráč odpovie *Áno*, systém aktivuje misiu, ktorú má hráč splniť a môže registrovať nové dátové tabuľky pre komunikáciu s NPC aby sa mohol spýtať na nové informácie o misii, na ktoré sa do teraz nemusel vedieť spýtať.
 - Ak hráč odpovie *Nie*, systém uzatvorí dialógovú úlohu bez žiadnej doplnkovej funkcionality.
 - (c) Po vykonaní všetkých krokov na spustenie misie, zavoláme *Finish Task Execution* čím povieme systému, že sme vykonali

všetko čo sme chceli.

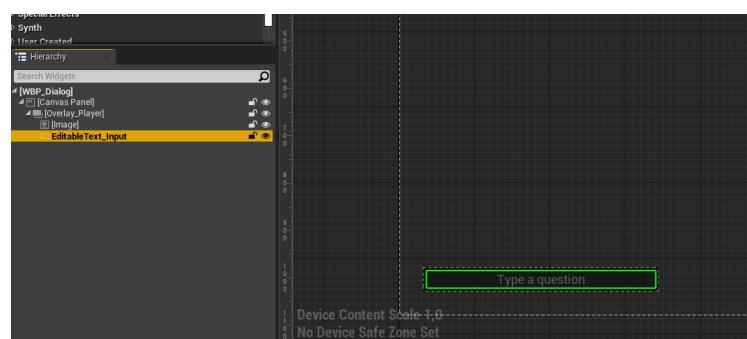
- (d) Pre event *On Task Finished* môžeme definovať funkciu, ktorou bud' podakujeme hráčovi za prijatie misie, alebo jej odmietnutie.
4. Dialógová úloha sa uzatvorí a hráč sa môže pýtať d'alej NPC na jej detaily.

4.6 Použitie systému

Po vykonaní všetkých predchádzajúci bodov z časti 4 máme systém pripravený na použitie. V tejto časti popíšeme postup použitia systému v hernom projekte. Postup bude prezentovaný pre C++ aj Blueprint kód.

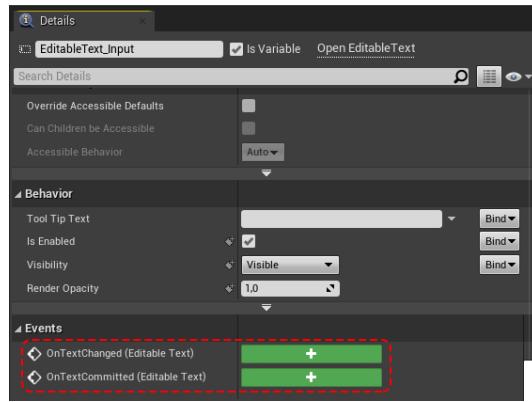
Postup použitia:

1. Vytvoríme si triedu zdedenú z triedy `UserWidget`, v ktorej pridáme do widget hierarchie prvky podporujúce vstup užívateľa prostredníctvom klávesnice. Na obrázku 4.21 vidíme príklad takéhoto widgetu



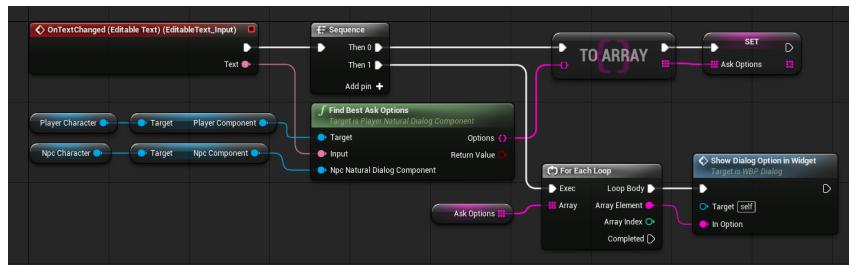
Obr. 4.21: Príklad widgetu so vstupnými prvkami

2. Pre vstupné prvky naviažeme eventy, ktoré sa zavolajú pri aktualizácii textu, alebo jeho komitnutí. Na obrázku 4.22 vidíme príklad takýchto eventov, ktoré sa používajú zo vstupných prvkov definovaných v hierarchii na obrázku 4.21



