Politechnika Łódzka
Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki
Instytut Informatyki Stosowanej

PRACA INŻYNIERSKA

Mobilna aplikacja do montażu i miksu dźwięku z opcją metronomu

Mobile application for sound editing and mixing with metronome option

Michał Nawrot
Numer albumu: 209405

Promotor pracy:
dr inż. Tomasz Kowalski

Łódź, Luty, 2020

Streszczenie

Celem pracy było opracowanie aplikacji mobilnej umożliwiającej łączenie ścieżek dźwiękowych w celu jednoczesnego odtwarzania. Dodatkowo aplikacja umożliwia dodanie do łączonych ścieżek sygnału metronomu z uwzględnieniem akcentów. Wygenerowane ścieżki zapisywane są do pliku w tworzonym przy pierwszym uruchomieniu przez aplikację folderze. Aplikacja umożliwia również uruchomienie oddzielnie metronomu, który swój stan może sygnalizować dźwiękiem, wibracjami i zmianą koloru tła. Postawione cele zostały zrealizowane, a aplikacja została przetestowana pod kątem funkcjonalności oraz oceniona przez użytkowników w różnych kategoriach. W pracy zawarto informacje o użytych narzędziach i technologiach oraz przybliżono teoretyczne zagadnienia związane z przetwarzaniem dźwięku.

Słowa kluczowe: Próbkowanie sygnału audio, Android, Unity, metronom, formaty plików audio.

Abstract

The aim of this work was to develop a mobile application that allows you to combine sound tracks for simultaneous playback. In addition, the application allows you to add the metronome signal to the combined paths including accents. The generated audio tracks are saved to a file in the folder created when the application is run for the first time. The application also allows you to run the metronome separately, which can signal its status not only with sound but also with vibration and by changing background color. All set goals have been achieved and the application has been tested for functionality and rated by users in different categories. The work contains information on the tools and technologies used, and introduces theoretical issues related to sound processing.

Keywords: Audio sampling, metronome, Android, Unity, audio file formats.

Spis Treści

1. Wstęp 4
1.1. Cel i zakres prac 4
2. Teoretyczne zagadnienia dźwięku 6
2.1. Fale w fizyce [1] 6
2.1.1. Rodzaje fal [2] 7
2.1.2. Fale akustyczne [3, 4] 8
2.2. Muzyka i metronom [3] 10
2.3. Reprezentacja dźwięku w komputerze [7, 8] 12
2.3.1. Format pliku WAV [9] 14
2.3.2. Różnice między popularnymi formatami plików [10] 15
3. Środowisko sprzętowe i programistyczne 17
3.1. Mobilne systemy operacyjne 17
3.2. System Android 18
3.3. Zintegrowane środowisko programistyczne Visual Studio 20
3.4. Silnik Unity 21
3.4.1. Wykorzystane komponenty 22
3.4.2. Tworzenie pliku APK 28
3.5. System kontroli wersji GIT [17] 30
4. Projekt 33
4.1. Założenia projektowe 33
4.2. Warstwa graficzna 33
5. Sposób działania aplikacji 34
5.1. Funkcja łączenia ścieżek 36
5.2. Funkcja metronomu 45
5.3. Funkcja odtwarzacza 47
6. Weryfikacja działania aplikacji 49
6.1. Testy funkcji łączenia ścieżek 49
6.2. Testy na różnych modelach telefonów i oceny użytkowników 50
7. Podsumowanie i wnioski 52
8. Literatura 53
9. Wykaz rysunków 55
10. Wykaz tabel 57
11. Wykaz listingów 58

1. Wstęp

Celem pracy było opracowanie aplikacji mobilnej umożliwiającej łączenie ścieżek dźwiękowych odczytanych z plików WAV (ang. *wave form audio format*) i OGG (ang. O*gg vorbis audio format*) z możliwością połączenia ich z dopasowanym sygnałem metronomu i zapisaniem ich do pliku WAV. Praca obejmuje między innymi takie zagadnienia jak obiektowe języki programowania, przetwarzanie dźwięku, komunikacja człowiek komputer i platformy mobilne. W części teoretycznej przybliżono zagadnienia fal akustycznych, próbkowania sygnału audio oraz określenia istoty dźwięku i metronomu. Oprócz tego zawarto informacje o różnych formatach plików audio i wyszczególniono różnice między nimi, a także opisano system urządzeń mobilnych Android, wraz z jego komponentami oraz wykorzystane zintegrowane środowisko programistyczne Visual Studio. Ponadto zawarto opis silnika do tworzenia gier Unity3D, który został wykorzystany przy tworzeniu i projektowaniu aplikacji, wraz z opisem wykorzystanych komponentów oraz opisano działanie i założenia systemu kontroli wersji GIT, z ukazaniem różnic między nim, a poprzednimi systemami kontroli wersji. W części praktycznej przybliżono założenia projektowe, opisano warstwę graficzną aplikacji, napisano w jaki sposób wykorzystano poszczególne metody klas, które umożliwiały wypełnienie założeń projektu oraz opisano sposób w jaki działają. Co więcej, zweryfikowano działanie poszczególnych funkcji aplikacji oraz zawarto oceny użytkowników aplikacji, którzy testowali ją pod wyszczególnionymi względami i na tej podstawie wyciągnięto wnioski co do przyszłego rozwoju aplikacji.

1.1. Cel i zakres prac

Jednym z głównych elementów rozwoju techniki jest dążenie ludzi do poprawy jakości życia. Z biegiem czasu zaczęto porzucać nieporęczne narzędzia na rzecz bardziej przemyślanych i kompaktowych rozwiązań. Konsekwencją tego może być fakt, że coraz więcej osób porzuca komputery osobiste na rzecz smartfonów, aby wykonać proste czynności typu przeglądanie poczty elektronicznej, zrobienie przelewu bankowego lub napisanie wiadomości. Również rynek konsol do gier, które przez wiele osób są utożsamiane z rozrywką domową, idzie w kierunku miniaturyzacji i mobilności, przez co na rynku występuje dużo konsol przenośnych, a sukces konsoli Nintendo Switch tylko potwierdza coraz większe zainteresowanie urządzeniami mobilnymi wśród użytkowników. Wraz z rozwojem mobilnych urządzeń elektronicznych rozwijają się również systemy na platformy

mobilne, dzięki czemu użytkownicy mogą wykonywać coraz bardziej skomplikowane operacje.

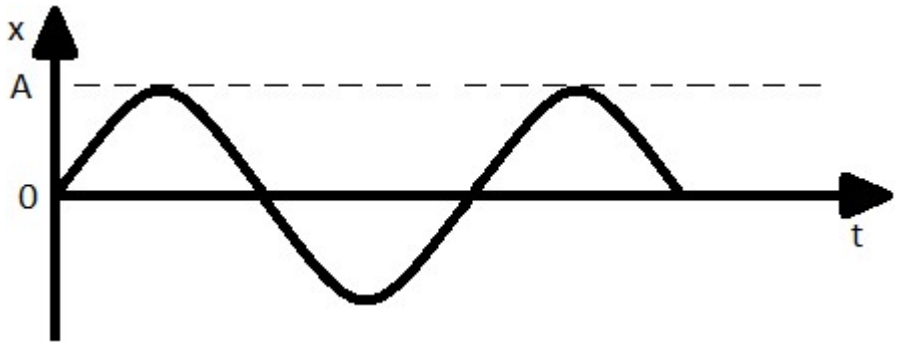
Z powyższych powodów w niniejszej pracy postanowiono opracować aplikację na urządzenia z systemem Android, które są najliczniejszą grupą urządzeń wśród smartfonów. Aplikacja głównie skierowana jest do osób grających na różnych instrumentach muzycznych oraz dla osób prowadzących zespoły muzyczne. Program ma za zadanie ułatwić ćwiczenie umiejętności poprawnego grania na instrumentach oraz usprawnić organizację związaną z tworzeniem podkładów do ćwiczeń.

2. Teoretyczne zagadnienia dźwięku

Dźwięk jest wrażeniem słuchowym, spowodowanym rozchodzącą się w ośrodku sprężystym falą akustyczną, a nauką o dźwięku jest akustyka. Akustyka jako dział fizyki i techniki zajmuje się zjawiskami związanymi z powstawaniem, rozchodzeniem się oraz oddziaływaniem fal akustycznych. Uznawana jest za naukę interdyscyplinarną.

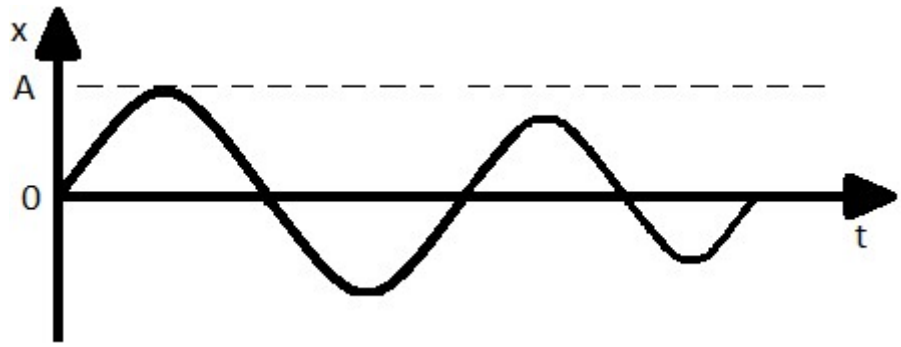
2.1. Fale w fizyce [1]

Fala to zjawisko transportu energii ze skończoną prędkością bez przeniesienia materii. Jest zaburzeniem rozprzestrzeniającym się w ośrodku, nie wpływającym na jego pierwotny stan. Fale mogą być różne w zależności od ośrodka, w którym występują. Mogą wpływać na gęstość gazów, cieczy i ciał stałych oraz kształt powierzchni swobodnej cieczy. Częstotliwością drgań nazywamy częstotliwość, z jaką drga ciało bez wpływu sił oporów będąc wytrącone z położenia równowagi. Amplituda drgań, czyli maksymalne wychylenie z położenia równowagi, pozostaje stała, ponieważ drgania nie wyciszają się. Zależność wychylenia od czasu przedstawiono na poniższym wykresie (Rysunek 1).



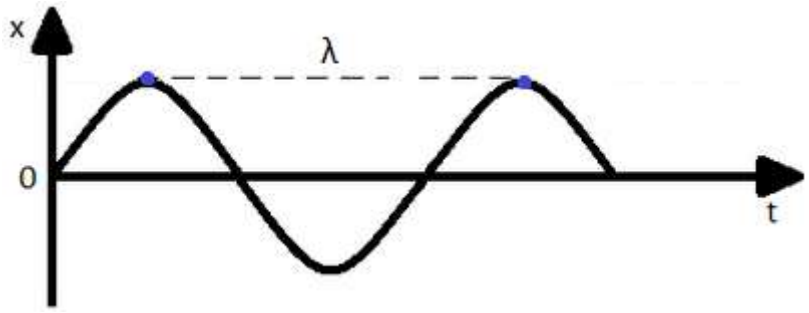
Rysunek 1: Stała amplituda - fala bez sił oporu (opracowanie własne)

W warunkach z występującymi oporami ruchu, drgania wygaszają się - amplituda drgań wraz z upływem czasu zmniejsza się (Rysunek 2).



Rysunek 2: Zmiana amplitudy - fala z siłami oporu (opracowanie własne)

Kolejnym atrybutem fali jest długość oznaczana przez λ (Rysunek 3). Określa odległość, między punktami będącymi w tej samej fazie drgań, a czas potrzebny na pokonaniu tej odległości nazywa się okresem fali oznaczony jako T.



Rysunek 3: Długość fali (opracowanie własne)

Fala rozchodzi się ze stałą prędkością, którą opisuje wzór:

$$V = \frac{\lambda}{T}$$

który można zapisać z pomocą częstotliwości f:

$$V = \lambda \times f$$

gdzie T=1/f.

2.1.1. Rodzaje fal [2]

Wszystkie fale są podobne z natury, ponieważ sposób ich powstawania oraz rozchodzenia się jest identyczny niezależnie od częstotliwości, jednakże proces rozchodzenia się fal może być różny zależnie od długości fal. Fale można podzielić na mechaniczne i elektromagnetyczne. Fala mechaniczna potrzebuje materialnego ośrodka sprężystego, w którym będzie mogła się rozchodzić, a fala elektromagnetyczna może rozchodzić się nie

tylko w ośrodku materialnym, ale również w próżni. Fale te można podzielić w zależności od kierunku drgań cząsteczek:

- fale poprzeczne,
- fale podłużne

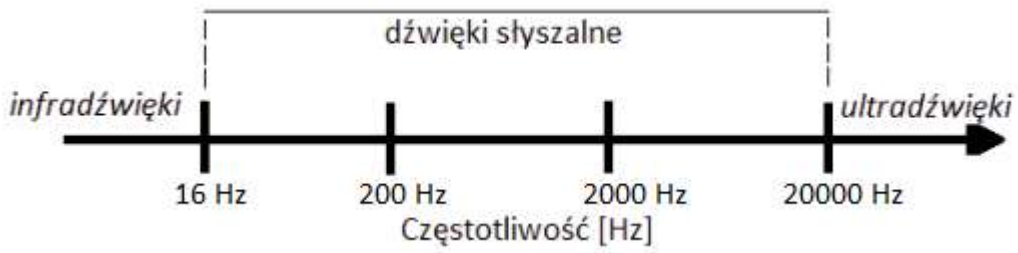
lub w zależności od ich kształtu:

- kuliste,
- kołowe,
- płaskie.

Kierunek drgań w falach poprzecznych jest prostopadły do ich kierunku rozchodzenia się (np. fale elektromagnetyczne), za to w falach podłużnych kierunek drgań jest zgodny do kierunku ich propagacji (np. fale akustyczne). Szczególnym przypadkiem fal są fale kuliste, które zawierają w sobie drgania poprzeczne i podłużne.

2.1.2. Fale akustyczne [3, 4]

Fala akustyczna, inaczej dźwiękowa, jest falą mechaniczną kulistą rozchodzącą się we wszystkich kierunkach. Prędkość fali akustycznej zależna jest od gęstości ośrodka. Słyszalne przez człowieka fale akustyczne posiadają częstotliwości między 16 Hz a 20000 Hz. Dźwięki są rozróżnialne dzięki brzmieniu (zależne od składu widmowego fali), wysokości tonu (im wyższa częstotliwość tym wyższy ton) i głośności (im większe natężenie, czyli amplituda fali tym większa głośność). Fale o częstotliwościach mniejszych niż 16 Hz nazywa się infradźwiękami, a o większych niż 20000 Hz ultradźwiękami (Rysunek 4).



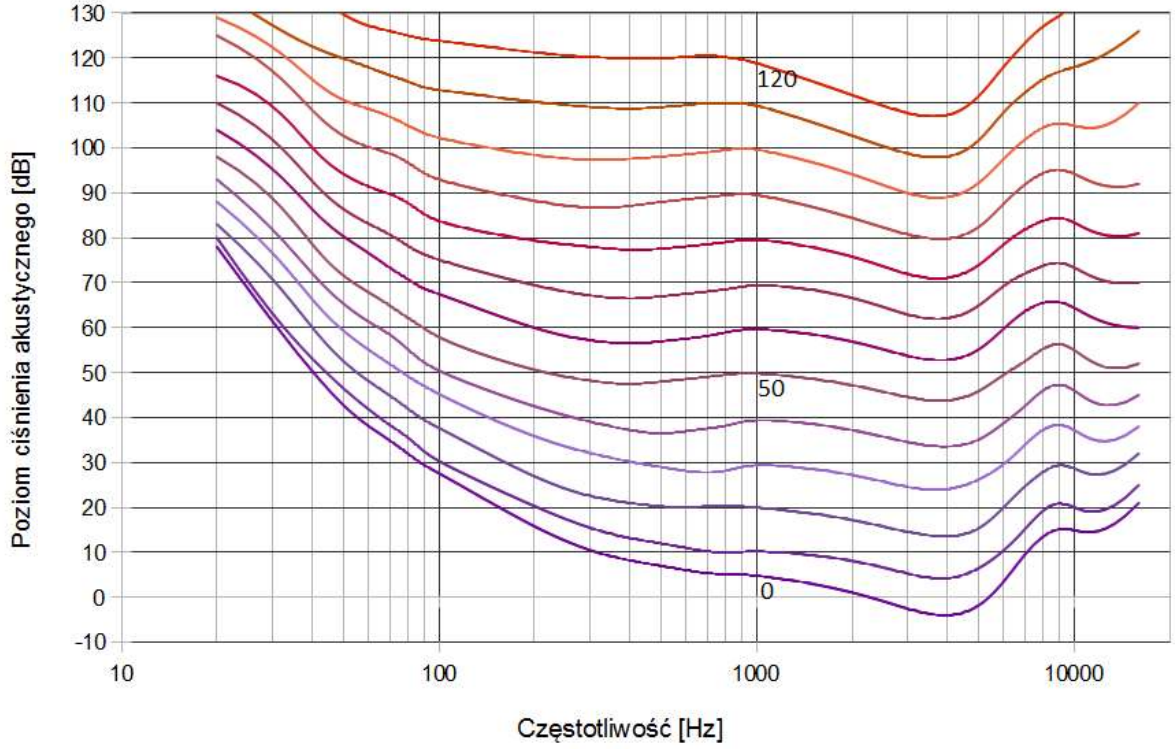
Rysunek 4: Rozkład dźwięków w zależności od częstotliwości (opracowanie własne)

Natężenie dźwięku wyraża się za pomocą decybeli (dB). Aby obliczyć natężenie używa się wzoru:

$$L = 10\ log\frac{I}{I_0}$$

gdzie L to poziom natężenia dźwięku, I to natężenie dźwięku docierającego do punktu pomiaru, a I_0 to najmniejsze natężenie dźwięku jakie jest wstanie odebrać ludzkie ucho, które jest równe $10^{-12}\frac{W}{m^2}$.

