UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ARTUR DE ALMEIDA SCHEIBLER

Syntherface: uma interface Android para sintetizadores no Arduino

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Marcelo de Oliveira Johann

Porto Alegre
2018

“When I come to my own beliefs, I find myself quite unable to discern any purpose in the universe, and still more unable to wish to discern one.”

— BERTRAND RUSSELL

AGRADECIMENTOS

Eu gostaria de agradecer aos meus pais pelo suporte que me deram e que possibilitou que eu chegasse até aqui. Também, ao meu orientador, pela ajuda e esclarecimentos quanto a esse trabalho. Além disso, agradeço muito à minha namorada pelo apoio e ajuda que foram de extrema importância para minha saúde mental durante a confecção desse trabalho. Por fim, gostaria de agradecer aos meus amigos, por serem quem eles são.

Eu também gostaria de agradecer ao LaTeX Project e à comunidade envolvida pela ajuda com a formatação do texto.

RESUMO

Sintetizadores musicais são programas ou peças de hardware que criam timbres sonoros. Esses timbres podem, então, ser tocados utilizando outro instrumento, como um teclado, e usados para composição musical. Placas microcontroladoras como as do Arduino — um projeto de hardware aberto — são uma escolha de hardware para hospedar sintetizadores. A interação com esses se dá através de componentes físicos conectados à placa como botões, *knobs*, *sliders*, entre outros. A proposta desse trabalho é criar uma interface Android virtual que usa Bluetooth para se comunicar com sintetizadores em Arduinos. Dessa forma, além do custo com componentes de interface cair, cria-se um maior potencial de interação com os sintetizadores.

Palavras-chave: Android. sintetizador. arduino. bluetooth.

Syntherface: an Android interface for Arduino synthesizers

ABSTRACT

Musical synthesizers are programs or pieces of hardware that create sound timbres. These timbres can then be played using another instrument, like a keyboard, and used for musical composition. Microcontroller boards such as those from Arduino — an open-source hardware project — are a hardware choice for hosting synthesizers. The interaction with these boards happens through widgets connected to it, like buttons, knobs, slidets, etc. The goal of this work is to create a virtual Android interface which uses Bluetooth to communicate with synthesizers on Arduinos. This way, not only the cost attributed to interface widgets is minimized, but also the potential of interaction with the synthesizers is increased.

Keywords: Android. arduino. synthesizer. bluetooth.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

FM	Frequency Modulation (Modulação de Frequência)
IDE	Integrated Development Environment (Ambiente Integrado de Desenvolvimento)
IEEE	Institute of Electrical and Electronics Engineers (Instituto de Engenheiros Elétricos e de Eletrônica)
MIDI	Musical Instrument Digital Interface (Interface Digital de Instrumento Musical)
SDK	Software Development Kit (Kit de Desenvolvimento de Software)
SDP	Service Discovery Protocol (Protocolo de Descoberta de Serviços)
USB	Universal Serial Bus (Barramento Serial Universal)
UUID	Universally Unique Identifier (Identificador Universalmente Único)

SUMÁRIO

1 INTRODUÇÃO

Sintetizadores são instrumentos musicais eletrônicos. Diferentes sintetizadores geram (sintetizam) diferentes timbres. Alguns procuram imitar sons conhecidos, como os sons feitos por outros instrumentos (como piano, saxofone ou flautas) ou outros sons cotidianos (o som do vento, de uma tosse humana, etc). Outros produzem timbres não necessariamente relacionados a sons já conhecidos. O sinal eletrônico produzido por sintetizadores pode ser controlado — "tocado" — por diferentes dispositivos. O mais comum é o teclado. Esse controle pode se dar via, por exemplo, MIDI. O sinal controlado, por sua vez, trafega para um amplificador, que o transmite para um alto-falante, que, por fim, produz o som de fato por meio de ondas sonoras no ar ao vibrar seu cone.

Pode-se dizer que a história dos sintetizadores começou com o Telharmonium, criado por volta do ano 1896 por Thaddeus Cahill (Apple, Inc., 2009). A técnica usada no Telharmonium para síntese sonora é a técnica de síntese aditiva. Essa técnica consiste na adição de ondas senoidais de diferentes frequências e amplitudes, resultando em uma nova onda, que se traduz em um novo timbre. Pelo seu imenso tamanho (o Telharmonium ocupava uma sala inteira), peso (a primeira versão pesava 7 toneladas, e as duas versões seguintes, 200 toneladas), preço (200.000 dólares na época — quase 5 milhões de dólares no ano atual) e outros fatores, o instrumento não teve sucesso comercial. Sua versão final foi projetada para ser tocada a 4 mãos. O gigantesco tamanho do instrumento se dava, em parte, pela necessidade de gerar grandes quantidades de energia, já que, na época, não haviam sido inventados amplificadores. Logo, o sinal precisava ser forte o suficiente para ser reproduzido pelos "auto-falantes" usados.

Em 1906, contudo, Lee de Forest inventou o primeiro amplificador de válvula (MILLARD, 1993), permitindo que sinais de menor potência fossem gerados por sintetizadores, já que poderiam ser amplificados. Dessa forma, a construção desses instrumentos foi simplificada em muitas vezes. A importância e influência dessa invenção se manifesta no órgão Hammond — inventado por Laurens Hammond e John M. Hanert e construído em 1935 (BUSH; KASSEL, 2006) —, que consiste, basicamente, em um versão mais compacta e acessível do Telharmonium, tornada passível de ser fabricada pelo uso de amplificadores. Após esse primeiro órgão Hammond, foi produzido um novo, nomeado Hammond Novachord (1939), que utilizava a técnica de síntese subtrativa. Diferentemente da síntese aditiva, que constitui em construir um sinal a partir de diferentes frequências, a síntese subtrativa parte de um sinal rico em tais frequências (harmônicas)

e filtra ele (COLLINS, 2008), atenuando a amplitude de certas frequências, gerando um sinal resultante de timbre diferente do original.

Com a disponibilização de transistores no mercado, a portabilidade e estabilidade dos instrumentos elétricos cresceu ainda mais. Começa, então, a surgir a série de sintetizadores desenvolvidos por Robert Moog. Os primeiros (1964) dessa série ainda eram de um tamanho considerável — certas vezes, lembrando mainframes — com diversos módulos que processavam o sinal sonoro sendo conectados por cabos, possibilitando uma alta customização do som. Contudo, seu uso não era prático, com os interessados em operá-lo encontrando dificuldade em sequer produzir algum som. Obras como *Switched-On Bach* (de Wendy Carlos, 1968), demonstrando o potencial desses sintetizadores, o revelaram ao público, mas os instrumentos da série Moog só começaram a consolidar-se de fato no mercado à medida que tornavam-se mais compactos, baratos e práticos (sacrificando, porém, potencial de customização). O Minimoog (1970) não fazia uso de cabos conectando seus módulos (Apple, Inc., 2009). Esses já vinham conectados internamente, e a configuração do processamento do sinal se dava, principalmente, por meio de *knobs*. Contudo, a interface ainda não era completamente satisfatória, visto que, para tornar a usar um timbre completamente diferente do antes utilizado, era necessário definir vários parâmetros, um a um, usando essas *knobs*.

Em 1973, a Yamaha patenteou o primeiro algoritmo de síntese digital (Yamaha Corporation, 2014), que utilizava a técnica de síntese FM, na qual um sinal tem sua frequência modulada por outro. A síntese FM digital, diferentemente da analógica, gozava de estabilidade tonal. Além disso, a síntese FM mostrou-se apropriada para a síntese de timbres percussivos, que dificilmente eram gerados fielmente por outras técnicas de síntese. Esse algoritmo patenteado foi utilizado em 1980 no Yamaha GS-1, mas a síntese FM digital só foi ser popularizada com o lançamento do DX7, também da Yamaha, em 1983, cujo som se tornou uma marca dos anos 80. Uma das razões para a popularidade desse instrumento foi a facilidade de uso: ao contrário do Minimoog, de anos antes, usando o DX7, era possível facilmente trocar entre timbres drasticamente diferentes, pois o sintetizador continha diversos sons pré-configurados à disposição do músico. Mesmo sendo possível construir (programar) novos timbres para uso no DX7, a maioria de seus usuários preferia a conveniência de usar sons “prontos”.

Com a disponibilidade de computadores pessoais no mercado, surgiram os sintetizadores de software. Esses, sem nenhum componente de hardware dedicado à síntese em si, gozam das possibilidades da programação em processadores de propósito geral.

A interação com tais sintetizadores também tira proveito da interação possibilitada pelo computador que o hospeda, como o uso de interfaces gráficas. Os modos de configuração dos seus parâmetros, logo, são altamente flexibilizados.

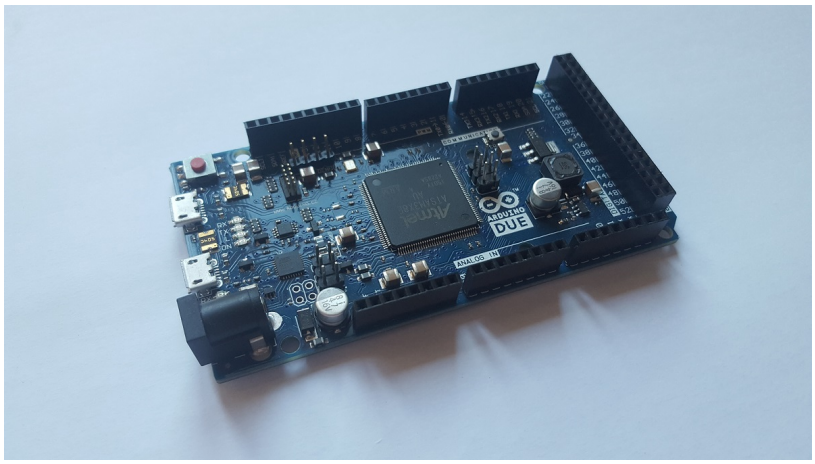
Seguindo os avanços no campo da engenharia de computação, microcontroladores se tornaram mais baratos e acessíveis. Com isso, projetos como o Arduino puderam surgir. O Arduino é um projeto de origem italiana de hardware aberto (Arduino AG, 2017). Seu principal produto é uma série de diferentes placas microcontroladoras que possuem diversos pinos de entrada e saída, possibilitando a conexão com outros circuitos e componentes elétricos, além de, geralmente, também terem uma porta USB, pela qual é possível conectar as placas a um computador e transmitir um pequeno programa a elas. A partir deste ponto, refere-se a uma placa microcontroladora da série de placas do projeto Arduino simplesmente como "Arduino".

