Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Arcabouço para Construção de Jogos Ubíquos com Foco em Reabilitação

Luciano Henrique de Oliveira Santos

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientadora
Prof.ª Dr.ª Carla Denise Castanho

Brasília
2016

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

SSA237a

Santos, Luciano Henrique de Oliveira
Arcabouço para Construção de Jogos Ubíquos com Foco em Reabilitação / Luciano Henrique de Oliveira Santos; orientador Carla Denise Castanho. -- Brasília, 2016.
73 p.

Dissertação (Mestrado - Mestrado em Informática) - Universidade de Brasília, 2016.

1. Computação Ubíqua. 2. Jogos Ubíquos. 3. Reabilitação. I. Castanho, Carla Denise, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Arcabouço para Construção de Jogos Ubíquos com Foco em Reabilitação

Luciano Henrique de Oliveira Santos

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof.ª Dr.ª Carla Denise Castanho (Orientadora)
PPGInf/CIC/UnB

Prof. Dr. Antonio Padilha Lanari Bo
ENE/FT/UnB

Prof. Dr. Rodrigo Bonifacio de Almeida
PPGInf/CIC/UnB

Prof.ª Dr.ª Célia Ghedini Ralha
Coordenadora do Programa de Pós-graduação em Informática

Brasília, 01 de abril de 2016

Dedicatória

Dedico este trabalho a todos aqueles que nunca se cansam de aprender, que renovam diariamente seu fascínio pelo saber e que, sábios, maravilham-se na eterna constatação da própria ignorância.

Esta obra é para os que têm a coragem de acordar cada manhã preferindo a labuta da descoberta e da investigação, que inquietam-se na certeza, porque entendem a beleza imensa, ainda que angustiante, da dúvida.

É, portanto, para os jovens de coração. Que permaneçam assim até o fim, e, mesmo então, terão outros mistérios para desvendar.

Agradecimentos

"O sucesso tem muitos pais, mas o fracasso é órfão."

Quando o presidente John F. Kennedy disse essa frase, trazia uma visão bastante pessimista de mundo, e referia-se à atitude de abandono dos indivíduos em face da adversidade. Mesmo assim, é possível apreciar a verdade de suas palavras em relação às conquistas.

Não poderia eu ter a arrogância de reinvidicar já ter atingido o sucesso, mas posso sem dúvida alardear o privilégio de ter a companhia daqueles me apoiaram sempre, nas vitórias e também – diferente do que previra Kennedy – nas derrotas. Por esse motivo, afirmo: se obtiver qualquer sucesso na vida, nunca terei a responsabilidade exclusiva pelos resultados, mas sempre o dever de expressar minha gratidão aos tantos que seguiram ao meu lado durante a jornada.

Começo, então, por aqueles que estiveram comigo desde o princípio. Minha mãe, Raquel, meu pai, José Maria, e minhas irmãs, Lissane e Vanessa. Obrigado pela compreensão das ausências, por suportarem as impaciências e maus-humores das noites sem dormir e do tanto a fazer no tão pouco tempo. Sou grato em especial por cada vez que expressaram sua admiração, mesmo que imerecida.

Agradeço com muito carinho à minha segunda família, meus companheiros de aventuras da Balance. Pelo apoio e entusiasmo, por cada momento de alegria e descontração que tornaram o esforço mais suportável, por cada ajuda e pitaco, e por continuarem sonhando junto comigo, mesmo quando não pude estar lá para ajudar. Agradeço em especial aos meus irmãos de outros pais: Rafael, Mari, Megas e Pedro. Este último ainda por me fazer companhia nessa loucura inexplicável dos que buscam a vida acadêmica.

Deixo minha lembrança aos integrantes do nosso grupo de pesquisa, UnBiquitous, onde tanto aprendi. Primeiro à minha orientadora, Profª Carla, que vem me ensinando novas lições desde que eu era calouro na UnB, e me aconselhou tão bem em cada passo. Também ao meu orientador informal e guru para tantos e variados temas, Fabricio, que tantas coisas novas me fez aprender, tantas formas diferentes de enxergar o mundo me fez

vislumbrar. Por ambos tenho não apenas o respeito e admiração de aluno, mas o carinho da amizade sincera.

Agradeço também aos colegas de pesquisa com quem tive o prazer de trabalhar e que me ajudaram imensamente nesse projeto, Matheus Pimenta e Luísa Nunes, bem como os amigos Lucas Fonseca, que tanto recursos mobilizou, e Marco Akira, por seus ótimos conselhos, que trouxeram nova perspectiva às nossas discussões. Ao Jeremias, nosso "Síndico do LAICO" e eterno representante, pela companhia nos dias de labuta no laboratório e pelas excelentes discussões sobre o universo, a vida e tudo mais.

Ao professor Tiago Barros, muito, muito obrigado! Por sua excelência na arte de imaginar e projetar experiências novas e belas. E, ao final, mas não menos, ao Prof. Ricardo Jacobi, que nos apoiou com sua experiência e disposição sempre que buscamos sua ajuda.

Também um agradecimento especial aos professores Antonio Padilha e Rodrigo Bonifacio, por terem aceitado o convite para participarem da banca de avaliação deste trabalho e pelos seus valiosos comentários e sugestões.

Ao Departamento de Ciência da Computação, a casa que me acolheu ainda quando a sabedoria era pouca e a ansiedade muita, expresso minha gratidão por ter-me aumentado um pouco a primeira. É triste que eu ainda não tenha conseguido diminuir a última, apesar do esforço constante nesse sentido. Meu carinho será eterno, impossível que não seja, depois de tanto tempo. Com um respeito especial, deixo também meus agradecimentos aos professores que marcaram minha vida acadêmica e me inspiraram a buscar também esse caminho, Prof Marcus Lamar e Prof^{a} Cláudia Nalon.

Agradeço, por fim, aos que participaram da jornada mas não foram citados aqui. A falha do autor, podem ter certeza, é o esquecimento momentâneo e limitação de espaço, não a ingratidão.

Resumo

A computação ubíqua busca empregar a grande diversidade de dispositivos computacionais disponíveis no dia a dia para facilitar tarefas ou processos. Jogos ubíquos, por sua vez, aplicam os princípios da computação ubíqua para criar novos tipos de mecânica e modelos de interação, com a inserção de elementos do mundo real interagindo e confundindo-se com o mundo virtual. A terapia de reabilitação pode beneficiar-se desse modelo, pois frequentemente envolve um processo longo e contínuo, no qual a motivação do paciente é considerada fator crucial.

Sistemas ubíquos em geral envolvem a adaptabilidade ao ambiente, com resposta constante à dinamicidade das características deste ambiente e das pessoas, dispositivos e aplicações. Jogos ubíquos voltados à reabilitação, no entanto, devem ainda lidar com a adaptabilidade ao paciente, ou seja, precisam levar em consideração os aspectos de saúde e as necessidades individuais de cada pessoa, no que diz respeito à natureza da lesão e às diferentes capacidades e amplitudes de movimentação.

Focando-se nestes problemas, este trabalho propôs uma nova solução que estende uma infraestrutura de computação ubíqua existente para trazer um novo conjunto de ferramentas para lidar com os desafios específicos da reabilitação. A proposta inclui nova forma de organização do *smartspace*, um conjunto de interfaces e mecanismos de conversão de entradas, bem como ferramentas auxiliares e suporte a dispositivos de captura de movimento.

A validação da solução foi realizada por meio de um protótipo do mecanismo de entrada de um jogo conceito, em que o paciente deve realizar movimentos fisioterápicos com sucesso para controlar um personagem virtual, que dá golpes mais ou menos intensos, de acordo com a taxa de acerto na execução de cada movimento. Foram realizadas medidas do processo de detecção desses movimentos pelo sistema, sendo analisado o funcionamento da infraestrutura implementada como um todo, inclusive com diferentes dispositivos de entrada, obtendo-se um comportamento de acordo com o esperado.

Palavras-chave: Computação Ubíqua, Jogos Ubíquos, Reabilitação

Abstract

Ubiquitous computing takes advantage of the great diversity of computing devices available on daily environments to ease the execution of tasks or processes. Ubigames, as the name suggests, employ ubiquitous computing's principles on the design of new kinds of mechanics and interaction models, bringing elements from the real world to interact and blend with the virtual world. Rehabilitation therapy may benefit from this paradigm, since it often requires a long and continuous process in which patient motivation is a crucial factor.

Ubiquitous systems, in general, deal with environment adaptability, i.e., the continuous response to the dynamic characteristics of the environment, as well as of the people, devices and applications. Ubigames targeting rehabilitation, for that matter, must also incorporate patient adaptability, which means taking into consideration aspects related to specific health conditions and to individual necessities of each person, with regards to nature of lesion and different capabilities and range of movement.

Focusing on these challenges, this research proposed a new solution that extends an existing ubiquitous computing infrastructure and brings a new set of tools to face the specific problems of rehabilitation. The proposal includes a new way of organizing the smartspace, a new set of interfaces and input conversion mechanisms as well as additional tools and support for movement capture devices.

The solution was validated via a prototype of the input mechanism of a conceptual game, in which patients must successfully execute physiotherapy exercises in order to control a virtual character, who performs blows with higher or lower strength, according to the success rate of each movement. Measurements were taken of these movements' detection process, with the solution as whole being analysed for its behavior, including with different input devices, and it worked as expected.

Keywords: Ubiquitous Computing, Ubiquitous Games, Rehabilitation

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A *computação ubíqua* [53], *i.e.*, onipresente, propõe um novo paradigma computacional, em que a enorme diversidade de dispositivos com poder computacional e capacidade de comunicação são empregados para facilitar tarefas ou processos. Seu principal objetivo é desenvolver sistemas computacionais úteis, que facilitem a vida das pessoas em vários aspectos sem, no entanto, exigir atenção ou interação constante dos usuários.

Aplicações ubíquas devem permitir ao usuário realizar tarefas com o auxílio de tecnologia, mas sem que o dispositivo computacional fique em evidência na execução dessas tarefas. Elas devem ser *proativas*, monitorando constantemente o ambiente e buscando prever ações dos usuários e melhorar sua experiência; *transparentes*, utilizando mecanismos de interação pouco invasivos, de tal maneira que a atenção do usuário seja focada apenas nas ações que deseja realizar; e *adaptativas* – reagindo às constantes mudanças que acontecem na rotina pessoal e no ambiente, comportando-se corretamente em diversos contextos.

Quando esses princípios são aplicados em conjunto, cria-se um *smartspace*, *i.e.*, *ambiente inteligente*, em que o sistema constantemente monitora os dispositivos, os recursos e os usuários, identificando-os e agindo proativamente para facilitar as diversas atividades no ambiente.

Os chamados jogos ubíquos [6] ou *ubigames*, por sua vez, aplicam os princípios da computação ubíqua para criar novos tipos de mecânica e modelos de interação. Os jogos deste novo gênero são conhecidos também como jogos "pervasivos" [51], "sensíveis ao contexto" [27], de "realidade misturada" [7] ou ainda de "trans-realidade" [23], pois caracterizam-se pela inserção de elementos do mundo real interagindo e confundindo-se com o mundo virtual. Essa mistura de mundos amplia fortemente o nível de engajamento entre os jogadores, bem como entre estes e o jogo em si, quebrando as fronteiras entre real e virtual [10].

Diferentes estudos mostram que jogos eletrônicos podem exercer um efeito positivo

em áreas que requerem dedicação e disciplina por parte do usuário, tais como educação [3, 34], treinamentos técnicos [43] e na área de saúde em geral [4, 35]. Jogos ubíquos, em especial, são particularmente adequados para este fim, visto que não apenas propõem novas mecânicas com foco em imersão e engajamento, mas também misturam o mundo real e o virtual, possibilitando a integração entre os vários ambientes presentes na vida do jogador.

Nesse contexto, a Terapia de Reabilitação, cujo foco está em restaurar as habilidades de pessoas que possuem alguma deficiência, pode se beneficiar com a utilização de jogos eletrônicos durante sessões fisioterápicas. Este tipo de terapia envolve um processo longo e contínuo, no qual a motivação do paciente é considerada fator crucial tanto para a efetividade do tratamento quanto para a velocidade da recuperação [26].

Jogos projetados com este fim devem levar em consideração os aspectos de saúde e as necessidades individuais de cada paciente. Por exemplo, em uma mesma clínica, é possível que haja uma pessoa com necessidade de reabilitação em um membro superior, outro em um membro inferior, outro ainda com dificuldade de movimentação nas mãos e mais um com uma lesão na coluna. Mesmo entre pacientes que possuem lesões similares, podem existir graus diferentes de capacidade de movimentação e amplitude de cada movimento.

Portanto, um jogo aplicado com o propósito de reabilitação deve ter flexibilidade, observando-se as limitações e características individuais de cada paciente, no que diz respeito à gravidade das lesões; capacidade e amplitude dos movimentos e nível de progresso no tratamento. O jogo não pode ser restritivo em relação ao tipo de entrada esperado, isto é, pacientes com diferentes tipos de traumas ou com limitações em partes do corpo diversas devem ser capazes de jogar o mesmo jogo, ou até mesmo interagirem dentro dele. Além disso, deve ser possível coletar metadados sobre as sessões que permitam o acompanhamento tanto pelo paciente quanto pelo profissional da saúde.

Logo, observa-se aqui que jogos ubíquos para reabilitação envolvem dois tipos de *adaptabilidade*. Quando vistos como jogos ubíquos em geral, o foco é a *adaptabilidade ao ambiente*. Nesse caso, espera-se que proponham mecânicas que mudem o comportamento do jogo de acordo com as características do ambiente e as pessoas, dispositivos, recursos e demais aplicações presentes. Por outro lado, quando vistos como aplicações voltadas à terapia de reabilitação, exige-se que tenham *adaptabilidade ao paciente*, ou seja, que possam ser personalizados para atender às necessidades de terapia e às limitações de cada indivíduo.

Fica claro, então, que o desenvolvimento de jogos ou outras aplicações ubíquas com fins específicos de reabilitação exigem mecanismos especialmente desenvolvidos para facilitar a adaptabilidade *ao paciente*, *i.e.*, que permitam sua utilização por uma gama variada de pacientes, com lesões variadas e diferentes capacidades de movimentação, além de

métodos para coleta de dados sobre o processo de tratamento.

Enquanto vários trabalhos abordaram desafios da computação ubíqua e dos jogos ubíquos, e outros focaram-se em casos específicos de reabilitação, nenhum propôs um mecanismo para lidar com as necessidades de adaptabilidade ao paciente que fosse genérico o suficiente para ser potencialmente utilizado em qualquer processo fisioterápico.

No contexto dos jogos comerciais, observou-se alguns casos em que aqueles voltados à prática de exercícios físicos foram empregados com algum grau de sucesso em tratamentos de lesões específicas. Nenhum deles, porém, pode ser utilizado para o tratamento de uma gama de pacientes variada, incluídos os casos em que os exercícios a serem praticados são incompatíveis com os padrões de entrada esperados pelo jogo. Não apenas estes jogos exigem movimentos bastante específicos, aplicáveis apenas a determinados fins, esses mesmos movimentos devem ser realizados dentro de uma amplitude fixa e o progresso no jogo não reflete o progresso na terapia, limitando ainda mais sua aplicabilidade para o fim de reabilitação.

Focando-se nestes problemas, este trabalho propôs uma nova solução que estende uma infraestrutura de computação ubíqua existente para trazer um novo conjunto de ferramentas para lidar com os desafios inerentes à reabilitação, em especial a *adaptabilidade ao paciente*.

Especificamente, foi necessário propor uma nova forma de organização do *smartspace*, criando a figura de um nó coordenador, que concentra os perfis de cada usuário. Esse perfil determina como os *inputs* capturados para um usuário específico devem ser convertidos para, ao final, gerarem entradas em um formato padronizado para o jogo.

Para viabilizar esse modelo, novas interfaces e mecanismos de conversão de entradas foram definidos e disponibilizados na forma de uma biblioteca, que permite tanto a criação de novos jogos quanto a adaptação dos existentes. Foram implementadas também ferramentas auxiliares, que permitem criar e modificar perfis para os usuários. Por fim, foi adicionado o suporte a dispositivos de captura de movimentos, que permitem o reconhecimento de exercícios fisioterápicos de forma precisa.

A validação da proposta foi realizada por meio de um protótipo que mapeia movimentos humanos em golpes de um personagem virtual em um jogo. Diferentes exercícios foram realizados, envolvendo os membros superiores e inferiores, resultando na detecção bem sucedida dos movimentos.

O restante deste texto está organizado da seguinte forma: o Capítulo 2 traz uma exploração mais detalhada das grandes áreas que norteiam este projeto, bem como o levantamento dos desafios a serem superados que motivaram este trabalho; o Capítulo 3 traz o estado da arte e as limitações encontradas nos trabalhos já publicados; os Capítulos 4 e 5 descrevem a solução proposta em detalhes, bem como sua implementação, tratando,

respectivamente, da biblioteca básica e das aplicações adicionais que auxiliam na sua utilização; o Capítulo 6 descreve os resultados obtidos e medidas realizadas; por fim, o Capítulo 7 traz as considerações finais e os trabalhos futuros.

Capítulo 2

Motivação

O tema deste trabalho agrega as áreas de computação ubíqua, jogos ubíquos e terapia de reabilitação. Nas seções a seguir, esses temas são explorados no âmbito de sua relevância para problema aqui abordado. Em seguida, são elencados seus principais desafios e, por fim, descritos o objetivo geral e os objetivos específicos dessa pesquisa.

