AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI
i TELEKOMUNIKACJI
KATEDRA TELEKOMUNIKACJI

Praca dyplomowa

inżynierska

Imię i nazwisko	**Paweł Helm**
Kierunek studiów	**Elektronika i Telekomunikacja**
Temat pracy dyplomowej	**Aplikacja do zakłócania mowy**
Opiekun pracy	**dr inż. Jarosław Bułat**

Kraków, rok 2013

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie (w zakresie wyszczególnionym we wstępie) i że nie korzystałem ze źródeł innych niż wymienione w pracy.

_____

Autor

Spis treści

Wstęp

Główną motywacją do napisania niniejszej pracy było opracowanie mechanizmu umożliwiającego bezinwazyjne i efektywne zakłócanie mowy. Do zrealizowania tego celu wykorzystano zjawisko słuchowego sprzężenia zwrotnego, które pozwala mózgowi na bieżąco regulować parametry ludzkiej mowy. Zaobserwowano, że nawet niewielka ingerencja w ten mechanizm skutecznie uniemożliwia człowiekowi płynne wypowiadanie się. Co ciekawe poza funkcją zakłócania mowy zjawisko to ma także drugie, zupełnie inne zastosowanie. Ponieważ pozwala ono zmodyfikować proces monitorowania własnego głosu, jest z sukcesem wykorzystywane w leczeniu osób jąkających się.

Symulacja działania opisanego mechanizmu wymaga użycia narzędzi pozwalających na jednoczesne nagrywanie i odtwarzanie dźwięku. W związku z tym, na potrzeby eksperymentalnej próby analizy zjawiska opóźnionego sprzężenia słuchowego zdecydowano się skonstruować specjalną aplikację przeznaczoną na urządzenia wyposażone w system Android. Efekt zakłócania mowy postanowiono tu osiągnąć przy pomocy algorytmu odtwarzającego mowę w czasie rzeczywistym z pewnym, płynnie regulowanym opóźnieniem. Kluczem do osiągnięcia tego celu było opracowanie programu, który optymalizuje szybkość przetwarzania dźwięku w dowolnym urządzeniu z systemem Android. Do realizacji tego zadania wykorzystano dwa różne środowiska programistyczne dostępne na tę platformę: standardowy pakiet narzędzi i bibliotek Android SDK oraz natywną bibliotekę audio OpenSL ES. Opracowane w ten sposób algorytmy pozwoliły na zakłócanie mowy dowolnej osoby wyłącznie za pośrednictwem telefonu komórkowego oraz podłączonego zestawu słuchawkowego, który dostarczając głos mówcy bezpośrednio do jego uszu uniemożliwia mu prawidłowe konstruowanie wypowiedzi.

Treść pracy została podzielona na dwa rozdziały. Pierwszy z nich wyjaśnia skąd wziął się pomysł na aplikację zakłócającą mowę i przedstawia istniejące w tej dziedzinie rozwiązania oraz ich zastosowania. Definiuje także główny problem, z którym zmierzono się podczas projektowania aplikacji i określa jego znaczenie z perspektywy systemu Android. Drugi rozdział zawiera propozycje algorytmów

zakłócających mowę wraz z analizą ich działania oraz dokładnym opisem narzędzi potrzebnych do ich zaimplementowania. Zakończenie poświęcone jest podsumowaniom efektów przeprowadzonych prac i propozycjom dalszego rozwoju aplikacji.

Rozdział 1

Zakłócanie mowy

Aby zrozumieć działanie aplikacji zakłócającej mowę należy uświadomić sobie w jaki sposób mózg człowieka kontroluje podstawowe parametry ludzkiego głosu takie jak jego wysokość, tempo czy barwa. Nadzór ten zawdzięczany jest między innymi zmysłowi słuchu, dzięki któremu człowiek jest w stanie na bieżąco analizować każde z wypowiadanych słów i modulować je w dogodny dla siebie sposób. Posługując sie terminologią informatyczną człowiek nie posiada domyślnie zakodowanych ustawień dotyczących parametrów swojego głosu. Fakt ten stanowi jedną z przyczyn dla której ludzie z ubytkiem słuchu tracą możliwość poprawnej modulacji mowy. W sytuacji kiedy głos dostarczany jest uszom z pewnym opóźnieniem, mózg człowieka podejmuje próbę korekty i synchronizacji wydawanych dźwięków z dźwiękami słyszanymi. Próba przezwyciężania sztucznego opóźnienia powoduje zazwyczaj powstanie dyskomfortu skutecznie uniemożliwiającego formułowanie poprawnie brzmiącej wypowiedzi, mowa staje się ciężka do zrozumienia, pełna zająknięć i powtórzeń. Skonstruowanie logicznie brzmiącego zdania okazuje się być dużym wyzwaniem.

Zupełnie inne zastosowanie tego zjawiska proponuje strona internetowa fundacji „The stuttering fundation” powołanej na rzecz osób jąkających się [1]. Według niej, opisany mechanizm pomaga osobom dotkniętym tą chorobą w płynnym konstruowaniu wypowiedzi. Teza ta poparta jest badaniami przeprowadzonymi przez Amerykańską Akademię Neurologii na grupie dorosłych osób ze skłonnościami do jąkania się. W doświadczeniu wzięto pod uwagę 3 czynniki: częstotliwość jąkania się, stopień niezrozumiałości wypowiedzi, oraz jej długość. Wykazano, że u osób z pewną asymetrycznością fragmentu kory mózgowej odpowiedzialnej m.in. za mowę, odtworzenie głosu z pewnym opóźnieniem wspomaga płynność wypowiedzi. Mimo, że zauważalną poprawę zaobserwowano jedynie u części badanych, warto podkreślić pozytywny wpływ zjawiska opóźnienia audio, które może być wykorzystywane także w badaniach medycznych.

1.1 Istniejące rozwiązania sprzętowe i programowe.

Wiedza na temat zjawiska zakłócania mowy pozwala wykorzystać go na wiele różnych sposobów. Jako jedni z pierwszych zastosowanie dla niego znaleźli uczeni z National Institute of Advanced Industrial Science and Technology w Japonii [2]. Stworzyli oni urządzenie, którego zadaniem jest utrzymywanie ciszy w pomieszczeniach, w których obowiązuje bezwzględny zakaz rozmów, takich jak biblioteki czy sale wykładowe. Cel ten zrealizowany został przy pomocy przyrządu wyglądem przypominającego pistolet (rysunek 1.1), zbudowanego z głośnika i mikrofonu kierunkowego, miernika odległości oraz laserowego wskaźnika. Aby wywołać efekt dyskomfortu podczas mówienia należy wycelować urządzenie w osobę zakłócającą ciszę. Wygenerowany dźwięk będzie powieleniem słów wypowiadanych przez mówiącego, ale odtworzony zostanie mu z pewnym opóźnieniem. Po zwolnieniu przycisku generującego zagłuszający sygnał, mówca jest w stanie wrócić do płynnego wypowiadania się, gdyż zakłócanie mowy nie powoduje trwałych efektów.

Rysunek 1.1 Speech Jammer

Poza rozwiązaniami sprzętowymi implementującymi mechanizm odtwarzania opóźnionej mowy istnieją także rozwiązania softwareowe przeznaczone na urządzenia wyposażone w system Android. W największym internetowym sklepie z aplikacjami mobilnymi dla tego systemu Google Play, dostępnych jest kilka programów, które

zapewniają podobną funkcjonalność. Na szczególną uwagę zasługują tu aplikacje „Voice Jammer", „Sound Jammer" oraz „Speech Jammer Brain Confuser". Pierwszy kontakt z nimi nie jest jednak pozytywny, gdyż w znacznej części są to programy niedopracowane. Taka opinia powodowana jest tym, że minimalne opóźnienia osiągane za ich pomocą najprawdopodobniej nie są dostosowane indywidualnie do urządzenia. Występują w nich także problemy związane ze zwalnianiem zarówno wykorzystywanej pamięci jak i komponentów urządzenia, co skutkuje nieprawidłowościami pojawiającymi się przy próbie zamknięcia programu. Żaden z programów nie daje możliwości modyfikowania tempa czy barwy głosu osoby z niego korzystającej. Wszystkie dostępne aplikacje są bezpłatne, a najpopularniejszą z nich jest „Voice Jammer" [3], której liczba pobrań według statystyk sklepu Google Play oscyluje pomiędzy 100 000, a 500 000. Podobne aplikacje znajdują się także w serwisach internetowych z aplikacjami przeznaczonymi na inne platformy mobilne. Apple Store oferuje płatny program o nazwie „Speech Zapper", a w sklepie Windows Phone znaleźć można aplikację japońskich konstruktorów wcześniej opisywanego urządzenia służącego do zakłócania mowy (rysunek 1.1). Ci sami uczeni dostarczają także rozwiązanie przeznaczone wyłączenie na komputery PC, które dostępne jest do pobrania na stronie internetowej projektu [2].

