POLITECHNIKA WARSZAWSKA

Wydział Elektroniki i Technik Informacyjnych

Instytut Telekomunikacji

PRACA DYPLOMOWA MAGISTERSKA

Dariusz Wawer

SYSTEM ROZPOZNAWANIA MOWY JAKO INTERFEJS GRY KOMPUTEROWEJ

Opiekun:
dr inż. Artur Janicki

. . . .
ocena pracy

. . . .
data i podpis Przewodniczącego
Komisji Egzaminacyjnej

Warszawa, grudzień 2011

Streszczenie

System rozpoznawania mowy jako interfejs gry komputerowej.

Praca ta opisuje proces projektowania, implementacji oraz wyniki systemu automatycznego rozpoznawania mowy ciągłej typu *command and control* dla języka polskiego, który ma służyć jako interfejs gry komputerowej *Pilot rajdowy*. System wykorzystuje framework *Sphinx*, którego działanie oparte jest na ukrytych modela Markowa.

Przedstawione i opisane są także podstawowe elementy systemu rozpoznawania mowy i objaśnione są interakcje między nimi. Omówiony jest także sposób działania takiego systemu. Rozważana jest problematyka zastosowania rozpoznawania mowy w grze komputerowej. Zaprezentowane są problemy, z którymi zetknąłem się w trakcie pracy.

Wyniki zaimplementowanego systemu są szczegółowo omówione, zbadany i przedstawiony jest wpływ jakości poszczególnych elementów systemu na jego ostateczną jakość. Opracowany system, z modelem akustycznym dedykowanym dla jednego mówcy, osiąga dokładność zdaniową rzędu 95%. Ten sam model akustyczny, zaadaptowany głosem innego mówcy, daje dla niego dokładność ok 90%. Wykorzystując model akustyczny trenowany 45. zestawami zdań z bazy corpora, dla dwóch testowych mówców uzyskuje dokładność zdaniową powyżej 85%.

Abstract

Automatic speech recognition system as a computer game interface.

This thesis describes the process of designing and implemening and the results of a *command and control* Automatic Speech Recognition system for the Polish language, which is used as an interface for a computer game *Rally Navigator*. This system uses *Sphinx* framework, which is based on Hidden Markov Models.

The basics of speech recognition are characterised, as well as elements of a speech recognition system and relations between them. Operation of such system is also described. The usage of speech recognition system in a computer game is discussed. Problems encountered during implementing the system and their solutions are presented.

The results of the ASR system are shown, the influences of individual elements of the system are explored and presented. The resulting system, with a single speaker dedicated acoustic model, achieved sentence accuracy of approximately 95%. Using the same acoustic model, adapted for a different speaker, resulted in 90% sentence accuracy. With an acoustic model trained on 45 corpora sentences sets, for two different speakers, the system reached sentence accuracy greater than 85%.

Życiorys

Dariusz Wawer
Kierunek: Telekomunikacja
Specjalność: Systemy i sieci telekomunikacyjne
Urodzony: 1 stycznia 1986 r. w Warszawie
Numer indeksu: 200832

W 2005 roku ukończyłem XIV L.O. im. Stanisława Staszica w Warszawie. Tego samego roku rozpocząłem studia na wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej na kierunku Telekomunikacja. Od marca 2008 roku pracuję jako programista w firmie CC Otwarte Systemy Komputerowe. Byłem członkiem drużyny, która zwyciężyła Międzynarodowy Turniej Młodych Fizyków w Brisbane w Australii w 2004 roku. Jestem członkiem Koła Naukowego Twórców Gier na wydziale Elektroniki i Technik Informacyjnuch Politechniki Warszawskiej. Główne obszary moich zainteresowań to programowanie symulacji fizycznych, gier oraz aplikacji internetowych, a także rozpoznawanie mowy.

....

Spis treści

Spis treści **9**

Spis rysunków **12**

Spis tabel **13**

Spis listingów **14**

1 Wstęp **15**

1.1 Motywacja 15
1.2 Cel pracy 16
1.3 Wybór gry 16
1.4 Układ pracy 17

2 Podstawy teoretyczne systemów rozpoznawania mowy **19**

2.1 Klasyfikacja systemów 19
2.2 Czynniki wpływające na skuteczność 20
2.3 Zależności między elementami systemu ASR 20
2.4 Model akustyczny 20
2.4.1 Przygotowanie modelu 21
2.4.2 Przygotowanie danych treningowych 21
2.5 Ukryty Model Markowa 22
2.6 Słownik 25
2.7 Model języka 26
2.7.1 Model n-gram 26
2.7.2 Negatywne n-gramy 28
2.7.3 Gramatyka JSGF 28
2.8 Kraty słów 29
2.9 Sieci niepewności 30
2.10 Istniejące rozwiązania 30

2.11 Wybór technologii 31

3 Framework *CMU Sphinx* **33**

3.1 Elementy systemu *Sphinx 4* 33

3.2 Korzystanie z systemu *Sphinx 4* 35

3.3 Wprowadzone modyfikacje 36

4 Specyfika gier komputerowych w kontekście systemów ASR **39**

4.1 Możliwości wykorzystania rozpoznawania mowy w grach 39

4.2 Wymagania dla systemu 40

4.3 Czynniki wpływające na system ASR 40

4.4 Analiza możliwości wykorzystania systemów ASR w wybranych gatunkach gier 41

4.4.1 Gry akcji *FPP/TPP* 41

4.4.2 Gry *RPG* 43

4.4.3 Gry strategiczne 44

4.4.4 Inne typy gier 44

5 Rozwiązania wykorzystane w grze *Pilot rajdowy* **45**

5.1 Fizyka 45

5.2 Integracja modułu rozpoznawania mowy z grą 46

5.3 Przetwarzanie rozpoznanego tekstu 46

5.4 Doszkalanie modelu akustycznego głosem gracza 49

6 Projektowanie i ewaluacja systemu ASR **51**

6.1 Opracowanie zestawu komend 51

6.2 Testy systemu 52

6.2.1 Stosowane metryki 53

6.3 Model akustyczny 54

6.3.1 Model angielski 54

6.3.2 Model polski 55

6.3.3 Adaptacja 56

6.4 Modele języka 58

6.4.1 N-gram 58

6.4.2 JSGF 59

6.5 Prototyp gry 60

7 Podsumowanie **63**

7.1 Jakość systemu 63

7.2 Możliwości komercyjnego wykorzystania 64

7.3 Dalszy rozwój 64

Bibliografia **65**

Wykaz skrótów **69**

A Lista komend **70**

B Zdania testowe **71**

C Zawartość płyty **75**

Spis rysunków

2.1 Przykład prostego modelu Markowa 23
2.2 Macierze przejścia i obserwacji 24
2.3 Wartości macierzy przejść i obserwacji 24
2.4 Przykładowa krata słów 30

3.1 Schemat systemu ASR *Sphinx 4* [25] 35

5.1 Schemat integracji systemu rozpoznawania mowy z grą *Pilot rajdowy* . . . 47

6.1 Zależność słownej stopy błędów od długości danych treningowych 55
6.2 Wpływ adaptacji na jakość systemu 57
6.3 Zależność dokładności i dokładności zdaniowej od zastosowanego modelu n-gram 59
6.4 Widok okna prototypu gry *Pilot rajdowy* 61

Spis tabel

6.1 Dane zwracane przez testy systemu ASR w *Sphinksie* 53
6.2 Wyniki modeli akustycznych w zależności od długości danych treningowych 56
6.3 Wyniki modeli języka: n-gram i JSGF 56

Spis listingów

2.1 Fragment słownika z gry *Pilot rajdowy* 25
2.2 Fragment modelu n-gram z gry *Pilot rajdowy* 27
2.3 Fragment definicji gramtyki JSGF z gry *Pilot rajdowy* 29
2.4 Określanie prawdopodobieństw wyrażeń alternatywnych w gramatyce JSGF 29
3.1 Przykład elementu konfiguracji frameworku *Sphinx 4* 36
6.1 Przykład wyniku wykonania testu systemu 52
6.2 Wyniki pierwszego testu systemu 54

Rozdział 1

Wstęp

1.1 Motywacja

Systemy rozpoznawania mowy tworzone i rozwijane są od wielu lat. Wraz z rozwojem technologii i wzrostem mocy obliczeniowej komputerów osobistych stały się one dostępne dla przeciętnego użytkownika posiadającego zwykły komputer PC.

Systemy ASR (ang. '*Automatic Speech Recognition*', systemy *automatycznego rozpoznawania mowy*) znajdują zastosowanie w wielu dziedzinach. Mogą obsługiwać telefoniczny system bankowości, ułatwiać korzystanie z komputera osobom niepełnosprawnym (w szczególności w przypadku niepełnosprawności wzrokowej), przyśpieszać wybieranie numeru w telefonie lub po prostu służyć jako system do dyktowania. Liczba zastosowań systemów rozpoznawania mowy wciąż wzrasta ze względu na rozpowszechnienie i popularyzację tej technologii. Nie bez znaczenia jest także fakt, iż systemy te działają coraz lepiej.

Koncepcja wykorzystania systemu ASR w grze komputerowej powstała z połączenia kilku dziedzin zainteresowań: lingwistyki, programowania oraz gier komputerowych, a także, w mniejszym stopniu, przetwarzania sygnałów. Programowanie i gry komputerowe są zainteresowaniami w pewnym sensie połączonymi, choć należy zaznaczyć, że programowanie gier znacząco różni się od wykonywania typowych projektów algorytmicznych, czy nawet biznesowych.

Istnieją co najmniej dwie komercyjne gry komputerowe, które wykorzystują rozpoznawanie mowy jako element interfejsu. Technologia ta jest w nich jednak jedynie dodatkiem i, w praktyce, nie jest efektywnie wykorzystywana. Problematyka zastosowania systemów ASR w grach zostanie szerzej omówiona w dalszej części pracy.

Rozpoznawanie mowy ciągłej jest procesem daleko bardziej skomplikowanym, niż rozpoznawanie pojedynczych słów. Tylko kilka tego typu systemów dla języka polskiego jest

rozwijanych [1][2]. Utworzony przeze mnie system, lub jego elementy, będą mogły być w przyszłości zastosowane do innych celów, niż tylko interfejs gry komputerowej.

1.2 Cel pracy

Celem pracy jest zaprojektowanie oraz zaimplementowanie systemu rozpoznawania mowy do wykorzystania w grze komputerowej. System powinien umożliwić grę dowolnej osobie, zapewniając jak najskuteczniejsze rozpoznawanie wypowiadanych komend. Czas analizy mowy powinien być możliwie krótki. Sama gra jest z punktu widzenia pracy mniej ważna, wobec czego będzie ona jedynie aplikacją typu *proof of concept* w możliwie prosty sposób demonstrującą rozwiązanie problemu.

1.3 Wybór gry

Nie we wszystkich grach komputerowych można wykorzystać sterowanie przy pomocy rozpoznawania mowy. W większości gier sterowanie jest na tyle skomplikowane, że wydawanie komend głosem byłoby nieefektywne. Ponadto, często wymagny jest od gracza bardzo krótki czas reakcji, by zadziałać adekwatnie do szybko następujących w grze wydarzeń. Ważnym kryterium jest także stopień skomplikowania gry - powinna być ona możliwie prosta.

Chciałem także, by tematyka gry była oryginalna, najchętniej, by wymyślona przeze mnie gra nie została przez nikogo wcześniej wykonana. Rozważałem dwie koncepcje: *Pilot rajdowy*, grę, w której wcielamy się w siedzącego obok kierowcy pilota, który informuje go o zbliżających się elementach trasy, oraz *Regaty*, grę, w której wcielamy się w kapitana żaglówki i wydajemy polecenia załodze.

Zachowanie żaglówki na wodzie jest, z fizycznego punktu widzeniae, bardzo skomplikowane. Implementacja silnika fizycznego opisującego to zachowanie byłaby bardzo trudna. Ważnym czynnikiem jest terminologia żeglarska, która jest dość szeroka, więc możliwych do wydania komend jest bardzo wiele. Opracowanie systemu rozpoznawania tego typu komend byłoby dużo bardziej czasochłonne, zaś sam system miałby prawdopodobnie niższą poprawność niż dla *Pilot rajdowego*.

Zdecydowałem się wobec tego na grę *Pilot rajdowy*. Fizyka jazdy samochodem jest mniej skomplikowana, bardziej intuicyjna, a wobec tego prostsza do implementacji. Ponadto pilot rajdowy ma znacznie mniejszy zakres poleceń niż kapitan żaglówki, wobec czego system rozpoznawania mowy powinien być skuteczniejszy.

1.4 Układ pracy

Praca podzielona jest na kilka części. W pierwszej omówione będą podstawy teoretyczne systemów rozpoznawania mowy i rozwiązania w nich wykorzystywane. W drugiej przedstawiona zostanie specyfika gier komputerowych w kontekście systemów rozpoznawania mowy. W kolejnej opisany będzie proces projektowania systemu, włączając w to opis napotkanych problemów oraz wybrany sposób ich pokonania. W części czwartej omówiona jest specyfika gier komputerowych, w kontekście wykorzystania w nich systemów rozpoznawania mowy jako interfejsów.

Część piąta zawiera konkretne informacje dotyczące wykonanego projektu *Pilot rajdowy*, napotkane problemy i sposób ich rozwiązania. Przedostatni rozdział przedstawia proces projektowania systemu oraz jego wyniki - tak te w trakcie prac jak i finalne. Omówiony jest wpływ poszczególnych elementów systemu na jego wyniki. Część siódma to podsumowanie pracy, na który składają się wnioski płynące z wyników zaimplementowanego systemu, omówienie możliwości wykorzystania i dalszego rozwoju tego i podobnych systemów.

Rozdział 2

Podstawy teoretyczne systemów rozpoznawania mowy

W rozdziale tym przedstawione zostaną podstawowe informacje na temat systemów rozpoznawania mowy.

2.1 Klasyfikacja systemów

Systemy rozpoznawania mowy można podzielić według kilku kategorii. Ze względu na użytkowników, mogą być to systemy dedykowane dla jednego mówcy lub dla wielu mówców. Drugi podział to systemy typu *command and control*, które opierają się na z góry zdefiniowanych komendach, oraz systemy do dyktowania, które mogą przyjąć dowolne sekwencje słów. Kolejne kryterium to sposób działania - czy jest to system czasu rzeczywistego, od którego oczekujemy wyników w bardzo krótkim czasie, czy system działający *offline*, który może analizować sygnał dowolnie długo.

Systemy rozpoznawania mowy można dzielić ze względu na rozmiar słownika. Wydziela się systemy o małych (<100 słów), średnich (<1000 słów) i dużych (>1000 słów) słownikach [4]. Systemy mogą być przystosowane do rozpoznawania mowy ciągłej lub izolowanych słów.

Sytem tworzony na potrzeby *Pilota rajdowego* to system o małym słowniku, typu *command and control*, dedykowany dla jednego mówcy (poprzez adaptację modelu akustycznego możliwy do wykorzystania przez wielu mówców), działający w czasie rzeczywistym i rozpoznający mowę ciągłą. Czyli *Real-time small vocabulary single-speaker continuous speech recognition command and control system.*

2.2 Czynniki wpływające na skuteczność

Na skuteczność systemu rozpoznawania mowy wpływ ma bardzo wiele czynników. Część z nich jest zależna od przygotowania systemu: dane treningowe modelu akustycznego (tak ilość jak i jakość), precyzja oraz zakres słownika, jakość modelu języka, czyli jak dobrze odzwierciedla on słowa wypowiadane do systemu. Część z nich zależy od użytkownika i okoliczności: jakość mikrofonu, odgłosy w tle, ew. wady wymowy lub akcent użytkownika.

2.3 Zależności między elementami systemu ASR

System rozpoznawania mowy składa się z kilku współdziałających i wzajemnie zależnych elementów. Model akustyczny to statystyczne odzwierciedlenie sygnału mowy na fonemy. Słownik zawiera reprezentacje słowa pisanego za pomocą sekwencji fonemów. Model języka z kolei opisuje prawdopodobieństwo wystąpienia określonego słowa lub sekwencji słów.

Ukryty Model Markowa to matematyczne narzędzie wykorzystywane w systemie do przetwarzania parametrów sygnału w celu zamiany go na fonemy lub sekwencje fonemów. Sekwencje te porównywane są następnie z dostępnymi w słowniku wyrazami, zaś te najbardziej prawdopodobne wykorzystywane są do tworzenia kraty słów. Pozwala ona, przy pomocy danych z modelu języka, obliczyć prawdopodobieństwa wystąpienia danych sekwencji wyrazów. Po zakończeniu sygnału dźwiękowego (natrafieniu na dłuższą ciszę) zwrócona zostanie sekwencja wyrazów o najwyższym prawodpodobieństwie. Może się to zdarzyć przed zakończeniem sygnału, jeśli jedna ze ścieżek w kracie słów będzie znacznie bardziej prawdopodobna niż pozostałe. Przetworzona reszta sygnału jest wówczas analizowana jako nowa krata słów.

W kolejnych podrozdziałach opisany zostanie każdy z wymienionych elementów systemu rozpoznawania mowy.

