LEONARDO GOMES TOROK

UM CONTROLE PARA JOGOS ADAPTADO AO COMPORTAMENTO DO USUÁRIO

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Computação Visual.

Orientador: Prof. Dra. Daniela Gorski Trevisan

Niterói

2015

LEONARDO GOMES TOROK

UM CONTROLE PARA JOGOS ADAPTADO AO COMPORTAMENTO DO USUÁRIO

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Computação Visual

Aprovada em Fevereiro de 2015.

BANCA EXAMINADORA

Prof. Dra. Daniela Gorski Trevisan – Orientador
UFF

Prof. Dr. Esteban Walter Gonzalez Clua
UFF

Prof. Dra. Nayat Sanchez-Pi
ILTC

Niterói
2015

"Dedico esse trabalho à minha mãe Zuléia e meus irmãos Rafael e Claudine, por todo seu amor e carinho”

AGRADECIMENTOS

Em primeiro lugar, agradeço a minha mãe Zuléia, meus irmãos Rafael e Claudine e todos os familiares que me apoiaram e incentivaram nessa etapa importante da minha vida. Vencer todos esses desafios não seria possível sem sua presença, ajuda e carinho. Agradeço a minha namorada Anelisa por seu apoio, ajuda, amor e compreensão, sempre me ajudando nessa jornada.

Agradeço a minha orientadora Daniela, por seus ensinamentos, atenção e tempo despendido para garantir que o presente trabalho pudesse ser concluído e defendido. Deixo meus agradecimentos ao professor Esteban e novamente à professora Daniela pelo grande trabalho realizado no projeto que dá origem a esse trabalho e a todas as publicações derivadas, sempre me direcionando, instruindo e garantindo que conseguisse atingir os objetivos propostos. Agradeço também aos amigos Mateus e Jefferson que participaram ativamente das publicações e de todas as sessões de testes, gerando um grande esforço conjunto sem o qual não poderia ter chegado onde estou.

Não poderia deixar de agradecer a todos os colegas de estudo que me acompanharam em todas as matérias do mestrado, muitas vezes me ajudando e sempre tornando o ambiente de estudo divertido e agradável, e a todos os professores que dedicaram seu tempo para me passar os ensinamentos valiosos que me trouxeram até esse momento.

Deixo meus profundos agradecimentos a todos os voluntários dos testes realizados em nossa instituição de ensino, que permitiram que todos os resultados alcançados fossem avaliados e validados. Por fim, agradeço a Universidade Federal Fluminense pela oportunidade de cursar o mestrado e por todo o apoio institucional para o projeto e à Nvidia, Faperj, CNPq e CAPES por seu apoio financeiro desse trabalho.

RESUMO

Ao interagir com um jogo eletrônico, o usuário espera uma interação fácil e intuitiva. Os métodos de controle atuais são compostos por componentes físicos, os *joysticks*, que possuem um número determinado de botões com tamanho e posição fixos. Contudo, jogos diferentes utilizam botões diferentes e demandam métodos variados de interação. Além disso, o estilo de jogo de cada usuário varia de acordo com suas características pessoais, como tamanho da mão, e experiências anteriores com *videogames.* Esse trabalho propõe um controle virtual inovador baseado em um dispositivo comum com tela sensível ao toque, como um *smartphone* ou *tablet*, que será utilizado como *joystick* para controlar um jogo em um computador ou console, coletando os dados dos toques do usuário e aplicando técnicas de aprendizado de máquina para adaptar em tempo real a posição e o tamanho de seus botões virtuais, minimizando os erros de interação do usuário e fornecendo uma experiência de uso mais agradável. Também são apresentados diversos testes de usabilidades com grupos de usuários de dois perfis diferentes para dois gêneros distintos de jogos. Com os resultados obtidos espera-se uma contribuição que possa iniciar uma nova forma de projetar jogos para computadores e consoles, permitindo que os controles sejam projetados especificamente para cada jogo e se adaptem a ergonomia do usuário.

Palavras-chave: Interfaces adaptativas, controle para jogos adaptativo, aprendizado de máquina, controle para jogos, dispositivos móveis, telas sensíveis ao toque

ABSTRACT

When interacting with a videogame, the user expects an easy and intuitive interaction. The current controllers are composed by physical components, the joysticks, which are composed by a number of buttons with fixed size and position. However, different games use different buttons and demand different interaction methods. Besides that, the game style for each user changes following his personal characteristics, such as hand size and previous experiences with videogames. This work proposes an innovative virtual controller based on a regular device with a touchscreen, like a smartphone or tablet, that will be used as a joystick to control a game running on a computer or game console, collecting user touch data and applying machine learning technics to adapt the position and size of the virtual buttons in real time, minimizing user interaction errors and providing a more pleasant user experience. The results of several usability tests with user groups with two different profiles for two different game genres are also presented. With the obtained results, it is expected a contribution that can start a new way to project games for computers and consoles, allowing controllers that can be projected specifically for a game and will adapt to the user's ergonomy.

Keywords: Adaptive interfaces, Adaptive game controller, machine learning, game controller, mobile devices, touchscreen

LISTA DE ILUSTRAÇÕES

Figura 1: Controles para smartphones (MOGA Pocket e Samsung GamePad).....14
Figura 2: O controle da empresa Donya para jogos de smartphones.....14
Figura 3: Posição tridimensional do dedo capturada pelos sensores ao se aproximar.....20
Figura 4: 3 layouts distintos para os alvos e distrações.....22
Figura 5: Arquitetura do projeto.....27
Figura 6: Exemplo das mensagens geradas pelo aplicativo móvel.....29
Figura 7: Modelagem das classes responsáveis por representar o joystick e seus botões.....31
Figura 8: Modelagem da camada de persistência de dados.....34
Figura 9: Visão simplificada da modelagem principal do projeto.....37
Figura 10: Demonstração do processo de adaptação.....40
Figura 11: Layout padrão do controle.....46
Figura 12: Os equipamentos utilizados para os testes.....47
Figura 13: As configurações iniciais e finais para um usuário.....51
Figura 14: Precisão média dos usuários inexperientes.....55
Figura 15: Precisão média dos usuários experientes.....56
Figura 16: Precisão média de todos os usuários.....56

LISTA DE TABELAS

Tabela 1: Resultados dos testes iniciais.....49
Tabela 2: Taxa de sucesso para os usuários inexperientes.....52
Tabela 3: Taxa de sucesso para os usuários experientes.....53
Tabela 4: Opinião dos usuários quanto à dificuldade de uso dos controles.....57

LISTA DE ABREVIATURAS E SIGLAS

ARFF	*Attribute-Relation File Format*
AWT	*Abstract Window Toolkit*
API	*Application Programming Interface*
DAL	*Data Access Layer*
INPI	*Instituto Nacional da Propriedade Industrial*
JPEG	*Joint Photographic Experts Group*
NES	*Nintendo Entertainment System*
PC	*Personal Computer*
SDK	*Software Development Kit*
SNES	*Super Nintendo Entertainment System*
VB	*Visual Boundary*
VB/SDB	*Visual Boundary or Shortest Distance to Circle Boundary*
VB/SDC	*Visual Boundary or Shortest Distance to Circle Center*

SUMÁRIO

Capítulo 1 – Introdução....12

1.1 Motivação....12

1.2 Objetivo e escopo....16

1.3 Estrutura do trabalho....17

Capítulo 2 – Trabalhos relacionados....19

2.1 Interfaces adaptativas....19

2.2 Interfaces para jogos....20

2.2.1 Dispositivos móveis....20

2.2.2 Consoles, portáteis e computadores....23

Capítulo 3 – Controle adaptativo para jogos....26

3.1 Interface proposta e implementação....26

3.1.1 Arquitetura....26

3.1.2 Modelagem e implementação....30

3.2 Adaptações ao usuário e ao jogo....39

3.3 Aprendizado de máquina....42

3.3.1 K-means....43

Capítulo 4 – Testes de usabilidade....46

4.1 Metodologia....46

4.1.1 Participantes e equipamentos....46

4.2 Configuração dos parâmetros de adaptação....47

Capítulo 5 – Resultados....51

5.1 Análise objetiva....51

5.2 Análise subjetiva....56

Capítulo 6 – Conclusão....58

6.1 Trabalhos futuros....58

6.2 Limitações....59

6.3 Contribuições.....60

Referências.....61

Apêndice A – Documentação dos métodos das classes.....64

CAPÍTULO 1 – INTRODUÇÃO

1.1 MOTIVAÇÃO

O mercado de jogos eletrônicos, ou *videogames*, evoluiu de uma diversão de nicho até se tornar a maior indústria do entretenimento. Esse crescimento foi acompanhado por avanços tecnológicos permitindo que os jogos atingissem um grau cada vez mais alto de complexidade em suas mecânicas e de fidelidade visual ao representar a realidade. Começando com mecânicas simples e objetivos triviais representados com poucos *pixels,* avançando até jogos com grandes mundos complexos, repletos de personagens com suas próprias inteligências artificiais, dezenas de formas de interação, histórias cinematográficas e uma qualidade visual quase fotorrealista. Mesmo com uma diferença tão notável, certos aspectos permanecem os mesmos. Todo jogo irá permitir controlar um (ou mais de um) avatar no mundo virtual, através de uma interface projetada com o objetivo de fornecer a melhor experiência possível, tradicionalmente um *joystick* físico. Durante o *design* de um jogo, o aspecto principal a ser trabalhado, aquele que define de forma mais direta a diversão que o jogo irá proporcionar é o *gameplay* do jogo: a forma como as fases são projetadas, a maneira como o avatar será representado e o tipo de ações que ele poderá realizar (KOSTER, 2013). A experiência de *gameplay* é implementada através das regras de jogo, executadas através da interface do jogo. Nesse sentido, a interface, composta pelos controles do jogo e sua interface visual, deve estimular o usuário a interagir. Esse processo de criação abrange diversas características que irão determinar a impressão final do usuário e o nível de imersão e interesse despertados. Dentre diversos aspectos, os desafios devem ser balanceados corretamente ou o usuário poderá ficar frustrado com uma dificuldade grande demais ou entediado pela falta de desafio (SCHELL, 2008). Historicamente, em geral as experiências de jogo mais memoráveis sempre estiveram associadas a esquemas de controle bem projetados, fornecendo o equilíbrio entre intuitividade, permitindo ao jogador aprender rapidamente a interagir com o jogo, e profundidade, permitindo que o usuário realize uma grande variedade de ações simples ou complexas da forma mais simples. A relação entre a qualidade final de um jogo e seus controles é fácil de compreender: os controles são a ponte responsável por realizar a correspondência entre a vontade do usuário e o que o seu personagem virtual realmente fará. Em um ambiente ideal, haveria uma correspondência total entre ambos.

A evolução tecnológica dos jogos, adicionando mais possibilidades de interação, precisou ser acompanhada por uma evolução equivalente nos dispositivos de controles, que

deveriam ser capazes de oferecer cada vez mais recursos de interação. No primórdio dos *games*, os jogos utilizavam poucos botões. Um Atari 2600 utilizava um controle com apenas um botão e um manche capaz de informar uma dentre 8 direções. Em um mundo de jogos complexos, os controles modernos se tornaram complexos, contando com dezenas de botões, alavancas analógicas e direcionais, fornecendo uma grande variedade de entradas possíveis. Com essas modificações, uma parte da intuitividade foi perdida ao oferecer ao usuário uma interface complexa e intimidadora para jogadores não-experientes. Outra limitação, inerente a componentes físicos, é a sua imutabilidade. Um *joystick* será o mesmo para qualquer jogo, independente da sua necessidade. Alguns usarão apenas um ou dois botões, além do direcional, enquanto outros jogos poderão fazer uso não só de todos os botões mas também recorrer a combinações entre eles para fornecer mais possibilidade de entradas de comandos. Tendo que fornecer uma interface unificada, o *joystick* se torna um periférico genérico, não-otimizado para nenhum jogo específico e sim buscando um ponto de equilíbrio comum que permita atender de forma satisfatória todos os tipos de jogos. Além disso, o controle possui um tamanho físico único, que será usado por todos os jogadores independentemente de suas características ergonômicas pessoais, como tamanho da mão e flexibilidade nos dedos. Nesse ponto, os controles não avançaram significativamente em décadas: mais botões eram adicionados mas o conceito básico do controle se manteve.

Em busca de uma revolução que trouxesse a intuitividade dos controles simples dos jogos antigos, a indústria passou a buscar a solução em sensores de movimento, como o Wii Mote da Nintendo, seguindo pelo PlayStation Move da Sony e o Kinect da Microsoft. Dessa vez, o próprio movimento do jogador era utilizado como método de entrada, efetivamente utilizando o corpo do usuário como *joystick*. A ideia era simples e brilhante: mais intuitivo do que jogar boliche apertando uma sequência de botões seria simplesmente permitir ao jogador realizar o movimento de arremesso da bola. Com isso, uma nova geração de jogos utilizando controles simplificados atraiu uma multidão de jogadores casuais que não tinham o hábito de jogar. Contudo, a nova solução tinha um grande problema. Apesar da facilidade de uso, a variedade de entradas possíveis e a precisão dos controles deixavam muito a desejar, impedindo que diversos gêneros fizessem a transição para esquemas alternativos de controles. Apesar do uso de sensores de movimento ser bem sucedido em jogos específicos, a grande maioria ainda necessita da precisão e variedade de comandos de um *joystick*. Com isso, a indústria permaneceu em grande parte em seu foco inicial nos controles tradicionais utilizados há várias décadas.

Figura 1: Controles para smartphones: à esquerda o controle MOGA Pocket e à direita o Samsung GamePad.

Figura 2: O controle com ventosas da empresa japonesa Donya para jogos de smartphones que utilizam controles virtuais.

Em paralelo às revoluções em controles para consoles e computadores, houve o surgimento e crescimento acelerado de uma nova classe de dispositivos: os *smartphones*, seguidos pelos *tablets*. Apesar de utilizar uma tela *touchscreen* para a maior parte das interações, a plataforma contava com diversos sensores destinados a melhorar a experiência do usuário. Acelerômetros permitiam detectar a orientação do dispositivo e girar a tela para a direção correta, sensores GPS permitem que aplicativos de navegação detectem a posição do usuário em tempo real, dentre muitos outros. Como todo dispositivo com algum poder computacional e uma tela, rapidamente surgiram jogos para a plataforma, baseados no uso da tela sensível ao toque. Dessa vez, a intuitividade estava em poder interagir com os objetos no jogo de acordo com uma física real: tocar e arrastar um objeto na tela realmente iria movê-lo,

como um objeto real. Mas os desenvolvedores não se limitaram a isso. Jogos de corrida passariam a utilizar o acelerômetro para detectar a inclinação do celular, que poderia ser utilizado como um volante. Jogos poderiam utilizar a posição do usuário para disparar ações. Com isso, o público casual passou a utilizar os *smartphones* como forma de entretenimento, jogando em um dispositivo que eles já possuíam e que era profundamente intuitivo. Porém, nem todos os gêneros de jogos podiam ser controlados com controles tão simplórios, chegando ao mesmo problema encontrado pelos sensores de movimentos em consoles. Em uma tentativa de solucionar o problema, muitos jogos passaram a utilizar um *joystick* virtual: os botões de um controle físico eram exibidos na tela e o usuário poderia pressioná-los com um toque na tela *touch*. A grande vantagem dessa solução, além de possuir as mesmas capacidades extensas de comandos de um controle tradicional, era que cada jogo podia projetor o controle mais adequado para seu estilo, contando com apenas os botões necessários e criando uma interface mais limpa e intuitiva. Apesar de aparentar trazer o melhor dos dois mundos, os controles virtuais sofriam com a falta de precisão das telas *touch* e a falta de *feedback* físico ao pressionar os botões, tornando-os menos eficientes que *joysticks* físicos.

Em busca de uma solução, diversas alternativas foram propostas, inclusive com a criação de controles físicos que pudessem ser utilizados por *smartphones*, como na Figura 1. Contando com um suporte para encaixar o dispositivo e uma comunicação *Bluetooth*, esses *joysticks* permitem jogar diversos jogos para dispositivos móveis. Mesmo assim, a solução não se tornou comum: o preço elevado do periférico e a necessidade de carregar um controle maior do que o próprio *smartphone* se mostraram uma barreira a sua adoção. O suporte em *software* se tornou outro grande problema. Como apenas uma minoria possui um controle físico para seu celular, a grande maioria dos jogos não suporta o uso de controles físicos, um problema acentuado pela falta de padronização entre os diversos fabricantes de tais periféricos. Com as limitações na adoção do periférico, o uso de controles virtuais, mesmo com suas limitações em suas versões atuais, permaneceu uma constante. Outra solução é a adotada pelo controle na Figura 2. Aderindo a tela *touch* através de ventosas, o controle deve ser posicionado em cima dos controles virtuais do jogo específico. Ao pressionar um botão, o controle irá pressionar a tela abaixo e realizar a ação, criando uma extensão física para o controle virtual. Contudo, apesar de incluir diversos *layouts* para se adaptar a vários controles virtuais, não há garantia de que todos os jogos serão atendidos. Além disso, a precisão e funcionalidade ainda são limitadas se comparada com as alternativas físicas anteriores.

O presente trabalho propõe uma interface inovadora, que busca unir o melhor dos *joysticks* físicos e virtuais: um controle executado em um dispositivo com tela sensível ao

toque utilizado para controlar um jogo em um computador tradicional. Com isso, o *joystick* físico é substituído por um controle virtual, capaz de apresentar apenas os botões pertinentes ao jogo em execução, criando uma interface mais simples e intuitiva. Essa característica permite o projeto de um controle que, no futuro, poderá ser configurado pelo próprio jogo, seguindo o projeto de interação desejado pelo *game designer* do mesmo, criando um *layout* específico e otimizado para cada jogo. Essa se torna o maior avanço em relação aos controles físicos: interfaces genéricas sem adaptações específicas para cada jogo.

Porém, ainda existe a questão da falta de precisão e *feedback* tátil dos controles virtuais, criando uma grave limitação em sua performance. Para solucionar esse problema, o trabalho propõe uma interface adaptativa, capaz de analisar dados de comportamento do usuário, como toques na tela, e determinar um posicionamento e tamanho ótimo para os botões na interface virtual, criando uma interface personalizada adaptada à ergonomia do jogador. Para isso, o controle utilizará algoritmos de aprendizado de máquina para analisar os toques do usuário na tela e otimizar os botões em busca da redução dos erros do usuário e de uma melhor ergonomia para o controle, corrigindo suavemente os botões na tela. Essa característica irá permitir que o *layout* padrão (ou a configuração especificada por um *game designer*) seja otimizada para se adequar à ergonomia de cada jogador, criando um controle adaptado ao jogo e ao jogador.

1.2 OBJETIVO E ESCOPO

O objetivo dessa dissertação é propor uma interface inovadora, baseada em um controle virtual adaptativo para dispositivos *touchscreen* capaz de controlar um jogo em execução em um computador tradicional. Essa interface deve ser capaz de melhorar a precisão do usuário de forma significativa se comparado com uma solução não-adaptativa. O controle deve ser capaz de ter flexibilidade em relação à quantidade, tamanho e posição dos botões, permitindo que no futuro o próprio criador de um jogo possa definir um *layout* inicial dos botões que corresponda a sua visão da interação com o jogo. Mesmo que o *designer* do jogo não possua os conhecimentos adequados de usabilidade para garantir uma interface ótima, o mecanismo de aprendizado deverá ser capaz de corrigir as imperfeições de usabilidade ao mesmo tempo que adapta o *layout* padrão às necessidades específicas do usuário.