1.2 Opóźnienie audio w systemie Android.

Opóźnienie audio w systemie Android jest jedną z jego powszechnie znanych wad konstrukcyjnych. Aby dobrze zrozumieć istotę problemu należy zdefiniować czym tak właściwie jest opóźnienie audio dla systemu operacyjnego. Jak podaje strona internetowa *developers.android.com* [5] opóźnieniem audio można nazwać czas jaki potrzebuje sygnał dźwiękowy na przejście przez tenże system. Zazwyczaj wyróżnia się dwa jego rodzaje:

- wejściowe - czas pomiędzy wygenerowaniem zewnętrznego dla urządzenia dźwięku, a pojawieniem się go w buforach systemowych,
- wyjściowy - czas pomiędzy rozpoczęciem odczytywania danych z buforów, a fizyczną generacją dźwięku przez głośnik.

Każda aplikacja, która w swoim działaniu uwzględnia tego typu przetwarzanie dźwięku narażona jest na wystąpienie problemu opóźnienia. Przykładem mogą być tu wszystkie programy zbliżone swoją specyfikacją do aplikacji Speech Jammer.

Problem opóźnienia audio stał się na tyle poważny, że do jego rozwiązania firma Google zaangażowała grupę inżynierów specjalizujących się w dziedzinie przetwarzania dźwięku. Wyniki ich prac przedstawiono w ramach corocznej konferencji *Google I/O* w San Francisco. Zamieszczony w maju tego roku zapis filmowy wykładu Glenna Kastena i Rapha Leviena [4] porusza tematykę opóźnienia wyjściowego dźwięku w urządzeniach opartych o system Android na podstawie aplikacji syntezatora dźwięku. Podczas wykładu przedstawiono wszelkie źródła problemu oraz dokonano głębokiej analizy w jaki sposób można przeciwdziałać długim opóźnieniom audio. Skupiono się także na dotychczas podjętych przez firmę Google krokach, które miały na celu usprawnienie systemu w kontekście skrócenia czasu przejścia sygnału dźwiękowego przez system. Działania te obejmowały:

- stworzenie narzędzi pozwalających pobrać optymalne parametry audio urządzenia takie jak częstotliwość próbkowania czy wielkość buforu przechowującego dźwięk,
- umożliwienie implementacji kolejki FIFO przeznaczonej wyłącznie do obsługi wątków audio za pośrednictwem natywnej biblioteki OpenSL ES,
- uproszczenie drogi jaką przebywa sygnał dźwiękowy w systemie Android.

Osiągnięcie minimalnego opóźnienia audio w systemie Android nie jest możliwe bez wykorzystania zaproponowanych w wykładzie rozwiązań. Ich szczegółowa analiza została przedstawiona w dalszej części pracy (podrozdział 2.3.4 oraz 2.5)

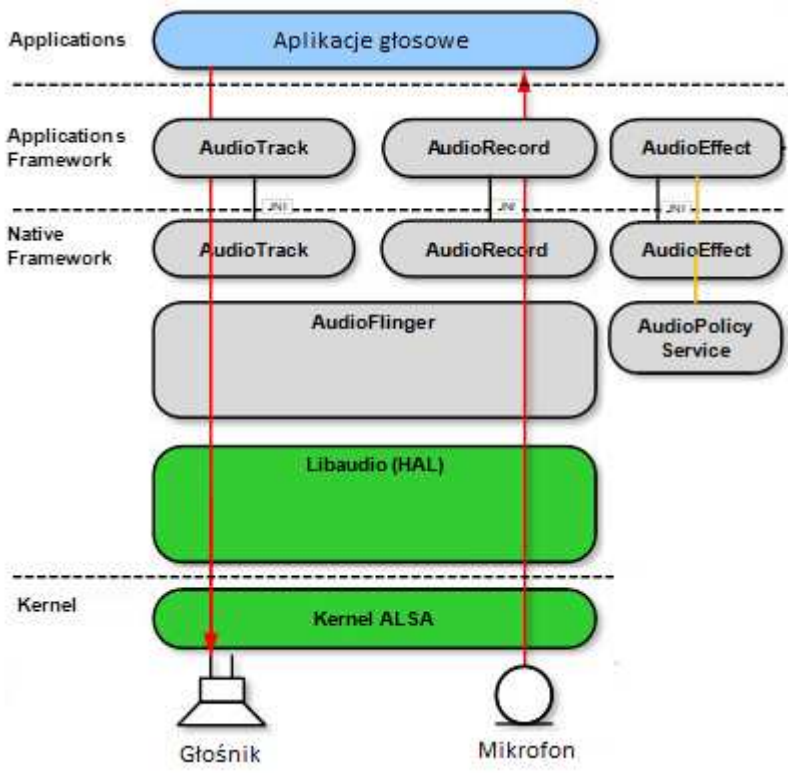
Rozdział 2

Implementacja algorytmów zakłócających mowę

2.1 Zarys architektury audio systemu Android

Architektura audio środowiska Android [6] została zaprojektowana w oparciu o budowę warstwową. Taki podział miał stworzyć podstawy dla aplikacji działających w ramach tego systemu. Stos złożony z poszczególnych warstw został przedstawiony na rysunku 2.1 i można w nim wyróżnić:

- warstwę aplikacji (*Applications Layer*),
- warstwę z którą bezpośrednio komunikują się aplikacje (*Applications Framework Layer*),
- warstwę natywnych bibliotek systemu Android (*Native Framework Layer*),
- warstwę jądra systemu (*Linux Kernel Layer*).



Rysunek 2.1 Architektura audio systemu Android

Na samym szczycie stosu znajduje się warstwa aplikacji odpowiedzialna za komunikacje z użytkownikiem. Zazwyczaj jest nią interfejs, za pośrednictwem którego uruchamiane są poszczególne funkcjonalności aplikacji. Kolejna warstwa przeznaczona jest wyłącznie dla programistów, gdyż jest nią kod aplikacji wydający polecenia komponentom urządzenia. W przypadku bibliotek audio, na tym poziomie architektury systemu Android wykorzystywane są głównie klasy *AudioTrack, AudioRecord* czy *AudioEffect*. Warstwa ta wywołuje odpowiadające jej natywne biblioteki. Zadaniem bibliotek natywnych jest bezpośrednia komunikacja z komponentami urządzenia. Napisane są głównie z językach C/C++ i posiadają zestaw funkcji instruujących urządzenie w zakresie obsługi różnych typów danych. Kod wywoływany w tej części stosu ma za zadanie uzyskać dostęp do serwisów, czyli zaimplementowanych w systemie procesów, wspierających wykonywanie określonych operacji. Serwisy te zostały zlokalizowane w części stosu określanej nazwą *AudioFlinger* i zaliczyć do nich można serwis *AudioPolicy* odpowiedzialny za zapewnienie odpowiedniej kolejności wykonywania zadań. Aby zagwarantować spójność całej architektury wprowadzono także element który łączy część logiczną systemu z częścią sprzętową. Jest nim *Hardware Abstraction Layer (HAL),* który dostarcza wymaganych przez system Android interfejsów dla sterowników zlokalizowanych w niższej warstwie. U podstawy stosu leży warstwa jądra systemu. Można uznać, że jest to serce całego systemu, ponieważ znajdują się w niej sterowniki odpowiedzialne za poprawne działania wszystkich komponentów urządzenia. Z perspektywy nagrywania i odtwarzania dźwięku komponenty te to głośnik oraz mikrofon.

2.2 Algorytm zakłócający mowę oparty na Android SDK

2.2.1 Konfiguracja środowiska

Stworzenie aplikacji na system Android wymaga skonfigurowania odpowiedniego środowiska programistycznego. Pierwszym etapem jest instalacja pakietu Java SE Development Kit, czyli zestawu narzędzi używanych do projektowania aplikacji w języku Java. Poza nim wymagane jest pobranie zestawu Android SDK, który przeznaczony został wyłącznie do tworzenia programów na system Android. W jego skład wchodzi m.in.:

- Android API, czyli zbiór bibliotek wykorzystywanych do projektowania aplikacji,
- emulator Android, czyli program komputerowy symulujący działanie prawdziwego urządzenia wyposażonego w system Android, którego zadaniem jest umożliwienie testowania aplikacji bez potrzeby posiadania fizycznego urządzenia,
- zestaw narzędzi do kompilowania i debugowania aplikacji.

Oba zestawy narzędzi należy odpowiednio skonfigurować ze środowiskiem programistycznym, w którym tworzone są aplikacje. Zalecanym przez Google środowiskiem jest *IDE Eclipse*, który posiada specjalną wtyczkę *ADT* (*Android Development Tool*) umożliwiającą integrację zestawu Android SDK w *Eclipse*.

