UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO"
CAMPUS DE BAURU
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MAYKON MICHEL PALMA

Áudio De Baixa Latência Em Dispositivos Móveis

Bauru - SP
2022

MAYKON MICHEL PALMA

Áudio De Baixa Latência Em Dispositivos Móveis

Trabalho de Conclusão de Curso do Curso de Bacharelado em Ciência da Computação da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Faculdade de Ciências, campus Bauru.

Orientador: Dr. Kleber Rocha de Oliveira

Bauru - SP
2022

Palma, Maykon Michel.
Áudio de baixa latência em dispositivos móveis / Maykon Michel Palma, 2022
38 f. : il.

Orientador: Dr. Kleber Rocha de Oliveira

Monografia (Graduação)–Universidade Estadual Paulista. Faculdade de Ciências, Bauru, 2022

1. Leitura (Ensino superior). 2. Livros e leitura. 3. Leitura – Meios auxiliares. I. Universidade Estadual Paulista. Faculdade de Ciências. II. Título.

Maykon Michel Palma

Áudio De Baixa Latência Em Dispositivos Móveis

Trabalho de Conclusão de Curso do Curso de Bacharelado em Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Prof Dr. Kleber Rocha de Oliveira
Orientador
Universidade Estadual Paulista
Faculdade de Engenharia e Ciências de Rosana
Coordenadoria do Curso de Engenharia de Energia

Profª Dra. Simone das Graças Domingues Prado
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Profª Dr. Renê Pegoraro
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Bauru, _____ de _____ de _____.

Dedico este projeto à minha família que sempre estive presentes em todos os momentos de minha formação. Dedico também à Kovver que me trouxe muito aprendizado como profissional. Por fim, dedico ao meu orientador que foi essencial para o desenvolvimento do TCC.

Agradecimentos

Em primeiro lugar, a Deus, pela minha vida, e por me permitir ultrapassar todos os obstáculos encontrados ao longo da realização deste trabalho, não me deixando desanimar em momento algum.

Agradeço minha família que esteve ao meu lado em todo meu processo acadêmico me dando todo suporte nas horas difícies e celebrando nas conquistas.

Agradeço todos os professores que fizeram parte e contribuiram com a minha formação.

Agradeço todos amigos pelo companheirismo, respeito, aprendizado, incentivo.

Agradeço a Kovver e todas as pessoas que passaram por lá. Por todo aprendizado como pessoa, pelos vínculos e as histórias criadas.

“Eu não falhei 10 mil vezes. Apenas encontrei 10 mil maneiras que não funcionam”.

(Thomas Edison)

Resumo

Desenvolver aplicativos voltados ao meio musical requer conhecimento em linguagens de baixo nível como C++, uma vez que as ferramentas de áudio disponibilizadas pelos sistemas operacionais de dispositivos móveis são limitadas. Linguagens de alto nível como Java e Swift demandam muito tempo entre o momento que o usuário pede para tocar um áudio e a execução pelo equipamento físico efetivamente. Dessa forma, programadores com pouca experiência tem extrema dificuldade para produzir sistemas do gênero. Com a biblioteca desenvolvida por este projeto, é possível gerar aplicativos React Native para Android e iOS com um único código, sem perder a baixa latência exigida pelos usuários.

Palavras-chave: Áudio. Baixa latência. React Native. Superpowered SDK.

Abstract

Developing applications aimed at the music environment requires knowledge in low-level languages such as C++, since the audio tools provided by mobile device operating systems are limited. High-level languages such as Java and Swift demand a lot of time between the moment the user asks to play an audio and the execution by the hardware effectively. Thus, programmers with little experience have extreme difficulty producing such systems. With the library developed by this project, it is possible to generate React Native applications for Android and iOS with a single code, without losing the low latency required by users.

Keywords: Audio. Low latency. React Native. Superpowered SDK.

Lista de ilustrações

Lista de abreviaturas e siglas

JS	JavaScript
JUCE	Jules' Utility Class Extensions
SDK	Software Development Kit
UML	Unified Modeling Language

Sumário

1 INTRODUÇÃO

Quando é preciso desenvolver uma aplicação para dispositivos móveis que envolva áudio, a primeira opção e de mais fácil implementação é utilizar as bibliotecas nativas integradas aos sistemas operacionais Android e iOS. Embora essa seja uma solução que resolve um simples toque de som, existem vários casos que isso não é o suficiente.

Ao tentar tocar uma música separada em duas faixas como instrumental e vocal, por exemplo, pode-se notar uma pequena falta de sincronização na composição. Talvez para um usuário comum isso não seja um problema, mas para aplicativos voltados ao público de músicos, com certeza essa dessincronização é inaceitável.

Isso ocorre porque o comando de tocar uma faixa não é executado instantaneamente, mas com uma latência variável de 20 a 30 milissegundos. Latência de áudio é o atraso de tempo conforme um sinal de áudio passa por um sistema. Muitas classes de aplicativos dependem de latências curtas para obter efeitos sonoros em tempo real (LAGO, 2004).

Esse é um problema recorrente que pode ser solucionado ao descer um pouco na abstração das linguagens de programação Kotlin para Android e Swift para iOS indo em direção à uma linguagem de mais baixo nível como o C++. Essa estratégia permite códigos mais velozes e otimizados, minimizando assim a latência.

Contudo, essa saída tem problemas relacionados às desvantagens da linguagem, principalmente referentes à velocidade de desenvolvimento e ocorrência de falhas por conta do gerenciamento manual de memória. A situação é agravada pela tendência de desenvolvedores especialistas em linguagens de programação de alto nível como Dart e JavaScript com bibliotecas e *frameworks* usados frequentimente como Flutter e React Native[1].

React Native é um *framework* de aplicativos móveis de código aberto criado pelo Meta, antigo Facebook. É usado para desenvolver aplicativos para Android, Android TV, iOS, macOS, tvOS, web, Windows e UWP, permitindo que os desenvolvedores usem a estrutura do React junto com as capacidades da plataforma nativa (EISENMAN, 2015).

Esse projeto faz uso do React Native para desenvolver uma biblioteca em TypeScript que serve de interface ao áudio de baixa latência implementada em linguagem de baixo nível. Assim, o código disponibilizado abertamente beneficia desenvolvedores que buscam controlar áudios de baixa latência em uma camada de abstração maior em linguagem de alto nível.

[1] https://reactnative.dev

2 PROBLEMA

Idealmente, um processador de sinal de áudio poderia gravar uma amostra, processá-la e reproduzir o resultado, fornecendo latência quase zero. Com exceção de alguns dispositivos profissionais, infelizmente isso não é possível na prática. Gravação e reprodução são realizadas por placas de som em blocos de amostras ou *buffers*. Embora o tamanho de um *buffer* geralmente possa ser configurado, a decomposição única do sinal em *buffers* introduz uma latência mínima ao sistema (JUILLET; ARISONA; SCHUBIGER-BANZ, 2007).

Além disso, linguagens de alto nível como Java acabam adicionando tempo extra ao pausar a execução do programa para limpar espaços de memória que não estão sendo utilizados através do chamado *garbage collector* (ECKEL, 2006). É esses poucos milissegundos que fazem linguagens de mais baixo nível como C, que também é conhecido pela rápida comunicação com o *hardware*, tornar-se uma opção mais interessante.

Contudo, não existe muito de especial em C, e é isso que a faz ser tão rápida. Linguagens mais novas que têm suporte para *garbage collector*, tipagem dinâmica e outras facilidades que tornam a experiência de escrever programas mais simples. O problema é que há sobrecarga de processamento adicional que degrada o desempenho do aplicativo. C não tem nada disso, mas exige que o programador seja capaz de alocar e liberar memória para evitar vazamentos e deva lidar com a tipagem estática de variáveis (WALTERS, 1996).