Obr. 4.22: Príklad eventov pre vstupné prvky z obrázku 4.21

3. Event **On Text Changed**, ktorý je zavolaný pri l'ubovoľnej zmene textu, môžeme definovať funkcionalitu na vypisovanie pomocných otázok, na ktoré sa budeme pýtať. Na obrázkoch 4.23 a 4.24 viďime príklad implementácie takejto funkcionality. Výsledok takejto implementácie môžeme vidieť v prílohe A.2



Obr. 4.23: Príklad implementácie nájdenia dostupných otázok v Blueprinti

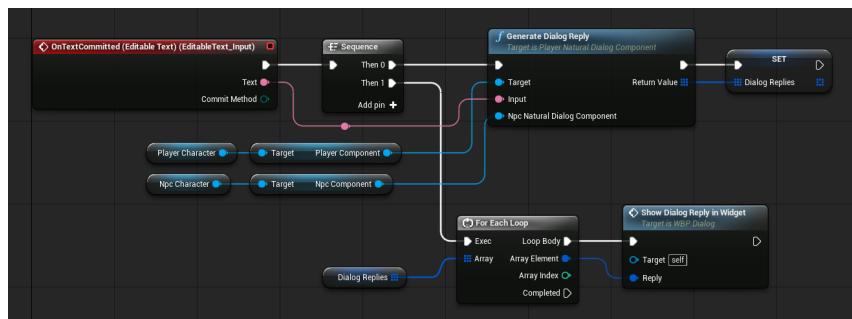
4. Event **On Text Committed**, ktorý je zavolaný pri potvrdení textu, môžeme definovať funkcionalitu na generovanie odpovede vstup-

```

20 |H| void APlayerCharacter::FindAskOptions(const FText& PlayerInput, const UNpcNaturalDialogComponent* NpcComponent)
21 |
22     TSet< FString> Result;
23     PlayerComponent->FindBestAskOptions(PlayerInput, NpcComponent, [&]Result);
24
25     for (const FString& Option : Result)
26     {
27         ShowDialogOptionInWidget(Option);
28     }
29 }
```

Obr. 4.24: Príklad implementácie nájdenia dostupných otázok v C++

ného textu. Na obrázkoch 4.25 a 4.26 vidíme príklad implementácie takejto funkcionality



Obr. 4.25: Príklad implementácie generovania odpovedí v Blueprintie

```

void APlayerCharacter::GenerateReply(const FText& PlayerInput, const UNpcNaturalDialogComponent* NpcComponent)
{
    const TArray< FNaturalDialogAnswer> Result = PlayerComponent->GenerateDialogReply(PlayerInput, NpcComponent);

    for (const FNaturalDialogAnswer& Reply : Result)
    {
        ShowDialogReplyInWidget(Reply);
    }
}
```

Obr. 4.26: Príklad implementácie generovania odpovedí v C++

5. Výsledky práce

V tejto časti sa budeme venovať vyhodnoteniu navrhnutého systému, ktorý sme implementovali v hernom projekte. Pre testovanie systému sme vytvorili projekt v prostredí UE4, ktorý je navrhnutý na odskúšanie funkcionality a ciel'ov tejto práce, pričom výsledky vyhodnotíme samostatne z pohľadu vývojára a užívateľ'ov, ktorí tento systém testovali.

V projekte sme vytvorili level, kde kompozícia sveta, dialógov a príbehu, predstavuje stredovekú dedinku napadnutú banditami, pričom dialógový systém vedie hráča jednoduchým príbehom. Z dôvodu veľkého objemu dát, ktoré táto kompozícia obsahuje, pridáme do elektronickej prílohy iba projekt s prázdnou mapou, NPC postavami a príbehom.

Vo výsledkoch sa zameriame na dve hlavné časti práce a to je návrh systému v prostredí Unreal Engine a vyhodnotenie implementácie algoritmov založených na vyhľadávaní kl'účových slov. Testovanie prebiehalo za pomoci siedmich testovacích subjektov, ktorí dostali za úlohu prejst' príbeh hry. Stĺpcové grafy v tejto časti predstavujú hodnoty 1 (najhoršie) až 5 (najlepšie).