2.2.2 Realizacja algorytmu

Najprostszy sposób realizacji zadania projektowego zakładał użycie narzędzi programistycznych Android SDK. W środowisku tym obsługę audio zapewnia kilka klas, każda z nich cechuje się innymi właściwościami. Wyróżnić tu można klasę:

- SoundPool [7] zaprojektowaną do obsługi krótkich plików audio, przechowywanych w pamięci podręcznej, zapewniając tym samym szybki dostęp do nieskompresowanego nagrania. Używana jest głównie do obsługi efektów dźwiękowych w aplikacjach i grach. Niewielki limit pamięci przeznaczony na obsługę tej klasy sprawia że bardzo łatwo jest go przekroczyć i doprowadzić do pojawienia się wyjątku *OutOfMemoryException*.
- MediaPlayer [8] stworzoną z myślą o obsłudze dużych plików muzycznych. W tym wypadku plik audio odczytywany jest z pamięci stałej urządzenia każdorazowo przy wywołaniu funkcji *create()*. Mechanizm ten na pewno odciąża pamięć podręczną jednak znacząco wydłuża czas dostępu do nagrania.
- AudioTrack [9] zapewniającą mechanizm polegający na wczytywaniu do buforów fragmentów pliku audio i późniejszym ich odtwarzaniu. Pozwala na dwa tryby pracy, statyczny i strumieniowy. Wyboru pomiędzy nimi dokonuję

się na podstawie długości bloku dźwięku przeznaczonego do umieszczenia w buforze, czy też szybkości próbkowania.

Na potrzeby aplikacji Speech Jammer zainicjowano obiekt klasy *AudioTrack* (tabela 2.1). Pierwszy z parametrów wymaganych przez konstruktor powinien określać rodzaj strumienia audio obsługiwanego przez klasę. W tym przypadku wybrano opcję *STREAM_MUSIC*, która najbardziej odpowiada potrzebom programu, gdyż umożliwia strumieniowanie plików muzycznych. Kolejnymi parametrami są częstotliwość próbkowania sygnału wyrażana w Hertzach, model kanału wyjściowego dźwięku oraz sposób kodowania sygnału określany w liczbie bitów przypadających na jedną próbkę. Zdefiniować należy także tryb w jakim pracować będzie obiekt klasy *AudioTrack.* W tym przypadku wyboru trzeba dokonać pomiędzy trybem statycznym, a strumieniowym. Bardzo ważnym parametrem jest wielkość buforu, w którym przechowywany będzie spróbkowany fragment sygnału. Dokumentacja biblioteki Android SDK [9] sugeruje, aby wielkość ta nie była mniejsza od wartości zwracanej przez funkcję *getMinBufferSize()*, gdyż spowodować to może błędne działanie programu.

Tabela 2.1 **Obiekt klasy *AudioTrack***

```
int NTrack = AudioTrack.getMinBufferSize(48000,
                        AudioFormat.CHANNEL_OUT_MONO,
                        AudioFormat.ENCODING_PCM_16BIT);

track = new AudioTrack(AudioManager.STREAM_MUSIC, 48000,
                        AudioFormat.CHANNEL_OUT_MONO,
                        AudioFormat.ENCODING_PCM_16BIT,NTrack,
                        AudioTrack.MODE_STREAM);
```

Kolejnym krokiem, który powinien zostać uwzględniony w projektowaniu aplikacji jest zapewnienie obsługi wejściowego strumienia dźwięku, czyli głosu który będzie wypowiadany. W tym celu skorzystano z funkcjonalności klasy *AudioRecord* [10], która swoje działanie opiera podobnie jak klasa *AudioTrack* na buforze (tabela 2.2). Dźwięk dostający się do urządzenia zamieniany jest w ciąg bitów zgodnie z parametrami zdefiniowanymi w konstruktorze i dostarczany jest do bufora. Parametry te powinny być jednolite z tymi, które przekazano do obiektu *AudioTrack*, aby zapewnić kompatybilność obu obiektów. W szczególności mowa jest tu o częstotliwości próbkowania, modelu kanału wyjściowego dźwięku oraz sposobie kodowania sygnału.

Klasa *AudioRecord* również wymaga wykorzystania funkcji *getMinBufferSize()*, która określa minimalny bufor potrzeby do stworzenia obiektu.

Tabela 2.2 **Obiekt klasy *AudioRecord***

```
int NRecord = AudioRecord.getMinBufferSize(48000,
                          AudioFormat.CHANNEL_IN_MONO,
                          AudioFormat.ENCODING_PCM_16BIT);

recorder = new AudioRecord(AudioSource.MIC, 48000,
                          AudioFormat.CHANNEL_IN_MONO,
                          AudioFormat.ENCODING_PCM_16BIT, NRecord);
```

Sercem programu jest niekończąca się pętla *while()*, która odpowiada za nieustanne nagrywanie i odtwarzanie dźwięku. W celu realizacji takiej funkcjonalności, wewnątrz pętli użyto dwóch funkcji. Pierwsza z nich, *read()*, będąca funkcją klasy *AudioRecord*, wczytuje emitowany sygnał do bufora, który następnie przekazywany jest do funkcji *write()*. Jej zadaniem jest dostarczenie zawartości bufora na wyjście, czyli odtworzenie zapisanego fragmentu dźwięku.

Koniecznie należy pamiętać także o nadaniu właściwego priorytetu wcześniej opisanym czynnościom. Na potrzeby programu zdefiniowano odrębną klasę *Audio*, która odpowiada za wszystkie czynności związane z nagrywaniem i odtwarzaniem dźwięku. W związku z tym, konstruktorowi tej klasy przypisano najwyższy możliwy priorytet wykonywania procesu (tabela 2.3). Funkcja *setThreadPriority()* przypisuje wątkom priorytet z zakresu od -20 (dla najbardziej pilnych wątków) do 19 (dla najmniej pilnych wątków). Opcja *THREAD_PRIORITY_URGENT_AUDIO* zapewnia wartość równą -19, co oznacza bardzo wysoki priorytet wykonywanego zadania. Bardzo ważne jest, aby operacja ta była wykonywana wewnątrz funkcji *setThreadPriority()*, ponieważ przyznanie tak wysokiego priorytetu przez użytkownika bez uprawnień ,,roota" w systemie Linux nie jest możliwe [11].

Tabela 2.3 **Przypisanie priorytetu wątkowi**

```
public Audio(){
      android.os.Process.setThreadPriority(
            android.os.Process.THREAD_PRIORITY_URGENT_AUDIO);
      start();
}
```

Bardzo ważne jest także zmodyfikowanie pliku *AndroidManifest.xml*. Manifest jest niezbędnym plikiem każdej aplikacji przeznaczonej na system Android, gdyż przekazuje kluczowe informacje na temat programu do systemu. Przykładem takich informacji są pozwolenia, których zadaniem jest ograniczenie dostępu do poszczególnych funkcjonalności urządzenia. Aby system pozwolił na nagrywanie i odtwarzanie dźwięku konieczne jest dodanie pozwoleń na rejestrowanie dźwięku oraz modyfikowanie ustawień audio (tabela 2.4).

Tabela 2.4 **Pozwolenia zdefiniowane w pliku *AndroidManifest.xml***

```
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
```

2.3 Algorytm zakłócający mowę oparty na Android NDK

2.3.1 Biblioteka OpenSL ES – wprowadzenie

Nieustanna fragmentacja rynku sprzętu multimedialnego zainspirowała działających na nim producentów do stworzenia uniwersalnej platformy programistycznej, która pozwoliłaby na to, aby raz napisany kod aplikacji działał identycznie na każdym urządzeniu multimedialnym, gwarantując przy tym ten sam poziom funkcjonalności.. Powołane do tego celu konsorcjum Khronos Group zrzeszające m.in. producentów telefonów komórkowych, sprzętu audio, komputerów osobistych czy też dostawców oprogramowania stworzyło natywną bibliotekę OpenSL ES (*Open Sound Library for Embedded Systems*), której głównym celem była obsługa aplikacji audio.

W rezultacie pracy powołanej grupy powstała bardzo obszerna biblioteka, której granicę możliwości wyznacza sprzęt, na którym jest implementowana. Szeroki zakres funkcjonalności biblioteki można podzielić na następujące grupy:

- odtwarzanie dźwięku – funkcje pozwalające na odtwarzanie plików dźwiękowych,
- efekty i kontrola – zawierają funkcje zmieniające głośność, tempo, wysokość generowanych dźwięków jak i pozwalają dodać do nich efekty przy użyciu chociażby korektora,
- nagrywanie audio – pozwala na nagrywanie dźwięku m.in. w formacie PCM, którego źródłem może być wbudowany w urządzenie mikrofon, czy też podłączony przez wejście Jack zestaw słuchawkowy.