Jeśli dźwięk osiągnie dostatecznie wysokie natężenie, fala akustyczna przestaje być odbierana jako dźwięk, w zamian tego wywołuje ból. W zależności od organizmu minimalna wartość natężenia, nazywana progiem bólu jest różna, jednakże przyjmuje się, że wartość minimalna wynosi $\frac{W}{m^2}$ (Wat na metr kwadratowy), czyli około 120dB. Czułość ludzkiego ucha w zależności od częstotliwości i natężenia dźwięku przedstawia wykres (Rysunek 5):



Rysunek 5: Krzywe fonów [5]

W sytuacji kiedy dwa dźwięki mają różną częstotliwość, a takie samo natężenie, ludzkie ucho odbiera je jako różniące się głośnością, dlatego wprowadzono jednostki głośności wyrażające wrażenie fizjologiczne nazywane fonami. Przyjęto, że dźwięk wzorcowy (czyli dźwięk, którego częstotliwość jest równa 1000 Hz) o określonym poziomie natężenia ma tyle samo decybeli i fonów. Z tego wynika, że dany dźwięk będzie posiadał ilość fonów równą dźwiękowi wzorcowemu, jeżeli będzie miał taką samą częstotliwość niezależnie od natężenia. Natomiast dźwięki z różnymi częstotliwościami, mimo jednakowego poziomu natężenia, będą

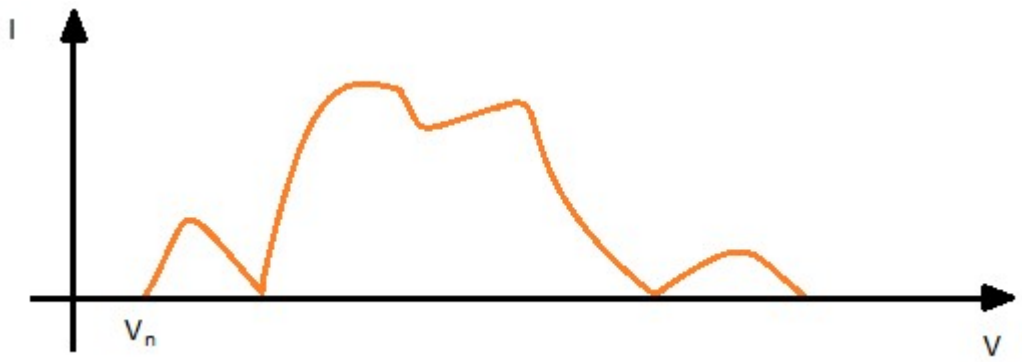
różniły się głośnością. Zatem fony odzwierciedlają wrażliwość ludzkiego ucha na dźwięki o różnych częstotliwościach.

2.2. Muzyka i metronom [3]

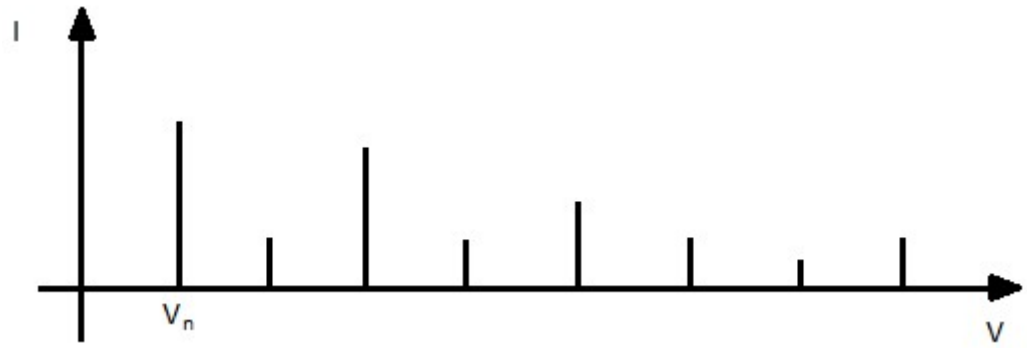
Każde drganie można rozbić na drgania harmoniczne proste różniące się częstotliwościami drgań i amplitudami. Widmem drgania złożonego nazywamy wykres amplitud drgań składowych i w zależności od charakteru widma, dźwięki można podzielić na:

1. Dźwięki o widmie dyskretnym:
   - proste – dźwięki, których częstotliwość jest ściśle określona,
   - harmoniczne – dźwięki złożone z wielu tonów prostych. Ich widmo posiada charakter okresowy. Ich częstotliwość jest zgodna z częstotliwością tonu podstawowego,
2. Dźwięki o widmie ciągłym:
   - szum – dźwięk, którego widmo nie posiada wyraźnych maksimów (Rysunek 6),
   - hałas – dźwięk o nadmiernym natężeniu.

Dźwięki tonalne to dźwięki posiadające widmo liniowe i mogą być wydawane przez różne instrumenty muzyczne i człowieka (Rysunek 7). Można je wyróżnić za pomocą wysokości i barwy. Wysokości dźwięku odpowiada jego częstotliwość, a barwie zawartość widma. Tonem prostym nazywa się dźwięk, którego widmo składa się z pojedynczej linii i może zostać przedstawiony za pomocą funkcji sinusoidalnej. Na obu wymienionych rysunkach ‘I’ oznacza wysokość dźwięku, a ‘V’ częstotliwość dźwięku.



Rysunek 6: Widmo szumu (opracowanie własne)



Rysunek 7: Widma liniowe dźwięku tonalnego (opracowanie własne)

Muzyką nazywamy zorganizowane w czasie fale akustyczne o dobranych częstotliwościach i amplitudach. Używana jest do samoekspresji oraz wywołania w osobach słuchających danych odczuć. Elementami muzyki są: melodyka, rytmika, dynamika, tempo, artykulacja, harmonika i kolorystyka. Melodyka odpowiada organizacji dźwięków o różnych wysokościach i czasie trwania. Rytmika ustala czas trwania dźwięków oraz relacje czasowe pomiędzy każdym dźwiękiem. Harmonika określa przebieg łączenia kilku dźwięków o różnych tonach (określa skalę dźwiękową używaną w utworze). Dynamika określa natężenie dźwięku. Tempo to składowa mówiąca z jaką prędkością powinny następować dźwięki określane przez rytmikę. Artykulacja określa w jaki sposób powinien zostać wydobyty dźwięk. Kolorystyka odpowiada barwie dźwięku. Każdy instrument muzyczny odznacza się własną barwą, dzięki czemu można tworzyć kompozycje, które zagrane przy użyciu różnych instrumentów będą brzmiały inaczej.

Tempo może być wyznaczane przez metronom (Rysunek 8), czyli urządzenie do dokładnego wyznaczania czasu między dźwiękami za pomocą drgań, dźwięku oraz zmian koloru w stałym czasie. Często używany podczas ćwiczeń na instrumentach oraz podczas sesji nagraniowych. Tradycyjne metronomy mechaniczne posiadają skalę od 40 do 208 uderzeń na minutę, a metronomy elektroniczne mogą posiadać skalę od 1 do 500 uderzeń na minutę (w zależności od producenta). Kolejną przewagą elektronicznych metronomów jest możliwość akcentowania poszczególnych uderzeń, ustawianie metrum muzycznego (układ akcentów w najmniejszym odcinku utworu muzycznego), głośności oraz wibracji.

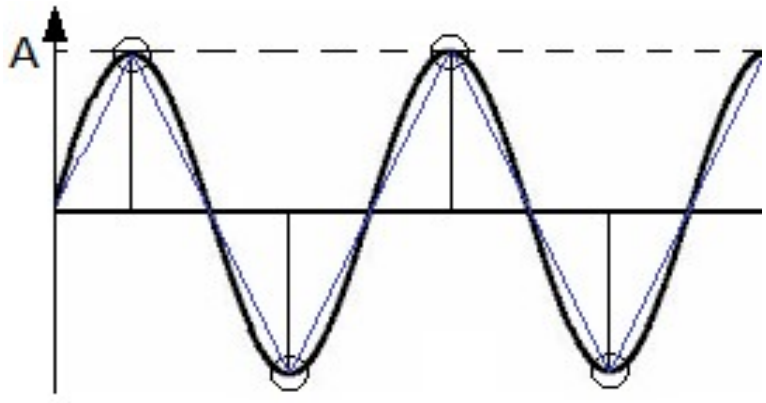
Rysunek 8: Metronom elektroniczny Boss DB-30 oraz metronom mechaniczny Cherub WSM 330 [6]

2.3. Reprezentacja dźwięku w komputerze [7, 8]

Wraz z końcem lat osiemdziesiątych XX wieku dla komputerów IBM wyprodukowane zostały pierwsze karty dźwiękowe będące oparte na kodowaniu PCM (ang. *Pulse-Code Modulation)* i formacie WAV. Aby poprawnie zaprezentować dźwięk na urządzeniu cyfrowym wykorzystuje się metodę nazywaną próbkowaniem sygnału PCM. Polega ona na odczytywaniu sygnału akustycznego w danym momencie i zapisanie go jako liczby binarnej. Na jakość konwersji wartości amplitudy sygnału analogowego do postaci liczbowej mają wpływ:

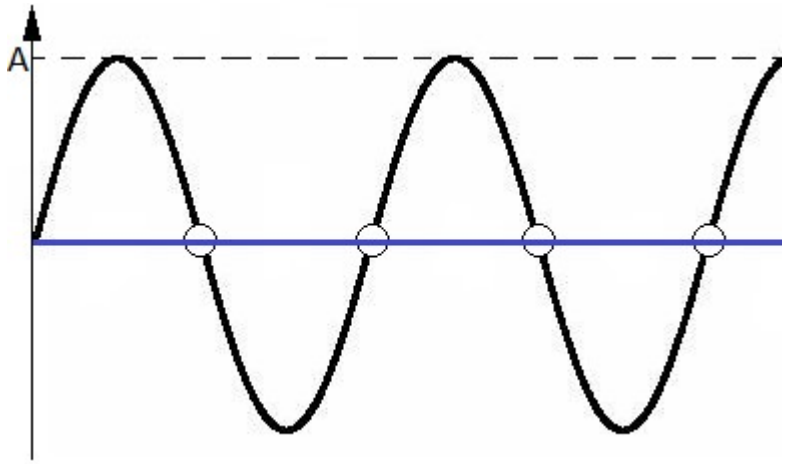
- częstotliwość próbkowania, czyli liczba próbek pobierana w ciągu jednej sekundy. Standardowa częstotliwość próbkowania sygnału dla płyt CD-Audio wynosi 44100Hz.
- rozdzielczość próbkowania, czyli liczba bitów przeznaczonych do zapisu każdej próbki. Ścieżka na płycie CD-Audio posiada 16-bitową rozdzielczością.

W zależności od częstotliwości można otrzymać mniej lub bardziej dokładne odwzorowanie fali akustycznej. Dlatego dla najlepszego efektu wykorzystuje się częstotliwość Nyquist-a, czyli maksymalną częstotliwość składowych widmowych sygnału, które mogą później zostać odtworzone bez zniekształceń. Przy wyższych wartościach częstotliwości, składowe widmowe ulegają zniekształceniu przez nakładanie się na składowe o innych częstotliwościach, przez co nie można ich odtworzyć w poprawny sposób. Dla fali sinusoidalnej można w dosyć dobry sposób uzyskać sygnał, który jest zbliżony do pierwowzoru (Rysunek 9):



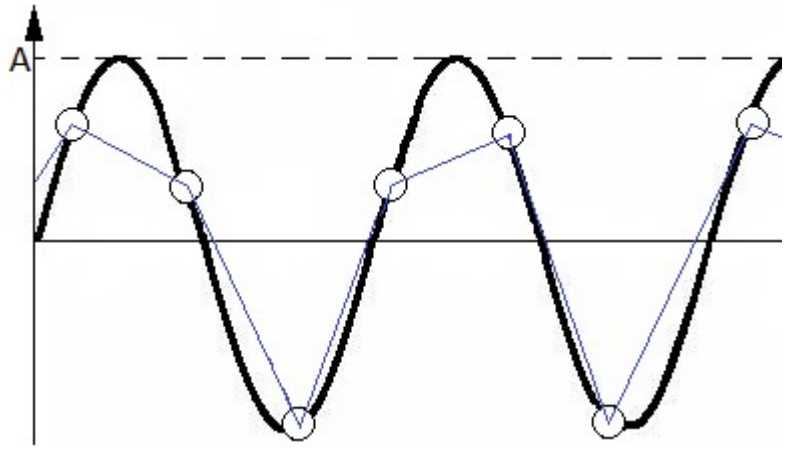
Rysunek 9: Sinusoida z punktami próbkowania na maksymalnych wychyleniach (opracowanie własne)

Jednak jeśli pozostaniemy przy czterech punktach próbkowania, a one rozłożą się na miejsca gdzie amplituda jest równa zero, wtedy nasz sygnał zaniknie (Rysunek 10):



Rysunek 10: Sinusoida z punktami próbkowania na miejscach zerowego wychylenia (opracowanie własne)

Możliwa jest również sytuacja, kiedy punkty próbkowania rozłożą się w jeszcze innej pozycji niż maksymalna lub zerowa amplituda. Daną sytuację prezentuje wykres (Rysunek 11):



Rysunek 11: Sinusoida z punktami próbkowania na niezerowych wychyleniach nie będącymi maksimami (opracowanie własne)

Jak widać na wykresie, z samego próbkowania pojawiły się dwa sygnały z mniejszą i większą amplitudą, dlatego nie zapisuje się kształtu zbudowanego z sygnałów, a tylko ich występowanie, gdyż może dochodzić do zapisu nieistniejących w oryginale dźwięków. Mogą powstawać szumy próbkowania, pojawić się nowe dźwięki lub oryginalne dźwięki mogą zostać pominięte. Część z tych efektów jest możliwa do załagodzenia przez mechanizmy antyaliasingowe (techniki zmniejszania liczby błędów zniekształceń) lub zmniejszenie pasma zapisywanych częstotliwości.

2.3.1. Format pliku WAV [9]

Nieskompresowany dźwięk PCM jest definiowany przez dwa parametry: częstotliwość próbkowania i ilość bitów. Można założyć, że częstotliwość próbkowania ogranicza maksymalną częstotliwość, którą może reprezentować format, a ilość bitowa określa maksymalny zakres dynamiczny, który może być reprezentowany przez format. Ilość bitów można traktować jako określenie ilości szumu w porównaniu do sygnału. Format WAV tworzy nieskompresowane pliki audio, co oznacza, że plik posiadający kilkuminutowe nagranie będzie miało taki sam rozmiar, bez względu na zawartość nagrania. Plik WAV zawiera nagłówek i nieprzetworzone dane w formacie czasu. Nagłówek jest początkiem pliku WAV (RIFF, ang. *Resource Interchange File Format*). Są to 44 bajty ułożone w schemacie przedstawione w tabeli (Tabela 1).

Pozycja	Wartość	Opis
1 - 4	"RIFF"	Oznacza plik jako plik riff. Znaki mają długość 1 bajta.
5 - 8	Rozmiar pliku (liczba całkowita)	Rozmiar całego pliku - 8 bajtów (32-bitowa liczba całkowita). Zazwyczaj wypełniane po utworzeniu.
9 -12	"WAVE"	Nagłówek typu pliku. Dla pliku WAV zawsze "WAVE".
13-16	"fmt "	Znacznik formatu. Obejmuje końcowy null.
17-20	16	Długość danych formatu, podanych wyżej.
21-22	1	Rodzaj formatu (1 to PCM). 2-bajtowa liczba całkowita.
23-24	2	Ilość kanałów. 2-bajtowa liczba całkowita.
25-28	44100	Częstotliwość próbkowania. 32-bajtowa liczba całkowita. Typowe wartości to 44100 (CD), 48000 (DAT). Częstotliwość próbkowania = liczba próbek na sekundę lub herc.
29-32	176400	(Częstotliwość próbkowania * bitów na próbkę * kanały) / 8.

33-34	4	(Bity na próbkę * kanały) / 8. (1 - 8 bitów mono, 2 - 8 bitów stereo lub 16 bitów mono, 4 - 16 bitów stereo)
35-36	16	Ilość bitów na 1 próbkę.
37-40	"data"	Nagłówek sekcji danych. Początek sekcji danych. Zawsze równy “data”.
41-44	Rozmiar danych	Rozmiar sekcji danych.
Przykładowe wartości dla 16-bitowego źródła stereo.		

Tabela 1: Nagłówek pliku WAV

Zaletami formatu WAV są: bezstratna jakość dźwięku, obsługa na wielu platformach, łatwość edycji oraz brak potrzeby kodowania i dekodowania. Program bądź urządzenie służące do zakodowania lub rozkodowania sygnału nazywa się kodekiem. W efekcie zakodowania sygnału powstaje plik dyskowy. Żeby móc odtworzyć dany plik konieczne jest zainstalowanie w systemie tego samego kodeka, który został użyty do zakodowania pliku. Główną wadą formatu WAV jest wielkość pliku, przez co transmisja plików w tym formacie przez Internet jest utrudniona i czasochłonna.

2.3.2. Różnice między popularnymi formatami plików [10]

Bezstratne formaty plików WAV oraz AIFF (ang. *Audio Interchange File Format)* są oparte na PCM i można je konwertować do siebie nawzajem bez strat w jakości. Oba formaty uważane są za nieskompresowane i używane są do miksowania, ponieważ zapewniają najlepszą jakość, jednak kosztem zajmowanej pamięci. Średnio 1 minuta zapisana w formacie WAV lub AIFF zabiera 10 Mb przestrzeni dyskowej. Dlatego jeśli nie ma potrzeby edycji dźwięku, a jakość wciąż jest ważna, dźwięk zapisuje się przy użyciu kodeka FLAC (ang. *Free Lossless Audio Codec*). Jest to format bezstratnej kompresji danych pozwalający w przybliżeniu zaoszczędzić połowę przestrzeni dyskowej, która byłaby potrzebna dla plików WAV i AIFF. Kompresja FLAC polega na predykcji liniowej, czyli zapisywaniu wartości różnicy między rzeczywistą wartością próbki, a obliczoną przewidywaną na podstawie kilku poprzednich próbek wartości, dzięki czemu przy dekompresji uzyska się zgodne z pierwowzorem wartości. Jednak, jeśli najważniejszym kryterium wyboru formatu jest waga pliku, a nie możliwość edycji danych i ich jakość, powinno zastosować się formaty kompresji stratnej. Najpopularniejszymi takimi kodekami są MP3 (ang. *MPEG-1/MPEG-2 Audio*

Layer3), AAC (ang. *Advanced Audio Coding*), WMA (ang. *Windows Media Audio)*. W przeciwieństwie do formatów opartych na PCM i formatach bezstratnej kompresji, informacji utraconych podczas kompresji stratnej nie można już odzyskać. Z tego względu, poprzez miksowanie takich plików, nigdy nie uzyska się takich samych efektów, jak przy miksowaniu plików w formatach opartych na PCM. Przy użyciu kodeka MP3, usuwane są dźwięki niesłyszalne przez człowieka oraz dźwięki występujące w otoczeniu silniejszego sygnału, a rozmiar pliku w porównaniu do pliku w formacie WAV zmniejsza się około dziesięciokrotnie, dlatego pliki MP3 stały się najczęściej przesyłanymi plikami muzycznymi w Internecie. Formaty ACC i WMA zostały opracowane jako usprawnione formaty kompresji stratnej względem MP3. Usprawnienia w formacie ACC to między innymi: lepsze przenoszenie częstotliwości ponad 15 kHz, poprawiony tryb kompresji sygnału stereofonicznego joint-stereo oraz większy zakres próbkowania 8-96 kHz, gdzie dla MP3 zakres ten wynosi 16-48 kHz. Za pozytywny odbiór formatu ACC wśród konsumentów, w głównej mierze odpowiedzialna jest firma Apple, która na tym kodeku oparła sklep z muzyką cyfrową iTunes. Format WMA został opracowany przez firmę Microsoft i zyskał popularność głównie dzięki rozpowszechnieniu go w systemach Windows. W porównaniu do MP3 cechuje się podobną jakością dźwięku przy znacznie mniejszym rozmiarze, dzięki czemu pozwala na wygodne strumieniowanie muzyki w sieci.