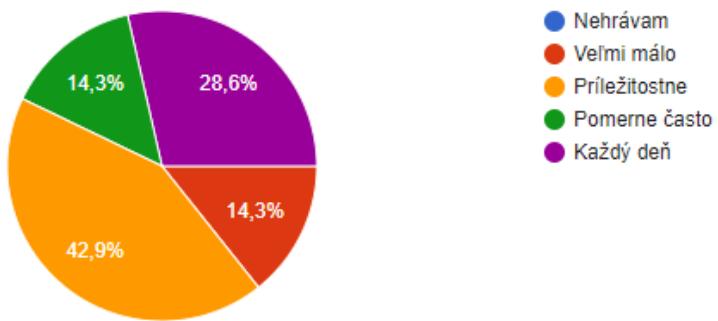
5.1 Portabilita a modularita

Systém sa nám podarilo navrhnúť ako samostatný plugin, čím sme dosiahli vysokú portabilitu. Jeho integrácia do prostredia UE je nenáročná a možná pre rôzne herné projekty a žánre. Vďaka tomu sme dosiahli, že systém je možné využiť pre širokú škálu hier a programov založených v tomto prostredí. Okrem toho sa nám podarilo samostatné algoritmy navrhnúť oddelene od jadra systému, čím sme dosiahli vysokú modularitu. Práve z tohto dôvodu dokážeme implementovať nové nezávislé algoritmy pracujúce s rovnakými, alebo podobnými dátovými štruktúrami.

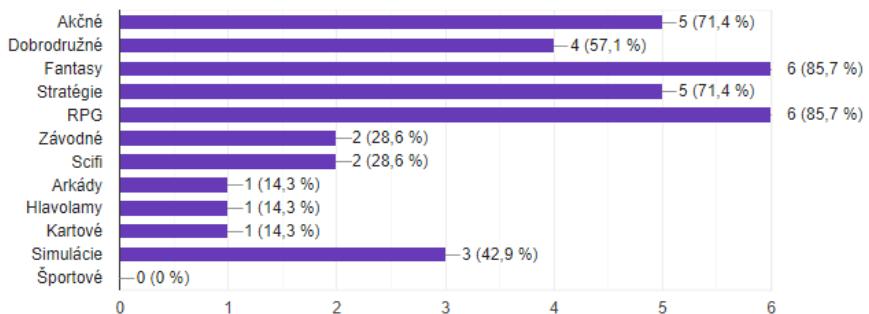
5.2 Testovanie

Po implementácii systému do UE projektu sme vytvorili množinu dát D , ktoré tvoria hlavnú dejovú os. Túto množinu tvorilo približne sto dialógových replík. Aby množina D obsahla čo najväčšie množstvo replík, rozhodli sme sa testovať systém v dvoch etapách. Do prvej etapy sme začlenili skupinku ľudí, ktorá systém odskúšala a na základe dialógov s NPC postavami, sme sa rozhodli doplniť množinu D o nové repliky, ktoré mali konverzácie s nimi rozšíriť. V druhej etape sme uskutočnili finálne testovanie, pomocou ktorého sme vypracovali a vyhodnotili výsledky práce pomocou online formulára. Celý formulár nájdeme v elektronickej prílohe tejto práce.

Na grafoch 5.1 a 5.2 môžeme vidieť, že sme sa snažili do testovania zahrnúť čo najväčšie spektrum ľudí podľa času, ktorý strávia pri počítačových hrách a hraných herných žánrov, aby sme zistili reakciu ľudí na systém, ktorí majú väčšie, alebo menšie skúsenosti s hernými aplikáciami.



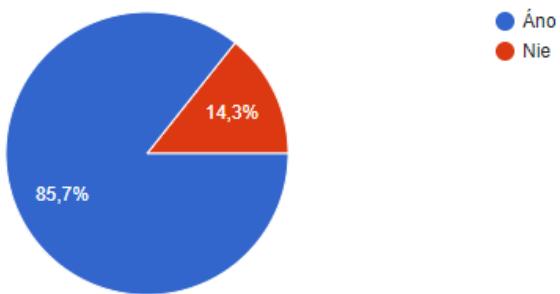
Obr. 5.1: Výsledky otázky: *Ako často hrávaš hry?*



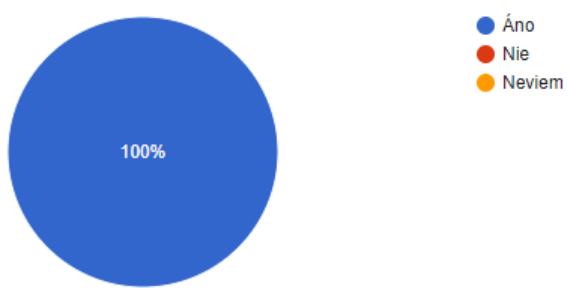
Obr. 5.2: Výsledky otázky: *Aké herné tituly preferuješ?*

Na grafe 5.3 vidíme, že do testovania sme sa snažili zahrnúť aj ľudí, ktorí nehrávajú hry založené na dialógových mechanikách. Ked'že veľa hier má integrované dialógy, väčšina z nich sa už stretla s niektorými variáciami od najjednoduchších, ktoré iba sprevádzajú hráča dejom až po komplexné ako sú dialógové stromy. Vďaka tomu môžeme vyhodnotiť aký náročný bol systém na používanie v závislosti od rozmanitosti testovacích subjektov.