2.3.2 Budowa aplikacji z użyciem OpenSL ES

OpenSL ES pozwala na obiektowe podejście do projektowania aplikacji wykorzystując przy tym język C. Założenie to opiera się na dwóch fundamentalnych elementach: obiekcie oraz interfejsie. Pierwszy z nich jest abstrakcyjną formą reprezentacji określonych zasobów i zadań. Każdemu stworzonemu obiektowi przypisywany jest określony typ. Typ ten zaś definiuje zbiór zadań, które może wykonać dany obiekt. Obiekt może być uważany za odpowiednik klasy w C++. Interfejs z kolei jest zbiorem powiązanych z danym obiektem funkcji. Podobnie jak obiekt, interfejs posiada typ, który jednoznacznie definiuje zbiór obsługiwanych przez niego metod. Interfejs można zdefiniować jako zbiór funkcji danego typu oraz przypisanych do nich obiektów. Ważnym pojęciem w nomenklaturze OpenSL ES jest także numer identyfikacyjny interfejsu, który dokładnie określa jego typ.

Aby rozpocząć pisanie kodu w OpenSL ES niezbędna jest wiedza na temat podstawowego interfejsu *SLObjectItf*, który jest dziedziczony przez każdy skonstruowany obiekt niezależnie od typu. Dzięki niemu aplikacja zyskuje podstawowe funkcjonalności takie jak:

- kasowanie obiektu za pomocą funkcji *Destroy()*,

- pobieranie innych interfejsów przy użyciu funkcji *GetInterface()*, należy przy tym pamiętać że dany obiekt może dziedziczyć tylko jeden interfejs,
- kontrola stanu obiektu uzyskiwana dzięki funkcjom *Realize()* oraz *Resume()*.

Kolejnym niezbędnym elementem architektury OpenSL ES jest obiekt silnika (*EngineObject*) oraz związany z nim interfejs *SLEngineItf*. Każda aplikacja musi być rozpoczęta od implementacji obiektu silnika. W dalszej fazie projektowania aplikacji posłuży on do pobrania innych niezbędnych interfejsów i do tworzenia obiektów wymaganych przykładowo przy nagrywaniu lub odtwarzaniu dźwięku. Silnik zbiera także dane na temat możliwości urządzenia i określa czy dana funkcjonalność jest dostępna [12]. Przykładowo aplikacja przetwarzająca jedynie 8-bitowe pliki audio z częstotliwością 8 kHz może nie współpracować z systemem, który używa częstotliwości próbkowania większej niż chociażby 16 kHz.

2.3.3 Aplikacje natywne w systemie Android

Aby rozpocząć pracę z natywnym środowiskiem programistycznym Android NDK należy, poza wcześniej skonfigurowanym środowiskiem *Eclipse*, pobrać pakiet bibliotek natywnych, który dostępny jest na oficjalnej, przeznaczonej dla programistów stronie internetowej *developer.android.com*. Oprócz biblioteki OpenSL ES do obsługi audio, w paczce znajduje się szereg innych natywnych narzędzi programistycznych, takich jak chociażby biblioteka graficzna OpenGL. Przykładowe programy dostarczone wraz z biblioteką pozwalają na praktyczne zapoznanie się z pisaniem kodu przy jej użyciu. Aby zainstalować bibliotekę wystarczy rozpakować ją w dowolnie wybranym katalogu, a następnie wskazać programowi *Eclipse* ścieżkę do pliku *ndk-build.cmd* odpowiedzialnego za kompilację. Niezbędne jest także dodanie folderu *libs* do drzewa katalogowego projektu na potrzeby przechowywania wszystkich bibliotek zewnętrznych używanych w programie. W aplikacji Speech Jammer będzie to biblioteka OpenSL ES. Dodatkowo należy pamiętać, że każda biblioteka natywna wymaga stworzenia oddzielnego folderu o nazwie *jni*, w którym umieszczane będą wszystkie pliki z kodem źródłowym napisanym przy użyciu natywnych języków programowania C/C++. Oprócz nich, w folderze tym musi znaleźć się plik *Android.mk*

(tabela 2.5), który definiując najważniejsze parametry kodu natywnego, dostarcza niezbędnych informacji narzędziom kompilującym.

Tabela 2.5 **Plik *Android.mk***

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := opensl_example
LOCAL_C_INCLUDES := $(LOCAL_PATH)
LOCAL_CFLAGS := -O3
LOCAL_CPPFLAGS :=$(LOCAL_CFLAGS)

LOCAL_SRC_FILES := opensl_example.c \ opensl_io.c \
java_interface_wrap.cpp

LOCAL_LDLIBS := -llog -lOpenSLES

include $(BUILD_SHARED_LIBRARY)
```

Jedną ze zmiennych w pliku *Android.mk* jest zmienna *LOCAL_PATH*, która wymaga określenia lokalizacji pliku z kodem natywnym. Użyta tu funkcja *my-dir* zwraca lokalizację pliku *Android.mk*, dlatego też plik ten powinien znaleźć się w jednym folderze wraz z plikiem zawierającym natywny kod źródłowy. Opcja *include $(CLEAR_VARS)* czyści wszelkie zmienne, które mogły pozostać w pamięci po ostatniej kompilacji programu. Zmienna *LOCAL_MODULE* odnosi się do nazwy biblioteki, z której program ma korzystać w trakcie kompilacji. Zmienna *LOCAL_C_INCLUDES* wskazuje lokalizację plików nagłówkowych używanych przez główny program. Przez użyciu zmiennych *LOCAL_CFLAGS* oraz *LOCAL_CPPFLAGS* definiowane są specjalne opcje używane przy kompilacji. Flaga -03 oznacza stopień optymalizacji kodu, który w podręczniku Sylvaina Ratabouila „Android NDK" określany jest mianem „agresywnego" [13]. Zmienna *LOCAL_SRC_FILES* wskazuje pliki, które powinny być wzięte pod uwagę w procesie kompilacji. Ostatnia linia zwiera komendę *include $(BUILD_SHARED_LIBRARY)*, która ostatecznie kompiluje program i definiuje typ biblioteki, który ma zostać wygenerowany. Zawarta tu opcja *SHARED_LIBRARY* jest jedyną dostępną opcją umożliwiającą bezpośrednie wywoływania z poziomu Javy.

Jedną z wad korzystania z bibliotek natywnych w aplikacji przeznaczonych na system Android jest skomplikowana interakcja pomiędzy kodem pisanym w języku

C/C++, a głównym kodem programu pisanym w języku Java. Aby zapewnić pełną integralność, kod pisany w języku C/C++ musi spełniać pewne warunki. Jeden z nich bezpośrednio wiąże się ze strukturą danych nazwaną „JNIenv", która zapewnia interfejs do implementacji większości funkcji dostępnych w środowisku natywnym. Kolejnym utrudnieniem jest inna nomenklatura nazw typów zmiennych. Dla porównania z językiem Java, odpowiednikiem zmiennej *true* typu *boolean* jest zmienna *JNI_TRUE* w kodzie natywnym [13].

Aby ułatwić interakcję pomiędzy kodem pisanym w języku C, a głównym kodem źródłowy programu skorzystano z narzędzia SWIG, które automatycznie generuje funkcje Java na podstawie napisanych przez programistę funkcji natywnych. SWIG jest środowiskiem programistycznym stworzonym w celu ułatwienia implementacji interfejsów pisanych w C lub C++ w aplikacjach wykorzystujących inne języki programowania. Kod generowany przez SWIG pozwala na płynne połączenie interfejsów C/C++ z interfejsami innych docelowych języków. Dzięki temu, w plikach źródłowych zawierających kod natywny nie ma potrzeby definiowania zmiennej środowiskowej „JNIenv" oraz dozwolone jest używanie klasycznych nazw typów zmiennych [14].

Obsługiwane przez system Android narzędzia do programowania aplikacji natywnych są w dużej mierze zależne od inżynierów pracujących w Google. Z tego powodu nie wszystkie funkcje biblioteki OpenSL ES można użyć przy tworzeniu programu na tę platformę. Wciąż brakuje tu wielu kluczowych funkcji jak chociażby kontroli wysokości generowanych dźwięków, jednak mimo to implementacja natywnej biblioteki znacznie rozszerza możliwości audio urządzenia. Tabela 2.6 ilustruje jakie interfejsy są obsługiwane na urządzeniach wyposażonych w system Android, w zależności od typu utworzonego obiektu. Kolor zielony oznacza, że interfejs jest obsługiwany. Dla przykładu, interfejs korektora, który został zaimplementowany w dalszej części programu, może być użyty zarówno dla obiektu *Audio Player* jak i obiektu *OutputMix*. Niezależnie od tego, z którego obiektów skorzysta użytkownik będzie miał on dostęp do tych samych funkcji.

Tabela 2.6 **Interfejsy biblioteki OpenSL ES obsługiwane w środowisku Android [15]**

	Audio player	Audio recorder	Engine	Output mix
Bass boost				
Buffer queue				
Dynamic interface mgmt.				
Effect send				
Engine				
Env. reverb				
Equalizer				
Metadata extraction	decode to PCM			
Mute solo				
Object				
Play				
Playback rate				
Prefetch status				
Preset reverb				
Record				
Seek				
Virtualizer				
Volume				
Buffer queue data locator	source			
I/O device data locator		source		
Output mix data locator	sink			
URI data locator	source			

2.3.4 Realizacja algorytmu

Jak dowiodły badania minimalnego opóźnienia, którego wyniki zostały przedstawione w podrozdziale 2.6, użycie standardowych narzędzi programistycznych na platformę Android nie zapewniło osiągnięcia optymalnie niskiej wartości czasu przejścia dźwięku przez system. Dlatego też, skorzystanie z biblioteki OpenSL ES w kontekście aplikacji Speech Jammer było podyktowane chęcią uzyskania jeszcze mniejszego opóźnienia audio niż to uzyskane w przypadku pakietu Android SDK. W

związku z tym, aplikacja została zaprojektowana w ten sposób, aby zapisywanie, analiza i odtwarzanie dźwięku odbywało się przy użyciu narzędzi natywnych.

We wstępnym etapie projektowania aplikacji stworzono plik nagłówkowy *opensl_io.h* zawierający strukturę *OPENSL_STREAM*, w której zadeklarowano wszystkie zmienne użyte w części natywnej aplikacji. Wewnątrz pliku nagłówkowego znalazły się także funkcje odpowiedzialne za przekazywanie danych pomiędzy kodem pisanym w C, a kodem pisanym w Java. W głównym pliku *opnesl_io.c* zdefiniowano szereg funkcji odpowiedzialnych za prawidłowe przetwarzanie sygnału dźwiękowego. Pierwszą z nich było zaimplementowanie obiektu silnika wymaganego w każdym programie wykorzystującym bibliotekę OpenSL ES. W tej fazie projektowania należy wspomnieć także o funkcji interfejsu *SLOutputMixItf*, który reprezentuje narzędzia wyjściowe dźwięku (głośnik lub zestaw słuchawkowy). Ponieważ odtwarzany dźwięk jest niejako automatycznie przesyłany do głośnika nie ma potrzeby używać tu specjalnie dedykowanego interfejsu. Na bazie wcześniej skonstruowanego silnika oparto rzeczywisty rdzeń aplikacji, czyli obiekty *bqPlayerObject* oraz *recorderObject*, stworzone z wykorzystaniem funkcji *CreateAudioPlayer()* oraz *CreateAudioRecorder()* należących do interfejsu silnika. Każda z nich wymaga dostarczenia szeregu argumentów, bez których nie możliwe będzie prawidłowe działanie metody.

Tabela 2.7 **Konstrukcja obiektu *bqPlayerObject***

