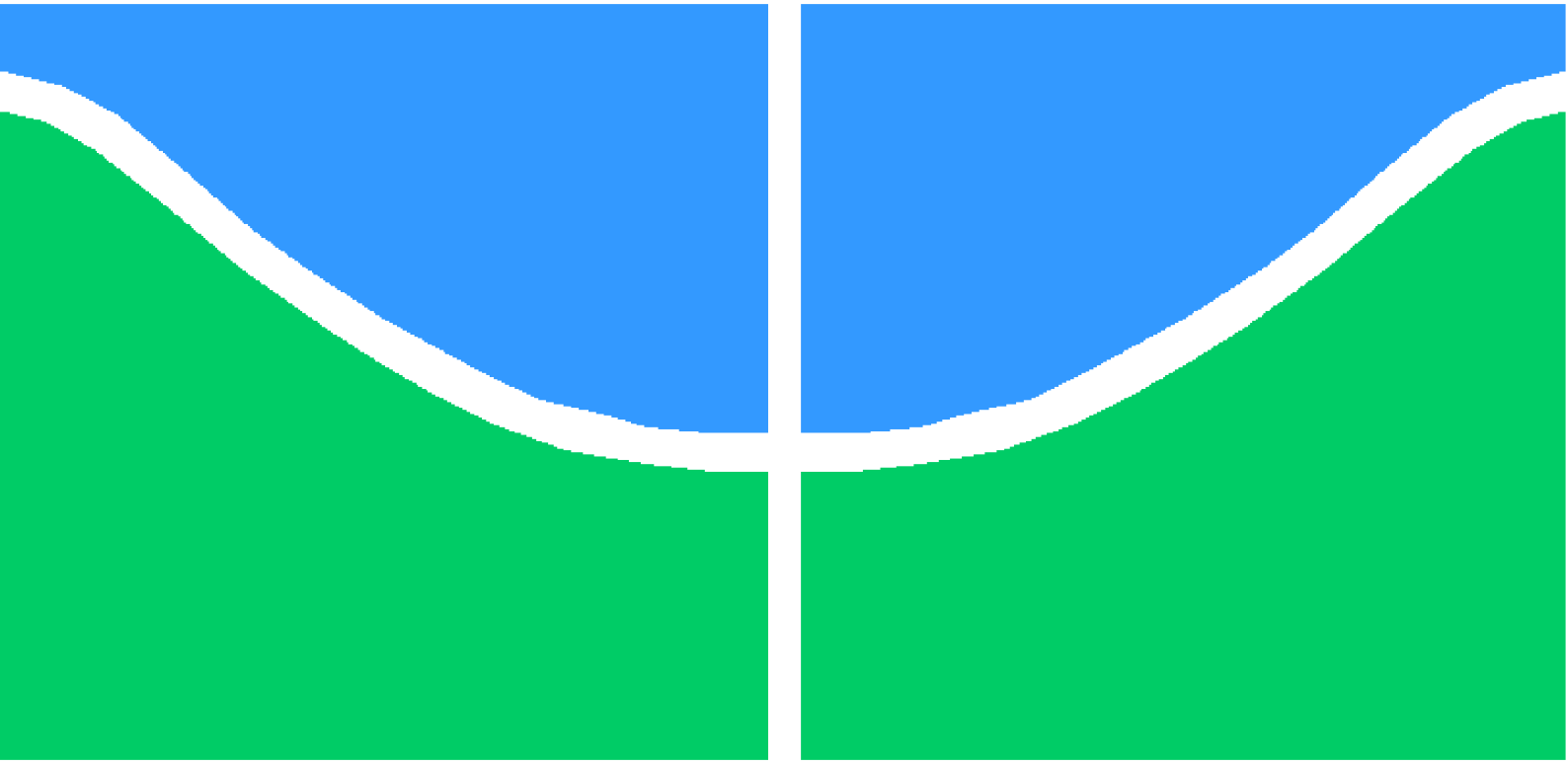


Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia Eletrônica

Sistema de detecção de quedas com interação por gestos usando sensor de profundidade

Autor: Victor Aguiar Coutinho

Orientador: Prof. Dr. Diogo Caetano Garcia

Brasília, DF

2021

Victor Aguiar Coutinho

Sistema de detecção de quedas com interação por gestos usando sensor de profundidade

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Diogo Caetano Garcia

Brasília, DF

2021

Victor Aguiar Coutinho
Sistema de detecção de quedas com interação por gestos usando sensor de profundidade/ Victor Aguiar Coutinho. – Brasília, DF, 2021-
74 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Diogo Caetano Garcia

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2021.