A criação de um controle adaptativo funcional e a avaliação da sua performance compõe o escopo deste trabalho. Inicialmente, é preciso determinar um algoritmo adequado de aprendizagem, capaz de analisar os toques do usuário na tela e inferir as posições ideais para os botões, bem como um algoritmo para determinar o tamanho ideal de cada botão. As

definições da metodologia de adaptação, incluindo a definição dos limites e a velocidade das adaptações também estão incluídas no projeto. Todas essas variáveis foram analisadas a partir do desenvolvimento de um protótipo funcional, permitindo comparar a relação entre as abordagens utilizadas e a performance final do controle. Com isso, o escopo inclui não só a criação de um controle e a definição das maneiras como a adaptação ocorreria, mas também buscou a maneira ótima de executar as alterações. O protótipo deveria ser capaz de controlar um jogo em um computador, o que resultou em avaliações de tecnologias para a comunicação sem fio e no desenvolvimento de um *software* cliente para receber os comandos do controle e transformá-los em ações no jogo.

Além do projeto e criação do controle em si, o escopo ainda abrange a avaliação da sua capacidade de melhorar a performance do usuário, definindo os parâmetros relevantes para a medida do desempenho do usuário e a coleta desses valores em sessões de testes.

1.3 ESTRUTURA DO TRABALHO

O capitulo 1 apresenta os aspectos relacionados à motivação que levaram ao desenvolvimento do trabalho, bem como o principal objetivo a ser alcançado com o mesmo. Por fim, a introdução definiu o escopo do projeto, fornecendo uma direção para as seções seguintes.

O capítulo 2 é dedicado aos trabalhos relacionados, apresentando inicialmente uma revisão bibliográfica de artigos, livros e projetos sobre interfaces adaptativas. A seguir, discute o estado atual das interfaces para jogos tradicionais já existentes, focando inicialmente em dispositivos móveis e depois nos dispositivos de jogos tradicionais como consoles, portáteis e computadores.

O tema abordado no capítulo 3 é o próprio controle adaptativo proposto. A primeira subseção trata da interface proposta e dos detalhes da implementação, fornecendo uma descrição cuidadosa das decisões do *software* e suas peculiaridades. A seção seguinte aborda as adaptações ao usuário e ao jogo, detalhando os diferentes tipos de adaptação implementados no controle, suas regras e limites. A seção final do capítulo se dedica ao aprendizado de máquina e aborda especialmente o algoritmo selecionado para o controle adaptativo, o K-means.

O capítulo 4 descreve a metodologia utilizada nos testes de usabilidade. São apresentados os perfis dos participantes, a configuração do experimento e as avaliações realizadas para determinar os melhores parâmetros de adaptação para o algoritmo de aprendizado e a escolha dos jogos para os testes.

Os resultados dos testes com usuários e a avaliação dos resultados são discutidos no capítulo 5. A primeira seção apresenta os resultados dos testes objetivos e a análise estatística das informações de performance do usuário seguida da análise de significância dos resultados obtidos. A seção seguinte apresenta os testes subjetivos, analisando as respostas dos formulários preenchidos pelos usuários após as sessões de avaliação.

A conclusão do trabalho é apresentada no início do capítulo 6, seguida das limitações do trabalho atual. Também neste capitulo são discutidos caminhos para trabalhos futuros, incluindo estudos e melhorias planejados. Por fim, o ultimo capitulo termina apresentando as contribuições geradas pelo presente trabalho como publicações científicas, demonstrações e outras.

CAPÍTULO 2 – TRABALHOS RELACIONADOS

2.1 INTERFACES ADAPTATIVAS

Uma interface de usuário adaptativa é um sistema de *software* interativo capaz de otimizar sua habilidade de interagir com o usuário baseado na experiência parcial dessa interação (LANGLEY, 1997). A adaptação de sistemas interativos descreve as mudanças que a interface irá executar para aumentar a usabilidade ou a satisfação do usuário. Segundo Bezold e Minker (2011, p. 41), sistemas adaptativos interativos observam a interação usuário-sistema e geram conclusões a partir das observações sobre as características e preferências do usuário.

Seguindo ainda Bezold e Minker (2011, p. 43), temos definido que sistemas adaptativos interativos precisam representar as observações do usuário em um formato comum para possibilitar que um sistema de modelagem do usuário consiga chegar a conclusões. Essas observações são chamadas de eventos básicos, incluindo entradas do usuário, como pressionamento de botões ou fala, ou reações do sistema, como mudanças de estado interno e propriedades. A visão do comportamento do usuário para um sistema interativo está limitada a esses eventos. Por isso, a interação sistema-usuário consiste em uma sequência de eventos de baixo nível, que podem ser armazenados em logs (no presente caso, um banco de dados). Os dados de log em um sistema adaptativo interativo podem ser processados *online* ou armazenados para processamento *offline*. Para incluir características do usuário atual, um componente de modelagem do usuário aplica algoritmos de modelagem em tempo real. Em alguns casos, dados gravados em logs são preferíveis, por exemplo, para treinar um modelo de usuário inicial ou características estáticas do usuário que não mudam com o tempo. Para esse caso, o sistema interativo armazena os eventos observados em arquivos de log e o sistema de modelagem do usuário os processa *offline*. No caso desse trabalho, o processamento será *online*, permitindo acompanhar as mudanças constantes no padrão de interação. Além disso, os tipos de eventos que contribuem para o processo de modelagem do sistema podem ser limitados aos eventos que forem relevantes para as adaptações. Para o controle virtual, as mudanças de estado internas do sistema não serão relevantes enquanto os dados de toques do usuário serão.

Abordagens orientadas aos dados reconhecem ações do usuário baseadas nos dados de amostra. Essas abordagens coletam dados de treinamento envolvendo todas as ações relevantes. Por exemplo, os dados podem ser coletados em sessões de testes com usuários,

sendo tratadas por algoritmos de aprendizado de máquina para extrair ações do usuário a partir da interação. (BEZOLD; MINKER, 2011, p.46). Para o projeto atual, a coleta dos dados não será feita durante as sessões de teste e sim durante a própria interação do usuário, coletando e analisando constantemente os dados em tempo real.

2.2 INTERFACES PARA JOGOS

2.2.1 DISPOSITIVOS MÓVEIS

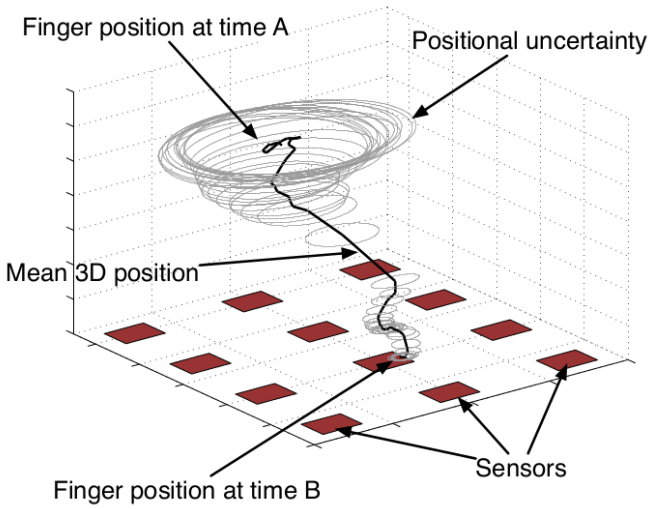

Figura 3: Posição tridimensional do dedo capturada pelos sensores ao se aproximar. As elipses mostram o nível de incerteza (ROGERS et al., 2010, p. 578).

Em Rogers et al. (2010), foram desenvolvidos modelos que tratam toques de usuários na tela sem padrões definidos e usam esses dados para lidar com os comandos do usuário para o sistema. Sensores capacitivos SK7, em um conjunto de 12 sensores distribuídos em 4 colunas e 3 linhas, foram utilizados. Esses sensores são capazes de informar a posição dos dedos do usuário em um espaço tridimensional, apesar da sua relativa baixa resolução. Além dos limites dos sensores, o nível de certeza sobre a posição real dos dedos diminui à medida que a distância para o sensor aumenta, gerando uma entrada ainda mais incerta, como indicado na Figura 3. Para solucionar o problema e aumentar a capacidade do sistema de prever a entrada do usuário à medida que a entrada se torna menos confiável, é utilizada uma abordagem Bayesiana para modelar o sistema. Cada sensor fornece uma curva de valores que varia de acordo com o tempo, com a base de dados sendo compostas por amostras coletada com filtros de partículas. Por fim, o trabalho demonstra uma aplicação com um mapa que irá

rolar para um certo ponto de interesse e aproximar ou afastar a visão a partir dos comandos incertos do usuário, tratados pelo sistema para determinar a real intenção do usuário com os comandos.

Com um objetivo similar, Weir et al. (2012) utilizou aprendizado de máquina para aprender modelos de toques específicos para cada usuário e utilizá-los para melhorar a precisão dos toques em dispositivos móveis. Nesse estudo, foi proposto o mapeamento dos dados ou localizações dos toques para o ponto do toque correto, baseado no comportamento histórico dos toques de um usuário específico. Uma abordagem de aprendizado de máquina, consistindo em um algoritmo de regressão não-paramétrico baseado no processo Gaussiano, foi utilizada para tratar o conjunto de amostras e criar uma correção particular para o usuário, ao contrário de soluções anteriores que se baseavam em *hardwares* personalizados ou em algoritmos de compensação genéricos. O dispositivo utilizado foi o Nokia N9, com o sistema operacional Meego, selecionado por ser o único dispositivo que fornece os dados brutos do sensor além dos pontos de toque determinados pelo sistema operacional. Para os testes, um coletor de dados foi desenvolvido em Python e PyGame, consistindo em um aplicativo que exibe um marcador em uma posição que deve ser tocado pelo usuário, em uma área da tela similar a ocupada pelo teclado virtual quando o telefone se encontra na posição horizontal. Para cada toque, o aplicativo armazenava a localização correta do toque (posição do marcador), a posição do toque real do usuário reportada pelo sistema operacional e os dados brutos dos sensores capacitivos da tela. Uma sessão de testes com 8 participantes foi realizada, aplicando os algoritmos em conjuntos de 1000, 800, 400 ou 200 toques com marcadores com 2, 3 ou 4 mm de largura e altura, avaliando em seguida as melhorias na precisão. Aplicando os algoritmos sobre os dados brutos dos sensores, a precisão dos toques pode ser aprimorada em média 23.47% para um marcador de 2 mm, 14.20% para 3 mm e 4.7% para 4 mm. Ao utilizar os dados de toques informados pelo sistema operacional (que seriam os únicos dados acessíveis na maioria dos sistemas operacionais para dispositivos móveis) a melhora na precisão foi ligeiramente melhor que no caso anterior: 23.79% para 2 mm, 14.79% para 3 mm e 5.11% para 4 mm. O trabalho aponta possíveis melhorias para a experiência do usuário e a possibilidade do uso de botões menores em interfaces para *touchscreen*.

O trabalho de Bi e Zhai (2013) conceitualizou entradas de usuários em telas sensíveis ao toque como um processo incerto e usou critérios de seleção estatística para determinar o componente visual que o usuário desejava tocar a partir da análise das probabilidades para cada componente. Para isso, foi usado o critério de toques Bayesiano, que trata o evento do

toque como um sinal estatístico da intenção do usuário, inferindo a probabilidade da intenção para cada alvo possível. Primeiramente, as regras de Bayes são utilizadas para estimar a probabilidade que cada componente possui de ser o alvo do toque. Em seguida, é utilizado o princípio da distribuição Gaussiana dupla dos toques, combinando um componente de precisão absoluta com um componente de precisão relativa ao tamanho do alvo para criar uma distribuição Gaussiana dos pontos para um alvo. As duas etapas são combinadas para criar o critério de toques Bayesiano. Para verificar a performance da correção proposta, foram realizados testes com um grupo de 18 usuários, conduzido através de um aplicativo Android em um *smartphone* Galaxy Nexus. Os participantes deveriam realizar a tarefa de tocar em um círculo verde na tela, cercado por círculos cinzas presentes como elementos de distração (Figura 4), de forma similar a interfaces reais para dispositivos móveis e seus múltiplos componentes em uma única tela. A partir dos resultados dos testes, a técnica de critério de toque Bayesiano proposta foi comparada a técnicas mais tradicionais:

- *Visual Boundary* (VB): o círculo selecionado é aquele que contém o ponto do toque em seu interior.
- *Visual Boundary or Shortest Distance to Circle Boundary* (VB/SDB): aplica a regra anterior. Caso o ponto não coincida com nenhum círculo, o toque é associado ao círculo no qual algum ponto de sua circunferência tenha a menor distância até o ponto.
- *Visual Boundary or Shortest Distance to Circle Center* (VB/SDC): aplica a regra VB primeiro. Caso o ponto não coincida com nenhum círculo, o toque é associado ao círculo cujo centro tenha a menor distância até o ponto.

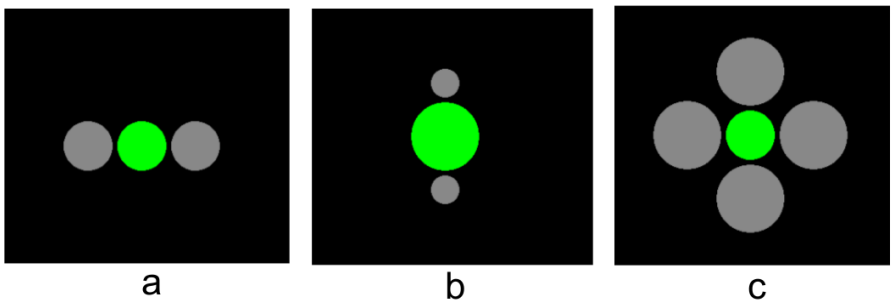

Figura 4: 3 layouts distintos para os alvos (círculos verdes) e distrações (círculos cinza), com tamanhos variados para cada estudo. A distância entre o limite do alvo e da distração é de 0.5 mm (BI; ZHAI, 2013, p. 57).

Os testes indicaram que a nova técnica reduzia a taxa de erro em 70% comparada à técnica VB (utilizada nas aplicações reais de *smartphones*), em 26% comparada à VB/SDB e 6% comparada à VB/SDC, indicando que pode resultar em melhorias significativas de usabilidade. Contudo, mesmo melhorando a precisão do usuário, a interface não corrige o posicionamento de seus componentes: o usuário irá continuar errando e a interface irá sempre calcular a posição correta e corrigir o comando.

No contexto de estudos que buscam não só corrigir o erro do usuário, mas também modificar a própria interface para evitar erros futuros, temos trabalhos como Baldwin e Chai (2012), que propõe a modificação dinâmica das áreas das teclas de teclados virtuais para dispositivos móveis. Primeiramente, a proposta do trabalho se diferencia de casos anteriores por não se basear em sistemas de calibração *offline* para treinar seu algoritmo, buscando uma abordagem que possa ser aplicada no contexto real de uso, sem a necessidade de milhares de pontos para realizar um treinamento do aprendizado. Três diferentes técnicas de coleta de dados são propostas e testadas para gerar e atualizar dinamicamente os modelos de redimensionamento das teclas virtuais. Os resultados alcançados através de uma metodologia de coleta de informações que infere melhorias baseadas no vocabulário e nas correções de erros de digitação se mostrou comparável às custosas alternativas *offline* e ao mesmo tempo reduziu a taxa de erro em 10,4% quando comparada a teclados virtuais tradicionais, permitindo o treinamento de um algoritmo de aprendizado a partir de apenas uma semana de uso normal do usuário médio de dispositivos móveis. Esse tipo de adaptação tem algumas desvantagens, como o uso limitado a contextos de processamento de linguagens, apenas se adaptando ao modelo do idioma e ao padrão de digitação do usuário.

2.2.2 CONSOLES, PORTÁTEIS E COMPUTADORES

O segmento de controles para computadores, consoles e portáteis sempre foi marcado por melhorias incrementais, com poucas revoluções em décadas de história. Basicamente, o *joystick* teve sua primeira versão baseada nos controles das antigas máquinas de arcade, com um ou mais botões para realizar ações e uma alavanca capaz de selecionar as quatro principais direções (cima, baixo, esquerda e direita) e quatro diagonais representando o uso conjunto de duas direções adjacentes. Posteriormente, os controles adotaram aquele que se tornaria o *design* atual, com o controle tradicional do NES da Nintendo, possuindo um botão direcional que cumpria a função do manche de arcade ocupando menos espaço e de forma mais ergonômica. A partir daí, os controles ganhariam mais botões: alavancas analógicas que

permitiam um melhor deslocamento em jogos com ambientes tridimensionais, botões no topo para serem acionados com os dedos indicador e médio, etc.

Apesar das mudanças, o conceito básico de controle permaneceu inalterado. Porém, jogos mais complexos trouxeram controles mais refinados, com uma grande quantidade de botões e uma operação bem menos trivial que as versões já obsoletas. Apesar de permitir uma maior riqueza de interação, houve um aumento da complexidade da tarefa de controlar seu avatar virtual. Em busca de controles mais simples, surgiram alternativas como o Wii Mote da Nintendo: um controle baseado em sensores de movimento, permitindo que o usuário usasse movimentos naturais do seu corpo para interagir com um jogo. O grande sucesso dessa abordagem logo trouxe competidores, como o PlayStation Move da Sony e o Kinect, que permite a captura de movimentos sem nenhum controle físico. Apesar do sucesso inicial, todas essas alternativas rapidamente desapareceram. O grande problema é que os sensores de movimento não tinham a mesma variedade ampla de opções de entrada de controles tradicionais, limitando seu uso a gêneros mais simples (dança, jogos de esporte simples) e gerando versões simplificadas de jogos de gêneros mais complexos. Com isso, os controles de *videogames* para a grande maioria dos gêneros permaneceu basicamente no patamar anterior: uma evolução de um conceito original da década de 1980. Mesmo assim, nenhuma dessas interfaces prevê a capacidade de se adaptar ao jogo, trabalhando sempre com o conceito de criar uma interface genérica e idêntica para todos os jogos.

Um produto que visa modificar a forma como controles de jogos são projetados e adicionar a capacidade do desenvolvedor configurar aspectos do controle é o Steam controller, da Valve. Projetado para uso em jogos para PCs, o *joystick* da Valve se diferencia por substituir as tradicionais alavancas analógicas, utilizadas para informar a direção, por dois *touchpads* clicáveis. Cada *touchpad* é capaz de gerar um *feedback* háptico para o usuário, programado pelo desenvolvedor do jogo, permitindo vibrar regiões de sua superfície, simular relevo e textura, com o objetivo de criar um controle que reúna os melhores aspectos dos *joysticks* e do uso de um *mouse* e teclado. O controle é projetado de forma aberta, permitindo que os usuários personalizem o controle ou projetem novas peças para substituir as utilizadas no controle padrão. Dessa maneira, o Steam controller busca inovar não somente a forma do controle como a capacidade de personalização da experiência, tanto por parte do usuário quando do desenvolvedor do jogo, um objetivo similar ao desse trabalho. A grande diferença é que, por ser constituído de uma componente físico, a alteração e personalização do controle depende da capacidade de produzir componentes físicos para substituir as peças incluídas no controle, algo que exige equipamento específico e um maior investimento, além de sofrer com

as limitações físicas do controle. A flexibilidade total da interface é um ponto chave desse projeto, obtida com o uso de um controle virtual que pode representar qualquer forma em seu *layout*.

Existem poucos trabalhos relacionados a interfaces adaptativas e seu uso para o projeto de controles para jogos. Uma solução comercial atual é o GestureWorks Gameplay, que permite que o usuário jogue utilizando um *smartphone* Android como controle, podendo personalizar o *layout* do *joystick*. Contudo, essa interface depende do usuário para sua criação e não dispõe de nenhuma adaptação ao comportamento do usuário. Esse último fator é considerado importante porque nem todo comportamento do usuário durante a interação é consciente, ou seja, o usuário pode errar e não saber se errou ou porque errou. O conceito de permitir que o criador do jogo defina a interface também não se encontra presente, sendo um objetivo futuro do presente trabalho, refletido em sua arquitetura e modelagem pensada na capacidade de personalização da experiência através de várias fontes (*layout* padrão do desenvolvedor do jogo e adaptação ao usuário).