Na grafoch 5.4, 5.5 a 5.6 vidíme štatistiku, ako ľudí zaujal typ dialógového systému. Ked'že takmer všetky doteraz existujúce hry obsahujú dialógové mechaniky, nie je prekvapujúce, že takáto mechanika je pre hráčov niečo nové vďaka čomu je to veľmi unikátne v hernej oblasti.



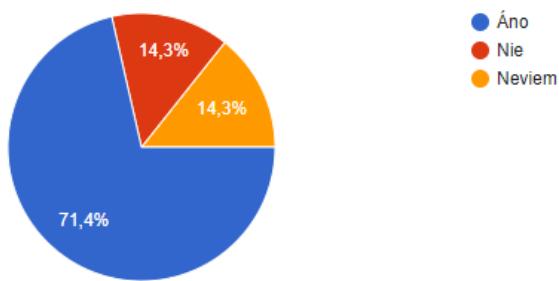
Obr. 5.3: Výsledky otázky: *Hráš hry založené na dialógových systémoch?*



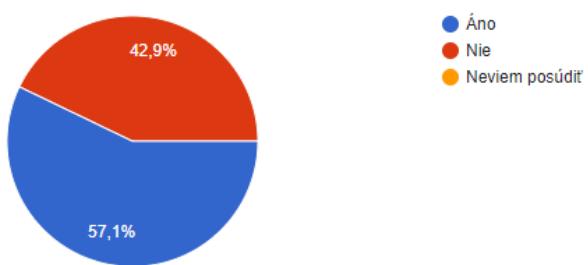
Obr. 5.4: Výsledky otázky: *Zaujal t'a typ odskúšaného dialógového systému?*

Ako vidíme na grafe 5.4, všetkých opýtaných zaujalo spracovanie systému, avšak z dôvodu chybovosti, ktorú tento systém obsahuje oproti starým mechanikám, ktoré sú deterministické, existujú pochybnosti, či má budúcnosť v hernom priemysle.

Grafy 5.5 a 5.6 ukazujú, že aj ľudí, ktorých zaujala mechanika, nevedia či by ju uvítali v blízkej budúcnosti. Ked'že je herný dej veľa krát závislý na dialógoch s NPC postavami, môže sa veľmi ľahko stať, že sa hráč dostane do bodu, kedy nievie ako pokračovať ďalej. Ak by dialógové dátá neboli dobre navrhnuté, herný zážitok môže byť minimálny. Avšak všetci opýtaní si myslia, že je to otázka blízkej budúcnosti kým nedosiahneme bezchybnosť tejto mechaniky.

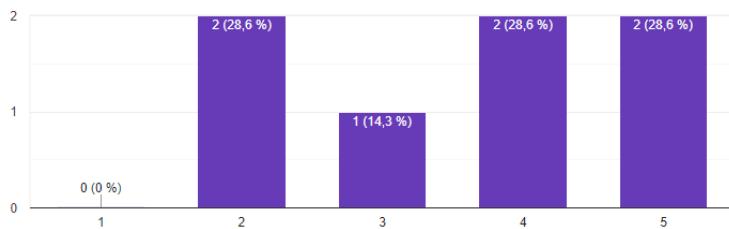


Obr. 5.5: Výsledky otázky: *Preferoval by si v budúcnosti viac hier s odskúšaným dialógovým systémom?*

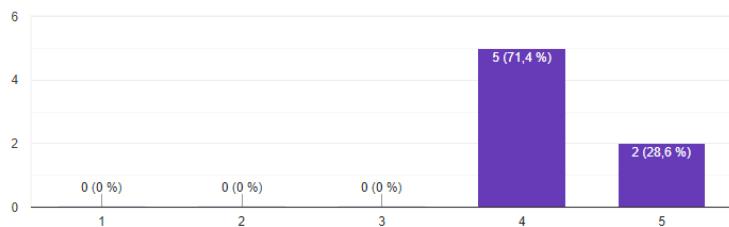


Obr. 5.6: Výsledky otázky: *Myslíš, že podobný systém je budúcnosť v hernom priemysle?*