Sendo assim, para usufruir de áudios de baixa latência, é preciso que o desenvolvedor seja capaz de lidar com as desvantagens de linguagens de baixo nível, tornando tal tecnologia menos acessível. Hoje, é inviável que programadores com pouca experiência e enfoque em linguagens de alto nível como Java e Swift implementem sistemas voltados ao meio musical.

3 JUSTIFICATIVA

O presente projeto busca democratizar o desenvolvimento de aplicativos de áudio que dependam de latências curtas para atingir o objetivo do usuário como tocadores de múltiplas faixas, efeitos eletrônicos em guitarras e simuladores de instrumentos musicais. Ademais, agrega conhecimentos de diversas áreas ao discente: sonora e musical, linguagens de baixo e alto nível, desenvolvimento híbrido de aplicativos e biblioteca com interface para uso de terceiros.

4 OBJETIVO

Abaixo são apresentados o objetivo geral e objetivos específicos deste trabalho.

4.1 Objetivo geral

Desenvolver uma biblioteca que permita a utilização de áudio com baixa latência implementado em linguagem de programação de baixo nível, como C++ e Objective-C, com interface única para dispositivos móveis com os sistemas operacionais Android e iOS em mais alto nível, como a linguagem de programação TypeScript com o *framework* React Native.

4.2 Objetivos específicos

1) Expor soluções para áudio de baixa latência
2) Apresentar a implementação uma das soluções em uma linguagem de baixo nível
3) Conectar essa implementação aos sistemas operacionais Android e iOS
4) Desenvolver uma interface em alto nível para o áudio de baixa latência
5) Avaliar o ganho de velocidade no desenvolvimento
6) Disponibilizar uma biblioteca de código aberto para a comunidade
7) Exemplificar o uso em um simples aplicativo React Native

5 MÉTODO DE PESQUISA

Para realização desse trabalho, inicialmente foi necessário identificar soluções para áudio de baixa latência. Esse passo facilitou o desenvolvimento em uma linguagem de mais alto nível. Além disso, tal técnica deveria poder ser aplicada em ambientes móveis, mais especificamente Android e iOS, que são os sistemas operacionais de enfoque.

Uma possibilidade era o uso do Jules' Utility Class Extensions, conhecido como JUCE[1], um *framework* de aplicativos C++ *cross-platform* (para diversas plataformas) parcialmente de código aberto, usado para o desenvolvimento de aplicativos *desktop* e móveis. Apesar de visar um propósito mais geral, o JUCE detém de ferramentas para o desenvolvimento de áudio de baixa latência e alta performance que tornam a experiência do desenvolvedor mais simples (ROBINSON, 2013).

A principal vantagem da utilização da tecnologia reside na sua gratuidade para sistemas com receitas de até 50 mil dólares por ano. Além disso, é extensível, possibilitando trabalhar com efeitos não nativos e possui grande comunidade com fórum ativo e tutoriais desde o básico até casos de uso bem avançados. Contudo, o código JUCE é muito verboso, no qual um simples programa já requer muito código. Por ser de uso geral, carrega código desnecessário ao desenvolvimento de aplicativos musicais, sendo grande demais para dispositivos com pouco memória interna (ROBINSON, 2013).

Existem também, soluções privadas como a Superpowered, mais exatamente a Superpowered Audio SDK[2], do inglês *kit* de desenvolvimento de *software* de áudio Superpowered. Se trata de uma ferramenta comercial e corporativa, precisando de licença mesmo em produtos sem renda e sua comunidade é quase inexistente. O suporte também é pago e muito precário de informações e tempo de resposta. Ademais, o código é fechando em quase sua totalidade, inflexível e não extensível.

Por outro lado, a Superpowered Audio SDK tem como foco apenas o áudio. Isso leva a vantagens como otimizações para baixo consumo de memória, processamento e bateria, junto à uma biblioteca leve em *kilobytes* adicionados ao aplicativo. Em função disso, grandes empresas como Spotify, Microsoft e Voloco utilizam a Superpowered em seus produtos reconhecidos pelo público. Portanto, é a partir desta que o projeto visará os objetivos.

A partir do *kit* de desenvolvimento de *software* escolhido, implementou-se uma das soluções em linguagem de baixo nível: C++. Então conectou-se aos sistemas operacionais móveis mais populares, Android e iOS, a partir de suas linguagens nativas, Kotlin e Objective-C respectivamente. Por fim, um *framework* de aplicativos híbridos, arbitrariamente escolhido React Native, conectou todos os componentes e interface de alto nível ao áudio de curta latência em forma de biblioteca.

Para avaliar o ganho de velocidade no desenvolvimento, foi codificado um pequeno

[1] http://juce.com/

[2] https://superpowered.com/audio-overview

aplicativo React Native com a biblioteca criada capaz de tocar faixas simultaneamente e aplicar alguns efeitos no áudio.

6 DESENVOLVIMENTO

Como o presente projeto trata de uma biblioteca para auxiliar aplicações react-native desenvolvidas por outros programadores, foi utilizado uma *interface* de linha de comando para geração de código automático chamada de React Native Builder Bob[1]. Assim, não foi necessário escrever manualmente o código responsável por habilitar o desenvolvimento em C++ nas plataformas nativas Android e iOS. Além disso, o projeto já fica configurado com ferramentas para análise estática de código, como ESlint[2] e Prettier[3], somado a integração e entrega contínuas por meio do Husky[4], Release-it[5] e CircleCI[6].

Para configurar automaticamente o projeto para usar react-native-builder-bob, o seguinte comando é executado no terminal do computador:
npx create-react-native-library react-native-superpowered-sdk

Com isso, uma série de perguntas é iniciada pelo *script* a fim de configurar corretamente o projeto para o caso de uso específico, documentadas e traduzidas para o português abaixo:

- **Qual é o nome do pacote npm?** *react-native-superpowered-sdk*
- **Qual é a descrição do pacote?** *Integração de Superpowered com React Native*
- **Qual é o nome do autor do pacote?** *maykonmichel*
- **Qual é o endereço de e-mail do autor do pacote?** *maykonmichelpalma@gmail.com*
- **Qual é o URL do autor do pacote?** *https://github.com/maykonmichel*
- **Qual é o URL do repositório?** *https://github.com/maykonmichel/react-native-superpowered-sdk*
- **Quais linguagens você deseja usar?** *C ++ para iOS e Android*

Ao final do questionário, uma mensagem de sucesso: "**Projeto criado com sucesso em react-native-superpowered-sdk!**" apareceu indicando que tudo aconteceu como esperado.

Por se tratar de uma biblioteca de código aberto, um repositório *git* foi criado no endereço *https://github.com/maykonmichel/react-native-superpowered-sdk* e associado ao projeto recém criado.

[1] https://github.com/callstack/react-native-builder-bob
[2] https://eslint.org
[3] https://prettier.io
[4] https://github.com/typicode/husky
[5] https://github.com/release-it/release-it
[6] https://circleci.com

Um dos pontos fortes do *react-native-builder-bob* é, além da biblioteca que será instalada por outros desenvolvedores, gerar uma pasta *example* com um aplicativo react-native já configurado para utilizar a biblioteca e facilitar os testes durante o desenvolvimento. O exemplo disponibilizado é uma simples função de multiplicação feita na camada do C++.

Executando o comando *react-native run-android* e *react-native run-ios* no terminal instala-se uma simples aplicação nos emuladores Android e iOS que mostram *Result: 21* no centro da tela. Ao observar-se o código responsável por esse resultado, vê-se que uma função JavaScript *multiply* está sendo chamada com os argumentos 3 e 7 direcionando a outra função de igual nome e parâmetros, mas escrita em C++ que cuida de efetuar o cálculo e retornar a solução para a camada acima mostrar ao usuário.