A base desse trabalho vem desde Zamith et al. (2013), que propõe um *framework* para um controle personalizado baseado em um dispositivo móvel *touchscreen*, capaz de se adaptar ao usuário de acordo com o padrão de toques na tela e criar uma interface otimizada para sua ergonomia e para o jogo específico.

CAPÍTULO 3 – CONTROLE ADAPTATIVO PARA JOGOS

3.1 INTERFACE PROPOSTA E IMPLEMENTAÇÃO

O controle adaptativo proposto é baseado em um componente físico (um *smartphone* ou *tablet*) e um *software* em execução no computador aonde o jogo é executado. O dispositivo móvel apresenta a interface, na forma de um controle virtual, e coleta os dados de entrada do usuário. A interface apresenta diversos botões de ação, que realizam comandos específicos para cada jogo, e um botão direcional com o qual o usuário pode indicar uma entre quatro direções e quatro diagonais.

Cada pressionamento de um botão virtual será enviado via *Bluetooth* para o computador, que receberá cada comando através do *software* cliente em execução. Esse comando será convertido em um evento simulado de teclado, que irá realizar a ação no jogo. O aprendizado de máquina é realizado em paralelo, executado no próprio dispositivo móvel e analisando os últimos toques do usuário na tela em intervalos de tempo regular. Após o aprendizado, as alterações necessárias passam a ser executadas gradualmente, alterando de forma suave o tamanho e posição dos botões até os valores ótimos.

A escolha da tecnologia de comunicação passou pela avaliação das várias tecnologias possíveis. O uso de uma comunicação simples via redes Wi-Fi com *sockets* seria a solução mais direta, mas apresentava diversas limitações. Seria necessário que ambos os dispositivos, PC e *mobile*, estivessem em uma mesma rede, o que seria uma inconveniência para os usuários. Uma variante dessa tecnologia, chamada Wi-Fi Direct permite que dois dispositivos se comuniquem diretamente através de uma conexão *wireless* dedicada, sendo inclusive a solução utilizada no console Wii U. Contudo, essa tecnologia não é suportada em todos os dispositivos Android, sendo limitada a aparelhos recentes. Por fim, a solução adotada foi o *Bluetooth*, presente em todos os dispositivos Android e em muitos computadores (além de adaptadores USB de baixíssimo custo), que ofereceu uma solução simples e de baixa latência capaz de atender os requisitos propostos.

3.1.1 ARQUITETURA

O uso de um dispositivo móvel como *joystick* trouxe a necessidade do projeto de uma arquitetura capaz de dividir as diversas etapas necessárias para o funcionamento do controle em dois aplicativos sendo cada um executado em um dispositivo diferente, com a comunicação entre ambos sendo realizada através de *sockets Bluetooth*.

Para esse trabalho, a plataforma Android foi selecionada, devido ao seu SDK maduro e amplo uso no mercado, com o aplicativo sendo desenvolvido na linguagem Java. Para realizar o aprendizado de máquina, foram utilizadas as bibliotecas do Weka, que fornecem acesso a uma grande variedade de algoritmos diferentes. Para esse caso específico, foi utilizada uma versão especialmente modificada para ser suportado pela máquina virtual Java do sistema operacional Android. A comunicação *Bluetooth* foi implementada com as bibliotecas nativas do SDK do Android.

O cliente *desktop*, responsável por receber os comandos, foi implementado na linguagem Java, seguindo o padrão do aplicativo móvel, utilizando a tecnologia AWT para aplicações *desktop*. A comunicação *Bluetooth* foi implementada com a biblioteca BlueCove, permitindo que o cliente abra um *socket Bluetooth* e aceite informações enviadas em forma de texto simples após o pareamento dos dois dispositivos.

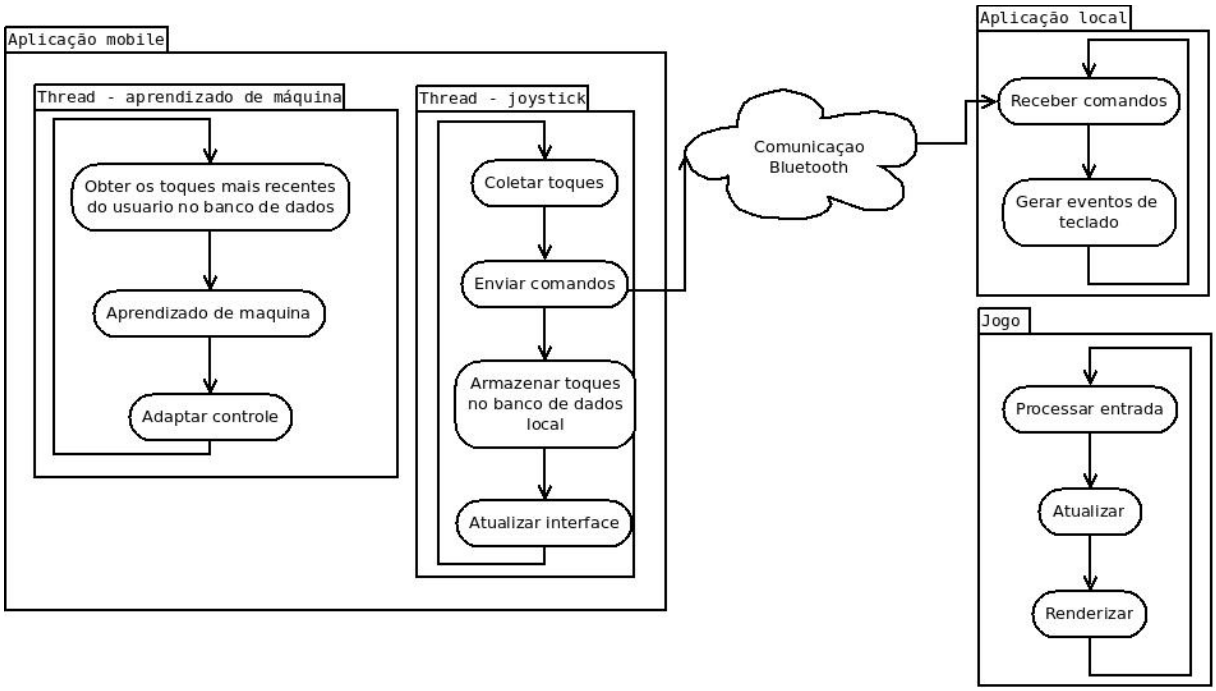

Figura 5: Arquitetura do projeto, indicando a distribuição das atribuições entre as aplicações e sua estrutura de comunicação.

A Figura 5 apresenta a arquitetura final do projeto, dividido em dois aplicativos. O aplicativo móvel tem a responsabilidade de coletar os comandos do usuário, informados através dos toques nos botões virtuais, e enviá-los para o aplicativo desktop via *Bluetooth*. Após isso, cada toque é armazenado em um banco de dados interno do dispositivo para análise pelo algoritmo de aprendizado. Por fim, o aplicativo móvel deve sempre atualizar a

interface, indicando de forma visual os pressionamentos dos botões (através de animações) e alterando a posição e/ou tamanho dos mesmos de acordo com os resultados do aprendizado de máquina.

A escolha de executar o aprendizado de máquina no próprio dispositivo móvel foi motivada pela performance da comunicação *Bluetooth.* Realizar esse processamento no computador desktop iria exigir que todos os dados de toque na tela fossem passados para o PC, que por sua vez deveria transmitir constantemente os resultados do aprendizado (consistindo da posição e tamanho para cada botão). Com o algoritmo em execução no *smartphone* ou *tablet*, os únicos dados passados serão os nomes dos botões pressionados, minimizando a quantidade de tráfego da comunicação e com isso o *input lag* do controle. Concluindo a decisão, os testes iniciais com o algoritmo indicaram que sua execução poderia ser feita no dispositivo móvel com um desempenho adequado.

Mesmo assim, o algoritmo de aprendizado leva mais tempo para ser executado do que pode ser permitido em uma aplicação de tempo real e resposta rápida como um *joystick*. Por isso, o aprendizado de máquina é tratado em uma *thread* separada. Dessa forma a *thread* principal (à direita dentro da aplicação *mobile* na figura) coleta os toques, envia os comandos para o usuário, atualiza a interface e armazena cada toque no banco de dados interno, que servirá como uma das pontes de comunicação entre ambas as *threads*. A *thread* do aprendizado de máquina obtém os pontos mais recentes do banco de dados, executa o algoritmo de aprendizado com os dados e determina as correções. A partir daí, a *thread* do aprendizado notifica a *thread* principal e repassa as correções, que são aplicadas ao atualizar a interface.

É preciso ressaltar que essas operações são paralelas: a *thread* de aprendizado sempre coleta os dados, os processa com o algoritmo e sugere as correções, enquanto a *thread* principal sempre coleta os toques, envia os comandos por *Bluetooth*, armazena-os no banco e atualiza a interface. As duas sempre permanecem em um *loop* contínuo, com a comunicação sendo realizada quando a *thread* principal armazena pontos no banco de dados para serem coletados pela *thread* de aprendizado e quando a *thread* de aprendizado conclui seu algoritmo e notifica a *thread* principal dos resultados para que as correções possam ser aplicadas.

A aplicação desktop é simplificada nessa arquitetura, já que a maior parte das operações complexas foi delegada para a aplicação móvel. O *software* inicialmente se conecta com o primeiro dispositivo pareado que abra um canal de comunicação destinado especificamente à aplicação. A partir daí, o aplicativo desktop se mantém em um *loop*, aguardando o recebimento de uma mensagem do aplicativo móvel com os botões

pressionados e gerando eventos simulados de teclado que realizem ações no jogo atual assim que recebê-los.

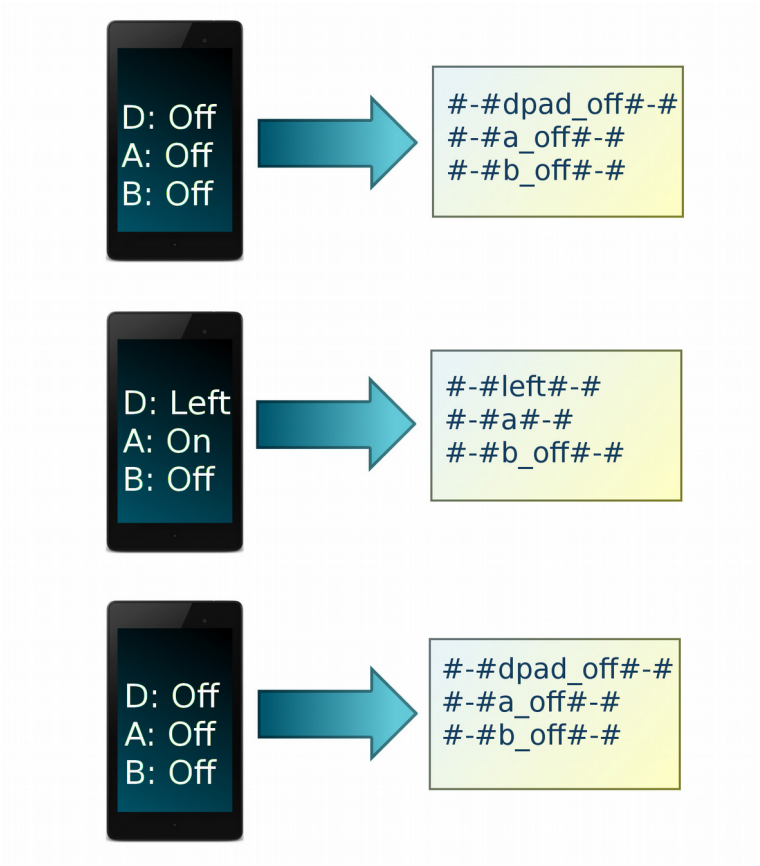

Figura 6: Exemplo das mensagens geradas pelo aplicativo móvel. "D" representa o botão direcional enquanto "A" e "B" indicam os botões de ação de mesmo nome.

A comunicação segue um protocolo simples. Cada mensagem enviada através do *socket* possui o prefixo e o sufixo "#-#", utilizado para determinar o início e o término da mensagem e concluem com uma quebra de linha. O aplicativo móvel sempre envia o estado atual de todos os botões. Os botões pressionados são indicados apenas pelo seu nome, enquanto os botões não-pressionados são indicados pelo nome mais o sufixo "_off". O direcional será uma exceção à regra. Isso ocorre porque o usuário não pode pressionar mais de uma direção ao mesmo tempo. Por isso, o aplicativo desktop irá desativar o pressionamento de uma direção caso outra seja recebida. Quando o usuário parar de pressionar qualquer direção, a mensagem de controle "#-#dpad_off#-#" é enviada pelo *joystick*, avisando o aplicativo desktop que qualquer tecla de direção pressionada deve ser desativada. O uso do sistema de envio constante de mensagens evita que algum fique preso em um estado pela falta

de recebimento de uma mensagem. O aplicativo desktop irá receber constantemente o estado de todos os botões, de forma que caso o usuário pressione ou solte uma tecla e a mensagem que notifica o aplicativo da alteração se perca, o próximo envio irá corrigir a situação.

A Figura 6 mostra um exemplo da comunicação. No primeiro estágio, nenhum botão está pressionado e a mensagem enviada indica que o direciona não possui nenhuma direção pressionada e que ambos os botões de ação não estão pressionados. Já no segundo estágio, o usuário pressionou o botão direcional para a esquerda (*left*) e pressionou o botão A, resultando em uma mensagem com a direção informada e o botão A sem o sufixo "_off". Por fim, no último estágio o usuário não está mais pressionando nenhum dos botões, resultando em uma mensagem igual a inicial que informa o aplicativo desktop que a simulação do pressionamento das teclas do teclado do computador deve ser encerrada.

3.1.2 MODELAGEM E IMPLEMENTAÇÃO

Com a arquitetura do projeto definida, a modelagem das classes foi a etapa seguinte. Uma das preocupações principais era garantir que toda a estrutura do código fosse implementada em uma forma que permitisse que, no futuro, as classes específicas que representam um *joystick* pudessem ser expostas em uma API. O objetivo por trás dessa exigência é permitir que trabalhos posteriores possam permitir que o jogo seja capaz de definir a quantidade, posição e tamanho dos botões através de uma API. Com isso, o *game designer* será capaz de projetar não somente as mecânicas do jogo, mas as mecânicas e configurações do controle em si. Esse *layout* específico do jogo será depois otimizado pelo algoritmo de aprendizado de máquina, corrigindo erros de usabilidade que possam estar presentes no mesmo e adaptando o controle ao usuário. Uma visão mais detalhada dessas possibilidades está na seção de trabalhos futuros. Essa seção irá apresentar a visão da modelagem e a descrição do papel de suas principais classes. Uma descrição ainda mais profunda, tratando dos parâmetro e funcionalidades de cada método nessas classes pode ser encontrada no apêndice A, no final deste documento.

A Figura 7 apresenta a visão da modelagem das classes responsáveis por representar o *joystick*. Essa estrutura é utilizada para manter o estado do controle, com a posição e tamanho dos botões, além do histórico de alterações nos mesmos. A classe *Button* representa o botão do controle virtual, contendo as informações básicas de posição (x, y) e de tamanho (*width* – largura e *height* – altura). Dentro dessa classe também estão representados os valores mínimos para essas variáveis. A largura e a altura do botão são limitadas pelo valor mínimo de, respectivamente, *minWidth* e *minHeight*.

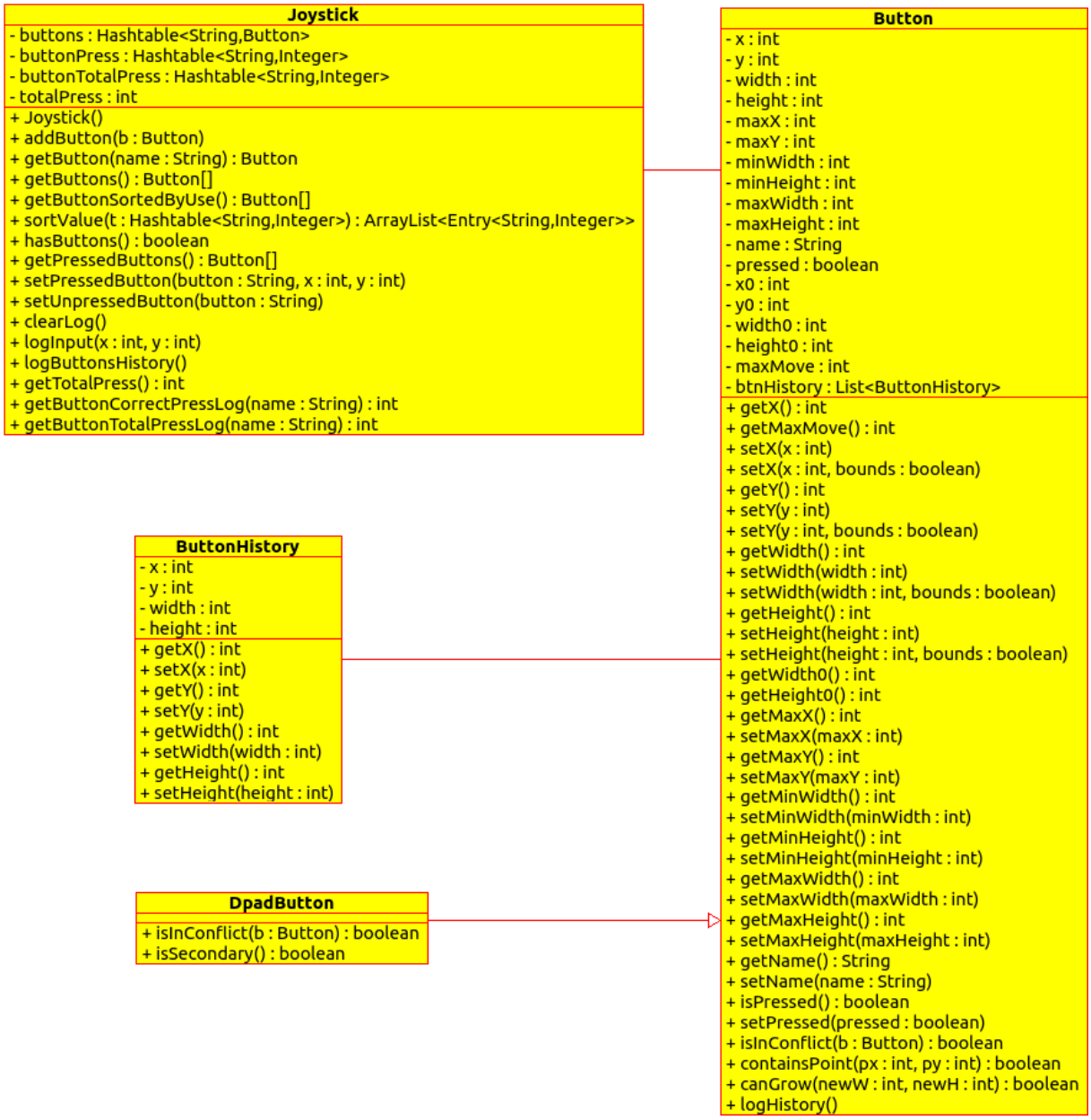

Figura 7: Modelagem das classes responsáveis por representar o joystick e seus botões.