1. detecção de quedas. 2. Kinect. I. Prof. Dr. Diogo Caetano Garcia. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Sistema de detecção de quedas com interação por gestos usando sensor de profundidade

CDU 02:141:005.6

Victor Aguiar Coutinho

Sistema de detecção de quedas com interação por gestos usando sensor de profundidade

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, DF, 06 de novembro 2021:

Prof. Dr. Diogo Caetano Garcia
Orientador

Prof. Dr. Renan Utida
Convidado 1

Prof. Dr. Tiago Alves da Fonseca
Convidado 2

Brasília, DF
2021

Esse trabalho é dedicado ao sr. Francisco das Chagas
que um dia sonhou com seus filhos chegando onde chegavam
somente os filhos dos seus patrões.

Agradecimentos

Antes de tudo, quero agradecer a Deus, pois somente Ele pôde me dar energia para superar tudo e conseguir chegar onde eu almejava chegar. Em segundo, meus pais, seu Francisco e dona Tarcísia, que estiveram comigo sempre, me dando todo o apoio, principalmente minha mãe, se jogando no chão para que o sistema pudesse ser testado. Por fim, ao meu orientador, professor Diogo, que me ajudou muito nessa trajetória e sem ele nada seria possível.

"Quando a educação não é libertadora,
o sonho do oprimido é ser o opressor."
Paulo Freire

Resumo

A população idosa tem crescido substancialmente em todo o mundo e, consequentemente, no Brasil. Estima-se que, em 2050, 28% da população brasileira terá 60 anos ou mais. Por conta disso, a quantidade de acidentes domésticos cresceu, sendo a queda a principal causa a levar idosos ao hospital. Diversos sistemas de detecção de quedas têm sido desenvolvidos, e um dos problemas encarados é a privacidade dos seus usuários. Neste trabalho, desenvolve-se um sistema de detecção de quedas que faz uso do sensor de profundidade do Kinect e que utilize gestos para interação entre o sistema e seus usuários. O sistema fez uso de uma RDT e *mean shift* para identificar a mão direita e o método DTW para identificar o gesto de aceno. Quando a mão estava próxima ao chão, o sistema entrava em estado de alerta e era desativado com o gesto. Por fim, a precisão de encontro da mão foi de 72% e, dentre os vídeos de amostra, detectaram-se todas quedas e gestos desejados.

Palavras-chaves: Kinect. Sensor de profundidade. Detecção de quedas. Detecção de gestos.

Abstract

The elderly population has grown substantially all over the world and so in Brazil. In 2050, 28% of the Brazilian population will be 60 years old or older. And, because of this, the number of domestic accidents has grown, with the fall being the main cause of taking elderly people to the hospital. Several detection systems have been developed, and one of the problems faced is the privacy of its users. In this work, a fall detection system is developed using the Kinect's depth sensor, using gestures for interaction between the system and its users. The system used an RDT and *mean shift* to identify the right hand and the DTW method to detect the wave gesture. When the hand was near the ground, the system went into alert state and it was deactivated with the gesture. Finally, the accuracy of finding the hand was 72% and, with the sample videos, it was detected all falls and desired gestures.

Key-words: Kinect. Depth sensor. Fall detection. Gesture detection.

Lista de ilustrações

Lista de abreviaturas e siglas

3D	Três dimensões ou tri-dimensional
AVA	*All-vs-All*
BOS	Base de apoio ou *base of support*
COM	Centro de massa ou *center of mass*
CCS	Sistema de Coordenadas da Câmera ou *Camera Coordinates System*
DTW	*Dynamic Time Warping*
HMM	Modelo Oculto de Markov ou *hidden Markov model*
IBGE	Instituto Brasileiro de Geografia e Estatística
IR	Infravermelho
LOG	Linha de Gravidade ou *line of gravity*
LSTM	*Long Short-Term Memory*
NIR	Infra-vermelho para proximidades
OVA	*One-vs-All*
RDT	Árvore de Decisão Aleatória ou *Randomized Decision Tree*
RGB	Padrão vermelho-verde-azul
RGB-D	Padrão de câmera RGB com valor de profundidade
RNN	Rede Neural Recorrente ou *Recurrent Neural Network*
ROI	Região de interesse ou *region of interest*
SDK	Kit de desenvolvimento de Software ou *Software Developer's Kit*
SVM	Máquinas de Vetores de Suporte ou *Support Vector Machines*