Ainda que sem efeito prático, esse modelo simples é didático o suficiente para exemplificar a técnica utilizada para alcançar a execução do áudio de baixa latência em uma linguagem de alto nível. Uma função JavaScript será declarada com os mesmos nomes e parâmetros de outra função C++, responsável por todo processamento pesado que o JS não seria capaz de executar com desempenho equivalente. Se necessário, também é possível retornar dados ao mais alto nível para que este o mostre na tela.

Ainda que muito tenha sido automaticamente gerado pelo *script* do *react-native-builder-bob*, uma configuração no lado Android ainda precisa ser feita antes de qualquer ponto relacionado a Superpowered SDK. Ao selecionar "*C++ para iOS e Android*" na pergunta "**Quais linguagens você deseja usar?**" durante o questionário inicial, o projeto é criado em Java na camada do Android. De modo a utilizar a recomendação do Google, detentora do sistema operacional móvel mais utilizado no mundo, todo código Java foi convertido para Kotlin com uma funcionalidade disponível no Android Studio, ambiente de desenvolvimento integrado oficial para Android da JetBrains' IntelliJ IDEA.

Como mencionado anteriormente, a Superpowered SDK é uma ferramenta comercial parcialmente de código aberto. Para utilizar as funcionalidades disponíveis pela tecnologia, copiam-se as bibliotecas estáticas e cabeçalhos disponíveis na pasta *superpowered/* do repositório da Superpowered SDK no GitHub[7]. Contudo, visando minimizar o tamanho gasto pela biblioteca e otimizar o aplicativo final, apenas os arquivos relacionados às plataformas Android e iOS foram incluídos, enquanto os sistemas operacionais Linux, Windows e macOS foram ignorados.

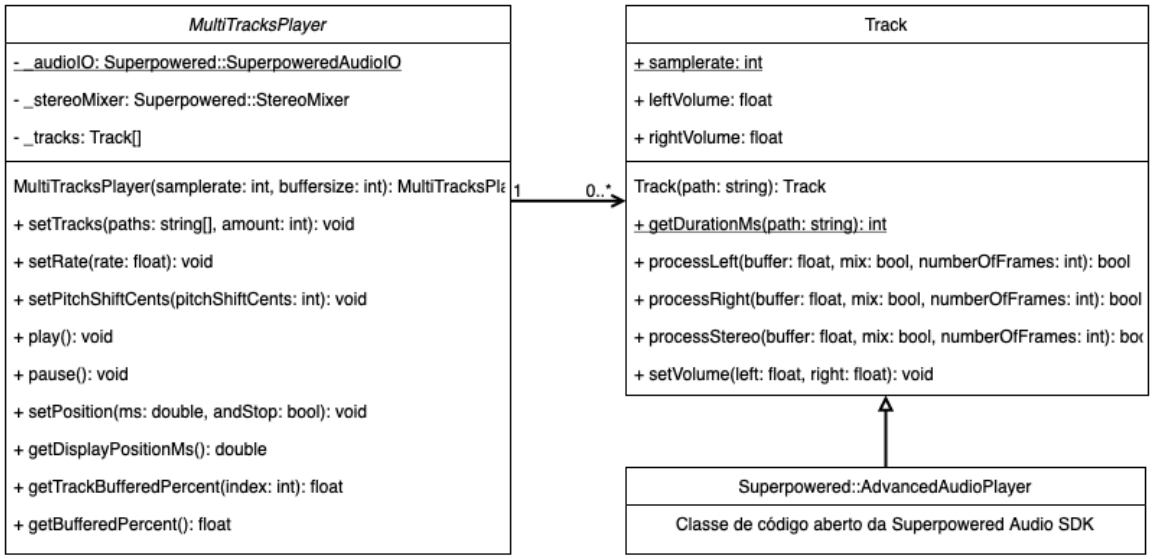
A biblioteca final desse projeto permite que o aplicativo exemplo seja capaz de tocar diversas faixas de áudio simultaneamente com baixa latência entre elas de tal forma que um músico não consiga distinguir a divergência de cada instrumento. O usuário final tem a possibilidade de alterar a distribuição do som para esquerda ou direita (*panning)* e modificar volume ou silenciar completamente cada faixa individualmente.

Para tanto, duas classes foram desenvolvidas em C++: *Track (apêndice A)* para

[7] https://github.com/superpoweredSDK/Low-Latency-Android-iOS-Linux-Windows-tvOS-macOS-Interactive-Audio-Platform/tree/master/Superpowered

cada faixa individual e *MultiTracksPlayer (apêndice B)* para o controle da obra na totalidade diagramadas na figura 1 na linguagem de modelagem unificada, do inglês UML. A primeira estende da classe *AdvancedAudioPayer* da Superpowered SDK com os métodos *processLeft, processRight* e *processStereo* para processamento de áudio nos canais da esquerda e direita individualmente ou simultaneamente permitindo o *panning*. Também expõe o método *setVolume* para alteração do volume e outro estático *getDurationMs* para retornar a duração de uma faixa em milissegundos. O construtor tem como parâmetro *path*, o caminho para a faixa desejada.

Figura 1 – UML das classes desenvolvidas



Elaborado pelo autor

MultiTracksPlayer no que lhe concerne tem um método *setTracks* responsável por receber um *array* de caminhos e criar cada *Track* relativo. *setRate* e *setPitchShiftCents* permitem alterar respectivamente a velocidade e tom musical. *play* e *pause* iniciam e param a reprodução do áudio. *setPosition* e *getDisplayPositionMs* definem e obtêm a posição atual de todas as faixas dado em milissegundos. *getTrackBufferedPercent* e *getBufferedPercent* retornam a porcentagem do áudio carregado por faixa e total.

Essas classes de baixo nível foram conectadas ao sistema operacional Android pela *interface* nativa Java, JNI do inglês, com Kotlin e ao iOS com Objective-C. Através do sistema de módulos nativos do react native as instâncias das classes Kotlin, Objective-C e C++ (nativas) são expostas para JavaScript como objetos JS, permitindo assim a execução de código nativo arbitrário de dentro do JS.

Para auxiliar ainda mais o desenvolvimento no alto nível, todos os tipos dos métodos e respectivos parâmetros foram declarados em TypeScript. Além disso, um *hook* - conceito do React.JS para compartilhamento de código - facilita a inicialização da biblioteca no ciclo

de vida dos componentes, criando uma instância e limpando a memória quando necessário.

Na aplicação exemplo, foi desenvolvido uma única tela com uma listagem das faixas da música Howlin da Amber Skye disponível para baixar gratuitamente em https://www.pure mix.net/zelab-mixing-contest/zelab-session-5-howlin.html. Cada faixa pode ser silenciada instantaneamente, ter o volume alterado e distribuição do som à esquerda ou direita de forma singular a partir de botões presentes na linha de cada instrumento. Ademais, um botão na parte inferior controla a reprodução da música, inciando-a ou parando-a.