3. Środowisko sprzętowe i programistyczne

Jak wspomniano we wstępie, w dzisiejszych czasach urządzenia mobilne potrafią niektórym całkowicie zastąpić komputery osobiste, dlatego w tym rozdziale zostaną przybliżone sformułowania związane z mobilnymi systemami operacyjnymi oraz zostanie wytłumaczone dlaczego jako środowisko uruchomieniowe dla aplikacji wybrano system Android. Środowiskiem programistycznym można nazwać narzędzia wewnątrz, których programista może tworzyć aplikację. W skład środowiska wchodzi zaawansowany edytor kodu, narzędzia do testowania go, a przy większych projektach system kontroli wersji oraz środowiska uruchomieniowe niezbędne do uruchomienia programu.

3.1. Mobilne systemy operacyjne

Według serwisu Statcounter.com w styczniu 2019 roku udziały w rynku platform mobilnych są zdominowane przez dwa systemy operacyjne: Android i iOS, z dużą przewagą tego pierwszego (Rysunek 12). Android posiada 74.3% udziałów w światowym rynku, podczas gdy iOS tylko 24.76%. W przypadku poszczególnych regionów procenty rozkładają się w dany sposób:

Europa: Android 72.58%, iOS 26.77%,

Azja: Android 85.46%, iOS 13.42%,

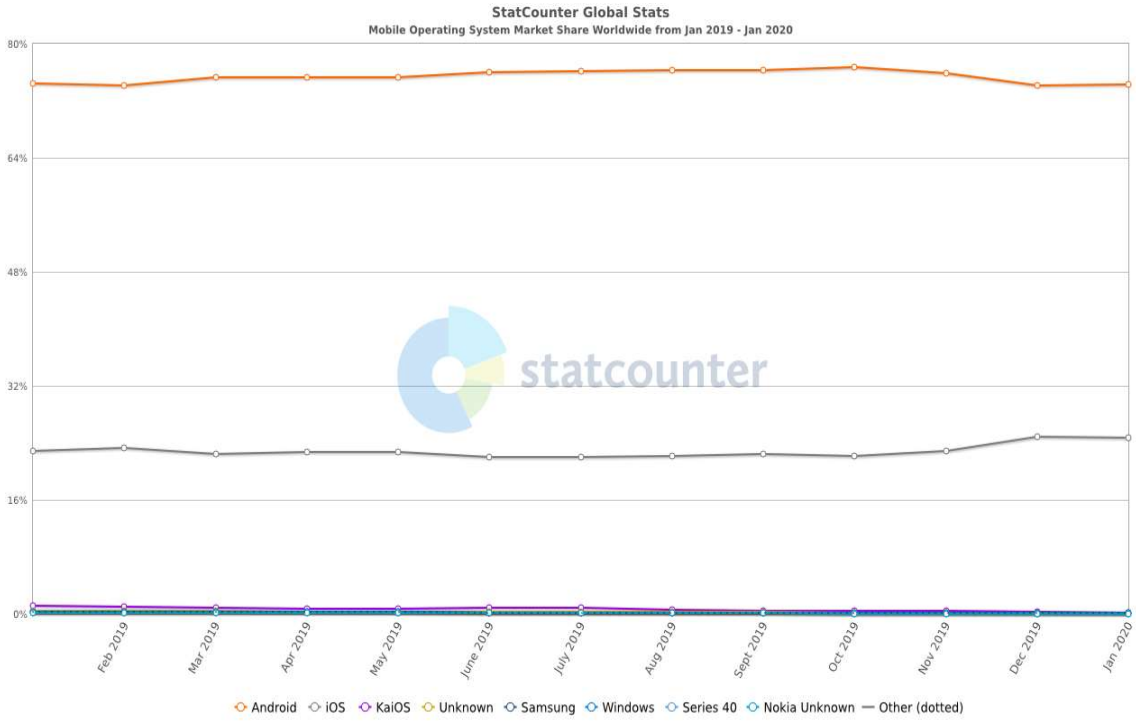
Afryka: Android 84.29%, iOS 11.33%,

Ameryka Południowa: Android 89.62%, iOS 9.76%,

Ameryka Północna: iOS 52%, Android 47.75%,

Oceania: iOS 51.37%, Android 48.3%.

Jak widać w Ameryce Północnej i Oceanii system iOS wyprzedza o kilka procent system Android. Jest to spowodowane tym, że firma Apple, do której należy system iOS, oferuje atrakcyjne ceny swoich produktów w Australii i USA. Za wysoki wynik Androida może odpowiadać fakt, że urządzeń z tym systemem występuje zdecydowanie więcej niż urządzeń z iOS. Wiele osób ceni sobie również możliwość przerabiania systemu pod własne potrzeby oraz możliwość rozszerzenia wewnętrznej pamięci za pomocą pamięci zewnętrznej w postaci kart micro SD. Dlatego ze względu na dużą popularność systemu Android, w niniejszym projekcie inżynierskim powstała aplikacja właśnie na ten system operacyjny.



Rysunek 12: Udział w rynku mobilnych systemów operacyjnych na całym świecie w roku 2019 [11]

3.2. System Android

Android to rozwijany przez sojusz biznesowy Open Handset Alliance system operacyjny na urządzenia mobilne: smartfony, smartwatche, tablety i netbooki. Jego działanie opiera się na jądrze Linuxa, dzięki któremu system może być wielowątkowy, posiadać pamięć wirtualną i łatwo zarządzać pamięcią. Jądro, wraz z niektórymi komponentami, które są częścią systemu Android, opublikowano na licencji GNU GPL, jednakże tym co odróżnia Androida od wielu istniejących dystrybucji Linuxa jest to, że nie zawiera kodu pochodzącego z projektu GNU. Aktualnie dla systemu Android dostępne jest ponad 1.4 miliona aplikacji w sklepie Google Play, co w dużej mierze jest zasługą dużej społeczności deweloperów oraz faktu, że aplikacje można pisać w językach Java, Kotlin, C++. Dzięki SDK (ang. *software development kit*) systemu Android skompilowany kod razem z wymaganymi plikami zostaje zapisany w pliku APK (ang. *Android application package*). Taki plik może zostać użyty przez system Android do zainstalowania w systemie aplikacji, gdyż posiada on wszystkie dane potrzebne do działania aplikacji. W systemie Android istnieje kilka zabezpieczeń związanych z aplikacjami:

- Każda aplikacja działa na własnym procesie. System rozpoczyna proces kiedy aplikacja musi wykonać dane działanie i zamyka go, gdy wszystkie wymagane

działania zostaną wykonane, bądź gdy system musi uzyskać pamięć dla innych procesów.

- Każda aplikacja ma przydzielany identyfikator użytkownika systemu, dzięki czemu system w prosty sposób zarządza dostępami do pamięci.
- Każda aplikacja domyślnie ma przyznawany najmniejszy poziom uprawnień. Zapobiega to sytuacji, kiedy aplikacja miałaby dostęp do komponentów systemu, które zagrażałyby jego bezpieczeństwu. Aplikacja może poprosić użytkownika o przyznanie dodatkowych praw takich jak dostęp do np. pamięci wewnętrznej, kamery, lokalizacji.

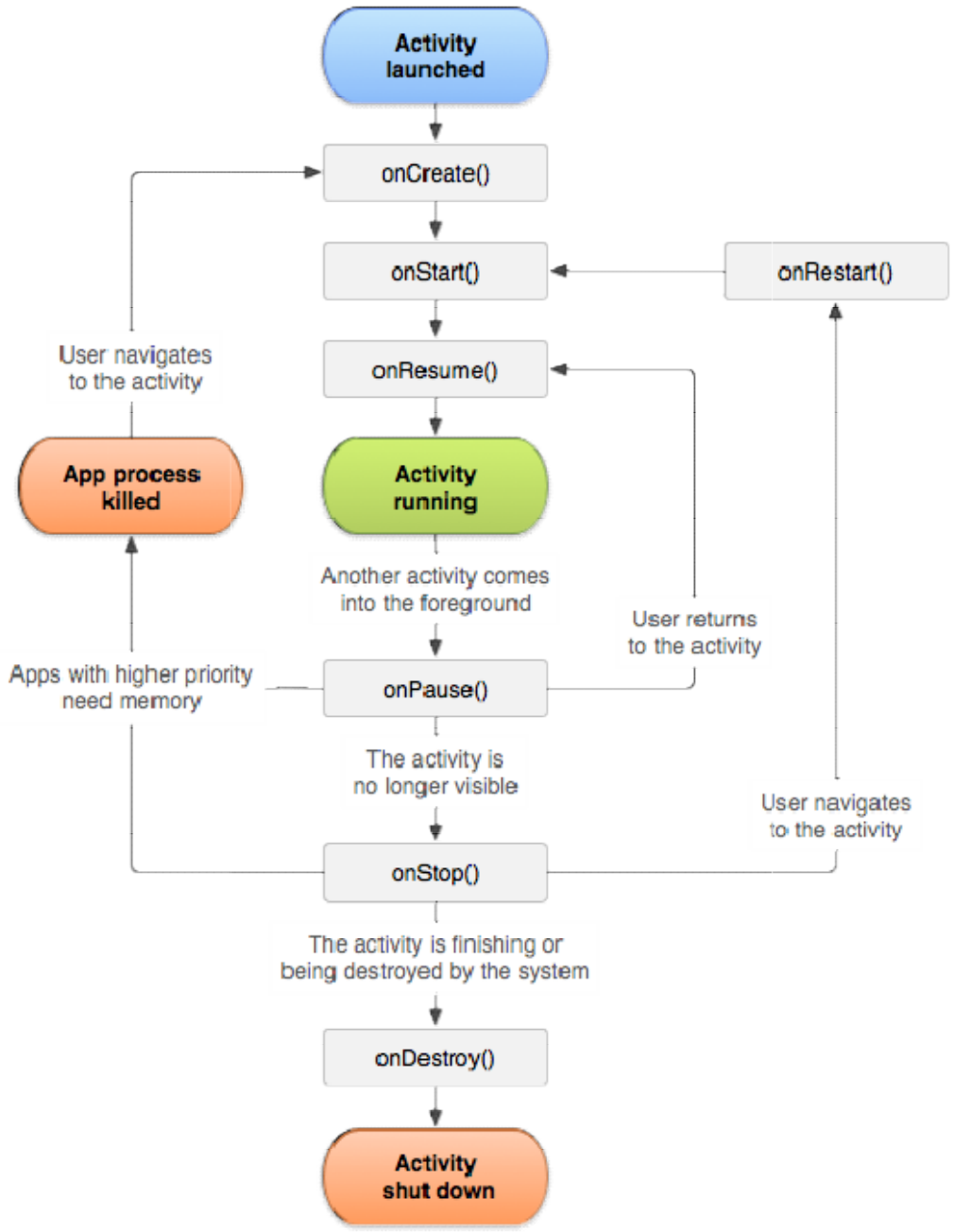
Każda aplikacja jest złożona z komponentów, które są odpowiedzialne za komunikację między użytkownikiem, a aplikacją oraz między aplikacją i systemem. Można wyróżnić cztery typy komponentów: aktywności (ang. *Activities*), serwisy (ang. *Services*), odbiorniki transmisji (ang. *Broadcast Receivers*), dostawcy kontentu (ang. *Content Providers*).

Aktywność [12] odpowiada za interakcję z użytkownikiem i reprezentuje pojedynczy ekran z interfejsem. Każda aktywność podlega cyklowi życia (Rysunek 13). W aplikacji może występować wiele aktywności, które mogą wymieniać się informacjami, dlatego nie są one niszczone natychmiast po przejściu użytkownika z danej aktywności na inną.

Usługi [13] lub inaczej serwisy mają za zadanie przetwarzać zadania, które nie potrzebują lub potrzebują minimalnej ingerencji użytkownika, dlatego działają w tle i nie blokuje interakcji użytkownika z innymi aktywnościami. Usługa działa na głównym wątku aplikacji, dlatego ważne jest, aby odciążyć główny wątek przenosząc działania na oddzielne wątki, które go obciążają. Działająca usługa ma większy priorytet niż nieaktywna aktywność, ponieważ przy zwalnianiu zasobów, w pierwszej kolejności jest nieaktywna aktywność. Cyklem życia usługi zajmują się inne komponenty systemu.

Odbiornik transmisji [14] to składnik, który umożliwia dostarczanie zdarzeń do aplikacji przez system poza działaniem użytkowników, co pozwala aplikacjom reagować na odbierane sygnały. System może dostarczać transmisje nawet do aplikacji, które nie są obecnie uruchomione. Odbiorniki nie posiadają interfejsu użytkownika, jednak mogą tworzyć powiadomienia na pasku stanu systemy, aby przekazać informacje użytkownikowi.

Dostawcy kontentu [15] umożliwiają aplikacjom wyszukiwać i modyfikować dane. W ten sposób aplikacja może zdecydować, w jaki sposób chce mapować dane do przestrzeni nazw URI (ang. *Uniform Resource Identifier*), przekazując je innym aplikacjom lub systemowi.



Rysunek 13: Cykl życia aktywności [16]

3.3. Zintegrowane środowisko programistyczne Visual Studio

Zintegrowane środowiska programistyczne IDE (ang. *Integrated Development Environment*) są edytorem kodu, zawierającym narzędzia do poprawiania jakości kodu (testowanie, znajdowanie błędów i nadmiernego kodu) oraz jego wyglądu, wraz ze środowiskiem uruchomieniowym i często, choć nie zawsze, z możliwą integracją z GIT'em. IDE są bardzo pomocne w pracy programisty, przyspieszają postęp prac oraz pomagają utrzymywać tę pracę na określonym poziomie. Ponadto współczesne IDE posiadają

przeglądarkę klas, obiektów oraz metod, a także udostępniają diagramy hierarchii klas do użytku w programowaniu obiektowym.

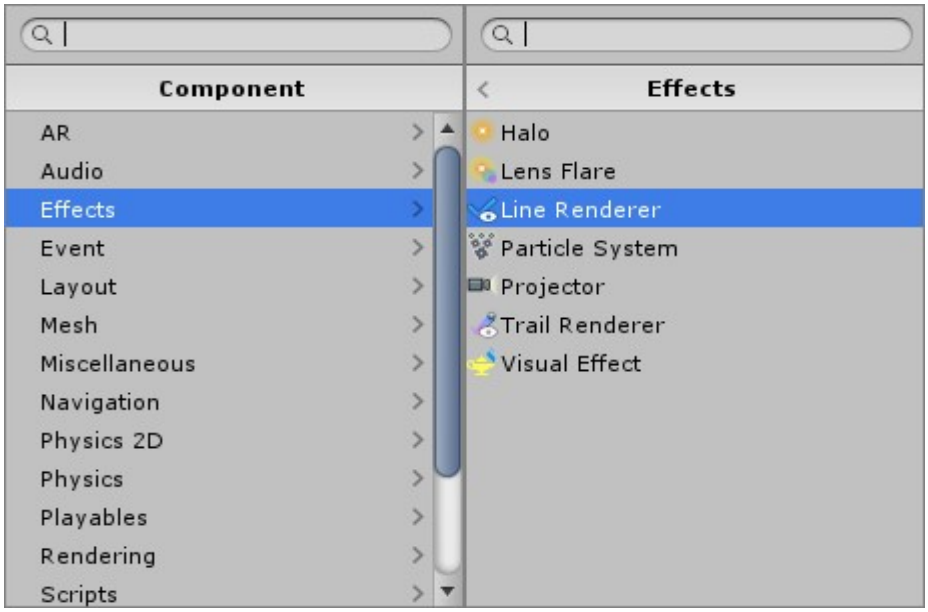
Visual Studio to IDE produkowane przez firmę Microsoft. Uważane jest za jedno z najlepszych zintegrowanych środowisk programowych, używane do tworzenia aplikacji w różnych językach programowania (najczęściej C#, C++, JavaScript ale domyślnie Visual Studio obsługuje więcej języków). Aplikacje mogą być pisane na różne platformy, między innymi: Microsoft Windows, Windows CE, Android, iOS. W każdej kolejnej wersji firma Microsoft wprowadza kolejne ulepszenia i usprawnienia, a w najnowszej wersji 2019 ulepszono wydajność dodając błyskawiczne oczyszczanie kodu i poprawiając wyniki wewnętrznej wyszukiwarki. Poprawiono refaktoryzacje kodu, rozszerzono możliwości kontroli wersji poprzez zwiększenie integracji z GIT'em, a podczas debugowania umożliwiono wyróżnienie, przejście do określonej wartości i optymalizację użycia pamięci. Dodano również funkcję *Live Unit Testing*, która pozwala obserwować wyniki testów jednostkowych bezpośrednio w kodzie.