2.4 Model akustyczny

Model akustyczny to odwzorowanie związku między sygnałem mowy a fonemami. W swojej pracy wykorzystuję model kompatybilny ze *Sphinksem*, oparty na 13 współczynnikach MFCC (ang. *'Mel Frequency Cepstral Coefficients'*). Ukryty Model Markowa przyjmuje te 13 parametrów jako dane wejściowe. W terminologii HMM sygnał na wejściu systemu ASR to obserwacje, zaś fonemy to stany. Wykorzystywany jest model ciągły, wobec czego dla jednego zestawu danych wejściowych może zwrócić jednocześnie kilka wyników o różnych prawdopodobieństwach. Zadaniem modelu akustycznego jest obliczenie najbar-

dziej prawdopodobnej sekwencji fonemów na podstawie danych akustycznych. By model działał, musi zostać odpowiednio wytrenowany.

2.4.1 Przygotowanie modelu

Do wygenerowania modelu akustycznego potrzebne są trzy elementy: dane audio z głosem osoby lub osób, na podstawie których trenujemy model, transkrypcje dla tych danych, oraz słownik zawierający wszystkie słowa pojawiające się w transkrypcji. Ponieważ model akustyczny operuje na fonemach, a nie grafemach, poprawne przygotowanie słownika jest warunkiem koniecznym wytrenowania dobrego modelu akustycznego.

Przygotowanie danych do treningu jest także bardzo ważnym elementem przygotowania systemu ASR. By model akustyczny dobrze działał, powinien być wytrenowany dużą ilością danych. Jeśli ma być systemem dla wielu mówców, liczba osób przygotowujących dane powinna być możliwie duża, osoby te powinny być różnych płci i mieć różne barwy głosu. By model był bardziej wszechstronny, dane testowe powinny uwzględniać różne sposoby mówienia - szybko, wolno, z drobnymi różnicami w akcentach.

Ilość danych potrzebnych do otrzymania dobrych wyników systemu zależy także od rozmiaru słownika oraz stopnia skomplikowania modelu języka. Jeśli istnieje stosunkowo niewiele podobnych słów lub sekwencji słów, które mogą wystąpić podczas pracy systemu, model akustyczny nie musi być bardzo mocno wytrenowany. W przypadku systemu dla jednego mówcy ilość potrzebnych danych także jest mniejsza, w szczególności jeśli jest to system *command and control* z niewielkim słownikiem i nieskomplikowanym modelem języka.

Do trenowania modelu wykorzystany był zestaw skryptów w pakiecie *SphinxTrain*. Do przetworzenia plików audio w celu wydobycia *MFCC* wykorzystany został program *bw* z *SphinxTrain*.

2.4.2 Przygotowanie danych treningowych

By model akustyczny dawał dobre wyniki, musi dobrze odzwierciedlać dane audio, które będą przy jego pomocy przetwarzane. W szczególności, musi on zawierać informacje o wszystkich występujących fonemach. Powinien także być wyuczony możliwie dużą ilością różnych przejść między poszczególnymi fonemami, co w praktyce sprowadza się do szkolenia modelu szerokim zakresem słów, zamiast wielokrotnymi nagraniami tych samych zdań. Oczywiście na potrzeby systemu *command and control* teoretycznie wystarczy wyuczyć model tylko zdaniami występującymi w tym systemie. W praktyce jednak warto dołączyć różnorodne sekwencje testowe, ponieważ w przeciwnym wypadku przy każdym

rozszerzeniu czy modyfikacji modelu języka należałoby doszkalać model akustyczny.

Jako podstawę danych treningowych wykorzystałem zestaw zdań *Corpora* [5]. Składa się on z 114 wypowiedzi, które mają tę cechę, że zawierają wszystkie występujące w języku polskim fonemy, oraz, co ważniejsze, wszystkie występujące w języku polskim difony. Zdania te zostały nagrane dwukrotnie, raz wypowiadane spokojnym głosem, raz szybko. Drugi zestaw nagrań motywuję koniecznością dostosowania modelu do sposobu mowy w grze - często komendy będą musiały być wydane szybko.

By model lepiej sprawował się w samej grze, przygotowałem dodatkowy zestaw danych doszkalających opartych na słowach pojawiających się w grze. Szczególnie dużą wagę przywiązłem do poprawnego rozpoznawania liczb. Po pierwsze, stanowią one kluczowy element mechaniki gry - od precyzyjnego określenia kąta zakrętu czy długości prostej zależy powodzenie w grze. Po drugie, istnieją pary liczb, które brzmią podobnie i z którymi system może mieć trudności podczas rozpoznawania. Przykładowe pary to liczebniki -*naście* i - *dzieścia*, np. *trzynaście* i *trzydzieści*.

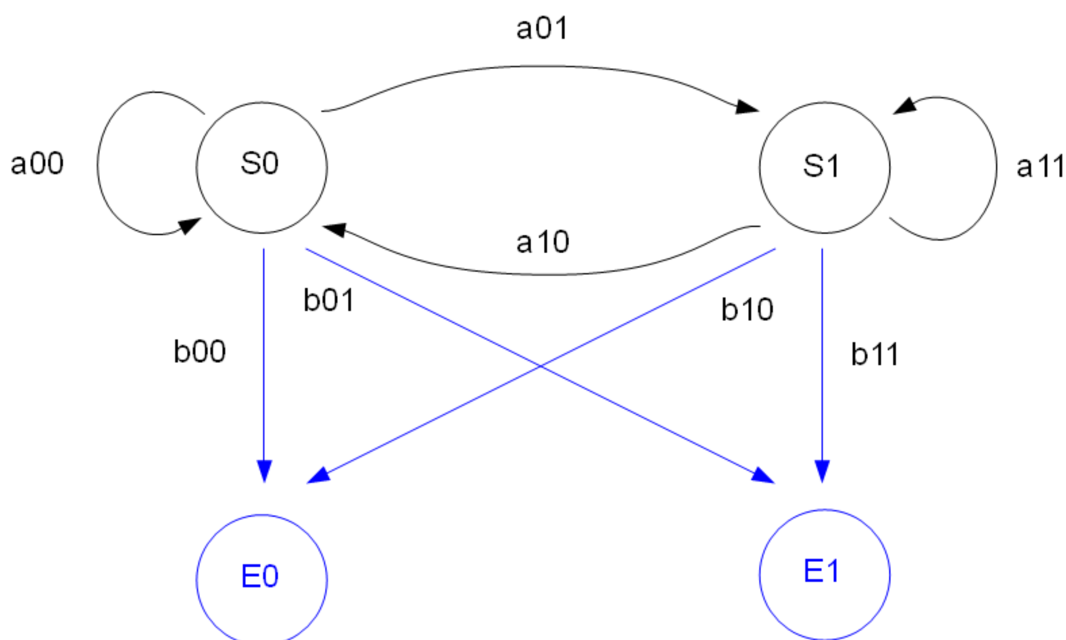
Podobne problemy sprawiają liczebniki w różnych formach, np. *pięćset* i *pięciuset*. Przykład ze słowem *tysiąc* można znaleźć w listingu 2.1. O ile błędy drugiego typu mogą zostać skompensowane przez moduł przetwarzania rozpoznanego tekstu, gdyż i tak zamienia on słowo na liczbę (w obu przypadkach byłoby to 500), to błędy pierwszego rodzaju zwykle nie mogą zostać w ten sposób naprawione. Dane treningowe specyficzne dla gry zawierały wszystkie słowa pojawiające się w systemie, większość z nich pojawiała się w nich wielokrotnie.

Podczas przygotowywania transkrypcji danych ważne jest, by zaznaczone były w niej fragmenty ciszy występujące w pliku audio. Pozwalają one algorytmowi przygotowującemu model z dużym prawodopodobieństwem określić, który fonem znajduje się w którym miejscu danych. Długie fragmenty mowy bez ciszy zmniejszają to prawodpodobieństwo.

W sumie do trenowania modelu wykorzystane zostało 25 minut danych audio. Przygotowane zostało ok. 10 minut danych więcej, lecz odrzuciłem je ze względu na zbyt długie fragmenty mowy bez ciszy - dane te zmniejszały skuteczność modelu akustycznego.

2.5 Ukryty Model Markowa

Ukryty Model Markowa (ang. *'Hidden Markov Model'*, *HMM*) to statystyczny model Markowa, w którym stany są niewidoczne dla obserwatora (ukryte) zaś wyjście, zależne od stanu, jest jawne. Wykorzystując HMM jesteśmy w stanie, na podstawie niejednoznacznych danych, wyznaczyć prawdopodobieństwa wystąpienia konkretnych sekwencji stanów. Z takim problemem borykają się systemy rozpoznawania mowy - dla danych aku-



Rysunek 2.1: Przykład prostego modelu Markowa

stycznych trzeba wyznaczyć najbardziej prawdopodobną reprezentację fonemową. Najłatwiejszy sposób na objaśnienie działania ukrytego modelu Markowa prowadzi poprzez konkretny przykład.

Jeden z najprostszych modeli przedstawiony jest na rysunku 2.1. Widzimy tam następujące elementy: S_m, które odpowiadają poszczególnym stanom w modelu, oraz E_n, które oznaczają możliwe obserwacje. Czarne strzałki mają oznaczenia a_{ij} i definiują możliwe przejścia między stanami (ze stanu i do stanu j), zaś niebieskie strzałki mają oznaczenia b_{mn} i definiują prawdopodobieństwa zaistnienia obserwacji n w stanie m.

Wartości strzałek można zapisać w postaci macierzy, jak na rysunku 2.2. Prawdopodobieństwa obserwacji w jednym stanie muszą sumować się do jedynki. Prawdopodobieństwa wyjść ze stanu także muszą sumować się do jedynki.

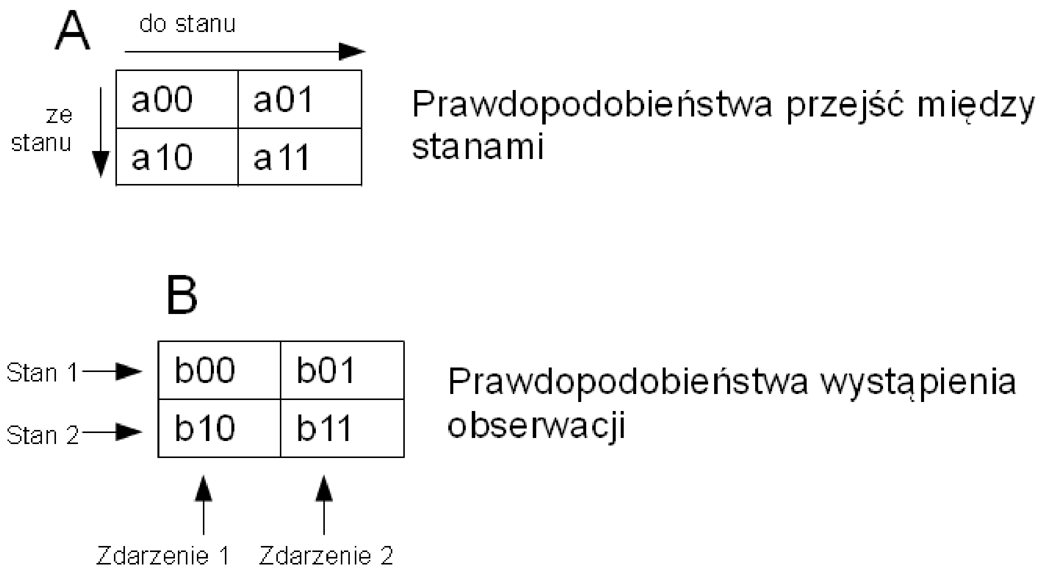
Przyjmijmy zatem wartości macierzy A i B jak z rysunku 2.3.

Spróbujmy obliczyć prawdopodobieństwo wystąpienia obserwacji E_0, E_1. Za stan pierwszy przyjmijmy S_0. Wobec tego istnieją dwa przejścia między stanami, w których może zaistnieć ciąg obserwacji E_0, E_1: S_0, S_0 oraz S_0, S_1. Obliczmy prawdopodobieństwa dla obu tych możliwości:

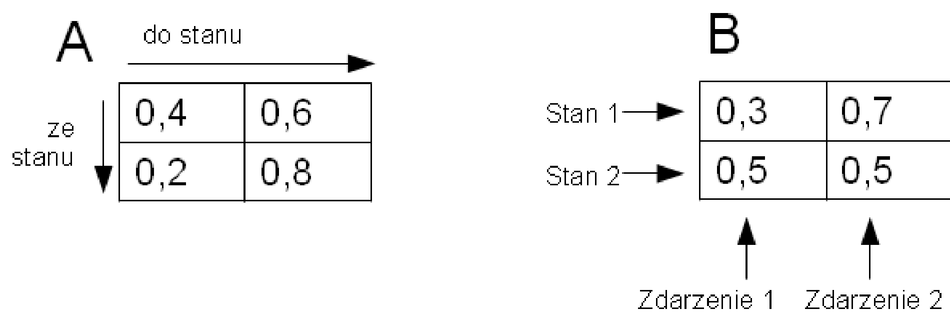
$$P(S_0, S_0) = b_{00} \cdot a_{00} \cdot b_{01} = 0.3 \cdot 0.4 \cdot 0.7 = 0.084$$

$$P(S_0, S_1) = b_{00} \cdot a_{01} \cdot b_{11} = 0.3 \cdot 0.6 \cdot 0.5 = 0.090$$

Widać zatem, że bardziej prawdopodobna jest sekwencja stanów S_0, S_1.



Rysunek 2.2: Macierze przejścia i obserwacji



Rysunek 2.3: Wartości macierzy przejść i obserwacji

Przedstawiony tu został przykład dyskretnego modelu Markowa, czyli takiego, w którym ilość obserwacji jest skończona i policzalna. W rzeczywistych systemach ASR wykorzystuje się zwykle modele HMM z ciągłymi wyjściami, które modelują wartość wyjściową za pomocą kombinacji liniowych rozkładów normalnych na podstawie danych wejściowych. Modele ciągłe są lepiej przystosowane od analizy mowy ze względu na jej dynamiczny, niedyskretny charakter. Właśnie takie modele stosowane są we frameworku *Sphinx* i w *Pilocie rajdowym*.

Obserwacjami w modelach wykorzystywanych w systemach ASR są wydobywane z mowy parametry, np. używane przeze mnie *MFCC*. Stany to wewnętrzne reprezentacje odpowiadających określonym parametrom fonemów.

2.6 Słownik

Słownik zawiera słowa występujące w systemie oraz ich zapis fonetyczny. Ten sam słownik jest wykorzystywany podczas treningu modelu oraz podczas rozpoznawania mowy. Jest on swego rodzaju pomostem między słowem pisanym a mową.

Zapis fonetyczny w słowniku *Pilota Rajdowego* wykorzystuje alfabet *SAMPA* [6], zmodyfikowany przez mnie na potrzeby wykorzystania w *Sphinksie*.

Ze względu na brak rozróżnienia małych i wielkich liter w aplikacji, niezbędna była zmiana znaków fonemów wykorzystujących w *SAMPA* wielkie litery na inne znaki.

Sam słownik powinien być jak najdokładniejszy. Niepoprawna transkrypcja fonetyczna wyrazów w słowniku może znacząco obniżyć jakość powstałego przy jego wykorzystaniu modelu, gdyż do niepoprawnego fonemu lub sekwencji fonemów zostaną przypisane niepoprawne obserwacje. Co gorsza, błędna transkrypcja może poważnie pogorszyć parametry modelu akustycznego, poprzez przypisanie niepasującego zestawu parametrów do danego fonemu.

Te same wyrazy mogą mieć różne reprezentacje fonetyczne. Uwzględnianie alternatywnych wymów jest ważnym elementem przygotowywania modelu, szczególnie, jeśli ma on być wykorzystywany przez więcej niż jedną osobę.

Poniżej zamieszczony jest fragment słownika z gry. Warto zwrócić uwagę na podobieństwo czterech różnych słów pochodzących od słowa *tysiąc*, z których każde posiada dwa sposoby wymowy. Tak podobne wyrazy często sprawiają problem podczas rozpoznawania mowy. W języku polskim, ze względu na jego silną fleksyjność [7], tego typu problemy pojawiają się często.

Listing 2.1: Fragment słownika z gry *Pilot rajdowy*

```
tysiąc      t y si o n ts
```

```
tysiąc(2)    t y si on ts
tysiące      t y si o n ts e
tysiące(2)   t y si on ts e
tysiącu      t y si o n ts u
tysiącu(2)   t y si on ts u
tysięcy      t y si e n ts y
tysięcy(2)   t y si en ts y
zakręt       z a k r en t
łagodnie     w a g o d ni e
```

2.7 Model języka

Model języka to narzędzie, które pozwala na obliczanie prawdopodobieństwa wystąpienia określonej sekwencji słów. W miarę napływu danych do systemu, analizowane są wszystkie możliwości pasujące do modelu. Są one także na bieżąco ewaluowane pod względem prawdopodobieństwa ich wystąpienia. Te najmniej prawdopodobne sekwencje są systematycznie usuwane z listy tych analizowanych, by zmniejszyć złożoność obliczeniową całego procesu. Istnieją różne struktury danych wykorzystywane w tym procesie, mogą to być np. kraty słów (ang. *'Word Lattices'*), acykliczne grafy skierowane, w których każda ścieżka od węzła początkowego odpowiada jednej możliwej sekwencji słów. Inną strukturą są sieci niepewności (ang. *'Confusion Network'*), które także są acyklicznymi grafami skierowanymi, z dodatkowym wymaganiem, by każda ścieżka przez graf przechodziła przez wszystkie węzły.