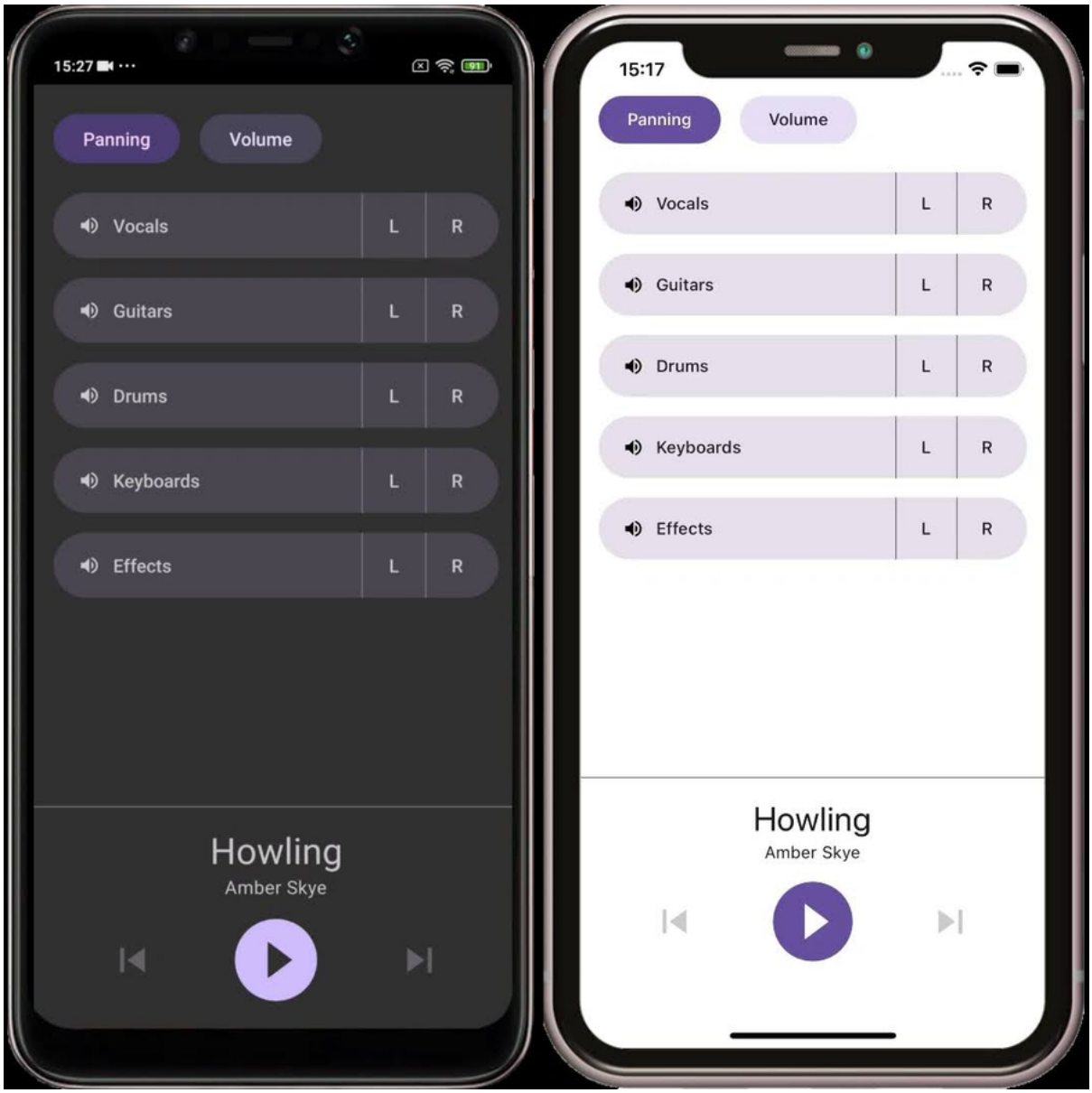
7 RESULTADOS E DISCUSSÃO

Dois principais produtos resultaram deste projeto: uma biblioteca de áudio de baixa latência em react native e um aplicativo Android e iOS para escutar e mixar músicas. O primeiro conclui a proposta do trabalho e o segundo auxilia na avaliação do ganho de velocidade no desenvolvimento. Além disso, a latência do áudio pela interface criada é imperceptível.

A versão final denominada *1.0.0* se encontra na aba *Releases*[1] do repositório GitHub anteriormente mencionado. Nela também é possível baixar o código para rodar o projeto e gerar os arquivos *.apk* e *.ipa* para instalação do aplicativo nos sistemas operacionais Android e iOS respectivamente. A biblioteca por sua vez, pode ser visualizada na aba *Packages*[2] do mesmo repositório. Além disso, um curto vídeo foi gravado demonstrando todas as funcionalidades presentes no aplicativo exemplo.

[1] https://github.com/maykonmichel/react-native-superpowered-sdk/releases/tag/v1.0.0
[2] https://github.com/maykonmichel/react-native-superpowered-sdk/packages/1197708