Considerando que tais microcontroladores gozam de maior poder computacional do que muitos computadores usados na época da gênese dos sintetizadores em software, tem-se, agora, então, a possibilidade de programar um sintetizador em uma pequena e barata (e, portanto, acessível) peça de hardware. Um exemplo de hardware é o Arduino Due, que se mostra uma placa completamente apropriada para projetos de sintetizador de alta qualidade, com um processador ARM RISC 84MHz de 32 bits com aritmética de ponto fixo. O Arduino Due também possuem pinos de conversão digital para analógico, mas cujos resultados não são suficientes para projetos de alta qualidade, sendo, então, relevante apenas para projetos que não almejam, necessariamente, alta fidelidade sonora.

Diversos trabalhos implementam sintetizadores por meio do Arduino Due. (JOHANN, 2015) implementa um órgão semelhante ao Hammond, com síntese aditiva e polifonia completa de 5 oitavas. Recebe sinais MIDI e também possibilita a aplicação de efeitos como *delay* no sinal sintetizado. (PIROTTI; JOHANN; PIMENTA, 2015) descreve os passos inciais de um trabalho que pretende construir uma plataforma modular para síntese subtrativa. Módulos incluem o oscilador (com o formato de onda pré-definido), um gerador de envelope, um filtro, e, também, módulos para efeitos. (PIROTTI; JOHANN; PIMENTA, 2017; PIROTTI, 2017) constroem um sintetizador de síntese subtrativa que, por meio de comunicação com um conversor digital para analógico externo, consegue prover áudio de alta qualidade (16 bits a 48KHz). Além disso, também demonstrou a possibilidade de ter a opção de síntese aditiva na mesma plataforma ao mesmo tempo. (FOLLE, 2015) implementa um sintetizador FM inspirado no Yamaha DX7. Assim como o DX7, o sintetizador conta com 32 algoritmos pré-definidos de geração de som. Cada

um desses é, na verdade, uma combinação de 6 operadores, que são combinados tanto paralelamente quanto serialmente. Com a alteração de parâmetros de cada operador e seu gerador de envelope, é possível modificar o som gerado em tempo real. Todos esses trabalhos demonstram e legitimam a possibilidade de desenvolver sintetizadores de alta qualidade em plataformas de hardware baratas, acessíveis, e com recursos limitados, como o Arduino Due.

Figura 1.1: Arduino Due.



Fonte: O autor.

A placas microcontroladoras como o Arduino Due, então, é possível conectar componentes como *knobs*, *sliders*, e outros usados em sintetizadores de outrora para a interação com o algoritmo hospedado na placa. Contudo, além do custo e onerosidade da implementação de tal modelo de interface, outras formas de interação devem ser consideradas, pois ater-se a apenas uma delas significaria abdicar de várias outras possíveis formas de usar o sintetizador.

1.1 Objetivo

Esse trabalho tem como objetivo criar uma nova interface modular de código aberto para sintetizadores hospedados em Arduinos. A interface em si consiste em um aplicativo hospedado em um dispositivo Android. Android é o nome do sistema operacional móvel desenvolvido pela Google. Atualmente, é o sistema operacional móvel mais usado no mundo, com mais de 70% de participação no mercado (SUI, 2017). Aplicativos para smartphones que rodam Android são escritos, em sua maioria, em Java, além de usarem o SDK do Android. Esse trabalho é incluído nessa maioria. O uso de um smartphone

possibilita a interação pelo toque, já intuitiva quando se considerando a forma tradicional de interação com um sintetizador. O aplicativo consiste em uma área de trabalho (denominada *workspace*), para a qual podem ser arrastados componentes de interface, sendo montada uma área de trabalho de forma modular. Os componentes foram nomeados de *synth widgets*. Esses *synth widgets* espelham-se em componentes comumente usados para a configuração de sintetizadores, como *knobs*, *sliders*, botões, entre outros.

Contudo, *synth widgets* não precisam se limitar a imitar esses componentes, podendo ser criados *synth widgets* que atendem necessidades mais específicas ou *synth widgets* que funcionam de forma não convencional. Isso é possível tanto pela disponibilização do código publicamente, quanto pela modelagem do código, que faz uso de conceitos de orientação a objetos para que a criação de novos *synth widgets* (e extensão geral do trabalho) se dê da forma mais simples possível, sem obstruir as outras funcionalidades da interface. Dessa forma, é possível, por exemplo, que sejam criados *synth widgets* que criam métodos de interação inovadores, como um que recebe uma imagem (já disponível no celular utilizado ou obtida a partir da câmera) e, após interpretar os dados da imagem, os repassa para o sintetizador, que os utiliza de maneira inovadora. Ou, então, é possível ter um *synth widget* que faz uso de sensores disponíveis no celular, como acelerômetro, microfone ou giroscópio. Também é possível o desenvolvimento de um *synth widget* que, fazendo uso da tela de toque do celular, possibilite que o usuário desenhe, com a ponta de seu dedo, a forma de onda a ser utilizada pelo sintetizador. Inúmeras possibilidades existem.

A comunicação com o sintetizador hospedado no Arduino se dá através de Bluetooth. Bluetooth é um padrão de comunicações sem fio. Foi padronizado pela IEEE como IEEE 802.15.1. Contudo, não é mais mantido por essa organização, mas sim pela Bluetooth Special Interest Group (Grupo de Interesse Especial Bluetooth). Seus usos mais comuns são de curto alcance (menos de 10 metros), apesar de um alcance de até 100 metros ser possível (ou até 400 metros para o Bluetooth 5.0). Dado que a maioria dos smartphones usados hoje suportam tanto Bluetooth quanto Wi-Fi, a comunicação também poderia se dar por Wi-Fi. Contudo, Bluetooth foi escolhido, já que apresenta maior simplicidade nas conexões entre 2 dispositivos, além de um consumo menor de bateria. A velocidade maior oferecida pelo Wi-Fi não é necessária para o uso desse trabalho, assim como a comunicação através de média distância.

O público-alvo desse trabalho seriam os desenvolvedores dos sintetizadores, visto que a informação enviada da interface para o sintetizador contém apenas o nome do *synth*

widget e seu valor. Logo, é necessário um conhecimento do código do sintetizador de forma a integrar essa informação recebida com os módulos do sintetizador.

1.2 Estrutura do texto

No capítulo 2, São discutidos conceitos necessários para a compreensão do trabalho. Em particular, conceitos relacionados ao sistema operacional Android. No capítulo 3, são descritos trabalhos que se assemelham ou se relacionam a esse trabalho. No capítulo 4, é discutida a metodologia do trabalho, incluindo as ferramentas utilizadas e o processo de desenvolvimento. No capítulo 5, é discutida a Syntherface, sua estrutura e implementação. Por fim, no capítulo 6, é feita uma reflexão sobre o trabalho e são eleitas futuras extensões e implementações que enriqueceriam o mesmo.

1.3 Arquivos complementares

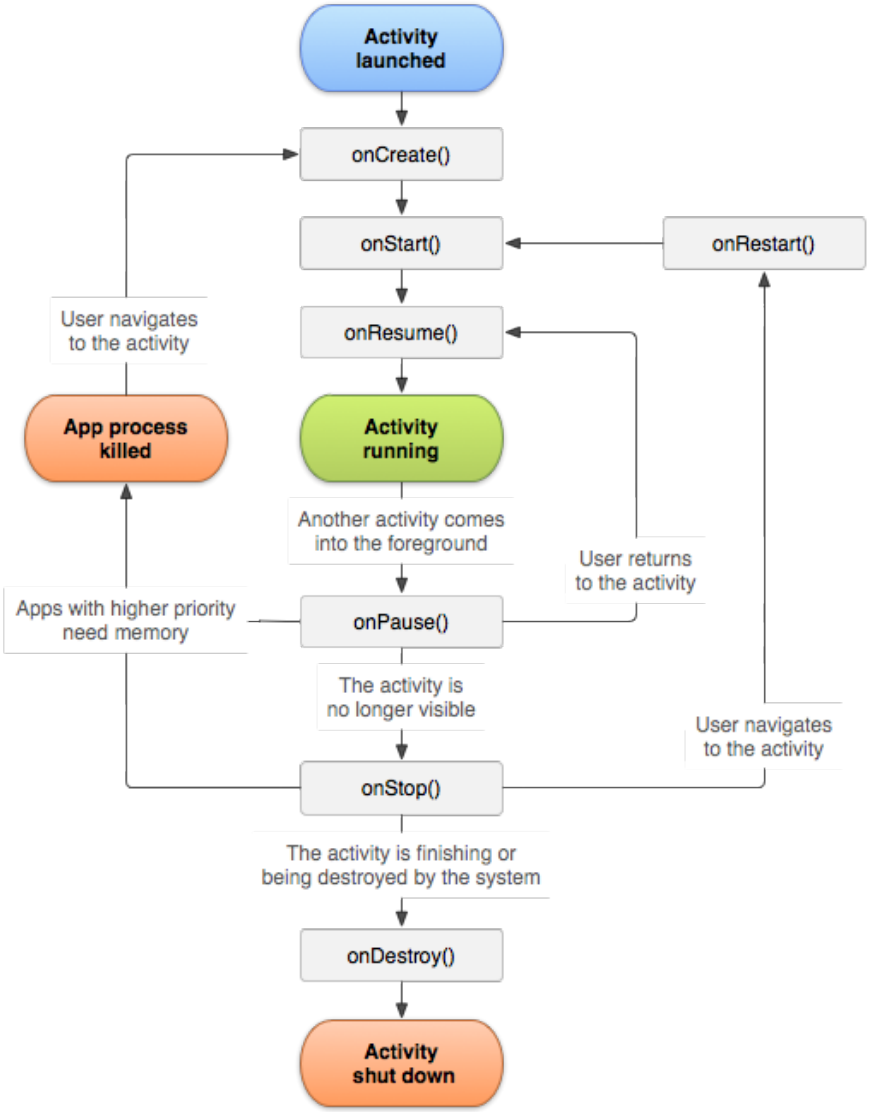
Os arquivos que complementam esse trabalho, que consistem no código-fonte e documentação do *Syntherface* podem ser encontrados em <https://github.com/Splitter4/syntherface>.

2 CONCEITOS RELACIONADOS

Neste capítulo, serão descritos conceitos Android necessários para a compreensão desse trabalho.

2.1 *Activity*

Figura 2.1: Fluxograma descrevendo o ciclo de vida de uma *activity* em inglês.