W pracy tej wykorzystałem dwa rodzaje modeli języka - model statystyczny n-gram oraz gramatykę JSGF.

2.7.1 Model n-gram

Model *n-gram* [8] składa się z zestawów *n-gramów*, czyli n-elementowych sekwencji słów. Każda taka sekwencja ma przypisane prawodpodobieństwo wystąpienia. Dla bigramowego modelu (n=2) prawdopodobieństwo wystąpienia zdania *Ala ma kota* obliczone zostałoby następująco:

$$P(Ala, ma, kota) = P(Ala| < s >) \cdot P(ma|Ala) \cdot P(kota|ma) \cdot P(< /s > |kota),$$

gdzie *P(a—b)* to prawdopodobieństwo wystąpienia słowa a pod warunkiem wcześniejszego wystąpienia słowa b, zaś$<s>$ i $</s>$ to symbole oznaczające ciszę - pierwszy ciszę przed rozpoczęciem, drugi ciszę po zakończeniu mówienia.

Modele wykorzystywane w *Sphinksie* mogą zawierać jednocześnie zarówno uni-, bi- jak i trigramy. Implementacja modelu w *Sphinksie* rekurencyjnie wyszukuje dla każdej sekwencji słów n-gramy w modelu, zaczynając od najdłuższych, kończąc na unigramach. Przykładowo, gdyby w modelu języka nie istniał bigram *Ala ma*, wówczas *Sphinx* obliczyłby prawdopodobieństwo nastepująco:

$$P(Ala, ma, kota) = P(Ala| < s >) \cdot (P(ma) \cdot uw) \cdot P(kota|ma) \cdot P(< /s > |kota),$$

gdzie *uw* to *UnigramWeight* - współczynnik konfigurowany w aplikacji, który określa wagę unigramów w stosunku do bigramów i trigramów.

Fragment definicji modelu n-gram z gry *Pilot rajdowy* znajduje się na listingu 2.2. Wszystkie wartości podawane są w skali logarytmicznej. Wartość w pierwszej kolumnie to prawdopodobieństwo wystąpienia danego wyrazu lub sekwencji wyrazów. W drugiej kolumnie znajduje się sekwencja słów, z której składa się n-gram.

Listing 2.2: Fragment modelu n-gram z gry *Pilot rajdowy*

```
-3.4792   zredukuj                                  -0.2986
-2.5871   zwolnij                                   -0.2986
-2.5984   łagodnie                                  -0.2790

-2.3385   prosta dwadzieścia                        -0.1445
-0.9031   skręć bardzo                              -0.2218
-0.6378   sześćdziesiąt </s>                        -0.3010
-1.0000   wrzuć piątkę                               0.0000

-2.3385   <s> prosta pięćdziesiąt
-0.3010   czterdziestu metrach przechodzi
-0.6021   dwieście pięćdziesiąt </s>
```

Wartość w trzeciej kolumnie to wartość prawdopodobieństwa *wycofania się* (stąd nazwa *backoff*) do wykorzystania krótszego n-gramu do obliczeń prawdopodobieństwa. De facto wartość ta zależy od ilości różnych słów mogących wystąpić po danym n-gramie - jeśli jest ich mało, algorytm może zdecydować się przeanalizować następny wyraz tylko biorąc pod uwagę wyraz poprzedni, zamiast dwóch poprzednich. W powyższym przykładzie widać bigram *wrzuć piątkę*. Wartość *backoff* wynosi dla tego n-gramu:

$$log_{10}1 = 0, \;\; backoff = 10^0 = 1$$

Czyli po napotkaniu takiej sekwencji wyrazów algorytm z pewnością zdecyduje się na wykorzystanie tego bigramu. Jest to słuszne, ponieważ bigram ten jest kompletnym poleceniem systemu i sekwencja ta nie występuje w innych konfiguracjach.

Algorytm *backoff* [9] jest rozszerzeniem typowego modelu n-gram, dającym nieco większą elastyczność i w niektórych przypadkach poprawiającym wyniki. Ten rodzaj modelu n-gram jest integralną częścią *Sphinksa*. Wpływ zastosowania rozszerzenia *backoff* na wyniki nie był badany.

2.7.2 Negatywne n-gramy

Model statystyczny oparty na n-gramach zawsze dopuszcza możliwość wystąpienia sekwencji nie opisanej bi- ani trigramami. Negatywne n-gramy to takie n-gramy, których prawdopodobieństwo wystąpienia wynosi zero. Można je wykorzystywać do wykluczania z możliwych do rozpoznania zdań pewnych sekwencji, które z jakiegoś powodu są nieprawidłowe. Jeśli dana sekwencja słów zostanie napotkana w kracie słów, algorytm oceniający przyzna danej ścieżce prawdopodobieństwo zero, przez co zostanie ona odrzucona.

Wygenerowanie wszystkich kombinacji słów, które nie mogą wystąpić w systemie jest raczej niepraktyczne, ponieważ ich ilość byłaby ogromna - wielokrotnie większa od ilości n-gramów opisujących dopuszczalne sekwencje. Można jednak przy wykorzystaniu negatywnych n-gramów wykluczać często pojawiające się w systemie błędy, w szczególności np. błędy gramatyczne.

W systemie w *Pilocie rajdowym* często spotykane były błędy w rodzaju *przyśpiesz do siedemdziesiąt*. Słowo *siedemdziesiąt* jest podobne do *siedemdziesięciu*. Poprzez dodanie negatywnego n-gramu *przyśpiesz do siedemdziesiąt* o zerowym prawdopodobieństwie, można wykluczyć występowanie tej sekwencji w jakimkolwiek rozpoznaniu.

Negatywne n-gramy, w innym zastosowaniu, wykorzystane zostały w pracy czeskich naukowców w 2003 roku [10].

Koncepcja wykorzystania ich na potrzeby eliminowania niepożądanych sekwencji w rozpoznawaniu mowy jest moim własnym pomysłem.

2.7.3 Gramatyka JSGF

Gramatyka JSGF [11] (ang. *'Java Speech Grammar Format'*) jest drugim sposobem opisu języka wykorzystanym w tej pracy. JSGF nie jest modelem statystycznym. Pozwala na precyzyjne definiowanie komend do wykorzystania w systemie ASR. Ważną cechą JSGF jest brak elastyczności definiowanych komend. W systemie opartym o tę gramatykę użytkownik musi nauczyć się odpowiedniej składni i ją wykorzystywać, ponieważ w przeciwnym wypadku jego słowa nie zostaną poprawnie rozpoznane.

Fragment definicji gramatyki z gry *Pilot rajdowy* znajduje się na listingu 2.3. Opisuje on komendę skrętu.

Listing 2.3: Fragment definicji gramtyki JSGF z gry *Pilot rajdowy*

```
<liczba10n>        = dziesięć · dwadzieścia · trzydzieści · czterdzieści
                     · pięćdziesiąt · sześćdziesiąt · siedemdziesiąt
                     · osiemdziesiąt · dziewięćdziesiąt;
<liczbakatn>       = [ sto ] <liczba10n> ;
<rodzajZakretu>    = bardzo ostry · łagodny · ostry;
<skretslowo>       = skręt · skręć · zakręt;
<kierunekskretu>   = lewo · prawo;
public <zakret1>   = [ <rodzajZakretu> ] <skretslowo> w <kierunekskretu>
                     <liczbakatn> [ stopni ];
public <zakret2>   = <skretslowo> [ <rodzajZakretu> ] w <kierunekskretu>
                     <liczbakatn> [ stopni ];
```

Zgodnie z tą definicją zdanie *zakręt ostry w lewo czterdzieści stopni* jest poprawne. Zdanie *łagodnie skręć czterdzieści stopni w lewo* już nie, ponieważ według definicji kierunek skrętu - lewo - musi wystąpić przed kątem zakrętu.

Gramatyka JSGF wymaga wobec tego przewidzenia oraz opisania, w formalny sposób, każdej możliwej składni komendy w aplikacji. Już z tego założenia wynika, że nie jest to dobre narzędzie do tworzenia systemów o dużym słowniku lub skomplikowanej składni, ponieważ liczba reguł potrzebnych do opisania takiego systemu byłaby ogromna.

Korzystanie z reguł JSGF utrudnia także fakt, iż nie ma możliwości określania prawdopodobieństwa poszczególnych zdań. Wszystkie komendy publiczne traktowane są tak samo. Możliwe jest jedynie określenie względnej wagi każdego z wyrazów wewnątrz grupy, co przedstawiono na listing 2.4

Listing 2.4: Określanie prawdopodobieństw wyrażeń alternatywnych w gramatyce JSGF

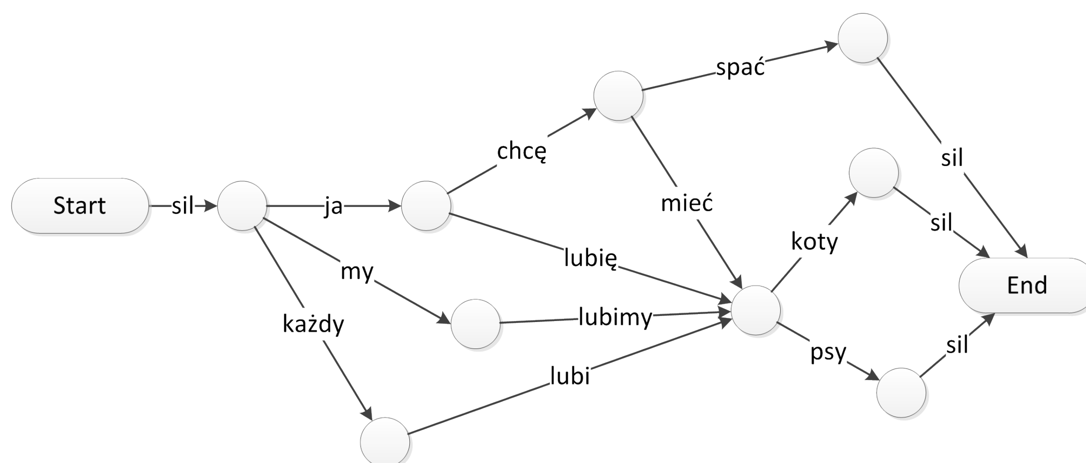
```
<liczba10n> = /10/ dziesięć | /10/ dwadzieścia | /5/ trzydzieści
            | /5/ czterdzieści | /5/ pięćdziesiąt | /3/ sześćdziesiąt
            | /3/ siedemdziesiąt | /3/ osiemdziesiąt | dziewięćdziesiąt;
```

Model n-gram daje dużo większe możliwości w kwestii różnicowania prawdopodobieństw poszczególnych zdań, sekwencji słów, czy nawet pojedynczych wyrazów.

2.8 Kraty słów

Kraty słów to acykliczne grafy skierowane, w których do każdej krawędzi przypisane jest słowo. Każda możliwa ścieżka w takim grafie reprezentuje jedno możliwe zdanie analizowane przez system. Przykład kraty słów znajduje się na rysunku 2.4

Na podstawie danych z modelu języka, w połączeniu z danymi z modelu akustycznego, możliwe jest określenie prawdopodobieństwa wystąpienia danej ścieżki. Ze względu



Rysunek 2.4: Przykładowa krata słów

na ogromną ilość możliwości wypowiedzi, te ścieżki, których prawdopodobieństwo jest najniższe, są w trakcie pracy systemu odrzucane. Ostatecznie, jako wynik rozpoznwania mowy, zwracane jest zdanie odpowiadające najbardziej prawdopodobnej ścieżce w kracie słów.

2.9 Sieci niepewności

Sieci niepewności są podobne do krat słów. Narzucone jest na nie dodatkowe ograniczenie - wszystkie ścieżki muszą wieść przez wszystkie węzły w sieci. Sprowadza się to do tego, że między niektórymi węzłami istnieje więcej niż jedna krawędź, a każda z nich odpowiada alternatywnemu słowu mogącemu się w tym miejscu pojawić.

Do każdej krawędzi jest także przypisane prawdopodobieństwo wystąpienia danego słowa. Suma prawdopodobieństw krawędzi łączących dwa określone węzły jest zawsze równa 1.

2.10 Istniejące rozwiązania

Systemy rozpoznawania mowy dynamicznie rozwijają się od lat 50. XX wieku, choć wykorzystywane technologie w zasadzie nie zmieniły się od lat 80. Ukryte łańcuchy Markowa, kraty słów, modele statystyczne wciąż są podstawą systemów ASR. Prowadzone są m. in. badania nad odpornością systemów ASR na szumy [12] czy transkrypcją mowy niegramatycznej [13]. Rozwijane są także instniejące metody i algorytmy, np. metody adaptacji modeli akustycznych [14] [15].

Frameworki akademickie, pozwalające na konstruowanie systemów rozpoznawania mo-

wy, rozwijane są od lat 80. XX wieku. Spośród nich na uwagę zasługuje framework *Sphinx* [16], w szczególności najnowsza jego odsłona, napisana w Javie, *Sphinx 4*, oraz *HTK* [17], napisany w C.

Niektóre koroporacje zajmujące się tworzeniem oprogramowania starają się ustandardyzować dziedzinę rozpoznawania mowy. Microsoft w 2005 roku udostępnił *SAPI* [18], która jest warstwą pośrednią między konkretnymi rozwiązaniami do rozpoznawania i syntezy mowy, a interfejsem użytkownika w systemie Windows. Analogicznym rozwiązaniem firmy Oracle (dawniej SUN Microsystems) jest *Java Speech API* [19].

Aplikacje i systemy zajmujące się stricte rozpoznawniem mowy także są dostępne komercyjnie. System *MagicScribe* [20] pozwala na zamianę słowa mówionego na pisane. Dostępne są dwie jego wersje dostosowane do potrzeb medycyny i prawa, *MagicScribe Medical* i *MagicScribe Legal*.

Systemy typu *Command and control* już od wielu lat stosowane są komercyjnie. Jednym z popularniejszych ich zastosowań są automatyczne, telefoniczne linie informacyjne lub interaktywna bankowość telefoniczna, tzw. *IVR* (ang '*Interactive Voice Response*'). Polskim przykładem *IVR* jest system [21] zastosowany przez Zarząd Transportu Miejskiego w Warszawie, uruchomiony w 2008 roku, który udziela dzwoniącemu informacji na temat rozkładów jazdy lub możliwych tras dojazdu.

W grach komputerowych systemy rozpoznawania mowy także zostały już wykorzystane. Dwie wysokobudżetowe gry, które osiągnęły pewną popularność, to *Tom Clancy's HAWX* [22] oraz *Tom Clancy's Endwar* [23].

2.11 Wybór technologii

Podstawową kwestią był wybór frameworka, którego mógłbym użyć jako podstawy swojego systemu rozpoznawania mowy. Poważnie brane pod uwagę były *Sphinx4* oraz *HTK*.

Dokonując wyboru, brałem pod uwagę możliwości obu frameworków, szybkość ich działania, prostotę wykorzystania oraz dostępność dokumentacji.

Sphinx4 jest frameworkiem napisanym w Javie. Wykorzystuje udostępnione przez Sun *Java Speech API*, działa na wszystkich systemach z wirtualną maszyną Javy. Ze względu na charakterystykę języka oferuje łatwą rozbudowę, modyfikację oraz wsparcie dla programistycznego IDE. Z tego samego względu wykorzystuje stosunkowo dużo pamięci, dodaje także pewien narzut w wykorzystaniu procesora. *Sphinx* udostępnia zestaw narzędzi umożliwiających generowanie i doszkalanie modeli akustycznych oraz generowanie modeli języka n-gram.

HTK jest napisane w C/C++. Daje podobne możliwości jak *Sphinx*, ale jest trudniej-

szy w wykorzystaniu z punktu widzenia programisty. Ze względu na wykorzystany język wprowadzanie zmian do kodu jest mozolne, podobnie jak ich testowanie. Framework ten także udostępnia zestaw narzędzi do pracy nad modelem akustycznym i języka.

Ostatecznie zdecydowałem się na wykorzystanie *CMU Sphinx*. Przeważyła łatwość użytkowania oraz dostępne narzędzia do programowania w Javie. Narzut wymaganej mocy obliczeniowej i pamięci, przy współczesnych możliwościach komputerowych jest mało ważny. Wybór *Sphinksa*, z perspektywy czasu, zdaje się być słuszny. Pozwolił mi on skupić się na zagadnieniach rozpoznawania mowy, a nie na pokonywaniu kolejnych problemów implementacyjnych.

Rozdział 3

Framework *CMU Sphinx*

CMU Sphinx to framework i zestaw narzędzi, który pozwala na budowanie aplikacji wykorzystujących rozpoznawanie mowy. Powstał na uniwersytecie *Carnegie Mellon* w Pittsburgh w USA. Pierwsza, historyczna wersja *Sphinksa* została zaprezentowana w 1990 roku [24].

W roku 2000, na zasadach licencji opartej na BSD, upubliczniony został *Sphinx 2*. Został napisany z myślą o efektywnym przetwarzaniu i rozpoznawaniu mowy ciągłej w czasie rzeczywistym.

Sphinx 3 miał z kolei rozpoznawać mowę z dużą dokładnością, ale nie w czasie rzeczywistym.

Sphinx 4, który jest wykorzystywany w mojej pracy, to przepisany na język Java silnik z poprzedniego wydania *Sphinksa*. Z założenia miał on być elastycznym frameworkiem do prowadzenia badań nad rozpoznawaniem mowy. Jest on wciąż rozwijany, nie tylko przez zespół na *CMU*, ale także przez naukowców z *Mitsubishi Electric Research Laboratories* i *MIT*.