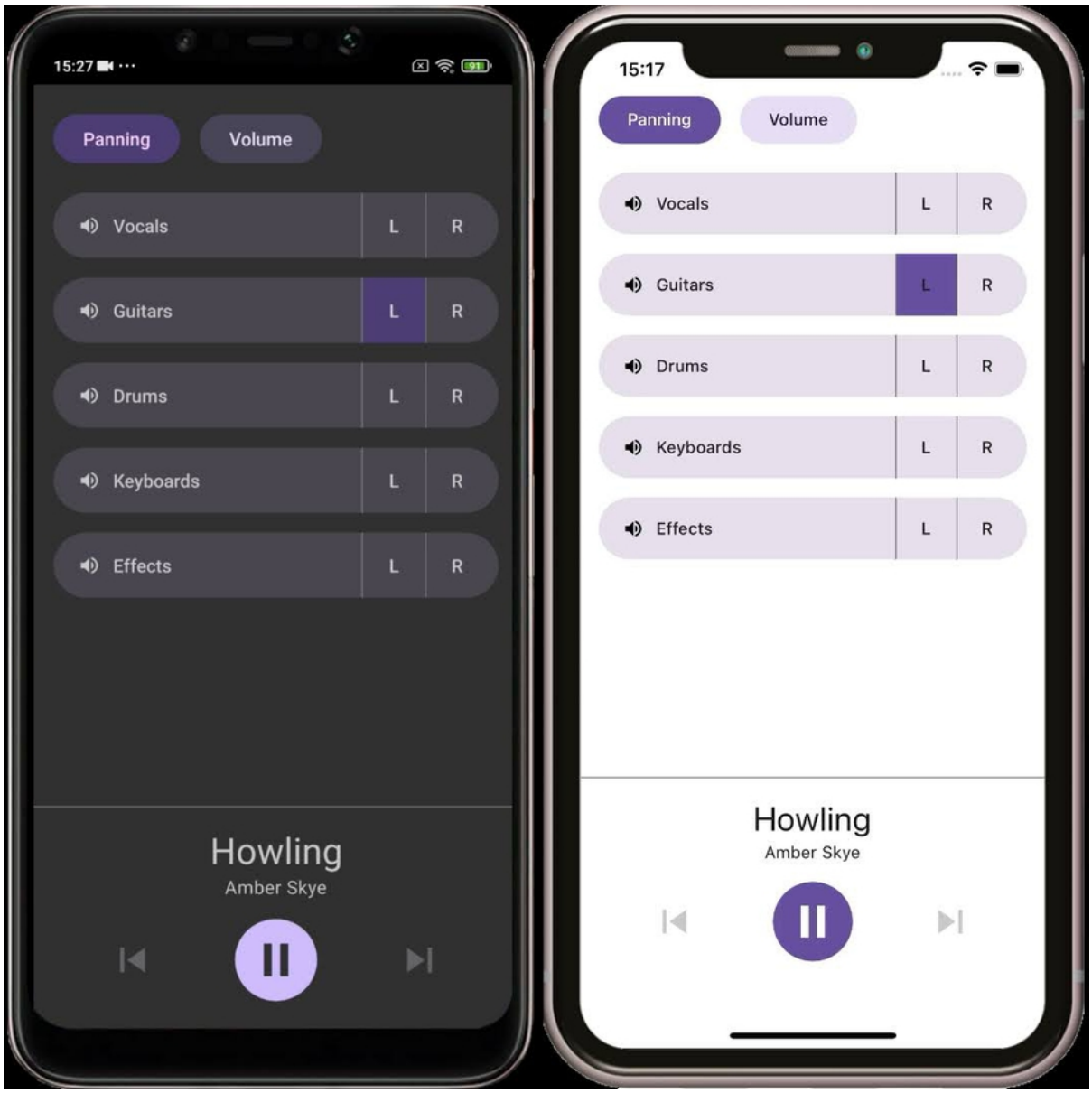
Figura 2 – Aplicativo exemplo



Elaborado pelo autor

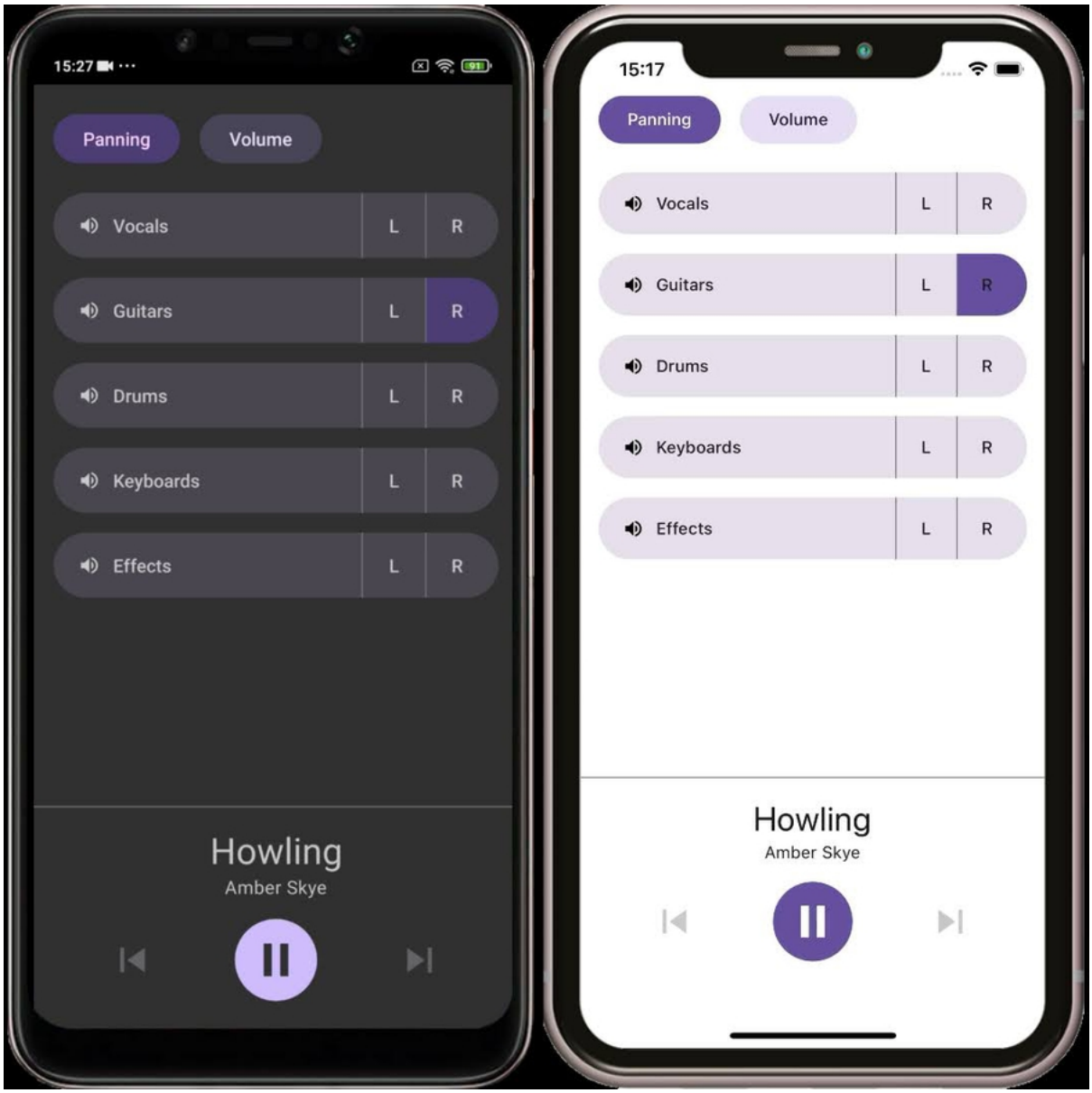
A figura 2 mostra o início da gravação, sendo do lado esquerdo um dispositivo Android com modo escuro ativado e do lado direito um iPhone 13 com iOS 15. A música carregada é a *Howling* da artista *Amber Skye* que foi dividida em 5 faixas: vocal, guitarras, baterias, teclados e efeitos.

Figura 3 – Espacialização esquerda da faixa guitarra



Elaborado pelo autor

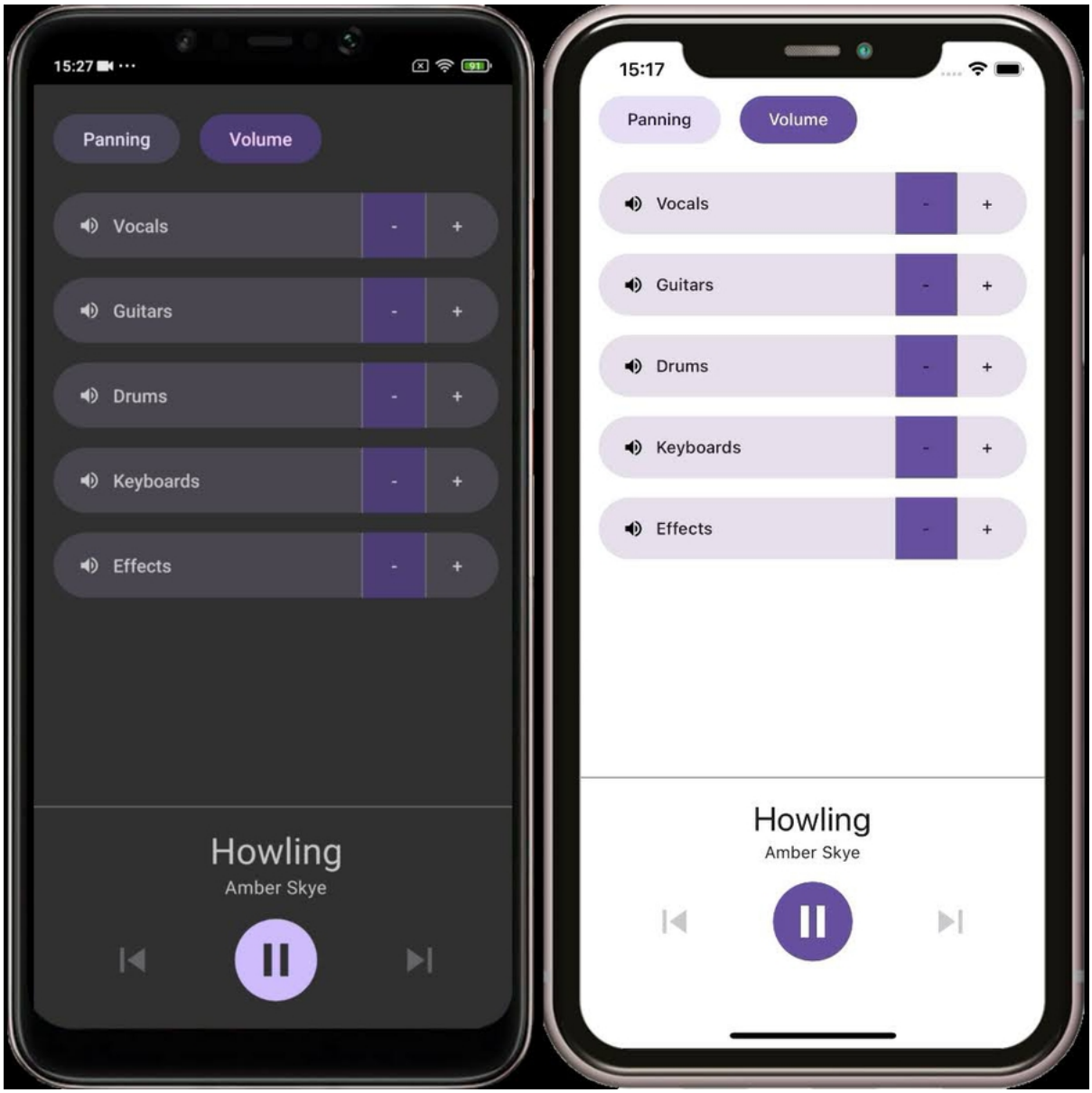
Figura 4 – Espacialização direita da faixa guitarra



Elaborado pelo autor

A princípio, ve-se no topo da tela que a aba *Panning* está selecionada e que cada faixa conta com um ícone de áudio ativado, o nome em inglês e os botões L/R. A música se inicia e então o *panning* esquerdo da faixa da guitarra, figura 3, é ativada localizando o instrumento apenas do lado esquerdo no áudio estéreo. Além disso, o botão L ganha um fundo de cor diferente indicando visualmente que está selecionado. Logo após, o botão R é pressionado, ganhando a cor roxa enquanto o botão L volta ao estado inicial, figura 4, indicando o *panning* direito.