2.1 Computação Ubíqua

A *computação ubíqua* [53] (do latim *ubiquu* – onipresente) propõe um novo paradigma tecnológico, em que os vários dispositivos com poder computacional e capacidade de comunicação, agora lugar-comum em diversos ambientes, trabalham em conjunto para facilitar tarefas.

De acordo com Mark Weiser, "as tecnologias mais profundas são aquelas que desaparecem", i.e, tornam-se tão integradas a nossa rotina que passam a ser invisíveis. Esta proposta de *tecnologia calma* [52] tem como principal objetivo desenvolver sistemas computacionais úteis, que facilitem a vida das pessoas em vários aspectos sem, no entanto, exigir atenção constante dos usuários, permitindo-lhes focar-se na tarefa que estão realizando no momento, em detrimento de detalhes tecnológicos.

Um exemplo clássico desse tipo de tecnologia é a escrita. Durante milênios ela foi acessível apenas a uma camada diminuta da população, no entanto, nos tempos atuais é onipresente nas mais diversas culturas. Ao escrever um texto, o autor não está particularmente concentrado na sintaxe, grafia ou na forma individual de cada caractere, mas, em vez disso, procura a melhor maneira de expressar uma ideia, sem preocupar-se com detalhes do sistema em si.

De maneira similar, pode-se citar também a eletricidade, que permite aos usuários o emprego da tecnologia sem requerer conhecimento sobre detalhes de sua operação. Para se utilizar um dispositivo que requer eletricidade, não é necessário saber a origem da

corrente e, em geral, pode-se assumir que ela esteja sempre disponível, de forma que a tecnologia se perde em meio à sua funcionalidade.

Por analogia, sistemas computacionais ubíquos devem permitir ao usuário realizar tarefas – *e.g.*, organizar compromissos, gerenciar ou manipular recursos, lembrar-se de atividades ou eventos – de maneira transparente, sem que o dispositivo computacional fique em evidência. Para atingir este objetivo, é necessário que aplicações ubíquas sejam:

- *proativas* – devem monitorar constantemente o ambiente em que estão inseridas, buscando prever ações dos usuários e tomando medidas para melhorar sua experiência;
- *adaptativas* – precisam ser flexíveis para reagir às constantes mudanças que acontecem na rotina pessoal e no ambiente, de tal forma que se comportem corretamente em diversos contextos;
- *transparentes* – utilizando mecanismos de entrada e *feedback* pouco invasivos, de modo que a atenção do usuário seja focada apenas em seus objetivos e nas ações que deseja realizar, sem preocupação com detalhes da tecnologia.

Quando todos esses princípios são aplicados em conjunto, cria-se um *smartspace*, *i.e.*, *ambiente inteligente*, em que aplicações ubíquas constantemente monitoram os dispositivos, os recursos e os usuários, identificando-os e agindo proativamente para facilitar as diversas atividades no ambiente.

A implementação desse conceito em aplicações reais traz consigo uma série de desafios tecnológicos a serem superados [16]. Em primeiro lugar, é necessário levar em consideração a enorme *heterogeneidade* tanto de dispositivos computacionais quanto de canais de comunicação, criando-se interfaces para que eles troquem informações de maneira eficiente e sem erros.

Além disso, é preciso que sistemas ubíquos sejam *escaláveis*, mantendo sua funcionalidade e tempo de resposta mesmo em face da imensa quantidade de dispositivos e recursos tecnológicos presentes. Devem ser também *tolerantes a falhas*, reagindo rapidamente a erros em serviços ou à constante movimentação de recursos e pessoas, mantendo a melhor qualidade possível para a experiência do usuário.

Espera-se que as aplicações sejam altamente *adaptativas*, modificando seu comportamento em resposta à constante entrada e saída de pessoas, dispositivos e recursos no ambiente, bem como às mudanças nas próprias condições do ambiente (tempo, temperatura, umidade, luminosidade, etc). Adicionalmente, suas *interfaces* devem ser cuidadosamente projetadas para manter o princípio de transparência e refletir corretamente todos esses elementos de adaptabilidade.

Por fim, é preciso que haja preocupação com questões de *privacidade* e *segurança*, criando-se medidas para proteger os dados dos diversos usuários de acessos indevidos ou não autorizados.

2.2 Jogos Ubíquos

À primeira vista, jogos eletrônicos e computação ubíqua são conceitos incompatíveis [54]. Em sentido oposto aos ideais ubíquos, jogos têm como principal objetivo atrair a atenção do jogador, imergi-lo em um novo mundo virtual com mecânicas, estória e desafios envolventes para atingir algum objetivo, seja aumentar o engajamento em alguma atividade – *e.g.* treinamento, educação, saúde – seja pelo simples entretenimento.

Nesse contexto, pesquisas sobre jogos ubíquos, os chamados *ubigames*, não têm como foco transformar diretamente jogos em aplicações ubíquas, mas sim em aplicar os princípios da computação ubíqua para criar novos tipos de mecânica e modelos de interação, tornando o *gameplay* mais imersivo e possivelmente ampliando os fatores de diversão.

Existem várias definições para o conceito de jogo ubíquo [6, 22], com algumas diferenças entre si. Todas elas, no entanto, têm como ponto comum a inserção de elementos do mundo real interagindo e confundindo-se com o mundo virtual, nos chamados jogos "pervasivos" [51], "sensíveis ao contexto" [27], de "realidade misturada" [7] ou ainda de "trans-realidade" [23]. Tomando como referência o contínuo "realidade-virtualidade" [30], os jogos ubíquos podem figurar desde a "realidade aumentada" até a "virtualidade aumentada", ou seja, em diferentes níveis de mescla entre o real e virtual. Essa mistura de mundos amplia fortemente o nível de engajamento entre os jogadores e entre estes e jogo em si [10].

Alguns jogos ubíquos podem ser organizados em "famílias", ou "modalidades" comuns, que utilizam conjuntos de mecânicas similares.

Em jogos no estilo "Caça ao Tesouro", o mundo (real) é usado como "mapa" e propõe-se ao jogador encontrar objetos ou locais em seu caminho. Um exemplo clássico é o jogo *Treasure Hunt* [27], em que jogadores buscam por tesouros no mundo, mas têm um tempo de espera antes de poderem efetivamente mover-se para uma nova localização e também podem colocar armadilhas em locais com tesouros para prejudicar outros jogadores. Este é um exemplo de mecânica que envolve não apenas objetos ou localização, mas também os aspectos temporal e social.

Jogos do tipo "Capture a Bandeira" permitem que jogadores formem times e cooperem entre si para conquistar alguma meta ou atingir algum objetivo antes dos outros times. Por exemplo, no jogo *Capture the Flag* [15], de temática medieval, dois times diferentes devem proteger seus respectivos castelos. As únicas entidades reais são as bandeiras dos times

– representadas por caixas de madeira dotadas de dispositivos internos com comunicação *bluetooth* –, e as pessoas, que assumem o papel de cavaleiros. No mundo virtual, existem entidades, chamadas de guias, que auxiliam os cavaleiros criando armadilhas para o time adversário e orientando seu próprio time para evitá-las. Os jogadores podem usar itens e magias para atacar adversários e devem trabalhar juntos para conquistar o castelo inimigo.

Um outro estilo de jogo são os "*Runners*" (Jogos de Corrida), em que o jogador é convidado a percorrer um caminho seguindo instruções dadas pelo jogo. Em *Zombies, Run!* [47], por exemplo, simula-se uma horda de *zombies* perseguindo o jogador pelo mundo (real), enquanto ele recebe instruções em áudio do gênero "à sua esquerda existe um abrigo, vá para lá". O jogo foi idealizado para ser utilizado em rotinas de *cooper* e utiliza as ruas da cidade e caminhos já existentes para gerar a simulação.

Um gênero bastante popular de jogos ubíquos são os ARGs (*Alternate Reality Games* ou Jogos de Realidade Alternativa), que propõem uma ruptura com a realidade – o mundo do jogo é visto como um mundo paralelo desconhecido das pessoas comuns, com suas próprias regras e narrativa e exclusivo para jogadores. Por exemplo, no jogo *Ingress* [19], os jogadores podem escolher associar-se a uma de duas facções. Em vários locais (físicos) espalhados pelo planeta existem portais, que são dominados por uma das facções e devem ser defendidos. Jogadores devem dirigir-se aos locais onde estão os portais e podem utilizar itens ou realizar ações para proteger os portais de sua facção ou atacar os portais da facção adversária, com o intuito de conquistá-lo para si. Esse estilo de jogo tipicamente também utiliza o conceito de *augmented reality* (realidade aumentada), em que um dispositivo computacional integra novos elementos virtuais que são visualizados em conjunto com o mundo real. No caso de *Ingress*, só é possível visualizar os portais na tela do *smartphone*, que os projeta sobre o mapa do mundo real.

Um outro ARG famoso é *Geocaching* [21], em que há também uma mistura com caça ao tesouro, já que itens (reais) são espalhados pelo mundo, mas somente os jogadores têm consciência de sua existência e devem decifrar pistas para encontrá-los.

Pode-se citar, ainda, os jogos de "Ativação" em que objetos do dia a dia recebem novas capacidades tecnológicas ou são adaptados para criar ou integrar mecânicas, que podem inclusive ser espontâneas, criadas pelos próprios jogadores. Bekker et al., por exemplo, criaram um dispositivo com *feedback* (luz e som) dependente da interação (mudança de orientação, agito, etc) e reuniram grupos de crianças, que desenvolveram suas próprias brincadeiras e jogos com base no dispositivo [5].

2.3 Jogos e Reabilitação

A Reabilitação é o ramo das ciências da saúde que se preocupa com a restauração das habilidades de pessoas com algum tipo de deficiência ou que sofreram algum trauma às suas capacidades máximas – seja no aspecto físico, psico-motor, psicológico, emocional ou profissional – com fins de reintegração social [48].

A terapia de reabilitação muitas vezes se caracteriza por um processo longo e contínuo, no qual a motivação do paciente é considerada fator crucial tanto para a efetividade do tratamento quanto para a velocidade da recuperação [26]. Este processo deve ser sempre acompanhado por profissionais de saúde qualificados, porém, muitas vezes beneficia-se ou mesmo depende do treino particular, praticado pelo paciente por si só, nos tempos livres e fora do ambiente controlado de uma clínica ou centro especializado.

Diferentes estudos mostram que jogos eletrônicos podem exercer um efeito positivo em áreas que requerem dedicação e disciplina por parte do usuário, tais como educação [3, 34], treinamentos técnicos [43] e na área de saúde em geral [4, 35]. Esses jogos incorporam a diferentes contextos aqueles fatores de diversão e imersão típicos das mídias de entretenimento, procurando aumentar o engajamento e gerar resultados mais efetivos.

Jogos ubíquos, portanto, são particularmente adequados para aumentar a motivação em terapias de reabilitação, já que não apenas propõem novas mecânicas com foco em imersão e engajamento, mas também aplicam os princípios da computação ubíqua para misturar o mundo real e o virtual, permitindo que a sessão de terapia envolva os vários ambientes em que o paciente pode estar inserido – clínica, rua, casa, academia, etc – e as várias pessoas que podem estar envolvidas no processo – demais pacientes, familiares e profissionais da saúde. No entanto, a criação de jogos ubíquos com fins de tratamento de saúde traz uma enorme gama de desafios para os projetistas.

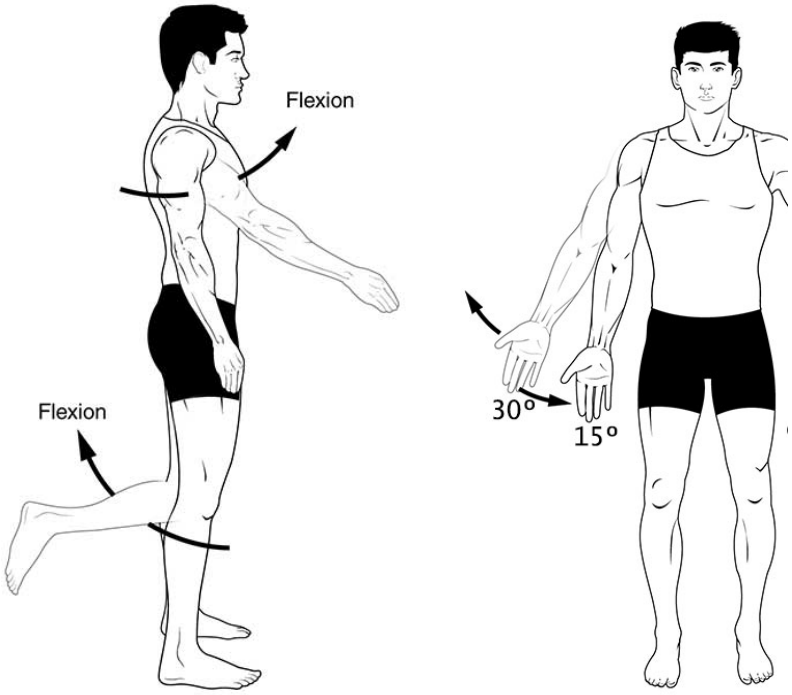
O primeiro ponto a ser considerado reside no fato de que *ubigames* são, naturalmente, aplicações ubíquas. Isso implica a necessidade de lidar com todos os desafios básicos de infraestrutura para este tipo de aplicação, tais como: heterogeneidade, escalabilidade, tolerância a falhas, adaptabilidade e segurança.

Além disso, o *design* do jogo deve ser inovador, buscando empregar adequadamente e tirar o máximo proveito dos vários dispositivos e recursos que podem estar presentes em diferentes ambientes inteligentes, buscando sempre combinar elementos dos mundos real e virtual e aumentar a imersão e a diversão dos jogadores.

Devem, também, levar em consideração os aspectos de saúde e de necessidades dos pacientes. Um mecanismo de entrada do jogo não pode agravar lesões ou causar novos traumas. Os jogos devem ser flexíveis e levar em conta as limitações e características individuais de cada paciente.

Por exemplo, em uma mesma clínica, há uma gama bastante diversa de pessoas procurando reabilitar-se de diferentes lesões ou outros problemas de saúde (Figura 2.1 (a)). Enquanto um paciente trabalha os membros superiores, um outro tem dificuldade de movimento nos membros inferiores. Um terceiro busca maior destreza em uma das mãos, ao mesmo tempo em que um quarto trata uma lesão na coluna.

Mesmo entre aqueles que possuem lesões similares, existem níveis diferentes de capacidade de movimentação e de amplitude em cada ação (Figura 2.1 (b)). Uma pessoa pode ter dificuldade ao mover o braço frontalmente, por exemplo, enquanto outra não consegue fazê-lo lateralmente. Além disso, enquanto um paciente pode ser capaz de abrir o braço 60º, por exemplo, um outro pode conseguir apenas 40º.



(a) partes do corpo diferente (b) margens de ação diferentes

Figura 2.1: Ilustração das limitações individuais de cada paciente (adaptado de [1]).

Para ser flexível, portanto, um jogo voltado ao propósito de reabilitação deve observar questões como gravidade das lesões, capacidade e amplitude dos movimentos e nível de progresso no tratamento. Também não pode ser restritivo em relação ao tipo de entrada esperado: pacientes com diferentes tipos de traumas ou com limitações em partes do corpo diversas devem ser capazes de jogar o mesmo jogo, ou até mesmo interagirem dentro dele.

Enquanto aplicações ubíquas, em geral, e jogos ubíquos, em particular, devem ter *adaptabilidade ao ambiente* – ou seja, devem alterar seu comportamento de acordo com a variedade de dispositivos e canais de comunicação, bem como com constantes mudan-

ças no ambiente dinâmico do *smartspace* – jogos ubíquos voltados à reabilitação devem, adicionalmente, ter *adaptabilidade ao paciente*, *i.e.*, atender corretamente aos diferentes objetivos com a terapia, suportando as variadas necessidades e limitações individuais, conforme exemplificado acima.

Além de todos esses aspectos considerados, deve-se ainda criar métricas para o progresso dentro do jogo que sejam compatíveis com o estado do tratamento, e também deve ser possível coletar metadados sobre as sessões que permitam o acompanhamento do próprio progresso pelo paciente e fomentem decisões sobre intervenções e melhorias por parte do profissional da saúde.

Há, portanto, a necessidade de uma solução que permita lidar com todos estes problemas de adaptação das entradas dos jogos aos diversos pacientes, além de viabilizar a construção de novos jogos ubíquos para reabilitação, com possibilidade de acompanhamento do tratamento pelos pacientes e profissionais da saúde.