Zestaw narzędzi, nazwany *Sphinx toolkit*, był rozwijany na przestrzeni lat równolegle z kolejnymi wersjami *Sphinksa*. Zawiera on aplikacje pozwalające na generowanie modeli akustycznych, adaptację modeli akustycznych oraz generowanie modeli języka, oraz szereg aplikacji wspomagających te procesy.

3.1 Elementy systemu *Sphinx 4*

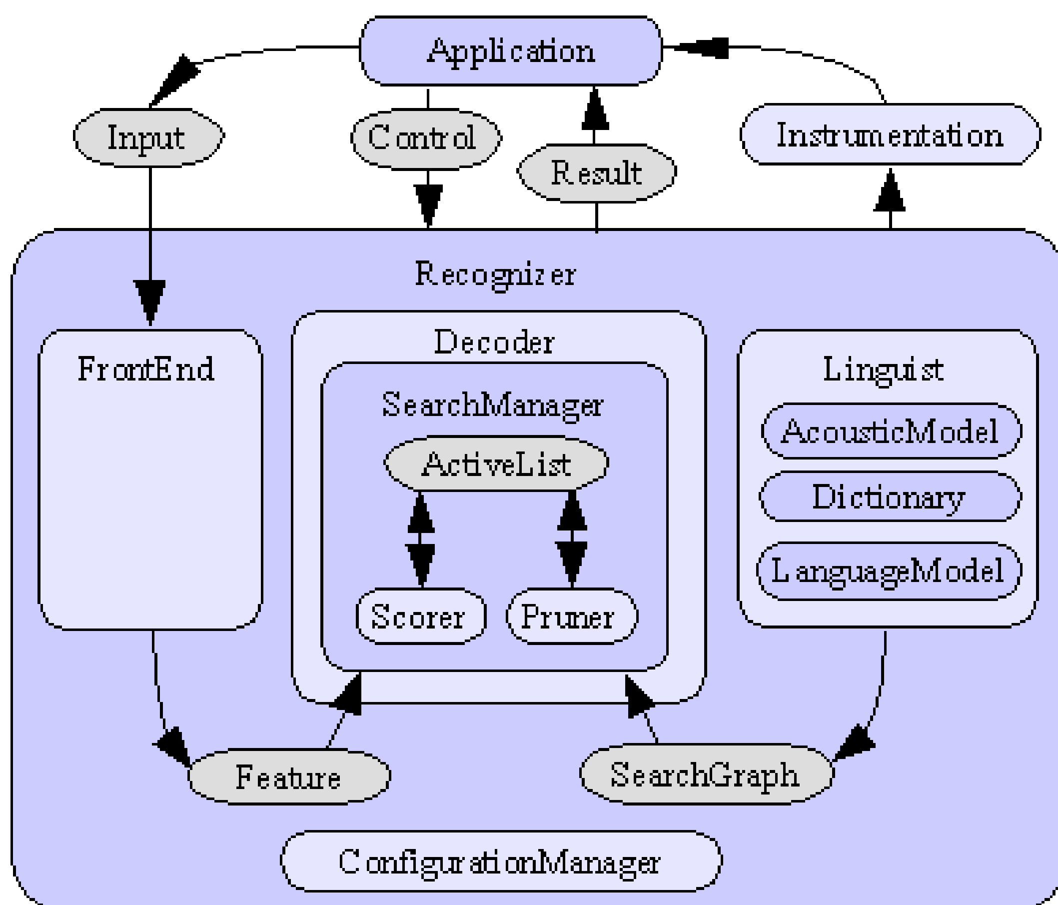
W ogólności *Sphinx* składa się z elementów opisanych w rozdziale 2. Posiada on jednak własną, specyficzną dla języka obiektowego architekturę, która zostanie teraz opisana, ze szczególnym uwzględnieniem relacji między jej elementami.

Każdy z elementów pełni w systemie jedną, konkretną funkcję. Większość z wymie-

nionych elementów posiada kilka implementacji. Dla przykładu, w *Sphinksie* dostępne są następujące modele języka: *ClassBased, Interpolated, Keyword, LargeNGram, Network, SimpleNGram*.

Zgodnie z rysunkiem 3.1, w *Sphinksie* rozróżniamy następujące elementy:

- Input - to urządzenie, lub w ogólności - strumień, wejścia. Może to być mikrofon, plik audio czy strumień danych z Internetu.
- Control - to obiekty oferujące dodatkowe, czyli nie uwzględniające danych audio, możliwości wpływania na elementy systemu. Możliwe jest np. zmienianie pewnych wartości konfiguracji w trakcie pracy systemu, w szczególności np. modyfikacja modelu języka.
- Recognizer - to obiekt kontrolujący pracę całego systemu i zapewniający współpracę wszystkich jego elementów.
- Linguist - łączy w całość model akustyczny, słownik oraz model języka. Na podstawie tych danych generuje graf poszukiwań, np. kratę słów.
- AcousticModel - reprezentuje model akustyczny wykorzystywany przez system.
- Dictionary - reprezentuje słownik.
- LanguageModel - reprezentuje model języka.
- FrontEnd - to zestaw wstępnych filtrów, które przetwarzają przychodzący strumień audio. Mogą one także ten strumień modyfikować, np. wzmacniać lub wycinać fragmenty. Mogą także dodawać do danych dodatkowe oznaczenia, jak np. czy dany fragment jest mową czy nie.
- Feature - moduł odpowiedzialny za przetwarzanie sygnału mowy na zestaw parametrów. Mogą to być np. współczynniki kepstralne (*MFCC*).
- Scorer - analizuje wygenerowane przez moduł *Feature* współczynniki i przy współpracy z modułem *Pruner* tworzy na podstawie grafu poszukiwań prawdopodobne łańcuchy rozpoznań, a następnie zapisuje je w *ActiveList*.
- Pruner - analizuje znajdujące się w *ActiveList* łańcuchy rozpoznań i usuwa te mniej prawdopodobne.
- Instrumentation - moduł pozwalający na informowanie użytkownika o stanie modułu Recognizer. Może np. powiadamiać o ilości zużytej pamięci, wykorzystanej mocy obliczeniowej procesora, lub generować dane statystyczne.



Rysunek 3.1: Schemat systemu ASR *Sphinx 4* [25]

- ConfigurationManager - moduł pomocniczy, czuwający nad konfiguracją wszystkich elementów systemu.

3.2 Korzystanie z systemu *Sphinx 4*

Rozległa wiedza z dziedziny programowania nie jest konieczna, by wykorzystać *Sphinx 4* jako system rozpoznawania mowy. Niezbędne jest natomiast opanowanie struktury samego frameworku, jego elementów i, co najważniejsze, składni i struktury pliku konfiguracyjnego.

By przygotować *Sphinksa* do pracy niezbędna jest maszyna wirtualna Java w wersji minimum 6. Kompilacja frameworku odbywa się przy pomocy skryptów *Ant*, więc ich posiadanie w systemie operacyjnym także jest konieczne. By wygodnie modyfikować lub uczyć się kodu *Sphinksa* przydatne jest IDE, jak np. *Eclipse* czy *NetBeans*.

Plik konfiguracyjny aplikacji korzystającej ze *Sphinksa 4*, zapisywany w postaci XMLa

i jest szkieletem całego systemu. Zdefiniowane są w nim wykorzystane implementacje konkretnych elementów systemu, ich parametry oraz niektóre zależności.

Przykład fragmentu pliku konfiguracyjnego widać na listingu 3.1. Opisuje on element *trigramModel*, który jest modelem języka n-gram wykorzystywanym w aplikacji. Podana jest ścieżka do pliku z definicjami n-gramów, referencja do obiektu *dictionary*, czyli słownika wykorzystywanego w systemie. Z kolei zmienna *maxDepth* określa maksymalną długość (w słowach) n-gramu wykorzystywanego przez model. Ustawienie jej na 2 oznaczałoby de facto korzystanie z modelu bigramowego, a nie trigramowego.

Listing 3.1: Przykład elementu konfiguracji frameworku *Sphinx 4*

```
<component name="trigramModel"
     type="edu.cmu.sphinx.linguist.language.ngram.SimpleNGramModel">
  <property name="location"
       value="resource:/racePilot.lm"/>
  <property name="logMath" value="logMath"/>
  <property name="dictionary" value="dictionary"/>
  <property name="maxDepth" value="3"/>
  <property name="unigramWeight" value=".7"/>
</component>
```

Sposób konfiguracji *Sphinksa* jest bardzo przystępny i przejrzysty. Dzięki wykorzystaniu XMLa zmiana konfiguracji jest bardzo prosta. Wprowadzono także dodatkowe usprawnienia, np. format pliku konfiguracji umożliwia definiowanie zmiennych globalnych, które można wykorzysytwać w różnych elementach systemu.

3.3 Wprowadzone modyfikacje

Choć *Sphinx 4* daje szerokie możliwości, by wykorzystać go do moich celów niezbędne było wprowadzenie do kodu kilku modyfikacji.

Przede wszystkim musiałem zmienić kodowanie znaków wspierane przez *Sphinksa*. Wymagało to odnalezienia we frameworku miejsc, w których wczytywane są dane i zmodyfikowanie wczytywania tak, by wspierało wymagane przeze mnie kodowanie. Domyślnie wykorzystywane było kodowanie typowe dla systemu, na którym uruchomiony był *Sphinx*. Wszędzie w moim projekcie wykorzystywane jest kodowanie UTF-8.

Inną modyfikacją było poprawienie modułu *Sphinksa* odpowiedzialnego za gramatykę JSGF. Niestety programiści w *CMU* pominęli implementację funkcji zwaracania informacji o błędach w pliku z definicjami gramatyki. Z punktu widzenia użytkownika próba wczytania niepoprawnych definicji kończyła się nic nie mówiącym błędem systemu. Konieczne było zapoznanie się z działaniem modułu obsługującego JSGF, następnie zlokalizowanie

źródła wiadomości o błędzie, a potem przygotowanie kodu obsługującego komunikaty o błędach w gramatyce.

By zgodnie z założeniami zintegrować *Pilota Rajdowego* ze *Sphinksem*, niezbędne było przygotowanie dodatkowego elementu *frontendu* (patrz rysunek 3.1) frameworku. Gra wymagała, by proces rozpoznawania zaczynał się wraz z naciśnięciem klawisza, zaś kończył się po puszczeniu go. Tego typu funkcjonalność nie była dostępna w *Sphinksie*. Zaimplementowałem więc klasę *KeyPressedSpeechClassifier*, która obserwuje naciśnięcia odpowiedniego klawisza i, w zależności od jego stanu, oznacza napływające do *Sphinksa* dane jako ciszę lub mowę.

Oprócz powyższych zmian, podczas pracy nad systemem przygotowałem zestaw skryptów wspomagających generowanie modelu akustycznego, adaptację modelu oraz wykonywanie sekwencji testów z różnymi konfiguracjami.

Rozdział 4

Specyfika gier komputerowych w kontekście systemów ASR

W rozdziale tym omówione zostaną możliwości zastosowania rozpoznawania mowy w najpopularniejszych typach gier. Ponieważ sposób wykorzystania systemu ASR zmienia się w zależności od gatunku gry, opisane zostaną także wymagania systemów ASR oraz specyficzne warunki, w których system będzie pracował.

4.1 Możliwości wykorzystania rozpoznawania mowy w grach

Do każdej gry można zaprojektować i zaimplementować interfejs użytkownika opierający się na rozpoznawaniu mowy. Nie w każdej grze tego typu interfejs byłby użyteczny, zaś w niektórych musiałby mieć ograniczoną funkcjonalność w stosunku do interfejsu tradycyjnego, takiego jak myszka czy klawiatura. Wydaje mi się słuszną koncepcją, by rozpoznawanie mowy wykorzystywać do wykonywania w grach czynności, które w świecie rzeczywistym także wiążą się z emisją głosu.

Tego typu interakcja z komputerem, nie dość, że bardzo ciekawa i oryginalna, byłaby dla gracza czymś zupełnie naturalnym, dzięki czemu łatwo byłoby mu przyzwyczaić się do wydawania komend głosem. Trzeba jednak zaznaczyć, że byłoby tak tylko wtedy, gdyby same komendy, czyli de facto gramatyka w zastosowanym modelu, dobrze odzwierciedlały rzeczywistość. Gracz nie musiałby się ich uczyć, a nawet gdyby komendy miały specyficzną składnię, jak terminologia wykorzystywana przez pilotów rajdowych, lub zawierały terminologię z nieznanej wcześniej dziedziny (np. terminologia żeglarska lub wojskowa), nauka nie byłaby trudniejsza, niż w podobnej sytuacji w rzeczywistości.

4.2 Wymagania dla systemu

Współczesne gry komputerowe to komercyjne produkty z milionowymi budżetami. By gra osiągnęła sukces (czyli dobrze się sprzedała), gra w nią musi sprawiać przyjemność i nie być w żaden sposób irytująca czy nużąca. Jeśli gra ma wykorzystywać system rozpoznawania mowy, musi on spełniać kilka warunków.

Przede wszystkim, czas reakcji systemu na wypowiedzianą komendę musi być krótki. Ze względu na dynamiczną rozgrywkę w większości gier komputerowych system z dużym czasem przetwarzania byłby nie do przyjęcia. Komendy wykorzystane w systemie muszą być proste, logiczne i spójne, powinny też mieć odzwierciedlenie w rzeczywistości. Poprawność działania systemu powinna być jak najwyższa. Minimum poprawnie rozpoznanych komend powinno wynosić 90%. Co więcej, taką poprawność powinna osiągać dla każdego z graczy i to nawet przy wykorzystaniu niskiej jakości mikrofonu.

System ASR wykorzystany w grze komputerowej powinien także wykorzystywać niewiele zasobów, gdyż gry bardzo silnie wykorzystują procesor i pamięć nawet bez wsparcia dla rozpoznawania mowy.

4.3 Czynniki wpływające na system ASR

Wykorzystanie systemów ASR w grach komputerowych ma swoje mocne i słabe strony. Przede wszystkim, wykorzystanie rozpoznawania mowy w charakterze interfejsu implikuje system typu *command and control* o stosunkowo niewielkim słowniku (małym, do 100 słów, lub średnim, do 1000 słów). Współczesne tego typu systemy często osiągają poprawność bliską 100%.

Ponieważ na jednym komputerze gra ma zwykle jednego użytkownika (a nawet, jeśli ma kilku, zwykle mają oddzielne tzw. profile, więc gra wie, kto z niej korzysta), bardzo łatwe jest gromadzenie danych do doszkalania modelu akustycznego. Może to być robione jeszcze przed rozpoczęciem prawdziwej rozgrywki w formie sesji treningowej lub po prostu w trakcie gry. Wobec tego nawet, jeśli na początku system nie miał dostatecznie wysokiej poprawności, po jego doszkoleniu znacząco się ona zwiększy.

Ponadto, model języka może być dynamicznie dostosowywany do tego, co dzieje się w grze. W zależności od sytuacji gra może przewidywać, że dana komenda zostanie wydana i nieznacznie zwiększać jej prawdopodobieństwo lub obniżać prawdopodobieństwo innej komendy, która jest do niej podobna a raczej nie pasuje do stanu gry.

Chociaż z założenia system ASR użyty w grze komputerowej powinien wykorzystywać niewielką część dostępnych zasobów, wcale nie oznacza to, że tych zasobów jest mało. W

chwili pisania tej pracy na rynku nowych komputerów standardem są procesory z czterema rdzeniami o taktowaniu nieco powyżej 3GHz, zaś ilość pamięci RAM waha się między 4 a 16 GB. Nawet jeśli system będzie miał do dyspozycji 10-20% tych zasobów, będzie to, w szczególności dla systemu *command and control*, ilość wystarczająca do pracy.

Czynniki negatywnie wpływające na potencjalną skuteczność wykorzystania systemów ASR w grach komputerowych to przede wszystkim czynniki środowiskowe, nie związane bezpośrednio ze specyfiką rozpoznawania mowy. Przykładowo, większość mikrofonów instalowanych w słuchawkach jest stosunkowo kiepskiej jakości. Nawet półprofesjonalne słuchawki stosowane przez graczy, choć oferują naprawdę dobrą jakość odtwarzania, nie dają nagrań dobrej jakości. Do rozmowy przez Skype lub w trakcie gry - jakość ta jest wystarczająca. W przypadku systemu rozpoznawania mowy - może sprawiać, że dokładność systemu drastycznie spadnie.

Dodatkowe problemy mogą sprawiać dźwięki tła obecne w domu - od szumu głośnego wiatraka na karcie graficznej, aż do rozmowy toczącej się gdzieś obok. Innymi słowy, wszystkie dźwięki, typowe dla domu, są z punktu widzenia rozpoznawania mowy czynnikiem negatywnym, ponieważ obniżają SNR nagrywanej mowy.

Oprócz tego, wyniki pogarszać mogą zmiany w sposobie mówienia podczas gry, spowodowane np. emocjami towarzyszącymi rozgrywce. System będzie też sobie słabiej radził z osobami mówiącymi inaczej niż te, na podstawie głosu których generowany był model akustyczny. Wliczają się w to różne akcenty, wady wymowy, gwary, etc.