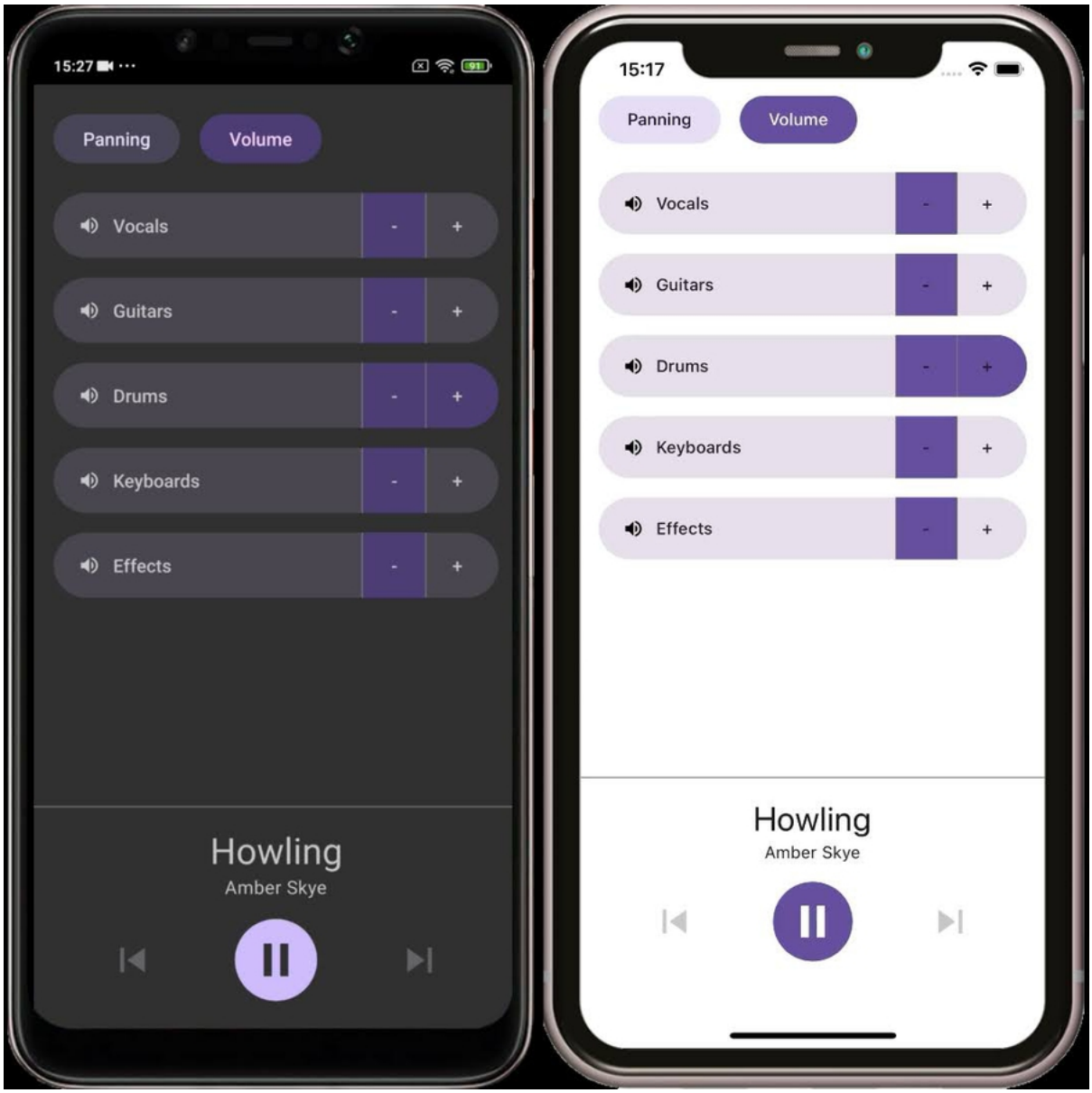
Figura 5 – Aba volume



Elaborado pelo autor

A figura 5 mostra o aplicativo na aba volume, alterando os botões de cada faixa de L/R para -/+. Além disso, os botões com símbolo negativo ganham fundo colorido para mostrar que podem ser pressionados enquanto os botões com símbolo positivo permanecem nas cores neutras. Assim, percebe-se que é apenas possível diminuir o volume das faixas no momento, uma vez que estas se encontram em sua altura máxima.

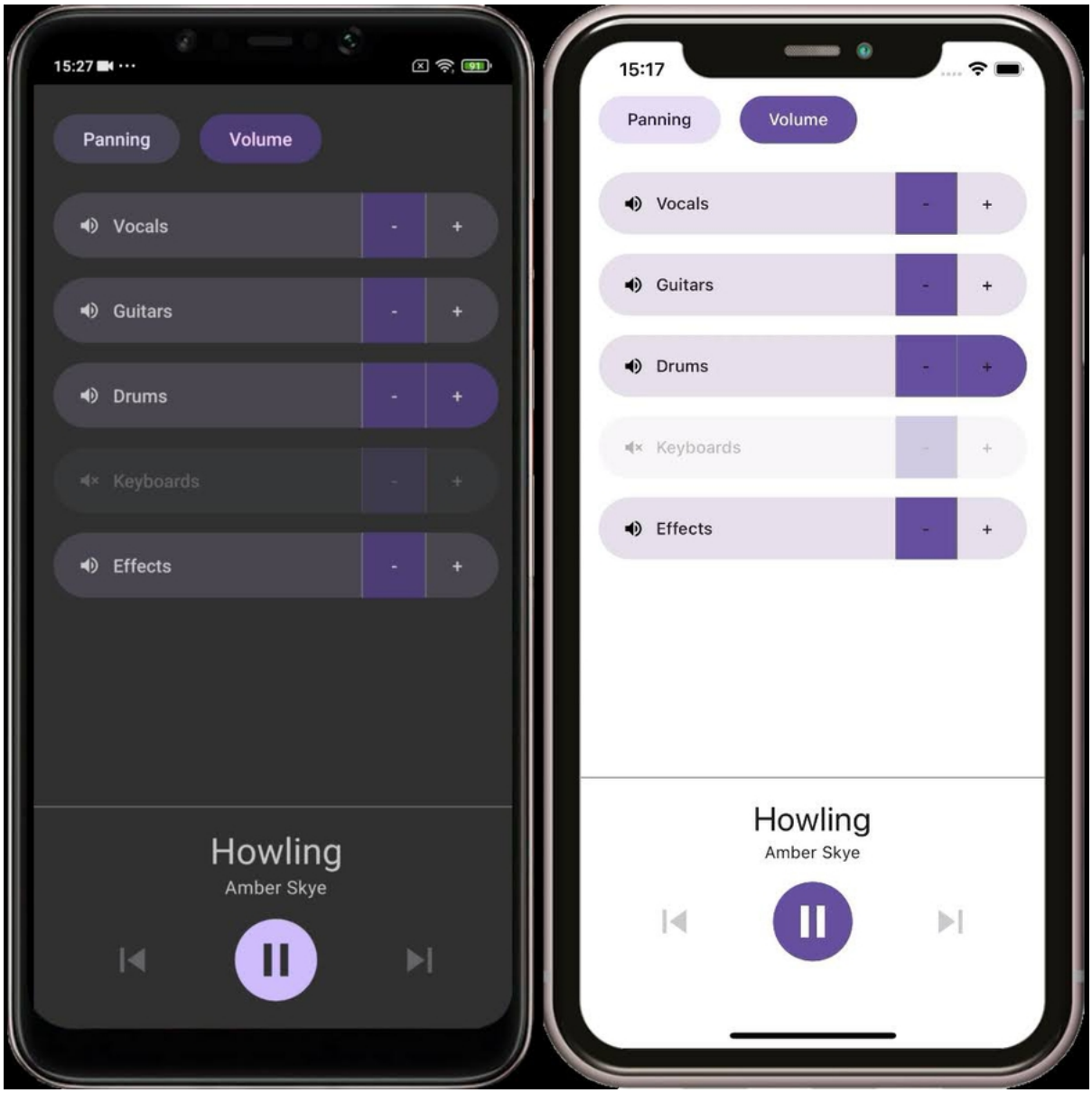
Figura 6 – Faixa bateria com volume reduzido