A posição x e y corresponde ao ponto na tela que representa o centro do botão. Os limites para esses valores são calculados utilizando as seguintes fórmulas simples:

- Mínimo de x: $0+\frac{width}{2}$

- Mínimo de y: $0+\frac{height}{2}$

- Máximo de x: $maxX - \frac{width}{2}$

- Máximo de y: $maxY - \frac{height}{2}$

Os valores de x e y correspondem a um espaço cartesiano nas dimensões da tela do dispositivo. A origem desse espaço estará no canto superior esquerdo da tela, com os valores máximos de x e y localizados no canto inferior direito e tendo o valor das duas dimensões da resolução da tela do dispositivo. Por isso, *maxX* e *maxY* serão definidos sempre como as dimensões da resolução do aparelho que executa o controle virtual. Porém, como a posição do botão corresponde ao centro do mesmo, definir a posição de um botão para os pontos mínimo e máximo iriam resultar em um botão que ultrapassaria os limites da tela. Por isso, o controle se certifica de compensar a diferença nas fórmulas, somando ou subtraindo a metade da dimensão adequada do botão. Os métodos *get* e *set* que expõem os atributos de tamanho e posição realizam a validação dos valores passados ao objeto do tipo *Button*, garantindo que nenhum valor possa ultrapassar os limites definidos.

A classe *Button* também irá manter informações sobre o estado de pressionamento do botão, permitindo definir o atributo booleano *pressed* como *true* quando o botão estiver pressionado. A posição e tamanho iniciais do botão também são armazenados, em *x0*, *y0*, *width0* e *height0*. Como regra final de validação, a variável *maxMove* permite que o *joystick* defina uma distância máxima que o botão pode estar afastado da posição inicial. Cada vez que houver uma tentativa de alterar o valor de x e y, *Button* irá validar se a nova posição não excede o limite máximo de movimentação, calculado através da distância cartesiana simples em relação à posição inicial, definida por *x0* e *y0*.

Por fim, a classe ainda armazena o histórico de movimentação completo do botão através de uma lista (*btnHistory*) de objetos do tipo *ButtonHistory*. Cada objeto contém uma informação de posição e tamanho, com a classe *Button* armazenando uma nova entrada sempre que o algoritmo de aprendizado realiza uma iteração (independente da existência de alterações). Esse histórico da sessão é utilizado na geração dos logs para permitir que todas as alterações do botão possam ser rastreadas em uma análise posterior detalhada de uma sessão única de testes. Para armazenar o log, o algoritmo de aprendizado chama o método *logButtonHistory* após cada iteração, criando uma nova entrada de log de posição do botão.

Além dos métodos tradicionais *get* e *set* para cada atributo que encapsulam as propriedades e aplicam as validações necessárias sobre cada alteração de valor, a classe *Button* possui diversos métodos auxiliares, detalhados no apêndice A.

A classe *DpadButton* herda de *Button* e é utilizada para adicionar as peculiaridades necessárias que diferenciam um botão do direcional de um botão de ação tradicional. As principais modificações são a adição de um método extra e a sobrescrita de um método da classe pai, alterando um aspecto chave de seu funcionamento.

A classe *Joystick* irá representar o objeto final que corresponde ao controle virtual e seus botões. O conceito por trás dessa classe é que os objetos do tipo *Button* apenas sejam utilizados ao adicionar, remover ou alterar a lista de botões do controle. Após a inicialização, *Joystick* irá manter as informações completas sobre o estado do botão, como números de pressionamentos e status atual, e irá encapsular todas as operações referentes ao controle.

Analisando a estrutura da modelagem, *Joystick* é a classe visível externamente no *framework* e através dela boa parte das validações e da lógica do controle ocorre. Ao iniciar o aplicativo pela primeira vez, uma nova instância dessa classe é criada a partir dos valores padrão. Após isso, qualquer alteração nessa instância de *Joystick* é persistida nas *SharedPreferences* através da camada de persistência, de forma que em inicializações posteriores do aplicativo o estado atual do controle armazenado nessa classe sempre possa ser recuperado. É necessário notar que essa instância inicial da classe será repassada para todos os objetos que dela necessitarem, garantindo que cada um possua um ponteiro para a mesma instância do objeto e sempre lide com os mesmos dados, sem conflitos.

Além das classes responsáveis por modelar a estrutura lógica de um *joystick*, era necessária uma forma de persistir os dados representados em memória pelas classes apresentadas anteriormente. Além disso, como detalhado em seções anteriores, cada toque na tela deve ser persistido em um banco de dados interno, que servirá de ponte para a comunicação entre a interface (que recebe os toques) e o algoritmo de aprendizado (que precisa processá-los em busca de padrões que sirvam para determinar a disposição ótima do controle). Por fim, a análise dos resultados dos testes de usuários exigia que as informações pertinentes pudessem ser exportadas em um log a partir de todos os dados coletados. Para atender todos esses requisitos, foi criada uma camada de persistência que atenderia cada necessidade com classes específicas, como visto no diagrama de classe da Figura 8.

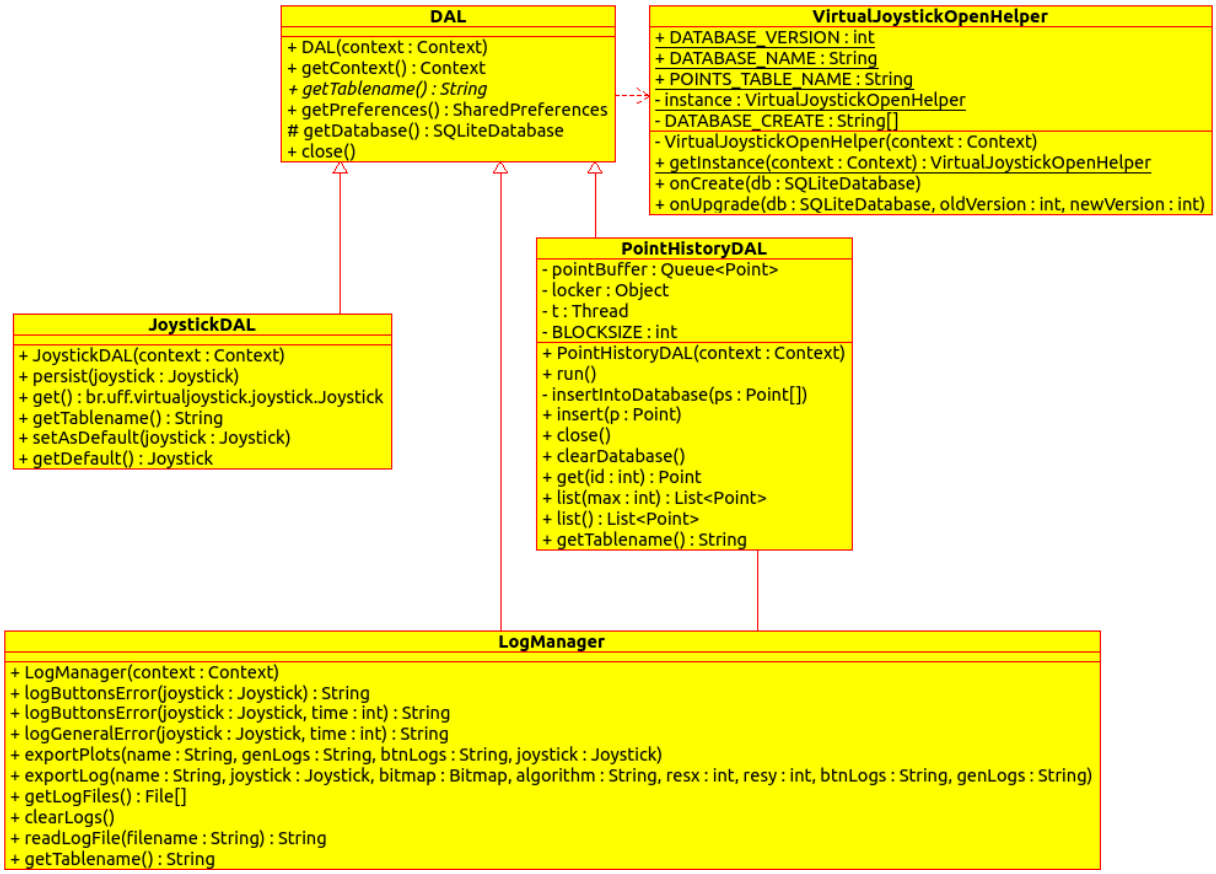

Figura 8: Modelagem da camada de persistência de dados.

A classe mais básica da camada de persistência é diretamente ligada ao acesso ao banco de dados. *VirtualJoystickOpenHelper* é uma classe utilitária, que herda da classe interna do SDK do Android *SQLiteOpenHelper* e é responsável por criar as tabelas do banco de dados no primeiro uso e fornecer um acesso que permita a abertura de conexões com o banco SQLite do aplicativo, criado automaticamente pelo sistema Android e armazenado no espaço interno da memória do dispositivo exclusivo do aplicativo. Os métodos mais importantes são *onCreate* e *onUpgrade,* ambos sobrescrevendo métodos abstratos na classe pai. O primeiro método contém todo o código necessário para criar as tabelas no banco de dados e será executado apenas ao acessar o banco pela primeira vez. O segundo método é responsável por realizar qualquer alteração necessária no banco durante uma atualização de versão do aplicativo que exija uma modificação nas tabelas do banco SQLite.

A classe *DAL* por sua vez, é o centro da camada de persistência e servirá de base para todas as demais classes, estabelecendo os padrões necessários para o funcionamento de qualquer classe de persistência no sistema. O primeiro dos parâmetros definidos será o

construtor, que irá receber um objeto do tipo *Context*. No Android, as aplicações são compostas por atividades, cada uma representando uma ação que o usuário pode realizar e sendo geralmente representadas através de uma tela, janela ou agrupadas em outra atividade. Durante a execução, cada atividade seguirá o ciclo de vida determinado pelo sistema operacional e serão organizadas em uma pilha durante a execução da aplicação. Estando em seu espaço específico na memória, com diversos recursos instanciados, cada atividade (e a própria aplicação como um todo) terão um contexto, que organiza as informações sobre o status atual da atividade ou da aplicação. Muitas vezes, como ao instanciar um componente visual ou acessar o armazenamento interno da aplicação é preciso possuir um objeto do tipo *Context* que permita o acesso a essas informações. O acesso a esse objeto sempre estará garantido em todas as classes que herdem de *DAL*.

Iniciando a lista das classes que herdam de *DAL*, temos *JoystickDAL*. Essa classe é responsável por persistir o estado de um objeto do tipo *Joystick* e, com isso, o layout do controle. O construtor da classe é uma implementação direta do construtor da classe pai.

PointHistoryDAL, assim como *JoystickDAL*, é uma classe filha de *DAL* e será responsável por fornecer o acesso ao histórico de toques do usuário na tela, utilizado para o aprendizado de máquina e as adaptações. Como a classe anterior, ela irá implementar o construtor da classe pai. Além disso, *PointHistoryDAL* implementa a interface *Runnable* de Java, que permite que parte de seu código seja executado em paralelo em uma *thread* separada (o código contido no método *run* implementado da interface). O método *run* irá conter o código que fará a inserção dos pontos no banco de dados.

A motivação para a paralelização dessa tarefa está no sistema de armazenamento de dados dos dispositivos Android. Celulares e tablets utilizam memória *flash* internas para armazenar todos os dados do dispositivo, o que traz uma limitação de performance já que o desempenho de tais memórias é muito inferior a discos rígidos tradicionais de computadores *desktop*. Alguns dispositivos possuem uma entrada para cartões de memória, utilizada para solucionar o problema da falta de espaço, porém com uma velocidade de acesso ainda menor. Em geral, essa performance é mais do que suficiente, na maioria dos casos. Para o controle virtual, era necessário inserir os toques do usuário constantemente no banco de dados. O banco SQLite, ao receber uma inserção, imediatamente requisita ao sistema operacional que sincronize os *buffers* do armazenamento e efetivamente persista o dado inserido na memória *flash* para garantir que não haja perdas de dados. O padrão de inserção dos dados no projeto, consistindo de dados pequenos inseridos diversas vezes por segundo, obriga o sistema Android a sincronizar os *buffers* do armazenamento constantemente, gerando um impacto na

performance do dispositivo e um *lag* perceptível no acesso ao banco e, por consequência, bloqueando o programa ao esperar uma inserção sequencial e constante de dados.

A solução foi paralelizar a inserção. O método *run* permanece em um *loop*, verificando a fila *pointBuffer*, que contém os pontos a serem inseridos. Quando a lista atingir o tamanho *BLOCKSIZE* (definido como a constante 20), cada ponto é removida da fila, com um intervalo de 10 ms, e adicionado a um vetor a ser inserido no banco de dados. Dessa forma, a chamada ao método *insert* feita a partir da *thread* principal do programa apenas adiciona o ponto na fila de inserção. Como *pointBuffer* é um objeto compartilhado entre as duas *threads* do objeto, é necessário definir três regiões críticas (sinalizando com o objeto *locker*): uma ao acessar a fila para inserir um novo ponto, outra ao remover um ponto da lista para inserção e uma região final ao encerrar o acesso ao banco que finaliza a inserção do *buffer* e encerra a *thread*. Com isso, é possível realizar uma inserção pausada em *background* que não gere uma carga grande demais no armazenamento interno com a implementação de um *buffer* de inserção de pontos.

A última classe da camada de persistência é *LogManager*, responsável pelas funcionalidades de exportação de logs em texto e gráficos com os resultados de sessões de teste. Boa parte dos métodos irão analisar os dados salvos sobre o usuário e retornar uma *string* contendo o texto parcial do log de uso, que serão combinados e armazenados no sistema de arquivos.

Por fim, é necessário detalhar o funcionamento da interface do usuário, responsável por coletar os toques e representar o *joystick* para o usuário ao mesmo tempo em que reproduz as adaptações na interface visual, e as classes responsáveis pelo aprendizado de máquina. A Figura 9 mostra o diagrama de classes para essa seção do projeto. Algumas das classes indicadas já foram detalhadas anteriormente ao tratar da modelagem ou da DAL e estão representadas no diagrama para demonstrar os pontos nos quais as camadas se integram.

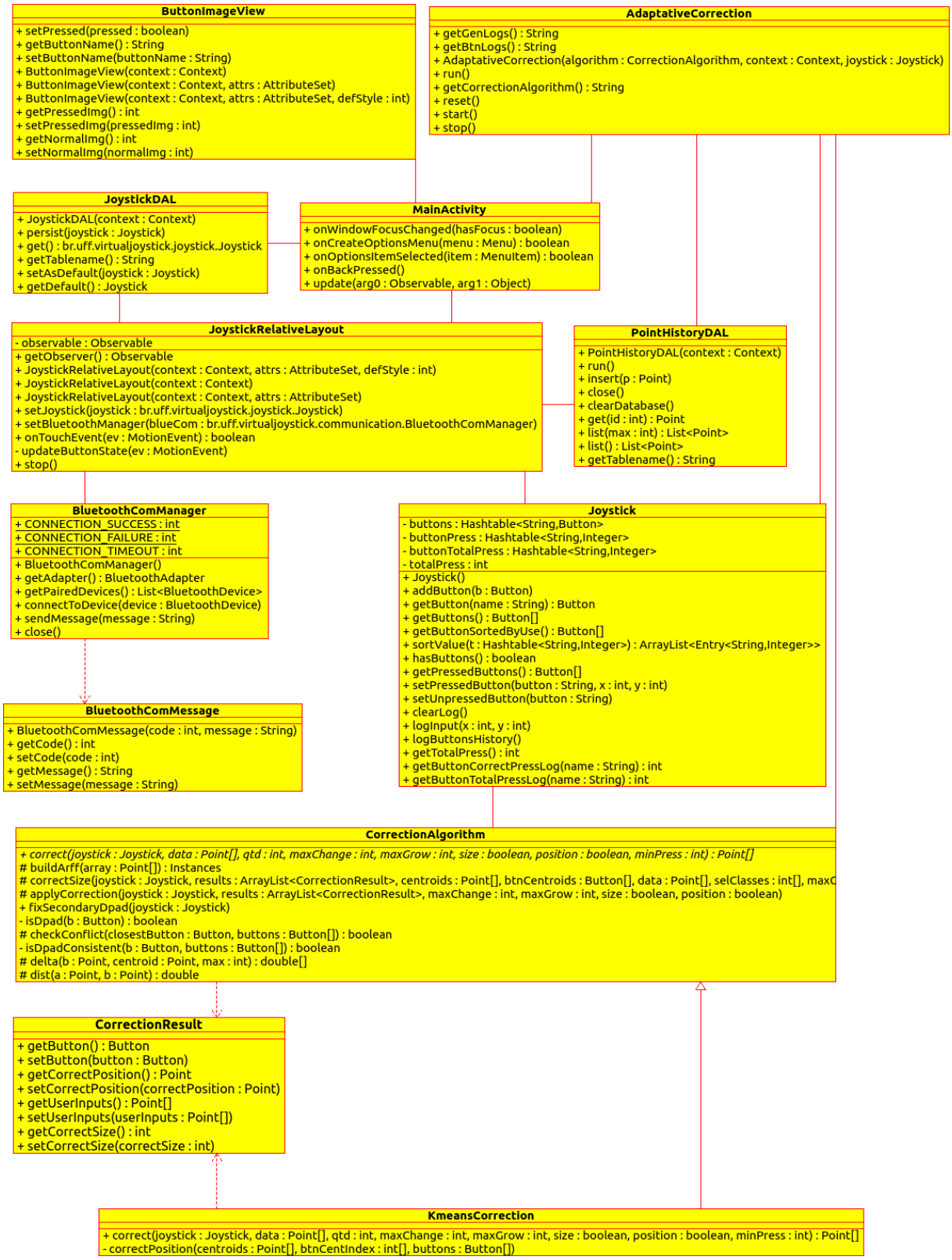

Figura 9: Visão simplificada da modelagem principal do projeto, indicando as classes necessárias para a coleta de entradas, comunicação e adaptação do controle.

MainActivity é a *Activity* principal do controle virtual, sendo responsável por apresentar a interface com o usuário e realizar todas as chamadas às demais classes. Ela possuirá uma instância de *Joystick*, que será repassada para as classes que necessitarem e é instanciada a partir do controle persistido por *JoystickDAL*. A partir da lista de botões no objeto *Joystick*, a classe irá adicionar um *ButtonImageView* na interface para cada botão do controle. *ButtonImageView* herda da classe Android *ImageView,* que representa um controle visual que exibe uma imagem. Cada objeto desse tipo irá representar um botão e armazenar o nome (utilizado para correlacionar os botões com os controles na tela) e poderá ter seu estado atuado entre pressionado e não-pressionado. Duas imagens são utilizadas para representar os dois estados do botão, sendo exibidas de acordo com o estado atual, criando uma interface de uso animada. *MainActivity* implementa a interface *Observer* de Java, que permite que a classe assine um evento a ser disparado por objetos do tipo *Observable*. Proveniente da interface, existe o método *update*, que é chamado automaticamente em cada *Observable* assinado por *MainActivity*, repassando qualquer dado novo que precise ser informado para a tela principal. Nesse projeto, *BluetoothComManager* é um *Observable* que irá notificar a tela principal sobre conexões e desconexões, permitindo que o usuário seja informado para tomar a decisão sobre o que será feito. *AdaptiveCorrection* é outro *Observable* e irá notificar a interface sobre uma nova adaptação determinada pelo aprendizado de máquina. Ao receber o controle adaptado, *MainActivity* irá atualizar o tamanho e posição dos *ButtonImageView* correspondentes na tela. Por fim, temos *JoystickRelativeLayout.*

JoystickRelativeLayout é uma classe vital para o projeto, herdando de *RelativeLayout,* um controle visual nativo do Android. Essa classe será instanciada por *MainActivity* como um controle visual transparente que ocupa a tela inteira, sendo posicionado a frente de todos os botões e irá interceptar todos os toques do usuário e executar as ações necessárias. Além dos construtores padrão herdados de *RelativeLayout*, a classe possui uma instância de um *Observable* que permite que a classe notifique a tela principal que o estado dos botões foi alterado, para que possam ser executadas as animações de pressionamento.