Fonte: Android Developers (Google LLC, 2017a)

Uma *activity* representa uma tela do aplicativo. No código, uma *activity* é implementada por uma classe que estende `Activity` (ou pela própria classe). Uma *activity*

é responsável por processar a interação do usuário com a interface, manipulando dados quando necessário e atualizando a interface com os resultados de seu processamento. Diferentes *activities* trabalham juntas para criar um fluxo de uso do aplicativo, mas uma dependência estrita em outra *activity* não é necessária (e nem recomendada). Por exemplo, em uma aplicação de mensagens, uma *activity* mostrando uma conversa pode receber informações de uma *activity* anterior de lista de conversas. Contudo, essas informações também podem vir de uma aplicação completamente diferente que invocou essa *activity* (caso isso seja permitido).

Uma *activity* tem um ciclo de vida específico. A cada evento desse ciclo, é chamado seu método correspondente. Na implementação da *activity*, é possível sobrepor esses métodos de forma a tomar as ações necessárias nesses momentos.

2.2 *Fragment*

Um *fragment* consiste, essencialmente, em um fragmento de uma tela. Um *fragment* é implementado por uma classe que estende `Fragment` (ou pela própria classe). *Fragments* são usados para compôr telas modularmente, já que cada *fragment* pode gerenciar seus componentes de interface (PHILLIPS; STEWART; MARSICANO, 2017c). Todo *fragment* é hospedado por alguma *activity*, e, como *activities*, os *fragments* capturam eventos de interação iniciados pelo usuário e respondem a eles de forma independente de outros *fragments* e *activities*. Por ser hospedado por uma *activity*, todo *fragment* tem seu ciclo de vida atrelado ao ciclo de sua *activity*, ou seja, cada momento na vida de um *fragment* corresponde a um momento apropriadamente relacionado da *activity*.

2.3 *Intent*

Um *intent* é uma mensagem enviada de um componente do aplicativo para outro (e.g. de uma *activity* para outra). Essa mensagem indica que foi requisitado que uma ação seja executada. Essa ação pode ser — e, na maioria dos casos, é — simplesmente que seja iniciada uma *activity*. É possível, também, transmitir dados através de *intents*, tanto na requisição de uma ação, quanto no resultado dela (também comunicado por uma *intent*). *Intents* são implementados pela classe `Intent`.

2.4 *Dialog*

Um *dialog* é um componente exibido por cima do resto da interface, comumente ocupando o espaço da tela apenas parcialmente. Um *dialog* é implementado por uma classe que estende `Dialog`. É utilizado para exibir uma informação, pedir por informações adicionais ou pedir por uma decisão do usuário. Em particular, *dialogs* são usados apenas se o fluxo de uso do aplicativo não deve seguir sem que o usuário veja ou interaja com o *dialog*. Caso o motivo de uso do *dialog* não seja prioritário ao ponto de dever interromper o fluxo de uso, um *dialog* não deve ser utilizado, sendo escolhido outro meio mais apropriado.

2.5 *View*

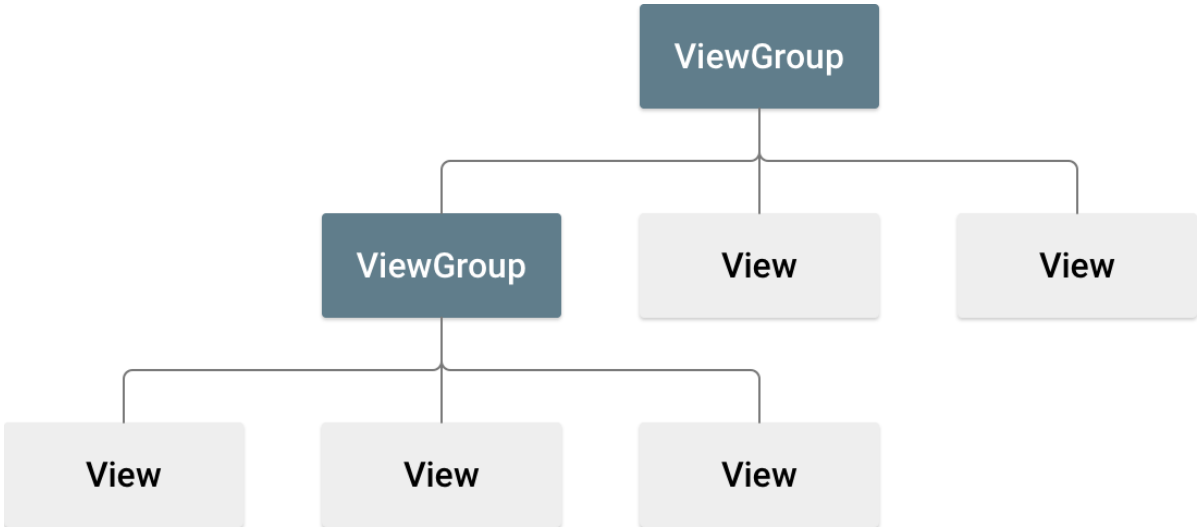
Uma *view* é, simplesmente, um elemento da interface visual de um aplicativo Android. Toda implementação de *view* estende `View` ou utiliza essa própria classe. Em geral, as *views* tem um componente visual, exibindo algo na tela. Além disso, comumente é possível interagir com elas. Exemplos de *views* incluem botões, *switches*, entradas de texto e listas. O *layout* de uma tela de um aplicativo Android é construído através de uma hierarquia de *views* utilizando *view groups* (implementados por `ViewGroup`, que estende `View`, ou uma estensão dessa classe), que são *views* invisíveis que contém outras *views*.

2.6 *Adapter*

Um *adapter* se encarrega de fazer a conexão entre uma coleção de dados e *views* específicas destinadas a exibi-los. Essas *views*, mais precisamente, são *views* que estendem `AdapterView`. Além disso, costumam ser *view groups*, contendo outras *views*. Um caso de uso de um *adapter* seria, por exemplo, uma lista, que contém vários elementos, cada um representado na tela por uma *view*. O *adapter* é responsável por obter os dados para cada elemento a partir da coleção de dados e, também, criar as *views* utilizadas para exibí-los na tela.

Adapters também otimizam a performance do aplicativo utilizando estratégias como apenas desenhar *views* que estão visíveis na tela e reutilizar *views* para exibir diver-

Figura 2.2: Ilustração de uma hierarquia de *views* definindo o *layout* da interface visual de um aplicativo Android.



Fonte: Android Developers (Google LLC, 2017c)

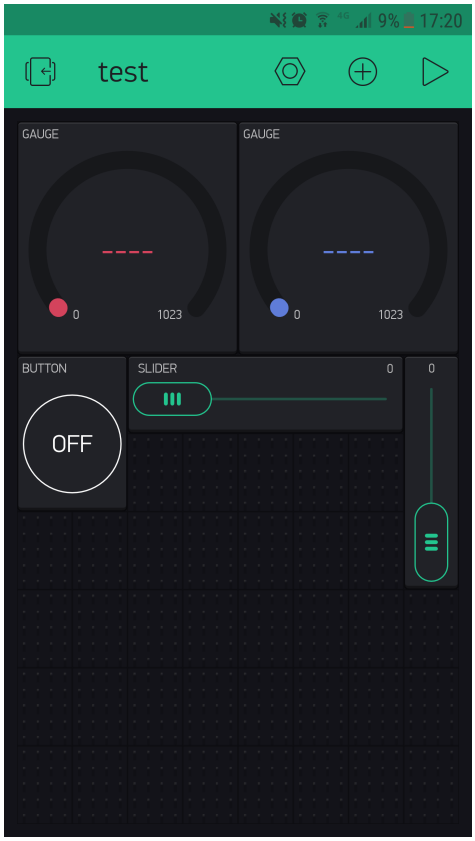
sos elementos, evitando a criação de uma quantidade muito grande de *views* para representar todos elementos de uma lista, e, assim, economizando memória e processamento (a criação de *views* pode ser um processo custoso em dispositivos móveis).

3 TRABALHOS RELACIONADOS

Diversas soluções existem para o problema da comunicação com um Arduino através de um dispositivo Android. Foram selecionadas, aqui, as soluções que mais se assemelham com a proposta do trabalho. Nenhuma delas, contudo, tem um foco em sintetizadores. Em vez disso, se propõem a atender projetos em geral feitos para micro-controladores e computadores de placa única.

3.1 Blynk

Figura 3.1: Interface de um projeto Blynk.



Fonte: O autor.

Blynk parece ser a solução mais completa. Propõe a se comunicar com não só placas Arduino, mas também Raspberry Pi, NodeMCU, Particle Photon, entre outros. A comunicação também pode se dar por vários meios: Bluetooth, Wi-Fi, Ethernet e USB são alguns exemplos. Ao criar um projeto no Blynk, define-se o dispositivo e o tipo de conexão. Então, um e-mail é enviado com um *token* de autenticação. Esse token

de autenticação é usado no código escrito para o dispositivo (seja ele uma placa Arduino, Raspberry Pi, etc). Além disso, também é disponibilizada uma biblioteca a ser usada nesse código, facilitando a configuração da comunicação e a comunicação em si. Dessa forma, concentra-se todo trabalho relativo à interface no projeto criado no Blynk. A interface em si é construída ao arrastar elementos (também chamados de *widgets*) como botões, *sliders*, e telas para a área de trabalho do aplicativo. Cada elemento pode ser configurado devidamente, conectando-o a um pino do dispositivo e definindo seus valores.

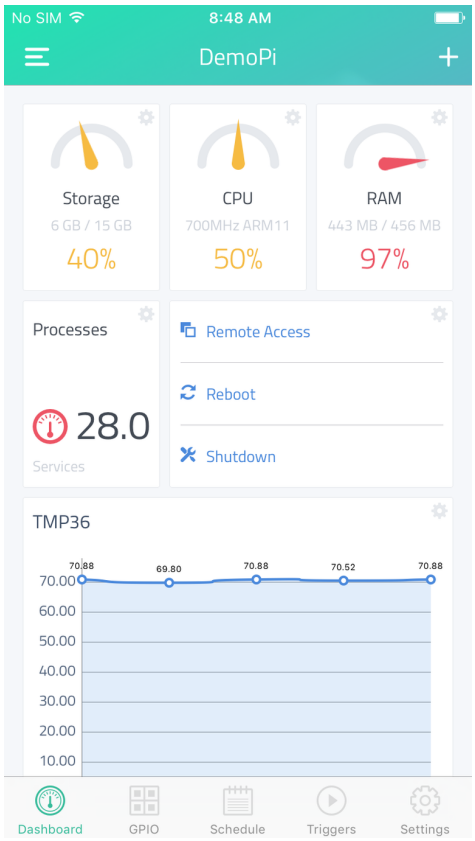
Apesar dos códigos-fonte da biblioteca (<https://github.com/blynkkk/blynk-library>) e do servidor utilizado para disponibilização do serviço da Blynk (<https://github.com/blynkkk/blynk-server>) serem abertos, os códigos dos aplicativos Android e iOS em si não são abertos. Assume-se que essa decisão se dá pela monetização da solução, que implementa um sistema econômico onde usuários possuem uma determinada quantia de "energia", que é consumida ao adicionar *widgets* ao projeto. A solução permite a compra de maiores quantidades de energia, possibilitando a construção de um projeto maior ou mais completo. No momento da confecção desse trabalho, os valores vão desde BRL 3,69 para 1.000 energia até BRL 69,99 para 28.000. Ao inciar-se um projeto, tem-se disponíveis 1.800 de energia para o posicionamento de *widgets*. Para referência, *widgets* mais baratos e comuns como botões, *sliders* e *displays* custam 200 de energia. Ao remover *widgets* do projeto, seu custo em energia é estornado para o usuário.