2.4 Desafios

Após a análise feita nas seções anteriores de todos esses aspectos relativos aos temas da computação ubíqua, jogos ubíquos e reabilitação, é possível, portanto, sumarizar os seguintes desafios para o desenvolvimento de jogos ubíquos para reabilitação:

- são aplicações ubíquas, e, portanto, devem tratar dos desafios já inerentes a este tipo de sistema, tais como heterogeneidade de dispositivos e canais de comunicação, escalabilidade, tolerância a falhas, adaptabilidade, proatividade e transparência, cuidadoso projeto de interfaces e mecanismos de entrada e saída, além de privacidade e segurança;
- como todos os jogos ubíquos, devem trazer mecânica inovadora, que misture elementos diversos do mundo real com o mundo do jogo, possivelmente envolvendo variáveis como localização, tempo, condições ambientes, objetos e pessoas; devem especialmente, no entanto, engajar os jogadores para manter a disciplina ao longo do tratamento, envolvendo também outros pacientes, profissionais da saúde, amigos e familiares;
- devem ser flexíveis, observando as limitações e características individuais dos jogadores, a gravidade das lesões, a capacidade e amplitude dos movimentos e o nível de progresso no tratamento, tudo isso sem prejudicar a saúde ou causar novos traumas;
- não podem ser restritivos em relação aos tipos de entrada, permitindo que pacientes com diferentes tipos de lesão ou limitações em partes diferentes do corpo possam jogar o mesmo jogo ou interagirem dentro dele;

- devem fornecer métricas de progresso compatíveis com o estado do tratamento;
- devem permitir a coleta de metadados sobre as sessões e o acompanhamento tanto pelo paciente quanto pelo profissional da saúde.

2.5 Objetivos

Nas seções anteriores foram identificados os principais desafios, gerais e específicos, encontrados quando da construção ou utilização de jogos ubíquos para fins de reabilitação. De fato, verificou-se que essas aplicações, por serem ubíquas, e, mais especificamente, jogos ubíquos, devem lidar com os desafios inerentes a essas áreas.

Seu emprego no contexto de reabilitação, no entanto, traz ainda como questões adicionais aquelas relacionadas à saúde – tanto do ponto de vista de segurança quanto de monitoramento – e também, em especial, à *adaptabilidade ao paciente*, *i.e.*, ao tratamento da entrada para diferentes tipos de deficiências e capacidades/amplitudes de movimentação.

Sob esse foco, o objetivo geral desta pesquisa é disponibilizar uma nova solução que permita a construção e/ou adaptação de jogos e aplicações ubíquas para reabilitação, em especial no que tange à *adaptabilidade ao paciente*.

Para atingir a este objetivo geral, busca-se especificamente:

- definir um mecanismo genérico que permita a pessoas diferentes, com limitações e necessidades de reabilitação diferentes, potencialmente utilizando diferentes dispositivos de entrada, utilizar o mesmo jogo;
- definir uma interface padronizada para os mecanismos de entrada e saída em aplicações voltadas à reabilitação;
- fornecer métodos para converter entradas heterogêneas em valores normalizados que possam ser processados pela aplicação, de tal maneira que esta não necessite lidar diretamente com todas estas entradas;
- criar mecanismos para que diferentes dispositivos de entrada e saída possam ser personalizados por paciente, atendendo às suas necessidades individuais;
- viabilizar a coleta de metadados relacionados à saúde e às sessões para acompanhamento do paciente e dos profissionais de saúde.

No próximo capítulo, serão elicitados os trabalhos que já abordaram todos ou parte dos desafios levantados anteriormente, bem como as limitações e restrições ainda existentes e que são foco desta pesquisa.

Capítulo 3

Trabalhos Correlatos

Neste capítulo, são listados alguns trabalhos relevantes que abordaram os temas de computação ubíqua, jogos ubíquos e jogos para reabilitação, bem como as limitações encontradas nas soluções que eles propõem.

3.1 Computação Ubíqua e *Ubigames*

Diversos trabalhos já foram realizados para abordar os desafios elementares da computação ubíqua [2, 32, 41, 49], em especial a modelagem de ambientes inteligentes e a integração de vários dispositivos heterogêneos, de tal maneira que a infraestrutura básica de comunicação para sistemas ubíquos já possui modelos de referência.

Estes trabalhos, no entanto, focaram-se em criar mecanismos genéricos para a criação de sistemas ubíquos em geral, sem a preocupação em atacar as necessidades particulares de jogos ubíquos e, menos ainda, de jogos para reabilitação.

Trabalhos posteriores concentraram-se mais especificamente em jogos ubíquos, propondo soluções para acelerar seu desenvolvimento com a utilização de *game engines* específicas [8, 37]. Nenhum desses projetos, no entanto, permite tratar diretamente os problemas intrínsecos à terapia de reabilitação, não havendo ainda mecanismos para facilitar a adaptação do jogo aos diversos métodos de entrada e às características individuais dos pacientes, além da coleta de metadados sobre saúde.

3.2 Reabilitação

No contexto de jogos eletrônicos em geral, diferentes trabalhos já foram publicados com foco no tema da saúde, destacando-se os voltados à prática de atividades físicas [28, 38]. Dentre os jogos que abordaram reabilitação [17, 36, 46], no entanto, há poucos jogos ubíquos, de tal modo que o potencial desse gênero ainda permanece pouco explorado.

Além disso, pela complexidade do tema, estes trabalhos procuraram focar-se em membros específicos ou grupos específicos de membros – *e.g.* membros superiores [17] – sem a preocupação com a construção de mecanismos genéricos de mapeamento ou conversão de diferentes movimentos, com diferentes amplitudes e limitações, ao controle do jogo.

Dentre os trabalhos que aproximaram-se dos jogos ubíquos, pode-se citar o de Gotsis *et al.*, que criaram uma plataforma para prototipação rápida de jogos de realidade misturada para reabilitação [20], todavia, ela requer *hardware* e *sofware* específicos e é voltada apenas à reabilitação de membros superiores, não sendo possível a extensão para reabilitação de outras partes do corpo ou a construção de jogos ubíquos em geral, com diferentes dispositivos.

O problema de detectar movimentos corporais humanos, em geral, já foi resolvido de várias maneiras. Por exemplo, as chamadas IMUs (*Inertial Measurement Units*) [33] – que são sensores de rotação proprioceptivos, *i.e.*, capazes de medir a própria rotação – já foram empregados para captura de movimentos [31]. Em sistemas recentes dedicados para este fim, pode-se destacar o Kinect [29], da Microsoft, que é um sensor exteroceptivo, ou seja, mede o movimento analisando o ambiente.

Sendo um sensor de propósito geral, o Kinect já foi empregado, por exemplo, em um sistema que permite a definição de poses e movimentos com diferentes partes do corpo e sua tradução para o *input* de um jogo qualquer, inclusive daqueles que originalmente não suportam esse tipo de dispositivo de entrada [50]. Um outro trabalho utilizou sensores de maior precisão para construir um sistema próprio de captura, realizando também uma comparação com Kinect [44]. Em ambos os casos, no entanto, o foco está em sensores específicos e não há integração com plataformas para construção de aplicações ou jogos ubíquos.

No âmbito da robótica aplicada à reabilitação, encontra-se, por exemplo, o sistema [14] proposto por Caurin *et al.*, que adapta o nível de ajuda ao usuário com base na motivação relatada entre sessões. Esse sistema empregou uma rede neural para adaptar o nível de intervenção a cada usuário, focando-se apenas na variação da motivação, além de usar um mecanismo particular de *input* (movimento do pulso), sendo, portanto, de uso específico.

Na observação de aplicações práticas em clínicas especializadas em reabilitação, como, por exemplo, a Rede Sarah[1] de Brasília, foram relatadas pelos profissionais de saúde experiências de sucesso na utilização de títulos comerciais, por exemplo, Wii Sports[2], como ferramentas complementares ao processo terapêutico.

[1] `http://www.sarah.br/` (acessado em Mar/2016)
[2] `http://wiisports.nintendo.com/` (acessado em Mar/2016)

De acordo com estes relatos, porém, embora tenha sido observado um aumento no engajamento e na motivação dos pacientes que utilizaram os jogos, não foi possível aplicar a metodologia a uma grande parte dos casos e, nos casos em que foi possível, verificou-se uma série de limitações, pelos seguintes fatores:

- sendo títulos comerciais para o público geral, esses jogos não foram projetados para jogadores com limitações de movimentos ou com o fim específico de reabilitação em mente, de tal maneira que apenas os pacientes com limitações nos movimentos específicos esperados como entrada pelo jogo poderiam tirar proveito da experiência – por exemplo, um jogo que detecta apenas o movimento dos braços não poderia ser utilizado por um paciente com necessidade de reabilitação do joelho, ou um jogo que espera movimentos verticais não seria efetivo para um paciente que deseja reabilitar um movimento horizontal;
- os jogos esperam a mesma capacidade de movimentação média de todos os jogadores, não sendo possível ajustar as entradas esperadas pelo jogo a casos específicos, ou seja, calibrar a amplitude do movimento de acordo com a desenvoltura de cada paciente;
- as métricas de progresso dentro dos jogos estão relacionadas apenas às mecânicas projetadas em cada um deles, não refletindo corretamente o progresso no tratamento de reabilitação;
- os jogos não provêem meios de coletar dados sobre as sessões, tais como tempo médio, tipos de exercícios realizados, taxa de progresso, amplitudes de movimento atingidas, sinais vitais, etc...

A análise de casos concretos utilizando títulos comerciais, portanto, mostrou o problema imediato de que, não tendo sido projetados para jogadores com limitações de movimentos ou com o fim específico de reabilitação em mente, estes jogos também estão limitados a uma parcela diminuta dos pacientes. Eles empregam as mesmas métricas para todos os jogadores, não necessariamente relacionadas ao progresso no tratamento de reabilitação. Além disso, não proveem meios de coletar dados sobre as sessões.

3.3 *Middleware* uOS

Dentre os trabalhos analisados que se focaram na criação de soluções para jogos ubíquos, pode-se destacar o projeto uOS, que propõe um novo *middleware* [3] para enfrentar alguns dos problemas encontrados em soluções anteriores [11].

[3] Camada de abstração de *software* que permite o desenvolvimento e execução de uma mesma aplicação em múltiplas plataformas heterogêneas.

O trabalho trata, em especial, das questões de heterogeneidade de dispositivos e canais de comunicação e de adaptabilidade e proatividade, além de permitir a implantação de mecanismos para privacidade e segurança. O *framework* é baseado em uma extensão do modelo de arquitetura SOA [45], a DSOA [9] (*Device Service Oriented Architecture*), especialmente projetada para modelar ambientes inteligentes, na forma de dispositivos, recursos e serviços.

Com base neste paradigma e em um novo protocolo leve de comunicação, o uP [13], o uOS permite às aplicações ubíquas publicar e consultar dispositivos, recursos e serviços no ambiente, por meio de uma camada de *software* transparente e independente de plataforma.

O sistema permite também que seja definida uma hierarquia de equivalência entre recursos, bem como ontologias sobre eles. Dessa forma é possível aplicar heurísticas e selecionar as melhores opções de recursos para a realização de tarefas em diferentes contextos.

A solução foi projetada desde o princípio com suporte a *plugins*, facilitando a introdução de novas funcionalidades.

O suporte do uOS à construção de jogos ubíquos [8] permitiu o desenvolvimento de uma nova *game engine* com este fim específico [37]. Além disso, também foram implementados mecanismos de compatibilidade entre *engines* existentes e o uOS [42], ampliando o rol de ferramentas que os projetistas podem empregar para construir novos jogos.

O grupo de pesquisa fez também uma investigação mais profunda dos jogos ubíquos dos pontos de vista de *design* e epistemológico [12], propondo uma nova classificação dos jogos eletrônicos de acordo com sua adaptabilidade a diferentes plataformas, ambientes e métodos de interação. Essa pesquisa resultou também na definição de um conjunto de referência dos mecanismos de entrada e de suas hierarquias de equivalência, facilitando o projeto de novos jogos.

Uma vez que este trabalho abordou as limitações encontradas nos trabalhos anteriores e trouxe soluções para os principais desafios na construção de jogos ubíquos e, naturalmente, de aplicações ubíquas em geral, optou-se por utilizá-lo como base para desenvolvimento dessa pesquisa, propondo-se uma extensão do *middleware* para abordar as questões específicas dos jogos voltados à Reabilitação, isto é, a *adaptabilidade ao paciente* (Seção 2.3).

Nos próximos capítulos, essa extensão e sua implementação são descritas em detalhes.

Capítulo 4

Infraestrutura Básica

Para atingir os objetivos propostos (Seção 2.5), foi concebida e implementada uma extensão ao *middleware* uOS, visto que ele atende às necessidades encontradas na construção de aplicações e jogos ubíquos em geral, e pode ser ampliado para abordar os novos desafios específicos da terapia de reabilitação.

O principal objetivo a ser atingido é permitir que novas aplicações (ou antigas, adaptadas) possam receber *inputs* de várias formas diferentes – *i.e.* de movimentos diversos, realizados com diferentes partes do corpo, em variadas amplitudes – sem precisar suportar diretamente cada um desses casos, mas abstraindo-os. Foram, portanto, definidos mecanismos para que cada aplicação declare sua interface (de jogo) uma única vez, e todos os mapeamentos e conversões de movimentos para o formato esperado pela interface sejam realizados por uma biblioteca, que abstrai essa complexidade, tornando-a transparente para as aplicações.

Esse modelo requer que o *smartspace* onde serão executados os jogos seja organizado de uma nova maneira. Foi necessário criar a figura de um nó que agrega as informações sobre cada usuário, ou seja, seus *perfis*, e as torna disponíveis no momento em que um usuário é "conectado" a uma aplicação. Este chamado *nó coordenador* também agrega os metadados sobre saúde, permitindo seu registro e consulta posterior.

Todos esses mecanismos compõem uma camada de abstração que permite a utilização do mesmo jogo por diversos usuários, realizando diversos tipos de movimento. Para viabilizar a utilização do sistema em ambientes reais, no entanto, foi necessária a criação de ferramentas adicionais, que facilitam a definição e edição de perfis para os usuários e a configuração do ambiente.

Neste capítulo, é inicialmente apresentada uma visão geral da infraestrutura básica dos uOS, bem como dos mecanismos pelos quais ele pode ser estendido, seguida por uma descrição detalhada da extensão proposta, incluindo as novas abstrações criadas para atender ao problema da *adaptabilidade ao paciente*.

No capítulo seguinte, são descritas as ferramentas adicionais disponibilizadas para facilitar a configuração e o uso desta solução.

4.1 Visão geral do uOS

É apresentada a seguir uma visão superficial da estrutura do *middleware* uOS, para fins de contextualização da extensão proposta. Uma descrição exaustiva do *middleware* e da arquitetura e protocolo envolvidos pode ser encontrada nas respectivas publicações [8, 9, 11, 13].

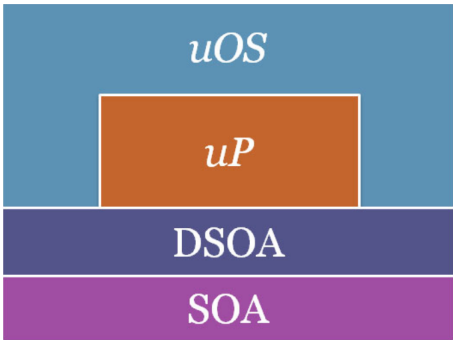


Figura 4.1: Representação do *middleware* uOS (retirado de [11]).

A Figura 4.1 traz uma visão de alto nível do uOS. Conforme descrito anteriormente (Seção 3.3), ele é baseado em um novo modelo de arquitetura, DSOA [9], que é por sua vez uma extensão da arquitetura SOA [45]. Na modelagem da infraestrutura proposta pela DSOA, o *smartspace* é composto por *pessoas* e *dispositivos*.

Um **dispositivo** é qualquer elemento do ambiente com capacidade computacional e de comunicação. Cada dispositivo pode conter um ou mais **recursos**, que são grupos de funcionalidades logicamente relacionadas, tais como os elementos acessíveis de *hardware* – câmera, teclado, mouse, sensores, etc – ou mesmo conceitos mais abstratos, como informações de usuários ou armazenamento de dados.

Os recursos, por sua vez, disponibilizam um conjunto de **serviços**. Cada serviço implementa uma funcionalidade específica, por meio de uma interface pública e bem definida. O recurso "câmera", por exemplo, poderia prover os serviços "tirar foto" ou "iniciar gravação".

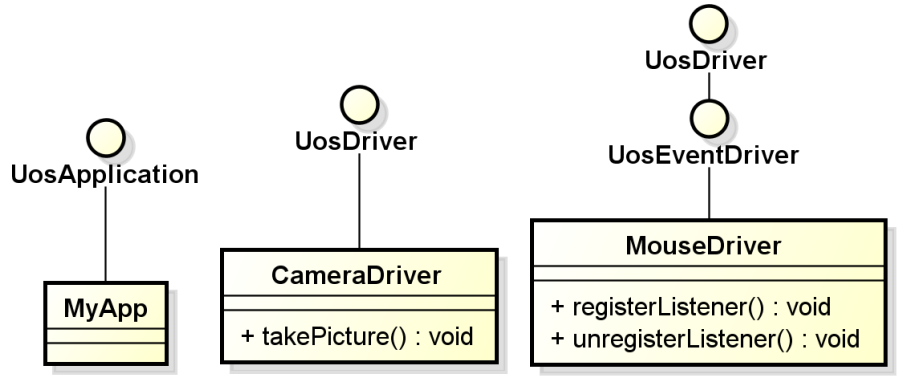
Conforme ilustrado na figura, o *middleware* é construído em conformidade com a DSOA e internamente utiliza o uP [13], um novo protocolo de comunicação leve.

Nesse protocolo, as mensagens são transmitidas em JSON[1], um formato padronizado para serialização de dados em modo texto, amplamente utilizado para troca de informa-

[1] `http://www.json.org` (acessado em Mar/2016).

ções entre sistemas. Isso permite que o *middleware* seja agnóstico em relação à implementação, *i.e.*, qualquer aplicação capaz de formatar suas requisições e processar respostas nesse padrão pode se comunicar com o ambiente de maneira transparente.