4.4 Analiza możliwości wykorzystania systemów ASR w wybranych gatunkach gier

W rozdziale tym zostaną omówione konkretne typy gier w kontekście wykorzystania w nich systemów rozpoznawania mowy. Rozważone zostaną elementy interfejsu w kontekście możliwości ich przystosowania do obsługi przez system ASR. Przedstawione zostaną wady i zalety takich rozwiązań.

4.4.1 Gry akcji *FPP/TPP*

Gry akcji, w szczególności gry *FPP* (ang. *First Person Perspective*, z widokiem z perspektywy postaci w grze), to w tej chwili jeden z najbardziej popularnych gatunków gier. Za przykład weźmy gry z serii Call of Duty, w których gracz wciela się w żołnierza i patrzy na świat gry z jego perspektywy.

Złym przykładem zastosowania systemu rozpoznawania mowy w tej grze byłoby wyko-

rzystanie go do kontrolowania celowania, oddawania strzałów i poruszania się. Ze względu na konieczność szybkiej reakcji na zdarzenia zachodzące w grze, system ASR nie sprawdziłby się w tej roli - wydanie polecenia trwa długo, zaś liczba poleceń, które gracz chciałby wydać, byłaby zbyt duża. Sterowanie ruchami żołnieża oraz celowanie wymaga jednak bardziej precyzyjnych i szybszych kontrolerów, jak myszka i klawiatura lub pad.

Inną możliwością zastosowania systemu ASR jest użycie go do inicjowania czynności takich jak przeładowanie broni, zmiana broni, zmiana trybu strzału broni, aktywacja noktowizora i tym podobne. Tego typu czynności mają zwykle przypisany jeden klawisz i są wykonywane stosunkowo rzadko - czasem też dzieją się automatycznie. Sterowanie tego typu czynnościami z wykorzystaniem rozpoznawania mowy jest sensowniejsze niż poruszaniem się i celowaniem, lecz także nie byłoby cennym dodatkiem, a raczej utrudnieniem dla gracza.

Elementem sterowania gry, do którego system rozpoznawania mowy nadaje się idealnie, jest wydawanie poleceń innym postaciom w grze. Od poczatku istnienia gier FPP, w których po stronie gracza byli także inni żołnierze, sterowanie nimi - o ile w ogóle było możliwe - odbywało się poprzez wybranie na klawiaturze sekwencji klawiszy. Najpierw wybierało się do kogo będzie adresowana dana komenda, a następnie jaka ma ona być.

Przykładowo, w grze Freespace (symulator lotu w kosmosie, *FPP*), by wydać swojemu skrzydłu rozkaz obrony siebie, należało wcisnąć kolejno: c-2-1-8. Wpisanie tej sekwencji, zakładając, że znało się na pamięć wszystkie skróty, zajmowało ok. sekundy. Co ważniejsze, sprawiało, że trzeba oderwać oczy od monitora i spojrzeć na klawiaturę, by trafić w dane klawisze. Oczywiście można było robić to nie patrząc na klawiaturę, ale trafienie w klawisz 8 po naciśnięciu klawisza 1, wykorzystując lewą rękę, gdyż prawa leżała na myszce i kontrolowała lot, nie było proste. Co więcej, gdy trafiło się w klawisz 7 lub 9, wykonywana była zupełnie inna komenda. Gdy trafiło się przypadkiem w 0, całe skrzydło uciekało bezpowrotnie z pola walki i w praktyce należało powtórzyć misję.

Wykorzystanie systemu ASR w tej sytuacji pozwoliłoby nie tylko wydawać polecenia szybciej, ale także utrudniłoby wydanie innego polecenia (komendy *uciekaj* i *atakuj* brzmią zupełnie inaczej, prawdopodobieństwo pomyłki jest raczej niskie). Co więcej, gracz nie musiałby nawet na chwilę oderwać wzroku od monitora ani rąk od kontrolerów. Wykorzystanie systemu rozpoznawania mowy w ten sposób spełnia także postawiony wcześniej warunek naturalności. W prawdziwym świecie dowódcy żołnierzy czy pilotów także wydają sobie polecenia przy pomocy mowy. Znaczący wpływ na jakość oraz użyteczność systemu miałby proces projektowania zestawu komend, którego specyfika jest opisana w rozdziale 6.1

Wykorzystanie systemów rozpoznawania mowy w tego typu grach byłoby tylko dodat-

kiem, urozmaicającym i ułatwiającym nieco rozgrywkę. Ponieważ gracze współzawodniczą ze sobą nawzajem, wykorzystanie sterowania głosem mogłoby dać im przewagę nad przeciwnikami. Sądzę, że właśnie dlatego wielu graczy mogłoby się zdecydować na stosowanie interfejsu głosowego.

4.4.2 Gry *RPG*

Gry *RPG* (ang. '*Role Playing Games*') to gatunek dający moim zdaniem najszersze i najciekawsze możliwości zastosowania systemów rozpoznawania mowy. W ogólności w tego typu grach użytkownik przejmuje kontrolę nad pojedynczą postacią lub niewielką grupą postaci. Interfejs w postaci systemu ASR może być wykorzystany do pełnego lub częściowego sterowania postacią gracza.

W przypadku gier *MMORPG* (ang. '*Massive Multiplayer Online Role Playing Game*', wieloosobowe gry *RPG*), które są sieciową odmianą gier *RPG*, ilość specjalnych zdolności postaci jest często na tyle duża, że nie mieszczą się na przeznaczonym na nie miejscu na ekranie. Uruchamianie ich przy pomocy mowy mogłoby być wartościowym dodatkiem do interfejsu.

Najciekawszym wykorzystaniem systemów rozpoznawania mowy w grach wydaje mi się zastosowanie ich w grach dla dzieci lub młodzieży. Doskonałym przykładem są gry z serii *Harry Potter*, w których gracz wciela się w młodego czarodzieja. Dla tej gry możnaby przygotować system, który rozpoznawałby wypowiadane przez gracza zaklęcia. Byłoby to pod każdym względem zgodne z realiami świata gry i wydaje mi się naturalnym rozszerzeniem istniejącego interfejsu. W połączeniu z popularnymi ostatnimi czasy technologiami, jak *PlayStation Move* czy *X-Box Kinect*, w celu synchronizacji wypowiadanych zaklęć z ruchem ręką z różdżką, byłoby świetną rozrywką dla młodszych graczy.

W przyszłości, wraz z rozwojem sztucznej inteligencji, ilość możliwych opcji konwersacji w grach *RPG* będzie się zwiększała. Być może, za kilka lat, będzie można prowadzić rozmowę z postacią w grze nie na zasadzie wyboru spośród możliwych opcji, a poprzez prawdziwą wymianę zdań. System rozpoznawania mowy mógłby się sprawdzić w takiej sytuacji - nie tylko dlatego, że prowadzenie rozmowy w ten sposób byłoby naprawdę ciekawe i przyjemne, ale także dlatego, że prędkość pisania na klawiaturze większości graczy jest stosunkowo niska. Innymi słowy, dialog w grze prowadzony poprzez prawdziwą rozmowę trwałby krócej.

4.4.3 Gry strategiczne

W przypadku gier strategicznych możliwości wykorzystania systemów rozpoznawania mowy są moim zdaniem raczej niewielkie, ze względu na charakter rozgrywki. W grach strategicznych czasu rzeczywistego *RTS* (ang. *'Real Time Strategy'*) wydawanie komend głosem byłoby stanowczo zbyt czasochłonne. Gry tego typu wymagają od gracza często dziesiątek, nawet setek, akcji na minutę.

Z kolei w strategiach turowych, które są zwykle dużo bardziej skomplikowane od *RTS*, ilość możliwych opcji, działań czy ekranów jest na tyle duża, że stosowanie systemów ASR byłoby bardziej męczące niż użyteczne.

4.4.4 Inne typy gier

Najszerszą - i prawdopodobnie najciekawszą - grupą gier są te, które jeszcze nie zostały napisane. Projektując grę z myślą o systemie rozpoznawania mowy, można wejść w jeszcze niezbadane dziedziny. Przykładami tego typu gier mogłyby być wspomniane *Pilot rajdowy* lub *Regaty*, w których gracz wcielałby się odpowiednio w pilota rajdowego lub kapitana żaglówki.

Co ważniejsze, systemy rozpoznawania mowy mogą być z powodzeniem wykorzystane w grach edukacyjnych, rodzinnych lub przeznaczonych dla osób niepełnosprawnych ruchowo. Aplikacje do nauki lub doskonalenia języka mogłyby stosować system ASR do sprawdzenia postępów użytkownika w mówieniu. Gry w pełni kontrolowane głosem mogłyby być formą rozrywki dla osób, które nie mogą w tradycyjny sposób sterować komputerem.

Rozdział 5

Rozwiązania wykorzystane w grze *Pilot rajdowy*

W rozdziale tym opisane będą rozwiązania programistyczne wykorzystane w grze.

Gra *Pilot rajdowy* została zaimplementowana w języku Java przy wykorzystaniu środowiska Eclipse. Podstawą modułu rozpoznawania mowy był *CMU Sphinx*, wykorzystana została także biblioteka *Java Speech API*. Aplikacja jest wielowątkowa, z oddzielnie przetwarzaną grafiką, fizyką, rozpoznawaniem mowy i dodatkowym sterowaniem. Wykorzystane zostały mechanizmy grafiki 2D ze standardowej biblioteki Java 2D.

5.1 Fizyka

Fizyka w grze oparta jest o siły działające na samochód, przybliżony poprzez punkt o określonej masie i posiadający dwa wektory: kierunku oraz kierunku skrętu kół. Siły mogą oddziaływać na samochód na dwa sposoby: zmieniając jego prędkość (przyśpieszając lub zwalniając) oraz zmieniając jego kierunek. W moim modelu siły oddziałujące na samochód to siła tarcia kół, przyśpieszenie, hamulce oraz skręt kół. Wszystkie te siły mają na celu przybliżone modelowanie całkowitej siły działającej na samochód w rzeczywistości, czyli oddziaływania kół i podłoża. Opór powietrza oraz pozostałe czynniki zostały w modelu pominięte.

W każdej iteracji modelu dokonywane są obliczenia wszystkich sił, są one sumowane, a wypadkowa siła oddziałuje na samochód. Wartości liczbowe sił zależą od czasu, który upłynął od poprzedniej iteracji, zaś ich wpływ zależy także od masy samochodu. Wobec tego szybkość zdarzeń w grze jest niezależna od szybkości komputera, na którym jest ona uruchomiona. Im częstsze iteracje, tym model jest dokładniejszy - choć przy jego niskim stopniu skomplikowania różnice w zachowaniu samochodu powinny być niewielkie.

Model ten jest stosunkowo prosty, ale na potrzeby gry w zupełności wystarczający. Dzięki jego prostocie przewidywanie zachowania samochodu jest dużo łatwiejsze niż gdyby model dobrze odzwierciedlał rzeczywistość.

5.2 Integracja modułu rozpoznawania mowy z grą

Sphinx4 dostarcza prosty w wykorzystaniu interfejs do komunikacji z aplikacją. Obsługa urządzeń wejścia jest jego częścią, wobec czego użycie mikrofonu nie wymaga dodatkowej pracy. Zakładając brak ingerencji w działanie frameworku, cała integracja sprowadza się do zaimplementowania klasy przetwarzającej zwracany przez *Sphinksa* łańcuch znaków.

W *Pilocie rajdowym* założyłem, że to gracz kontroluje moment wydawania komendy poprzez naciśnięcie odpowiedniego klawisza. Niezbędne było wobec tego wprowadzenie zmian do *Sphinksa*, by takie działanie było możliwe. Zaimplementowałem klasę, która pełni rolę elementu przetwarzającego dane wejściowe. Jeśli klawisz odpowiadający za wydawanie komend nie jest naciśnięty, klasa ta sztucznie oznacza dane wejściowe jako ciszę.

Jednym z alternatywnych rozwiązań było włączanie i wyłączanie mikrofonu zgodnie ze stanem naciśnięcia klawisza. Wiązało się to jednakże z każdorazową koniecznością kalibracji poziomu wzmocnienia w innych klasach *Sphinksa*. Powodowało to problemy podczas procesu rozpoznawania mowy, w szczególności z pierwszym wypowiadanym wyrazem, który często uznawany był przez aplikację za ciszę.

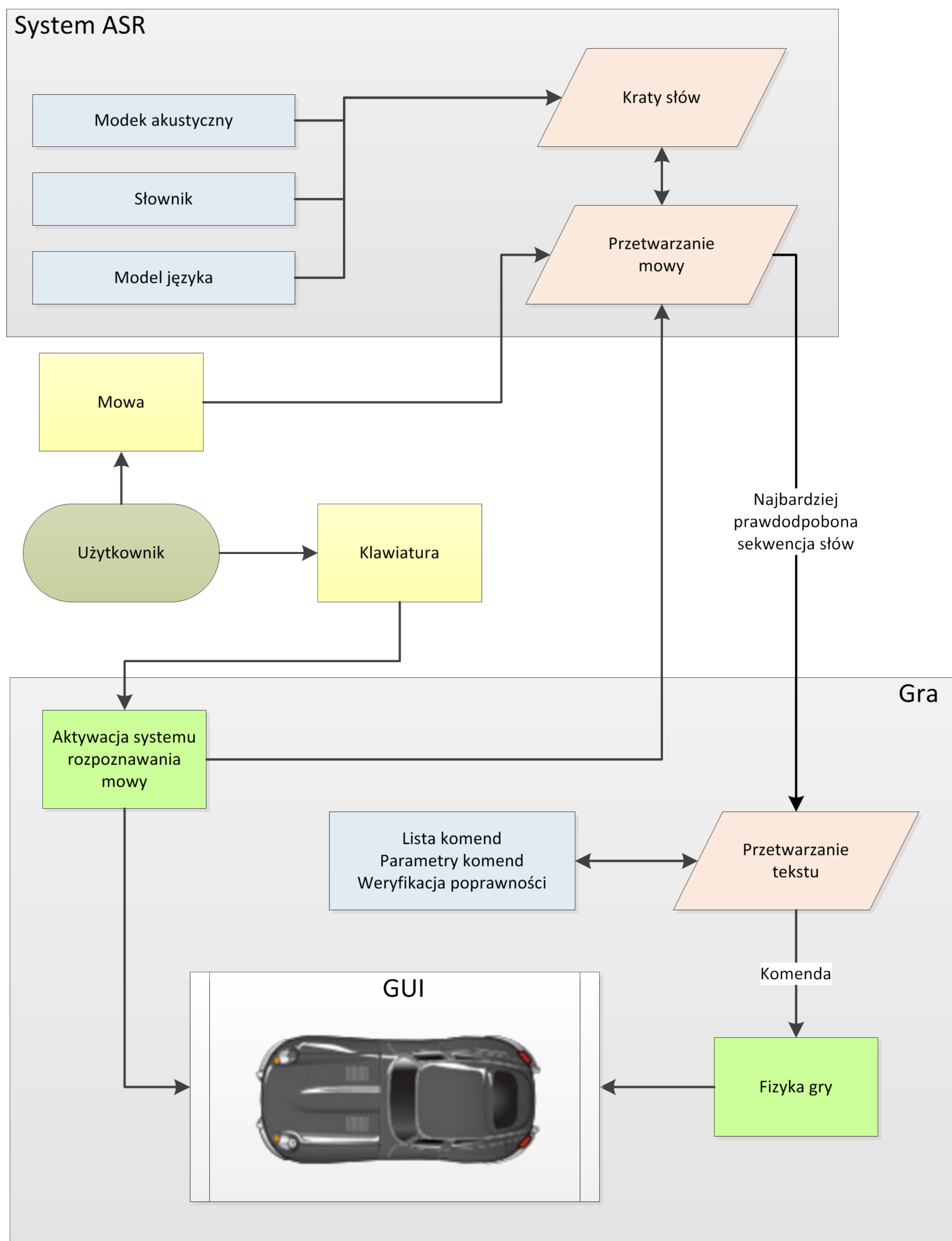
Schemat integracji aplikacji *Pilot rajdowy* z zystemem rozpoznawania mowy znajduje się na rysunku 5.1.

5.3 Przetwarzanie rozpoznanego tekstu

Wynikiem działania procesu rozpoznawania mowy, z punktu widzenia gry, jest ciąg znaków odpowiadający wypowiedzianym słowom. Słowa te mogą być *komendami*, czyli słowami decydującymi o podjętej akcji, *parametrami*, czyli słowami opisującymi wydaną komendę, lub być słowami nieważnymi dla interfejsu, jak np. *do*, *w*, *po*, etc.

Rozważając pojedynczą komendę, wynik rozpoznawania można podzielić na następujące kategorie:

- błędnie rozpoznana komenda, pasujące i poprawne parametry - kiedy słowo odpowiadające komendzie zostało przekłamane na inne, ale parametry zostały rozpoznane poprawnie i są one zgodne z przekłamaną komendą. Wówczas wykonana zostanie nieprawidłowa komenda,



Rysunek 5.1: Schemat integracji systemu rozpoznawania mowy z grą *Pilot rajdowy*

- błędnie rozpoznana komenda, niepasujące parametry lub ich brak - wówczas polecenie nie zostanie wykonane,
- poprawnie rozpoznana komenda, ale niepoprawnie rozpoznany jeden lub więcej parametrow - wówczas możliwe jest wykonanie polecenia, ale nie będzie ono wykonane w sposób, jaki sobie tego życzył gracz,
- poprawnie rozpoznana komenda, ale pominięty któryś z parametrów - w zależności od skladni komendy i wymagalności parametrów, może ona zostać wykonana niepoprawnie, albo nie zostać wykonana wcale,
- niepoprawnie rozpoznana komenda, ale poprawne parametry - jeśli rodzaj i wartość parametrów jednoznacznie implikuje wydaną komendę, może ona zostać poprawnie wykonana. W przeciwnym wypadku nie zostanie wykonana wcale,
- niepoprawnie rozpoznane lub pominięte któreś ze słów nieważnych - w pełni poprawne wykonanie komendy
- poprawnie rozpoznana komenda i wszystkie parametry - w pełni poprawne wykonanie komendy.