```
SLDataLocator_AndroidSimpleBufferQueue loc_bufq =
{SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEUE, 2};

SLDataFormat_PCM format_pcm = {SL_DATAFORMAT_PCM,channels, sr,
                               SL_PCMSAMPLEFORMAT_FIXED_16,
                               SL_PCMSAMPLEFORMAT_FIXED_16,
                               speakers, SL_BYTEORDER_LITTLEENDIAN};

SLDataSource audioSrc = {&loc_bufq, &format_pcm}; // source

SLDataLocator_OutputMix loc_outmix = {SL_DATALOCATOR_OUTPUTMIX,
                                      p->outputMixObject};

SLDataSink audioSnk = {&loc_outmix, NULL};   // sink

const SLInterfaceID ids1[4] = {SL_IID_ANDROIDSIMPLEBUFFERQUEUE,
                               SL_IID_VOLUME,
                               SL_IID_EQUALIZER,
                               SL_IID_PLAYBACKRATE};

const SLboolean req1[4] = {SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE,
                           SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE};

result = (*p->engineEngine)->CreateAudioPlayer(p->engineEngine,
                                               &(p->bqPlayerObject),
                                               &audioSrc, &audioSnk,
                                               4, ids1, req1);
```

Aby wygenerować obiekt *bqPlayerObject* (tabela 2.7) należy określić źródło dźwięku *source* oraz miejsce jego ujścia *sink*. Pierwszemu z tych argumentów przypisano bufor, w którym znajduje się oczekujący na odtworzenie, spróbkowany sygnał dźwiękowy. Na potrzeby drugiego z argumentów stworzono specjalny obiekt *loc_outmix*, który pełni rolę standardowego wyjścia audio, czyli głośnika. Przy okazji generowania obiektu z użyciem funkcji *CreatePlayerObject()* należy zdefiniować podstawowe parametry przesyłanego dźwięku takie jak format danych, liczba kanałów czy też częstotliwość próbkowania. Tablice *ids1[4]* oraz *req1[4]* wspólnie określają liczbę i rodzaj interfejsów, które zostaną stworzone z użyciem obiektu *bqPlayerObject*.

Biblioteka OpenSL ES wykorzystuje mechanizm wywołań zwrotnych *Callback*[1], którego głównym zadaniem jest obsługa buforów przechowujących spróbkowany dźwięk. *Callback* sygnalizuje programowi, że obecnie obsługiwany bufor jest przepełniony i wskazuje inny bufor gotowy przyjąć nadchodzące dane. Aby ostatecznie

[1] W dalszej części pracy określenia *Callback* oraz „wywołanie zwrotne” będą używane zamiennie.

uruchomić obiekt odtwarzacza należy zmienić jego status na *SL_PLAYSTATE_PLAYING*.

W podobny sposób konstruowany jest obiekt *recorderObject* służący do nagrywania dźwięku. Tak samo jak w poprzednim przypadku należy zdefiniować parametry *source* oraz *sink*. Zadaniem obiektu *recorderObject* jest gromadzenie próbkowanego dźwięku w kolejce buforów, z której później korzystać będzie obiekt *bqPlayerObject*.

Aby mechanizm odwołań działał poprawnie należy zaprojektować sposób w jaki bufory będą kolejkowane. Tu z pomocą przychodzi funkcja *Enqueue()* interfejsu *SLBufferQueueItf*. Wywoływana jest ona zawsze wtedy, gdy kolejka buforów jest gotowa przyjąć nowy bufor danych wejściowych lub wyjściowych. Funkcji *Enqueue()* można użyć zarówno w ciele wywołania zwrotnego, jak również w dowolnym innym miejscu programu. Jeśli jednak wykorzystana zostanie druga z tych opcji, należy pamiętać o tym, że bufor powinien być zakolejkowany już podczas uruchamiania nagrywania, bądź odtwarzania dźwięku, gdyż w przeciwnym wypadku *Callback* nigdy nie zostanie wywołany.

W algorytmie odpowiedzialnym za uzyskanie minimalnego opóźnienia audio zastosowano mechanizm wywołań zwrotnych, który powiadamia program główny o tym, że pełny bufor gotowy jest do dostarczenia na wyjście, czyli do głośnika. Aby jednocześnie obsłużyć strumień wejściowy oraz wyjściowy zaimplementowany został podwójny bufor, w którym wyodrębniono dwie części logiczne: jedną przeznaczoną na zapis sygnału oraz drugą używaną przez aplikacje do odtwarzania dźwięku z próbek, które zostały w nim wcześniej umieszczone. Tak skonstruowany algorytm obsługi bufora pozwala na wykonywanie dwóch zadań jednocześnie co znacznie przyspiesza działanie programu [16].

Poza zapewnieniem minimalnego opóźnienia, aplikację wyposażono w szereg innych funkcjonalności. Można do nich zaliczyć implementację *Equalziera*, czyli korektora dźwięku służącego do podbijania lub tłumienia określonego zakresu częstotliwości, który zmienia przy tym barwę emitowanego dźwięku (tabela 2.8). Biblioteka OpenSL ES umożliwia pobranie liczby obsługiwanych podpasm sygnału i modyfikowanie ich w określonym zakresie. Zakres ten może być pobrany przy pomocy

funkcji *getBandLevel()* i określany jest w milibelach. Typowe wartości w tym przypadku zawierają się w przedziale od -1500 mB do 1500 mB. Dla każdego działającego poprawnie korektora dźwięku liczba obsługiwanych podpasm musi być nie mniejsza niż dwa. Wszystkie te wartości przekazywane są do programu głównego, w którym stworzono łatwy w obsłudze interfejs. Przy pomocy suwaka, czyli komponentu w nomenklaturze Android SDK nazwanego *SeekBar*, umożliwiono konfiguracje dowolnych podpasm w pełnym ich zakresie.

Tabela 2.8 **Implementacja korektora dźwięku**