A implementação[2] de referência do uOS é um *framework*[3] escrito em Java[4] que modela as abstrações da DSOA, bem como as entidades básicas e serviços do uP, por meio de um conjunto de classes e interfaces. São disponibilizados, ainda, *plugins*[5] para utilização das tecnologias de comunicação mais comuns, tais como TCP/UDP [39, 40], *Bluetooth* [25] e *WebSockets* [18].



(a) aplicação (b) *driver* geral (c) *driver* com eventos

Figura 4.2: Diagrama das interfaces básicas do uOS.

Um desenvolvedor que deseje criar uma nova aplicação deve implementar a interface `UosApplication`, conforme ilustrado na Figura 4.2(a). Essa interface declara os métodos acionados pelo *middleware* durante o ciclo de vida da aplicação.

O componente de *software* que efetivamente controla um recurso é chamado de ***driver***. Para criar um novo *driver*, o desenvolvedor deve prover uma classe que implemente a interface `UosDriver`. Os serviços desse recurso são implementados por meio de métodos da classe criada, acionados via reflexão[6]. Em outras palavras, se o recurso "câmera" é implementado pela classe `CameraDriver` (filha de `UosDriver`), por exemplo, o serviço "tirar foto" poderia ser o método `takePicture` dessa mesma clase. Quando o *middleware* recebe uma mensagem solicitando a chamada a um serviço de um dado recurso, ele busca, via reflexão, por um método no respectivo driver que tenha o mesmo nome daquele serviço. A Figura 4.2(b) ilustra essa hierarquia.

[2] `https://github.com/UnBiquitous/uos_core` (acessado em Mar/2016).

[3] Conjunto de componentes para construção de software que impõem um modelo particular a ser seguido e estendido pelas aplicações que o utilizam.

[4] `https://www.java.com` (acessado em Mar/2016).

[5] `https://github.com/UnBiquitous` (acessado em Mar/2016).

[6] Técnica de programação em que os métodos e atributos de uma classe ou objeto são consultados e/ou manipulados dinamicamente, em tempo de execução.

Um caso particular é modelado pela interface `UosEventDriver`, que deve ser implementada por todo *event driver* (Figura 4.2(c)). A especificidade desse tipo de *driver* é que ele sempre possui os serviços `registerListener` e `unregisterListener`, que permitem a observadores interessados registrar-se e desregistrar-se para receberem **eventos**. Enquanto serviços são funcionalidades acionadas de maneira deliberada e *síncrona*, eventos são notificações *assíncronas* geradas pelo *driver* e enviadas a todos os subscritos no momento em que ocorrem.

Cada serviço ou evento tem um **nome** único, uma **funcionalidade** bem definida, e (potencialmente) **parâmetros**, que podem ser **opcionais** ou **obrigatórios**. Os valores dos parâmetros, mapeados por seu nome, são enviados na mensagem quando uma chamada de serviço ou notificação de evento ocorre.

O uOS traz um *driver* padrão embutido, o `DeviceDriver`, responsável por atender a serviços de descoberta e *handshake*[7] entre dispositivos.

Todas as mensagens trocadas no *smartspace* obedecem às regras do uP, e empregam um conjunto básico de entidades para descrever as requisições e respectivas respostas. Essas entidades são descritas no *middleware* por meio de *beans*[8]. Um dispositivo é descrito por uma instância de `UpDevice` e um recurso, por um objeto `UpDriver`. Além disso, dados sobre serviços são armazenados em instâncias de `UpService` e as interfaces de comunicação são descritas em objetos `UpNetworkInterface`. Essas classes são internamente serializadas em JSON quando da construção das mensagens.

Do ponto de vista do programador, todas as interações com o *middleware* são realizadas por meio de uma instância de `Gateway`. Essa *façade*[9] provê métodos para listar os dispositivos e recursos conhecidos; adicionar *drivers* dinamicamente; registrar-se, desregistrar-se e propagar notificações de eventos; além de realizar chamadas de serviços.

As chamadas de serviços e notificações de eventos processadas pelo *gateway* são descritas pelas *beans* `Call` e `Notify`, respectivamente. A resposta é descrita na forma de uma `Response`. Todas essas mensagen carregam parâmetros que tipicamente são instâncias de `UpDevice`, `UpDriver`, `UpService`, etc...

O *middleware* como um todo é modelado pela classe `UOS`. Para iniciar sua execução, é necessário criar uma nova instância desta classe e chamar seu método `start`, que recebe como parâmetro um mapa com um conjunto de configurações na forma de pares $<$*chave*, *valor*$>$.

[7] Operação comum em protocolos de comunicação em que duas entidades se indentificam uma à outra e trocam informações básicas sobre si.

[8] Objeto simples, com apenas um conjunto de atributos e respectivos métodos `get` e `set`, cuja única função é o armazenamento e troca de dados.

[9] Classe que agrega as interfaces de vários componentes diferentes em uma visão única.

Diversas classes auxiliares foram disponibilizadas para permitir a imediata utilização de diferentes conjuntos pré-definidos de configurações ou para carregá-las a partir de um arquivo. Em qualquer um desses casos, é possível alterar ou incluir dinamicamente novos valores. Dentre as várias opções de configuração, pode-se definir a aplicação padrão e os *drivers* a serem automaticamente instanciados, bem como seus parâmetros; quais interfaces de comunicação estarão disponíveis; qual(is) o(s) método(s) para anunciar a presença e descobrir novos dispositivos; dentre outras. A Figura 4.3 traz um exemplo simples de execução do *middleware* e de uma chamada de serviço por meio do *gateway*.

Visto que o uOS foi implementado na forma de um *framework* e permite o carregamento dinâmico de componentes via reflexão, a criação de *plugins* é bastante simples. Basicamente, basta a disponibilização de uma biblioteca (arquivo *jar*) com as classes que estendem e/ou adicionam novas entidades ao *middleware*, *i.e.*, *drivers*, *beans*, aplicações, etc. O novo *plugin* pode então ser utilizado por aplicações diretamente, via referência a estas classes, ou dinamicamente, por meio da configuração de carregamento do *middleware*. No código ilustrado na Figura 4.3, por exemplo, é utilizado um novo *driver* chamado `IMUDriver`, por meio do mapa de configurações, que também recebe alguns parâmetros adicionais específicos para este *driver*.

```
import org.unbiquitous.unbihealth.imu.IMUDriver;
import org.unbiquitous.uos.core.InitialProperties;
import org.unbiquitous.uos.core.UOS;
import org.unbiquitous.uos.core.adaptabitilyEngine.Gateway;
import org.unbiquitous.uos.core.applicationManager.UosApplication;
import org.unbiquitous.uos.core.messageEngine.messages.Call;
import org.unbiquitous.uos.core.messageEngine.messages.Response;
import org.unbiquitous.uos.core.ontologyEngine.api.OntologyDeploy;
import org.unbiquitous.uos.core.ontologyEngine.api.OntologyStart;
import org.unbiquitous.uos.core.ontologyEngine.api.OntologyUndeploy;
import org.unbiquitous.uos.network.socket.radar.MulticastRadar;

public class DummyApplication implements UosApplication {
    public static void main(String[] args) throws Throwable {
        UOS uos = new UOS();

        InitialProperties settings = new MulticastRadar.Properties() {
            {
                addApplication(DummyApplication.class); // default application
                addDriver(IMUDriver.class, "imudriver42"); // requests instantiation of IMUDriver
                // Sets some parameters for IMUDriver.
                put(IMUDriver.DEFAULT_SENSOR_ID_KEY, "arm");
                put(IMUDriver.VALID_IDS_KEY, "forearm");
                put(IMUDriver.SENSITIVITY_KEY, 0.01);
            }
        };
        uos.start(settings);

        // Gets the gateway...
        Gateway gateway = uos.getGateway();

        // Calls a service on IMUDriver.
        Call call = new Call(IMUDriver.DRIVER_NAME, IMUDriver.LIST_IDS_NAME, "imudriver42");
        Response response = gateway.callService(null, call); // call on myself

        // Handles the service response.
        String[] ids = (String[]) response.getResponseData(IMUDriver.IDS_PARAM_NAME);
        for (String id : ids)
            System.out.println(id);

        uos.stop();
    }

    // UosApplication
    public void init(OntologyDeploy knowledgeBase, InitialProperties properties, String appId) {}

    public void start(Gateway gateway, OntologyStart ontology) {}

    public void stop() throws Exception {}

    public void tearDown(OntologyUndeploy ontology) throws Exception {}
}
```

Figura 4.3: Exemplo simples de código para execução do uOS.

4.2 Organizando o *smartspace*

Empregando-se apenas a infraestrutura básica do uOS, o cenário típico de um jogo ubíquo seria ilustrado pelo esquema da Figura 4.4. Cada jogo gerencia ativamente todos os possíveis dipositivos de entrada e todas as possíveis aplicações interessadas em seus metadados.

Nesse modelo, o jogo necessariamente precisaria ter "consciência" de seu propósito de saúde/reabilitação, visto que, não apenas uma boa parte dos mecanismos de E/S são específicos para este tipo de aplicação (*e.g.*, sensores, estimuladores, detectores de movimentos corporais, etc), mas também cada medida precisa ser adaptada ao propósito específico da terapia.

Suponha que o jogo fosse implementar um grau de adaptabilidade ao paciente tal que a maior parte dos pacientes de reabilitação pudessem utilizá-lo. Nesse caso, seria preciso definir explicitamente um grande conjunto de movimentos, utilizando partes do corpo diferentes, com amplitudes suportadas diferentes. Cada um desses movimentos deveria ter um conjunto de parâmetros a serem ajustados por usuário.

Não apenas a implementação do jogo seria trabalhosa, mas cada alteração futura para incluir novas entradas envolveria um esforço descomunal. Adicionalmente, este esforço seria restrito a um jogo em particular, a criação de novos jogos envolveria, no mínimo, o trabalho de adaptar os mecanismos de detecção.

Essa abordagem é possível, mas requer um significativo esforço de desenvolvimento e manutenção. O modelo rapidamente torna-se impraticável na medida em que cada nova mecânica exige o planejamento e implementação de um conjunto totalmente novo de parâmetros.

Para solucionar este problema, propõe-se aqui um esquema de mapeamento e conversão, ajustado de acordo com o perfil de cada paciente, entre os vários possíveis dispositivos de entrada e saída e todos os jogos ubíquos presentes no ambiente (Figura 4.5).

Cada jogo compatível deve apenas declarar sua interface única e padrão. A tarefa das rotinas de conversão é transformar um conjunto de entradas diversas no formato esperado pelo jogo. Essa conversão é única para cada conjunto de dispositivos de entrada e para cada paciente. Dessa maneira, o *perfil* de cada paciente é criado e ajustado em um único ponto, um *nó coordenador*. Duas aplicações quaisquer podem agora ser conectadas utilizando-se apenas a interface pública declarada e o perfil. Essa proposta retira a complexidade do suporte ao *input* de diferentes pessoas de dentro do jogo e a encapsula em uma biblioteca embutida na aplicação.

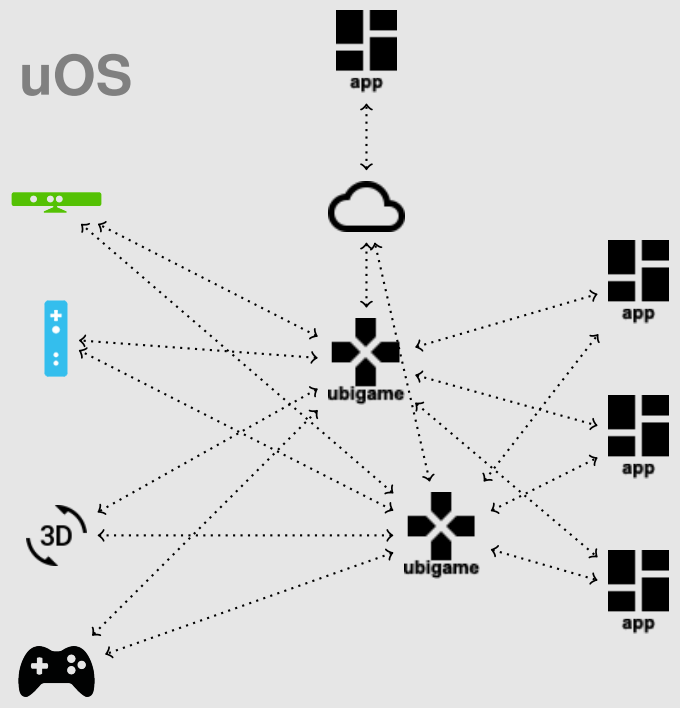


Figura 4.4: Modelo simples de *ubigame* para um ambiente uOS.

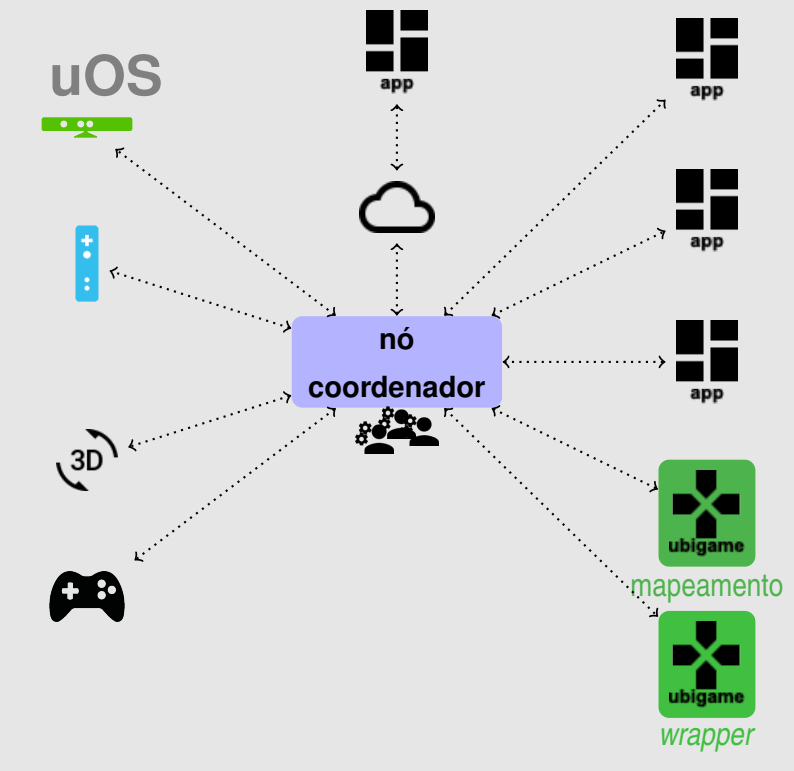


Figura 4.5: Mapeamento/conversão com base em perfis, concentrados em um nó coordenador.

Nesse modelo, uma aplicação ubíqua especializada, o *nó coordenador*, assume a responsabilidade por guardar os mapeamentos dos dados de entrada e saída dos múltiplos dispositivos e recursos no *smartspace*, com respeito a cada paciente, deixando para os jogos ubíquos apenas a tarefa de definir um formato padronizado para seus controles, liberando esforço de desenvolvimento e permitindo um foco maior na mecânica.

O nó coordenador mantém a base de dados dos vários usuários e das configurações específicas para cada um, parametrizadas pelo profissional da saúde responsável por aquele paciente.

A camada de adaptação, incorporada à aplicação que controla os dispositivos, concentra toda a complexidade de conversão e mapeamento dos vários dados de entrada/saída de/para os formatos esperados pelo jogo.

Com esta técnica, torna-se possível definir os parâmetros de um dispositivo qualquer para um paciente específico apenas uma vez, e, em seguida, usar essas configurações para jogos diferentes ou ainda para controlar diferentes "botões" do jogo. Por exemplo, o mesmo movimento de entrada poderia ser utilizado para mover uma personagem para frente em um caso ou para fazê-la pular, em outro, conforme o estado das associações realizadas no nó de mapeamento.

Uma vez que a aplicação centralizada tem acesso aos dispositivos e recursos do *smartspace*, ela pode também identificar as pessoas e associar a configuração correta para os vários jogadores, ou mesmo orquestrar modos colaborativos entre eles.

Como vantagem adicional, o nó coordenador pode expor uma interface para coletar e compilar dados para pacientes e profissionais da saúde. Nesse esquema, os jogos e/ou dispositivos fornecem dados sobre sensores e sessões, e estes dados ficam disponíveis para que outras aplicações ubíquas possam consultá-los e gerar relatórios, se necessário.

Nesse esquema, inclusive, qualquer jogo, independente de ser ubíquo ou voltado à saúde, poderia potencialmente ser utilizado como ferramenta complementar à reabilitação. Isso é possível com o desenvolvimento de *wrappers*, isto é, aplicações "envelope", ou de adaptação, que comunicam-se com a infraestrutura proposta respeitando suas interfaces de um lado, e com os jogos já existentes de outro, com suas particularidades e metodologias próprias de comunicação.

Criada esta ponte, abre-se a possibilidade de adaptar diversos jogos à infraestrutura já estabelecida.

4.3 Interfaces e Protocolos

Para permitir que jogos definam uma interface pública de entrada e saída, bem como realizem conversões corretamente entre os dados, foi necessária a criação de algumas

abstrações e de *drivers* adicionais, que implementam serviços com estas funcionalidades.

As seções a seguir descrevem esses elementos em detalhes.