3.2 Cayenne

Cayenne é uma solução muito similar à Blynk. Porém, não suporta comunicação por Bluetooth, requerindo que seja conectado, na placa microcontroladora, um *shield* para conexão Ethernet ou Wi-Fi. Assim como Blynk, suporta o uso de diversas placas microcontroladoras, além de fornecer um *token* de autenticação e bibliotecas para auxiliar o desenvolvimento do código a ser utilizado na placa. Sua interface também consiste de diversos elementos disponibilizados pelo aplicativo posicionados por meio de arraste e solta, além de ser possível definir condições para ações a serem executadas, semelhante a expressões `if`/`then` em linguagens de programação.

Assim como Blynk, códigos de bibliotecas e outros trabalhos são disponibilizados (<https://github.com/myDevicesIoT/Cayenne-Agent>, mas não os códigos referentes aos aplicativos. Ao contrário de Blynk, contudo, Cayenne não limita o uso de *widgets* com base em uma moeda de uso interno do aplicativo.

Figura 3.2: Interface de um projeto Cayenne.



Fonte: O autor.

3.3 Outros trabalhos

Outras soluções, como *Gadget Controller - Arduino*, são semelhantes às listadas acima, seguindo a ideia de se conectar a diversos dispositivos, por diversos meios de comunicação, e organizar sua interface por meio de *widgets*, mas são mais simples. Outras, como *Arduino bluetooth controller*, limitam a interação com o dispositivo a comandos de texto digitados e enviados pelo usuário do aplicativo — uma solução, muitas vezes, simples o suficiente para uma prova de conceito ou um protótipo.

4 METODOLOGIA

Neste capítulo, serão descritas as ferramentas e tecnologias utilizadas no desenvolvimento do trabalho, além do processo de desenvolvimento em si.

4.1 Ferramentas e Tecnologias

Nesta seção, serão discutidas as principais ferramentas e tecnologias escolhidas para o desenvolvimento deste trabalho e a justificativa para tais escolhas.

4.1.1 Linguagem

A linguagem principal usada nesse trabalho foi Java — linguagem mais comumente usada no desenvolvimento de aplicativos para Android. Foi considerada a escolha de desenvolver o trabalho em Kotlin — uma linguagem desenvolvida mais recentemente (sua primeira aparição foi em 2011, e sua primeira versão estável foi publicada em 15 de fevereiro de 2016) pela JetBrains. Kotlin foi desenvolvida para ser executada pela máquina virtual Java ou para ser compilada para código em JavaScript. Recentemente, ganhou suporte oficial para Android. Contudo, a escolha desse trabalho tendeu a Java por ser, há mais tempo, uma linguagem estável e pela abundância de documentação e suporte que vem com seus anos de presença na esfera de desenvolvimento de software.

No lado do Arduino, a linguagem utilizada foi C++ por ser a linguagem principal utilizada em conjunção com o IDE oficial do Arduino. Contudo, devido à simplicidade do código necessário no dispositivo Arduino para se comunicar com esse trabalho, não existe uma limitação real do código a ser utilizado, visto que o algoritmo pode ser facilmente compreendido e traduzido para a linguagem de preferência do desenvolvedor do sintetizador.

4.1.2 Ambientes de desenvolvimento

Esse trabalho foi desenvolvido em 3 computadores diferentes. Seguem as 3 configurações diferentes:

Tabela 4.1: Ambientes utilizados no desenvolvimento desse trabalho. Todos sistemas operacionais usados são de 64 bits.

	Computador 1	**Computador 2**	**Computador 3**
Processador	Intel Core i7-4700MQ	Intel Core i5-2410M	Intel Core i7-7500U
Memória RAM	8 GB	6 GB	8 GB
Placa de vídeo	NVIDIA GeForce GTX 765M	NVIDIA GeForce GT 540M	Intel HD Graphics 620
Sistema operacional	Windows 10	Windows 7	Ubuntu 16.04

O IDE utilizado para o desenvolvimento do aplicativo Android foi o Android Studio. Como IDE oficial do desenvolvimento para Android, foi considerado a melhor opção. A outra opção considerada foi Eclipse, antigamente considerado, até mesmo por algum momento após o lançamento do Android Studio, a melhor opção. Mais de uma versão do Android Studio foi utilizada no desenvolvimento desse trabalho, à medida que o IDE requisitava novas atualizações. O desenvolvimento iniciou na versão 2.3.3, lançada em junho. Em seguida, foram utilizadas as versões 3.0.0, lançada em outubro, e 3.0.1, lançada em novembro. Todas as versões citadas foram lançadas no ano de 2017. O Android Studio também utiliza o Gradle para automação de compilação, e essa ferramenta também foi atualizada ao longo do projeto.

4.2 Estudo e desenvolvimento

O desenvolvimento do trabalho iniciou com a integração do módulo Bluetooth HC-05 com o Arduino Due. Para isso, estudou-se a forma adequada de conectar o módulo à placa. Após ser feita a conexão da forma adequada, testou-se a comunicação da placa com o módulo Bluetooth por meio de comandos AT enviados da placa para o módulo. Para isso, não foi utilizado o par de pinos seriais padrão do Arduino Due na conexão com o módulo. Dessa forma, foi possível utilizar o monitor serial da IDE do Arduino para enviar mensagens à placa. Essas mensagens, por sua vez, eram ecoadas no próprio monitor, além de serem transmitidas ao par de pinos seriais utilizados na conexão com o módulo Bluetooth, efetivamente possibilitando a comunicação com esse. Através de comandos AT definidos pelo fabricante do módulo, portanto, foi possível averiguar o sucesso da conexão do módulo com a placa, além da configuração do nome do módulo (usado para

identificá-lo no momento da conexão com a interface) e da taxa de transmissão (usada para a comunicação entre o módulo e a placa).

Dado que o módulo Bluetooth e seu uso com o Arduino Due foram devidamente compreendidos e implementados, o foco passou a ser a comunicação da interface com o sintetizador através desse módulo. Contudo, para isso, foi necessário, primeiro, estudar o desenvolvimento de um aplicativo Android. Para isso, as principais fontes foram (PHILLIPS; STEWART; MARSICANO, 2017a) e o próprio guia oficial de desenvolvimento Android criado pela Google. Inicialmente, conceitos básicos do *framework* e arquitetura Android foram estudados. Assim que compreensão suficiente foi obtida para iniciar o desenvolvimento de um aplicativo simples, foi feito o estudo da comunicação Bluetooth de um dispositivo Android com outros dispositivos. Esse período de estudos provou-se um dos mais extensos desse trabalho. Isso se deu pela complexidade do processo de estabelecer uma comunicação com outro dispositivo via Bluetooth utilizando os recursos disponibilizados pelo *framework* Android. Essa complexidade é evidenciada na seção seguinte, onde é descrita a implementação do aplicativo, incluindo o processo de comunicação via Bluetooth.

Paralelamente ao aprendizado da implementação da comunicação Bluetooth entre o dispositivo Android e outro dispositivo, foi desenvolvido um simples aplicativo cujo propósito consistia em aplicar os conhecimentos adquiridos e validá-los, levantando, também, dúvidas ao longo de seu uso e desenvolvimento. Essas dúvidas, por sua vez, moviam o aprendizado do desenvolvimento de um aplicativo Android. Dúvidas mais especificas sobre o uso correto da *framework* eram solucionadas pela referência da API Android, em vez das fontes supracitadas. Dessa forma, por exemplo, foi possível implementar o descobrimento de UUIDs do dispositivo ao qual deseja-se conectar, função que não havia sido mencionada nas fontes estudadas anteriormente. Esse simples aplicativo destinado ao processo de aprendizado da comunicação Bluetooth no lado Android consistia apenas nas funcionalidades mais básicas necessárias para testar o funcionamento correto da comunicação. Assim, era disponibilizado um botão para a conexão com o dispositivo (que, ao ser utilizado, era ocultado e, por sua vez, um botão para a desconexão era exivido), uma entrada de texto para definir o conteúdo a ser enviado, um botão para limpar essa entrada, outro para enviá-la, e, por fim, uma área dedicada a exibir os textos enviados e recebidos, assim como outras informações da comunicação. A estrutura e *layout* desse aplicativo, portanto, não tinham a intenção de se assemelhar com a estrutura e *layout* da Syntherface, limitando-se ao necessário para o aprendizado da implementação da comunicação

via Bluetooth partindo de um aplicativo Android.

No momento em que foi dada como compreendida a implementação da comunicação Bluetooth partindo de um dispositivo Android, o objetivo passou a ser a estruturação da Syntherface em si, mas, principalmente, da implementação da sua interface de arraste e solta. Para isso, também, foi consultado o guia de desenvolvimento Android para o suporte a eventos de arraste e solta. À medida que o desenvolvimento do suporte a esse tipo de interação progredia, a arquitetura do aplicativo também se tornava mais clara e bem-definida. Uma das maiores dificuldades desse processo foi a implementação de uma grade subjacente ao posicionamento de *synth widgets* na *workspace*. Em particular, pois acreditava-se que existiria um *view group* adequado e construído para o propósito de posicionar *views* em coordenadas específicas, que seriam distribuídas de forma regular seguindo a lógica de uma grade. Certos *view groups* do *framework* Android pareciam ser ideais para a implementação. Contudo, ao estudá-los mais profundamente e usá-los no desenvolvimento, tornou-se claro que não existia um *view group* orientado a uma grade com os requerimentos desse trabalho. Também foram pesquisadas soluções de terceiros, mas não foi encontrada uma solução ideal. Por fim, utlilizou-se o *view group* `RelativeLayout`, cujo método de posicionamento não é orientado a grades, mas sim à posição de *views* relativa a outros *views*. Foi possível posicionar as *views* dos *synth widgets* nas coordenadas desejadas por meio de ajustes aos tamanhos de suas margens.