3.4. Silnik Unity

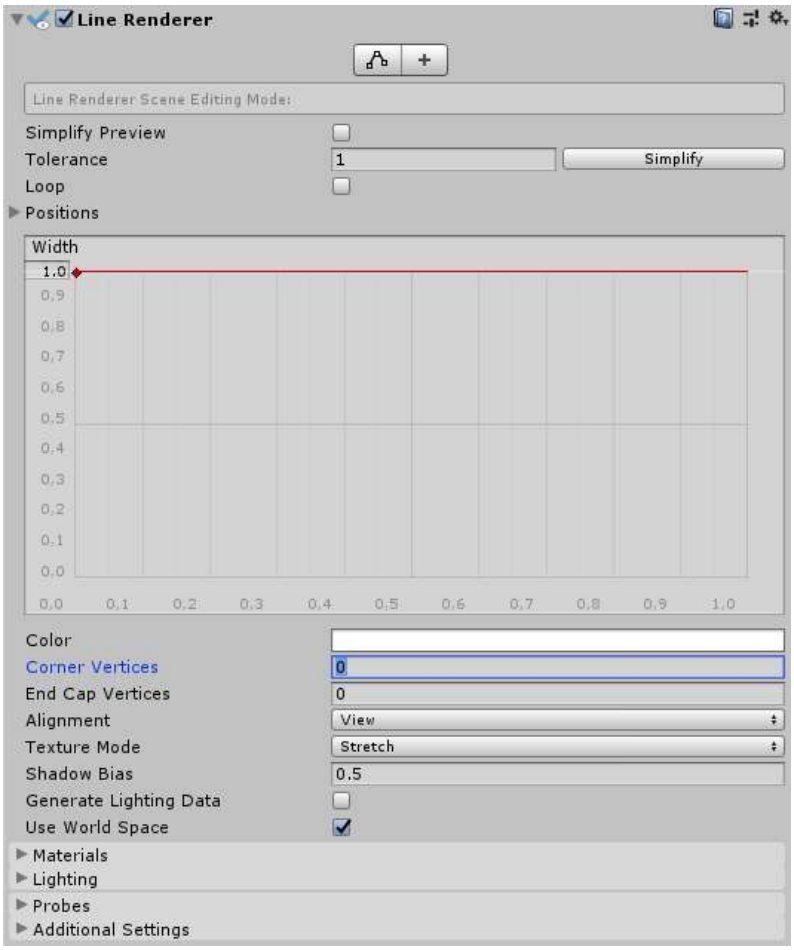
Unity to zintegrowane środowisko działające na systemach Microsoft Windows, macOS oraz Linux, do tworzenia trójwymiarowych oraz dwuwymiarowych gier. Oprócz gier pozwala tworzyć aplikacje na przeglądarki internetowe, komputery osobiste, konsole gier wideo oraz urządzenia mobilne. Umożliwia pisanie skryptów przy użyciu 2 języków programowania: JavaScript, C#. Do wersji 5 silnika korzystać można było jeszcze z języka Boo, a do wersji 2018.2 także przy użyciu UnityScript. Kolejną zmianą, która została wprowadzona wraz z wersją 5, było przejście silnika z licencji płatnej na darmową dla twórców, których dochód nie przekracza 100 tysięcy rocznie. Silnik posiada możliwość zaimportowania bibliotek DLL (ang. *Dynamic-Link Library*), których można używać podczas pisania skryptów. Oprócz tego Unity pozwala także na skorzystanie z *AssetStore*, czyli sklepu internetowego, umożliwiającego pobranie płatnych lub darmowych tekstur, modeli oraz skryptów. Aplikacje tworzone przy użyciu silnika Unity mogą zostać wydane na wiele platform wliczając w nie między innymi systemy Microsoft Windows, macOS, Linux, Android, iOS, konsole PS4, XBOX ONE. Przy pomocy Unity można też tworzyć programy działające w oparciu o hełmy wirtualnej rzeczywistości (np. OculusRift) oraz przeglądarki internetowe (tylko po wcześniejszej instalacji wtyczki Unity Web Player).

3.4.1. Wykorzystane komponenty

Obiekt gry (ang. *GameObject*) to jeden z najważniejszych elementów silnika Unity i każdy z obiektów w tym silniku jest obiektem gry. Do każdego obiektu gry można dodawać różne komponenty w zależności od efektu, który chce się uzyskać. Istnieje możliwość napisania własnych komponentów przy użyciu skryptów. Domyślnie wszystkie obiekty gry posiadają komponent transformacja (ang. *Transform*) odpowiadający za określanie położenia, obrotu i skali danego obiektu na scenie. Komponenty można dodać wybierając je z paska narzędzi (Rysunek 14) lub przesuwając go z okna zasobów (ang. *Assets*) na okno inspektora (ang. *Inspector*). Komponenty są modyfikowalne, a ich właściwości można zmieniać z poziomu okna inspektora (Rysunek 15). Aby usunąć komponent z *obiektu gry* wystarczy kliknąć na niego prawym przyciskiem myszy i z menu kontekstowego wybrać opcję "*Remove Component*". Usunięcia komponentu nie da się cofnąć i wszystkie zmodyfikowane dane zostają usunięte.



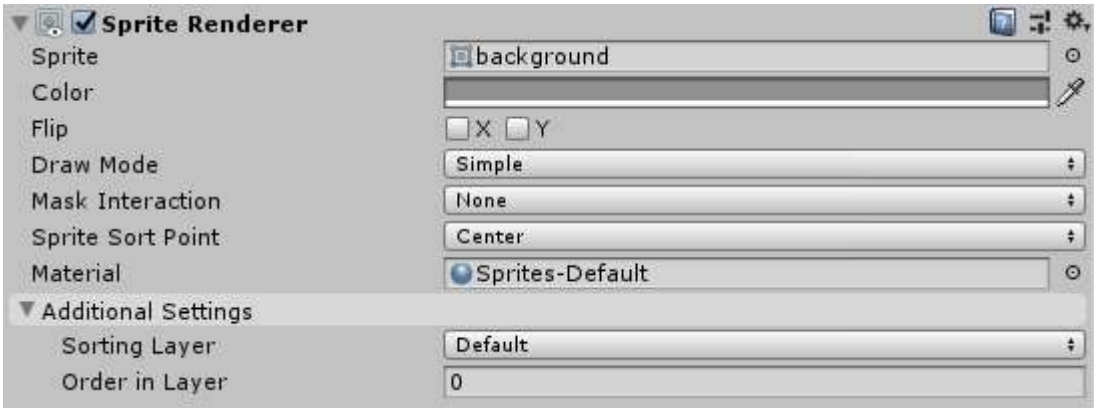
Rysunek 14: Wybór komponentu z paska narzędzi



Rysunek 15: Modyfikowanie właściwości komponentu „*Line Renderer*”

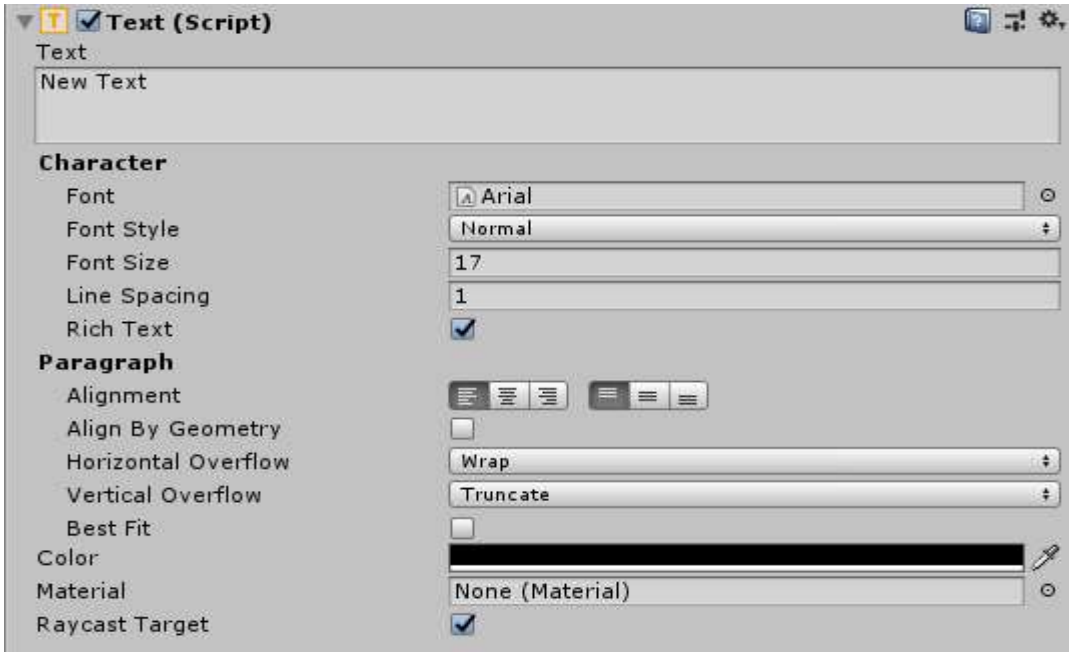
W niniejszym projekcie inżynierskim wykorzystano komponenty: *SpriteRenderer, Text, Button, Audio Listener, AudioSource, Input Field, Dropdown, Slider.*

Renderer Spritów (ang. *SpriteRenderer)* odpowiada za renderowanie sprite’ów oraz kontrolowanie, w jaki sposób ma wyglądać podany mu sprite. W polu *Sprite* podano plik background.png przedstawiający biały prostokąt. Za pomocą atrybutu *Color* zmieniono jego kolor na szary. Ten atrybut został też użyty przy funkcji metronomu opisanej w rozdziale 6.2.



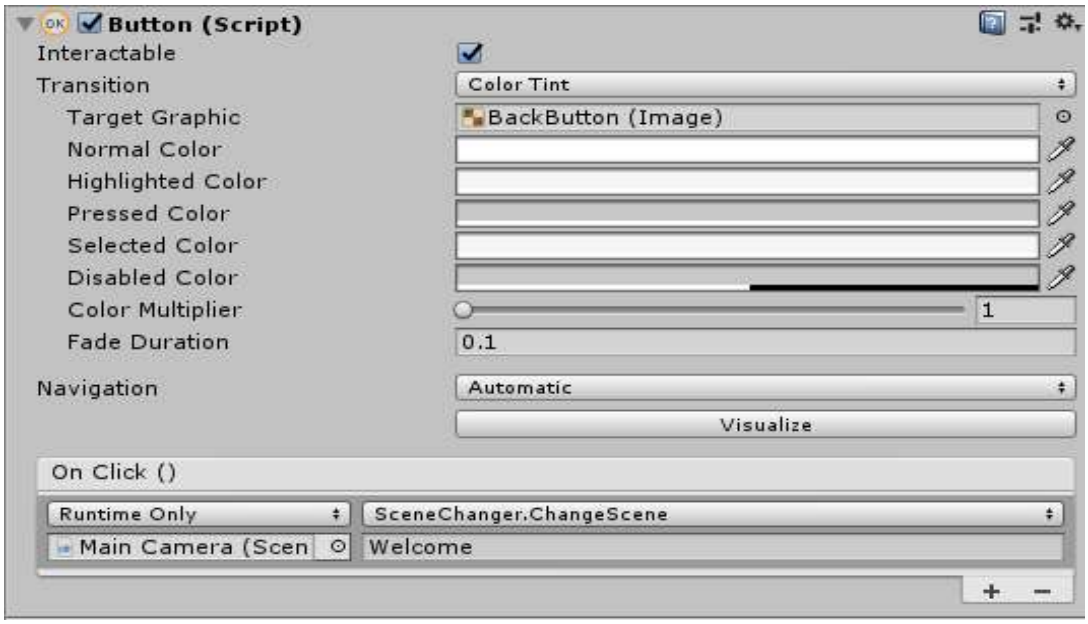
Rysunek 16: Modyfikowalne właściwości komponentu „*SpriteRenderer*”

Tekst (ang. ***Text***) to domyślny komponent graficzny do rysowania tekstu na scenie. Używany w projekcie do komunikowania użytkownikowi zaistniałych zdarzeń oraz jako podpisy komponentów.



Rysunek 17: Modyfikowalne właściwości komponenty „*Text”*

Przycisk (ang. *Button*) to element, który po kliknięciu wywołuje dane zdarzenia zdefiniowane w elemencie *On Click*. Komponent został użyty do wywoływania różnych zdefiniowanych wcześniej skryptów, między innymi do przechodzenia między scenami, uruchamiania metronomu, zapisu zmiksowanych ścieżek dźwiękowych do pliku i definiowania czy dodać do nich metronom.

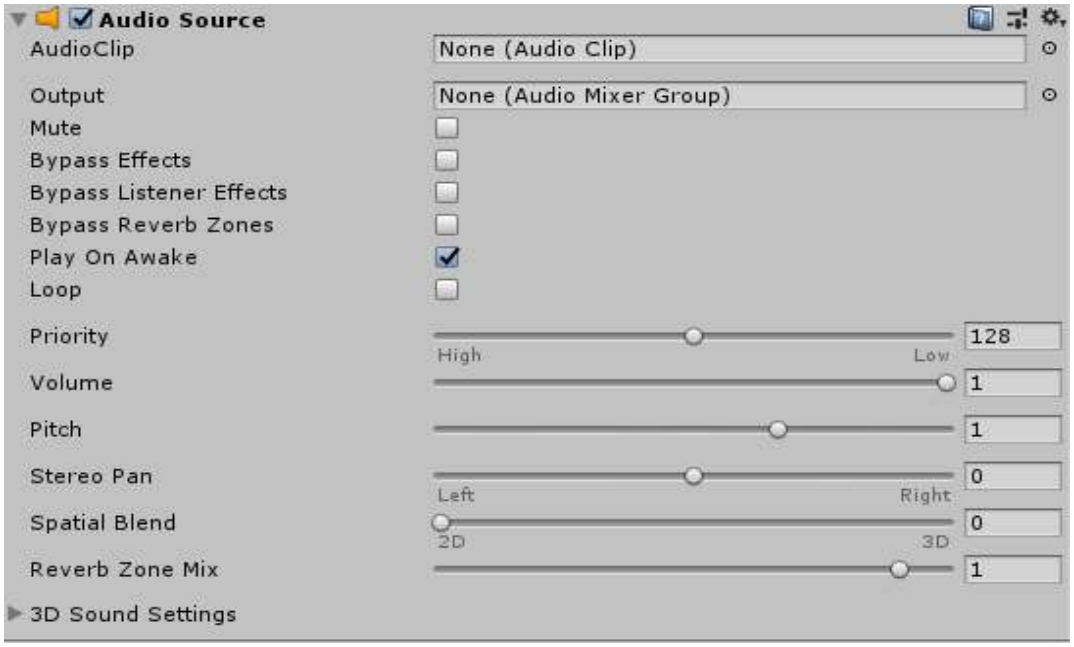


Rysunek 18: Modyfikowalne właściwości komponentu „*Button*”

Słuchacz audio (ang. *Audio Listener)* jest implementacją wirtualnego mikrofonu. Nagrywa dźwięki i odtwarza je. W scenie powinno się używać jednego słuchacza audio, nadmiar informowany jest przez stosowne ostrzeżenie w konsoli. Zawsze jest dodany do głównej kamery i nie posiada właściwości. Został użyty do przechwytywania dźwięku odtwarzanego przez metronom oraz do umożliwienia odsłuchiwania plików. Jak zaprezentowano na Rysunek 19, ten komponent nie posiada żadnych modyfikowalnych właściwości.

Rysunek 19: Komponent „*Audio Listener*”

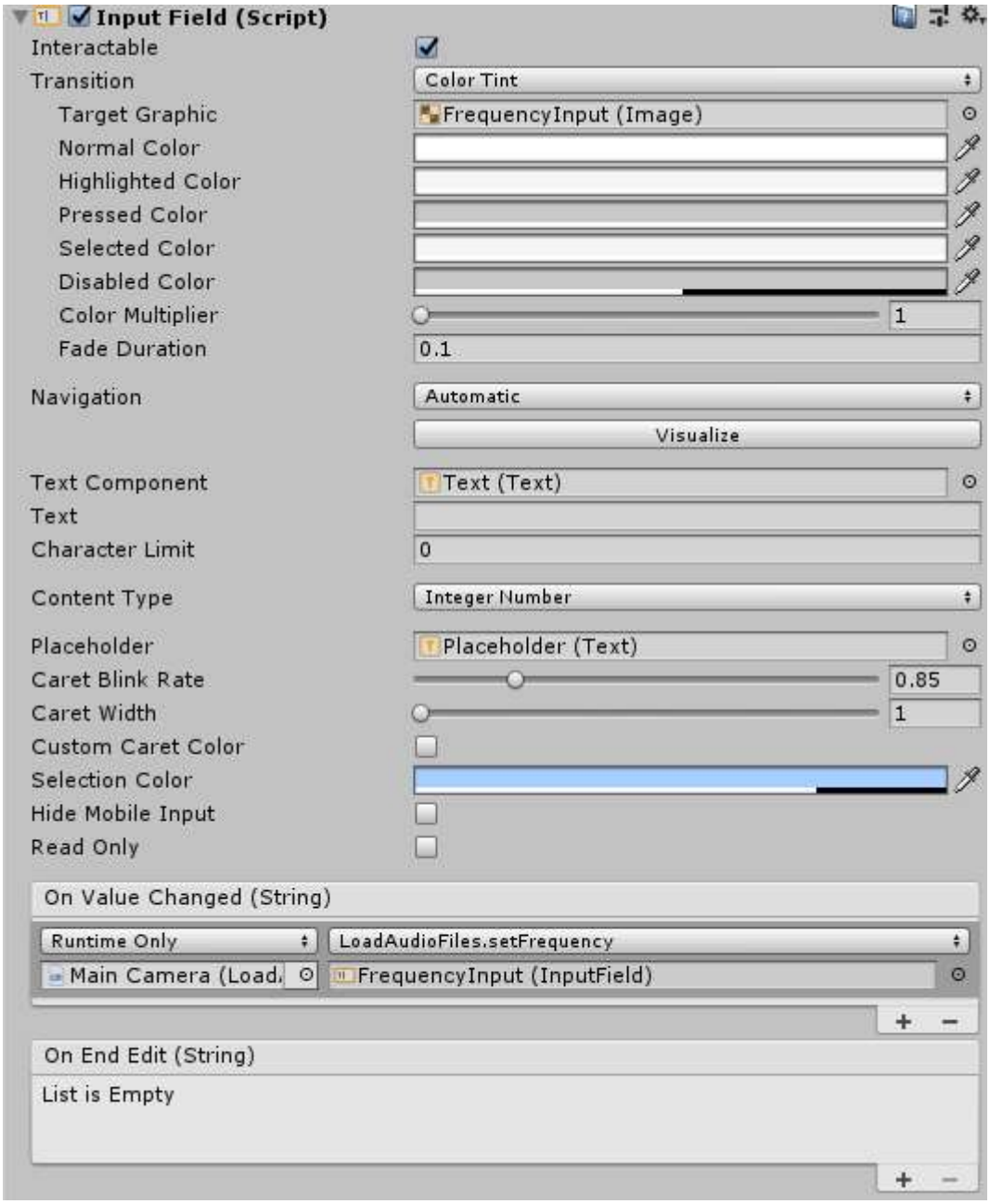
Źródło dźwięku (ang. *Audio Source***)** służy do emitowania sygnału dźwiękowego określonego w atrybucie *AudioClip* przez obiekt. Aby odtwarzać dźwięk, potrzebne jest również dodanie słuchacza audio do sceny. Przy jego użyciu można wpływać na głośność i ton dźwięku, wygenerować pogłos oraz określić stosunek przestrzeni prawego i lewego kanału względem siebie (panorama stereo) w odtwarzanym dźwięku.