4.3.1 Entidades

A primeira tarefa necessária para modelar o problema de suportar mecanismos de entrada e saída diversas foi a definição de algumas entidades, que são utilizadas pelo uHP (*Ubiquitous Health Protocol*). Este não é um protocolo no sentido formal do termo, mas um conjunto de serviços, modelados sob o uP e voltados ao fim específico de reabilitação. A Figura 4.6 traz uma visão geral dessas entidades e seu relacionamento. As mais elementares delas são **usuários** e **aplicações**.

Uma aplicação contém **pinos**, que são os pontos de conexão com o ambiente externo. Para um jogo, um pino é equivalente a um botão de ação ou eixo de um *joystick*, por exemplo. Todo pino tem um **tipo** de dado associado e um **modo de E/S**, além de outros atributos.

Quando é estabelecido um mapeamento entre os pinos de uma aplicação e um usuário, fica definido um **perfil**. Cada perfil pode ter zero ou mais **adaptadores**, cuja função é converter os dados vindos do ambiente externo no formato esperado pelo jogo.

Figura 4.6: Entidades do uHP.

A seguir, esses termos e atributos são definidos com maior precisão.

Tipo

Um **tipo** (`UhpType`) é uma descrição de como um dado deve ser interpretado, e é composto pelos seguintes atributos:

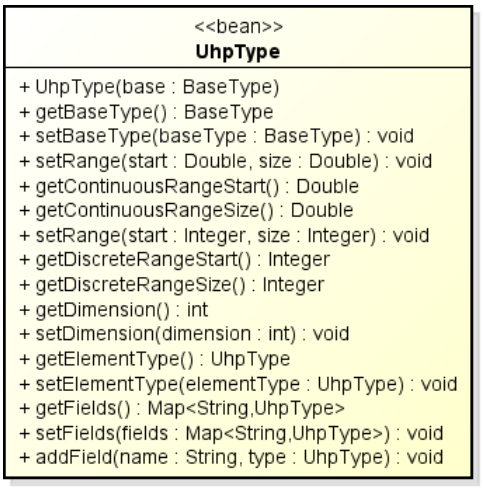
- **tipo elementar** (`baseType`) – descreve a categoria básica de um tipo, e pode assumir os valores: **discreto** (`DISCRETE`), para valores escalares inteiros; **contínuo** (`CONTINUOUS`), para valores escalares reais; **vetor** (`ARRAY`), para coleções de elementos de mesmo tipo; ou **estruturado** (`STRUCT`), para coleções de elementos de tipos diferentes;

- **intervalo** (`range`) (se discreto ou contínuo) – a faixa de valores em que os dados deste tipo podem assumir;
- **tipo de elemento** (`elementType`) (se vetor) – uma outra instância de tipo, que descreve o tipo de todos os elementos do vetor;
- **dimensão** (`dimension`) (se vetor) – o número de elementos no vetor;
- **campos** (`fields`) (se tipo estruturado) – um mapa de tuplas *<nome, tipo>* que nomeia e descreve cada um dos campos do tipo estruturado;

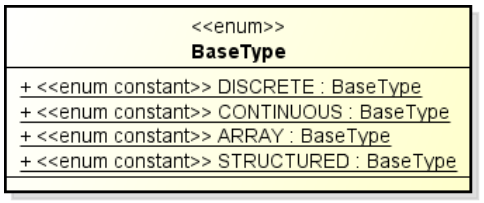
A Figura 4.7 traz uma representação da classe `UhpType` (a) e da enumeração aninhada `UhpType.BaseType` (b).

Por conveniência, alguns tipos básicos já são definidos por padrão, e podem ser referenciados apenas por um identificador, sem necessidade de serem descritos explicitamente:

- `bit` – discreto, no intervalo inteiro $[0, 1]$;
- `uniform` – contínuo, no intervalo real $[-1, 1)$;
- `v2` – vetor bidimensional de contínuo (elementos do espaço vetorial $\mathbb{R}^2$);
- `v3` – vetor tridimensional de contínuo (elementos do espaço vetorial $\mathbb{R}^3$).



(a) Classe `UhpType`.



(b) Enumeração `UhpType.BaseType`.

Figura 4.7: Representações das entidades relacionadas a um tipo.

Modo de E/S

O **modo de E/S** (`UhpPin.IOMode`) descreve a direção do fluxo de informação em relação a uma entidade, podendo ser, de **entrada** (`IN`), de **saída** (`OUT`) ou de **entrada e saída** (`INOUT`);

A Figura 4.8 traz uma representação da enumeração `UhpPin.IOMode`.

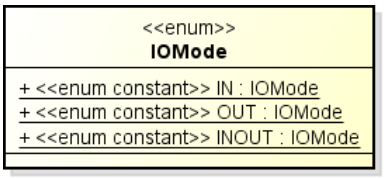


Figura 4.8: Representação da enumeração `UhpPin.IOMode`.

Pino

Conforme descrito anteriormente, um **pino** (`UhpPin`) é uma abstração para um ponto de entrada e/ou saída de uma aplicação de saúde. Pode ser usado para representar qualquer canal do jogo que permita o envio de comandos – por exemplo, um botão "pular", a teclas direcionais ou o *feedback* de vibração.

Um pino contém os seguintes atributos:

- **nome** (`name`) – um nome único (id) para o pino dentro do contexto da aplicação;
- **modo** (`mode`) – o modo de E/S do pino;
- **tipo** (`type`) – o tipo de dado que flui através do pino;
- [*opcional*] **mnemônico** (`mnemonic`) – um nome mnemônico para o pino;
- [*opcional*] **descrição** (`description`) – um comentário ou descrição para o pino.

A Figura 4.9 traz uma representação da classe `UhpPin`.

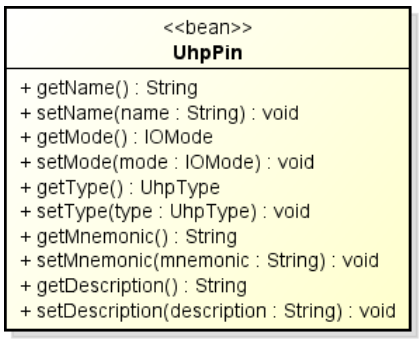


Figura 4.9: Representação da classe `UhpPin`.

Adaptador

Um **adaptador** (`UhpAdapter`) é um objeto responsável por converter um dado de um *tipo de origem* para um *tipo de destino*, usando algum método específico. Caso necessário, o

adaptador pode declarar **parâmetros** que configuram a conversão – um limiar, um fator de correção, uma opção, etc.

Um objeto adaptador deve implementar o método `convert`, que recebe um objeto genérico, interpretado de acordo com o tipo de origem, e produz um novo objeto no formato do tipo de destino. Adaptadores podem ser conectados em uma cadeia para tratar tipos que não possuem método de mapeamento direto, ou para controlar com maior precisão as etapas da conversão.

Uma adaptador contém os seguintes atributos:

- **identificador** (`id`) – um valor único que será referenciado pela aplicação;
- **tipo de origem** (`sourceType`) – o formato do dado de entrada;
- **tipo de destino** (`targetType`) – o formato do dado convertido;
- **parâmetros** (`params`) – um mapa dos parâmetros <*id*, *tipo*>, onde *id* é um identificador único (pode ser um índice inteiro) e *tipo* é o nome de um tipo interno ao adaptador.

A Figura 4.10 traz uma representação da interface `UhpAdapter`.

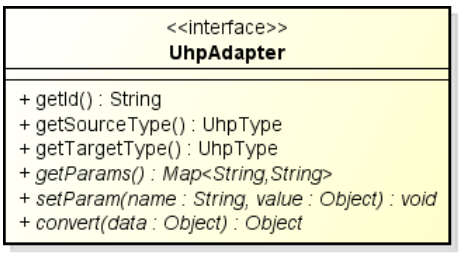


Figura 4.10: Representação da interface `UhpAdapter`.

Por conveniência, os seguintes adaptadores padrão são fornecidos:

- `intToReal`()[*discreto* → *contínuo*] – interpreta diretamente um valor discreto como contínuo;
- {`floor`, `ceil`, `round`}()[*contínuo* → *discreto*] – convertem valores contínuos em discretos usando, respectivamente, as operações matemáticas de chão, teto e arredondamento;
- `threshold`(`limit` : `double`)[*contínuo* → `bit`] – recebe como parâmetro um limiar e mapeia um tipo contínuo em um `bit`, todo valor acima do limiar torna-se 1, os demais tornam-se 0;

- `steps`([`0 : double`, ...])[*contínuo* → *discreto*] – recebe comos parâmetros uma lista crescente de limiares que definem patamares, dado um contínuo qualquer, indica o patamar (0 indica o intervalo < que o primeiro limiar), em que ele cai;
- `scale`(`factor : double`)[*contínuo* → *contínuo*] – recebe como parâmetro um fator de escala real e multiplica o valor de entrada por este fator;
- `getAt`(`index : int`)[*vetor* → *tipo do elemento básico*] – recebe como parâmetro um índice k (base 0) e retorna o elemento naquela posição;
- `magnitude`()[`v2, v3` → *contínuo*] – retorna a norma de um vetor;
- `normalizeVector`()[`v2, v3` → *resp.* `v2, v3`] – retorna um vetor normalizado;
- `normalizeScalar`()[*contínuo*, *discreto* → `uniform`] – retorna o valor normalizado pelo intervalo de origem, no intervalo real $[0, 1]$.

Aplicação de Saúde

Uma **aplicação de sáude** (`UhpHealthApp`) contém os dados de uma aplicação específica. A característica básica de uma aplicação é o conjunto de pinos que disponibiliza para conexão com o ambiente.

Uma aplicação contém os seguintes atributos:

- **nome** (`name`) – um nome que será referenciado pelo nó coordenador,
- **pinos** (`pins`) – o conjunto de pinos,
- [*opcional*] **ícone** (`mnemonic`) – um ícone para identificar a aplicação,
- [*opcional*] **descrição** (`description`) – um comentário ou descrição para a aplicação;

A Figura 4.11 traz uma representação da classe `UhpHealthApp`.

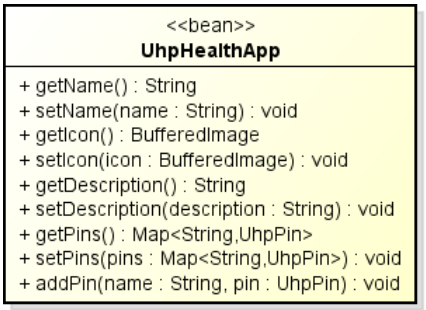


Figura 4.11: Representação da classe `UhpHealthApp`.

Perfil

Um **perfil** associa uma aplicação a uma pessoa. No nó coordenador, cada pessoa e cada aplicação estão registrados por seus identificadores. A cada par <*pessoa*, *aplicação*> pode ser associado um perfil, que será utilizado cada vez que uma conexão for realizada com a aplicação.

O elemento básico do perfil é o **mapeamento**. Cada mapeamento é um par <*pino*, *cadeia de conversão*>. A *cadeia de conversão* é simplesmente uma sequência de zero ou mais pares <*id do adaptador*, *mapa de parâmetros*>, ou seja, uma lista de adaptadores a serem aplicados em sequência, cada um com seus valores de parâmetros associados.

4.3.2 `PinDriver`

O *driver* que permite a uma aplicação declarar e manipular seus próprios pinos, bem como listar e realizar conexões com os pinos de outras aplicações é o `PinDriver`. Ele expõe duas interfaces, descritas nas Tabelas 4.1 e 4.2.

A interface externa é o conjunto de serviços e eventos publicados no *smartspace*. A interface interna é acessível apenas à própria aplicação e consiste em um conjunto de métodos públicos da classe `PinDriver`. É essa última interface que viabiliza ler e/ou alterar os dados dos pinos internamente, de acordo com mudanças em outras partes do sistema.

`PinDriver` - Serviços e Eventos	
Serviços	
`list` ()	retorna a lista de pinos atualmente disponíveis para conexão
`connect` (**`pin`** : `String`)	dado seu nome, ativa um pino para receber ou enviar dados junto ao chamador
`disconnect` (**`pin`** : `String`)	dado seu nome, desativa um pino previamente ativado
Eventos	
`update` (**`pin`** : `String`, **`value`** : `Object`)	notificação enviada quando o valor de um pino é atualizado (o `PinDriver` já se registra automaticamente para ouvir este evento quando é inicializado, de modo que notificações externas de mudanças em pinos de entrada sejam processadas adequadamente)

Tabela 4.1: Serviços e Eventos do `PinDriver`.

PinDriver - Interface Interna	
add (**pin** : UhpPin)	adiciona um novo pino
remove (**pin** : String)	remove um pino pelo nome
addPinListener (**pin** : String, **listener** : PinListener)	adiciona um novo observador de eventos no pino
removePinListener (**pin** : String, **listener** : PinListener)	remove um observador de eventos do pino
update (**pin** : String, **value** : Object)	notifica o *driver* sobre uma mudança no valor de um pino

Tabela 4.2: Interface Interna do `PinDriver`.

As aplicações que desejam publicar pinos para o ambiente e ouvir pelos eventos relacionados devem recuperar a instância interna do `PinDriver` quando da sua inicialização, conforme ilustrado na Figura 4.12.

```
/* ... */
public void start(Gateway gateway, OntologyStart ontology) {
    // Gets the local instance of PinDriver...
    pinDriver = null;
    for (UosDriver d : ((SmartSpaceGateway) gateway).getDriverManager().listDrivers())
        if (PinDriver.DRIVER_NAME.equals(d.getDriver().getName()) && (d instanceof PinDriver)) {
            pinDriver = (PinDriver) d;
            break;
        }
}
/* ... */
```

Figura 4.12: Recuperando uma instância do `PinDriver`.

Uma vez que tenha acesso a esta instância, a aplicação pode utilizar a interface interna do *driver* para adicionar ou alterar novos pinos (Figura 4.13), registrar-se para ouvir eventos dos pinos de entrada (Figura 4.14); ou gerar notificações de mudanças para os pinos de saída (Figura 4.15).

```
    /* ... */
    public void start(Gateway gateway, OntologyStart ontology) {
        /* ... */

        // Adds a pin named "punch" to the public interface.
        UhpPin pin = new UhpPin("punch");
        pin.setType(UhpType.bit);
        pin.setMode(UhpPin.IOMode.IN);
        pin.setMnemonic("punch");
        pin.setDescription("used to make the character punch");
        pinDriver.add(pin);
    }
    /* ... */
```

Figura 4.13: Adicionando um pino.

```
public class DummyApplication implements UosApplication, PinListener {
    /* ... */
    public void start(Gateway gateway, OntologyStart ontology) {
        /* ... */

        // Listens for events on the pin.
        pinDriver.addPinListener("punch", this);
    }

    // PinListener
    public void valueChanged(UhpPin pin, Object newValue) {
        System.out.print("pin '" + pin.getName() + "' new value: " + newValue);
    }
    /* ... */
}
```

Figura 4.14: Ouvindo por eventos em um pino de entrada.

```
    /* ... */
    private void internalProcess() {
        /* ... */

        // Changes the value of an output pin.
        pinDriver.pinValueChanged("vibration", 0.5);
    }
    /* ... */
```

Figura 4.15: Notificando eventos em um pino de saída.

4.3.3 `HealthAppDriver`

O `HealthAppDriver` é responsável por gerenciar os dados básicos da aplicação, comunicar-se ou assumir o papel de nó coordenador e manter o registro de conversões necessárias para cada pino.

Da mesma maneira que o `PinDriver`, ele expõe duas interfaces: uma junto ao *middleware* e outra interna, para a interação local. Elas estão descritas nas Tabelas 4.3 e 4.4.

O serviço `getApp` retorna um objeto `UhpHealthApp` serializado que descreve a aplicação de saúde e, portanto, serve ao duplo propósito de identificar se uma aplicação ubíqua no ambiente é compatível e, em caso positivo, de descrever seus atributos.

Uma vez que uma aplicação compatível tenha sido identificada, é possível utilizar o serviço `getCoordinator` para tentar descobrir quem é o nó coordenador no ambiente. Se a aplicação for o nó coordenador ou conhecê-lo, essa chamada de serviço retorna sua descrição na forma de um `UpDevice`.

Qualquer `HealthAppDriver` presente no ambiente pode atuar como nó coordenador, desde que seja configurado para isso. Se houver mais de um, eles terão bases de dados de usuários e perfis distintas, cabendo a cada aplicação definir critérios para selecionar seu coordenador adequado. O papel de nó coordenador é definido via configuração do *middleware*, pelas propriedades de carregamento do *driver*.

Dado que uma aplicação seja nó coordenador, uma chamada ao serviço `list` vai retornar todos os perfis armazenados naquele nó, sendo possível filtrá-los por usuário e/ou aplicação. Cada perfil contém os nomes do usuário e da aplicação, além de potencialmente uma cadeia de adaptadores. Os adaptadores são identificados por seu nome (ou classe) e podem conter um mapa com os valores dos parâmetros.

O serviço `connect` efetivamente faz o *driver* mudar seu estado para ficar conectado a uma dada aplicação, de acordo com o perfil especificado como parâmetro da chamada. Apenas uma conexão é possível por vez – a conexão anterior, se existir, é desfeita.

Uma vez que o `HealthAppDriver` esteja operando sob um determinado perfil, ele pode aplicar as conversões especificadas. Isso é feito automaticamente por meio do método `convert`, chamado para esse fim pelo `PinDriver` sempre que ocorrem atualizações nos estados dos pinos.