Por fim, com tanto funcionalidades básicas da interface de arraste e solta quanto a comunicação Bluetooth com o sintetizador desenvolvidas, integrou-se ambos trabalhos. Nesse momento, também, foi desenvolvido um simples código a ser utilizado no Arduino Due em conjunção com o código do sintetizador. Esse código desenvolvido tem como objetivo receber as mensagens da interface Syntherface e interpretá-las, tomando as decisões necessárias no código do sintetizador por meio dos valores recebidos.

5 SYNTHERFACE

Neste capítulo, é discutida a arquitetura e implementação de Syntherface, a interface Android com sintetizadores no Arduino.

5.1 Arquitetura

O código do trabalho consiste em 7 classes (sendo uma delas um exemplo de estensão de `SynthWidget`).

1. `WorkspaceActivity`
2. `DeviceDialogFragment`
3. `SynthWidget`
4. `Knob` (extensão de `SynthWidget`)
5. `SynthWidgetAdapter`
6. `SynthWidgetViewHolderz`
7. `SynthWidgetDialogFragment`

A `WorkspaceActivity` consiste na *activity* principal do aplicativo. É nela que ocorre o gerenciamento da *workspace*, espaço onde são posicionados os *synth widgets*, além do gerenciamento das principais interações com o usuário. Além disso, é responsável por iniciar a conexão Bluetooth com o sintetizador. Para isso, utiliza a `DeviceDialogFragment`, encarregada de gerenciar o *dialog* que exibe dispositivos Bluetooth disponíveis para conexão, também fazendo a descoberta desses dispositivos.

`SynthWidget` é uma classe abstrata representando um *synth widget*. Ela contém diversas variáveis-membro e métodos essenciais para o funcionamento de todo *synth widget*. De particular importância são seus métodos abstratos, pois a implementação desses é necessária e suficiente para a extensão da classe para criar novos tipos de *synth widget*. Esses métodos abstratos, por exemplo, devem definir a aparência do *synth widget* através da definição do *layout* da sua *view*. Além disso, através desses métodos também é fornecido suporte à configuração do *synth widget* através do *dialog* exibido na sua criação.

A classe `Knob` exemplifica como deve ser feita uma extensão de `SynthWidget`. Ela define suas próprias variáveis-membro e alguns métodos próprios, além de implementar todos métodos abstratos de `SynthWidget`. É nessa extensão que deve ser gerenciada

a interação com a *view* do *synth widget* de forma a implementar o modo de interação desejado. Essa classe, então, usa métodos herdados de `SynthWidget` para enviar valores para o sintetizador.

`SynthWidgetAdapter` é utilizada em conjunção com `SynthWidgetViewHolder` para exibir, em uma lista, os diferentes *synth widgets* disponíveis para uso.

`SynthWidgetDialogFragment` gerencia o *dialog* exibido ao posicionar um *synth widget* na *workspace* pela primeira vez. É através do *dialog* gerenciado por ela que é configurado o *synth widget*, definindo seu nome e outras informações necessárias.

5.2 Implementação

Nesta seção, é discutida a implementação da comunicação por Bluetooth da interface com o sintetizador, além, também, da implementação da interface de arraste e solta.

5.2.1 Comunicação por Bluetooth

Nesta subseção, será detalhado o processo pelo qual se dá a comunicação entre o dispositivo Android e o sintetizador no Arduino usando o padrão Bluetooth.

Primeiro, é discutido o processo efetuado no lado Android para se comunicar com outro dispositivo por Bluetooth. Esse processo consiste em 3 passos:

1. Preparação da conexão
2. Descoberta de dispositivos
3. Conexão com o dispositivo

Após, é detalhado o simples processo para capturar mensagens no Arduino, em conjunção com os componentes utilizados e a configuração do hardware.

5.2.1.1 Preparação da conexão

Inicia-se esse fluxo dentro da *activity* principal do aplicativo Android, a `WorkspaceActivity`. Dentro da sobreposição do método `onCreate(Bundle)` da *activity* (que é chamado no momento em que a *activity* é criada), obtém-se uma ins-

tância de `BluetoothAdapter`, objeto que representa o rádio Bluetooth do dispositivo Android. Isso se dá através de um método estático de `BluetoothAdapter` que não recebe nenhum parâmetro. Caso não seja possível obter uma instância, isso significa que o dispositivo não suporta Bluetooth, e, por isso, não poderá usar o aplicativo. É improvável que essa situação ocorra, contudo, visto que o Google Play (centro de distribuição de aplicativos da Google) não permitiria a instalação do aplicativo para dispositivos que não suportam Bluetooth, já que o aplicativo declara a necessidade desse suporte.

Usando a instância não nula de `BluetoothAdapter` recém obtida, verifica-se se o Bluetooth está habilitado no dispositivo. Caso não esteja, inicia-se uma *intent* de forma a requisitar ao usuário que seja habilitado o Bluetooth. Inicia-se essa *intent* esperando por um resultado, que é capturado na sobreposição do método `onActivityResult(int, int, Intent)` da *activity*. Dentro desse método, é necessário verificar se o resultado capturado provém da requisição para habilitar o Bluetooth. No caso positivo, verifica-se se foi aceita a requisição, ou se foi negada. Caso tenha sido negada, exibe-se uma mensagem indicando que é necessário que o Bluetooth seja habilitado. Caso a requisição tenha sido aceita, faz-se o mesmo que é feito no caso em que o Bluetooth já estava habilitado: exibe-se o *dialog* de escolha de dispositivos para conexão.

5.2.1.2 Descoberta de dispositivos

`DeviceDialogFragment` estende `DialogFragment` e foi criado para exibir dispositivos Bluetooth para conexão.

`DialogFragment` é um *fragment* que hospeda um *dialog*. O uso de um `DialogFragment` é recomendado em vez de diretamente usar uma classe que implementa `Dialog` (PHILLIPS; STEWART; MARSICANO, 2017b), pois o `DialogFragment`, por ser um *fragment*, tem seu ciclo de vida atrelado ao da *activity* que o hospeda. Dessa forma, a criação e destruição do *dialog* é automaticamente gerenciada em vários casos, poupando escrita de código e garantindo um comportamento consistente do dialog frente aos eventos que ocorrem no aplicativo.

Anexação do *fragment*

O primeiro evento a acontecer no ciclo de vida do `DeviceDialogFragment` é a anexação desse *fragment* à sua *activity*, que ocorre no método `onAttach(Context)`. Esse método é sobreposto em `DeviceDialogFragment`. Nessa sobreposição, antes de tudo, é executada a superimplementação do método, garantindo que o *fragment* seja

corretamente anexado à *activity*, já que não é desejado interferir nesse processo.

O motivo da sobreposição se dá pois o momento em que a anexação ocorre é o momento mais cedo em que é possível ter uma referência à *activity* que hospeda o *fragment*. Portanto, é o momento mais apropriado para verificar que a *activity* hospedando o `DeviceDialogFragment` implementa a interface `DeviceDialogListener`, criada para alertar a *activity* que foi feita a escolha de um dispositivo Bluetooth para conexão (e qual é esse dispositivo). Assim, caso a *activity* não implemente essa interface, é lançada uma exceção indicando que é necessária a implementação, e o fluxo de vida do *fragment* é interrompido tanto quanto antes.

No caso em que a *activity*, de fato, implementa essa interface definida em `DeviceDialogFragment`, é armazenada uma referência à *activity* em uma variável destinada a conter uma instância de `DeviceDialogListener`. A conversão de tipos apropriada é feita na execução.

Criação do *dialog*

Após o atrelamento do *fragment* à *activity* (onAttach()), o *fragment* é criado (`onCreate()`). Após sua criação, o próximo evento do ciclo de vida de um `DialogFragment` envolve a criação do *dialog* em si, que ocorre em `onCreateDialog()`.

Esse método também é sobreposto em `DeviceDialogFragment` e, nele, obtém-se, a partir do adaptador Bluetooth do dispositivo (que é obtido da mesma maneira que anteriormente), a lista de dispositivos (`BluetoothDevices`) já pareados com o dispositivo Android. Essa lista, então, é utilizada na criação do *adapter* a ser utilizado para exibi-la. A implementação desse *adapter* consiste em uma implementação de `DeviceAdapter`. Por fim, retorna-se um `AlertDialog` (uma das implementações nativas de `Dialog`) para o qual foi definido o `DeviceAdapter` recém criado como seu *adapter*, assim como também foi definido o comportamento a ser executado no caso de um clique efetuado em um dos itens desse *adapter*. Esse comportamento é definido a partir de uma instância interna anônima de `DialogInterface.OnClickListener`, que implementa apenas o método `onClick(DialogInterface, int)`, dentro do qual é cancelada a descoberta de dispositivos próximos, e é chamado o método `onDeviceChosen(BluetoothDevice)` da instância de `DeviceDialogListener`, que é uma interface de `DeviceDialogFragment`. Essas duas ações se tornarão mais claras ao longo do texto.

`DeviceAdapter`, por sua vez, estende `ArrayAdapter<BluetoothDevice>`,

e possui uma implementação simples: um construtor que recebe uma lista de `BluetoothDevices` (e chama seu superconstrutor fornecendo essa mesma lista), e o método `getView(int, View, @NonNull ViewGroup)`. Esse último método apenas infla uma simples *view* para o dispositivo listado, caso essa já não exista, obtém o dispositivo Bluetooth em si, e, caso ele e seu nome não sejam nulos, popula sua *view* com seu nome. Caso o dispositivo Bluetooth ou seu nome sejam nulos, um texto padrão é inserido na *view*.

Iniciação do *fragment*

O iniciar do *fragment*, marcado pela chamada do método `onStart()`, é o primeiro momento em que o *fragment* está visível ao usuário. Logo, é o melhor momento para, caso desejado, exibir ainda outro *dialog*, pois, antes desse momento, esse novo *dialog* criado teria sua exibição obstruída pelo *dialog* principal, criado em `onCreateDialog()`.

Exibe-se um novo *dialog* por cima do principal (que lista dispositivos Bluetooth) pois, para versões 6.0 (de codinome Marshmellow) do Android e acima, é necessária a concessão de uma permissão adicional para exibirmos dispositivos Bluetooth não pareados na lista de dispositivos para conexão. O requerimento da permissão em si ocorre em ainda outro *dialog*. Esse, criado e exibido pelo sistema Android em si. Contudo, é necessário informar ao usuário o requerimento dessa permissão antes de o fazê-lo de fato. Caso contrário, ao não ser compreendida a necessidade dessa nova permissão, é possível que sua concessão seja negada, e, assim, uma importante funcionalidade do aplicativo não seria habilitada, prejudicando a experiência do usuário.