O aprendizado de máquina é realizado por um conjunto de classes. Primeiramente, *AdaptiveCorrection* é a classe que é responsável por paralelizar o aprendizado de máquina e serve como interface para fornecer o acesso ao aprendizado para a classe *MainActivity*. Seu construtor irá receber como parâmetros, além da instância de *Joystick* e um objeto do tipo *Context*, uma instância da classe *CorrectionAlgorithm* que indicará qual algoritmo de aprendizado será utilizado (caso esse parâmetro seja *null*, o controle será não-adaptativo). O aprendizado, por se tratar de uma operação custosa, é executado em uma *thread* separada,

enquanto *AdaptiveCorrection* implementa a interface *Runnable* para executar as tarefas paralelas no método *run*. Por fim, a classe ainda herda de *Observable*, permitindo que ela notifique o *Observer* (classe *MainActivity*) sobre qualquer nova adaptação.

O aprendizado de máquina em si fica a cargo de classes que herdem de *CorrectionAlgorithm*. Essa classe define o método abstrato *correct* que deverá ser implementado em todas as suas subclasses e deverá realizar a adaptação. O código para a correção de tamanho dos botões, que será igual para qualquer algoritmo, também está nessa classe. As correções são retornadas como objetos do tipo *CorrectionResult,* que contém novas informações de posição e tamanho, bem como o botão que deve ser corrigido.

A adaptação utilizada nesse trabalho é baseada no algoritmo K-means. Sua implementação está na classe *KmeansCorrection*, que herda de *CorrectionAlgorithm* e possui acesso a todos os métodos da classe pai, além de possuir seus métodos próprios.

3.2 ADAPTAÇÕES AO USUÁRIO E AO JOGO

Para atingir o objetivo de criar um controle adaptado ao estilo e às necessidades de cada jogador, serão necessárias diversas adaptações possíveis para a interface. No presente trabalho, foram utilizadas duas adaptações diferentes para os botões: tamanho e posição, com ambas sendo executadas simultaneamente para todos os botões. A adaptação de tamanho busca facilitar o uso dos botões mais importantes para o jogo atual, aumentando o tamanho dos mais utilizados.

O segundo tipo de adaptação é a posição dos botões, que busca determinar a posição ótima para o botão, que forneceria a melhor taxa de acerto de toques possível. Para realizar essa adaptação, são utilizados dados sobre os toques dos usuários na tela, corretos ou incorretos. Um toque correto é aquele que atinge a área interna de um botão e, portanto, realiza uma ação. Por sua vez, um toque incorreto é definido como aquele que não coincide com a área de nenhum botão e, com isso, não executa nenhuma ação. A partir desses toques, o objetivo do controle adaptativo é tentar encontrar os pontos que representam melhor o centroide de todos os toques do usuário, corretos ou não, destinados a um botão e mover o mesmo para essa localização seguindo limites de deslocamento e velocidade para garantir uma transição suave, como indicado na Figura 10. Os pontos vermelhos representam os toques do jogador na tela e os pontos azuis correspondem ao centroide para o *cluster* associado ao botão 1, representando sua posição ideal. No lado esquerdo da figura, o controle está em sua configuração padrão. O algoritmo irá mover o botão 1 gradativamente até que atinja a posição demonstrada no lado direito da figura. É necessário notar que alguns botões podem não ter um

centroide associado no momento, como em situações em que o botão ainda não foi utilizado, e nesse caso não precisarão de nenhuma alteração no momento.

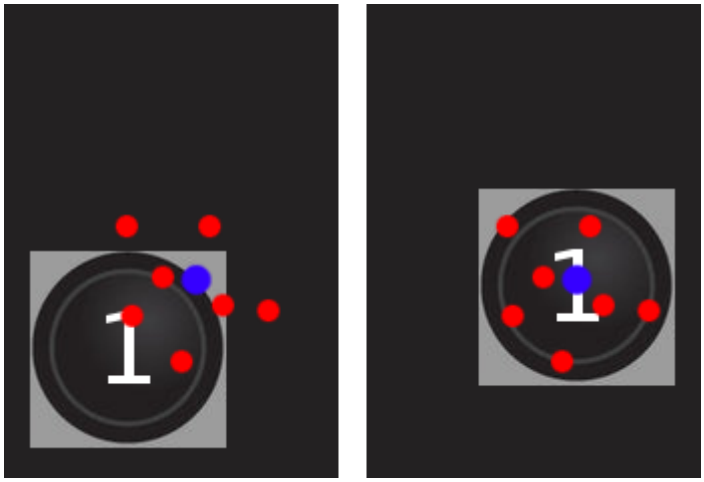

Figura 10: Demonstração do processo de adaptação.

Para garantir a consistência da interface, diversas regras e limites foram definidos. Para cada modificação de tamanho, foi determinado que um botão só poderia crescer até no máximo o dobro de seu tamanho original. O tamanho mínimo definido é o mesmo tamanho original do botão. Com essas regras, são evitados botões grandes ou pequenos de mais para serem utilizados. Apesar dessas regras mais simples, ainda é preciso lidar com os outros botões nas áreas adjacentes. Se um botão se mover ou crescer de forma a conflitar com a área de um vizinho, o usuário não conseguiria operar a interface. Para evitar essa situação, antes de cada mudança em posição ou tamanho, o controle verifica se a nova posição e/ou tamanho farão o botão invadir a área de algum vizinho (ou seja, uma interseção entre a área de ambos). Se essa alteração gerar um conflito, ela não será executada. Isso significa que se o centro dos toques de um botão estiver próximo demais de outro, de uma forma que mover o botão para aquela posição geraria um conflito, o *joystick* irá apenas mover o botão o mais próximo o possível da posição correta sem gerar conflitos. Caso, por alguma outra adaptação, o botão vizinho seja movido para fora do caminho, a adaptação irá continuar o movimento já que o conflito não existe mais. Com essas regras, a interface do controle sempre é mantida consistente.

Um conjunto separado de regras mais restritas foi necessário para o direcional do controle. O botão direcional, presente em todos os *joysticks* modernos, é composto por 4 direções principais (cima, baixo, esquerda e direita) e todas as diagonais entre direções adjacentes (como cima e esquerda ou baixo e direita, por exemplo). Em controles físicos, as

diagonais podem ser áreas específicas do direcional ou simplesmente serem realizadas pressionando simultaneamente dois botões de direção. Algumas características dos direcionais precisam ser reproduzidas para garantir o uso efetivo da interface. Primeiramente, a zona morta, uma área no meio dos 4 botões de direção que não realiza ação nenhuma. Essa região neutra facilita o uso de sequências de botões, comuns em jogos de luta, como o pressionamento rápido e consecutivo da direita e esquerda, sem que haja o pressionamento acidental de uma direção indesejada, já que o usuário apenas desliza o dedo entre as direções para realizar o movimento. Outra característica é a ausência de espaços vazios entre os botões. Esse aspecto é importante porque o uso típico do direcional envolve arrastar o dedo entre os botões de direção e o comportamento esperado é que sempre haja uma direção pressionada, exceto na zona morta. O formato do direcional é outro aspecto que precisa ser consistente, seguindo a disposição padrão em cruz. O usuário espera que o botão da esquerda esteja na direção oposta do botão direita, de forma que a adaptação precisa ser mais restrita.

Como o direcional do controle adaptativo é composto por 8 botões, uma regra específica foi criada para garantir sua consistência. Os 4 botões principais passam pelo processo normal de adaptação seguindo os resultados do algoritmo de aprendizado. Porém, uma regra extra de consistência é aplicada para garantir que as direções principais se mantenham o mais próximo o possível do formato de cruz. O direcional do controle é dividido em 9 quadrantes, cada um correspondendo ao espaço exato dos botões do direcional em seu tamanho original e sendo ocupado por um único botão. Com essa divisão, cada botão irá ocupar uma coluna e uma linha: por exemplo, o botão da esquerda irá ocupar a primeira coluna e a segunda linha, enquanto o botão para baixo se encontrará na segunda coluna e terceira linha. A partir daí, a regra de consistência determina que nenhum dos 4 botões principais pode ultrapassar as colunas e linhas adjacentes por um espaço maior que 1/3 da área do botão. A validação dessa regra é feita antes de implementar qualquer alteração sugerida pela adaptação para o direcional, negando qualquer alteração que viole a validação.

As 4 diagonais do direcional seguem um conjunto de regras separado. Em primeiro lugar, elas não serão afetadas pela adaptação e serão alteradas de acordo com as mudanças nos botões principais do direcional. Como analisado anteriormente, um dos comportamentos esperados do direcional é que qualquer área nele, com exceção da zona morta no quadrante central, aponte uma direção ao ser pressionada. Dessa forma, não podem existir espaços vagos entre os 8 botões que o compõem. Por isso, as diagonais passam a ser incumbidas da tarefa de preencher o espaço vago. Cada mudança sugerida pela adaptação poderá alterar o tamanho e/ou posição dos 4 botões de direção principal. Com as modificações nesses 4 botões, o

controle irá reposicionar e redimensionar os botões das diagonais de modo que possam preencher o espaço entre os botões principais. Para isso, a regra seguida será alterar a posição, altura e largura de cada botão diagonal até que seus limites coincidam com os limites dos botões principais vizinhos ou com os limites externos do direcional (definido como um retângulo traçado ao redor dos 4 botões principais que possa conter os mesmo com tamanho mínimo). Com isso, estará garantido que o direcional não terá espaços vagos entre os botões, certificando que o usuário terá uma experiência consistente.

3.3 APRENDIZADO DE MÁQUINA

Com as abordagens das duas adaptações definidas, era necessário definir o método de coleta de informações de entrada do usuário e como traduzi-las para informações significativas que pudessem melhorar a resposta do controle às necessidades do usuário. Os dados utilizados para avaliar o quão correto a configuração do controle se encontra vem de um banco de dados contendo todos os toques na tela executados pelo usuário. A tela é mapeada em coordenadas Cartesianas, com a origem localizada no canto superior esquerdo seguindo a convenção do sistema Android, e os toques são armazenados em um banco SQLite no próprio dispositivo móvel. O algoritmo de aprendizado irá utilizar os pontos mais recentes, seguindo os critérios detalhados no capítulo 4. As duas operações ocorrem simultaneamente, com o controle constantemente inserindo novos pontos no banco de dados e o algoritmo processando os mais recentes a cada iteração.

Baseado nos dados coletados, o próximo passo é executar as adaptações propriamente ditas. A heurística para a adaptação de tamanho irá aumentar os botões mais utilizados, respeitando os limites, seguindo um algoritmo simples que registra e conta a quantidade de toques em um botão (incluindo os toques corretos no botão e os toques incorretos cujo botão mais próximo seja esse) e armazena os botões em um vetor ordenado pela quantidade de toques. Esse vetor é dividido em 3 partes, com tamanhos iguais. A primeira parte irá conter os botões mais utilizados, que irão ter seu tamanho aumentado gradativamente até o limite máximo. A segunda parte, contendo botões menos utilizados que os primeiros, possuirá os botões que não terão seu tamanho alterado (independentemente de estarem no tamanho original ou já terem sofrido alguma modificação). A parte final contém os botões menos utilizados. Qualquer botão nessa última parte da lista deverá ter o tamanho original, sendo reduzido caso necessário de forma gradual até atingir o tamanho mínimo. A cada iteração do aprendizado, essa lista é atualizada e reordenada, modificando o comportamento da adaptação. Como um exemplo, podemos considerar o botão A, que em uma iteração pode se

encontrar entre os mais utilizados e passar a ter seu tamanho aumentado. Agora, suponhamos que algumas iterações mais tarde, ele passe a ocupar a lista do meio, o que irá parar seu crescimento imediatamente. Caso após mais iterações ele passe a ocupar o último grupo, A irá diminuir até seu tamanho original.

Já a heurística para as adaptações de posição exige uma abordagem mais sofisticada ao tentar encontrar a posição ideal para cada botão na tela seguindo as necessidades específicas de um jogador. Para isso, é preciso encontrar coordenadas na tela que representem o centro de um conjunto de toques destinado a um botão, o que foi feito utilizando uma variação do algoritmo K-means, detalhado na próxima seção.

3.3.1 K-MEANS

Para determinar a posição ótima dos botões, foi utilizado o algoritmo de aprendizado de máquina não-supervisionado K-means, um método de quantização vetorial para *data mining*. O objetivo desse classificador é, a partir de um conjunto de pontos de entrada, separá-los em K classes de entradas relacionadas. A escolha desse algoritmo foi motivada pela sua característica de ser capaz de separar um conjunto de toques na tela em diversas classes que corresponderiam aos botões na tela, criando uma associação entre cada toque do usuário e o seu botão de destino. Além disso, o K-means resulta em um centroide para cada classe, representando um ponto ótimo para a posição do botão que permitiria o aumento da taxa de sucesso. Com essas características, o K-means se mostrou bastante adaptado às características particulares do problema abordado e foi uma escolha natural para a solução do problema.

Apesar de ser um problema NP-difícil, diversas heurísticas permitem soluções mais simples que convergem rapidamente para um ótimo local. Dado $x = \{x_1, \ldots, x_m\}$, o objetivo do classificador é particionar x em k *clusters* com valores semelhantes (SMOLA e VISHWANATHAN, 2008). Para atingir esse objetivo, o algoritmo define os vetores protótipo $\mu_1, \ldots, \mu_k$ e um vetor indicador r_{ij}, que é igual a 1 se e somente se x_i for associado ao cluster j. Ao clusterizar o conjunto de dados, a medida de distorção e a distância de cada ponto para o vetor protótipo serão minimizadas:

$$J(r,\mu)=\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{k} r_{ij}\|x_i-\mu_j\|^2,$$

aonde $r = \{ r_{ij} \}$, $\mu = \mu_j$ e $\|\cdot\|^2$ indica a distância Euclidiana. Para obter r e μ, uma estratégia de dois estágios é utilizada. Primeiro, μ é fixado e o objetivo é determinar r. A solução para o i-ésimo ponto x_i pode ser encontrada definindo:

$$r_{ij} = 1 \; if \;\; j = argmin \| x_i - \mu_{j'} \|^2 ,$$

e 0 caso contrário. No segundo passo, r é fixado e μ será determinado. Como os valores r são fixos, J será uma função quadrática de μ que pode ser minimizada definindo a derivada em relação a μ_j como 0:

$$\sum_{i=1}^{m} r_{ij} (x_i - \mu_j) = 0 \; for \, all \; j$$

Que pode ser rearranjado como:

$$\mu_j = \frac{\sum_i r_{ij} x_i}{\sum_i r_{ij}} .$$

Como $\sum_i r_{ij}$ conta o número de pontos associados ao *cluster* j, o algoritmo define μ_j como a média amostral dos pontos associados ao *cluster* j. O algoritmo executa diversas iterações até parar quando as associações com os *clusters* não variem mais de forma significativa.

O conjunto de entrada no caso deste trabalho são os pontos mais recentes aonde o usuário tocou a tela. O objetivo é separá-los em k classes, cada uma contendo um centroide, que é a informação mais pertinente para nossa implementação. Originalmente, k seria o número de botões virtuais na tela. Porém, durante os testes iniciais, ficou claro que em alguns jogos nem todos os botões seriam usados e diversos dos k centroides na verdade terminavam agrupados em uma única área, mostrando que poderiam na verdade ser representados por uma única classe. Como um efeito indesejado, várias vezes um único botão possuiria diversos centroides associados a ele, criando complicações para o algoritmo de adaptação ao tentar determinar os pares centroide-botão. Para atingir um resultado mais preciso nesses pareamentos, foi criada uma variação da abordagem inicial do K-means, resultando abordagem do K-means dinâmico, aonde o número de classes k é o número de botões virtuais que foram pressionados pelo menos uma vez, variando dinamicamente o valor de k para melhor atender os requisitos do trabalho. Com essa versão do algoritmo, o número excessivo de centroides foi evitado nos casos aonde o jogo não faz uso de todos os botões e evitou o acúmulo de centroides em um único botão. Como um exemplo de seu funcionamento, se um usuário estiver jogando um jogo de tiro onde ele apenas move uma pequena espaçonave da esquerda para a direita e atira com suas armas utilizando o botão 1, o algoritmo K-means irá

classificar os toques do usuário em 3 classes. Porém, caso o usuário receba um *power-up* especial e o utilize com um pressionamento do botão 2, a próxima iteração do K-means será executada para 4 classes. É interessante ressaltar que a abordagem anterior iria classificar os toques sempre em 10 classes (4 direções principais, 4 direções diagonais e 2 botões de ação), mesmo se a maioria nunca for utilizada.

O algoritmo K-means irá retornar diversos subconjuntos do conjunto completo de toques na tela contendo os pontos relacionados, agrupando os mais próximos na mesma classe. Para cada uma dessas classes, o *clustering* K-means também irá detectar o centroide da classe, que será o resultado mais pertinente nesse caso. Os toques do usuário estarão localizados na área do botão ou em suas proximidades, tendo em vista que o usuário sempre tenta atingir a posição correta do botão para realizar uma ação no jogo. Após diversas interações, haverá um padrão de toques próximos de cada botão, permitindo que o K-means separe as entradas em classes que representem a área do botão e seu entorno. Após encontrar os centroides, cada um será pareado com o botão mais próximo. Em casos aonde haja dois centroides cujo botão mais próximo seja o mesmo, o par correto será aquele com a menor distância, deixando o outro centroide sem nenhum botão associado. O centroide pareado com um botão será considerado como a posição ótima daquele botão, já que representa o ponto médio de todos os toques direcionados ao mesmo. Com esses dados, o controle irá mover gradativamente o botão em direção ao centroide de sua classe, até que o centro de sua área coincida exatamente com o local do centroide.

CAPÍTULO 4 – TESTES DE USABILIDADE

4.1 METODOLOGIA

Para validar o comportamento das adaptações propostas durante a interação com o usuário final e sua experiência de *gameplay*, foi realizado uma série de testes de usabilidade, observando a eficiência das adaptações e a satisfação do usuário. Esse processo foi divido em duas etapas: testes para averiguar a melhor configuração dos parâmetros de adaptação, e os testes finais com usuários. Após isso, os resultados foram analisados através de testes de hipóteses para comprovar a significância ou não dos resultados obtidos.

4.1.1 PARTICIPANTES E EQUIPAMENTOS

O processo de avaliação utilizou dois controles diferentes: um adaptativo e um não-adaptativo, ambos com a mesma funcionalidade e *layout* (como visto na Figura 11). Os usuários não foram informados que o controle teria a capacidade de se adaptar e realizaram o teste apenas sabendo que testariam dois protótipos de controle. Cada jogador usou ambos os controles em dois jogos de gêneros diferentes: Super Mario Bros do NES, um jogo de plataforma, e Sonic Wings do SNES, um *shooter* 2D clássico, em 4 sessões de avaliação.

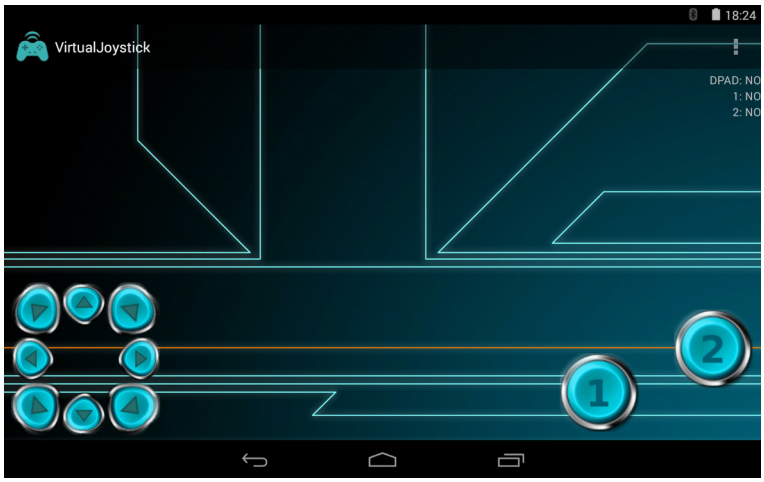

Figura 11: Layout padrão do controle.