Para permitir a geração e coleta de metadados, são disponibilizados os serviços `logMD` e `queryMD` (no nó coordenador). Um registro interno é mantido para entradas de metadados idenficadas por uma chave. O formato de dados é específico de cada aplicação, ficando o `HealthAppDriver` responsável apenas por centralizar esta informação.

Por meio da interface interna exposta pelo *driver*, é possível criar, substituir e remover perfis, conforme ilustra a Figura 4.16.

```
/* ... */
private void internalInit() {
    /* ... */

    // Creates the profile map.
    Map<Object, Object> params = new HashMap<>();
    params.put("tolerance", 0.5);
    MutablePair<String, Map<Object, Object>> adapter = new
         MutablePair<>("org.unbiquitous.unbihealth.avatar.AvatarDriver", params);
    List<Pair<String, Map<Object, Object>> chain = new ArrayList<>();
    chain.add(adapter);
    Map<String, List<Pair<String, Map<Object, Object>>> profile = new HashMap<>();
    profile.put("punch", chain);

    // Adds (or updates) the new profile.
    healthAppDriver.putProfile("Luciano", "fisiogame", profile);

    // Removes the profile.
    healthAppDriver.removeProfile("Luciano", "fisiogame");
}
/* ... */
```

Figura 4.16: Manipulando perfis.

HealthAppDriver - Serviços	
getApp ()	retorna um objeto `UhpHealthApp` que descreve a aplicação atual
getCoordinator ()	retorna o `UpDevice` correspondente, caso conheça ou seja o coordenador no *smartspace* atual; do contrário, não retorna nenhum dado
list ([**user** : `String`], [**app** : `String`])	retorna a lista dos perfis armazenados no nó coordenador, opcionalmente filtrada por usuário e/ou aplicação
connect (**device** : `UpDevice`, **profile** : `Map<String, List<Pair<String, Map<Object, Object>>>>`)	conecta a aplicação corrente à aplicação remota no dispositivo *device*, utilizando o perfil no formato descrito na Seção 4.3.1
logMD (**id** : `String`, **data** : `Object`)	dada uma chave e um valor, registra uma entrada de metadado junto ao nó coordenador
queryMD (**id** : `String`, **start** : `Timestamp`, **end** : `Timestamp`)	dada a chave, recupera a lista correspondente de entradas de metadados no intervalo

Tabela 4.3: Serviços do `HealthAppDriver`.

HealthAppDriver - Interface Interna	
putProfile (**personId** : String, **appId** : String, **profile** : Map<String, List<Pair<String, Map<Object, Object>>>)	adiciona ou define um novo perfil ao par <personId, appId>
removeProfile (**personId** : String, **appId** : String)	remove um perfil previamente adicionado
convert (**pin** : String, **value** : Object)	realiza a cadeia de conversões, se houver, para o pino, sobre o valor informado

Tabela 4.4: Interface Interna do `HealthAppDriver`.

Capítulo 5

Ferramentas

As novas entidades básicas e *drivers* definidos até aqui já permitem que aplicações utilizem uma série de funcionalidades. É possível expor e obter dados sobre as outras aplicações compatíveis existentes no ambiente, incluindo aquelas que atuam como nó coordenador. Adicionalmente, as aplicações já podem declarar e consultar as interfaces padronizadas para entrada e saída de dados, e realizar conexões com outras aplicações, inclusive empregando adaptadores parametrizados para a conversão automática destes dados.

O projetista de um jogo ubíquo para reabilitação pode agora modelar a mecânica sob a certeza de que os dados recebidos respeitarão formatos bem definidos, já que esta garantia é dada pelo contrato das interfaces. Do ponto de vista do jogo, cada pino tem apenas um tipo, com um conjunto de restrições claro, e fica a cargo das aplicações clientes mapear diferentes dispositivos aos pinos, com base nos dados fornecidos pelo nó coordenador.

Finalmente, foram providos serviços genéricos para registrar e consultar metadados de saúde, identificados por uma chave e de formato definido pelas próprias aplicações.

Essas facilidades, no entanto, ainda estão acessíveis apenas via código. Para utilização do sistema em ambientes reais, com usuários que potencialmente não possuem conhecimento técnico de programação, é interessante que exista uma ferramenta que facilite a geração dos dados para o sistema, *i.e.*, os perfis de usuários. Por esse motivo, foi disponibilizada uma interface gráfica para a criação e edição de perfis, incluindo as cadeias de adaptadores e seus parâmetros, descrita na Seção 5.1.

Para realizar testes sobre a infraestrutura proposta, bem como prover um esquema simples para modelar os movimentos humanos, foram ainda desenvolvido dois *drivers* adicionais, que são decritos na Seção 5.2.

5.1 Painel de Controle

O painel de controle é uma interface gráfica que permite aos usuários finais cadastrar pessoas e visualizar os dispositivos e aplicações presentes no *smartspace* (entre elas, os jogos), podendo em seguida definir as conexões entre esses elementos.

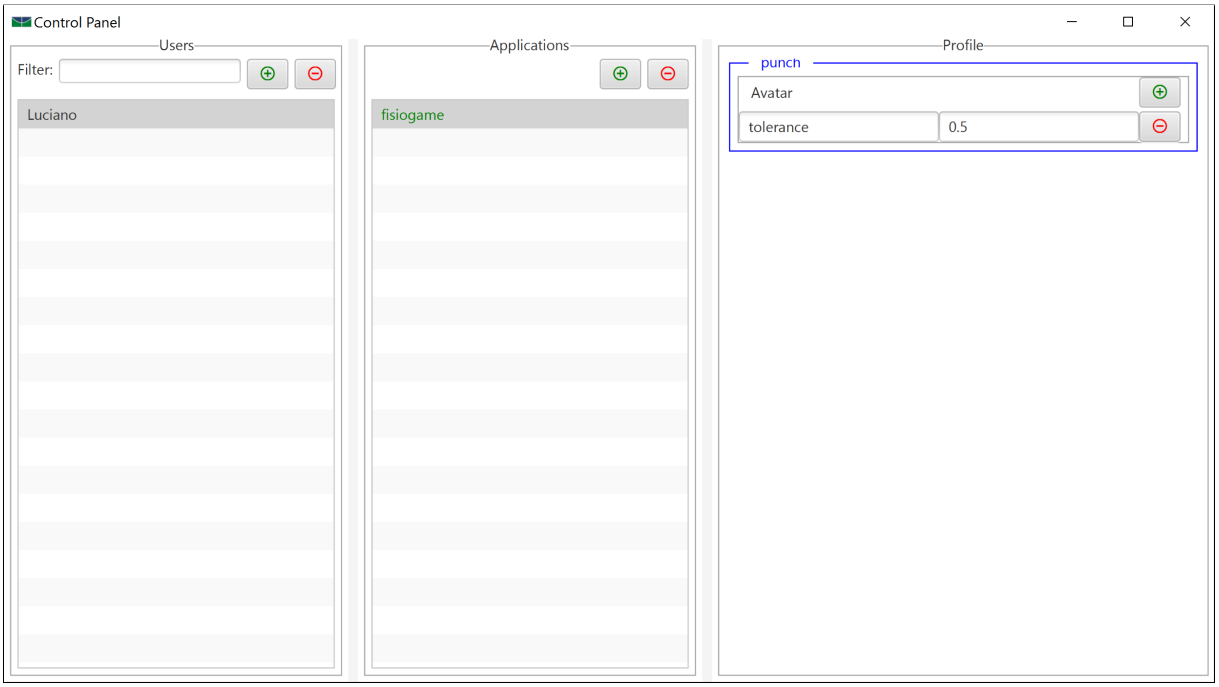


Figura 5.1: Interface do painel de controle.

A Figura 5.1 ilustra essa interface. No primeiro painel, à esquerda, são listados os usuários cadastrados, além de ser possível adicionar novos ou remover existentes. No painel central, são listadas as aplicações para as quais já existem perfis associados; aquelas atualmente presentes no *smartspace* são destacadas em verde. Além dessas, são também listadas as aplicações presentes no ambiente para as quais ainda não existem perfis. O usuário pode adicionar uma nova aplicação diretamente, pelo nome, ou implicitamente, editando uma aplicação da lista para a qual ainda não existia um perfil associado.

Uma vez que um usuário e uma aplicação estejam selecionados nas listas, o último painel, à direita, vai mostrar a interface que permite editar o perfil. Nesse painel, são listados todos os pinos expostos pela aplicação e, para cada um deles, é possível adicionar ou remover adaptadores, que ficam organizados em uma lista que representa a cadeia de conversão. Para cada adaptador, é possível adicionar parâmetros e definir seus valores.

A aplicação painel de controle sempre age por padrão como nó coordenador no ambiente, de tal maneira que centraliza a interface gráfica e a base de dados de perfis em um único ponto.

5.2 *Drivers* para captura de movimento

Para testar a solução proposta e prover ferramentas que facilitem a captura de movimentos do corpo humano – uma funcionalidade importante para jogos que utilizem exercícios fisioterápicos – foram implementados dois *drivers* adicionais.

O `IMUDriver` modela dispositivos capazes de medir uma rotação, por exemplo, a rotação do braço ou da perna do usuário. O `AvatarDriver`, por sua vez, modela hierarquias de componentes e representa as rotações relativas entre esses componentes, da mesma maneira que o corpo humano pode ser decomposto em uma hierarquia de partes e os movimentos em rotações relativas entre essas partes.

Esses dois *drivers* são descritos a seguir.

5.2.1 `IMUDriver`

Uma IMU (*Inertial Measurement Unit*), conforme descrito anteriormente (Seção 3.2), é um sensor de rotação proprioceptivo, ou seja, capaz de medir a própria rotação.

Tipicamente, um dispositivo desse tipo utiliza uma combinação de sensores elementares (acelerômetro, magnetômetro e giroscópio) para determinar sua própria rotação absoluta (Figura 5.2), *i.e.*, em relação ao planeta. Usualmente, é possível tarar (calibrar) o sensor para que meça a rotação a partir de uma posição de referência, de modo a obter rotações relativas. Dessa maneira, é possível comparar diferentes curvas de variações na rotação do sensor sem que seja necessário iniciar sempre da mesma posição, bastando calibrá-lo no início da captura.

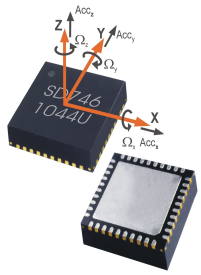


Figura 5.2: Exemplo de IMU (retirada de [13]).

Apesar de o `IMUDriver` ter sido projetado com um sensor IMU dedicado em mente, ele é agnóstico em relação à implementação, de tal maneira que pode ser utilizado por qualquer dispositivo ou módulo de software capaz de gerar uma leitura de rotação na forma de um quatérnio [24]. De fato, o *driver* não fornece serviços para realizar a leitura diretamente junto a um *hardware* particular, mas sim expõe uma interface interna que permite a outros componentes do sistema notificá-lo sempre que a leitura varia.

As interfaces do `IMUDriver` são descritas a seguir.

Serviços e Eventos

As Tabelas 5.1 e 5.2 descrevem, respectivamente, os serviços e eventos do *driver*.

Notificações de evento `change` são geradas aos observadores registrados quando há variações na medida do sensor. Essas notificações ocorrem apenas se a diferença (em módulo) do novo valor em relação ao último informado ultrapassar uma certa *sensibilidade*. A sensibilidade é um valor real definido em tempo de carregamento do *driver*, e é verificada componente a componente, *i.e.*, uma nova notificação é gerada apenas se alguma das componentes do quatérnio tiver uma variação maior que a sensibilidade corrente. O serviço `getSensitivity` permite consultar esse valor.

Para permitir que múltiplos sensores atuem simultaneamente em um mesmo *smart-space* (ou até em um mesmo dispositivo), o `IMUDriver` sempre informa uma `id` do sensor junto com a medida de rotação. O serviço `listIds` permite consultar quais são as `id`s que podem ser utilizadas por uma instância particular.

A sensibilidade, a lista de `id`s válidas e a `id` padrão a ser utilizada pelo *driver* são definidas em tempo de carregamento, via mapa de configurações do uOS. É possível também estabelecer um intervalo mínimo entre uma notificação e outra, de tal forma a limitar a quantidade de mensagens gerada pelo sistema.

O serviço `tare` faz o sistema guardar a posição (absoluta) atual como posição de referência, de tal maneira que todas as medidas subsequentes serão dadas em relação a ela – *i.e.*, a medida efetivamente reportada será a diferença normalizada entre a posição absoluta informada ao driver e esta posição de referência.

Os serviços `startRecording` e `stopRecording` permitem, respectivamente, iniciar e finalizar a gravação de uma curva de medidas. Quando a captura é finalizada, a curva é retornada na forma de uma lista de pontos com seus tempos relativos. Na chamada do serviço `startRecording`, é possível definir o intervalo mínimo entre duas medidas subsequentes na curva. Essa funcionalidade de captura de curvas é útil, por exemplo, para registrar movimentos de referência que deverão ser realizados por uma pessoa durante um exercício.

O *driver* possui um modo de *matching* para facilitar a verificação de movimentos em tempo real. O serviço `match` recebe como parâmetros um quatérnio alvo e um limiar, que define o grau de tolerância para o atingimento do objetivo. O sistema monitora a medida do sensor e gera uma notificação de evento `matchEnded` se o valor atingir o alvo especificado, dentro da tolerância definida. É possível também especificar um intervalo de *timeout* após o qual o *matching* falha, ou ainda interromper a detecção explicitamente, por meio do serviço `stopMatch`.

O propósito desses serviços é permitir que uma curva de movimentação gravada previamente seja dividida em etapas que serão então detectadas sucessivamente, cada um

com uma posição alvo final. Esse modelo de detecção de curvas é extremamente simples e atende apenas aos objetivos de testes deste trabalho, sem buscar fazer uma investigação profunda sobre algoritmos para detecção de movimentos.

IMUDriver - Serviços	
getSensitivity ()	retorna o valor da sensibilidade, *i.e.*, o valor real ao qual cada componente da variação entre medidas é comparada quando da verificação se uma nova notificação deve ser gerada
listIds ()	retorna uma lista com todas as `id`s válidas, ou seja, que podem potencialmente ser utilizadas por esse sensor
tare ()	tara o sensor, *i.e.*, define que a rotação (absoluta) atual deve passar a ser utilizada como referência para medidas subsequentes
startRecording (**sensorId** : String, [**stepTime** : int])	inicia a captura da curva de rotação para o dado sensor, opcionalmente especificando um intervalo mínimo entre dois pontos consecutivos; se essa operação for executada com sucesso, uma `recordId` será gerada e retornada
stopRecording (**sensorId** : String, **recordId** : String)	encerra a captura previamente iniciada da curva de rotação para o dado sensor, informando necessariamente a `recordId` gerada quando do início da captura
match (**sensorId** : String, **target** : Quaternion, **threshold** : double, [**timeout** : long])	inicia o modo de *matching* para um sensor, aguardando a leitura atingir o quatérnio alvo, com o limiar especificado (tolerância), opcionalmente falhando com o dado *timeout*
stopMatch (**sensorId** : String)	interrompe o modo de *matching* para um sensor

Tabela 5.1: Serviços do `IMUDriver`.

IMUDriver - Eventos	

change (
  newData : {
  timestamp : long,
  quaternion : Quaternion,
  id : String
  }
)
	notificação enviada quando o valor do sensor varia com uma diferença maior que a sensibilidade atual

matchEnded (
  id : String,
  result : {SUCCESS, FAIL, INTERRUPTED},
  time : long
)
	notificação enviada quando o modo de *matching* é encerrado, ou seja, o objetivo foi atingido, houve *timeout* ou ele foi deliberadamente interrompido

Tabela 5.2: Eventos do IMUDriver.

Interface Interna

Uma vez que, do ponto de vista da aplicação, o IMUDriver é um *driver* passivo, sua interface interna (Tabela 5.3) é bastante simples. Cabe à aplicação que lida com o *hardware* IMU específico comunicar-se com o sensor e realizar chamadas ao método sensorChanged quando detectar variações na medida.

Os métodos {get, set}DefaultSensorId são utilitários que permitem definir um sensorId padrão que será utilizado pelo *driver*, caso um outro não seja informado.

IMUDriver - Interface Interna	
{get, set}DefaultSensorId ([**sensorId** : String])	*getter* e *setter* para a id padrão, que será utilizada pelo driver caso uma id não seja fornecida ao método sensorChanged

sensorChanged (
  newData : Quaternion,
  [**sensorId** : String,
  timestamp : Long]
)
	deve ser chamado pelo sistema para notificar o *driver* de alterações na medida do sensor; se sensorId não for informado, o padrão será utilizado; se timestamp não for informada, o horário do sistema é utilizado

Tabela 5.3: Interface Interna do IMUDriver.

Plataformas

Conforme explicado anteriormente, o `IMUDriver` é de propósito geral, podendo potencialmente ser empregado em conjunto com qualquer sensor que seja capaz de gerar um valor de rotação na forma de um quatérnio. Além disso, o *driver* não comunica-se diretamente com nenhum *hardware* particular, ou seja, é agnóstico em relação à implementação. Por esse motivo, para sua utilização em ambientes reais, é necessária a criação de aplicações adicionais, que estabeleçam a comunicação com os sensores em diferentes plataformas, respeitando suas especificidades.