Na obrázku 5.7 a 5.8 vidíme posúdenie užívateľov, ako náročný je systém na používanie a obsluhu. Jeho náročnosť je hlavne závislá od skúseností s hernými žánrami a komplexnosti hraných hier samotnými hráčmi. Pre užívateľov, ktorí preferujú jednoduché herné žánre ako sú napr. kartové hry a nevyhľadávajú komplexnosť, je systém náročnejší na použitie, ale pre pravidelných hráčov je používanie systému nenáročné. V testovacom scenári viacerým užívateľom chýbala možnosť automatického dokončovania viet, čo kazilo intuitívnosť systému pri písaní vstupov.



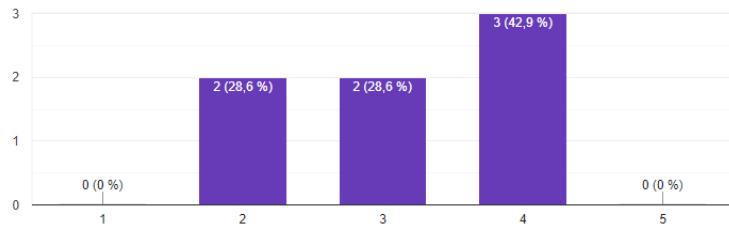
Obr. 5.7: Výsledky otázky: *Ako náročný bol systém na používanie?*



Obr. 5.8: Výsledky otázky: *Ako intuitívny bol pre teba systém?*

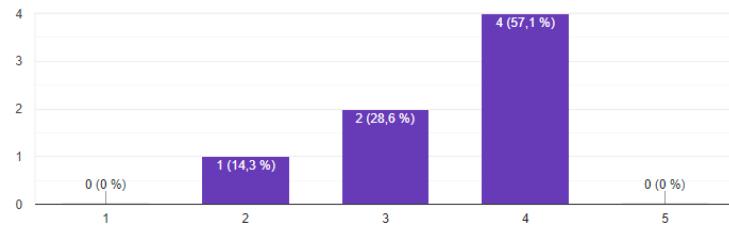
5.3 Chybovost'

Úspešnosť systému reagovať na vstupy od užívateľa závisí od použitého algoritmu a od spektra dát. Z vyhodnotenia testovania sme zistili, že algoritmus *Tf-Idf* nie je veľmi vhodný pre malé množiny dátových assetov. Z toho dôvodu pri testovaní systému na testovacej mape, kde sme použili malú množinu dát oproti plnohodnotnej hre, algoritmus nepracoval podľa očakávaní. Veľmi často sa stávalo, že vybrané kľúčové slová neboli dôležité a mohli patriť do pomyselného stoplistu. Z toho dôvodu algoritmus niekedy priradoval otázku k nesprávnej odpovedi. Kvôli nedostatočnému spektru replík bol systém tiež obmedzený na používanie výhradne na dejovú linku a nevedel odpovedať na otázky mimo herného dej, ale v prípade plnohodnotnej hry, by tento problém mohol byť odstránený rozšírením dialógových replík.

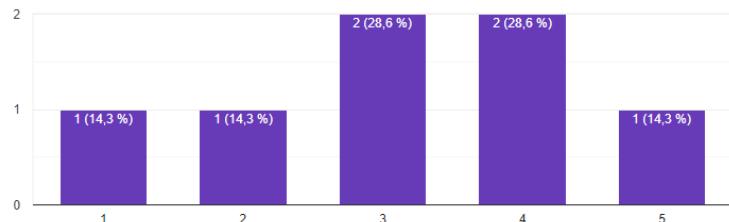


Obr. 5.9: Výsledky otázky: *Ako hodnotíš odskúšaný dialógový systém?*

Na grafe 5.9 vidíme celkové hodnotenie systému. Je veľmi závislé od presnosti odpovedí (ako vidíme aj na grafe 5.10) a schopnosti odpovedať na rôznu škálu odpovedí (ako vidíme aj na grafe 5.11).