O grupo de voluntários foi composto por 16 usuários, 9 homens e 7 mulheres, com idades entre 18 até mais de 60 anos. O grupo foi selecionado e separado em dois subgrupos: 8 jogadores experientes e 8 jogadores inexperientes. O primeiro grupo era composto basicamente por usuários que jogam em consoles ou computadores regularmente, estão

acostumados com *joysticks* tradicionais e jogam *videogames* a mais de 10 anos. O segundo grupo foi composto por jogadores com pouca experiência com jogos e consumidores de jogos casuais ou simples.

O equipamento utilizado foi um *tablet* Nexus 7 (modelo 2012) com o sistema operacional Android 4.4.4. A tela capacitiva tem 7 polegadas com aspecto 16:9 e resolução 1280 x 800. A Figura 12 mostra o equipamento utilizado no teste, com o controle adaptativo em execução controlando um jogo no computador.

Figura 12: Os equipamentos utilizados para os testes.

4.2 CONFIGURAÇÃO DOS PARÂMETROS DE ADAPTAÇÃO

A versão adaptativa do controle deve ser configurada com dois parâmetros principais, que irão interferir no processo final de adaptação. O primeiro parâmetro é o limite de deslocamento por iteração, que determina quantos pixels um botão pode se deslocar a cada execução completa do algoritmo de correção. Um valor mais elevado irá permitir que o controle se adapte imediatamente a qualquer solicitação do algoritmo de aprendizado, enquanto um valor menor irá tornar a adaptação mais lenta e conservadora, evitando que o controle siga tendências passageiras da adaptação.

O segundo parâmetro determina quantos dos pontos mais recentes no banco de dados serão avaliados pelo algoritmo de aprendizado por iteração. Se mais pontos forem utilizados, o algoritmo irá gerar uma adaptação mais conservadora, com menos tendências de seguir mudanças recentes e focada em buscar um resultado que atenda melhor a interação inteira do usuário com o controle. Com menos pontos, o controle irá buscar uma adaptação mais

adequada ao contexto atual do jogo, que mudará rapidamente para seguir qualquer mudança no *gameplay*.

Durante a avaliação dos parâmetros adequados para os testes com os usuários, vários testes piloto foram executados com diversos jogos em busca dos parâmetros ideais e dos jogos mais adequados para as sessões de teste finais. Para isso, as seguintes configurações foram exploradas, buscando atingir a maior variação possível entre adaptações rápidas e conservadoras:

- 5 pixels de deslocamento máximo por iteração, utilizando os 100 últimos pontos: essa configuração usa uma grande quantidade de toques, resultando em uma adaptação baseada na sessão inteira do jogo ao invés dos acontecimentos recentes e move os botões lentamente, dando tempo ao algoritmo de determinar a solução ótima.
- 10 pixels de deslocamento máximo por iteração, utilizando os 30 últimos pontos: a opção que representa uma configuração intermediária entre a anterior e a configuração a seguir.
- 100 pixels de deslocamento máximo por iteração, utilizando os 10 últimos pontos: com esses parâmetros, a adaptação irá analisar somente os últimos toque e configurar o controle para atender o requisito do jogo nos últimos instantes, ao invés de buscar uma solução ótima para toda a sessão de jogo. Com o deslocamento mais rápido do botão, as alterações serão imediatas.

Os resultados dos testes podem ser vistos a seguir na Tabela 1. A adaptação mais rápida permitiu ao controle uma resposta rápida às modificações no jogo e no comportamento do usuário, como situações em que o jogador altera a posição das mãos no controle, atingindo uma melhor performance medida através da taxa de sucesso (porcentagem de toques na tela que atingiram um botão).

Durante os testes, diversas opções de jogos foram avaliadas, de diversos gêneros. Sonic (Master System) foi a escolha inicial de jogo de plataforma até ser abandonado devido a uma interação no início da primeira fase que se mostrou complexa para os usuários com menos experiência. Ainda no gênero de plataforma, Robocop vs Terminator (Master System) foi uma opção interessante, por conter elementos de plataforma combinados com a capacidade de atirar nos inimigos. Contudo, seu elevado nível de dificuldade, se mostrou um problema. A dificuldade também foi o fator que afastou o uso do jogo LifeForce (NES), do gênero de tiro 2D. Porém, sua avaliação mostrou que o gênero era bem recebido por usuários de todas as experiências, o que levou a adoção de um jogo do mesmo tipo, mas com um nível mais

adequado de dificuldade: Sonic Wings (SNES). O último jogo descartado foi Pac-man (Atari 2600). Apesar da dificuldade adequada e da boa recepção, o jogo fazia uso apenas do direcional, um fator que impedia a realização de testes mais completos e conclusivos com um maior número de botões. Com isso, houve o retorno ao gênero de plataforma com o jogo mais tradicional do tipo, Super Mario Bros (NES), que se mostrou adequado para os usuários nas sessões de teste.

Uma observação interessante foi que, em alguns casos, o jogo pode alterar seu *gameplay* de forma significativa sem aviso prévio. O jogo Sonic, do Master System, é um jogo tradicional de plataforma, aonde o usuário deve mover seu personagem com o direcional e pular com qualquer um dos botões de ação, coletando anéis, que correspondem a pontos, e transpondo obstáculos até o fim da fase. Porém, caso o usuário colete mais de 50 anéis em uma fase, o jogo irá passar para uma fase bônus que simula um jogo de *pinball*, com o personagem Sonic representando a bola. Dessa vez, o botão de pulo não é mais utilizado e o direcional é mais exigido. O estágio bônus é rápido, durando entre 30 e 50 segundos antes do jogo retornar ao seu modo normal de plataforma. Enquanto a adaptação mais conservativa não se alterou durante o estágio, a versão mais rápida se corrigiu rapidamente para atender aos novos requisitos. Em uma avaliação subjetiva, o controle com a adaptação mais rápida foi mais confortável de utilizar e respondeu melhor e mais rápido as necessidades de uso.

Tabela 1: Resultados dos testes iniciais

Configuração	Taxa de sucesso	
	Sonic	Lifeforce
5 pixels, 100 pontos	97.56	96.62
10 pixels, 30 pontos	97.86	97.63
100 pixels, 10 pontos	98.58	97.71

Com os parâmetros definidos a configuração mais rápida (100 pixels, 10 pontos) foi selecionada para os testes. Cada sessão de avaliação foi limitada em 5 minutos de jogo com um tempo extra de 1 minuto no início para que o usuário pudesse aprender os controles do jogo. Durante esse minuto, os usuários foram instruídos sobre o objetivo do jogo e os controles, sendo que não receberiam mais nenhum auxílio após a introdução inicial. O processo de avaliação foi composto por uma avaliação subjetiva, através de um questionário

no fim dos testes, e uma avaliação objetiva a partir dos dados capturados pelos logs de teste no dispositivo Android.

Todos os eventos ocasionados pela interação usuário-controle foram registrados em logs de texto nos dispositivos, um para cada sessão de 5 minutos. Esse arquivo contém as coordenadas dos toques do usuário, a taxa de sucesso do controle para cada instante de tempo e botão e a taxa final de sucesso para o controle todo e para cada botão. A taxa de sucesso é um valor real entre 0 e 100 representando a porcentagem de toques que atingiram um botão. Já a taxa de sucesso por botão é um cálculo similar, porém só considerando os toques cujo botão mais próximo for aquele para o qual queremos calcular o valor.

Cada usuário testou ambos os controles com ambos os jogos, totalizando 4 sessões. A ordem de teste (tipo de controle e jogos) foi variada para eliminar a influência do aprendizado do usuário em relação às regras do jogo sobre sua performance.

Para verificar como a adaptação está melhorando a experiência do usuário com o controle e se o padrão dessa adaptação é influenciado pelo gênero do jogo e perfil dos usuários, as seguintes hipóteses foram criadas:

- H1: O controle adaptativo irá aumentar a taxa de sucesso dos usuários inexperientes.
- H2: O controle adaptativo irá aumentar a taxa de sucesso dos usuários experientes.
- H3: O controle adaptativo irá aumentar a taxa de sucesso dos usuários independente de sua experiência.
- H4: O controle adaptativo irá aumentar a taxa de sucesso dos usuários independente do gênero do jogo.

CAPÍTULO 5 – RESULTADOS

5.1 ANÁLISE OBJETIVA

Para avaliar a performance dos usuários, o critério selecionado foi a taxa de sucesso final do controle adaptativo confrontada com a taxa de sucesso final do controle não-adaptativo, comparadas através do teste de Wilcoxon. A taxa de sucesso foi definida como a taxa como que o usuário acertou o botão virtual desejado do controle.

O teste de Wilcoxon é um teste de hipótese estatístico não-paramétrico usado para comparar duas amostras relacionadas, amostras pareadas ou medições repetidas de uma única amostra para verificar se a média das duas populações difere de forma significativa. É frequentemente utilizado como uma alternativa ao teste t de Student em casos aonde não seja possível assumir uma distribuição normal da população (LOWRY, 1998).

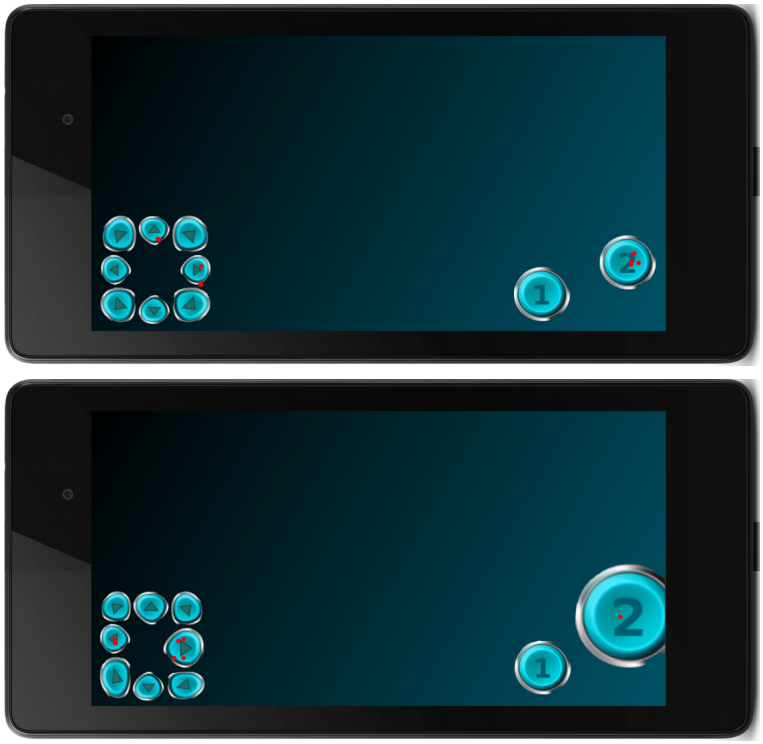

Figura 13: As configurações iniciais e finais para um usuário.

Para nossos testes, o nível de significância foi definido como 0,05 e a hipótese *two-tailed* foi utilizada. O uso da hipótese *two-tailed* garante que será testado se o controle adaptativo é significativamente melhor ou significativamente pior que sua contraparte não-adaptativa simultaneamente, enquanto um teste com a hipótese *one-tailed* apenas avaliaria

uma dessas hipóteses, ignorando a possibilidade da existência do outro cenário e gerando um resultado mais genérico (FISHER, 1936).

A Tabela 2 apresenta os resultados para os 8 usuários inexperientes com ambos os controles, enquanto a Tabela 3 mostra os resultados dos usuários experientes. Em todos os testes, foram utilizadas 12 casas decimais de precisão, com os resultados arredondados para as tabelas a seguir para uma visualização simplificada. Para exemplificar as alterações que o controle sofreu durante o teste, a Figura 13 demonstra a diferença entre o controle inicial e o final, após as adaptações, para um dado usuário.

Tabela 2: Taxa de sucesso para os usuários inexperientes. Todos os valores estão arredondados para 2 casas decimais

Usuário	Super Mario Bros.		Sonic Wings	
	Não-adaptativo	Adaptativo	Não-adaptativo	Adaptativo
1	96.64	99.12	84.18	91.06
2	100.00	96.69	77.11	92.85
3	85.85	95.20	88.94	88.72
4	92.21	98.64	85.28	89.77
5	85.81	97.47	65.26	76.24
6	95.88	97.67	78.55	86.25
7	90.66	98.57	89.70	99.00
8	93.57	98.55	88.55	92.01

Tabela 3: Taxa de sucesso para os usuários experientes. Todos os valores estão arredondados para 2 casas decimais

Usuário	Super Mario Bros.		Sonic Wings	
	Não-adaptativo	Adaptativo	Não-adaptativo	Adaptativo
1	81.17	90.41	81.42	83.79
2	98.99	98.69	79.21	87.32
3	81.70	92.00	74.82	84.53
4	88.98	93.42	66.99	84.40
5	92.66	97.26	83.41	88.88
6	96.27	97.84	92.07	96.45
7	89.26	95.08	92.60	95.74
8	94.80	98.68	94.27	96.60

O primeiro caso de estudo é o jogo Super Mario Bros para o grupo dos usuários inexperientes, comparando as duas versões do controle. Com os parâmetros detalhadas anteriormente, o *z-value* foi -2,1004, mas como o tamanho do grupo (n = 8) é pequeno demais, utilizamos o *w-value*, que equivale a 3 nesse caso. Como o W crítico para 0,05 e tamanho 8 é 4 e com um *p-value* de 0,035692, podemos rejeitar a hipótese nula para esse estudo e garantir que a diferença é significante. Para o jogo Sonic Wings, o *z-value* é -2,380476, mas novamente o tamanho do grupo exige o uso do *w-value*, sendo 1 nesse caso. O W crítico é 4 e o *p-value* é 0,017290, nos permitindo rejeitar a hipótese nula e confirmar que a diferença entre os dois controles também foi significativa. A Figura 14 mostra a diferença entre a precisão dos usuários inexperientes entre ambos os controles para os dois jogos, mostrando como o controle adaptativo sempre fornece uma performance mais alta. Com isso, a hipótese H1 está comprovada e o controle melhora de forma significativa a taxa de sucesso dos usuários inexperientes.

No segundo caso de estudo, iniciamos a análise dos usuários experientes com o jogo Super Mario Bros, comparando os dois controles. O *z-value* obtido é -2,3805 com *p-value* de 0,017290. Devido ao tamanho reduzido do grupo, calculamos o *w-value* que resulta em 1, com o W crítico sendo 4, provando que a diferença entre os controles é significativa. Com o jogo Sonic Wings, o *z-value* obtido foi-2,52 e o *p-value* foi 0,0. De forma similar, ambos podem ser descartados e o *w-value* assume seu lugar, sendo, nesse caso, 0. Como esse valor é

inferior a 4, o valor crítico de W para o tamanho do grupo, podemos considerar a diferença significativa. A Figura 15 mostra a diferença entre a precisão dos usuários experientes entre ambos os controles para os dois jogos, mostrando como o controle adaptativo sempre fornece uma performance mais alta também nesse caso. Com isso, a hipótese H2 está comprovada e o controle melhora de forma significativa a taxa de sucesso dos usuários experientes.

Contudo, durante nossa avaliação dos arquivos de log, foi notado que as taxas de sucesso de usuários experientes eram similares, resultando no levantamento da hipótese H3, indicando que, talvez, a experiência dos usuários não fosse um fator importante para o controle adaptativo. Para verificar esse caso, utilizamos o teste de Wilcoxon para comparar os resultados de ambos os grupos com os mesmos controles. Primeiramente, comparamos o jogo Super Mario Bros com a versão não-adaptativa do controle entre os dois grupos, obtendo o *z-value* de -0,98 e o *w-value* de 11, que é o valor a ser utilizado devido ao tamanho do grupo. Como o valor crítico é 4, menor que o *w-value* encontrado, podemos concluir que a diferença entre os grupos não foi significativa para $p <= 0,05$. Para concluir a prova de que os usuários estão no mesmo grupo, o teste é repetido para o mesmo jogo comparando dessa vez a versão adaptativa: obtendo um *z-value* de -1,54 e *w-value* de 7, superior ao valor crítico 4, novamente a diferença não é significativa. Alterando o jogo para o Sonic Wings, obtemos o *z-value* de -0,28 e o *w-value* de 16 para a versão não-adaptativa, com o *z-value* de 0 e o *w-value* de 18 para a versão adaptativa. Em ambos os casos, os valores superam o W crítico de 4 e as diferenças não são significativas. Como indicado pelos 4 resultados, a diferença entre os grupos não é significativa e, com isso, teremos um único grupo de 16 usuários ao invés dos dois grupos originais. Mais uma vez, é preciso verificar se a melhoria proporcionada pelo controle adaptativo é significativa.

Primeiramente, comparamos as versões adaptativa e não-adaptativa do controle para o novo grupo unificado para o jogo Super Mario Bros. Com o novo grupo de 16 usuários, agora é possível utilizar tanto o *z-value* quanto o *w-value* para validar o teste de Wilcoxon. O *z-value* obtido foi -3,21 e o *p-value* foi 0,001. Esse resultado é significativo quando $p <= 0,05$. O *w-value* é 6 e o valor crítico de W para n = 16 quando $p <= 0,05$ é 29. Como confirmado pelo *z-value* e *w-value*, o controle adaptativo melhorou de forma significativa o desempenho dos usuários. Para o jogo Sonic Wings, o mesmo teste para o grupo unificado resultou em um *z-value* de -3,46 e *p-value* de 0,0005, que para $p <= 0.05$ é significativo, com um *w-value* de 1, para um valor crítico de 29. Com isso, nesse outro caso a melhora do controle adaptativo também é significativa. A Figura 16 mostra a diferença entre a precisão dos usuários do grupo unificado entre ambos os controles para os dois jogos, mostrando como o controle adaptativo

sempre fornece uma performance mais alta. Com isso, a hipótese H3 está comprovada e o controle melhora de forma significativa a taxa de sucesso dos usuários independente de sua experiência.

Por fim, testamos a hipótese H4, que propõe que a melhoria não depende do gênero do jogo. Para isso, usaremos o grupo unificado de usuários, possibilitando o uso do z-value e do w-value, para comparar as versões adaptativas do controle no Sonic Wings e no Super Mario Bros. O z-value obtido foi -3,36 com p-value de 0.008. Esse resultado é significativo para p <= 0.05. O w-value de 3 é inferior ao valor crítico de 29, mais uma vez comprovando que a diferença é significativa. Com isso, podemos rejeitar a hipótese H4 e afirmar que o controle pode ser adaptar melhor em determinados gêneros de jogos.