Logo, assim que o *dialog* de dispositivos Bluetooth é exibido, caso versão do Android do dispositivo rodando o aplicativo desse trabalho seja maior ou igual à versão 6.0, verifica-se se a permissão de acesso impreciso à localização do dispositivo Android já foi concedida ao aplicativo. Caso contrário, exibe-se o *dialog* alertando da necessidade dessa permissão para a descoberta de dispositivos Bluetooth não pareados com o dispositivo Android. Caso o usuário prossiga normalmente, a permissão necessária é requisitada. O resultado dessa requisição é capturado na sobreposição do método `onRequestPermissionsResult(int, @NonNull String[], @NonNull int[])`. Dentro desse método, primeiro, identifica-se se o resultado sendo informado pertence à requisição recém feita, comparando um identificador definido em `DeviceDialogFragment` (usado no momento da requisição) e o identificador informado pelo método. Sendo ambos identificadores iguais, caso a permis-

são tenha sido concedida, inicia-se o processo de descoberta de dispositivos Bluetooth. Esse processo também é iniciado no caso em que a permissão já tenha sido concedida, além de no caso em que a permissão não é necessária para a versão do Android utilizada no dispositivo rodando o aplicativo. Nos casos em que a permissão não é concedida ou que é cancelado o *dialog* que explica a necessidade da permissão, é exibido um texto alertando que serão exibidos apenas dispositivos já pareados com o dispositivo Android no *dialog* de dispositivos para conexão.

A descoberta dos dispositivos não pareados em si se dá da seguinte forma: primeiro, é criado um *intent filter* com as ações que indicam que um dispositivo Bluetooth foi encontrado e que a descoberta de dispositivos foi encerrada. Esse filtro, então, é informado no momento do registro de um `BroadcastReceiver`. Finalmente, é iniciada a descoberta de dispositivos. A descoberta é feita pelo sistema de forma assíncrona, e os dados e informações relevantes são obtidos por meio de *intents*, que, por sua vez, são capturados pelo `BroadcastReceiver` recém registrado. O `BroadcastReceiver` registrado em `DeviceDialogFragment` contém, em sua implementação apenas a sobreposição do método `onReceive(Context, Intent)`, por meio do qual recebe os *intents* que contém ações adicionadas ao seu *intent filter*. Assim, para cada execução de `onReceive(Context, Intent)`, é extraída a ação do *intent*. Verifica-se, então, qual a ação. No caso de uma ação correspondente à descoberta de um dispositivo, extrai-se o dispositivo do *intent* e adiciona-se ele ao `DeviceAdapter` usado pelo *dialog* de dispositivos, para que seja exibido para o usuário na lista de dispositivos. No caso da ação correspondente ao término da descoberta, simplesmente cancela-se a mesma.

5.2.1.3 Conexão com o dispositivo

O fluxo da conexão Bluetooth com um dispositivo continua na `WorkspaceActivity`, onde é implementado o método `onDeviceChosen(BluetoothDevice)` da interface `DeviceDialogListener` definida em `DeviceDialogFragment`. Ao ser feita, pelo usuário, a escolha de um dispositivo para conexão, então, é chamado o método `onDeviceChosen(BluetoothDevice)`. Nesse método, o dispositivo é armazenado em uma variável membro da *activity*. Em seguida, é criado um *intent filter* com a ação `BluetoothDevice.ACTION_UUID`, que é usado no registro de um `BroadcastReceiver`. Em seguida, inicia-se a captura, por meio do SDP do Bluetooth, dos UUID do dispositivo escolhido. Esse UUID identifica o serviço provido

pelo dispositivo Bluetooth, e é utilizado para a conexão. A captura desse UUID, assim como a descoberta de dispositivos, ocorre assincronamente. A sobreposição do método `onReceive(Context, Intent)` do `BroadcastReceiver` registrado garante a captura dos dados desejados. Na sobreposição desse método no `BroadcastReceiver` que foi registrado para obter o UUID do dispositivo Bluetooth, primeiro, verifica-se que a ação extraída do *intent* equivale à ação que indica a obtenção de UUID. Então, esse UUID é extraído do *intent*, armazenado em uma variável membro da *activity*, e é iniciada a conexão com o dispositivo.

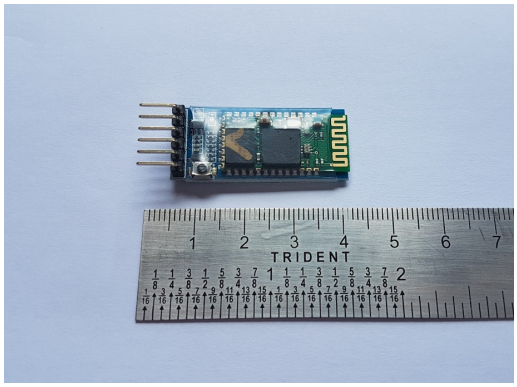
O primeiro passo do procedimento de conexão é desfazer o registro do `BroadcastReceiver` encarregado de receber o UUID, caso tenha sido previamente registrado. Após, seguindo boas práticas de código, é cancelada a descoberta de dispositivos. A maneira com a qual o fluxo de escolha e conexão com um dispositivo Bluetooth foi implementada até aqui garante que a descoberta de dispositivos não estará ocorrendo. Contudo, por ser conhecido que a descoberta de dispositivos interfere negativamente com a conexão com um dispositivo Bluetooth, aumentando as chances de uma falha na conexão, é recomendado, pela referência do Android quanto ao método que inicia a conexão, que a descoberta de dispositivos sempre seja cancelada antes de ser feita uma conexão com um dispositivo (Google LLC, 2017b). Em seguida, então, é criado um *socket* com o dispositivo escolhido e armazenado usando o UUID previamente coletado. Esse procedimento é síncrono, e, em caso de falha, é exibido, novamente, o `DeviceDialogFragment`, de forma que o usuário possa escolher outro dispositivo adequado ou tentar novamente. Caso a criação do *socket* tenha ocorrido com sucesso, em uma nova *thread* (de forma a não interromper a *thread* da interface com o usuário), é efetuada a conexão usando o *socket*. Logo após, é obtido o *stream* de saída, utilizado para o envio de dados para o dispositivo Bluetooth. Aqui, também, em caso de falha, após serem fechados o *socket* e o *stream* de saída, é exibido, novamente, o `DeviceDialogFragment`.

5.2.1.4 Recebimento no Arduino

Para o recebimento do sinal eletromagnético contendo as mensagens dessa comunicação, foi utilizado o módulo HC-05 B28090W de interface serial Bluetooth. O HC-05 é uma opção barata e de uso prático. Tem um alcance de até 10 metros, suficiente para o desenvolvimento e uso desse trabalho. É um módulo amplamente utilizado em conjunção com placas Arduino para diversos projetos que envolvem comunicação por Bluetooth. O

HC-05, como os outros módulos seriais da série HC terminando em número ímpar, pode ser configurado (por meio de comandos AT) para executar tanto as funções de *master* quanto *slave*. Já os terminando em número par vem com uma das funções definidade fábrica, e seu modo de execução não pode ser alterado posteriormente. Toda comunicação Bluetooth deve se dar entre um *master* e um *slave*, mas esses só diferem no fato que quem gerencia a conexão é o dispositivo *master*, além de que ele pode estar conectado a mais *slaves*, enquanto *slaves* podem estar conectados apenas a um dispositivo (o *master*). Na comunicação desse trabalho, o *master* é o adaptador Bluetooth do dispositivo Android.

Figura 5.1: Módulo HC-05 de interface serial Bluetooth utilizado no desenvolvimento desse trabalho.



Fonte: O autor.

A placa Arduino utilizada nesse trabalho foi uma do modelo Due R3. A conexão do HC-05 com o Arudino Due se dá de forma simples. Os pinos terra (GND) e de alimentação (5V) são conectados aos respectivos pinos do Arduino. O pino de transmissão (TX) é, simplesmente, conectado a um dos pinos seriais de recebimento (RX) do Arduino Due (que possui 5 pares de pinos destinados a comunicação, enquanto o Arduino Uno, modelo mais popular, por exemplo, possui apenas um).

Pela abundância de pinos de transmissão no Arduino Due, é possível não utilizar os pinos 0 e 1, que, quando não utilizados, possibilitam a comunicação serial com o computador à qual a placa está conectada pelo cabo USB. Dessa forma, caso necessário, é possível enviar comandos AT para o HC-05 para configuração, por exemplo, do nome do módulo Bluetooth e da sua taxa de transmissão.

No código do sintetizador, para receber as mensagens via Bluetooth enviadas pelo dispositivo Android, simples passos devem ser tomados. Primeiro, na função `setup()`, chamada antes da execução de *loop()* (a função principal, chamada continuamente após finalizado o *setup*), inicializa-se a comunicação serial do par de pinos utilizados definindo

a taxa de transmissão informada em baud.

As mensagens enviadas para o sintetizador são compostas no formato `{NOME_DO_SYNTH_WIDGET}={valor}`. Um exemplo consistiria em `VOLUME=0.5`. Recebe-se, então, essa mensagem dentro da função `serialEvent()` correspondente ao par de pinos seriais utilizados (`serialEvent()` corresponde à comunicação referente aos pinos 0 e 1). Essa função é chamada quando existem dados disponíveis para a leitura. Nessa função, então, enquanto houver bytes disponíveis para leitura, lê-se um caractere da comunicação serial estabelecida. Então, verifica-se, consultado uma variável booleana, se está sendo feita a leitura do nome do *synth widget*. No caso positivo, verifica-se se o caractere recém lido equivale ao caractere =. No caso positivo, é atribuído o valor `false` para a variável que determina se está sendo feita a leitura do nome do *synth widget*. O caractere = é simplesmente ignorado. Caso o caractere recém lido não seja esse, concatena-se o caractere ao nome do *synth widget*. No caso em que não está sendo feita a leitura do nome do *synth widget*, consideram-se os caracteres lidos como referentes ao valor sendo enviado. Verifica-se, então, se o caractere recém lido equivale ao caractere que indica uma nova linha. No caso positivo, é atribuído o valor `true` à variável booleana que indica que está sendo feita a leitura do nome do *synth widget* e também à variável que indica o fim da leitura. Além disso, o caractere recém lido é ignorado. Caso o caractere recém lido não represente uma nova linha, ele é concatenado ao texto representativo do valor enviado.