Wszystkie te sytuacje muszą być wzięte pod uwagę podczas implementacji modułu odpowiedzialnego za analizę semantyczną tekstu. Najgorszym możliwym przypadkiem jest przypadek pierwszy, ponieważ działanie w grze zupełnie różni się od intencji gracza. Niekiedy przypadek trzeci jest równie zły, gdyż parametr może diametralnie zmienić samą komendę. Gdyby gracz powiedział *skręć dziewięćdziesiąt stopni w prawo*, a system rozpoznał to jako *skręć siedemdziesiąt stopni w prawo*, pomyłka byłaby stosunkowo niewielka. Gdyby system, dla tej samej komendy, rozpoznał *skręć dziewięćdziesiąt stopni w lewo*, wówczas byłbty to poważny błąd. Z tej przyczyny bardzo ważne jest, by tak projektować składnię komend, by istotne parametry były od siebie możliwie różne z punktu widzenia fonetyki.

Warto zauważyć, że zdanie, które z punktu widzenia systemu rozpoznawania mowy zostało rozpoznane nieprawidłowo, z punktu widzenia gry może być w zupełności poprawne. To, co w systemie do dyktowania byłoby błędem, tutaj może być niezauważalne. Jest to kolejna zaleta stosowania systemów *command and control*, ponieważ kontekst wypowiedzi jest zawsze znany i możliwości korekty są dość duże.

Trzeba także zauważyć, że sposób implementacji modułu przetwarzania tekstu jest bardzo ważny z punktu widzenia całej gry. Zastosowanie mechanizmów korygujących nie w pełni poprawne wypowiedzi, mechanizmów wstrzymujących wykonanie poleceń, co do których istnieje duża szansa, że zostały przekłamane, etc.

5.4 Doszkalanie modelu akustycznego głosem gracza

Choć przygotowanie modelu akustycznego na podstawie danych audio od wielu osób daje dość dobre wyniki, przystosowanie modelu do głosu konkretnej osoby zawsze te wyniki poprawi. Dlatego tak ważny jest proces adaptacji modelu akustycznego do głosu gracza.

W przypadku gier komputerowych adaptacja modelu jest dużo prostsza, z technicznego punktu widzenia, niż np. w systemach telefonicznych. Aplikacja może wykorzystać moc obliczeniową komputera, na którym uruchomiona jest gra do przeprowadzenia niezbędnych obliczeń. Proces adaptacji istniejącego modelu jest dużo szybszy niż generowanie nowego modelu. Z danymi dźwiękowymi i transkrypcjami także nie będzie problemu, ponieważ mogą one być gromadzone podczas działania aplikacji - przykładowo poprzez zapisywanie zdań rozpoznanych z najwyższym prawdopodobieństwem poprawności. Można także, jeszcze przed rozpoczęciem właściwej rozgrywki, poprosić gracza o wypowiedzenie kilkunastu, kilkudziesięciu sekwencji testowych.

Nie jest to w grach komputerowych niespotykane - tutoriale różnego typu, uczące podstaw rozgrywki są w zasadzie standardem. A gracz i tak musi w jakiś sposób zapoznać się z komendami dostępnymi w grze. Kilkadziesiąt zgromadzonych w ten sposób zdań powinno wystarczyć do przeprowadzenia adaptacji. Jak pokażę w rozdziale 6.3.3, nawet bardzo niewielka ilość danych pozwala znacząco poprawić wyniki systemu.

Rozdział 6

Projektowanie i ewaluacja systemu ASR

Na system ASR do gry *Pilot rajdowy* składa się kilka elementów: zestaw komend, wynikający z niego model języka, model akustyczny oraz framework *Sphinx*. O ile komendy zostały zaprojektowane już na wstępnym etapie prac i podlegały niewielkim modyfikacjom, o tyle model języka i akustyczny były sukcesywnie rozwijane przez cały czas, równocześnie z przygotowaniem aplikacji demonstracyjnej i modyfikacjami systemu *Sphinx*.

W poprzednich rozdziałach opisane zostały teoretyczne podstawy projektowania systemu rozpoznawania mowy. W tym opisana zostanie praktyka projektowania interfejsu głosowego do gry *Pilot rajdowy*.

6.1 Opracowanie zestawu komend

Komendy systemu opracowywane były wyłącznie na potrzeby interfejsu użytkownika. Ich wpływ na potencjalną poprawność rozpoznawania nie był rozważany, tzn. wykorzystywane słowa nie były analizowane pod względem wzajemnego podobieństwa i wynikającym z niego większym prawdopodobieństwie błędu. Podczas projektowania komend korzystałem z dostępnych w Internecie materiałów dotyczących rajdów samochodowych, starając się zachować pewne podobieństwo komend z interfejsu do rzeczywistych zwrotów wykorzystywanych przez pilotów rajdowych. Szczególnie pomocna była strona Macieja Handwerkera [26], zawierająca wiele ciekawych informacji na temat zawodu pilota rajdowego.

Sterowanie samochodem w *Pilocie rajdowym* odbywa się przy pomocy następujących komend podstawowych: *przyśpiesz, zwolnij, wrzuć, zredukuj, skręć, zakręt, prosta. Przyśpiesz* i *zwolnij* odpowiadają za sterowanie prędkością. *Wrzuć* i *zredukuj* kontrolują sterowanie biegami. *Skręć* i *zakręt* powodują zmianę kierunku jazdy. Polecenia są wykonywa-

ne natychmiast po rozpoznaniu. Polecenie *prosta* może być wykorzystane do opóźnienia wykonania wydanego po nim polecenia skrętu do momentu przebycia przez samochód określonego dystansu w linii prostej. Każde z opisanych poleceń ma swoje wymagania dotyczące ilości i rodzaju parametrów po nich występujących.

Oprócz tego możliwe jest wydanie polecenia wykonania sekwencji *zakręt-prosta-zakręt* w postaci jednego zdania, np. *zakręt w lewo trzydzieści stopni po trzydziestu metrach przechodzi w zakręt w prawo sześćdziesiąt stopni.* Komendy te zostaną wykonane jedna po drugiej.

W późniejszej fazie prac nad systemem dodane zostały nowe komendy. Przyczyną rozszerzenia zestawu komend były problemy osób niezapoznanych ze sterowaniem w grze. Problem ten został dokładnie opisany w artukule dotyczącym problemu słownictwa w komunikacji między człowiekiem a maszyną [27]. Osoby nieznające systemu wydawały polecenia, które pasują do sterowania samochodem, lecz nie zostały uwzględnione w systemie. Dodane na tym etapie polecenia to: *ruszaj, rusz, jedź, hamuj, stop, stój, zatrzymaj, cofnij, cofaj.* Nie wymagają one żadnych parametrów. Pełna lista komend wraz z ich parametrami znajdje się w dodatku A.

6.2 Testy systemu

Możliwość ewaluacji jakości systemu rozpoznawania mowy jest bardzo ważna podczas jego tworzenia. Ważne jest, by testy były miarodajne, by dobrze odzwierciedlały rzeczywiste warunki pracy systemu, wliczając w to zdania testowe, sprzęt i warunki panujące podczas nagrywania. Dzięki dobrym testom możliwa jest jednoznaczna weryfikacja wprowadzonych zmian w każdym elemencie systemu i na każdym etapie jego tworzenia.

Listing 6.1: Przykład wyniku wykonania testu systemu

```
# --------------- Summary statistics ---------
  Accuracy: 99,289%    Errors: 4  (Sub: 2  Ins: 1  Del: 1)
  Words: 422   Matches: 419    WER: 0,948%
  Sentences: 85   Matches: 82   SentenceAcc: 96,471%
  Total Time Audio: 222,27s  Proc: 13,76s  Speed: 0,06 X real time
```

Przygotowanie testów było jednym z pierwszych etapów prac nad systemem. Pierwszym etapem było spisanie sekwencji testowych - 85. zdań o różnej długości opartych na komendach z systemu. Następnie sekwencje te zostały przeze mnie nagrane - w trakcie dalszych prac te same sekwencje zostały nagrane także przez inne osoby, by móc badać działanie systemu na kilku mówcach. Pełna lista zdań testowych znajduje się w dodatku B.

Tabela 6.1: Dane zwracane przez testy systemu ASR w *Sphinksie*

pole	znaczenie
Accuracy	Dokładność
Errors	Całkowita liczba błędów
Sub	Liczba zastąpień
Ins	Liczba wtrąceń
Del	Liczba usunięć
Words	Całkowita liczba słów
Matches	Liczba poprawnie rozpoznanych słów
WER	Word Error Rate, słowna stopa błędów
Sentences	Liczba sekwencji testowych
Matches	Liczba poprawnie rozpoznanych zdań
SentenceAcc	Dokładność zdaniowa
Total Time Audio	Całkowita długość danych testowych
Proc	Czas wykonywania testu
Speed	Stosunek czasu wykonywania testu do długości danych testowych

Sphinx zapewnia możliwość ustawienia źródła mowy na plik dźwiękowy, posiada także zestaw klas, które można wykorzystywać do przeprowadzenia testów. Przygotowane zdania i pliki audio zostały zintegrowane z modułem testowym *Sphinksa*.

Wynikiem przeprowadzenia testu są wartości metryk (opisanych w następnym rozdziale, 6.2.1). *Sphinx* podaje także dane podstawowe, na podstawie których te metryki są wyliczane, oraz informacje o danych testowych i samym teście. Zwracane informacje są opisane w tabeli 6.1.

6.2.1 Stosowane metryki

Spośród danych wyjściowych testów najbardziej i najdokładniej świadczą o jakości systemu *Accuracy*, czyli dokładność, *WER*, słowna stopa błędów, oraz *Sentence accuracy*, czyli dokładność zdaniowa.

Dokładność obliczana jest wg wzoru

$$accuracy = \frac{N - D - S}{N} \cdot 100\% \tag{6.1}$$

gdzie N - liczba słów, D - liczba usunięć, S - liczba zastąpień.

Metryka ta mówi o procencie poprawnie rozpoznanych słów. Trzeba zauważyć, że nie

uwzględnia ona wtrąceń. Dlatego nieco inne wartości daje metryka słownej stopy błędów, liczona następująco:

$$WER = \frac{I + D + S}{N} \cdot 100\% \tag{6.2}$$

gdzie N - liczba słów, I - liczba wtrąceń, D - liczba usunięć, S - liczba zastąpień.

WER mówi, jaki procent słów został rozpoznany błędnie, czy to poprzez przekłamanie, usunięcie czy wtrącenie. Ostatnią analizowaną metryką jest dokładność zdaniowa. Obliczana jest ona następująco:

$$SentenceAcc = \frac{S - S_e}{N} \cdot 100\% \tag{6.3}$$

gdzie S to liczba zdań testowych, zaś S_e to liczba zdań, w których popełniony został co najmniej jeden błąd.

Metryka ta jest najbardziej restrykcyjna i jej wysoka wartość najlepiej świadczy o jakości systemu.

6.3 Model akustyczny

Model akustyczny jest najważniejszym elementem łączącym sygnał dźwiękowy i jednostki akustyczne, czyli fonemy. Jego jakość najsilniej wpływa na dokładność systemu. Podczas prac nad grą rozważane były dwa modele - angielski model WSJ oraz polski, przygotowany specjalnie na potrzeby systemu.

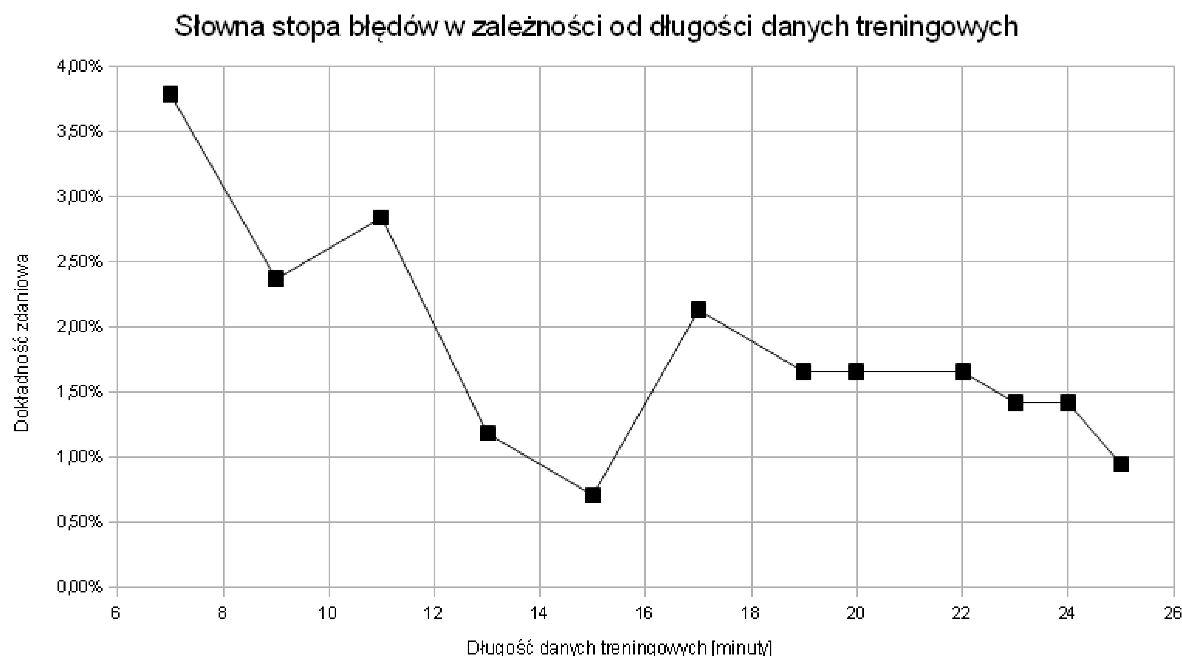
6.3.1 Model angielski

W *Sphinksie* dostępne są dwa modele akustyczne: WSJ, oparty na danych audio z dziennika *Wall Street Journal*, oraz *TIDIGITS* [28], oparty na danych zebranych od 326 mówców w firmie *Texas Instruments*. Oba te modele są modelami dla języka angielskiego.

Listing 6.2: Wyniki pierwszego testu systemu

```
# --------------- Summary statistics ---------
   Accuracy: 36,493%    Errors: 272  (Sub: 111  Ins: 4  Del: 157)
   Words: 422   Matches: 154    WER: 64,455%
   Sentences: 85   Matches: 10   SentenceAcc: 11,765%
   Total Time Audio: 222,27s  Proc: 29,34s  Speed: 0,13 X real time
```

W pierwszych wersjach systemu ASR dla *Pilota Rajdowego* wykorzystany został model WSJ. Choć język angielski posiada inny zestaw fonemów niż język polski, duża ich część



Rysunek 6.1: Zależność słownej stopy błędów od długości danych treningowych

jest wspólna, a część polskich fonemów można przybliżyć podobnymi fonemami angielskmi. Na listingu 6.2 widoczne są wyniki pierwszego kompletnego systemu wykorzystującego model WSJ. Jak widać, dokładność jest bardzo niska. Tylko nieco ponad 10% zdań zostało poprawnie rozpoznanych.

Nawet, gdyby udało się dwukrotnie poprawić wyniki, np. poprzez modyfikację modelu języka, i tak byłyby one niezadowalające. Przy rezultatach na tak niskim poziomie konieczne okazało się przygotowanie modelu akustycznego dla języka polskiego.

6.3.2 Model polski

Pierwszym etapem oprawcowywania modelu był wybór i przygotowanie danych treningowych, co zostało opisane w rozdziale 2.4.2. Dane do treningu modelu przygotowywane były partiami, na każdym etapie przeprowadzane były testy. Model był przygotowany przeze mnie i trenowany moim głosem. Sekwencje testowe także były nagrane przeze mnie. Zależność słownej stopy błędów (WER) od długości danych treningowych przedstawiona jest na wykresie 6.1.