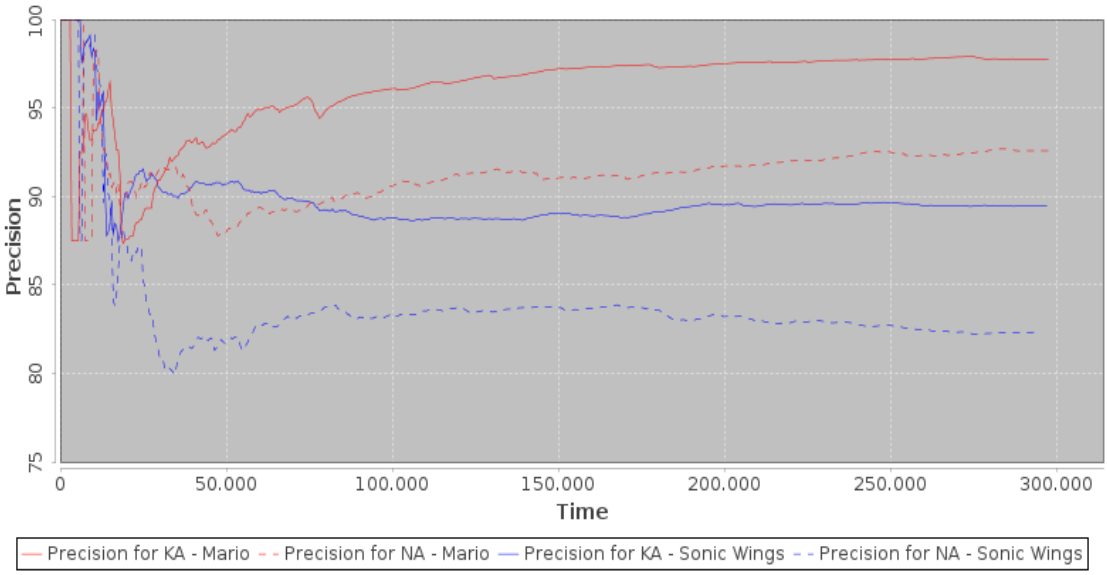

Figura 14: Precisão média dos usuários inexperientes no Super Mario Bros e Sonic Wings para os dois controles.

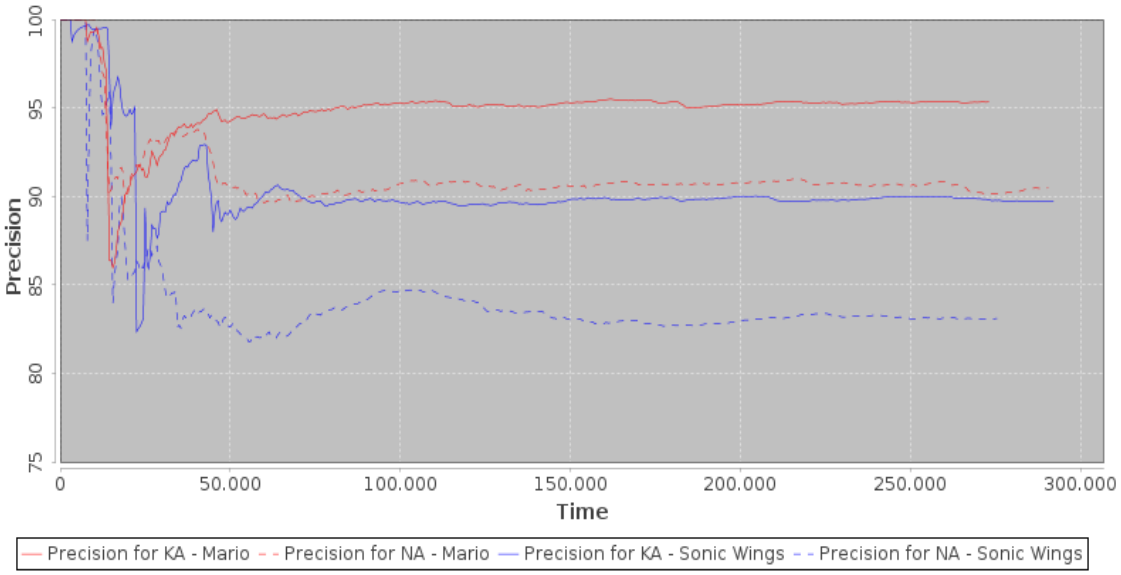

Figura 15: Precisão média dos usuários experientes no Super Mario Bros e Sonic Wings para os dois controles.

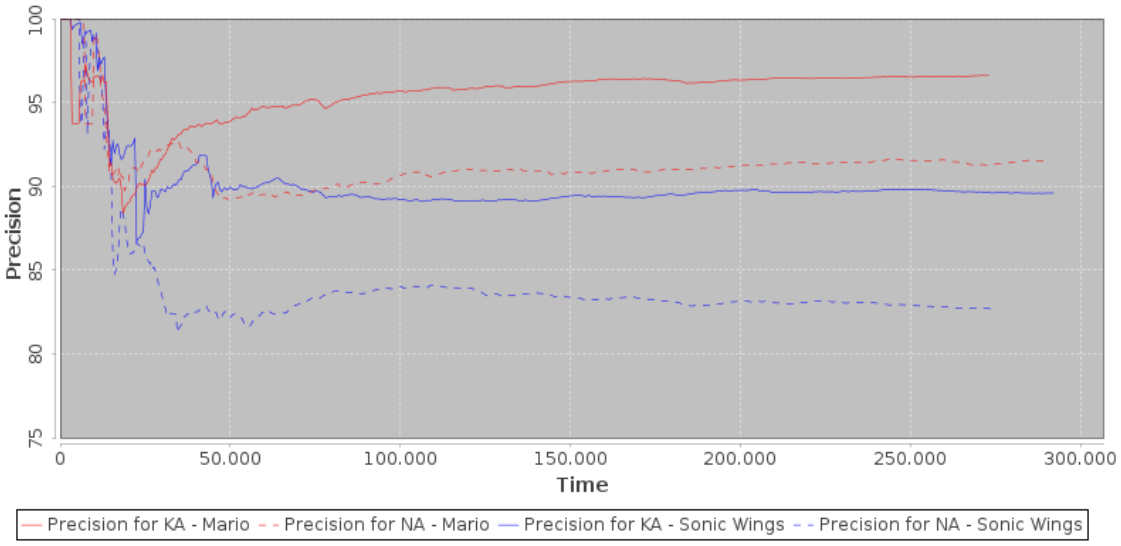

Figura 16: Precisão média de todos os usuários no Super Mario Bros e Sonic Wings para os dois controles.

5.2 ANÁLISE SUBJETIVA

A segunda parte da análise consistiu em um formulário, preenchido por todos os usuários após os 4 testes, para coletar a opinião pessoal de cada voluntário sobre ambos os protótipos. Para o jogo Super Mario Bros, 69% dos usuários acreditaram terem jogado melhor

com o controle adaptativo, com 88% afirmando que preferiam utilizar o controle adaptativo caso fossem jogar novamente. Já para o jogo Sonic Wings, a porcentagem de usuários que acreditou ter ido melhor com o controle adaptativo cresceu para 81%. Novamente, 88% dos usuários afirmaram que, em uma próxima oportunidade, escolheriam a versão adaptativa.

Além dessas duas perguntas, os usuários classificaram ambas as versões do controle seguindo uma escala de Likert, que inicia em 1 (muito difícil de usar) e iria até 5 (muito fácil de usar), permitindo aos usuários classificar a facilidade de uso de cada controle. Os resultados, apresentados na Tabela 4, mostram uma clara preferência pelas versões adaptativas.

A análise subjetiva complementou e confirmou os resultados dos testes objetivos, com ampla preferência pelo controle adaptativo. Uma observação interessante é que alguns usuários preferiram o controle não-adaptativo ao utilizá-lo na segunda sessão de testes, quando já se encontravam mais familiarizados com o jogo. Mesmo assim, o controle adaptativo ainda excedeu a performance, medida através da taxa de sucesso, da versão normal do controle. Isso indica que os usuários nem sempre possuem uma noção exata de sua performance e da quantidade de erros que cometeram, associando sua performance subjetiva diretamente ao desempenho no jogo, que pode ser influenciado por fatores como sorte e o próprio aprendizado das mecânicas do jogo. Com isso, podemos concluir que a análise objetiva oferece uma visão mais clara das vantagens do controle com adaptação.

Tabela 4: Opinião dos usuários quanto à dificuldade de uso dos controles.

Dificuldade de uso	Super Mario Bros.		Sonic Wings	
	Não-adaptativo	Adaptativo	Não-adaptativo	Adaptativo
Muito difícil	6%	0%	6%	0%
Difícil	44%	19%	38%	0%
Neutro	31%	25%	38%	31%
Fácil	19%	38%	13%	50%
Muito fácil	0%	19%	6%	19%

CAPÍTULO 6 – CONCLUSÃO

Novas formas de controles para jogos de *videogames* estão sendo propostas pela indústria de jogos para atrair mais jogadores e aumentar a imersão durante o *gameplay*. Muitos usuários evitam jogar *videogames* devido à complexidade dos controles, afastando-os da comunidade *gamer*. O uso de um dispositivo móvel como *joystick* permite a oportunidade de projetar interfaces específicas para cada jogo, tornando a interação mais precisa e adaptada para as mecânicas de cada jogo.

O trabalho apresentado não apenas investigou se a adaptação proposta melhora a interação usuário-controle, mas também buscou determinar em quais condições as adaptações funcionam melhor, comparando jogos de diferentes gêneros e realizando testes com usuários com diferentes níveis de experiência para avaliar a relação entre essas variáveis e a performance das adaptações.

Podemos concluir que o controle adaptativo traz benefícios para uma grande parte dos usuários e, a partir dessa avaliação, podemos apontar uma interface ideal.

Além disso, foi possível verificar que a adaptação irá ajudar os usuários independentemente de seu nível de experiência, mostrando resultados similares para usuários com e sem experiência. Também para os dois gêneros de jogos verificados, o controle adaptativo apresentou melhorias superiores a 5%, um valor considerável quando confrontado com o fato de que o usuário realiza em média 2500 toques na tela ao jogar Sonic Wings e 500 ao jogar Super Mario Bros em um intervalo de cinco minutos.

6.1 TRABALHOS FUTUROS

A interface proposta é capaz de melhorar a precisão do usuário ao tentar acertar um botão. Porém, o trabalho foi focado em adaptações motivadas pelo toque do usuário. Uma limitação dessa abordagem é que os toques são classificados como corretos apenas por atingirem um botão, mas sem nenhuma maneira de verificar se aquele era o botão que o usuário desejava apertar. Por isso, um trabalho futuro proposto é uma adaptação baseada no contexto do jogo, aonde o próprio jogo em execução poderia informar ao controle as ações corretas para cada situação e sugerir mudanças de *layout* de acordo com eventos vindouros no jogo. Essa melhoria especificamente iria permitir que o *game designer*, além de alterar o controle devido a eventos no jogo, pudesse definir a quantidade e o *layout* inicial dos botões. Com isso, o controle adaptativo permitiria que cada jogo possuísse uma interface de interação

projetada pelo próprio criador do jogo, que ainda seria otimizada durante o uso pelo algoritmo de aprendizado de máquina e se adequando às necessidades de cada usuário.

O foco de atenção dos usuários foi outro fator observado. Através de captura de vídeo, foi possível observar que os usuários olhavam mais vezes para o controle ao jogar com a interface não-adaptativa, perdendo o foco no jogo. Um trabalho futuro consiste em capturar dados valiosos de foco de atenção, como por exemplo, através de *eye-tracking*. O efeito de aprendizado, onde o usuário passa a jogar melhor à medida que aprende o jogo, é outro ponto a ser aprimorado. Sessões de treinamento mais longas ou testes com usuário já familiarizados com um determinado jogo são soluções a serem exploradas para nivelar a performance natural dos usuários. Testes com grupos maiores de usuários, com grupos com diferenças ergonômicas (deficientes físicos, pacientes com síndrome de *down*, idosos) e grupos específicos de canhotos são possibilidades que podem indicar se o controle apresenta um ganho de performance diferente para grupos com características diferentes.

Um fator difícil de ser medido é a capacidade do controle de melhorar o nível de diversão do usuário. Para isso, é proposto o uso de um *neuroheadset* para avaliar a relação entre a precisão do controle e a ativação de áreas específicas do cérebro, com foco em regiões ligadas à satisfação do jogador.

Por fim, outro avanço proposto pode vir na forma de novas maneiras de adaptar o controle, mudando não somente a posição e tamanho dos botões, mas sua forma. O desenvolvimento, estudo e avaliação dessa interface seria um grande avanço sobre o trabalho atual.

6.2 LIMITAÇÕES

A solução implementada ainda apresenta algumas limitações a serem solucionadas. Primeiramente, o cliente *desktop* não suporta a conexão de múltiplos controles, impossibilitando que mais de um jogador utilize o mesmo computador.

Em relação ao aplicativo móvel, o controle ainda não permite que o usuário configure os botões do controle, personalizando sua experiência inicial de uso que seria posteriormente aprimorada pelo aprendizado de máquina. De forma similar, ainda não existe a API completa para que o jogo possa alterar o controle ou personalizá-lo, apesar da estrutura de classes ser projetada para facilitar essa futura implementação.

Ainda em relação ao controle, caso o usuário troque de jogo, as adaptações iniciais ainda serão baseadas na experiência anterior. Além disso, após a sessão com o novo jogo, caso o usuário retorne ao jogo inicial, toda a experiência de treinamento será perdida. A separação

do controle em perfis de jogo, que permitam persistir os dados relativos a um jogo somente, melhoraria a experiência do usuário em um cenário de uso normal com diversos jogos.

6.3 CONTRIBUIÇÕES

Como resultado deste trabalho diversos trabalhos foram publicados. Os primeiros resultados obtidos com os testes com usuários foram submetidos em um *full paper* para o *HCI International 2015* (*International Conference on Human-Computer Interaction*), já aceito para publicação, e para uma sessão de demonstração no *ACM SIGGRAPH Asia 2014* (TOROK et al., 2014).

Apesar de conter resultados interessantes, a metodologia dos testes foi revisada, bem como a implementação do controle. Como resultado, a adaptação se tornou mais eficiente e uma nova sessão de testes mais rigorosos foi realizada, sendo essas a versão apresentada no presente trabalho. O novo controle e os novos resultados dos testes também irão gerar uma nova submissão de um *full paper*, a ser submetido ao *Mobile HCI 2015* (*International Conference on Human-Computer Interaction with Mobile Devices and Services*).

O presente trabalho gerou um depósito de patente no INPI, abrangendo o controle adaptativo e sua capacidade de ser configurado pelos usuários ou desenvolvedores do jogo (PELEGRINO et al., 2013).

Por fim, a próxima etapa do trabalho foi iniciada com o protótipo e análise da adaptação ao contexto do jogo, permitindo que o controle, além da adaptação ao usuário através de aprendizado de máquina, receba alterações diretamente do jogo. Com as adaptações sugeridas pelo contexto atual do jogo através da API e as sugestões da aprendizagem, se tornou necessário integrar ambas as correções através de um sistema de pesos para cada modificação. Esse trabalho foi publicado no SBGames 2014 (Simpósio Brasileiro de Jogos e Entretenimento Digital) na forma de *full paper* (PELEGRINO et al., 2014).

REFERÊNCIAS

Android SDK e plugin do Eclipse. Disponível em <http://developer.android.com/sdk/index.html?hl=sk>. Acesso em: 5 nov. 2014.

BALDWIN, T.; CHAI, J. *Towards online adaptation and personalization of key-target resizing for mobile devices*. In: Proceedings of the 2012 ACM international conference on Intelligent User Interfaces, ACM, p. 11-20, 2012.

BEZOLD, M.; MINKER, W. *Adaptive multimodal interactive systems*. Springer, p. 5, 2011.

BI, X.; ZHAI, S. *Bayesian Touch – A Statistical Criterion of Target Selection with Finger Touch*. ACM UIST '13, p. 51-60, 2013.

BlueCove - BlueCove JSR-82 project. Disponível em: <http://bluecove.org/>. Acesso em: 3 dez. 2014.

BRANDAO, A.; TREVISAN, D.; BRANDAO, L.; MOREIRA, B.; NASCIMENTO, G.; VASCONCELOS, C.; CLUA, E.; MOURAO, P. *Semiotic inspection of a game for children with down syndrome*. In: Games and Digital Entertainment (SBGAMES), 2010 Brazilian Symposium, p. 199 –210, nov. 2010.

BURKE, J. W.; MCNEILL, M.; CHARLES, D.; MORROW, P.; CROSBIE, J; MCDONOUGH, S. *Serious games for upper limb rehabilitation following stroke*. In: Proceedings of the 2009 Conference in Games and Virtual Worlds for Serious Applications, ser. VS-GAMES '09. Washington, DC, USA, IEEE Computer Society, p.103–110, 2009. [Online]. Disponível em: <http://dx.doi.org/10.1109/VS-GAMES.2009>. Acesso em: 10 set. 2014.

DRETSKE. *Explaining Behavior. Reasons in a World of Causes*. MIT Press, Cambridge, MA, USA, 1988.

FISHER, R.A. *Statistical methods for research workers.* Genesis Publishing Pvt Ltd, 1936.

GestureWorks Gameplay on Steam. Disponível em <http://store.steampowered.com/app/296610>. Acesso em: 2 out. 2014.

GOLOMB, M. R.; MCDONALD, B. C.; WARDEN, S. J.; YONKMAN, J.; SAYKIN, A. J.; SHIRLEY, B.; HUBER, M.; RABIN, B.; ABDELBAKY, M.; NWOSU, M. E.; BARKAT-MASIH, M.; BURDEA, G. C. *In-home virtual reality videogame telerehabilitation in adolescents with hemiplegic cerebral palsy*. Archives of Physical Medicine and Rehabilitation, e1, vol. 91, no. 1, 1 – 8, 2010. [Online]. Disponível em: <http://www.sciencedirect.com/science/article/pii/S000399930900817X>. Acesso em: 12 nov. 2014.

JOSELLI, M.; JUNIOR, J.R.S.; ZAMITH, M.; CLUA, E.; SOLURI, E. *A content adaptation architecture for games*. In SBGames, SBC, 2012.

JOSELLI, M.; SILVA JUNIOR, J. R.; ZAMITH, M.; SOLURI, E.; MENDONCA, E.; PELEGRINO, M.; CLUA, E. W. G. *An architecture for game interaction using mobile*. In Games Innovation Conference (IGIC), 2012 IEEE International, p.73–77, aug. 2012.

KOSTER, R. *Theory of fun for game design*. O'Reilly Media, Inc., 2013.

LAIKARI, A. *Exergaming - gaming for health: A bridge between real world and virtual communities*. In: Consumer Electronics. ISCE ’09. IEEE 13th International Symposium, p.665 –668, maio 2009.

LANGLEY, P. *Machine learning for adaptive user interfaces*. InKI-97: Advances in artificial intelligence, Springer Berlin Heidelberg, p.53-62, 1997.

LOWRY, R. *Concepts & Applications of Inferential Statistics.* Disponível em <http://vassarstats.net/textbook/ch12a.html>, 1998. Acessado em: 10 nov. 2014

MALFATTI, S. M.; DOS SANTOS, F. F.; DOS SANTOS, S. R. *Using mobile phones to control desktop multiplayer games*. In: Proceedings of the 2010 VIII Brazilian Symposium on Games and Digital Entertainment, ser. SBGAMES ’10. Washington, DC, USA: IEEE Computer Society, p.74–82, 2010.

PELEGRINO, M.; ZAMITH, M.; MENDONÇA, E.; JOSELLI, M.; TOROK, L.; CLUA, E. *Controle orgânico virtual para jogos por dispositivos eletrônicos com tela sensível ao toque configurável e adaptável ao uso*. BR 1020130300039, 22 nov. 2013. Disponível em: <http://www.inpi.gov.br/portal/artigo/busca_patentes>. Acesso em: 10 fev. 2013.

PELEGRINO, M.; TOROK, L.; CLUA, E.; TREVISAN, D. *Creating and designing customized and dynamic game interfaces using smartphones and touchscreen.* In: Procedings of the SBGames 2014, SBC, p. 819-826, 2014.

ROGERS, S.; WILLIAMSON, J.; STEWART, C.; MURRAY-SMITH, R. *Fingercloud: uncertainty and autonomy handover in capacitive sensing*. In: ACM CHI ’10, p. 577–580, 2010.

SCHELL, J. *The Art of Game Design: A book of lenses*. CRC Press, 2008.

SMOLA, A.; VISHWANATHAN, S. *Introduction to Machine Learning*. Cambridge University, UK, p. 32-34, 2008.

Steam controller. Disponível em <http://store.steampowered.com/livingroom/SteamController/>. Acesso em: 2 out. 2014.