Rysunek 20: Modyfikowalne właściwości komponentu „*Audio Source*”

Pole wejściowe (ang. *Input Field*) umożliwia wprowadzanie i edycję przez użytkownika tekstu widzianego w komponencie. Może wywołać zdarzenia w trakcie zmieniania tekstu lub po jego całkowitym wprowadzeniu, zdefiniowane kolejno w polach *On Value Changed* oraz *On End Edit*. Za pomocą *Content Type* można określić jaki typ danych może zostać wprowadzony do pola wejściowego. Na przykład, ustawienie tej właściwości na *Integer*

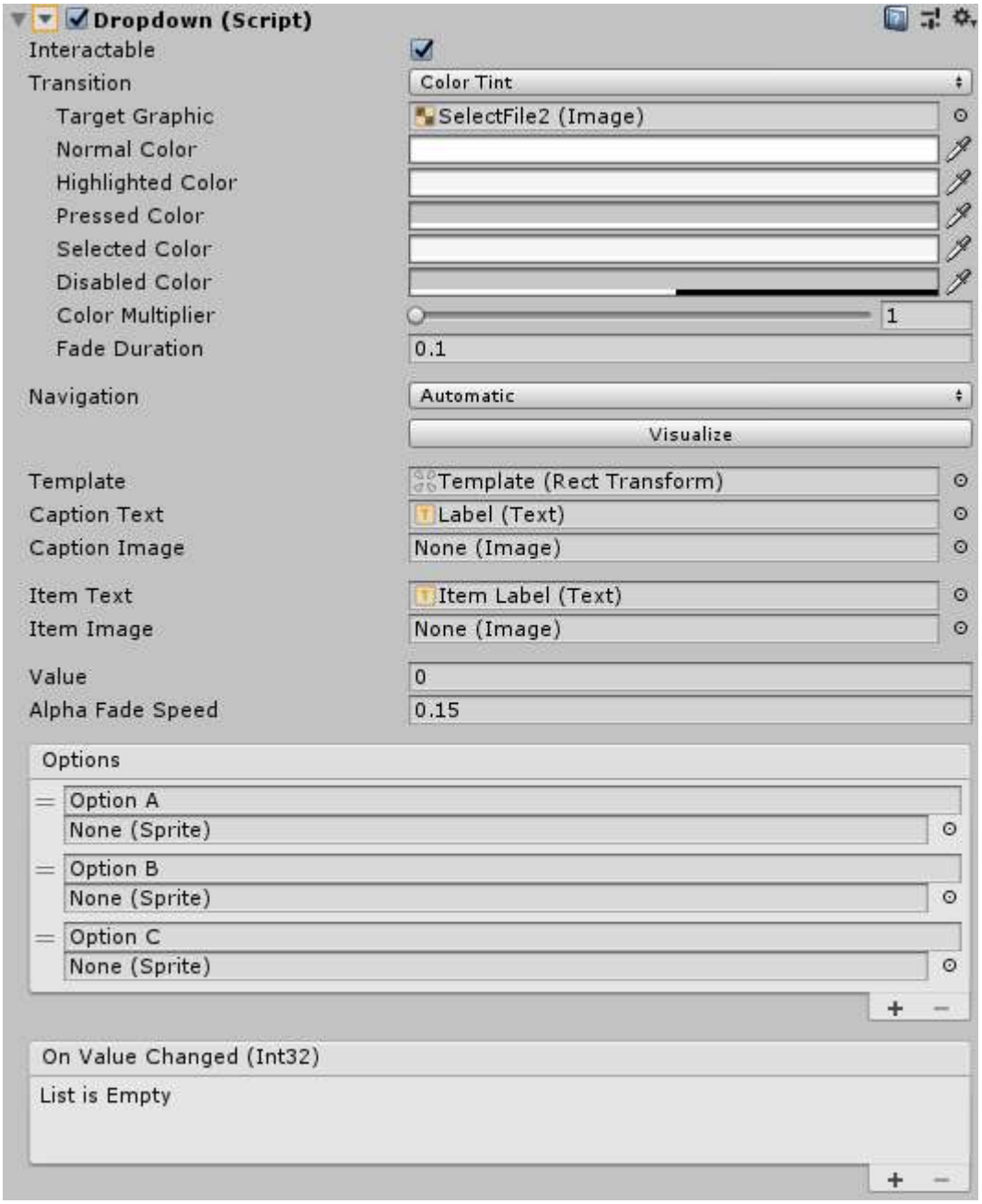
Number sprawi, że użytkownik nie będzie mógł wprowadzić innej wartości niż liczba całkowita. Atrybut *Placeholder* określa jaki napis zostanie wyświetlony w obrębie komponentu, jeżeli nie zostanie podana przez użytkownika żadna wartość dla atrybutu *Text*. Instancje pól wejściowych zostały wykorzystane do przekazywania od użytkownika informacji o tym jak ma nazywać się wyjściowy plik audio, ile ma posiadać kanałów oraz próbek na sekundę, jakie ma być tempo metronomu, na które uderzenie ma być odegrany akcent oraz jaka ma być długość taktu.



Rysunek 21: Modyfikowalne właściwości komponentu „*Input Field*”

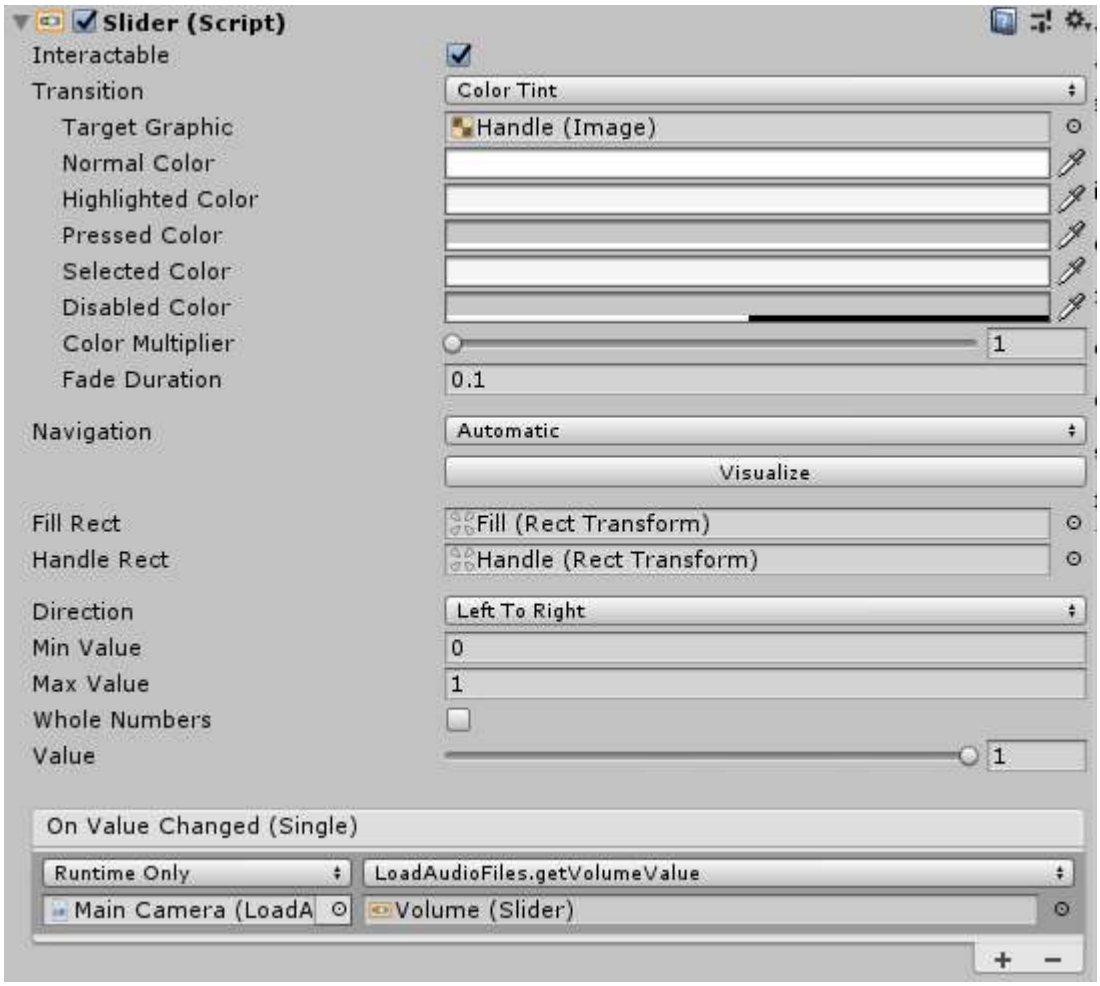
Rozwijane menu (ang. *Dropdown*) to rozwijane menu, które po kliknięciu przedstawia listę opcji, z których można wybrać jedną. Po wybraniu opcji etykieta zmienia się na aktualnie

wybraną opcję i mogą zostać wywołane zdarzenia zdefiniowane w polu *On Value Changed*. W niniejszym projekcie inżynierskim ten komponent został wykorzystany do umożliwienia użytkownikowi aplikacji wyboru plików do odtwarzania i łączenia. Gdy wywoływana jest scena na której występuje ten komponent, jego wszystkie opcje wyboru są usuwane (ponieważ posiada trzy domyślne opcje, których nie da się usunąć z poziomu edytora) i wypełniany jest opcjami reprezentującymi nazwy plików dźwiękowych. Gdy użytkownik utworzy nowy plik, opcje te zostają tak zaktualizowane, aby posiadały wszystkie pliki ze zdefiniowanej ścieżki.



Rysunek 22: Modyfikowalne właściwości komponentu „*Dropdown*”

Suwak (ang. ***Slider*)** umożliwia użytkownikowi wybranie wartość liczbowej z określonego zakresu poprzez przeciągnięcie znacznika. Może wywołać zdarzenie zdefiniowane w *On Value Changed*, tak samo jak pole wejściowe i rozwijane menu. Za jego pomocą użytkownik określa głośność (przedział od 0 do 1), panoramę (od -1 do 1, gdzie -1 oznacza lewy, a 1 prawy kanał), ton (od -3 do 3) oraz pogłos (od 0 do 1). Wraz z wysokością dźwięku zmienia się prędkość odtwarzania. Gdy jest większa od 1 ton jest wyższy, a plik odtwarzany jest szybciej, a gdy jest niższa od 1 ton jest niższy, a plik odtwarzany jest wolniej. W przypadku wartości ujemnych plik odtwarzany będzie od tyłu, a zależności odwracają się: im niższa wartość tym wyższy ton i prędkość odtwarzania.

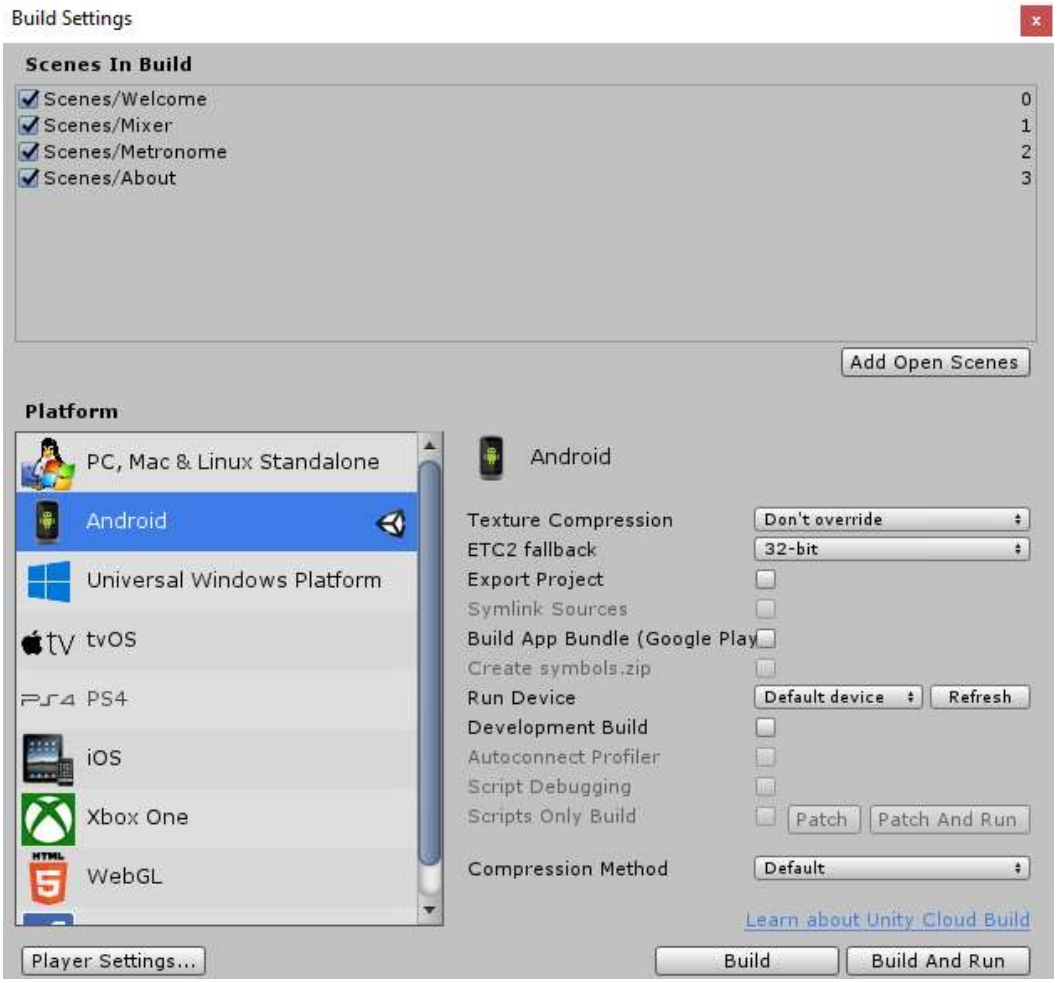


Rysunek 23: Modyfikowalne właściwości komponentu „*Slider*”

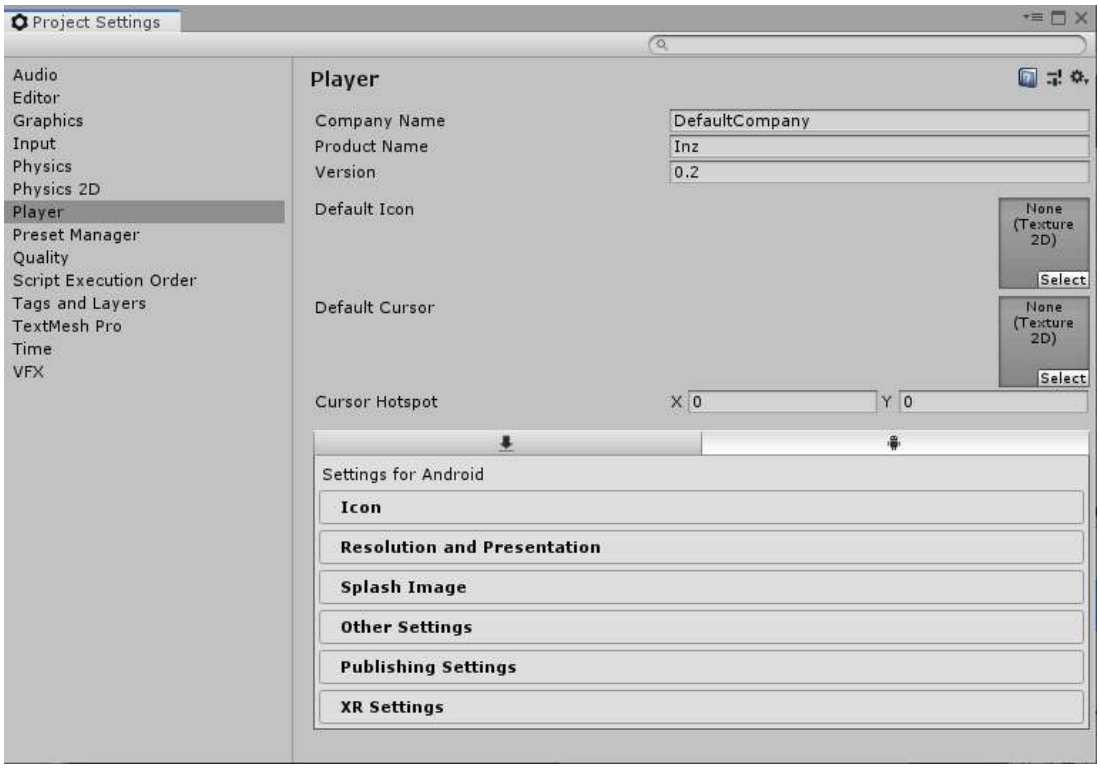
3.4.2. Tworzenie pliku APK

Aby zapisać projekt Unity jako plik APK, należy zbudować projekt zaznaczając platformę Android na liście dostępnych platform (Rysunek 24). Przed zbudowaniem projektu można jeszcze określić kolejność scen podczas budowania, wybrać w jaki sposób chcemy

skompresować tekstury, wyeksportować projekt, określić czy budujemy AAB(ang. *Android App Bundle)*, wybrać na jakim urządzeniu ma zostać uruchomiony projekt, określić czy projekt ma być zbudowany jako wersja deweloperska oraz związane z tym opcję, a także w jaki sposób aplikacja ma zostać skompresowana. Oprócz tego klikając na przycisk *Player Settings,* otworzymy ustawienia projektu z aktywną zakładką *Player* (Rysunek 25). W tym oknie można ustawić ikonę aplikacji, opcje związane z rozdzielczością i wyglądem aplikacji, obraz startowy oraz obraz wyświetlany podczas ładowania aplikacji, opcje publikacji, renderowania, identyfikacji, optymalizacji i konfiguracji.