```
static SLresult openSLsetBandLevel(OPENSL_STREAM *p, int band, int level){

SLresult result;

if (NULL != p->bqPlayerEqualizer) {

      result = (*p->bqPlayerEqualizer)->SetEnabled(p->bqPlayerEqualizer,
      SL_BOOLEAN_TRUE);

      result = (*p->bqPlayerEqualizer)->SetBandLevel(p->bqPlayerEqualizer,
      band, level );

      if(result != SL_RESULT_SUCCESS)
            return result;
}
return 0;
}
```

Dodatkową funkcjonalnością jest także możliwość przyspieszenia lub opóźnienia odtwarzanego dźwięku w stosunku do tego, który został nagrany. Pozwala na to funkcja *SetRate()* interfejsu *SLPlaybackRateItf* (tabela 2.9). Zmiana ta nie pozostaje bez wpływu na wysokość emitowanych tonów. W przypadku gdy odtwarzany dźwięk jest dłuższy od oryginalnego, ton podstawowy zostaje obniżony, gdy odtwarzany dźwięk jest krótszy, ton podstawowy staje się wyższy. Brak korekcji wysokości tonu przy zmianie tempa odtwarzania jest zachowaniem domyślnym dla biblioteki OpenSL ES i określane jest stałą *SL_RATEPROP_NOPITCHCORAUDIO*. Gdy tempo odtwarzanego dźwięku jest zgodne z tempem dźwięku nagranego, wartość zwrócona przy użyciu funkcji *getRate()* wynosi 1000. W programie pozwolono na zmianę tej wartości w zakresie od 800 do 1200.

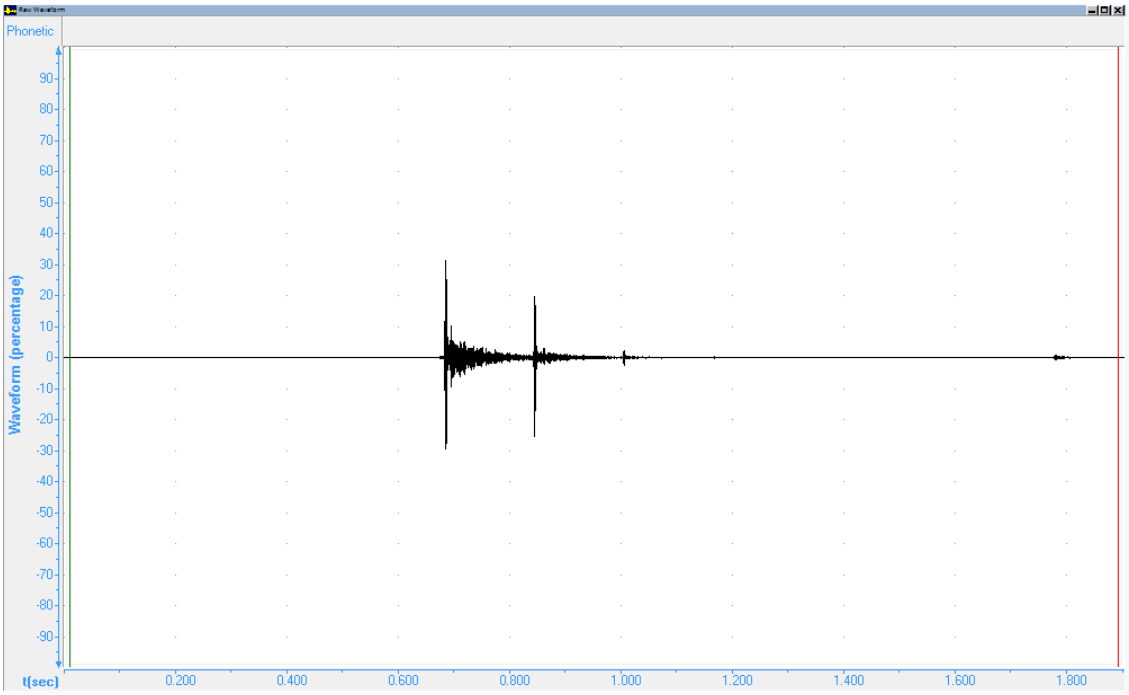
Tabela 2.9 **Implementacja funkcji regulacji tempa odtwarzanego dźwięku**

```
static SLresult openSLsetRate(OPENSL_STREAM *p, int rate){

SLresult result;
      if (NULL != p->bqPlayerRate) {

            result = (*p->bqPlayerRate)->SetRate(p->bqPlayerRate, rate);

            if(result != SL_RESULT_SUCCESS)
            return result;
      }
      return 0;
}
```

2.4 Badanie minimalnego opóźnienia

Opóźnienie audio, czyli czas jaki potrzebuje sygnał dźwiękowy na przejście przez system można zbadać na wiele sposobów, a wybór metody zależny jest jedynie od preferencji badacza. W przypadku aplikacji Speech Jammer opracowano metodę, która do tego celu wykorzystuje komputer klasy PC wraz z wbudowanymi w niego głośnikami oraz mikrofonem. Do przeprowadzenia testu wymagane było także specjalne oprogramowanie, które pozwoliło na rejestrację dźwięku dostającego się do mikrofonu oraz analizę tego nagrania. Do nagrywania dźwięku zdecydowano się użyć dostarczanego wraz systemem Windows 7 programu *Rejestrator Dźwięku*. Analiza nagranych sygnałów przeprowadzona była dostępnym na licencji freeware programem *SpeechAnalyzer*.

Badanie opierało się na analizie kształtu fali dźwiękowej w dziedzinie czasu. Materiałem badawczym były krótkie pliki audio w formacie *.wav* zawierające sygnał powstały w wyniku wygenerowania krótkiego, głośnego i wysokiego dźwięku przypominającego impuls w warunkach dostatecznej ciszy. Sygnał ten dostarczano zarówno do mikrofonu w komputerze PC oraz mikrofonu zainstalowanego w telefonie, na którym uruchomiona była aplikacja Speech Jammer. *Rejestrator Dźwięku* pracujący na komputerze natychmiastowo rejestrował wygenerowany impuls. Z pewnym opóźnieniem docierał do niego także nieco cichszy impuls, który powstawał w wyniku przejścia sygnału przez urządzenie z aktywnie działającą aplikacją Speech Jammer. Tak stworzone próbki dźwiękowe poddawane był analizie w programie *SpeechAnalzier* (rysunek 2.2**)**.



Rysunek 2.2 Wykres próbki dźwięku w programie Speech Analizer

Uzyskane wykresy obrazowały różnicę w czasie pomiędzy dwoma najgłośniejszymi fragmentami nagrania. Pierwszy widoczny impuls powstał natychmiast na skutek wygenerowania wcześniej opisywanego krótkiego, wysokiego dźwięku. Drugi wyróżniający się impuls powstał już po pewnym opóźnieniu, wynikającym z czasu potrzebnego na przejście sygnału przez system Android.

Czas trwania opóźnienia zależał od wielu czynników. Pod uwagę wzięto zarówno algorytm stworzony przy użyciu klasycznych narzędzi Android SDK, jak również algorytm napisany z wykorzystaniem biblioteki natywnej OpenSL ES. Osiągnięte wyniki różniły się od siebie w zależności od modelu urządzenia, na którym testowano programy, zdefiniowanej częstotliwości próbkowania czy długości buforu odpowiedzialnego za przechowywanie próbek. Komplet wyników przedstawiono w tabeli 2.11.

Tabela 2.10 **Wyniki badania opóźnienia audio w zależności od długości buforu, częstotliwości próbkowania i zastosowanego algorytmu.**

Urządzenie	Bufor (ilość próbek)	Częstotliwość próbkowania (Hz)	Algorytm	Opóźnienie teoretyczne (ms)	Opóźnienie zmierzone (ms)
HTC One	1024 (n)	48000 (n)	SDK	21	270
HTC One	1024 (n)	48000 (n)	NDK	21	250
HTC One	1024 (n)	44100	NDK	23	305
HTC One	4096	48000 (n)	NDK	85	268
Sony Ericsson Xperia Neo	1024 (n)	48000 (n)	SDK	21	170
Sony Ericsson Xperia Neo	1024 (n)	48000 (n)	NDK	21	150
Sony Ericsson Xperia Neo	1024 (n)	44100	NDK	23	240
Sony Ericsson Xperia Neo	4096	48000 (n)	NDK	85	201

Na pierwszy rzut oka widać znaczące różnice pomiędzy teoretycznymi wartościami opóźnienia wynikającymi z długości buforów i częstotliwości próbkowania, a wartościami, które zostały zmierzone. Przyczyną, dla której czasy te w niektórych przypadkach odbiegają od siebie nawet dziesięciokrotne jest to, że żadne z badanych urządzeń nie obsługiwało opcji *android.hardware.audio.low_latency*, która do przetwarzania dźwięku wykorzystuje specjalnie zaprojektowany kanał audio, wrażliwy na opóźnienie [15][17]. Dokładna analiza przyczyn nie pozwalających osiągnąć krótszego opóźnienia została przedstawiona w podrozdziale 2.5. Aby dobrze zrozumieć wyniki z tabeli 2.11 należy wiedzieć, że zawarte w niej oznaczenie SDK odnosi się do algorytmu wykorzystującego standardowe narzędzia Android SDK, a oznaczenie NDK do algorytmu wykorzystującego bibliotekę OpenSL ES. Litera n oznacza tu pobrane z systemu, optymalne wartości ustawień.

2.5 *Przyczyny długiego opóźnienia audio w systemie Android*

Badania opóźnienia przeprowadzone z użyciem obu algorytmów wykazały że głównym czynnikiem mającym wpływ na jego długość jest przede wszystkim wielkość bufora, w którym przetrzymywane są próbki dźwięku. Każde urządzenie charakteryzują optymalne ustawienia w kontekście osiągnięcia najmniejszego możliwego opóźnienia audio. Parametry, które należy wziąć pod uwagę to częstotliwość próbkowania oraz wcześniej wspomniana wielkość bufora. Informacje te uzyskano odnosząc się do stałych systemowych klasy *AudioManager* [18].

Tabela 2.11 **Pobieranie stałych systemowych klasy *AudioManager***