Para este trabalho, foram implementadas duas aplicações desse tipo. A primeira delas (Figura 5.3) é voltada para sistema operacional Android[1]. Essa aplicação acessa o sensor `ROTATION_VECTOR` – disponível na maioria dos celulares modernos utilizando sistema – e alerta o `IMUDriver` sempre que há variação na medida, transformando qualquer celular compatível em um IMU.

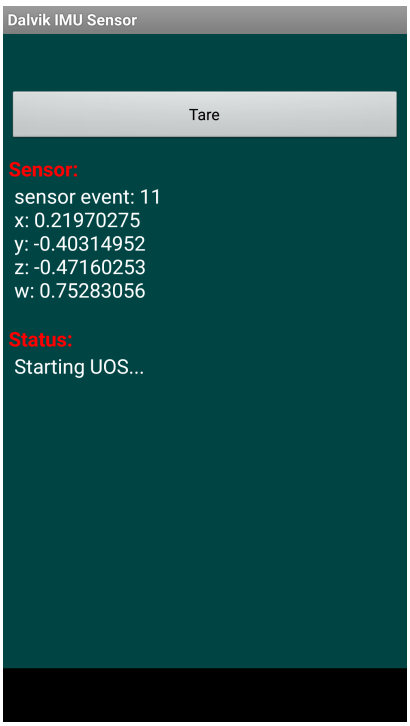


Figura 5.3: Aplicação IMU para celulares Android.

Além da aplicação Android, foi também implementada uma aplicação para *desktop* em Python[2], que comunica-se com sensores "3 Space" (Figura) da YOST Labs [55] e faz uma implementação básica do `IMUDriver` para prover as mesmas funcionalidades, gerando mensagens JSON compatíveis, conforme o uP.

[1] `https://www.android.com/` (acessado em Mar/2016)
[2] `https://www.python.org/` (acessado em Mar/2016)

Figura 5.4: Sensor IMU “3 Space” da YOST Labs [55].

5.2.2 `AvatarDriver`

Uma vez que seja possível medir as rotações de sensores individuais, pode-se combinar um ou mais sensores e registrar suas rotações relativas entre si para capturar o movimento do conjunto como um todo. O `AvatarDriver` foi desenvolvido para este fim.

Apesar de ter sido projetado para representar especificamente o corpo humano, um *Avatar* é simplesmente uma hierarquia (Figura 5.5) em que cada elemento armazena sua rotação relativa, *i.e.*, em relação ao seu pai. A rotação relativa C_r entre as rotações absolutas do filho, C_a, e do pai, P_a, é aqui definida como o quatérnio tal que $C_r * P_a = C_a$, ou seja, a multiplicação da rotação relativa do filho pela rotação absoluta do pai recupera a rotação absoluta do filho. Essa rotação relativa é, portanto, dada por $C_r = C_a * P_a^{-1}$.

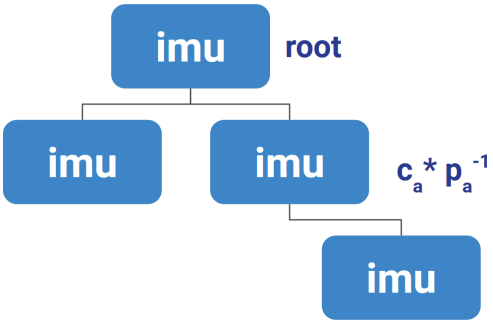


Figura 5.5: Hierarquia de sensores IMU no `AvatarDriver`.

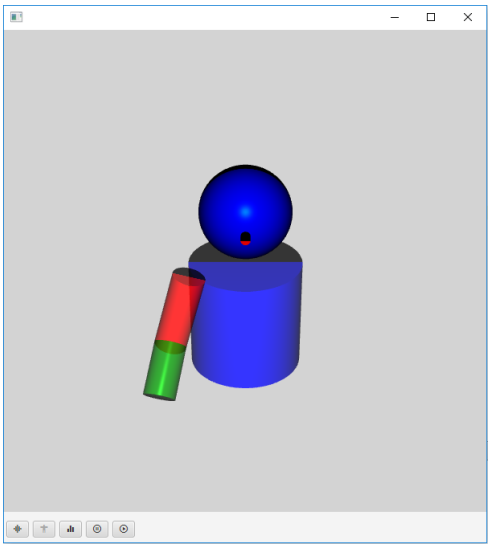
Do ponto de vista do programador, a hierarquia é descrita via mapa de inicialização do *driver*, na forma um *array* JSON. Conforme ilustrado na Figura 5.6, cada elemento da lista contém um identificador único, opcionalmente um `id` de sensor IMU (se não informado, será utilizado o mesmo identificador do elemento) e o identificador do seu nó pai, exceto, naturalmente, pela raiz.

```
[
  {
    "id" : "arm",
    "sensorId" : 1,
    "parendId" : null
  },
  {
    "id" : "forearm",
    "sensorId" : 2,
    "parendId" : "arm"
  }
]
```

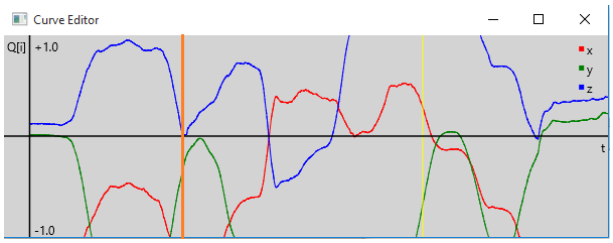
Figura 5.6: Exemplo de especificação de hierarquia do `AvatarDriver`.

Em tempo de execução, é possível alterar os sensores associados aos elementos da árvore, indicando-se IMUs disponíveis (instâncias do `IMUDriver` no ambiente). O *driver* ouve eventos dos sensores associados e atualiza os nós da árvore sempre que necessário, gerando, nesse caso, eventos `change` relativos ao avatar como um todo.

Uma vez que utiliza o `IMUDriver`, o `AvatarDriver` também disponibiliza seus próprios serviços `startRecording`, `stopRecording` e `match`, que funcionam de maneira análoga, com a diferença que a gravação de curvas lida com a hierarquia como um todo e o *matching* só será bem sucedido se os alvos forem atingido para todos os nós. Dessa maneira, é possível guardar e verificar movimentos complexos envolvendo várias partes do corpo.



(a) visualização do avatar



(b) gráfico da curva

Figura 5.7: Interface do Avatar no painel de controle.

O painel de controle disponibiliza uma interface, ilustrada pela Figura 5.7, para visualizar (a) um avatar padrão (ou subpartes deste), que representa o corpo humano, com

feedback em tempo real o movimento realizado. Nessa interface estão disponíveis controles para iniciar e interromper a captura de curvas. É possível visualizar a curva capturada na forma de um gráfico com cada componente do quatérnio em uma cor diferente (b) e dividí-la em etapas (barra alaranjada).

Aqui é implementado um modo de batimento de curva bastante simples. Primeiro, uma *curva de referência* é capturada. Após a decomposição dessa curva em E_t etapas, o sistema interpretará a posição final de cada uma delas como uma meta para o `AvatarDriver`. A duração medida para a etapa E_i, $T_{R,i}$, é utilizada como critério para definir o *timeout* de aceite no momento da realização do movimento a ser verificado. Em tempo de *matching*, quando o usuário executa o movimento, é utilizado $T_{R,i} + 50\%$ como tempo limite. Caso o alvo não seja atingido nesse tempo, a etapa é considerada falha e a captura da curva termina. Observa-se, portanto, que a verificação das etapas é cumulativa, a etapa E_{i+1} só é verificada se a etapa E_i tiver sido bem sucedida. Ao final do processo, é obtido E_S, ou seja, o total de etapas bem sucedidas até o *matching* da curva terminar ($E_S \leq E_t$).

O sistema determina como taxa de acerto na curva a razão E_S/E_t, gerando, portanto, um valor no intervalo real $[0, 1]$.

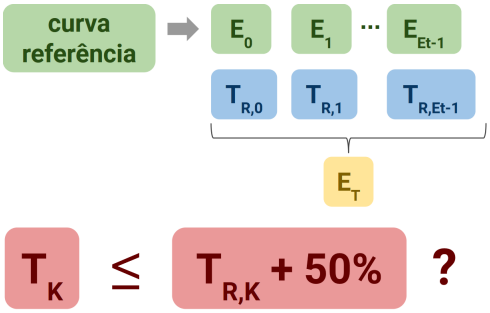


Figura 5.8: Sistema simples para *matching* de curvas.

É importante observar que este algoritmo de *matching* implementado serve apenas ao propósito de verificar conjuntos de movimentos simples e realizar testes com a solução. Está fora do escopo desse trabalho a investigação de mecanismos sofisticados para detecção de movimentos complexos ou com alta precisão.

Capítulo 6

Resultados

O foco desse trabalho é tecnológico, *i.e.*, está na disponibilização de infraestrutura para construção de jogos ubíquos com foco em reabilitação, em especial permitindo a configuração do ambiente para *adaptabilidade ao paciente*, conforme discutido nos capítulos anteriores. Por esse motivo, para validar a solução implementada, um sistema simples que exercita esta infraestrutura foi utilizado.

Um jogo conceito foi idealizado e um protótipo de seu mecanismo de entrada foi implementado. No jogo, cada paciente deve realizar com sucesso um conjunto de exercícios fisioterápicos com o objetivo de coletar recursos e trabalhar colaborativamente com outros pacientes na construção de uma cidade. Um exemplo em que o personagem virtual atua como um lenhador é ilustrado na Figura 6.1.

Figura 6.1: Ilustração do personagem lenhador.

Um ponto importante a ser levado em consideração é que o jogo não pode distrair o paciente a tal ponto que não observe a correta execução dos exercícios, visto que, praticado de forma incorreta, o treinamento não apenas não seria efetivo como potencialmente poderia prejudicar a terapia ou mesmo causar novas lesões.

Além disso, é necessário desvincular o movimento que o personagem virtual realiza no jogo do movimento praticado pelo paciente [46]. Não apenas porque diferentes exercícios,

praticados por diferentes pessoas devem ser compatíveis com o jogo, como também porque uma conexão direta das ações do paciente com as ações do personagem seriam mais um fator de distração e tornariam o jogo extremamente específico, desconsiderando toda a adaptabilidade trazida pela plataforma.

Com estas duas restrições em mente, foi desenvolvido um protótipo que funciona em dois estados distintos: captura do movimento e *feedback* ao usuário. No momento de captura, o jogador deve realizar o exercício de acordo com um padrão previamente calibrado de acordo com suas características individuais. Em seguida, o jogo avalia a precisão com que o movimento foi realizado e gera um nível de acerto correspondente, que determina o nível do golpe que será dado pelo personagem: fraco, médio ou forte. Esses três níveis foram assim escolhidos por serem a divisão mais simples que não é puramente binária (apenas acerto ou erro), o que é suficiente para verificar a infraestrutura implementada.

O sistema de *matching* descrito na Seção 5.2.2 foi utilizado para captura de movimentos. A taxa de acerto gerada serve de entrada para o golpe dentro do jogo (Tabela 6.1).

$\mathbf{E_S/E_T}$	**Tipo de Golpe**
0	–
> 0 e < 0.4	Fraco
$>= 0.4$ e < 0.7	Médio
$>= 0.7$	Forte

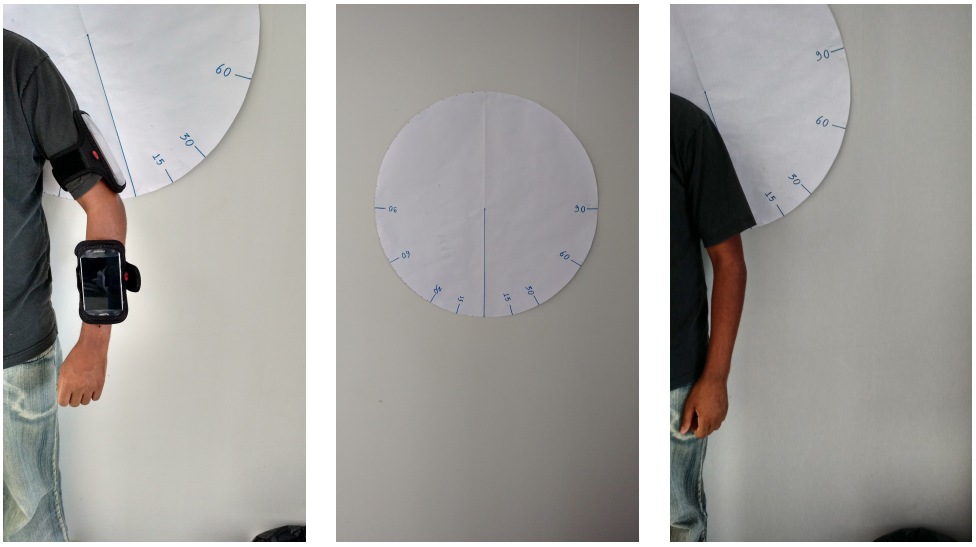
Tabela 6.1: Conversão de taxa de acerto em golpe.

Definido esse método de entrada, foram realizados os testes. Inicialmente, foi verificada a detecção de uma única etapa. Em seguida, de movimentos mais complexos com múltiplas etapas. Por fim, foi realizada uma validação do uso de dispositivos diversos. O processo é descrito nas seções a seguir.

6.1 Detecção de uma etapa

Para verificação da detecção de uma única etapa pelo sistema, foram executados alguns movimentos de teste, com sensores Android (Figura 6.2(a)) e utilizando como referência um *template* com ângulos de abertura (Figura 6.2(b)) que pode ser alinhado tanto com os braços quanto com as pernas. Partindo-se da posição inicial em que a parte do corpo a ser movimentada está alinhada à reta de referência (Figura 6.2(c)), cada um dos movimentos a seguir foi realizado nas amplitudes 15º, 30º, 60º e 90º:

1. levantar o braço direito frontalmente esticado;
2. levantar o braço direito lateralmente esticado;
3. levantar o braço esquerdo frontalmente esticado;
4. levantar o braço esquerdo lateralmente esticado;
5. levantar a coxa direita frontalmente;
6. levantar a coxa esquerda frontalmente.



(a) disposição do sensores (b) *template* com os ângulos (c) corpo em relação ao *template*

Figura 6.2: Ambiente para os testes de movimento.

Esses exercícios foram realizados 11 vezes cada um, com uma execução inicial para calibração e, em seguida, 10 execuções de teste, sendo 5 casos de sucesso (dentro de $T_R + 50\%$) e 5 casos de falha (ultrapassando o tempo limite), gerando os dados mostrados na Tabela 6.2. Os valores T_S e T_F representam os tempos médios das medidas de sucesso e de falha, respectivamente.

Movimento	**Tempo Médio** (ms)											
	15º			**30º**			**60º**			**90º**		
	$\mathbf{T_R}$	$\mathbf{T_S}$	$\mathbf{T_F}$	$\mathbf{T_R}$	$\mathbf{T_S}$	$\mathbf{T_F}$	$\mathbf{T_R}$	$\mathbf{T_S}$	$\mathbf{T_F}$	$\mathbf{T_R}$	$\mathbf{T_S}$	$\mathbf{T_F}$
1	715	730 ±87	1080 ±18	1112	1176 ±102	1670 ±15	1545	1574 ±134	2340 ±17	1924	2080 ±201	2890 ±16
2	723	730 ±85	1095 ±18	1120	1122 ±100	1681 ±16	1550	1564 ±135	2337 ±14	1930	2043 ±217	2904 ±13
3	728	747 ±83	1108 ±17	1140	1154 ±109	1715 ±15	1583	1577 ±132	2375 ±14	1944	2055 ±211	2920 ±15
4	730	751 ±86	1113 ±15	1142	1139 ±104	1722 ±14	1586	1590 ±127	2397 ±13	1946	2060 ±209	2934 ±15
5	789	815 ±84	1190 ±19	1187	1188 ±101	1796 ±13	1630	1634 ±130	2452 ±15	2050	2142 ±210	3087 ±14
6	792	810 ±87	1194 ±23	1193	1199 ±105	1780 ±15	1632	1637 ±133	2463 ±13	2054	2175 ±213	3093 ±17

Tabela 6.2: Tempos para reconhecimento de uma única etapa de curva.

6.2 Detecção de movimentos complexos

Após os testes iniciais, a mesma metodologia foi empregada, no entanto com os seguintes movimentos que envolvem múltiplas etapas, denominados neste contexto de *movimentos complexos*:

1. (a) levantar o braço direito frontalmente esticado, (b) dobrar o antebraço em direção ao tronco até um ângulo de 90 graus e (c) retorná-lo à posição esticada;
2. (a) levantar o braço esquerdo lateralmente esticado, (b) dobrar o antebraço para cima até um ângulo de 90 graus e (c) retorná-lo à posição esticada;
3. (a) dobrar a perna direita para trás, (b) levantar a coxa até a altura da cintura e (c) voltar à posição original.

No caso dos movimentos complexos, cada um consiste de 3 etapas (a, b e c). Conforme estabelecido no início deste capítulo, cada etapa é verificada individualmente e considerada bem sucedida quando o tempo medido é inferior a $T_R + 50\%$. A próxima etapa só é verificada se a etapa anterior for bem sucedida. Caso contrário, o teste para o movimento complexo é interrompido e o total de etapas bem sucedidas é computado.