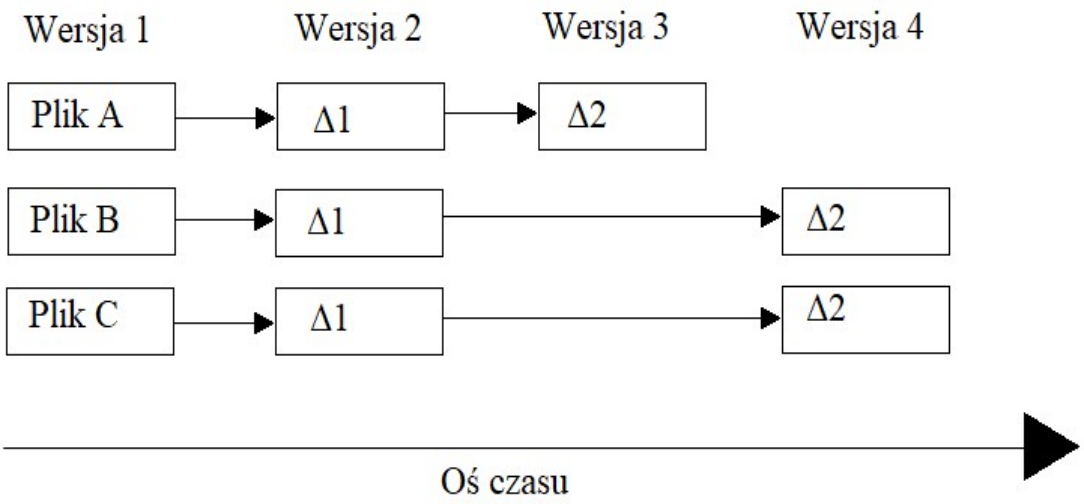
Rysunek 24: Ustawienia budowania aplikacji (*BuildingSettings*)



Rysunek 25: Ustawienia projektu (*Project Settings*)

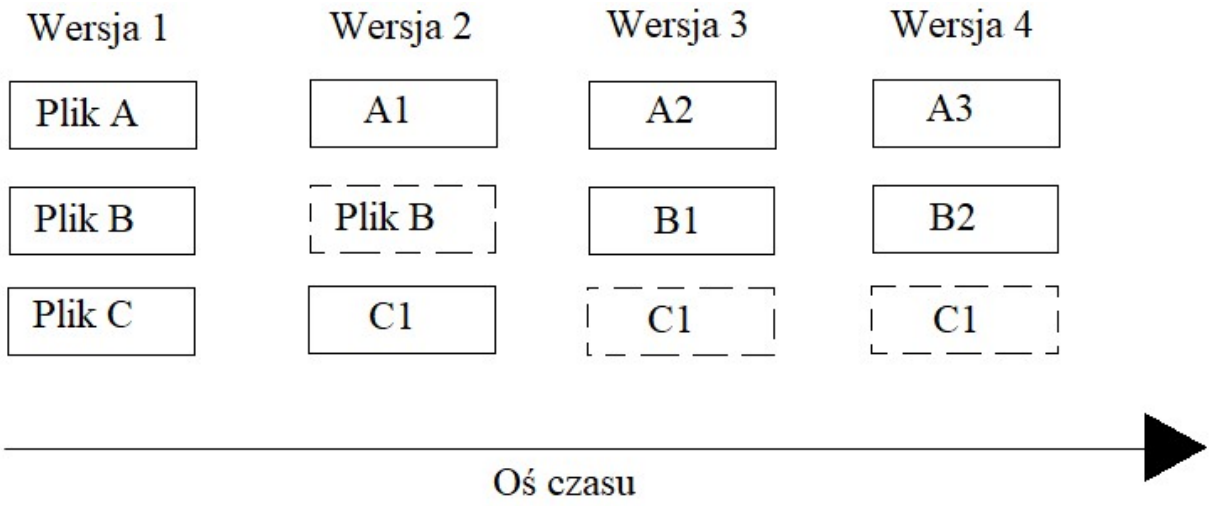
3.5. System kontroli wersji GIT [17]

Kontrola wersji to system, który rejestruje zmiany w pliku lub zestawie plików w czasie, umożliwiający przywoływanie oraz modyfikowanie określonych wersji pliku lub zestawu plików. Pozwala porównać zmiany w czasie, zobaczyć ostatnie modyfikacje, kiedy zostały wprowadzone i wiele więcej, ale przede wszystkim pozwala na łatwe przywrócenie ostatniej poprawnej wersji. Główną różnicą między GIT'em, a innymi systemami kontroli wersji, jest sposób w jaki GIT przetwarza i przechowuje dane. Większość starszych systemów kontroli wersji przechowuje informacje jako listę zmian opartych na plikach. Informacje przechowywane są jako zestaw plików i zmian tych plików w czasie (kontrola wersji oparta na delcie), co przedstawia Rysunek 26.



Rysunek 26: Kontrola wersji oparta na delcie [17]

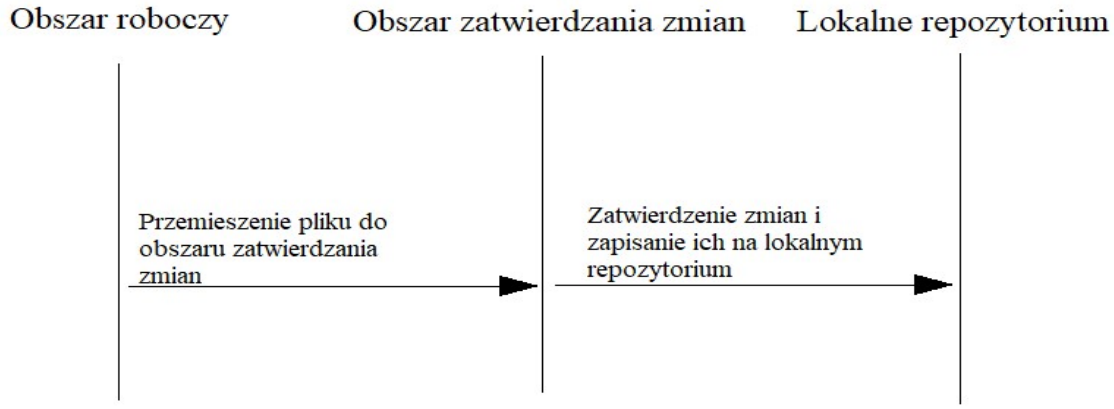
GIT przechowuje dane w formie serii obrazów systemu plików. Za każdym razem, gdy zostaną zatwierdzone zmiany, GIT tworzy obraz plików i przechowuje odniesienie do tego obrazu. Dzięki temu, gdy jakiś plik nie zostaje zmieniony, GIT przechowuje link do identycznego pliku, który został zapisany wcześniej (Rysunek 27), zamiast zapisywać na nowo niezmieniony plik.



Rysunek 27: Kontrola wersji GIT [17]

Dzięki takiemu rozwiązaniu większość operacji w GIT wymaga tylko lokalnych plików i zasobów. Jednak, dopóki komputer jest odłączony od sieci, zmiany mogą zostać zapisane tylko lokalnie i nie będzie możliwe przeniesienia plików na repozytorium zdalne. Jak w przypadku każdego VCS, można utracić lub zepsuć zmiany, których jeszcze nie zapisano, ale po zatwierdzeniu obrazu w GIT bardzo trudno jest je stracić, szczególnie jeśli zmiany często są wysyłane na zdalne repozytorium. GIT posiada trzy główne stany,

w których mogą znajdować się pliki: zmodyfikowane (ang. *modified*), przemieszczone (ang. *staged*) i zatwierdzone (ang. *committed*). Zmodyfikowany plik to po prostu plik, w którym dokonano jakiejkolwiek zmiany, ale nie zostały one jeszcze zatwierdzone w repozytorium. Przemieszczenie oznacza, że dany zmodyfikowany plik został zaznaczony, aby mógł przejść do zatwierdzenia. Zatwierdzenie oznacza, że dane zostaną zapisane w lokalnym repozytorium (Rysunek 28). Ważne jest, aby zwrócić uwagę na jakiej gałęzi (ang. *branch*) mają zostać zapisane dane. Gałęzie są wskaźnikami na zapisane zestawy danych na repozytorium. Domyślna nazwa gałęzi GITa to *master*. Kiedy zatwierdzone zostaną pierwsze zmiany, tworzona jest gałąź *master*, która wskazuje na ostatni zatwierdzony zestaw danych. Z każdym zatwierdzeniem automatycznie przesuwa się ona do przodu, wskazując na zaktualizowany zestaw danych. Po zapisaniu danych na lokalnym repozytorium, kolejnymi krokami jest wybór gałęzi na repozytorium zdalnym i wysłanie na nią zatwierdzonych zmian (ang. *push*).



Rysunek 28: Zapisywanie danych na repozytorium lokalnym [17]

W niniejszym projekcie inżynierskim korzystano z hostingowego serwisu internetowego przeznaczonego dla projektów programistycznych wykorzystujących system kontroli wersji GIT, który udostępnia darmowy hosting prywatnych repozytoriów - GitHub. Od czerwca 2018 roku serwis został przejęty przez firmę Microsoft.

4. Projekt

Omawiany projekt inżynierski został stworzony przy pomocy środowiska Unity na urządzenia z systemem operacyjnym Android. Posiada cztery sceny odpowiadające za prezentacje różnych funkcjonalności, które zostaną opisane w następnym podrozdziale.

4.1. Założenia projektowe

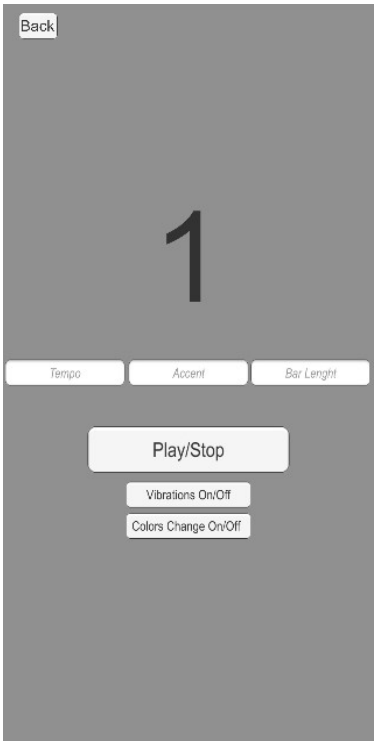
Według założeń, projekt powinien umożliwić użytkownikowi odtworzenie wybranego pliku dźwiękowego, pozwolić na modyfikacje odtwarzanego dźwięku, połączyć dwa wybrane pliki (jako jeden za drugim lub równolegle), dodać do tych plików dopasowany przez użytkownika sygnał metronomu, a także uruchomić metronom oddzielnie. Metronom może przekazywać informacje o kolejnych sygnałach za pomocą dźwięku, wibracji, zmiany wyświetlanej liczby oraz koloru tła. Użytkownik powinien móc zdecydować o sposobie sygnalizowania stanu przez metronom. Co więcej użytkownik może określić czy tworzony plik ma mieć jeden kanał (mono) lub dwa kanały (stereo), jego nazwę oraz jaką ma mieć częstotliwość próbkowania.

4.2. Warstwa graficzna

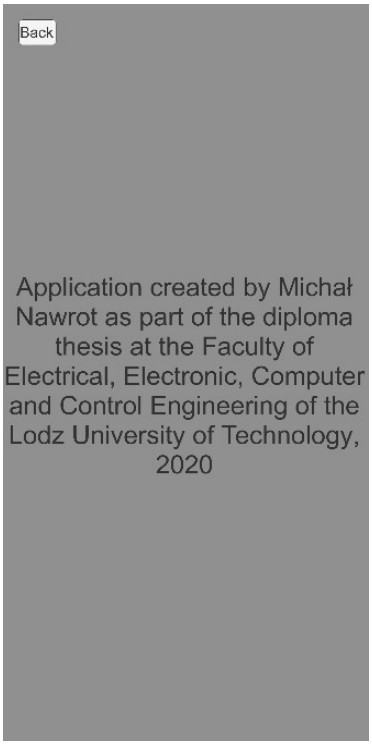
Dla zachowania przejrzystości, aplikacja bazuje na podstawowym wyglądzie komponentów silnika Unity. Kolory zostały wybrane w taki sposób, aby tekst kontrastował z tłem i mógł być łatwo odczytany, a rozmiary komponentów zostały dopasowane, tak aby można było wygodnie z nich korzystać. Każde kliknięcie przycisku jest sygnalizowane przez zmianę jego koloru z białego na szary. Większość przycisków znajduje się w obrębie środka i dołu ekranu, ponieważ znaczna ilość użytkowników smartfonów korzysta z nich używając jednej ręki i takie rozmieszczenie przycisków jest dla nich najwygodniejsze. Obraz aplikacji skaluje się wraz z rozmiarem ekranu urządzenia w równych proporcjach szerokości i wysokości.

5. Sposób działania aplikacji

Pierwsza scena, która zostaje pokazana użytkownikowi aplikacji to scena powitalna (Rysunek 32), z której za pomocą przycisków może wybrać pozostałe sceny. Scena o nazwie *About* (Rysunek 29) posiada jeden komponent tekst z informacjami, kto jest autorem aplikacji i w jakich okolicznościach została zaimplementowana. Scena *Metronome* (Rysunek 30) posiada trzy pola wejściowe, do przekazywania tempa, numeru, akcentowanego uderzenia i długości taktu przez użytkownika, jeden tekst odpowiedzialny za reprezentowanie wartości liczbowej sygnału metronomu oraz cztery przyciski, gdzie trzy z nich odpowiadają za włączanie i wyłączanie metronomu, wibracji i zmian koloru tła, a czwarty za powrót do poprzedniej sceny. Scena *Mixer* (Rysunek 33 i Rysunek 31) posiada cztery suwaki umożliwiające regulacje atrybutów odtwarzanej ścieżki dźwiękowej, trzy rozwijane menu odpowiedzialne za wybory plików, pięć pól wejściowych do wprowadzenia ilości kanałów, częstotliwości próbkowania, nazwy zapisywanego pliku, tempa metronomu i co ile uderzeń ma być generowany akcent, a także osiem przycisków pozwalających użytkownikowi włączyć i wyłączyć odtwarzanie, zapauzować odtwarzanie, wyciszyć ścieżkę, określić czy do pliku wyjściowego ma być dodany sygnał metronomu oraz w jaki sposób mają zostać połączone pliki. Scena *Mixer* podzielona jest na dwa widoki, pomiędzy którymi można przechodzić przesuwając w poziomie palcem po ekranie. Na samym początku aplikacja sprawdza czy w głównym folderze istnieje folder FilesInz, z którego będą wczytywane pliki oraz do którego będą zapisywane nowe. Jeśli nie, tworzy go.

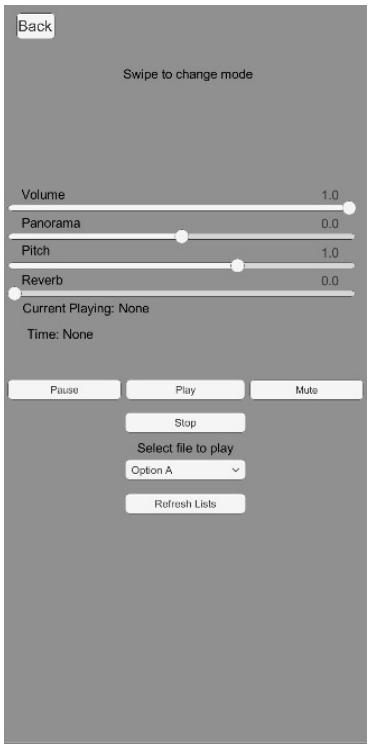


Rysunek 30: Scena *Metronome*



Rysunek 29: Scena *About*

Rysunek 32: Scena powitalna



Rysunek 33: Widok odtwarzania ścieżek sceny *Mixer*

Rysunek 31: Widok łączenia ścieżek sceny *Mixer*

5.1. Funkcja łączenia ścieżek

Użytkownik korzystając z rozwijanych menu może wybrać pliki dźwiękowe, które mają zostać połączone w jeden plik. Wpisując dane do pól wejściowych może określić jak ma nazywać się nowy plik, jaka ma być ilość kanałów, częstotliwość próbkowania, tempo metronomu i długość taktu, a także za pomocą przycisków może zdecydować w jaki sposób mają zostać zmiksowane oraz czy ma zostać dodany sygnał metronomu. Do zapisu nowego pliku wymagane jest wpisanie nazwy dla tego pliku w pola wejściowe oraz wybór dwóch różnych plików z rozwijanych menu. Pozostałe dane jeżeli nie będą zmienione przez użytkownika pozostają domyślne. Wartości startowe dla zmiennych to: dla liczby kanałów dwa, dla częstotliwości próbkowania 44100, dla tempa metronomu 120 i dla długości taktu 4. Aby zapisać plik zawierający ścieżki ułożone jedna po drugiej trzeba wywołać metodę *SaveSeparatelyButton* (Listing 1) przypisaną do odpowiedniego przycisku, a aby zapisać plik ze ścieżkami nałożonymi na siebie trzeba kliknąć przycisk wywołujący metodę *SaveMixedButton* (Listing 2).

```
public void SaveSeparatelyButton(InputField fileNameInput){

	String fileName = fileNameInput.text;
	if (fileName != null&&fileName != ""){
		SetSelectedAudioClips(dropdown1, dropdown2);
		if (selectedAudioClips.Count> 1){
			AudioClip result =
			audioClipCombine.Combine(selectedAudioClips[0],
				selectedAudioClips[1], this.metroClip,this.AccentClip,
				this.channels, this.frequency, this.addMetronome,
				this.metronomeBPM, this.metronomeAccents);
			SavWav.Save(fileName, result);
			text.text = "File is saved as:" + fileName + ".wav";
			selectedAudioClips.Clear();
			GetFilesInDirectory();
		}
		else{
			text.text = "Choose files to concatenate";
		}
	}
	else{
		text.text = "Your File need to have a name";
	}
}
```

Listing 1: Metoda SaveSeparatelyButton

```
public void SaveMixedButton(InputField fileNameInput)
{
	String fileName = fileNameInput.text;

	if (fileName != null&&fileName !=""){
		SetSelectedAudioClips(dropdown1, dropdown2);
		if (selectedAudioClips.Count> 1){
			AudioClip result =
			audioClipCombine.MixAudioFiles(selectedAudioClips[0],
			selectedAudioClips[1],this.metroClip, this.AccentClip,
			this.channels, this.frequency,this.addMetronome,
			this.metronomeBPM, this.metronomeAccents);
			SavWav.Save(fileName, result);
			text.text = "File is saved as: " + fileName + ".wav";
			selectedAudioClips.Clear();
			GetFilesInDirectory();
		}
		else{
			text.text = "Choose files to mix";
		}
	}
	else{
		text.text = "Your File need to have a name";
	}
}
```

Listing 2: Metoda *SaveMixedButton*

Obie metody przyjmują jako argument pole wejściowe, z którego pobierają wartość jego atrybutu *text*, który zostanie wykorzystany jako nazwa pliku. Jeżeli pobrany łańcuch znaków (ang. *String*) jest pusty albo nie posiada żadnych znaków, zmieniana jest wartość *text* komponentu Tekst na scenie, informując użytkownika, że aby plik został zapisany, musi zostać podana jego nazwa. Jeżeli użytkownik podał nazwę pliku, następnym krokiem jest wybranie plików, które mają zostać połączone. Metoda *SetSelectedAudioClips* (Listing 3) pobiera informacje o wybranych przez użytkownika opcjach rozwijanych menu i na tej podstawie wypełnia listę *selectedAudioClips*. Jeżeli wybrano te same opcje w obu rozwijanych menu, użytkownik zostanie poinformowany, że drugi plik nie może być tym samym plikiem co pierwszy wybrany. Dodatkowo sprawdzane jest czy nie istnieje więcej plików o tej samej nazwie w folderze z plikami (czy w liście klipów audio (ang. *AudioClip*) istnieją dwa z tą samą nazwą). Jeżeli tak, klip audio danego pliku jest dodany tylko raz do listy *selectedAudioClips*. Po wykonaniu metody *SetSelectedAudioClips*, dla *SaveMixedButton*uruchamiana jest metoda *MixAudioFiles*, klasy *AudioClipCombine* zwracająca klip audio złączonych ścieżek, lub dla *SaveSeparatelyButton* metoda *Combine* tej samej klasy zwracająca klip audio ścieżek połączonych jedna za drugą. Przyjmowane argumenty to kolejno klip audio z pierwszego pliku, klip audio drugiego pliku, klip audio podstawowego sygnału metronomu, klip audio akcentowego sygnału metronomu, liczba całkowita (ang. *int*) z liczbą kanałów, liczba całkowita z częstotliwością próbkowania, typ

logiczny (ang. *bool*) czy dodać metronom, liczba całkowita z tempem metronomu oraz liczba całkowita z długością taktu.

```
public void SetSelectedAudioClips(Dropdown dropdown1, Dropdown dropdown2){

	firstName = dropdown1.options[dropdown1.value].text;
	secondName = dropdown2.options[dropdown2.value].text;
	inti = 0;

	foreach(AudioClipaudioClipinaudioClips){
		if (audioClip.name == firstName){
			if (i==0){
				selectedAudioClips.Add(audioClip);
				i++;
			}
		}
	}
	i = 0;
	foreach (AudioClipaudioClipinaudioClips){
		if (selectedAudioClips[0].name != audioClip.name) {
			if (audioClip.name == secondName){
				if (i == 0){
					selectedAudioClips.Add(audioClip);
					i++;
				}
			}
		}
		else{
			text.text = "The second file to mix can't be same file like
		first";
		}
	}
}
```