```
AudioManager am = (AudioManager) getSystemService(Context.AUDIO_SERVICE);

String sampleRate =
am.getProperty(AudioManager.PROPERTY_OUTPUT_SAMPLE_RATE);

String framesPerBuffer =
am.getProperty(AudioManager.PROPERTY_OUTPUT_FRAMES_PER_BUFFER);
```

W ten sposób otrzymane dane zostały zaimplementowane w każdym z programów. W zależności od urządzenia długość bufora może oscylować pomiędzy 512, a 1024 próbkami. Częstotliwości próbkowania dla starszych urządzeń wynoszą zazwyczaj 44100 Hz, zaś dla nowszych z reguły 48000 Hz. Pod tym względem bezkonkurencyjne jednak okazują się produkty Google z serii Nexus (tabela 2.12). Dla przykładu, model Galaxy Nexus wyposażony w system Android JellyBean może posiadać bufor wielkości 144 próbek, co przy częstotliwości próbkowania 44100 Hz daje czas trwania bufora na poziomie 3,26 ms.

Tabela 2.12 **Wielkości buforów urządzeń z serii Nexus [4]**

	Częstotliwość Próbkowania (Hz)	Długość Bufora (ilość próbek)	Długość Bufora (ms)
Nexus S	44100	880	19.95
Galaxy Nexus	44100	144(JB) / 968 (ICS)	3.26(JB) / 21.95(ICS)
Nexus 4	48000	240	5
Nexus 7	44100	512	11.6

Nexus 10	44100	256	5.8

Przy dobieraniu wielkości bufora należy zwrócić uwagę dodatkowo na dwie kwestie. Pierwsza z nich dotyczy sytuacji, gdy użyty w programie bufor jest mniejszy niż natywny bufor urządzenia. Jest to stan wysoce niepożądany, gdyż w odtwarzanym dźwięku wyraźnie słyszalne są wtedy artefakty. Drugą jest fakt, że wielkość bufora powinna być wielokrotnością wartości natywnej, otrzymanej przy pomocy klasy *AudioManager* [15]. Przyczyną takiego stanu rzeczy jest mechanizm wywołań zwrotnych używany przy kolejkowaniu buforów. Funkcją wywołań *Callback* jest informowanie głównego programu o zapełnieniu aktualnie obsługiwanego bufora i wskazanie kolejnego bufora gotowego przyjąć dane. W idealnym systemie czas potrzebny na każdorazowe wywołanie odwołań powinien być stały. Rzadko zdarza się jednak, aby założenia teoretyczne idealnie sprawdzały się w praktyce, w rezultacie czego, zdarzają się przypadki, kiedy *Callback* wywoływany jest z pewnym opóźnieniem. Dlatego też, aby pokryć tę różnicę w czasie stosuje się duże bufory, których czas trwania jest zazwyczaj większy niż 10 ms (512 próbek przy 44100 Hz).

Aby w pełni poznać źródło problemu należy zwrócić uwagę na przyczynę stojąca za występowaniem tych rzadkich opóźnień przy wywoływaniach *Callback.* Teoretycznie, przy każdym ich użyciu system powinien przydzielić tym procesom ilość CPU, która gwarantowałaby im działanie bez opóźnień. W systemie Android harmonogramem procesów zarządza algorytm o nazwie *Completely Fair Scheduler (CFS).* Jego praca opiera się na uczciwym podziale CPU, które nie faworyzowałby żadnego z wykonywanych procesów. Mimo spełnienia tego warunku, CFS niekiedy usypia procesy, nie zapewniając im przy tym jednakowo krótkich czasów potrzebnych na wybudzenie *(WakeUpTime).*

Możliwa jest przez to sytuacja, w której wątek związany z przetwarzaniem audio przez pewien czas pozostanie w uśpieniu, podczas gdy inne serwisy będą wykonywane w tle. Wyeliminowanie tego niepożądanego działania wiąże się ze zwiększeniem priorytetu wywoływania *Callback*, przy użyciu mechanizmu zarządzającego kolejką FIFO. Ponieważ system Android nie wspiera obsługi kolejek FIFO, należy skonstruować własny mechanizm wywołań zwrotnych, wykorzystując do tego celu

natywną bibliotekę OpenSL ES. Możliwe jest to dzięki temu, że biblioteka OpenSL ES do uruchomienia kodu nie korzysta z wirtualnej maszyny Dalvik, która jest domyślnie stosowana wraz z użyciem standardowych bibliotek pakietu Android SDK. Korzystanie z silnika OpenSL ES jest także dużo bardziej wydajne, gdyż nie wprowadza opóźnień związanych z odśmiecaniem pamięci *(garbage collection)* [4].

Biblioteka OpenSL ES może mieć jednak znaczący wpływ na osiągane opóźnienie jedynie w przypadku gdy urządzenie obsługuje opcję *android.hardware.audio.low_latency.* Niestety opcja ta jest dostarczana jedynie z urządzeniami Google Nexus. W urządzeniach które nie obsługują tej funkcji wpływ biblioteki OpenSL ES na opóźnienie nie będzie aż tak bardzo znaczący, a w niektórych przypadkach jej użycie może wiązać się nawet z wydłużeniem opóźnienia [15]. To tłumaczy niewielką różnicę w zmierzonym opóźnieniu w zależności od użytego algorytmu (tabela 2.10). Dla każdego z testowanych modeli telefonów biblioteka OpenSL ES pozwoliła na osiągniecie opóźnienia o ok. 20 ms krótszego niż w przypadku użycia standardowych narzędzi Android SDK.

Na długość opóźnienia audio mogą mieć wpływ także wszelkie modyfikacje, jakim poddawany jest sygnał, zanim wydostanie się z urządzenia. Tak jest na przykład w przypadku zastosowania korektora dźwięku w programie używającym algorytm natywny. Testy przeprowadzone na urządzeniu HTC One dowodzą, że w zależności od liczby zmodyfikowanych pasm opóźnienie zwiększa się stopniowo od 250 ms do 270 ms.

2.6 Konfiguracja zapewniająca najefektywniejsze zakłócanie

Zaimplementowanie algorytmów regulujących opóźnienie odtwarzanego głosu było głównym czynnikiem, który pozwolił na realizację celu projektowego, jakim było zakłócanie mowy. Stworzono aplikację, która dostarcza szereg metod powodujących trudności w mowie. Oprócz regulacji opóźnienia w zakresie od ok. 150 ms do 1500 ms. Zaprojektowano mechanizm zwalniający lub przyspieszający odtwarzaną mowę oraz korektor dźwięku mogący zmienić barwę głosu osoby mówiącej. Opcje te pozwoliły na opracowanie szeregu testów umożliwiających skonfigurowanie aplikacji z parametrami, które najefektywniej zakłócałyby mowę.

Testy przeprowadzono na grupie kilkunastu osób i podzielono je na dwie kategorie. Pierwszą z nich było czytanie ciągłego tekstu. Badano tu przede wszystkim czas potrzebny na przeczytanie danego tekstu oraz liczbę zająknięć i przerw w czytaniu. Drugą kategorię testów stanowiły odpowiedzi na spontanicznie zadawane pytania. Dotyczyły one m.in. streszczenia fabuły ostatnio oglądanego filmu, czy też opisu przyrządzenia ulubionego posiłku. W tym przypadku oceniano zrozumiałość wypowiedzi oraz logiczność i gramatyczną poprawność konstruowanych zdań.

Na każdym z badanych przeprowadzono test 4 konfiguracji programu. Były to ustawienia zapewniające:

1. krótkie opóźnienia na poziomie 150 ms,
2. długie opóźnienia na poziomie 350 ms,
3. średnie opóźnienia na poziomie 250 ms wraz z zastosowaniem mechanizmu przyspieszenia odtwarzanej mowy,
4. średnie opóźnienie na poziomie 250 ms wraz z zastosowaniem mechanizmu opóźnienia odtwarzanej mowy.

Tabela 2.13 **Wyniki pomiarów uwzględniające średni czas czytanego tekstu oraz średnią liczbę zająknięć i przerw przy włączonej aplikacji Speech Jammer**

Ustawienie	Średni Czas (s)	Średnia Liczba Zająknięć i Przerw
1	33	22
2	38	25
3	32	17
4	30	16