Os resultados dos testes para os 3 movimentos complexos listados acima podem ser observados na Tabela 6.3, que contém o tempo de referência (T_R) e os tempos totais para diferentes E_S (número de etapas bem sucedidas até o teste ser interrompido). Quando $E_S = 0$, a tabela mostra o tempo de falha da primeira etapa. Para $E_S = 1$, é representado o tempo de sucesso da primeira etapa, acrescido ao tempo de falha da segunda, e assim sucessivamente, até que quando $E_S = 3$, é o tempo de sucesso em todas as etapas para cada movimento complexo.

Movimento	**Tempo Médio** (ms)				
	$\mathbf{T_R}$	$\mathbf{T_{E_S=0}}$	$\mathbf{T_{E_S=1}}$	$\mathbf{T_{E_S=2}}$	$\mathbf{T_{E_S=3}}$
1	3704	1657 ±18	2411 ±15	3365 ±17	3876 ±256
2	3745	1709 ±15	2598 ±16	3401 ±15	3940 ±274
3	3780	1692 ±16	2650 ±15	3469 ±18	3987 ±304

Tabela 6.3: Tempos para reconhecimento de movimentos complexos.

6.3 Diferentes dispositivos

Foi realizada uma validação final para verificar a adaptabilidade da solução a diferentes dispositivos de entrada no ambiente.

Inicialmente, o teclado convencional foi empregado para simular o envio de golpes, com diferentes teclas gerando níveis diferentes de acerto (Tabela 6.4), que são então repassados ao jogo como se fossem o resultado de um movimento capturado.

Tecla	**Valor Enviado**
N	0 (Golpe Falho)
F	0.3 (Fraco)
M	0.5 (Médio)
G	0.8 (Forte)

Tabela 6.4: Mapeamento entre teclas e taxas de acerto.

Além disso, os testes conduzidos nas Seções 6.1 e 6.2 empregaram sensores Android (aplicação dedicada em conjunto com o `IMUDriver`), sendo dois dispositivos, cada um fixado a um membro – braço/antebraço e coxa/perna –, conforme a Figura 6.2(a).

Por fim, estes mesmos testes foram realizados utilizando-se dois sensores IMU "3 Space" da YOST Labs e a aplicação *desktop* descrita na Seção 5.2.1. A captura dos movimentos neste teste foi bem sucedida, tendo em vista que os resultados coletados foram semelhantes àqueles com dispositivos Android.

Capítulo 7

Conclusão

Neste trabalho, foi desenvolvida uma uma nova solução, que tira proveito da infraestrutura já disponibilizada pelo *middleware* uOS para aplicações ubíquas e jogos ubíquos, ampliando suas capacidades para permitir a construção e/ou adaptação de jogos e aplicações ubíquas para reabilitação, em especial no que tange à *adaptabilidade ao paciente*.

Uma nova forma de organizar o *smarspace* foi proposta (Seção 4.2), com a definição de um nó coordenador, que armazena os perfis de cada usuário para diferentes jogos, além de concentrar os metadados relativos a saúde disponíveis no ambiente.

Foram definidas novas entidades e recursos (Capítulo 4) no *middleware* que permitem aos jogos definirem interfaces padronizadas de entrada e saída. Um conjunto de conversões e mapeamentos permite traduzir dados de entradas variadas para o formato padronizado esperado pelo jogo.

Para facilitar o registro de exercícios fisioterápicos, foi implementado o suporte a dispositivos de captura de movimentos e de modelagem do corpo humano (Seção 5.2).

A validação da proposta foi realizada por meio de um protótipo que mapeia movimentos humanos em golpes de um personagem virtual em um jogo (Capítulo 6). Diferentes exercícios foram realizados, envolvendo os membros superiores e inferiores, resultando na detecção bem sucedida dos movimentos.

Como trabalhos futuros, é possível realizar algumas proposições:

- concluir e testar a versão completa do jogo em um grupo de pacientes real, comparando a utilização complementar do jogo ubíquo em relação à terapia convencional;
- projetar e implementar novos jogos que utilizem a solução proposta e validá-los em diferentes contextos;
- implementar o suporte a novos dispositivos de entrada, tais como, por exemplo, o Kinect da Microsoft ou o WiiMote[1] da Nintendo;

[1] `http://www.nintendo.com/` (acessado em Mar/2016)

- adaptar jogos já existentes por meio da criação de um *wrapper* e comparar este novo modelo com a utilização dos jogos sem adaptabilidade e/ou com jogos desenvolvidos especificamente para fisioterapia;
- investigar quais os tipos de *input* e/ou movimentos são tipicamente utilizados em processos fisioterápicos e como poderiam ser melhor suportados na solução proposta;
- analisar técnicas e algoritmos mais sofisticados e precisos para detecção de curvas e integrá-los à solução;
- definir uma interface padrão ou conjunto básico de chaves para metadados de saúde;
- investigar e desenvolver facilidades que permitam a utilização do mesmo jogo em diferentes contextos da vida diária do paciente, tais como clínica, academia, casa, rua, etc;
- incorporar outros elementos à construção dos jogos, tais como interações sociais entre o paciente, profissionais de saúde, amigos e família, elementos de clima e geografia, características do ambiente, dentre outros;
- analisar e verificar quais são os elementos de jogos ubíquos e de aplicações ubíquas em geral que efetivamente influenciam o processo terapêutico, e como devem ser aplicados para este fim;
- verificar potenciais necessidades de outras áreas de saúde além da Reabilitação.

Referências

[1] *Anatomy and Physiology*. OpenStax College, 2013. 10

[2] Erwin Aitenbichler, Jussi Kangasharju, e Max Mühlhäuser. MundoCore: A Lightweight Infrastructure for Pervasive Computing. *Pervasive Mob. Comput.*, 3(4):332–361, 2007. 13

[3] Heber Amaral, JL Braga, e Aziz Galvao. Game Architecture for teaching-learning process: An application on an undergraduate course. *Games Innovation Conference ...*, pages 1–6, September 2013. 2, 9

[4] Zhen Bai, Alan F. Blackwell, e George Coulouris. Through the looking glass: Pretend play for children with autism. In *2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 49–58. IEEE, October 2013. 2, 9

[5] Tilde Bekker, Janienke Sturm, e Berry Eggen. Designing Playful Interactions for Social Interaction and Physical Play. *Personal Ubiquitous Comput.*, 14(5):385–396, 2010. 8

[6] Staffan Björk, Jussi Holopainen, Peter Ljungstrand, Karl-Petter Akesson, e Karl-Petter Å kesson Staffan Björk, Jussi Holopainen, Peter Ljungstrand. Designing Ubiquitous Computing Games: A Report from a Workshop Exploring Ubiquitous Computing Entertainment. *Personal Ubiquitous Comput.*, 6(5-6):443–458, 2002. 1, 7

[7] Elizabeth M. Bonsignore, Derek L. Hansen, Zachary O. Toups, Lennart E. Nacke, Anastasia Salter, e Wayne Lutters. Mixed Reality Games. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work Companion*, CSCW '12, pages 7–8, New York, NY, USA, 2012. ACM. 1, 7

[8] F N Buzeto, M A M Capretz, C D Castanho, e R P Jacobi. uOS: A Resource Rerouting Middleware for Ubiquitous Games. In *Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*, pages 88–95. Ieee, December 2013. 13, 16, 18

[9] F N Buzeto, C B Paula, C D Castanho, e R P Jacobi. DSOA: A Service Oriented Architecture for Ubiquitous Applications. In Paolo Bellavista, Ruay-Shiung Chang, Han-Chieh Chao, Shin-Feng Lin, e PeterM.A. Sloot, editors, *Advances in Grid and Pervasive Computing*, volume 6104 of *Lecture Notes in Computer Science*, pages 183–192. Springer Berlin Heidelberg, 2010. 16, 18

[10] F N Buzeto, T B P Silva, C D Castanho, e R P Jacobi. Reconfigurable Games - Games that change with the environment. In *Games and Digital Entertainment (SBGAMES), 2014 Brazilian Symposium on*, November 2014. 1, 7

[11] Fabricio N. Buzeto. Um conjunto de soluções para a construção de aplicativos de computação ubíqua. Dissertação (Mestrado), Universidade de Brasília – Brazil, Brasilia, DF, Brazil, 2010. 15, 18

[12] Fabricio N. Buzeto. *Jogos Ubíquos Reconfiguráveis - Concepção e Arcabouço de Desenvolvimento*. Tese (Doutorado), Universidade de Brasília – Brazil, Brasilia, DF, Brazil, 2015. 16

[13] Fabricio Nogueira Buzeto, Carla Denise Castanho, e Ricardo Pezzuol Jacobi. uP: A Lightweight Protocol for Services in Smart Spaces. In *Ubi-Media Computing (U-Media), 2011 4th International Conference on*, pages 25–30. Ieee, July 2011. 16, 18, 40

[14] G. A. P. Caurin, A. A. G. Siqueira, K. O. Andrade, R. C. Joaquim, e H. I. Krebs. Adaptive strategy for multi-user robotic rehabilitation games. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1395–1398, Aug 2011. 14

[15] Adrian David Cheok, Anuroop Sreekumar, Cao Lei, e Le Nam Thang. Capture the Flag: Mixed-Reality Social Gaming with Smart Phones. *IEEE Pervasive Computing*, 5(2):62–69, 2006. 7

[16] Cristiano Andre da Costa, Adenauer Correa Yamin, e Claudio Fernando Resin Geyer. Toward a General Software Infrastructure for Ubiquitous Computing. *IEEE Pervasive Computing*, 7(1):64–73, 2008. 6

[17] M C D'Ornellas, D J Cargnin, e A L Prado. A thoroughly approach to upper limb rehabilitation using serious games for intensive group physical therapy or individual biofeedback training. In *Games and Digital Entertainment (SBGAMES), 2014 Brazilian Symposium on*, November 2014. 13, 14

[18] I. Fette e A. Melnikov. The WebSocket Protocol. Request for Comments (Standard) 6455, Internet Engineering Task Force, December 2011. 19

[19] Google. Ingress. Available at http://www.ingress.com/ (accessed at 11/25/2014), 2014. 8

[20] M. Gotsis, A. Tasse, M. Swider, V. Lympouridis, I. C. Poulos, A. G. Thin, D. Turpin, D. Tucker, e M. Jordan-Marsh. Mixed reality game prototypes for upper body exercise and rehabilitation. In *2012 IEEE Virtual Reality Workshops (VRW)*, pages 181–182, March 2012. 14

[21] Groundspeak. GeoCaching. Available at http://www.geocaching.com/ (accessed at 11/25/2014), 2014. 8

[22] L Gutierrez, I Nikolaidis, E Stroulia, S Gouglas, G Rockwell, P Boechler, M Carbonaro, e S King. fAR-PLAY: A framework to develop Augmented/Alternate Reality Games. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*, pages 531–536, 2011. 7

[23] Lucio Gutierrez, Eleni Stroulia, e Ioanis Nikolaidis. fAARS: A Platform for Location-aware Trans-reality Games. In *Proceedings of the 11th International Conference on Entertainment Computing*, ICEC'12, pages 185–192, Berlin, Heidelberg, 2012. Springer-Verlag. 1, 7

[24] William Rowan Hamilton. On Quaternions; or On a New System of Imaginaries in Algebra. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 25(163):10–13, 1844. 40

[25] IEEE. IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 15.1a: Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications for Wireless Personal Area Networks (WPAN). *IEEE Std 802.15.1-2005 (Revision of IEEE Std 802.15.1-2002)*, pages 1–700, June 2005. 19

[26] Bryan J Kemp. Motivation, rehabilitation, and aging: a conceptual model. *Topics in Geriatric Rehabilitation*, 3(3):41–51, 1988. 2, 9

[27] K Koskinen e R Suomela. Rapid prototyping of context-aware games. In *Intelligent Environments, 2006. IE 06. 2nd IET International Conference on*, volume 1, pages 135–142, 2006. 1, 7

[28] Andrew Macvean e Judy Robertson. Understanding exergame users' physical activity, motivation and behavior over time. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*, page 1251, New York, New York, USA, 2013. ACM Press. 13

[29] Microsoft. Microsoft Kinect, 2013. 14

[30] Paul Milgram, Haruo Takemura, Akira Utsumi, e Fumio Kishino. Augmented reality: a class of displays on the reality-virtuality continuum, 1995. 7

[31] N. Miller, O. C. Jenkins, M. Kallmann, e M. J. Mataric. Motion capture from inertial sensing for untethered humanoid teleoperation. In *Humanoid Robots, 2004 4th IEEE/RAS International Conference on*, volume 2, pages 547–565 Vol. 2, Nov 2004. 14

[32] M Modahl, I Bagrak, M Wolenetz, P Hutto, e Umakishore Ramachandran. MediaBroker: an architecture for pervasive computing. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pages 253–262, 2004. 13

[33] M.M. Morrison. Inertial measurement unit, 1987. US Patent 4,711,125. 14

[34] H Ogata, C Yin, MM El-Bishouty, e Yoneo Yano. Computer supported ubiquitous learning environment for vocabulary learning. *... Journal of Learning ...*, X, 2010. 2, 9

[35] Yoshihiro Okada, Takayuki Ogata, e Hiroyuki Matsuguma. Component-based Approach for Prototyping of Movie-based Physical Therapy Games. In *Proceedings of the Workshop at SIGGRAPH Asia*, WASA '12, pages 39–45, New York, NY, USA, 2012. ACM. 2, 9

[36] Joel C Perry, Julien Andureu, Francesca Irene Cavallaro, Jan Veneman, Stefan Carmien, e Thierry Keller. Effective game use in neurorehabilitation: user-centered perspectives. *Handbook of Research on Improving Learning and Motivation through Educational Games, IGI Global*, 2010. 13

[37] M C S C Pimenta, F N Buzeto, L H O Santos, C D Castanho, e R P Jacobi. uImpala - A Game Engine for Ubigames developers. In *Network and Systems Support for Games (NetGames), 2014 13th Annual Workshop on*, 2014. 13, 16

[38] Erika Shehan ES Erika Shehan Poole, Andrew D AD Miller, Yan Xu, Elsa Eiriksdottir, Richard Catrambone, e Elizabeth D Mynatt. The Place for Ubiquitous Computing in Schools: Lessons Learned from a School-based Intervention for Youth Physical Activity. In *Proceedings of the 13th International Conference on Ubiquitous Computing*, UbiComp '11, pages 395–404, New York, NY, USA, 2011. ACM. 13

[39] J. Postel. User Datagram Protocol. Request for Comments (Standard) 768, Internet Engineering Task Force, August 1980. 19

[40] J. Postel. Transmission Control Protocol. Request for Comments (Standard) 793, Internet Engineering Task Force, September 1981. 19

[41] V Sacramento, M Endler, H K Rubinsztejn, L S Lima, K Goncalves, F N Nascimento, e G A Bueno. MoCA: A Middleware for Developing Collaborative Applications for Mobile Users. *Distributed Systems Online, IEEE*, 5(10):2, 2004. 13

[42] L H O Santos, F N Buzeto, L N Carvalho, e C D Castanho. A game engine plugin for ubigames development. In *Games and Digital Entertainment (SBGAMES), 2014 Brazilian Symposium on*, November 2014. 16

[43] Marcus Kolga Schlickum, Leif Hedman, Lars Enochsson, Ann Kjellin, e Li Felländer-Tsai. Systematic video game training in surgical novices improves performance in virtual reality endoscopic surgical simulators: a prospective randomized study. *World journal of surgery*, 33(11):2360–2367, 2009. 2, 9

[44] Christian Schönauer, Thomas Pintaric, e Hannes Kaufmann. Full body interaction for serious games in motor rehabilitation. In *Proceedings of the 2Nd Augmented Human International Conference*, AH '11, pages 4:1–4:8, New York, NY, USA, 2011. ACM. 14

[45] W. Roy Schulte e Yefim V. Natis. *Service Oriented Architectures Parts 1 and 2.* Gartner, 1996. 16, 18

[46] J. Serradilla, J. Q. Shi, Y. Cheng, G. Morgan, C. Lambden, e J. A. Eyre. Automatic assessment of upper limb function during play of the action video game, circus challenge: validity and sensitivity to change. In *Serious Games and Applications for Health (SeGAH), 2014 IEEE 3rd International Conference on*, pages 1–7, May 2014. 13, 48

[47] Six to Start. Zombies, Run!. Available at https://www.zombiesrungame.com/ (accessed at 11/25/2014), 2014. 8

[48] McKay Moore Sohlberg e Catherine A Mateer. *Cognitive rehabilitation: An integrative neuropsychological approach*. Guilford Press, 2001. 9

[49] João Pedro Sousa e David Garlan. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, WICSA 3, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V. 13

[50] E A. Suma, B Lange, A Rizzo, D M. Krum, e M Bolas. Faast: The flexible action and articulated skeleton toolkit. In *Proceedings of the 2011 IEEE Virtual Reality Conference*, VR '11, pages 247–248, Washington, DC, USA, 2011. IEEE Computer Society. 14

[51] Jan-Peter Tutzschke e Olaf Zukunft. FRAP: A Framework for Pervasive Games. In *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '09, pages 133–142, New York, NY, USA, 2009. ACM. 1, 7

[52] Marc Weiser. The World is Not a Desktop. *interactions*, 1(1):7–8, 1994. 5

[53] Mark Weiser. The computer for the 21st century. *Scientific american*, 3(3):66–75, 1991. 1, 5

[54] Mark Weiser e JS John Seely Brown. Designing Calm Technology. *POWERGRID JOURNAL*, 1:1–5, 1996. 7

[55] YOST Labs. Wireless IMU, 2016. 44, 45