Ao fim desse ciclo de leituras de caracteres disponíveis, ainda dentro de `serialEvent()`, verifica-se se foi terminada a leitura verificando o valor da variável booleana referente ao final da leitura. No caso negativo, nada é feito, pois espera-se que `serialEvent()` seja chamada novamente assim que os caracteres restantes estiverem disponíveis para a leitura. Caso a leitura do nome e valor enviados tenha, de fato, terminado, verifica-se, então, qual a variável referente ao nome de *synth widget* recebido. A essa variável, então, é atribuído o valor recebido, após ter sido convertido para um tipo compatível com o da variável. Por fim, as variáveis tanto do nome quanto do valor são esvaziadas para a correta concatenação de caracteres futuros. Quanto à variável recém atualizada com o valor recebido, quando for utilizada, será utilizada, então, com o valor atribuído pela interface Android, completando o fluxo que inicia desde a interação do usuário com o aplicativo até o uso dessa variável no código do sintetizador.

5.2.2 Interface de arraste e solta

A ideia principal da interface desse trabalho é orientada a arrastar e soltar componentes utilizados para a interação com o sintetizador (dispositivo Bluetooth ao qual foi feita a conexão). Foi atribuído, a esses componentes, o nome de *synth widget*. O espaço no qual os componentes são posicionados foi nomeado de *workspace*. Todo o gerenciamento da *workspace* (incluindo o arraste e solta de *synth widgets*) ocorre na `WorkspaceActivity`. O layout dessa *activity* consiste, simplesmente, do *view group* da *workspace* em si, e, ancorada à parte inferior da tela, a *view group* da lista de *synth widgets* disponíveis para utilização. Essas duas *views* são configuradas na sobreposição de `onCreate(Bundle)` da `WorkspaceActivity`.

Primeiro, para a *workspace*, é atribuída uma implementação de `View.OnDragListener`, que escuta aos eventos emitidos pelo sistema relacionados ao arraste e solta de componentes. Após, atribui-se à *workspace* uma classe interna anônima encarregada de executar um código no momento em que foi feito o layout das *views* da *activity* pelo sistema. Só após ter ocorrido esse processo é possível obter as dimensões da *workspace*, já que essa não tem dimensões fixas definidas. A largura da *workspace* foi definida de forma a ocupar toda largura disponível da tela (que varia de dispositivo para dispositivo), e sua altura foi definida de forma a ocupar toda altura disponível dado que a lista de *synth widgets* foi definida com uma altura fixa. As dimensões da *workspace* são utilizadas para definir o número de linhas e colunas que compõem a grade por trás do posicionamento de *synth widgets* na *workspace*. Consequentemente, também é definido o tamanho de uma célula dessa grade. Todas as células são quadradas, e seu tamanho provém de um valor aproximado definido previamente, sendo seu valor final dinâmico, de forma a se adaptar a diferentes tamanhos de tela. Tanto o número de colunas quanto o de linhas e o tamanho final de uma célula são armazenados através de variáveis membro da `WorkspaceActivity`. Em seguida, é criada uma lista de `SynthWidgets` — uma classe abstrata representando *synth widgets* —, e são adicionadas a ela implementações de `SynthWidget` que representam os componentes disponíveis para composição da interface e interação com o sintetizador. Essa lista, então, é utilizada para a construção do `SynthWidgetAdapter` atribuído à `RecyclerView` utilizada para exibir a lista de *synth widgets*.

5.2.2.1 `SynthWidgetAdapter`

O papel desse *adapter* se resume a, quando necessário, criar as *views* para os itens da lista e ligar essas *views* com os itens os quais elas representam. No momento em que ocorre a ligação de uma *view* com um item, contudo, ocorre uma ação crucial para a interface desse trabalho. É atribuído à *view* um código ao ser executado quando um toque longo é feito nela. Esse código é definido por uma implementação interna anônima de `View.OnLongClickListener`, que sobrepõe apenas o método `onLongClick(View)`. Nele, primeiro, é construída uma sombra a partir da *view*. É, então, extraído o *synth widget* que está sendo representado por essa *view*. A *view* utilizada como sombra é armazenada nesse *synth widget*. Por fim, é iniciado o processo de arraste e solta desse *synth widget*, passando o próprio como argumento, de forma a disponibilizá-lo para as entidades que tratarem os eventos seguintes de arraste e solta.

5.2.2.2 `WorkspaceDragListener`

O `WorkspaceDragListener` é a implementação de `View.OnDragListener` atribuída, anteriormente, à *workspace*. Sua definição reside dentro de `WorkspaceActivity`, tendo acesso aos métodos definidos dentro dessa. A implementação de `WorkspaceDragListener` consiste apenas no método `onDrag(View, DragEvent)`, que é chamado sempre que um `DragEvent` é emitido pelo sistema (dado que `WorkspaceDragListener` esteja registrado para recebê-lo).

Assim, ao início desse método, são feitas, antes de tudo, duas atribuições: uma define a *view* recebida como a *workspace*; a outra extrai o *synth widget* sendo arrastado a partir do evento. Então, é extraída a ação do evento, e, para cada ação (que são definidas em `DragEvent`), é executado um caso diferente.

`ACTION_DRAG_STARTED`

Essa ação indica que foi iniciado o arraste. Para todas as ações (com exceção de um caso, que será comentado abaixo), é retornado `true`, de forma a indicar que o `WorkspaceDragListener` deseja continuar recebendo os eventos seguintes. Para esta ação, isso é tudo que é feito.

`ACTION_DRAG_ENTERED`

Essa ação indica que o *synth widget* sendo arrastado entrou dentro da área da *workspace* ao qual o `WorkspaceDragListener` foi atribuído. Aqui, a partir do *synth*

widget previamente extraído do evento, infla-se sua *view* informando a *workspace* para a qual a *view* está sendo inflada, além do tamanho da célula da grade. Essa *view*, então, é extraída do *synth widget* e adicionada à *workspace*.

`ACTION_DRAG_LOCATION`

Essa ação indica que o *synth widget* sendo arrastado foi movido dentro da área da *workspace*. Usando as coordenadas da posição do *synth widget* (que são extraídas do evento), são calculadas, a partir do tamanho de uma célula, a linha e a coluna em que o *synth widget* deve ser posicionado. O *synth widget*, então, é posicionado nessas coordenadas através da definição de novos valores para suas margens superior e esquerda.

Após, utilizando as coordenadas recém calculadas e, também, as informações de quantas linhas e colunas o *synth widget* ocupa, é verificado se é possível posicionar o *synth widget* nessa posição permanentemente, ou se alguma das células que ele ocuparia está já ocupada (efetivamente o colidindo com outro *synth widget*). Essa verificação é feita utilizando uma variável membro constituída de uma lista bidimensional de booleanos indicando se a célula na linha e coluna correspondentes aos índices da lista está ocupada. Após ser obtido esse resultado, também é feita uma tentativa de obter uma *view* cujo propósito é o de exibir um tom avermelhado por cima do *synth widget* para alertar sobre a impossibilidade de posicionar o componente nas coordenadas atuais.

São, então, tratados dois casos: caso não seja possível posicionar o *synth widget* na posição desejada e não exista a *view* de tom avermelhado, então essa *view* é inflada e sobreposta à *view* do *synth widget*. Caso seja possível posicionar o *synth widget* na posição atual, mas exista a *view* de tom avermelhado sobreposta à *view* do *synth widget*, então essa *view* avermelhada é removida.

`ACTION_DRAG_EXITED`

Essa ação indica o caso em que o *synth widget* deixou de estar dentro da área da *workspace*. Portanto, nesse caso, remove-se a *view* do *synth widget* da *workspace*.

`ACTION_DROP`

Essa ação indica que foi efetuado a solta do *synth widget* sobre a *workspace*. Como para a ação que indica o movimento do *synth widget* dentro da *workspace*, primeiro, são obtidas a linha e coluna onde deve ser posicionado o *synth widget*, assim como as quantidades de linhas e colunas ocupadas por ele. Então, é verificado se o *synth widget* pode ser posicionado nessas coordenadas, da mesma forma que anteriormente. No caso negativo, remove-se, da *workspace*, a *view* do *synth widget* e retorna-se false de modo a indicar que a solta não foi permitida. No caso em que o *synth wid-*

get possa ser posicionado nas coordenadas em que foi solto, primeiro, é alterada a informação de ocupação de células usada, justamente, para a consulta sobre o posicionamento de *synth widgets* na *workspace*. Dessa forma, registra-se o espaço como oficialmente ocupado, impossibilitando a colisão com *synth widgets* futuros. Após, é criado um `SynthWidgetDialogFragment`. A esse, é informado o *synth widget* de interesse atual, além de ser atribuída uma implementação interna anônima de OnCancelListener (que é definido em `SynthWidgetDialogFragment`) constituída apenas pela sobreposição do método `onCancel()`. Nessa sobreposição, é removido o *synth widget* da *workspace*, por meio tanto da remoção da sua *view* quanto da liberação das células que ocupava. Feito esse preparo, é exibido esse *dialog*, e retornado `true` para indicar que a solta foi processada devidamente.

6 CONCLUSÃO E TRABALHOS FUTUROS

6.1 Conclusão

Nesse trabalho, foi apresentada uma interface Android para sintetizadores no Arduino chamada *Syntherface*. Essa interface consiste em um aplicativo que se comunica com o sintetizador por meio de Bluetooth. É necessário que a placa Arduino que hospeda o sintetizador tenha um módulo Bluetooth conectado a ela para o recebimento das mensagens vindas da interface. Componentes virtuais dessa interface, chamados de *synth widgets*, são organizados na tela através de um sistema de arraste e solta.

O objetivo de *Syntherface* é facilitar a interação com tais sintetizadores ao dispensar a necessidade de conectar componentes físicos ao Arduino para a configuração e alteração de parâmetros do sintetizador, além de potencializar a interação com esse. 4 fatores contribuem para isso:

1. A interface com o sintetizador é virtual;
2. O código da interface é aberto e encontra-se disponível na Internet;
3. A modelagem do código facilita a criação de novos *synth widgets* e extensão do trabalho;
4. O código adicionado ao sintetizador para receber e interpretar as mensagens vindas da interface é pequeno e simples.

Pela interface ser virtual, os componentes criados para ela não precisam se ater às restrições e funções de componentes físicos tradicionais. Em vez disso, existe um potencial para a criação de novos e inovadores componentes, que não necessariamente se espelham em componentes já conhecidos.

O fato do código da interface ser aberto e estar disponível em uma plataforma conhecida de códigos de projetos é um diferencial frente a trabalhos semelhantes a esse, além de convidar desenvolvedores interessados a estenderem esse trabalho, principalmente através da criação de novos componentes virtuais de interface, mas não sendo a extensão limitada a ela.