Listing 3: Metoda *SetSelectedAudioClips*

Obie metody na początku sprawdzają czy przekazane klipy audio nie są puste, jeżeli tak metoda zwraca *null*. Jeżeli nie, tworzone są tablice liczb zmiennoprzecinkowych o rozmiarach równych ilości próbek pomnożonej przez liczbę kanałów, które zostają wypełniane są danymi pobranymi z klipów audio (Listing 5). W przypadku metody *Combine* tworzona jest tablica liczb zmiennoprzecinkowych o łącznym rozmiarze dwóch poprzednio stworzonych tablic (Listing 4). Do tej tablicy przekazywane są dane pierwszego (od indeksu 0) oraz drugiego klipu audio (od indeksu równego długości przekazanych danych pierwszego klipu audio).

```
if (clipA == null || clipB == null)
       returnnull;

int length = 0;
length += clipA.samples * clipA.channels;
length += clipB.samples * clipB.channels;

float[] data = newfloat[length];
length = 0;

float[] buffer1 = newfloat[clipA.samples * clipA.channels];
clipA.GetData(buffer1, 0);
buffer1.CopyTo(data, length);
length += buffer1.Length;

float[] buffer2 = newfloat[clipB.samples * clipB.channels];
clipB.GetData(buffer2, 0);
buffer2.CopyTo(data, length);
length += buffer2.Length;
```

Listing 4: Tworzenie tablic danych pobranych z sampli w metodzie *Combine*

```
       float[] floatSamplesA = newfloat[clipA.samples * clipA.channels];
       clipA.GetData(floatSamplesA, 0);

       float[] floatSamplesB = newfloat[clipB.samples * clipB.channels];
       clipB.GetData(floatSamplesB, 0);

       float[] mixedFloatArray = MixAndClampFloatBuffers(floatSamplesA,
              floatSamplesB);
```

Listing 5: Tworzenie tablic danych pobranych z ampli w metodzie *MixAudioFiles*

Następnie, jeżeli użytkownik określił, że nie chce aby został dodany sygnał metronomu, zwracany jest nowy klip audio wypełniony danymi z tablicy posiadającej dane z obu klipów audio (Listing 6).

```
       AudioClip result = AudioClip.Create("Combined",
                           length, channels,frequency, false, false);
       result.SetData(data, 0);
       return result;
```

Listing 6: Tworzenie klipu audio

Jednak jeśli użytkownik zaznaczył, że chce aby został dodany sygnał metronomu, tworzony zostaje klip audio dla metronomu o długość klipów audio łączonych plików, wypełniony od początku do końca powtórzonymi w, obliczonym na podstawie podanego przez użytkownika tempa, czasie klipami audio podstawowego sygnału metronomu, z uwzględnieniem dodania akcentowanego sygnału co określoną przez użytkownika liczbę określającą długość taktu. Na końcu tworzona jest nowa tablica liczb zmiennoprzecinkowych

wypełniona odpowiednio przygotowanymi przez metodę *MixAndClampFloatBuffers* danymi, a potem na jej podstawie tworzony wyjściowy klip audio (Listing 8).

W przypadku metody *MixedAudioFiles*, długość wyjściowego klipu audio jest równa długości dłuższego z dwóch miksowanych plików audio, a dane nie są przepisywane bezpośrednio tylko przy użyciu metody *MixAndClampFloatBuffers* (Listing 7). Dodawanie sygnału metronomu oraz generowanie wyjściowego klipu audio odbywa się na takiej samej zasadzie jak w metodzie *Combine*. Metoda *MixAndClampFloatBuffers* jako argumenty przyjmuje dwie tablice wartości liczb zmiennoprzecinkowych, z których pobierane są dane o próbkach sygnału. Z obu tablic zapisuje się ich długości wybierając odpowiednio, która jest mniejsza, a która większa. Aby połączyć sygnały, do nowo powstałej tablicy liczb zmiennoprzecinkowych, o długości dłuższej z podanych jako argumenty tablic zapisuje się kolejno dane korzystając przy tym z funkcji *ClampToValidRange* (Listing 9), aby upewnić się, że uśredniona suma sygnałów będzie należała do zakresu -1 do 1. Jeżeli łączone pliki są różnej długości, to na podstawie pobranych długości wybiera się, które sygnały mają zostać zapisane bezpośrednio, bez liczenia średniej z sumy sygnałów.

```
private float[] MixAndClampFloatBuffers(float[] bufferA, float[] bufferB){
	int minLength = Math.Min(bufferA.Length, bufferB.Length);
	int maxLength = Math.Max(bufferA.Length, bufferB.Length);

	float[] mixedFloatArray = newfloat[maxLength];

	for (int i = 0; i < minLength; i++){
		mixedFloatArray[i] = ClampToValidRange((bufferA[i] + bufferB[i]) / 2);
	}

	if (minLength < maxLength){
		if (bufferA.Length > bufferB.Length){
			for (int i = minLength; i < maxLength; i++){
				mixedFloatArray[i] = ClampToValidRange(bufferA[i]);
			}
		}
		elseif (bufferA.Length < bufferB.Length){
			for (int i = minLength; i < maxLength; i++){
				mixedFloatArray[i] = ClampToValidRange(bufferB[i]);
			}
		}
	}
	return mixedFloatArray;
}
```

Listing 7: Metoda *MixAndClampFloatBuffers*

```
if (addMetronome)
{
	double timeOfResult = clipB.length + clipA.length;
	float[] floatSamplesC = new float[clipC.samples * clipC.channels];
	clipC.GetData(floatSamplesC, 0);
	float[] floatSamplesD = new float[clipD.samples * clipD.channels];
	clipD.GetData(floatSamplesD, 0);
	float[] floatSampleMetro = new float[length];
	double tick = 60.0 / bpm;
	double ticksAmount;
	ticksAmount = (int)(timeOfResult / tick);
	int j = 0;
	int addedTicksAmount = 0;
	int innerTicks = 1;
	int samplesInTime = clipC.frequency / (int)(1 / tick);

	AudioClipresultClip = AudioClip.Create("Temp", length, channels,
		frequency, false);

	for (int g = 0; g < floatSampleMetro.Length; g++){
		if (g % samplesInTime == 0){
			if (addedTicksAmount < ticksAmount){
				if (innerTicks == 1){
					resultClip.SetData(floatSamplesD,
					 (int)(0 +addedTicksAmount *samplesInTime));
					innerTicks++;
					addedTicksAmount++;
				}
				else{
					resultClip.SetData(floatSamplesC,
					 (int)(0 +addedTicksAmount * samplesInTime));
					innerTicks++;
					addedTicksAmount++;
				}
				if (innerTicks > Accent){
				innerTicks = 1;
				}
			}
		}
	}
	resultClip.GetData(floatSampleMetro, 0);
	float[] mixedFloatArrayWithMetronome = MixAndClampFloatBuffers(data,
			floatSampleMetro);
	AudioClip result = AudioClip.Create("Mixed",
		mixedFloatArrayWithMetronome.Length, channels, frequency, false);
	result.SetData(mixedFloatArrayWithMetronome, 0);
	return result;
                                                                        }
```

Listing 8: Tworzenie klipu audio z sygnałem metronomu

```
private float ClampToValidRange(float value){
	float min = -1.0f;
	float max = 1.0f;
	return (value < min) ? min : (value > max) ? max : value;
}
```

Listing 9: Metoda *ClampToValidRange*

Następnym krokiem jest zapis utworzonego klipu audio do pliku WAV z pomocą metody *Save* klasy *SavWav* (Listing 10). Ta metoda przyjmuje jako argumenty łańcuch znaków z nazwą pliku, który ma zostać utworzony oraz utworzony wcześniej klip audio. Aby uzyskać pełną ścieżkę dla zapisu pliku pobrano ścieżkę absolutną do pamięci wewnętrznej i dodano do niej nazwę folderu FilesInz oraz nazwę dla nowego pliku przekazaną w argumencie. Następnie tworzony jest nowy pusty plik w tej lokalizacji, a 44 pierwsze bajty są zapisywane puste (Listing 11). Kolejnym krokiem jest wyciągnięcie danych z klipu audio do tablicy liczb zmiennoprzecinkowych i przekonwertowanie ich na tablice bajtów, dzięki czemu możliwe będzie zapisanie danych do pliku. Tą operacją zajmuje się metoda *ConvertAndWrite* (Listing 12). Ostatnim krokiem jest nadpisanie wcześniej utworzonych 44 pustych bajtów danymi, które określą dany plik jako prawidłowy plik WAV. W tym celu wywoływana jest metoda *WriteHeader* (Listing 13).

```
public static void Save(string filename, AudioClip clip){
	if (!filename.ToLower().EndsWith(".wav")){
		filename += ".wav";
	}
	var filePath = "";
	if (Application.isEditor) {
		filePath = Path.Combine(Application.dataPath, filename);
	}
	else{
		String tempPath = "";
		AndroidJavaClassjc = new
			AndroidJavaClass("android.os.Environment");
		tempPath =
			jc.CallStatic<AndroidJavaObject>("getExternalStorageDirectory")
			.Call<string>("getAbsolutePath");
		tempPath += "/FilesInz";
		filePath = Path.Combine(tempPath, filename);
	}

	using (var fileStream = CreateEmpty(filePath))
	{
		ConvertAndWrite(fileStream, clip);
		WriteHeader(fileStream, clip);
	}
}
```

Listing 10: Metoda *Save*

```
static void ConvertAndWrite(FileStream fileStream, AudioClip clip){
              var samples = new float[clip.samples];
              Byte[] bytesData = new byte[2];
              clip.GetData(samples, 0);
              Int16[] intData = new Int16[samples.Length];
              bytesData = new Byte[samples.Length * 2];
              int rescaleFactor = 32767;

              for (int i = 0; i<samples.Length; i++){
                     intData[i] = (short)(samples[i] * rescaleFactor);
                     Byte[] byteArr = new Byte[2];
                     byteArr = BitConverter.GetBytes(intData[i]);
                     byteArr.CopyTo(bytesData, i * 2);
              }
              fileStream.Write(bytesData, 0, bytesData.Length);
}
```

Listing 12: Metoda *ConvertAndWrite*

```
static FileStreamCreateEmpty(string filepath){
       var fileStream = new FileStream(filepath, FileMode.Create);
       byte emptyByte = new byte();
       for (int i = 0; i < HEADER_SIZE; i++){
              fileStream.WriteByte(emptyByte);
       }
       returnfileStream;
}
```

Listing 11: Metoda *FileStreamCreateEmpty*

```
static void WriteHeader(FileStream fileStream, AudioClip clip){
        var hz = clip.frequency;
        var channels = clip.channels;
        var samples = clip.samples;

        fileStream.Seek(0, SeekOrigin.Begin);

        Byte[] riff = System.Text.Encoding.UTF8.GetBytes("RIFF");
        fileStream.Write(riff, 0, 4);

        Byte[] chunkSize = BitConverter.GetBytes(fileStream.Length - 8);
        fileStream.Write(chunkSize, 0, 4);

        Byte[] wave = System.Text.Encoding.UTF8.GetBytes("WAVE");
        fileStream.Write(wave, 0, 4);

        Byte[] fmt = System.Text.Encoding.UTF8.GetBytes("fmt ");
        fileStream.Write(fmt, 0, 4);

        Byte[] subChunk1 = BitConverter.GetBytes(16);
        fileStream.Write(subChunk1, 0, 4);

        UInt16 two = 2;
        UInt16 one = 1;

        Byte[] audioFormat = BitConverter.GetBytes(one);
        fileStream.Write(audioFormat, 0, 2);

        Byte[] numChannels = BitConverter.GetBytes(channels);
        fileStream.Write(numChannels, 0, 2);

        Byte[] sampleRate = BitConverter.GetBytes(hz);
        fileStream.Write(sampleRate, 0, 4);

        Byte[] byteRate = BitConverter.GetBytes(hz * channels * 2);
        fileStream.Write(byteRate, 0, 4);

        UInt16 blockAlign = (ushort)(channels * 2);
        fileStream.Write(BitConverter.GetBytes(blockAlign), 0, 2);

        UInt16 bps = 16;
        Byte[] bitsPerSample = BitConverter.GetBytes(bps);
        fileStream.Write(bitsPerSample, 0, 2);

        Byte[] datastring = System.Text.Encoding.UTF8.GetBytes("data");
        fileStream.Write(datastring, 0, 4);

        Byte[] subChunk2 = BitConverter.GetBytes(samples * channels * 2);
        fileStream.Write(subChunk2, 0, 4);
}
```

Listing 13: Metoda *WriteHeader*

5.2. Funkcja metronomu

Scena metronomu umożliwia użytkownikowi określenie tempa, długości taktu oraz pozwala wybrać, na które uderzenie ma być wyemitowany dźwięk akcentu za pomocą pól wejściowych, a przy użyciu przycisków określić czy kolor tła ma być zmieniany, czy podczas odtwarzania sygnału akcentu urządzenie ma zawibrować oraz uruchomić i zatrzymać metronom. Domyślnie wibracje oraz zmiana koloru są oznaczone jako aktywne, jednak dopóki użytkownik nie zdefiniuje pozostałych parametrów, sygnał metronomu nie będzie odtwarzany. Po kliknięciu przycisku odpowiedzialnego za uruchamianie metronomu zmieniana jest wartość zmiennej *isPlaying* na *true*. Metody *FixedUpdate* i *LateUpdate* (Listing 14) odpowiadają za liczenie czasu do następnego odtworzenia sygnału metronomu oraz za odtworzenie go *AudioSettings.dspTime* zwraca aktualny czas systemu Audio silnika Unity i jest bardziej dokładny niż wartość czasu zwracana przez *Time.time*. Po wyemitowaniu wiadomości ”OnTick” wywoływana zostaje metoda *OnTick* (Listing 15), która odpowiada za zmianę numeru, w zależności od tego które uderzenie w ciągu taktu jest aktualnie aktywne, koloru tła i numeru, wibrację urządzenia oraz odtworzenie podstawowego sygnału oraz akcentowanego sygnału metronomu, w zależności od wyborów użytkownika. Wartości tempa mogą być ustawione przez użytkownika między 0 a 500, a długość taktu między 0 a 16.

```
void FixedUpdate(){
        double timePerTick = 60.0f / bpm;
        double dspTime = AudioSettings.dspTime;
        while (dspTime >= nextTick){
            ticked = false;
            nextTick += timePerTick;
        }
}
void LateUpdate(){
       this.background.color = backgroundColor;
        if (playing && !ticked && nextTick >= AudioSettings.dspTime){
            ticked = true;
            BroadcastMessage("OnTick");
        }
}
```

Listing 14: Metody *FixedUpdate* i *LateUpdate*

```
void OnTick(){
	this.textNumber.text = i.ToString();
	if (i == barLenght){
		if (i == accent){
			accentTick.Play();
			if (colorChange) {
				this.background.color = new Color(1, 0, 0, 1);
				this.textNumber.color = new Color(0, 0, 1, 1);
			}
			if (vibrationsOn) {
				Vibration.Vibrate(50, 255, true);
			}
			i = 1;
		}
		else{
			tick.Play();
			if (colorChange){
				this.background.color = new Color(0, 1, 0, 1);
				this.textNumber.color = new Color(1, 0, 0, 1);
			}
			i = 1;
		}
	}
	else{
		if (i == accent){
			accentTick.Play();
			if (colorChange){
				this.background.color = new Color(1, 0, 0, 1);
				this.textNumber.color = new Color(0, 0, 1, 1);
			}
			if (vibrationsOn){
				Vibration.Vibrate(50, 255, true);
			}
			i++;
		}
		else{
			tick.Play();
			if (colorChange){
				this.background.color = new Color(0, 1, 0, 1);
				this.textNumber.color = new Color(1, 0, 0, 1);
			}
			i++;
		}
	}
}
```

Listing 15: Metoda *OnTick*

5.3. Funkcja odtwarzacza

Na scenie *Mixer* użytkownik może wybrać plik, który zostanie odtworzony przy użyciu rozwijanego menu oraz wpłynąć na jego parametry za pomocą suwaków. Aby odtworzyć wybrany plik użytkownik musi kliknąć odpowiedni przycisk, który wywoła metodę *PlayMusic* (Listing 17). Tak samo jeśli użytkownik chce zatrzymać odtwarzanie, wyciszyć odtwarzaną ścieżkę lub ją zapauzować musi nacisnąć odpowiadające za to przyciski, które wywołają funkcje *StopMusic*, *PauseMusic* (Listing 18), *MuteSound* (Listing 16). Jeżeli użytkownik nie wybierze pliku, który chce odtworzyć, zostanie odtworzony pierwszy plik z listy rozwijanego menu.