Obr. 5.10: Výsledky otázky: *Ako hodnotíš konverzáciu s postavami v hre?*

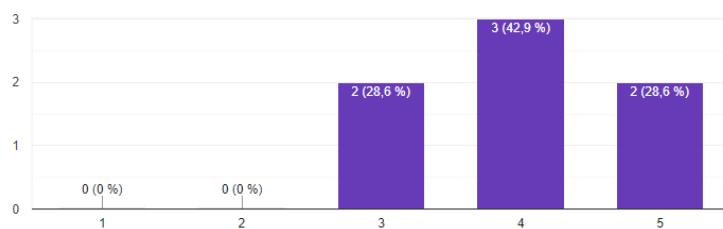


Obr. 5.11: Výsledky otázky: *Ako často postava nevedela odpovedať na tvoju otázku?*

Veľkým problémom algoritmu pre vyhľadávanie odpovedí nastával pri rozpoznávaní kontextu. V prípade, že hráč pristupoval k systému metódou konverzácie (podobne ako konverzácia človek s človekom), mal systém väčšie problémy. Čím viac sa subjekt snažil konverzovať klasickým

spôsobom, tak chybovost' systému sa výrazne zvýšila (hodnotenia 2 a 3 na grafe 5.10). Túto chybovost' by mohlo zmenšiť rozšírenie replík v dátových tabuľkách, ale vďaka tomu vzniká náročnosť na prekladanie textov a správu dátových tabuliek pre samotné NPC.

V prípade, že hráč pristupoval k systému formou dotazovania a otázky boli smerované na herný príbeh, vedel systém poskytnúť správne odpovede s malou chybovost'ou (hodnotenie 4 na grafe 5.10). Schopnosť systému spracovať vstupy a generovať výstupy bola celkom správna, avšak mala tiež miernu chybovost'.



Obr. 5.12: Výsledky otázky: *Ako náročné bolo postupovať príbehom?*

Ako sme popísali vyššie, problémy s konverzáciami viedli, že každý testovací subjekt mal inú mieru náročnosti postupovania príbehom. Aj keď bol príbeh jednoduchý, mechanika spôsobovala mierne problémy s pokrokom v hernom deji a každý testovací subjekt sa zasekol v nejakom bode. Tento problém by mohol byť tiež odstránení samotným dizajnom hry, ktorý by užívateľa navádzal ako pokračovať (UI a 3D indikátory v hre).

Záver

Podarilo sa nám splniť všetky ciele, ktoré sme si stanovili v tejto práci. Výsledkom je návrh nového experimentálneho dialógového systému s vysokou portabilitou a modulárnosťou. Systém je možné ľahko rozšíriť a doplniť o nové algoritmy, pričom nie je potrebné zasahovať do jadra programu.

Súčasťou riešenia je implementácia a odskúšanie predvolených algoritmov založených na analýze klíčových slov, pomocou ktorých sa systém snaží odpovedať na herné vstupy od užívateľa. Systém s implementovanými algoritmami sa nám podarilo úspešne integrovať do herného projektu, kde kompozícia sveta a dialógových replík vytvárajú jednoduchú dejovú linku, ktorú sme použili na testovanie systému.

Po otestovaní sme zistili, že implementované algoritmy nie sú vhodné pre malé dátové množiny a preto výsledky testovania neboli veľmi pozitívne. Systém vykazoval škálu problémov spojených s analyzovaním herného vstupu a vybraním klíčových slov pre výhodnotenie výstupu, čím vznikala mierna náhodnosť pri výbere odpovede. Dátové množiny boli tiež veľmi obmedzené na dejovú linku a preto testovacie subjekty nedokázali viest' plnohodnotnejší dialóg s postavami. Vďaka tomu mali mierne problémy s postupovaním v príbehu a narúšaním zážitku z hry. Výhodou implementovaných algoritmov, vďaka ich jednoduchosti, bola nízka ná-

ročnosť na výpočtový výkon k pomeru správnosti vyhodnotenia odpovedí.

Pre zníženie chybovosti môžeme rozšíriť herný dej pomocou nových herných replík a dejových liniek, čo je ale veľmi náročné na čas a nie je možné to dosiahnuť z krátkodobého hľadiska. Pre lepšie výsledky môžeme v budúcnosti tiež implementovať zložitejšie algoritmy, založené na novších metodikách spracovania prirodzeného jazyka, alebo využitia neurónových sietí. Veľkou výhodou v budúcnosti by bolo implementovať množinu algoritmov, z ktorých by sme si mohli vybrať v závislosti od veľkosti a komplexity herného príbehu.