Elaborado pelo autor

Instantes após a troca de funcionalidade, a faixa da bateria tem o volume diminuído parcialmente enquanto o seu botão + fica no estado habilitado como mostra a figura 6.

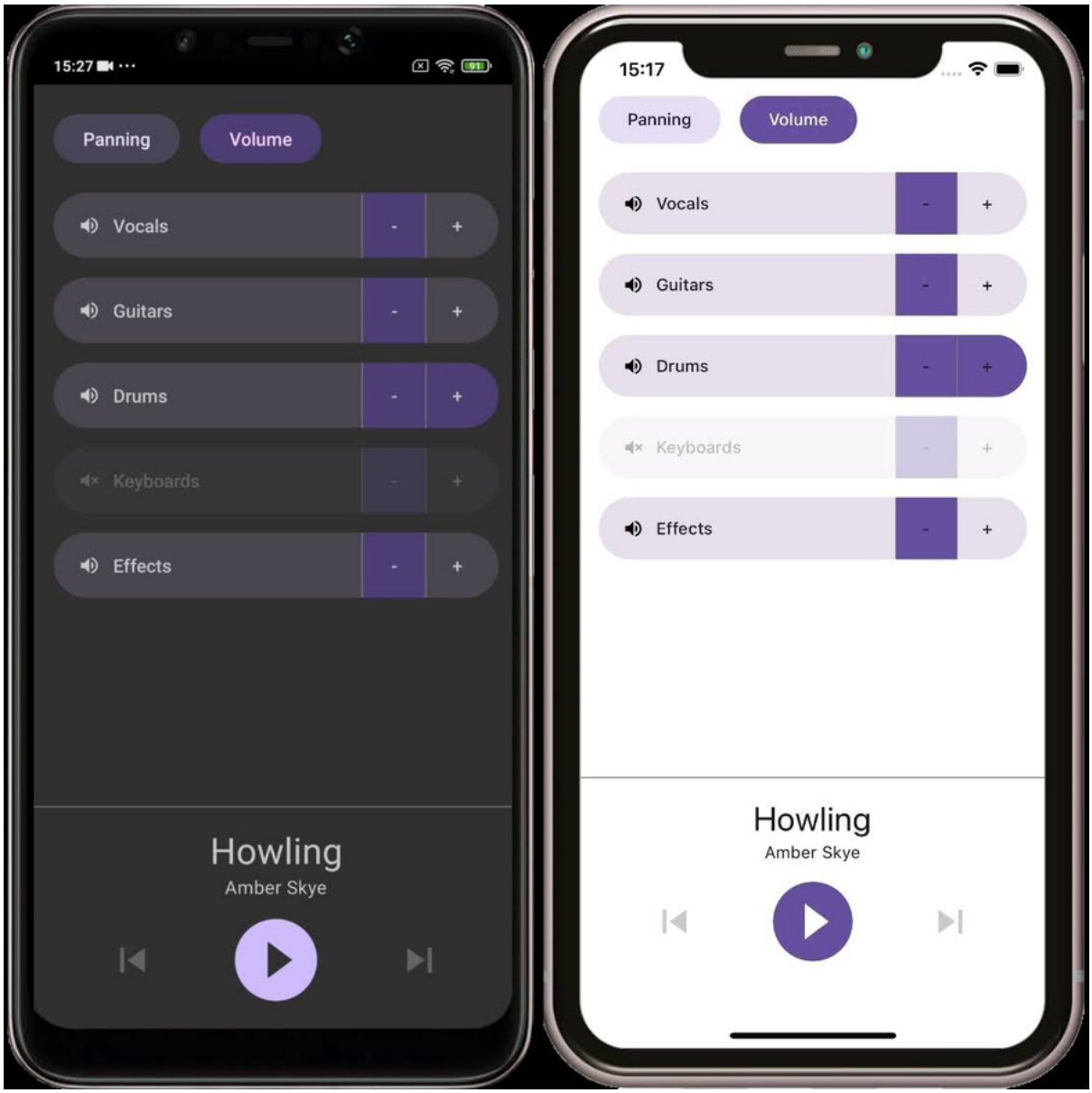
Figura 7 – Faixa teclado silenciada



Elaborado pelo autor

Também é possível silenciar uma faixa completamente. Pode-se observar isso na prática na figura 7, na qual os teclados estão com opacidade reduzida e o ícone da linha indica que o instrumento não está sendo reproduzido.

Figura 8 – Música pausada



Elaborado pelo autor

Por fim, a música é brevemente pausada na figura 8, mostrando graficamente através do ícone inferior central. Outros botões estão presentes no exemplo, indicando que poderia haver músicas anteriores e posteriores, mas se encontram desabilitados. Estes servem apenas para harmonização da interface, sem real funcionalidade, mas poderiam ser facilmente implementados em versões futuras.

8 CONCLUSÃO

O avanço das tecnologias para aumento da velocidade de desenvolvimento de aplicativos e maior entendimento do código leva a perdas de desempenho aceitáveis na maioria dos sistemas. Contudo, certos casos de uso necessitam de programas mais específicos e otimizados que usualmente exigem maior conhecimento e controle do desenvolvedor. Contudo, é possível mesclar ambos os cenários de modo a utilizar o melhor do dispositivo físico, sem perder as facilidades trazidas pela evolução.

A Superpowered Audio SDK é uma ferramenta poderosa, mas pouco acessível a iniciantes. Com a biblioteca gerada pelo presente projeto, programadores pouco experientes são capazes de desenvolver aplicativos para o meio musical. Não obstante, é possível estender as funcionalidades implementadas nesse projeto limitado apenas ao disponibilizado pela Superpowered.

Referências

ECKEL, B. **Thinking in Java**. 4. ed. [S.l.]: Prentice Hall PTR, 2006. 1482 p.

EISENMAN, B. Learning react native: Building native mobile apps with JavaScript. O'Reilly Media, Inc, v. 255, p. – 1, 2015. Disponível em: https://books.google.com.br/books?hl=pt-BR&lr=&id=274fCwAAQBAJ&oi=fnd&pg=PR2&dq=+Learning+React+Native+BUILDING+NATIVE+MOBILE+APPS+WITH+JAVASCRIPT&ots=tFxpbElal-&sig=FxiAItc73fOjnjs8XuORa_pS3SU#v=onepage&q=Learning%20React%20Native%20BUILDING%20NATIVE%20MOBILE%20APPS%20WITH%20JAVASCRIPT&f=false. Acesso em: 11 mar.2020.