Testy wykazały (tabela 2.13), że najefektywniej zakłócającą mowę konfiguracją była ta, uwzględniająca mechanizm przyspieszenia odtwarzanej mowy. Czas potrzebny na przeczytanie przykładowego tekstu był z reguły o około 15% dłuższy niż w przypadku pozostałych ustawień, co wynikało z o 26% większej liczby odnotowanych zająknięć i przerw w mowie. Biorąc pod uwagę testy oceniające płynność odpowiedzi na zadawane pytania, konfiguracja ta sprawiła badanym największe trudności w

logicznym konstruowaniu zdań, w kilku przypadkach niemal całkowicie blokując możliwość wypowiedzenia się.

Badania z udziałem pozostałych konfiguracji nie dostarczyły wyników tak znacznie od siebie odbiegających, aby pozwoliły na sklasyfikowanie ich efektywności w dziedzinie zakłócania mowy. Pojedyncze rezultaty czasowe osiągane podczas czytania przykładowego tekstu zależne były wyłącznie od badanego i nie cechowały się powtarzalnością. Z tego powodu przedstawione w tabeli 2.13 uśrednione wartości pomiarów należy traktować wyłącznie jako punkt odniesienia dla pomiarów przeprowadzonych z użyciem konfiguracji nr 3, która regularnie okazywała się być najskuteczniejsza.

Wnioski i dalsze prace

Prace nad aplikacją zakłócającą mowę w dużej mierze dotyczyły analizy problemu opóźnienia audio w systemie Android. W toku poszukiwań rozwiązania tej kwestii zaimplementowano algorytmy wykorzystujące zarówno biblioteki Android SDK jak i natywną bibliotekę OpenSL ES. Osiągnięte rezultaty pozwalają stwierdzić, że dostępne narzędzia programistyczne na platformę Android nie gwarantują opóźnienia audio mniejszego niż ok. 150 ms na urządzeniach innych niż Google Nexus. Bardzo liberalna polityka licencyjna systemu Android (i firmy Google) powoduje, że każdy z producentów może stosować własną politykę dotyczącą m.in. sterowników audio, kolejkowania procesów czy zarządzania mocą co bezpośrednio wpływa na różnicę w wydajności działania programów narażonych na występowanie opóźnienia audio. Mimo to, brak możliwości uzyskania czasu przejścia sygnału dźwiękowego przez system na poziomie kilku milisekund można uznać za spore niedociągnięcie systemu w tego typu urządzeniach. Stwierdzenie to potęguje fakt, że system operacyjny największego rywala Google, Apple IOS pozbawiony jest tej wady i w tematyce opóźnienia audio może być traktowany jako wzór [19].

Opóźnienie na poziomie 150 ms jest zależne od wielu czynników, na które programista projektujący aplikację często nie ma żadnego wpływu. Narzędzia oddane w ręce twórców aplikacji kompatybilnych z systemem Android pozwalają jednak na osiągnięcie opóźnienia skutecznie realizującego założenia projektowe i tym samym zakłócającego mowę. Co więcej dają one możliwość implementacji szeregu funkcjonalności związanych z modulacją odtwarzanego głosu. W aplikacji Speech Jammer udało się zaprojektować korektor dźwięku oraz mechanizm regulujący tempo odtwarzanej mowy. Jak wykazały przeprowadzone badania, funkcja przyspieszająca odtwarzany dźwięk miała znaczący wpływ na skuteczność zakłócania mowy i zwiększyły efektywność działania algorytmu niemal o 15%. Z ogólnego punktu widzenia każdorazowe działanie aplikacji Speech Jammer prawie dwukrotnie wydłużało czas potrzebny na przeczytanie przykładowego tekstu w porównaniu do czytania go w warunkach kompletnej ciszy.

Dalsze prace nad aplikacją w pierwszej kolejności powinny uwzględniać implementację algorytmów kasujących echo, tak aby do poprawnego działania programu nie wymagane było użycie słuchawek. Wstępne badania w tym obszarze dowiodły, że najlepszym narzędziem do zrealizowania tego celu będzie natywna biblioteka Speex, skonstruowana specjalnie na potrzeby analizy mowy w systemach wbudowanych [20].

Podsumowując, stworzona aplikacja do zakłócania mowy może być świetną baza do testowania możliwości audio urządzeń mobilnych wyposażonych w system Android. Przewiduje się, że w niedalekiej przyszłości Google upora się z problemem opóźnienia audio w swoim systemie, co da podstawę do tworzenia coraz bardziej zaawansowanych aplikacji z dziedziny przetwarzania dźwięku.

Bibliografia

[1] Delayed Auditory Feedback, http://www.stutteringhelp.org/delayed-auditory-feedback, *stan na dzień: 20.12.2013.*

[2] Speech Jammer: A System Utilizing Artificial Speech Disturbance with Delayed Auditory Feedback, https://sites.google.com/site/qurihara/top-english/speechjammer, *stan na dzień: 20.12.2013.*

[3] Voice Jammer – Android Apps on Google Play, https://play.google.com/store/apps/details?id=com.dreigon.voicejammer&hl=en, *stan na dzień: 20.12.2013.*

[4] Google I/O 2013 – High Performance Audio, http://www.youtube.com/watch?v=d3kfEeMZ65c, *stan na dzień: 20.12.2013.*

[5] Audio Latency – Android Developers, http://source.android.com/devices/audio_latency.html, *stan na dzień: 20.12.2013.*

[6] Architektura Audio – Android Developers, http://source.android.com/devices/audio.html, *stan na dzień: 20.12.2013.*

[7] SoundPool – Android Developers, http://developer.android.com/reference/android/media/SoundPool.html, *stan na dzień: 20.12.2013.*

[8] MediaPlayer – Android Developers, http://developer.android.com/reference/android/media/MediaPlayer.html, *stan na dzień: 20.12.2013.*

[9] AudioTrack – Android Developers, http://developer.android.com/reference/android/media/AudioTrack.html, *stan na dzień: 20.12.2013.*

[10] AudioRecord – Android Developers, http://developer.android.com/reference/android/media/AudioRecord.html, *stan na dzień: 20.12.2013.*

[11] Process – Android Developer, http://developer.android.com/reference/android/os/Process.html#THREAD_PRIO

RITY_URGENT_AUDIO, *stan na dzień: 20.12.2013*

[12] Khronos Group, *OpenSL ES Specification Version 1.0.1*, 2009.

[13] Ratabouil S., *Android NDK Beginner's Guide*, Packt Publishing, Birmingam, 2012.

[14] Simplified Wrapper and Interface Generator, http://www.swig.org/, *stan na dzień: 20.12.2013.*

[15] OpenSL ES for Android, http://mobilepearls.com/labs/native-android-api/ndk/docs/opensles/index.html, *stan na dzień: 20.12.2013.*

[16] The Audio Programming Blog, http://audioprograming.wordpress.com/, *stan na dzień: 20.12.2013*

[17] PackageManager – Android Developers, http://developer.android.com/reference/android/content/pm/PackageManager.html, *stan na dzień: 20.12.2013*

[18] AudioManager – Android Developers, http://developer.android.com/reference/android/media/AudioManager.html, *stan na dzień: 20.12.2013.*

[19] Android, High Performance Audio in 4.1 and What it Means, http://createdigitalmusic.com/2012/07/android-high-performance-audio-in-4-1-and-what-it-means-plus-libpd-goodness-today/, *stan na dzień: 20.12.2013*

[20] Jean-Marc Valin, *The Speex Codec Manual Version 1.2 Beta 3*, 2007

Dodatek A.

Spis zawartości dołączonej płyty CD

Praca

paweł_helm_242608.doc – Tekst pracy zapisany w formacie MS Word,

paweł_helm_242608.pdf – Tekst pracy zapisany w formacie .pdf,

Speech Jammer – Katalog projektowy aplikacji zakłócającej mowę.