Z dôvodu chybovosti systému je pravdepodobné, že nie je pripravený na plnohodnotné využitie v hernom priemysle v blízkej budúcnosti a je nutné implementovať sériu úprav a novších algoritmov, ktoré budú využívať vstupné a výstupné dátá. Avšak aj s chybovosťou ktorú tento systém obsahoval, všetkým testovacím subjektom sa odsúšaný systém veľmi páčil z dôvodu jeho unikátnosti v dnešnej dobe a v prípade odstránenia vyššie uvedených závad by takýto spôsob dialógovej mechaniky uvítali v budúcnosti ako súčasť herného priemyslu.

A. Interakcia s NPC



Obr. A.1: Príklad interakcie s NPC



Obr. A.2: Príklad vstupu pre dialóg s NPC

Literatúra

- [1] Not Yet. Not yet: Dialogue plugin system. <https://www.unrealengine.com/marketplace/en-US/product/not-yet-dialogue-system>, Mar 2, 2018. [Online; accessed 2-Marec-2020].
- [2] CodeSpartan. Codespartan: Dialogue system. <https://www.unrealengine.com/marketplace/en-US/product/dialogue-plugin>, May 27, 2016. [Online; accessed 27-May-2020].
- [3] Zork. <https://en.wikipedia.org/wiki/Zork>. [Online; accessed 27-May-2020].
- [4] Jonathan Lessard. Designing natural-language game conversations. https://www.lablablab.net/papers/lablablab_CR.pdf, 2016. [Online; accessed 2020].
- [5] Elizabeth D Liddy. Natural language processing. 2001. [Online; accessed 2020].
- [6] Jan Wijffels. Textrank for summarizing text. <https://web.eecs.umich.edu/~mihalcea/papers/mihalcea.emnlp04.pdf>, Dec 12, 2020. [Online; accessed 12-December-2020].

- [7] Purva Huilgol. Quick introduction to bag-of-words (bow) and tf-idf for creating features from text. <https://www.analyticsvidhya.com/blog/2020/02/quick-introduction-bag-of-words-bow-tf-idf/>, Feb 28, 2020. [Online; accessed 28-February-2020].
- [8] Stuart Rose, Dave Engel, Nick Cramer, and Wendy Cowley. Automatic keyword extraction from individual documents. *Text mining: applications and theory*, 1:1-20, 2010.
- [9] Ian Rogers. The google pagerank algorithm and how it works. 2002. [Online; accessed 2020].
- [10] Diego Lopez Yse. Text normalization for natural language processing (nlp). <https://towardsdatascience.com/text-normalization-for-natural-language-processing-nlp-70a314bfa646>, Feb 17, 2021. [Online; accessed 17-February-2021].
- [11] Sergey Grashchenko. Levenshtein distance computation. <https://www.baeldung.com/cs/levenshtein-distance-computation>, Nov 5, 2021. [Online; accessed 5-November-2021].
- [12] Epic Games. Epic games. <https://www.epicgames.com/site/en-US/home>. [Online; accessed 27-October-2019].
- [13] Epic Games. Jet brains. <https://www.jetbrains.com/>. [Online; accessed 27-October-2019].
- [14] Digital Ocean. Digital ocean. <https://www.digitalocean.com/>. [Online; accessed 27-October-2019].

- [15] Perforce. Helix. <https://www.perforce.com>. [Online; accessed 27-October-2019].
- [16] Bhavika R Ranoliya, Nidhi Raghuvanshi, and Sanjay Singh. Chatbot for university related faqs. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1525–1530. IEEE, 2017.
- [17] Menal Dahiya. A tool of conversation: Chatbot. *International Journal of Computer Sciences and Engineering*, 5(5):158–161, 2017.
- [18] Bayan Abu Shawar and Eric Atwell. Different measurement metrics to evaluate a chatbot system. In *Proceedings of the workshop on bridging the gap: Academic and industrial research in dialog technologies*, pages 89–96, 2007.
- [19] JuliaStrings. utf8proc. <https://github.com/JuliaStrings/utf8proc>, Dec 17, 2021. [Online; accessed 17-December-2021].
- [20] Massimo Lusetti, Tatyana Ruzsics, Anne Göhring, Tanja Samardžić, and Elisabeth Stark. Encoder-decoder methods for text normalization. Association for Computational Linguistics, 2018.