STENGER, B.; WOODLEY, T.; CIPOLLA, R. *A vision-based remote control***.** In: Computer Vision: Detection, Recognition and Reconstruction, p.233–262, 2010..

TOROK, L.; PELEGRINO, M.; LESSA, J.; TREVISAN, D.; CLUA, E. *AdaptControl: an adaptive mobile touch control for games*. In: SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications. ACM, p. 18, 2014.

VAJK, T.; COULTON, P.; BAMFORD, W.; EDWARDS, R. *Using a mobile phone as a wii-like controller for playing games on a large public display*. Int. J. Comput. Games Technol. vol. 2008, p. 4:1–4:6, jan. 2008. [Online]. Disponível em: http://dx.doi.org/10.1155/2008/539078. Acesso em: 10 set. 2014.

WEI, C.; MARSDEN, G.; GAIN, J. *Novel interface for first person shooting games on pdas*. In: OZCHI '08: Proceedings of the 20th Australasian Conference on Computer-Human Interaction, OZCHI. New York, NY, USA, ACM, p.113–121, 2008.

WEIR, D.; ROGERS, S.; MURRAY-SMITH, R.; LÖCHTEFELD, M. *A user-specific machine learning approach for improving touch accuracy on mobile devices*. In: ACM UIST '12, p.465-476, 2012.

Weka: Data Mining Software in Java. Disponível em <http://www.cs.waikato.ac.nz/ml/weka/>. Acesso em: 5 jul. 2014.

Wi-Fi Direct | Wi-Fi Alliance. Disponível em: <http://www.wi-fi.org/discover-wi-fi/wi-fi-direct>. Acesso em: 3 dez. 2014.

ZAMITH, M.; JOSELLI, M.; SIVA JUNIOR, J.; PELEGRINO, M.; MENDONÇA, E.; CLUA, E. *AdaptControl: An adaptive mobile touch control for games*. In: SBGames, p.137-145, 2013.

ZYDA, M.; THKRAL, D.; JAKATDAR, S.; ENGELSMA, J.; FERRANS, J.; HANS, M.; SHI, L.; KITSON, F.; VASUDEVAN, V. *Educating the next generation of mobile game developers*. IEEE Computer Graphics and Applications, vol. 27, no. 2, pp.96, 92–95, 2007.

APÊNDICE A – DOCUMENTAÇÃO DOS MÉTODOS DAS CLASSES

Esse apêndice se destina a detalhar a documentação, incluindo os métodos específicos das classes abordadas no capítulo 3 com suas respectivas assinaturas e documentação de seu funcionamento.

Classe *Button*:

- *isInConflict(Button b):* recebe outro botão como parâmetro e verifica se existe algum conflito entre eles, ou seja, se existe alguma interseção entre a área de ambos, retornando *true* caso haja um conflito e *false* caso não haja nenhum. Durante as adaptações, após cada alteração na posição de um botão será verificado se ele conflita com qualquer um dos demais botões. Caso exista um conflito, a alteração de posição ou tamanho não será realizada.
- *canGrow(int newW, int newH)*: os parâmetros *newW* e *newH* representam, respectivamente, uma nova largura e uma nova altura para o botão. Esse método irá verificar se o crescimento do botão não o faria exceder as dimensões da tela, retornando *false* caso isso ocorra e *true* caso o crescimento possa ocorrer.
- *containsPoint(int px, int py)*: verifica se o ponto x = *px*, y = *py* se encontra dentro da área do botão, retornando *true* nesse caso. Esse método é utilizado para verificar se o toque do usuário acertou ou não o botão.

Classe *DpadButton*:

- *isInConflict(Button b):* os parâmetros e o retorno desse método funcionam da mesma forma que o método original sobrescrito. A diferença é que, ao verificar pelo conflito, o método irá ignorar qualquer conflito com o parâmetro b caso ele seja do tipo *DpadButton* e seja um dos botões secundários do direcional (diagonais), verificado através do método abaixo. Os conflitos dos botões principais do direcional com botões das diagonais são ignorados porque os botões das diagonais são ajustados automaticamente para ocupar os espaços vazios do direcional. Por isso, qualquer conflito resultante da adaptação será corrigido naturalmente ao adequar os botões diagonais ao novo *layout* do direcional, uma operação realizada após qualquer adaptação no direcional.

- *isSecondary():* o método irá retornar *true* se o objeto for um botão diagonal do direcional e *false* caso o objeta seja um dos botões de direção principais do direcional.

Classe *Joystick*:

- *addButton(Button b)*: adiciona o botão b à tabela *hash buttons*, com a chave correspondendo ao nome do botão (*b.getName()*). Caso já exista um botão com o mesmo nome, será substituído.
- *getButton(String name):* retorna o botão de nome *name* na tabela *hash buttons* ou *null* caso não haja um botão com o nome passado por parâmetro.
- *getButtons():* retorna um vetor com todos os botões do *joystick*.
- *getButtonsSortedByUse():* retorna um vetor de botões, ordenado pela quantidade de vezes que foi utilizada. A posição 0 do vetor irá conter o botão mais utilizado enquanto a última posição do vetor irá conter o botão menos utilizado. Em caso de empate no número de usos, os botões em questão serão retornados na mesma ordem que se encontram na lista original (ordem de inserção na lista).
- *sortValue(Hashtable<String, Integer> t):* retorna os itens da tabela *hash* t na forma de um *ArrayList* de *Entry<String, Integer>* ordenado. É utilizado internamente pela classe para a ordenação no método *getButtonsSortedByUse*.
- *hasButtons():* retorna *true* caso o *joystick* possua pelo menos um botão e false caso a lista de botões esteja vazia.
- *getPressedButtons():* retorna um vetor de botões (*Button[]*) com todos os botões que se encontram no estado "pressionado".
- *setPressedButton(Button b):* altera o estado do botão b na lista interna para pressionado.
- *setUnpressedButton(Button b):* altera o estado do botão b na lista interna para não-pressionado.
- *clearLog():* reinicia os contadores de toques totais na tela, toques totais por botão e toques corretos por botão para o valor inicial, 0.
- *logInput(int x, int y):* esse método incrementa os contadores corretos para um toque na posição *x*, *y*. Ao ser chamado, o método irá sempre incrementar o contador *totalPress*, que contém o número total de toques na tela. Após isso, o método irá verificar se o toque atingiu a área de algum botão, incrementando o contador *buttonPress* na posição determinada pela chave do botão na tabela *hash* (nome). Em seguida, o método irá

determinar o botão mais próximo do toque na tela (caso atinja algum botão, ele será o mais próximo necessariamente) e incrementa a tabela *hash buttonTotalPress* na posição correta. É interessante observar que a soma de todos os toques individuais nos botões armazenados em *buttonTotalPress* irá ser igual ao total *totalPress*. *buttonTotalPress* irá armazenar o total de toques por botão enquanto *buttonPress* irá armazenar o total de toques corretos por botão. Subtraindo o valor para o botão em *buttonTotalPress* pelo valor em *buttonPress*, o resultado será o número de toques incorretos destinados ao botão.

- *logButtonsHistory():* o método invoca o método *logHistory()* em cada botão do controle, armazenando a posição e tamanho atuais de cada botão em sua própria lista interna, para uso ao exportar os logs da sessão de uso do controle.
- *getTotalPress():* retorna o total de pressionamentos na tela armazenados na variável *totalPress*.
- *getButtonCorrectPressLog(String name):* retorna o total de toques que atingiram a área do botão de nome *name*, armazenadas na posição determinada pela chave *name* na tabela *hash buttonPress.*
- *getButtonTotalPressLog(String name):* retorna o total de toques que atingiram a área do botão de nome *name* ou cujo botão mais próximo fosse o botão *name*, armazenadas na posição determinada pela chave *name* na tabela *hash buttonTotalPress.*

Classe *DAL*:

- *getContext()*: retorna o objeto do tipo *Context* recebido originalmente pelo construtor da classe.
- *getTablename()*: método abstrato. O retorno é o nome da tabela no banco de dados que será acessada pela classe. Cada classe que implementar *DAL* irá sobrescrever esse método e ele deverá retornar o nome da tabela que será acessada pela classe filha.
- *getPreferences()*: retorna o objeto do tipo *SharedPreferences* da aplicação. Cada aplicativo Android pode criar seções no seu armazenamento interno que serão automaticamente preenchidas com informações gerenciadas na forma de um objeto *SharedPreferences*. Seu funcionamento é similar a uma tabela *hash*, permitindo armazenar valores em posições denominadas por uma chave (nesse caso, uma *String*). A principal diferença é que uma *SharedPreferences*, apesar de aceitar apenas strings como chaves, permite armazenar qualquer objeto de tipos básicos ou que possa ser

serializado, o que inclui classes como *String* e mesmo qualquer classe personalizada que implemente a interface *Parcelable*, bem como listas compostas pelos tipos de objetos mencionados. É importante mencionar que os dados armazenados dessa forma são persistentes, sendo mantidos após fechar a aplicação ou desligar o dispositivo, fornecendo uma persistência equivalente ao banco de dados. Alguns dos dados mais simples a serem armazenados, como preferências do usuário, não necessitam de uma tabela no banco de dados e por isso serão armazenados em uma *SharedPreferences.*

Classe *JoystickDAL*:

- *persist(Joystick joystick)*: o método persiste um objeto *Joystick.* Por ser uma classe mais simples, o objeto é serializado na forma de uma *String* e armazenado na *SharedPreferences*.
- *get()*: retorna o objeto *Joystick* persistido na *SharedPreferences*. É importante observar que apenas um objeto é persistido, já que apenas uma instância existe em qualquer momento e representa o controle exibido na tela para o usuário.
- *setAsDefault(Joystick joystick)*: define a posição e tamanho dos botões de *joystick* como valor padrão, armazenado na *SharedPreferences*. Caso o controle seja reconfigurado pelo usuário, esse será o *layout* padrão.
- *getDefault()*: retorna o objeto *Joystick* definido como padrão pelo método acima, ou a configuração inicial caso nenhum padrão jamais tenha sido configurado.

Classe *PointHistoryDAL*:

- *run()*: método implementado da interface *Runnable*. O construtor de *PointHistoryDAL* irá instanciar a *Thread t*, passando a própria instância da classe como o *Runnable* que irá ser executado pela *thread*. Após isso, *run* irá permanecer em um *loop* verificando se a fila *pointBuffer* atingiu o tamanho BLOCKSIZE. Nesse caso, os pontos serão removidos da fila e inseridos no banco (através do método *insertIntoDatabase*). Caso a fila ainda não tenha atingido o tamanho do *buffer, run* irá aguardar 200 ms e verificar novamente.
- *insertIntoDatabase(Point[] ps)*: insere todos os pontos do vetor de pontos *ps* no banco de dados. Esse método é privado e será chamado somente no método *run*, sendo executado na *thread* secundária em paralelo.

- *insert(Point p)*: o método público chamado por qualquer classe que precise inserir um novo toque no banco de dados. Essa chamada irá adicionar o ponto na fila de inserção no banco.
- *close()*: encerra a conexão com o banco de dados e finaliza a *thread* de inserção de pontos. Antes de terminar a *thread*, o método irá finalizar a inserção dos pontos restantes na fila, evitando qualquer perda de dados.
- *clearDatabase()*: apaga todas as entradas no banco de dados. Esse método é utilizado caso o usuário deseje reconfigurar o controle para seu estado inicial, limpando os dados de aprendizado e recomeçando do zero.
- *get(int id)*: retorna o ponto do banco de dados identificado por *id*.
- *list(int max)*: retorna uma lista com os últimos *max* pontos inseridos no banco de dados.
- *list()*: retorna uma lista com todos os pontos contidos no banco de dados.

Classe *LogManager*:

- *logButtonsError(Joystick joystick)*: retorna uma *string* contendo o log de toques corretos, incorretos e o total de toques para todos os botões do controle.
- *logButtonsError(Joystick joystick, int time)*: apresenta o mesmo retorno e funcionalidade do método anterior, porém contendo os dados amostrados a cada *time* milissegundos desde o início da sessão de testes.
- *logGeneralError(Joystick joystick, int time)*: retorna uma *string* contendo o log de toques corretos, incorretos e o total de toques para o *joystick* como um todo somando os eventos para todos os botões.
- *exportPlots(String name, String genLogs, String btnLogs, Joystick joystick)*: o método analisa os dados do log geral *genLogs* e do log por botões *btnLogs*, gerando gráficos com a taxa de acertos geral por tempo e a taxa de acertos por tempo para dada botão. Os logs serão prefixados pela *string name* e armazenados no sistema de arquivos do dispositivo em uma pasta específica acessível ao usuário no formato JPEG.
- *exportLogs(String name, Joystick joystick, Bitmap bitmap, String algorithm, int resx, int resy, String btnLogs, String genLogs)*: o método salva os logs *btnLogs* e *genLogs* em um arquivo de texto no mesmo diretório dos gráficos gerados e armazena uma captura da tela atual do controle na variável *bitmap*. O log textual ainda irá conter, como referência, a resolução do dispositivo (*resx* e *resy*) e o nome do algoritmo de

aprendizado utilizado (ou *"none"* caso a versão não-adaptativa tenha sido utilizada) indicado em *algorithm*.

- *getLogFiles()*: retorna um vetor do tipo *File*, contendo referências para todos os arquivos de log e gráficos armazenados.
- *clearLogs()*: apaga todos os arquivos de logs e gráficos do sistema de arquivos.
- *readLogFile(String filename)*: acessa o arquivo de log de nome *filename* no diretório de logs e retorna seu conteúdo como uma *string*.

Classe *JoystickRelativeLayout*:

- *getObserver()*: retorna a instância do *Observable* para assinar os eventos da classe.
- *SetJoystick(Joystick joystick)*: define a instância de *Joystick* a ser utilizada pela classe. Será um ponteiro para a mesma instância da classe *MainActivity* e o estado dos botões será alterado diretamente.
- *setBluetoothManager(BluetoothComManager blueCom)*: define a instância de *BluetoothComManager* que será utilizada pela classe. *MainActivity* ira inicializar uma instância e efetuar a conexão e pareamento com o computador. Após isso, *JoystickRelativeLayout* irá assumir o controle da classe e enviar qualquer comando de pressionamento de botões para o computador.
- *onTouchEvent(MotionEvent ev)*: método sobrescrito da superclasse *View*. Esse método é chamado pelo sistema Android sempre que houver um toque no controle *JoystickRelativeLayout* (como ele ocupa a tela inteira, qualquer toque no controle é interceptado pelo mesmo). Cada toque é repassado para o método *updateButtonState*, que irá realizar a atualização do estado dos botões, e depois inserido no banco de dados através de *JoystickDAL*.
- *updateButtonState(MotionEvent ev)*: método privado chamado por *onTouchEvent* para atualizar o estado dos botões. Durante sua execução, será verificado se o ponto tocado indicado por *ev* atingiu um botão, definindo seu estado para pressionado. Além disso, os botões que não estão mais na área de um toque do usuário terão seu estado definido como não-pressionado. Cada alteração de estado é informada para o cliente *desktop* através do canal de comunicação *Bluetooth* na instância de *BluetoothComManager* da classe.
- *stop()*: o método realiza uma chamada para *close* na instância de *PointHistoryDAL*, encerrando a *thread* de inserção de pontos e fechando a conexão com o banco.

Classe *AdaptiveCorrection*:

- *getGenLogs()*: retorna uma *string* com os logs gerais do controle, coletados constantemente através de um *LogManager.*
- *getBtnLogs()*: retorna uma *string* com os logs dos botões do controle, coletados constantemente através de um *LogManager.*
- *run()*: método implementado da interface *Runnable*. Dentro desse método existe um *loop*, aonde os últimos pontos do banco são obtidos através de *PointHistoryDAL* e repassados para o algoritmo de aprendizado. Após o retorno com as adaptações, *MainActivity* é notificada através de um evento sobre as adaptações determinadas pelo aprendizado.
- *getCorrectionAlgorithm()*: retorna o nome do algoritmo de aprendizado em uso, como uma *string*.
- *reset()*: reconfigura o controle, apagando o banco de dados e reiniciando o log atual.
- *start():* inicia a *thread* do aprendizado de máquina.
- *stop():* encerra a *thread* do aprendizado de máquina.

Classe *CorrectionAlgorithm*:

- *correct(Joystick joystick, Point[] data, int qtd, int maxChange, int maxGrow, boolean size, boolean position, int minPress)*: método abstrato, que retorna um vetor de pontos representando as posições corretas. Os botões são retirados de *joystick* e *data* contém os últimos toques do usuário. *maxChange* indica a quantidade máxima de *pixels* que o botão pode se deslocar por iteração, enquanto *maxGrow* indica seu crescimento máximo em *pixels* nesse mesmo intervalo. Os valores de *size* e *position* permitem definir se as alterações de tamanho e posição serão efetuadas. Por fim, *minPress* determina a quantidade mínima de toques para que o aprendizado seja iniciado.
- *buildArff(Point[] arrays)*: método protegido. Gera um arquivo ARFF em memória dos pontos em *array*, utilizado como entrada pelos algoritmos de aprendizado no Weka, retornando um objeto do tipo *Instances* do Weka.
- *correctSize(Joystick joystick, ArrayList<CorrectionResult> results, Point[] centroids, Button[] btnCentroids, Point[] data, int[] selClasses, int maxGrow, int minPress)*: método protegido. Realiza a correção de tamanho, com os botões em *joystick*. Em *results,* temos os resultados da correção de posição efetuada antes em *correct* e esse

método irá alterar as informações de tamanho dos resultados quando necessário, avaliando os toques do usuário em *data*.

- *applyCorrection(Joystick joystick, ArrayList<CorrectionResult> results, int maxChange, int maxGrow, int minPress, boolean size, boolean position)*: método protegido. Aplica as correções de posição e tamanho em *results* diretamente em *joystick*. Os demais parâmetros seguem o mesmo padrão dos métodos anteriores.
- *fixSecondaryDpad(Joystick joystick)*: reposiciona os botões diagonais do direcional para que ocupem os espaços vazios do direcional.
- *isDpad(Button b)*: método privado. retorna *true* se o botão for parte do direcional e *false* caso contrário.
- *checkConflict(Button closestButton, Button[] buttons)*: método protegido. Verifica se o botão *closestButton* está em conflito com qualquer botão em *buttons* (que não seja ele mesmo), retornando *true* em caso de conflitos.
- *isDpadConsistent(Button b, Button[] buttons)*: método privado. Se *b* for um botão do direcional, verifica se a nova posição e tamanhos em *b* violam alguma regra de consistência do direcional, contido no vetor *buttons*, retornando *false* caso o direcional não esteja em uma configuração válida.
- *delta(Point b, Point centroid, int max)*: método protegido. Retorna o delta em x e y que o botão em *b* deve percorrer para atingir o ponto *centroid*, limitado pelo deslocamento máximo *max.*
- *dist(Point a, Point b)*: método protegido. Utilitário para retornar a distância entre os pontos *a* e *b.*

Classe *KmeansCorrection*:

- *correct(Joystick joystick, Point[] data, int qtd, int maxChange, int maxGrow, boolean size, boolean position, int minPress)*: métodos que sobrescreve o método abstrato na classe pai, detalhado anteriormente, com o diferencial de utilizar o K-means. Após corrigir a posição, o método *correctSize* da superclasse é chamado. Retorna uma lista de *CorrectionResult* com as correções de posição.
- *correctPosition(Point[] centroids, int[] btnIndex, Button[] buttons)*: método privado. Contém a implementação do K-means, que analisa os pontos em *data* convertidos em um ARFF, determinando os centroides das classes e os associa aos botões em *buttons*. Os centroides são armazenados em *centroids*, com os índices que correlacionam os

centroides com os botões em *buttons* armazenados em *btnIndex*. Todos os parâmetros são ponteiros que são acessíveis em *correct*.