Pierwsze 6 minut danych (pierwszy punkt na wykresie 6.1 i pierwszy wiersz w tabeli 6.2) to 114 zdań *corpora*. Punkt 11 minut to drugi zestaw nagrań zdań *corpora*, tym razem mówionych nieco szybciej. Widzimy więc, że już dla tak małej ilości danych model akustyczny dedykowany dla pojedynczego mówcy daje dobre wyniki. Dla 25 minut otrzymujemy poprawność zdaniową powyżej 95%. Jest to wynik, który powinien pozwolić na

Tabela 6.2: Wyniki modeli akustycznych w zależności od długości danych treningowych

czas audio [m]	Dokładność	WER	Dokładność zdaniowa
6	96,2%	3,8%	88,2%
11	97,2%	2,8%	90,6%
17	98,1%	2,1%	92,9%
22	98,6%	1,7%	94,1%
25	99,3%	0,9%	96,5%

Tabela 6.3: Wyniki modeli języka: n-gram i JSGF

model języka	mówca	Dokładność	WER	Dokładność zdaniowa
pierwszy	A	90,8%	11,4%	68,2%
n-gram	B	88,7%	12,1%	69,4%
n-gram bez	A	97,2%	5,0%	81,2%
negatywnych	B	96,5%	3,5%	83,6%
ostateczny	A	97,4%	2,8%	90,6%
n-gram	B	97,2%	2,8%	87,1%
gramatyka	A	76,5%	27,5%	61,2%
JSGF	B	70,0%	33,1%	38,8%

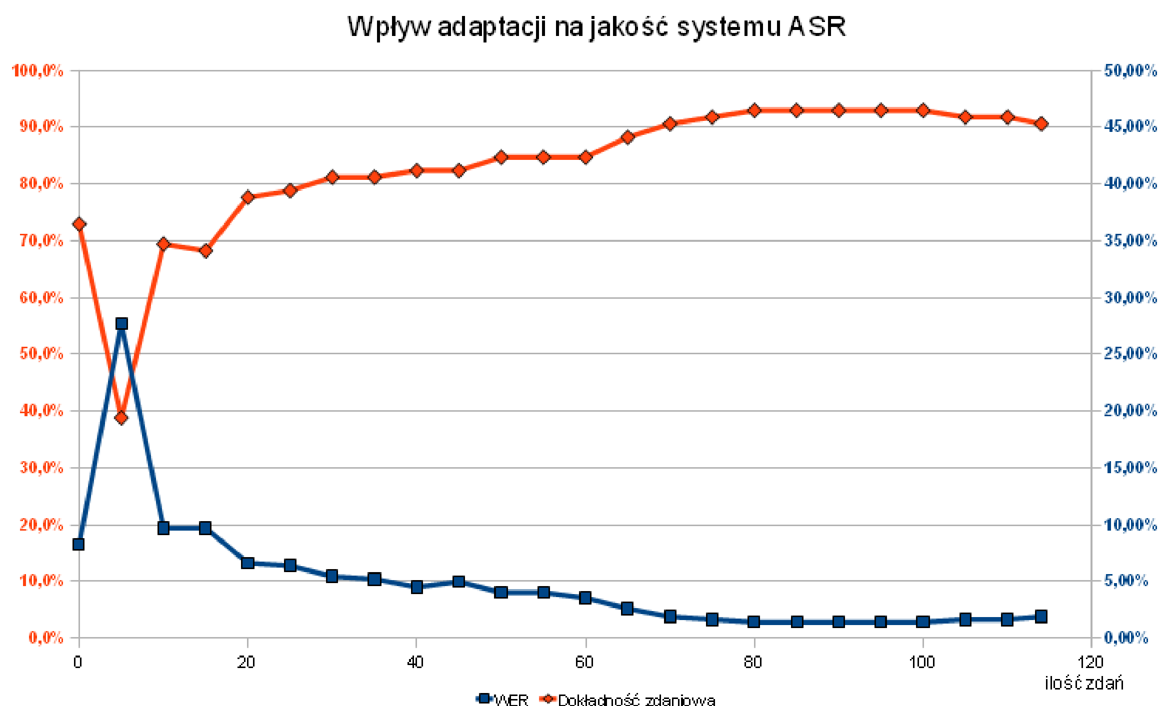
swobodną grę w *Pilota rajdowego*.

Podczas dalszej pracy wygenerowałem model oparty na 45 zestawach zdań *corpora*. Pochodziły one od ludzi różnej płci i w różnym wieku. Wyniki dla tego modelu akustycznego, dla różnych modeli języka, znajdują się w tabeli 6.3. Szczegółowe omówienie różnych modeli języka znajduje się w rozdziale 6.4. Nawet bez adaptacji daje on bardzo dobre wyniki dla obu testowych mówców.

6.3.3 Adaptacja

Adaptacja modelu akustycznego to proces, który polega na modyfikacji istniejącego modelu na podstawie pewnej ilości danych tak, by model ten lepiej odpowiadał tym danym. Powstały w ten sposób model jest w większym stopniu przystosowany do jednego mówcy, którego głosem był adaptowany. Zaletą adaptacji jest także jej szybkość, która jest duża w porównaniu do procesu generowania nowego modelu, oraz fakt, że stosunkowo niewiele danych jest potrzebnych do przeprowadzenia skutecznej adaptacji.

Pierwsza próba adaptacji modeli *Sphinksa* odbyła się już na etapie prac z modelem WSJ. Zaadaptowałem wówczas model WSJ zdaniami testowymi, tymi samymi, które wy-



Rysunek 6.2: Wpływ adaptacji na jakość systemu

korzystuję do badania jakości systemu. Te same dane, które są wykorzystywane do testów, nie powinny być wykorzystywane do adaptacji czy generowania modelu. Z założenia jednak, ta pierwsza próba adaptacji miała zweryfikować sens dalszych prób wykorzystania modelu WSJ.

Po zaadaptowaniu modelu okazało się, że poprawa jakości systemu jest, choć wyraźnie widoczna, niewielka. WER zmalał o ok. 5%, o ok. 5% zwiększyła się dokładność, zaś dokładność zdaniowa zwiększyła się o niecałe 2%. Tak słabe wyniki przesądziły o decyzji wygenerowania polskiego modelu akustycznego.

Adaptacja modelu polskiego miała inne przyczyny. Po przygotowaniu modelu na podstawie własnego głosu, chciałem sprawdzić jak będzie się on zachowywał przy adaptacji głosem innej osoby. Inny mówca, także mężczyzna, w podobnym wieku jak ja, przygotował dla mnie dwa zestawy nagrań zdań *corpora* oraz sekwencje testowe. Nagrania *corpora* były używane do adaptacji modelu, zaś nagrania testowe do ewaluacji wyników systemu.

Rezultaty testów dla różnej ilości zdań wykorzystanych do adaptacji modelu widoczne są na wykresie 6.2. Przy 40 zdaniach dokładność zdaniowa podniosła się o 10%, przy 80 - o 20%. Jest to bardzo duża poprawa, mogąca, w przypadku gry, decydować o jej porzuceniu lub dalszej grze.

Warto także zauważyć, że powyżej 80 zdań dokładność przestaje rosnąć, a nawet, choć bardzo delikatnie, spada. Można z tego wyciągnąć dwa wnioski: po pierwsze, adaptacja

może podnieść wyniki systemu tylko w ograniczonym zakresie i nie pozwala osiągnąć tak dobrych wyników jak dedykowany model. Po drugie, zbyt duża ilość danych do adaptacji może spowodować pogarszanie się wyników w stosunku do mniejszej ilości danych. Przyczyny tego efektu nie zostały przeze mnie zbadane, ale można się spodziewać, że adaptacja zachowuje się podobnie jak generowanie modelu. I tak, jak możliwe jest przetrenowanie modelu, tak możliwa jest nadmierna adaptacja. W obu przypadkach model traci swą elastyczność i daje gorsze wyniki.

6.4 Modele języka

Dwa modele języka były rozważane i testowane podczas prac nad systemem - statystyczny model n-gram, opisany w rozdziale 2.7.1, oraz gramatyka JSGF, opisana w rozdziale 2.7.3. W rozdziale tym przedstawione oraz omówione zostaną wyniki obu tych modeli.

6.4.1 N-gram

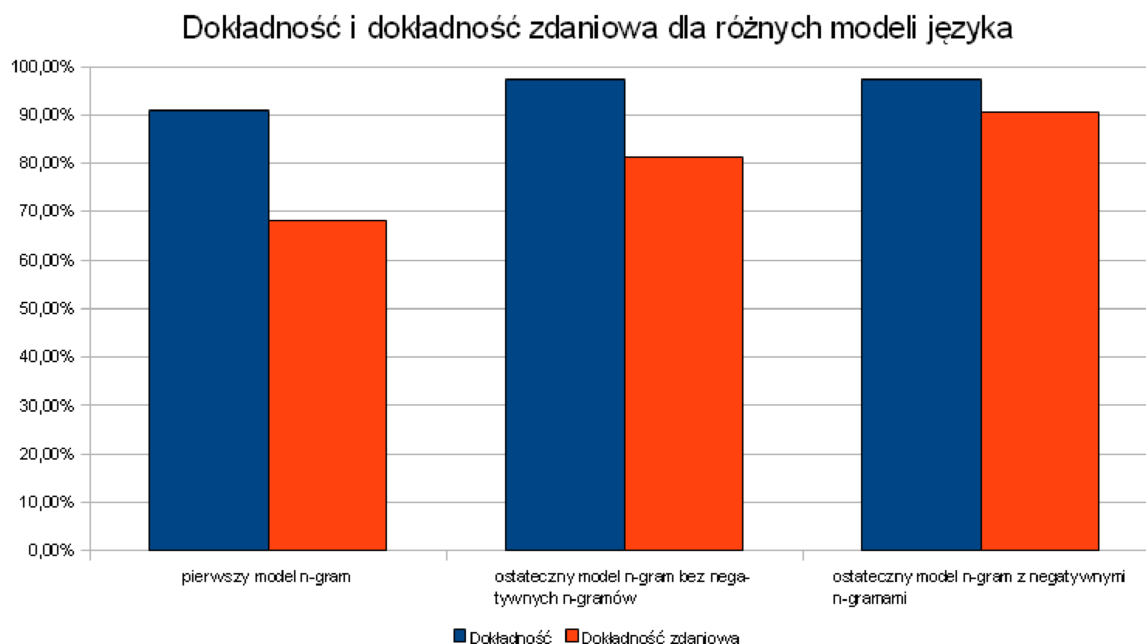
Pierwszy model n-gram do *Pilota rajdowego*, był modelem wygenerowanym przy pomocy prostego programu napisanego w Javie z arbitralnie przydzielonymi prawdopodobieństwami. W połączeniu z wykorzystywanym wówczas modelem akustycznym WSJ (prace te opisane były w rozdziale 6.3.1) system dawał dokładność zdaniową rzędu 10%, kompletne wyniki pierwszego testu znajdują się na listingu 6.2.

Przygotowanie modelu języka przy pomocy prostego skryptu z arbitralnymi prawdopodobieństwami było błędem. Przewidzenie częstotliwości występowania określonych n-gramów jest praktycznie niemożliwe, a przypisywanie wszystkim tych samych wartości jest bez sensu.

Skoro model n-gram jest modelem statystycznym, spróbowałem innego podejścia. Przygotowałem skrypt, który wygenerował kilkusetkilobajtowy plik zawierający wszystkie możliwe komendy, ze wszystkimi kombinacjami dostępnych dla nich parametów, występujące w grze. Im więcej różnych parametrów miała dana komenda, tym więcej zdań zostało wygenerowanych. Wobec tego zwielokrotniłem zdania dla komend, które miały mniej kombinacji parametrów, tak, by liczba zdań dla każdej z komend była podobna. W ten sposób przygotowaną bazę zdaniową wykorzystałem jako plik wejściowy dla aplikacji lmtool, która generuje model n-gram na podstawie właśnie takiej bazy.

Wygenerowany w ten sposób model, z niewielkimi poprawkami w postaci negatywnych n-gramów, został wykorzystany w ostatecznej wersji systemu dla *Pilota rajdowego*.

Zestawienie wyników różnych modeli języka znajduje się w tabeli 6.3. Wszystkie te modele zostały przetestowane dla tego samego modelu akustycznego, opartego na bazie



Rysunek 6.3: Zależność dokładności i dokładności zdaniowej od zastosowanego modelu n-gram

corpora.

Negatywne n-gramy dodawane były w ostatnim etapie przygotowania systemu. Dzięki testom, zarówno automatycznym jak przeprowadzanym ręcznie, mogłem zaobserwować błędy najczęściej popełniane przez system, a następnie przygotować odpowiedni n-gram negatywny, który dany błąd wyeliminuje. Zastosowanie tego rozwiązania w nieznacznym stopniu zwiększa dokładność, ale ma duży wpływ na dokładność zdaniową. Przy dokładności rzędu 95% na zdanie przypada zwykle nie więcej niż jeden błąd - więc jeśli uda się zmniejszyć ich liczbę choćby o kilka, wówczas liczba poprawnie rozpoznawanych zdań wyraźnie rośnie. Wzrost ten obrazuje rys. 6.3

6.4.2 JSGF

Gramatyka JSGF wydaje się być dobrze przystosowana do opisywania systemów *command and control*. Sądziłem, że, o ile przygotuję dostatecznie dużo alternatywnych sposobów formułowania komend, systemowi uda oię osiągnąć bardzo dobre wyniki, być może nawet lepsze niż dla modelu n-gram.

Wyniki najlepszej wersji gramatyki JSGF widoczne są w tabeli 6.3. Są one dużo gorsze niż wyniki osiągnięte przez model n-gram. Są nawet gorsze niż wyniki pierwszego, bardzo prymitywnego, modelu n-gram.

Analizując dokładnie zdania testowe zauważyłem, ze niektóre błędy pasują do pewnego schematu, związanego ze strukturą i składnią samej gramatyki. Mianowicie, jeżeli system napotkał kompletne zdanie zdefiniowane w gramatyce, praktycznie zawsze kończył rozpoznawanie danego zdania, nawet, jeśli istniała dalsza jego część. Przykładowo, dla zdania *Skręć w lewo dwadzieścia stopni po trzydziestu metrach skręć w lewo czterdzieści stopni* system oparty o JSGF zwykle rozpoznawał *Skręć w lewo dwadzieścia stopni*, zupełnie pomijając pozostałą część zdania. Powodowało to znaczący spadek wartości metryk dokładności i WER, ponieważ takie zachowanie systemu to kilka-kilkanaście usunięć.

Próbowałem zbadać przyczyny takiego zachowania poprzez analizę kodu, próbowałem zmieniać kolejność definicji w gramatyce, próbowałem różnych wariantów definiowania tego samego. Bez rezultatów.

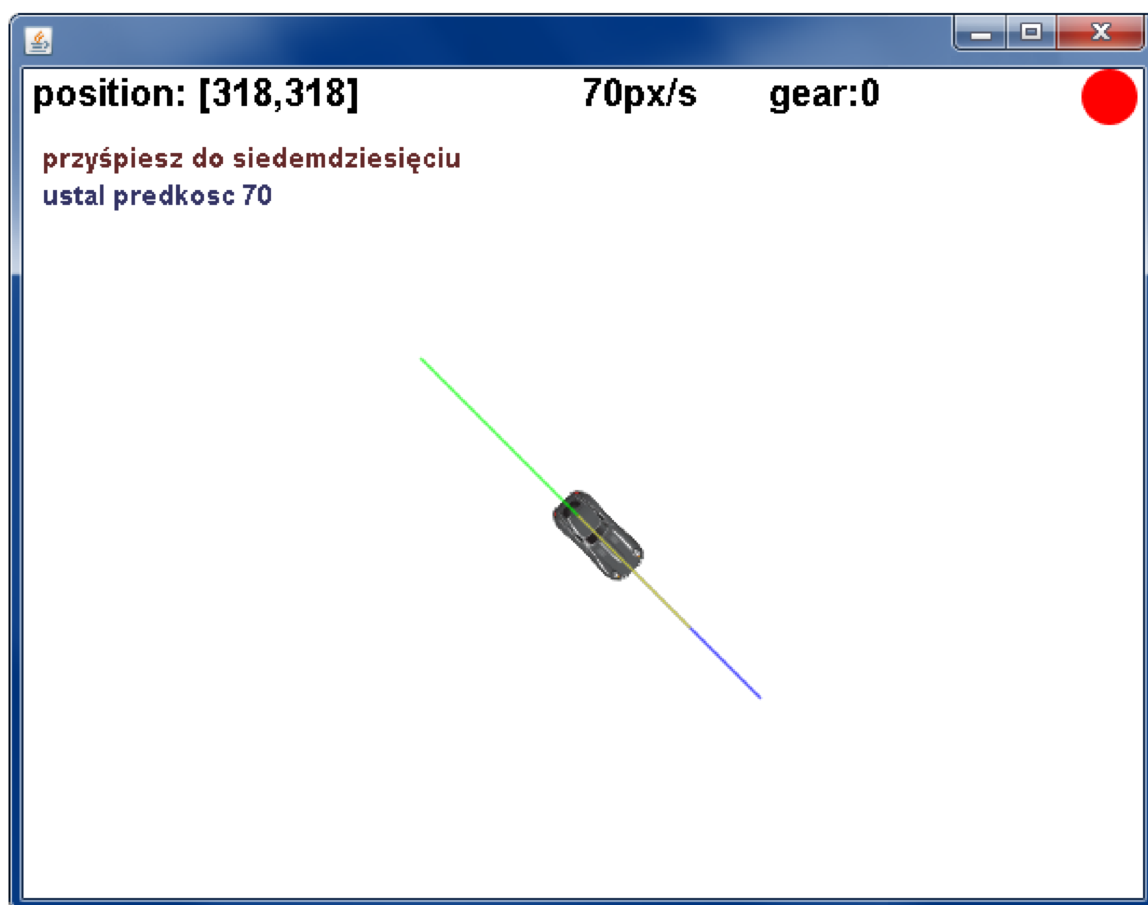
Podsumowując, gramatyka JSGF nie sprawdziła się w systemie ASR dla *Pilota rajdowego*. Niepoprawne i niezrozumiałe zachowanie systemu wykluczyło użycie JSGF w moim projekcie.

6.5 Prototyp gry

Przygotowany system został zintegrowany z aplikacją, która pozwala na sterowanie samochodem. Nie ma ona jeszcze charakteru gry, pozwala za to w pełni przyjrzeć się możliwościom systemu rozpoznawania mowy.