Aby móc prawidłowo wyświetlać aktualny czas odtwarzanego pliku, wykorzystano mechanizm silnika Unity nazywany *Coroutine*. Wywołana funkcja przejmuje wątek aplikacji aż do zakończenia jej instrukcji, co oznacza, że każde działanie mające miejsce w funkcji musi nastąpić w ramach aktualizacji pojedynczej klatki. Z tego powodu standardowe wywołanie funkcji nie może być użyte do kontrolowania zdarzeń w czasie. *Coroutine* zachowuje się jak funkcja, jednak umożliwia wstrzymanie swojego wykonania i przywrócenie kontroli do silnika Unity, a następnie kontynuowanie działań tam, gdzie *Coroutine* została przerwana w kolejnej wygenerowanej klatce. Gdy użytkownik uruchamia odtwarzanie, uruchamiany jest *Coroutine* o nazwie „WaitForMusicEnd”, a gdy zatrzymuje odtwarzanie również zatrzymywany jest *Coroutine*.

```
public void MuteSound(){
	if(audioSource.mute == false) {
		audioSource.mute = true;
	}
	else{
		audioSource.mute = false;
	}
}
```

Listing 16: Metoda *MuteSound*

```
public void PlayMusic(){
	if(audioSource.clip == null){
		audioSource.clip = audioClips[0];
	}
	if (audioSource.isPlaying){
		return;
	}
	else{
		setSelectedAudioToPlay(dropdown3);
		fullLenght = (int)audioSource.clip.length;
		fileTitle.text = audioSource.clip.name;
		audioSource.Play();
		StartCoroutine(WaitForMusicEnd());
	}
}
```

Listing 18: Metoda *PauseMusic*

```
public void PauseMusic(){
	if (audioSource.isPlaying){
		audioSource.Pause();
	}
	else{
		audioSource.Play();
		StartCoroutine(WaitForMusicEnd());
	}
}
```

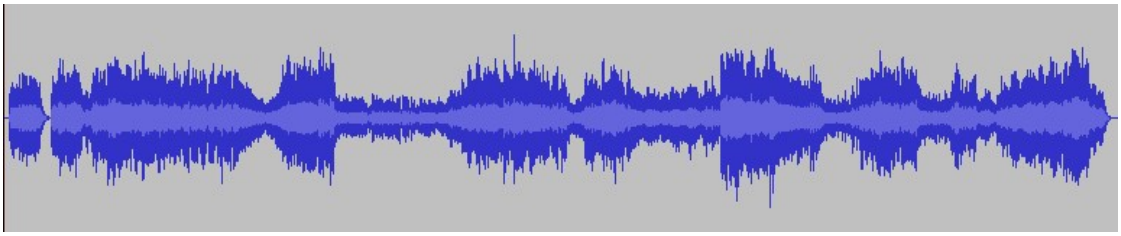
Listing 17: Metoda *PlayMusic*

6. Weryfikacja działania aplikacji

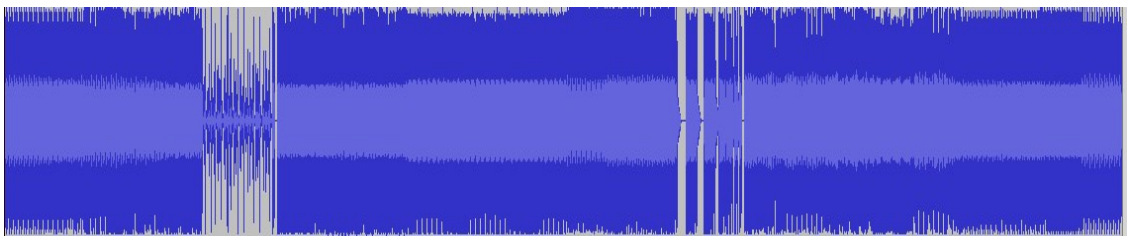
Weryfikacja działania jest jednym z kluczowych etapów wytwarzania oprogramowania. Celem weryfikacji jest uzyskanie informacji zwrotnej od testerów, aby pozwolić programistom określić co trzeba zmienić w aplikacji, aby poprawić jej działanie i wygląd oraz czy zostały spełnione założenia projektu. Żeby zweryfikować aplikację, trzeba odpowiedzieć na pytania, czy aplikacja została stworzona zaimplementowana zgodnie z założeniami, czyli czy rozwiązuje problem i wykorzystuje określone w założeniach technologie oraz czy działa na różnych urządzeniach i czy użytkownicy są zadowoleni ze sposobu działania aplikacji. Na podstawie odpowiedzi na te pytania można określić czy aplikacja została przygotowana prawidłowo.

6.1. Testy funkcji łączenia ścieżek

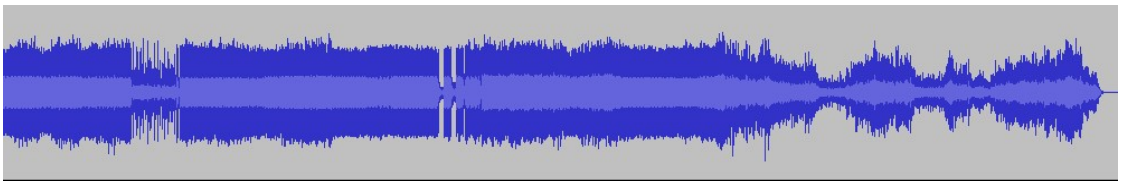
Jedną z głównych funkcjonalności aplikacji, jest możliwość łączenia ścieżek audio plików z lub bez sygnału metronomu. Funkcjonalności były sprawdzane w kolejności: łączenie ścieżek nakładając je na siebie bez metronomu, łączenie ścieżek nakładając je na siebie z metronomem, łączenie ścieżek bez nakładania ich na siebie bez metronomu, łączenie ścieżek bez nakładania ich na siebie z metronomem. Na rysunkach (Rysunek 34 i Rysunek 35) zaprezentowano wejściowe ścieżki audio plików, które będą łączone, a na rysunkach (Rysunek 36 i Rysunek 37) połączenie ich nakładając je na siebie oraz kolejno jeden po drugim. Jak widać gdy ścieżki są nakładane na siebie ich głośności są odpowiednio dopasowywane, tak aby głośniejsza ścieżka nie zagłuszała drugiej cichszej.



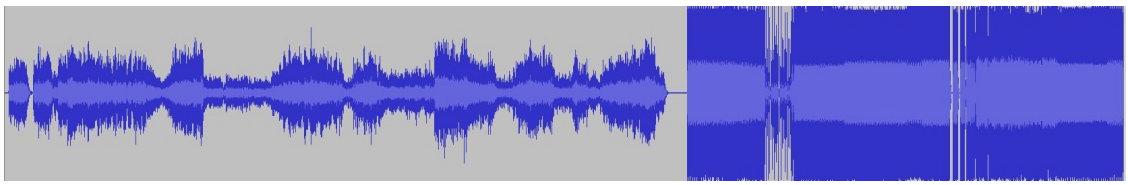
Rysunek 34: Widmo pierwszej ścieżki



Rysunek 35: Widmo drugiej ścieżki

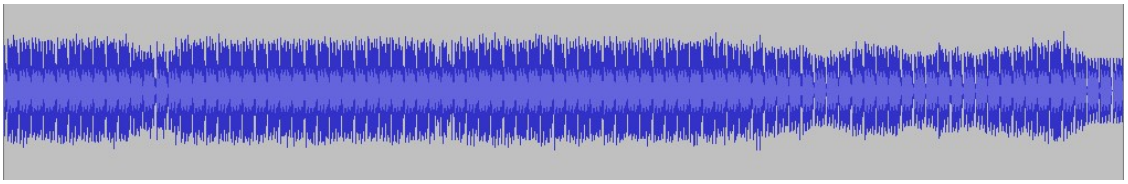


Rysunek 36: Widmo nałożonych na siebie ścieżek

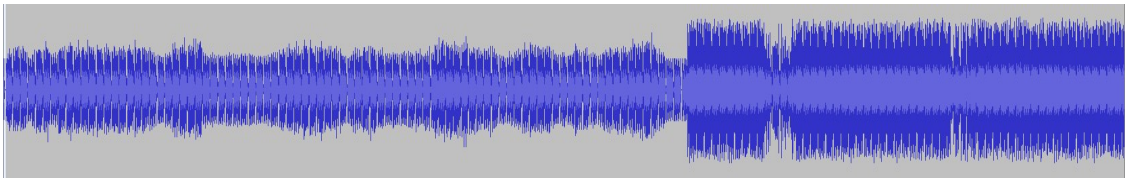


Rysunek 37: Widmo ścieżek połączonych jedna za drugą

Na kolejnych rysunkach (Rysunek 38 i Rysunek 39) przedstawiono tak samo połączone ścieżki jak wyżej omówione przykłady ale połączone z sygnałem metronomu.



Rysunek 38: Widmo nałożonych na siebie ścieżek z nałożonym sygnałem metronomu



Rysunek 39: Widmo ścieżek połączonych jedna za drugą z nałożonym sygnałem metronomu

Jak można wywnioskować z powyższych rysunków sygnał metronomu jest dodawany w równych odstępach czasu, a podczas dodawania go głośność ścieżek również jest dostosowywana, tak aby dodany sygnał był słyszalny. Na podstawie przeprowadzonych testów można założyć, że dane funkcjonalności zostały zweryfikowane pomyślnie.

6.2. Testy na różnych modelach telefonów i oceny użytkowników

Aplikacja była testowana przez cztery osoby na czterech urządzeniach różniącymi się specyfikacją. Użytkownicy testowi oceniali aplikacje w skali od jednego do pięciu, a głównymi ocenianymi podmiotami były: wygląd aplikacji, płynność działania, wygoda użytkowania, użyteczność aplikacji. Oceny wraz ze średnią zostały zestawione w Tabela 2.

Urządzenie / Ocena	Samsung Galaxy S8	Xiaomi RedmiNote 7	Xiaomi Redmi 6	Huawei P10 Lite	Średnia ocen w wierszu
Wygląd	4.00	3.50	3.00	4.00	3.63
Płynność	5.00	4.00	3.50	4.00	4.13
Wygoda	4.50	4.00	4.00	5.00	4.38
Użyteczność	4.00	4.50	4.00	3.50	4.00
Średnia ocen w kolumnie	4.38	4.00	3.63	4.13	4.04

Tabela 2: Oceny użytkowników

Jak wynika z ocen aplikacja została odebrana pozytywnie. Wygoda oraz płynność w końcowym rozrachunku zostały ocenione najlepiej, a najsłabszym ogniwem aplikacji jest jej wygląd. Użytkownik smartfona Xiaomi Redmi 6 ocenił ją najgorzej, a ocenę uargumentował domyślnym wyglądem komponentów i całościowo oszczędnym wyglądem. Użytkownicy urządzeń Xiaomi określili metronom jako przydatny i łatwy w obsłudze, a użytkownik urządzenia Samsung był zadowolony z funkcji łączenia ścieżek przy użyciu jednego przycisku.

7. Podsumowanie i wnioski

Podczas realizacji projektu inżynierskiego udało się opracować aplikację mobilną, która spełnia ustalone założenia projektu. Aplikacja, jak wykazały testy, prawidłowo łączy wybrane przez użytkownika klipy audio oraz zapisuje je do właściwego pliku. Przy okazji przeprowadzone prace pozwoliły wyjaśnić dlaczego przy pracy realizatora dźwięku w większości pracuje się na plikach nieskompresowanych oraz w jaki sposób nagłówek pliku WAV wpływa na jego brzmienie. Źle stworzony nagłówek skutkuje uszkodzonym plikiem – przy odtwarzaniu pliku zamiast zakładanego dźwięku uzyska się szum. Aby silnik Unity generował wibracje na urządzenia mobilnych, które programista chce kontrolować (określić siłę wibracji, ich długość oraz wzór), musi napisać własną klasę, która będzie je kontrolować. Dodatkowo zauważono, że domyślnie Unity umożliwia uzyskanie dostępu między innymi do Internetu, kamery, mikrofonu, wibracji, jednak nie do pamięci wewnętrznej urządzenia. Aby uzyskać dostęp do zasobów trzeba stworzyć folder o nazwie *Plugin* i utworzyć plik *AndroidManifest.xml*, w którym będą znajdować się linijki kodu umożliwiające zapytanie użytkownika o dostęp do pamięci wewnętrznej urządzenia.

Informacja zwrotna od użytkowników, którzy testowali aplikację, nakierowuję przyszłą pracę nad aplikacją na rozwój graficzny w zakresie UI (ang. *User Interface*) i UX (ang. *User Experience*). Aplikację można rozwinąć o tworzenie list odtwarzania aby użytkownicy mogli łatwo organizować pliki dźwiękowe, które mogły by zostać użyte jako podkłady do nauki na instrumentach oraz pasek przewijania utworów, aby użytkownicy mogli decydować, który moment utworu ma zostać w danym momencie odtworzony. Kolejnym elementem możliwym do dodania jest funkcja wyboru fragmentu odtwarzanego pliku, w celu jego zapętlenia.

8. Literatura

1. Marian Kozielski, Krystyna Wosińska, Przemysław Duda. *Multimedialny podręcznik do nauki fizyki dla szkół ponadgimnazjalnych.,* (2014). [Online]. http://ilf.fizyka.pw.edu.pl/podrecznik/3/5/7 (dostęp 02.2020).
2. Marian Kozielski, Krystyna Wosińska, Przemysław Duda. *Multimedialny podręcznik do nauki fizyki dla szkół ponadgimnazjalnych.,* (2014). [Online]. http://ilf.fizyka.pw.edu.pl/podrecznik/3/5/6 (dostęp 02.2020).
3. Everest, F. A."*Podręcznik akustyki*" (s. 33-44, 128). Sonia Draga (2003).
4. Marian Kozielski, Krystyna Wosińska, Przemysław Duda. *Multimedialny podręcznik do nauki fizyki dla szkół ponadgimnazjalnych.,* (2014). [Online]. http://ilf.fizyka.pw.edu.pl/podrecznik/3/5/13/ (dostęp 02.2020).
5. Everest, F. A."*Podręcznik akustyki*" (s. 75). Sonia Draga (2003).
6. M.Ostrowski. *Sklepinternetowy muzyczny.pl,*. (2020). [Online]. https://muzyczny.pl/ (dostęp 02.2020).
7. Bruce Fries, M. F.."*Digital Audio Essentials*"(s. 140-148). O'REILLY (2005).
8. RagnhildBrovig-Hanssen & Anne Danielsen. "*Digital signatures: The Impact of Digitization on Popular Music Sound*" (s. 9-13).The MIT Press (2016).
9. Craig Stuart Sapp. *A Soundfile Reading/Writing Library in C++,*.(2005). [Online]. http://soundfile.sapp.org/doc/WaveFormat/ (dostęp 01.2020).
10. Bruce Fries, M. F.."*Digital Audio Essentials*"(s. 155-162). O'REILLY (2005).
11. StatCounter. *StatCounterGlobalStats.*. (2020). [Online]. https://gs.statcounter.com/os-market-share/mobile/worldwide/(dostęp 02.2020).
12. Google. *Android Documentation,Activities.* (2019). [Online].https://developer.android.com/guide/components/activities/intro-activities/ (dostęp 02.2020).
13. Google. *Android Documentation, Services.* (2019). [Online].https://developer.android.com/guide/components/services/ (dostęp 02.2020).
14. Google. *Android Documentation,Broadcasts.* (2019). [Online].https://developer.android.com/guide/components/broadcasts/ (dostęp 02.2020).
15. Google. *Android Documentation, Content Providers.* (2019). [Online].https://developer.android.com/guide/topics/providers/content-providers/ (dostęp 02.2020).

16. Google. *Android Documentation, Activity Lifecycle.* (2019). [Online]. https://developer.android.com/guide/components/activities/activity-lifecycle/ (dostęp 02.2020).
17. Scott Chacon, Ben Straub. "*Pro git"* APRESS(2020).

9. Wykaz rysunków

Rysunek 1: Stała amplituda - fala bez sił oporu (opracowanie własne) 6
Rysunek 2: Zmiana amplitudy - fala z siłami oporu (opracowanie własne) 7
Rysunek 3: Długość fali (opracowanie własne) 7
Rysunek 4: Rozkład dźwięków w zależności od częstotliwości (opracowanie własne) 8
Rysunek 5: Krzywe fonów [5] 9
Rysunek 6: Widmo szumu (opracowanie własne) 10
Rysunek 7: Widma liniowe dźwięku tonalnego (opracowanie własne) 11
Rysunek 8: Metronom elektroniczny Boss DB-30 oraz metronom mechaniczny Cherub WSM 330 [6] 12
Rysunek 9: Sinusoida z punktami próbkowania na maksymalnych wychyleniach (opracowanie własne) 13
Rysunek 10: Sinusoida z punktami próbkowania na miejscach zerowego wychylenia (opracowanie własne) 13
Rysunek 11: Sinusoida z punktami próbkowania na niezerowych wychyleniach nie będącymi maksimami (opracowanie własne) 13
Rysunek 12: Udział w rynku mobilnych systemów operacyjnych na całym świecie w roku 2019 [11] 18
Rysunek 13: Cykl życia aktywności [16] 20
Rysunek 14: Wybór komponentu z paska narzędzi 22
Rysunek 15: Modyfikowanie właściwości komponentu „*Line Renderer*” 23
Rysunek 16: Modyfikowalne właściwości komponentu „*SpriteRenderer*” 23
Rysunek 17: Modyfikowalne właściwości komponenty „*Text”* 24
Rysunek 18: Modyfikowalne właściwości komponentu „*Button*” 24
Rysunek 19: Komponent „*Audio Listener*” 25
Rysunek 20: Modyfikowalne właściwości komponentu „*Audio Source*” 25
Rysunek 21: Modyfikowalne właściwości komponentu „*Input Field*” 26
Rysunek 22: Modyfikowalne właściwości komponentu „*Dropdown*” 27
Rysunek 23: Modyfikowalne właściwości komponentu „*Slider*” 28
Rysunek 24: Ustawienia budowania aplikacji (*BuildingSettings*) 29
Rysunek 25: Ustawienia projektu (*Project Settings*) 30
Rysunek 26: Kontrola wersji oparta na delcie [17] 31
Rysunek 27: Kontrola wersji GIT [17] 31

Rysunek 28: Zapisywanie danych na repozytorium lokalnym [19] 32
Rysunek 29: Scena *About* 35
Rysunek 30: Scena *Metronome* 35
Rysunek 32: Widok łączenia ścieżek sceny *Mixer* 35
Rysunek 33: Scena powitalna 35
Rysunek 31: Widok odtwarzania ścieżek sceny *Mixer* 35
Rysunek 34: Widmo pierwszej ścieżki 49
Rysunek 35: Widmo drugiej ścieżki 49
Rysunek 36: Widmo nałożonych na siebie ścieżek 50
Rysunek 37: Widmo ścieżek połączonych jedna za drugą 50
Rysunek 38: Widmo nałożonych na siebie ścieżek z nałożonym sygnałem metronomu 50
Rysunek 39: Widmo ścieżek połączonych jedna za drugą z nałożonym sygnałem metronomu 50

10. Wykaz tabel

Tabela 1: Nagłówek pliku WAV 15
Tabela 2: Oceny użytkowników 51

11. Wykaz listingów

Listing 1: Metoda SaveSeparatelyButton 36
Listing 2: Metoda *SaveMixedButton* 37
Listing 3: Metoda *SetSelectedAudioClips* 38
Listing 4: Tworzenie tablic danych pobranych z sampli w metodzie *Combine* 39
Listing 5: Tworzenie tablic danych pobranych z ampli w metodzie *MixAudioFiles* 39
Listing 6: Tworzenie klipu audio 39
Listing 7: Metoda *MixAndClampFloatBuffers* 40
Listing 8: Tworzenie klipu audio z sygnałem metronomu 41
Listing 9: Metoda *ClampToValidRange* 42
Listing 10: Metoda *Save* 42
Listing 12: Metoda *FileStreamCreateEmpty* 43
Listing 11: Metoda *ConvertAndWrite* 43
Listing 13: Metoda *WriteHeader* 44
Listing 14: Metody *FixedUpdate* i *LateUpdate* 45
Listing 15: Metoda *OnTick* 46
Listing 16: Metoda *MuteSound* 47
Listing 17: Metoda *PlayMusic* 48
Listing 18: Metoda *PauseMusic* 48