JUILLET, N.; ARISONA, S. M.; SCHUBIGER-BANZ, S. REAL-TIME, LOW LATENCY AUDIO PROCESSING IN JAVA. In: **International Computer Music Conference**. São Paulo: [s.n.], 2007. Disponível em: https://diuf.unifr.ch/main/pai/sites/diuf.unifr.ch.main.pai/files/publications/2007_Juillerat_Mueller_Schubiger-Banz_Real_Time.pdf. Acesso em: 14 abr. 2021.

LAGO, N. P. **"Processamento distribuído de áudio em tempo real"**. 2004. Dissertação (Mestrado) — Universidade de São Paulo. Disponível em: http://www.teses.usp.br/teses/disponiveis/45/45134/tde-05102004-154239/.

ROBINSON, M. **Getting Started with JUCE**. [S.l.]: Packt Publishing, 2013. 158 p.

WALTERS, D. E. C Programming - A Modern Approach. By K. N. King. W. W. Norton & Company: New York, 1996, 661 pp, ISBN 0-393-96945-2. **Journal of Chemical Information and Computer Sciences**, v. 36, n. 5, p. 1050 –, 1996. Disponível em: http://dx.doi.org/10.1021/ci960432m.

APÊNDICE A – Track.cpp

A classe *Track* foi desenvolvida em C++ e estende da *AdvancedAudioPlayer* disponibilizada pela *Superpowered.* É responsável por controlar uma única faixa de áudio abstraindo conceitos de áudio de baixo nível.

```
#include "Track.h"

int Track::getDurationMs(const char *path) {
  auto player = new Superpowered::AdvancedAudioPlayer(44100, 0);
  player->open(path);

  usleep(5000);

  auto duration = player->getDurationMs();
  delete player;

  return duration;
}

int Track::samplerate;

Track::Track(const char *path) : AdvancedAudioPlayer(samplerate, 0) {
  leftVolume = rightVolume = 1.0f;
  open(path);
}

bool Track::processLeft(float *buffer, bool mix, unsigned int numberOfFrames) {
  return AdvancedAudioPlayer::processStereo(buffer, mix, numberOfFrames, leftVolume
}

bool Track::processRight(float *buffer, bool mix, unsigned int numberOfFrames) {
  return AdvancedAudioPlayer::processStereo(buffer, mix, numberOfFrames, rightVolum
}

bool Track::processStereo(float *buffer, bool mix, unsigned int numberOfFrames) {
  return processLeft(buffer, mix, numberOfFrames);
}

void Track::setVolume(float left, float right) {
```

```
  leftVolume = left;
  rightVolume = right;
}
```

APÊNDICE B – MultiTracksPlayer.cpp

A classe *MultiTracks* armazena e controla múltiplas faixas, garantido que sejam corretamente unificadas em um sinal de áudio para o hardware.

```
#include "MultiTracksPlayer.h"

SuperpoweredAndroidAudioIO *MultiTracksPlayer::_audioIO;

void MultiTracksPlayer::onBackground() {
  _audioIO->onBackground();
}

void MultiTracksPlayer::onForeground() {
  _audioIO->onForeground();
}

static bool audioProcessing(
  void *__unused clientdata,
  short int *audioIO,
  int numFrames,
  int samplerate
) {
  return ((MultiTracksPlayer *) clientdata)
    ->process(audioIO, (unsigned int) numFrames, (unsigned int) samplerate);
}

MultiTracksPlayer::MultiTracksPlayer(int samplerate, int buffersize) {
  Track::samplerate = samplerate;

  _stereoMixer = new Superpowered::StereoMixer();
  _stereoMixer->inputGain[1] = _stereoMixer->inputGain[2] = 0;

  _audioIO = new SuperpoweredAndroidAudioIO(
    samplerate,
    buffersize,
    false,
    true,
    audioProcessing,
    this,
```

```
    -1,
    SL_ANDROID_STREAM_MEDIA
  );
}

MultiTracksPlayer::~MultiTracksPlayer() {
  delete _audioIO;

  for (auto iterator = _tracks.rbegin(); iterator != _tracks.rend(); iterator++) {
    auto track = *iterator;
    delete track;
    _tracks.pop_back();
  }
}

bool MultiTracksPlayer::process(
  short int *output,
  unsigned int numberOfFrames,
  unsigned int samplerate
) {
  for (auto track : _tracks) {
    track->outputSamplerate = samplerate;
    track->syncMode = Superpowered::SyncMode_TempoAndBeat;
  }

  float buffer[3][numberOfFrames * 8 + 64];
  bool silence[3];

  silence[0] = silence[1] = silence[2] = true;

  for (auto track : _tracks) {
    auto hasLeftVolume = track->leftVolume > 0;
    auto hasRightVolume = track->rightVolume > 0;
    auto hasBothVolumes = hasLeftVolume && hasRightVolume;

    if (hasBothVolumes) {
      if (track->processStereo(buffer[2], !silence[2], numberOfFrames))
        silence[2] = false;
    } else if (hasRightVolume) {
```

```
      if (track->processRight(buffer[1], !silence[1], numberOfFrames))
        silence[1] = false;
    } else if (track->processLeft(buffer[0], !silence[0], numberOfFrames)) {
        silence[0] = false;
    }
  }

  _stereoMixer->process(
    silence[0] ? nullptr : buffer[0],
    silence[1] ? nullptr : buffer[1],
    silence[2] ? nullptr : buffer[2],
    nullptr,
    buffer[0],
    numberOfFrames
  );

  silence[0] &= silence[1] && silence[2];

  if (!silence[0])
    Superpowered::FloatToShortInt(buffer[0], output, numberOfFrames);
  return !silence[0];
}

void MultiTracksPlayer::setTracks(const char **paths, int amount) {
  int i;

  for (i = 0; i < amount && i < _tracks.size(); i++)
    _tracks[i]->open(paths[i]);

  for (; i < amount; i++)
    _tracks.push_back(new Track(paths[i]));

  for (auto it = _tracks.rbegin(); it != _tracks.rend() && _tracks.size() != amount
    auto track = *it;
    delete track;
    _tracks.pop_back();
  }
}
```

```
float MultiTracksPlayer::getTrackBufferedPercent(unsigned int index) {
  try {
    return _tracks.at(index)->getBufferedEndPercent() * 100;
  } catch (std::out_of_range &outOfRange) {
    return 0;
  }
}

float MultiTracksPlayer::getBufferedPercent() {
  float bufferedPercent = 0.0f;

  for (auto track : _tracks) bufferedPercent += track->getBufferedEndPercent();

  return bufferedPercent / _tracks.size() * 100;
}

double MultiTracksPlayer::getDisplayPositionMs() {
  return _tracks.front()->getDisplayPositionMs();
}

void MultiTracksPlayer::setPosition(double ms, bool andStop) {
  for (auto track : _tracks)
    track->setPosition(ms, andStop, true, false, true);
}

void MultiTracksPlayer::play() {
  for (auto track : _tracks) {
    if (track == _tracks.front()) track->play();
    else track->playSynchronized();
  }

  Superpowered::CPU::setSustainedPerformanceMode(true);
}

void MultiTracksPlayer::pause() {
  for (auto track : _tracks) track->pause();

  Superpowered::CPU::setSustainedPerformanceMode(false);
}
```

```
void MultiTracksPlayer::setTrackVolume(int index, float left, float right) {
  _tracks.at(index)->setVolume(left, right);
}

void MultiTracksPlayer::setVolume(float value) {
  _stereoMixer->outputGain[0] = _stereoMixer->outputGain[1] = value;
}

void MultiTracksPlayer::setRate(float rate) {
  for (auto track : _tracks) track->playbackRate = rate;
}

void MultiTracksPlayer::setPitchShiftCents(int pitchShiftCents) {
  for (auto track : _tracks) track->pitchShiftCents = pitchShiftCents;
}
```