Zaimplementowany prototyp gry widoczny jest na zrzucie ekranu 6.4. Na górze okna gry wyświetlane są informacje na temat stanu samochodu (prędkość, bieg, położenie). Na prawo od nich znajduje się znacznik przyjmowania komend - jeśli jest zielony, to znaczy, że system przyjmuje polecenie, co w aktalnej wersji sprowadza się do tego czy spacja została naciśnięta. Pod informacjami o pozycji znajdują się dwie linie informacyjne dotyczące przyjętych komend. Pierwsza z nich, o kolorze brązowym, to tekst rozpoznany przez system ASR. Druga linia to komenda po przetworzeniu przez grę.

Widoczny na zrzucie samochód porusza się zgodnie z wydawanymi instrukcjami. Linie do niego przyczepione to wektory: zółty - orientacji samochodu, niebieski - kierunku poruszania się środka masy, czerwony - przyśpieszenia a zielony - hamowania(sił spowalniających samochód). Samochód jedzie prosto, więc wektor przyśpieszenia nie jest widoczny, ponieważ znajduje się pod wektorem kierunku.



Rysunek 6.4: Widok okna prototypu gry *Pilot rajdowy*

Rozdział 7

Podsumowanie

W pracy tej opisany został proces przygotowania systemu rozpoznawania mowy przystosowanego do wykorzystania w grze komputerowej. Przygotowany system został przetestowany, przeprowadzone zostały testy różnych elementów systemu; opisany został ich wpływ na jakość całego systemu. Przedstawione zostały także informacje teoretyczne na temat systemów ASR - z czego się one składają i jak działają. Do tego omówione zostały problemy pojawiające się w zastosowaniu rozpoznawania mowy w grach, rozważane były także możliwości zastosowania tego tyu systemów w różnego typu grach.

Część wyników z tej pracy magisterskiej zostało zaprezentowanych na konferencji FedCSIS 2011 w Szczecinie oraz opublikowanych w artykule [29] napisanym na jej potrzeby.

Na Ogólnopolskiej Konferencji Inżynierii Gier Komputerowych wygłoszony został przeze mnie referat na temat możliwości wykorzystania systemów rozpoznawania mowy w grach komputerowych [30].

Podstawowa wiedza dotycząca systemów ASR zebrana została przeze mnie w postaci artykułu opublikowanego w Software Developer's Journal [3].

7.1 Jakość systemu

Przeprowadzone testy pokazują bardzo dobre wyniki utworzonego systemu. Można z nich wysnuć wniosek, że stosunkowo niewielkim kosztem czasu i pracy (w stosunku do projektów komercyjnych), możliwe jest przygotowanie systemu *command and control* o małym słowniku o bardzo dobrej dokładności.

Zaskakująca jest też niewielka ilość danych potrzebna do osiągnięcia dobrych rezultatów. Oczywiście naprawdę dobre modele akustyczne potrzebują dziesiątek - nawet setek - godzin nagrań. Lecz tak dobre modele wymagane są dla systemów dyktacyjnych, nie są

konieczne dla systemów *command and control*, co zostało w tej pracy udowodnione.

Nieduża jest też ilość danych potrzebna do adaptacji modelu akustycznego. Kilkadziesiąt zdań to zwykle kilka minut danych audio. Możliwość łatwej adaptacji jeszcze bardziej rozszerza możliwości i grupę docelową tego typu systemów.

7.2 Możliwości komercyjnego wykorzystania

Wykorzystanie systemów rozpoznawania mowy dedykowanych dla konkretnych aplikacji komputerowych, przy podobnie ograniczonym słownictwie i składni jak system z *Pilota rajdowego*, jest niedocenionym - w zasadzie niewykorzystanym komercyjnie - pomysłem. Tego typu rozszerzenie możliwości aplikacji mogłoby być cennym dodatkiem, ułatwiającym i przyśpieszającym pracę.

Choć nowoczesne systemy operacyjne często posiadają funkcję wykorzystania rozpoznawania mowy, to, moim zdaniem, błędne jest do niej podejście. Projektanci systemów operacyjnych zdają się skupiać na dobrych systemach dyktowania, zamiast na uprzyjemnieniu i przyśpieszeniu pracy użytkownika. Projektanci aplikacji komputerowych praktycznie w ogóle nie rozważają systemów rozpoznawania mowy jako elementów interfejsu. A właśnie w takich, ściśle zdefiniowanych i ograniczonych zastosowaniach systemy ASR sprawdzają się najlepiej.

7.3 Dalszy rozwój

Pilot rajdowy jest w momencie składania tej pracy jedynie aplikacją typu *proof of concept*. By był pełnoprawną grą, niezbędne jest jeszcze wiele pracy programistycznej oraz opracowanie grafiki i dźwięku.

Jeśli chodzi o sam system - można go dalej ulepszać. Możliwa jest poprawa modelu akustycznego - zgromadzenie większej ilości danych. Pełna automatyzacja procesu adaptacji modelu i integracja go z grą to kolejny element do wykonania. W trakcie prac progamistycznych może się okazać konieczne wprowadzenie nowych komend, lub modyfikacja istniejących - będzie to musiało znaleźć odzwierciedlenie w modelu języka.

Rozszerzenie testów, zarówno o ilość zdań, jak i o ilość mówców, dałoby dokładniejsze informacje o jego jakości. Liczba zestawów zdań testowych powinna wzrosnąć, mówcy zaś powinni być bardziej zróżnicowani. Powinny powstać zestawy z głosami żeńskimi, być może z głosami dziecięcymi, głosami osób starszych czy głosami osób z wadami wymowy.

Rozważam także możliwość zdefiniowania własnej gramatyki podobnej do JSGF, oraz zaimplementowania elementów *Sphinksa* ją obsługujących. Niezrozumiałe zachowanie sys-

temu z gramatyką JSGF testowanego w czasie pracy, oraz jego słabe wyniki, nie oznacza, że sama koncepcja jest błędna. Być może możliwe jest opracowanie składni gramatyki łączącej elastyczność modeli statystycznych i jednoznaczność sztywnych gramatyk.

Bibliografia

[1] Bartosz Ziółko, Suresh Manandhar, Richard C. Wilson, Mariusz Ziółko, and J. Gałka. Application of HTK to the Polish language. In *Proceedings of IEEE International Conference on Audio, Language and Image Processing, Shanghai*, 2008. http://www-users.cs.york.ac.uk/~suresh/papers/HTKPOLISH.pdf.

[2] Szymański Marcin, Jerzy Ogórkiewicz, Marek Lange, Katarzyna Klessa, Stefan Grocholewski, and Grażyna Demenko. First evaluation of Polish LVCSR acoustic models obtained from the jurisdic database. *Speech and Language Technology*, 11, 2008. publisher: Polish Phonetic Association.

[3] Dariusz Wawer. Systemy rozpoznawania mowy. *Software Developer's Journal*, pages 60–66, 1 2011.

[4] L. Rabiner and B.-H. Juang. Historical perspective of the field of ASR/NLU. In Yiteng Huang, editor, *Springer Handbook of Speech Processing*, Lecture Notes in Computer Science, page 522. Springer, 2008.

[5] Stefan Grocholewski. Corpora - speech database for Polish diphones. *EUROSPEECH '97*, (5):1735–1738, 9 1997.

[6] D. Gibbon, R. Moore, and R. Winski (red.). *Handbook of Standards and Resources for Spoken Language Systems.* Mouton de Gruyter, 1997.

[7] Ron F. Feldstein. *A Concise Polish Grammar.* Slavic and Eurasian Language Resource Center, 2001.

[8] Steve Young. *HMMs and related speech recognition technologies*, pages 539–558. Springer, 2008.

[9] Slava M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, volume 35, pages 400–401, 1987.

[10] Karel Oliva, Pavel Kveton, and Roman Ondruska. The computational complexity of rule-based part-of-speech tagging. In Vaclav Matousek and Pavel Mautner, editors, *Text, Speech and Dialogue*, volume 2807 of *Lecture Notes in Computer Science*, pages 82–89. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-39398-6-12.

[11] Sun Microsystems. Java speech grammar format. version 1, 1998, http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/.

[12] A. Acero and J. Droppo. Robustness in speech recognition. In J. Benesty, editor, *Springer Handbook of Speech Processing and Speech Communication*. Springer, Berlin, Heidelberg, 2007.

[13] Jun Hou, Lawrence Rabiner, and Sorin Dusan. Automatic speech attribute transcription (asat) – the front end processor. In *icassp*, volume 1, pages 333–336, 2006.

[14] Stavros Tsakalidis, Vlasios Doumpiotis, and William Byrne. Discriminative linear transforms for feature normalization and speaker adaptation in HMM estimation. In *in Proc. ICSLP*, pages 2585–2588, 2003.

[15] Kai Yu and M. J. F. Gales. Discriminative cluster adaptive training. *IEEE Transactions On Audio Speech And Language Processing*, 14(5):1694–1703, 2006.

[16] Cmu sphinx. http://cmusphinx.sourceforge.net/.

[17] Hidden markov model toolkit. http://htk.eng.cam.ac.uk/.

[18] Microsoft speech api. http://www.microsoft.com/en-us/tellme/.

[19] Oracle. Java speech api. http://java.sun.com/products/java-media/speech/.

[20] Unikkon Integral Sp. z o.o. Magicscribe. http://www.magicscribe.pl/.

[21] Prime Speech. Portal głosowy zarzątu transportu miejskiego w warszawie. http://www.primespeech.pl/ZTM_case.pdf.

[22] Ubisoft. Tom Clancy's HAWX. http://hawxgame.us.ubi.com/.

[23] Ubisoft. Tom Clancy's Endwar. http://endwargame.us.ubi.com/.

[24] Kai-Fu Lee, Hsiao-Wuen Hon, and Raj Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics Speech and Signal Processing*, 38(1), 1 1990.

[25] Carnegie Mellon University. Sphinx 4 javadoc. http://cmusphinx.sourceforge.net/sphinx4/javadoc/overview-summary.html#overview_description.

[26] Maciej Handwerker. Strona domowa Macieja Handwerkera. http://www.maciekhandwerker.com/.

[27] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *COMMUNICATIONS OF THE ACM*, 30(11):964–971, 1987.

[28] R. Gary Leonard and George R. Doddington. A speaker-independent connected-digit database. http://www.ldc.upenn.edu/Catalog/docs/LDC93S10/tidigits.readme.html.

[29] Artur Janicki and Dariusz Wawer. Automatic speech recognition for Polish in a computer game interface. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 711–716, Szczecin, Poland, 8 2011. IEEE Xplore Digital Library.

[30] Dariusz Wawer. System rozpoznawania mowy jako interfejs użytkownika w grze komputerowej. Ogólnopolska Konferencja Inżynierii Gier Komputerowych, 3 2011.

Data ostatniej aktualizacji adresów: 20.11.2011 r.

Wykaz skrótów

AM	Acoustic Model
ASR	Automatic Speech Recognition
BSD	Berkeley Software Distribution
CMU	Carnegie Mellon University
FPP	First Person Perspective
HMM	Hidden Markov Model
HTK	Hidden Markov Model Toolkit
IVR	Interactive Voice Response
JSGF	Java Speech Grammar Format
LM	Language Model
MFCC	Mel-Frequency Cepstral Coefficients
MIT	Massachusetts Institute of Technology
MMORPG	Massive Multiplayer Online Role Playing Game
RPG	Role Playing Game
SNR	Signal-to-Noise Ratio
TPP	Third Person Perspective
WSJ	Wall Street Journal

Dodatek A

Lista komend

komendy	parametry (możliwe wartości)
skręć, zakręt, skręt	kierunek (lewo, prawo), kąt (10+ stopni)
prosta	długość (10+ metrów)
przyśpiesz	prędkość (10+ km/h)
zwolnij	prędkość (0+ km/h)
wrzuć	bieg (1-6)
zredukuj do	bieg (1-6)
cofnij, cofaj	*brak*
hamuj, zatrzymaj, stop, stój	*brak*

Dodatek B

Zdania testowe

01 przyśpiesz do pięćdziesięciu
02 przyśpiesz do sześćdziesięciu
03 przyśpiesz do siedemdziesięciu
04 przyśpiesz do osiemdziesięciu
05 przyśpiesz do dziewięćdziesięciu
06 przyśpiesz do stu
07 przyśpiesz do stu dziesięciu
08 przyśpiesz do stu dwudziestu
09 przyśpiesz do stu trzydziestu

10 przyśpiesz do stu czterdziestu
11 przyśpiesz do stu pięćdziesięciu
12 przyśpiesz do stu sześćdziesięciu
13 przyśpiesz do stu siedemdziesięciu
14 przyśpiesz do stu osiemdziesięciu
15 przyśpiesz do stu dziewięćdziesięciu
16 przyśpiesz do dwustu
17 zwolnij do pięćdziesięciu
18 zwolnij do sześćdziesięciu
19 zwolnij do siedemdziesięciu

20 zwolnij do osiemdziesięciu
21 zwolnij do dziewięćdziesięciu
22 zwolnij do stu
23 zwolnij do stu dziesięciu

24 zwolnij do stu dwudziestu
25 zwolnij do stu trzydziestu
26 zwolnij do stu czterdziestu
27 zwolnij do stu pięćdziesięciu
28 zwolnij do stu sześćdziesięciu
29 zwolnij do stu siedemdziesięciu

30 zwolnij do stu osiemdziesięciu
31 zwolnij do stu dziewięćdziesięciu
32 zwolnij do dwustu
33 zredukuj do jedynki
34 zredukuj do dwójki
35 zredukuj do trójki
36 zredukuj do czwórki
37 zredukuj do piątki
38 wrzuć jedynkę
39 wrzuć dwójkę

40 wrzuć trójkę
41 wrzuć czwórkę
42 wrzuć piątkę
43 prosta sto metrów
44 prosta sto pięćdziesiąt
45 prosta siedemdziesiąt metrów
46 prosta trzysta metrów
47 prosta dwieście pięćdziesiąt metrów
48 prosta osiemdziesiąt
49 prosta czterdzieści

50 prosta dwieście siedemdziesiąt metrów
51 prosta trzysta dwadzieścia
52 prosta pięćset metrów
53 zakręt w lewo dziewięćdziesiąt stopni
54 zakręt w prawo trzydzieści stopni
55 zakręt w lewo dwadzieścia
56 zakręt w prawo sześćdziesiąt stopni

57 ostry zakręt w prawo osiemdziesiąt
58 bardzo ostry zakręt w prawo siedemdziesiąt
59 zakręt w prawo pięćdziesiąt stopni ostry

60 zakręt w prawo ostry
61 ostry zakręt w lewo
62 skręć w lewo dwadzieścia stopni
63 skręć ostro w prawo trzydzieści stopni
64 zakręt w lewo dwadzieścia stopni łagodny
65 zakręt w prawo łagodny sześćdziesiąt stopni
66 skręć w lewo łagodnie trzydzieści stopni
67 skręć łagodnie w lewo pięćdziesiąt stopni
68 zakręt w lewo czterdzieści stopni przechodzi w zakręt w prawo trzydzieści stopni
69 zakręt w prawo trzydzieści stopni przechodzi w zakręt w lewo pięćdziesiąt stopni

70 zakręt ostry w lewo siedemdziesiąt stopni przechodzi w zakręt łagodny w prawo trzydzieści stopni
71 zakręt łagodny w prawo czterdzieści stopni przechodzi w zakręt ostry w lewo sześćdziesiąt stopni
72 zakręt w prawo trzydzieści stopni po dwudziestu metrach przechodzi w ostry lewo dwadzieścia stopni
73 zakręt w lewo czterdzieści stopni po trzydziestu metrach przechodzi w prawo pięćdziesiąt stopni
74 zakręt ostry w lewo siedemdziesiąt stopni po dwudziestu metrach przechodzi w prawo czterdzieści stopni
75 zakręt w prawo pięćdziesiąt stopni po czterdziestu metrach przechodzi w ostry zakręt w lewo siedemdziesiąt stopni
76 prosta sto metrów przyśpiesz do sześćdziesięciu
77 prosta sto pięćdziesiąt przyśpiesz do stu pięćdziesięciu
78 prosta siedemdziesiąt metrów przyśpiesz do stu czterdziestu
79 prosta trzysta metrów przyśpiesz do stu czterdziestu

80 prosta dwieście pięćdziesiąt metrów przyśpiesz do siedemdziesięciu
81 prosta osiemdziesiąt przyśpiesz do stu dwudziestu
82 prosta czterdzieści przyśpiesz do stu dziesięciu
83 prosta dwieście siedemdziesiąt metrów przyśpiesz do osiemdziesięciu

84 prosta trzysta dwadzieścia przyśpiesz do siedemdziesięciu

85 prosta pięćset metrów przyśpiesz do stu

Dodatek C

Zawartość płyty

Na płycie, dołączonej do pracy, znajdują się następujące elementy:

- artykuly - katalog, w którym znajdują się opisane w bibliografii artykuły mojego autorstwa lub współautorstwa,
- pilot_rajdowy_zrodla - katalog, w którym znajdują się źródła projektu *Pilot rajdowy*, włącznie ze zmodyfikowanymi źródłami *Sphinksa*,
- praca.pdf - elektroniczna wersja tej pracy magisterskiej,
- readme.txt - instrukcja uruchomienia *Pilota rajdowego*.