A simples disponibilização do código, contudo, não é suficiente para acolher contribuições externas. Por nisso, na escrita de todas as partes do código, foram feitas escolhas de modelagem que fizeram uso de conceitos de orientação a objetos. Um foco foi dado para a criação de novos *synth widgets*, sendo necessário, para isso, apenas estender

a classe abstrata `SynthWidget`, implementando seus métodos abstratos. Dessa forma, é minimizada a quantidade de conhecimento necessária para contribuir com o trabalho.

Além disso, para a utilização da interface, também foi minimizado o código necessário a ser adicionado ao sintetizador. Assim, também é minimizada a chance do código referente à comunicação com a interface interferir no código referente ao funcionamento geral do sintetizador. Dessa forma, também evita-se que seja necessária uma reescrita do código do sintetizador, caso a modelagem desse não oferecesse suporte à forma com a qual a comunicação com a interface interage com o código. Em vez disso, espera-se que seja necessário apenas tornar globais as variáveis de interesse que teriam seu valor alterado pela interface.

6.2 Trabalhos futuros

De muitas formas esse trabalho ainda pode ser estendido. Algumas delas serão discutidas a seguir.

6.2.1 Funcionalidades básicas

Ao iniciar o desenvolvimento desse trabalho, o autor não possuía nenhum conhecimento prévio quanto a desenvolvimento móvel. Isso significa que, ao longo do desenvolvimento desse trabalho, conhecimento sobre a arquitetura Android, o *framework* Android, e desenvolvimento móvel em geral foi adquirido, começando pelo básico, e, então, estudando tópicos específicos *ad hoc*. Assim, à medida que maturidade de desenvolvimento era aqduirida, o tempo disponível para desenvolvimento tornava-se escasso. Portanto, diversas funcionalidades consideradas básicas pelo autor não puderam ser desenvolvidas a tempo, mesmo que, à altura da entrega do trabalho, o desenvolvimento dessas não fosse de tanta dificuldade quanto quando comparado ao início do desenvolvimento do trabalho.

Uma dessas funcionalidades é o reposicionamento de *synth widgets*, visto que, no atual estágio de desenvolvimento, ao posicionar um *synth widget* na *workspace*, não é possível reposicioná-lo. Para dar suporte a essa funcionalidade, simplesmente deve-se fazer o mesmo feito para arrastar *synth widgets* da lista para a *workspace*. Contudo, usando a presença ou ausência do nome do *synth widget*, deve determinar-se se esse já se encontra posicionado na *workspace*. Tendo esse conhecimento, para *synth widgets* já

posicionados, não é necessário inflar sua *view*. Em vez disso, usa-se a *view* já existente. Ao não ser efetuado o reposicionamento previamente iniciado, contudo, deve-se retornar a *view* para a posição do *synth widget* antes do início do reposicionamento.

Outras funcionalidades básicas são a deleção e edição de *synth widgets*. De acordo com a intenção original, ao iniciar o arraste de um *synth widget* para a *workspace*, a lista de *synth widgets* seria recolhida, e, no seu lugar, seriam exibidas as áreas onde, ao soltar o *synth widget*, o mesmo seria excluído ou editado. Para a edição, seria exibido o mesmo *dialog* exibido no momento da criação. Porém, os valores do *synth widget* estariam pre-enchidos nos respectivos campos. Dessa forma, poderia ser alterada a configuração de um *synth widget* de forma prática, sem muito mais desenvolvimento, visto que o *dialog* de criação seria reutilizado. Ao soltar um *synth widget*, as áreas de exlusão e edição de *synth widgets* são recolhidas, e a lista de *synth widgets* é exibida novamente.

6.2.2 Rolagem da *workspace*

Atualmente, não é possível rolar a área de trabalho da interface, onde são posicionados os *synth widgets*, ainda que a intenção original do desenvolvimento do aplicativo incluísse disponibilizar essa funcionalidade. Logo, o espaço disponível para o posicionamento de *synth widgets* ainda é limitado — e, por consequência, a quantidade de *synth widgets* também — o que restringe muitas das vantagens da virtualidade da interface.

Para tornar a *workspace* rolável, é preciso encapsulá-la em um *view group* que suporte rolagem. Então, a *workspace* pode ter uma altura maior que a da tela do dispositivo Android utilizando essa interface. Uma *view*, contudo, não pode ter uma altura indefinida. A intenção, portanto, é que a altura da *workspace* fosse sempre incrementada ao ser posicionado um *synth widget* a uma certa distância de seu fim. Esse incremento garantiria que sempre houvesse uma distância mínima entre o *synth widget* mais baixo da *workspace* e o fim dessa. Essa distância mínima deve ser suficiente para garantir que sempre seja possível efetuar a rolagem da *workspace* enquanto um *synth widget* esteja visível na tela. Dessa forma, comunica-se efetivamente para o usuário que a *workspace* tem uma altura virtualmente indefinida, e que, portanto, podem ser posicionados tantos *synth widgets* quanto desejado.

6.2.3 Rotação da tela

Diferentes *synth widgets* possuem diferentes dimensões, e diferentes usuários possuem diferentes formas de posicionar esses *synth widgets*. Assim, a rotação da tela se faz uma funcionalidade importante para a usabilidade do aplicativo.

O suporte para rotação da tela, contudo, não se dá de forma trivial. Uma das modificações a serem feitas seria a construção de um novo *layout* para a nova orientação da tela, visto que, por exemplo, a lista de *synth widgets* deve mudar de posição, deixando de ficar na parte de baixo da tela, para ficar em uma das laterais dessa. Além disso, essa lista deixaria de ser uma lista horizontal, passando a ser uma lista vertical.

Outra modificação seria a persistência dos *synth widgets* e suas posições. Essa persistência é necessária para a rotação da tela pois, ao rotacionar uma *activity*, ela é destruída e criada novamente. Caso contrário, ao rotacionar a tela, toda configuração de *synth widgets* criada pelo usuário seria perdida. O *framework* Android provém de funcionalidades para salvar e recuperar o estado de uma *activity* através de `Bundle`s, uma classe usada para a transferência de dados entre *activities* e processos, inclusive sendo informado esse `Bundle` no momento da criação da *activity*. Os *synth widgets* e seus dados deveriam, então, ser armazenados em um `Bundle`, sendo feito o processamento necessário para esse armazenamento.

O último desafio para a implementação da rotação de tela seria a tradução das posições dos *synth widgets* para as novas dimensões da *workspace*. Caso seja desejado manter as posições absolutas dos *synth widgets*, ao rotacionar da orientação de retrato para paisagem, à direita dos *synth widgets*, uma certa quantidade de espaço encontraria-se desocupada. O maior problema, contudo, seria a rotação no sentido contrário — de paisagem para retrato. É impossível manter as posições absolutas dos *synth widgets* sem que esses acabem posicionados fora da tela. Logo, as posições dos *synth widgets* devem ser adaptadas ao rotacionar a tela.

REFERÊNCIAS

Apple, Inc. **Logic Studio Instruments: A Brief History of the Synthesizer**. 2009. <https://documentation.apple.com/en/logicstudio/instruments/chapter_A_section_5.html>. Acessado em: 2018-01-02.

Arduino AG. **Arduino - Introduction**. 2017. <https://www.arduino.cc/en/Guide/Introduction>. Acessado em: 2018-01-02.

BUSH, D.; KASSEL, R. **The Organ: An Encyclopedia**. Routledge, 2006. (Encyclopedia of Keyboard Instruments: The Organ: an Encyclopedia). ISBN 9780415941747. Disponível em: <https://books.google.com.br/books?id=cgDJaeFFUPoC>.

COLLINS, K. **Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design**. MIT Press, 2008. 10 p. ISBN 9780262033787. Disponível em: <https://books.google.co.in/books?id=gnw0Zb4St-wC>.

FOLLE, L. **Implementação de síntese FM na plataforma Arduino Due**. Brasil: [s.n.], 2015. Universidade Federal do Rio Grande do Sul.

Google LLC. **The Activity Lifecycle | Android Developers**. 2017. <https://developer.android.com/guide/components/activities/activity-lifecycle.html>. Acessado em: 2017-12-28.

Google LLC. **BluetoothAdapter | Android Developers**. 2017. <https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#cancelDiscovery()>. Acessado em: 2018-01-03.

Google LLC. **UI Overview | Android Developers**. 2017. <https://developer.android.com/guide/topics/ui/overview.html>. Acessado em: 2017-12-28.

JOHANN, M. An additive synthesis organ with full polyphony on arduino due. In: VI UBIMUS. Växjö, Linnaeus University, Sweden, 2015.

MILLARD, M. Lee de forest, class of 1893: Father of the electronics age. **Northfield Mount Hermon Alumni Magazine**, 1993.

PHILLIPS, B.; STEWART, C.; MARSICANO, K. **Android Programming: The Big Nerd Ranch Guide**. Pearson Education, 2017. (Big Nerd Ranch Guides). ISBN 9780134706078. Disponível em: <https://books.google.com.br/books?id=1igDDgAAQBAJ>.

PHILLIPS, B.; STEWART, C.; MARSICANO, K. Dialogs. In: **Android Programming: The Big Nerd Ranch Guide**. Pearson Education, 2017. (Big Nerd Ranch Guides), p. 211. ISBN 9780134706078. Disponível em: <https://books.google.com.br/books?id=1igDDgAAQBAJ>.

PHILLIPS, B.; STEWART, C.; MARSICANO, K. Introducing fragments. In: **Android Programming: The Big Nerd Ranch Guide**. Pearson Education, 2017. (Big Nerd Ranch Guides), p. 126. ISBN 9780134706078. Disponível em: <https://books.google.com.br/books?id=1igDDgAAQBAJ>.

PIROTTI, R.; JOHANN, M.; PIMENTA, M. A modular platform for a subtractive synthesizer on arduino due. In: SBCM. [S.l.], 2015.

PIROTTI, R.; JOHANN, M.; PIMENTA, M. Design and implementation of an open-source subtractive synthesizer on the arduino due platform. In: SBCM. [S.l.], 2017.

PIROTTI, R. P. **Arquitetura e implementação aberta de um sintetizador subtrativo e aditivo para plataforma de baixo custo**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, Brasil, 2017.

SUI, L. **Global Smartphone OS Market Share by Region: Q3 2017**. [S.l.], 2017. Acessado em: 2018-01-02. Disponível em: <https://www.strategyanalytics.com/access-services/devices/mobile-phones/smartphone/smartphones/reports/report-detail/global-smartphone-os-market-share-by-region-q3-2017>.

Yamaha Corporation. **Chapter 2: FM Tone Generators and the Dawn of Home Music Production**. 2014. <https://usa.yamaha.com/products/contents/music_production/synth_40th/history/chapter02/>. (History, Yamaha Synth 40th Anniversary). Acessado em: 2018-01-02.