Lista de símbolos

pl_E	*pixel* do emissor (ou projetor) IR
pl_R	*pixel* do receptor (ou câmera) IR
z	Valor de profundidade
b	Comprimento entre os sensores IR em um sistema de visão estéreo
$\|f\|$	é a distância focal de ambos sensores em um sistema de visão estéreo
I_K	Imagem recebida pelo sensor IR de forma matricial
Λ_K	Padrão de emissão IR do sensor
$\hat{D}_K$	Matriz dos valores estimados de disparidades
$\hat{Z}_K$	Matriz com os valores estimados de profundidade
$d_{I_K}(x)$	Valor de profundidade do *pixel* x na imagem I
ϕ	Parâmetros de u e v *offsets* para o cálculo de estimação de juntas usado pelo Kinect
$f_\phi(I, x)$	Característica de uma imagem I e um *pixel* x
τ	Limite para comparação da característica f ou f_ϕ
C	Grupo de partes detectáveis de um corpo
c	Parte do corpo detectado
J_c	Densidade estimada da parte do corpo c
w_{ic}	Peso do *pixel* i dado uma parte do corpo c
P	Função de probabilidade
λ_c	Limite de probabilidade aprendido no método RDT do Kinect
ζ_c	*Offset* na coordenada z para a detecção da junta da parte do corpo c
R_{ij}	matriz de sinais dos das funções classificadoras no modelo SVM
$c_{p^*}(X, Y)$	Valor total de custo no método DTW do vetor X e Y do melhor percurso p^*

Sumário

1 Introdução

1.1 Justificativa

A população idosa mundial tem crescido consideravelmente nas últimas décadas. Cerca de 8,5% da população mundial tinha 65 anos ou mais em 2016 (CIRE, 2016), a expectativa é de que, em 2050, a população idosa chegue a 17%. Já a população ainda mais velha (a partir dos 80) triplicará nesse prazo (CIRE, 2016). Com essa perspectiva, surge a necessidade de dar atenção a esses grupos etários.

As pessoas com 60 anos ou mais no Brasil equivalia a 14% em 2020, segundo a projeção do IBGE (2018), e em 2050 chegará a 28%. Ao comparar a mesma faixa etária dos dados mundiais, a proporção da população brasileira com 65 ou mais será 29% a mais do que a esperada para a população mundial na projeção citada por Cire (2016), chegando a 22% de sua população total em 2050 (IBGE, 2018).

Segundo um estudo feito pela Universidade de São Paulo, 13% da população idosa brasileira (60 anos ou mais) morava sozinha no ano de 2013 (NEGRINI et al., 2018). Ainda nesse estudo, é dito que esse grupo de idosos sofreu mais com quedas que o grupo dos que moravam com outras pessoas (35,7% a mais). Pode ser visto que existe um grupo ainda mais vulnerável dentre os idosos, aqueles que moram sozinhos.

Conforme o SUS (Sistema Único de Saúde), 75% dos casos de lesão que levaram um idoso ao hospital foram ocasionados em casa e, dentre os acidentes domésticos, a queda é o que lidera (PELLIZON, c2020). Mas a situação se torna ainda mais preocupante, pois um a cada três idosos (de 65 anos em diante) da população mundial cai a cada ano (CHASE et al., 2012).

Nos últimos anos, a pesquisa em detecção de quedas tem crescido substancialmente tanto pela academia quando pela indústria (WANG; ELLUL; AZZOPARDI, 2020). Esse crescimento tornou-se visível por volta de 2018. Pessoas (acadêmicos e empresários) têm posto seus esforços para avançar em métodos e tecnologias de detecção.

De acordo com Mubashir, Shao e Seed (2013), há três categorias para método de detecção de quedas: aquela baseada em equipamentos vestíveis, outro baseada em sensores de ambiente e, por fim, a baseada em câmeras. Nesse estudo, eles afirmam que nenhum dessas categorias possui satisfatoriamente alta sensibilidade nem uma boa especificidade. Dentre esses, entretanto, o aquela por câmeras (visão) é a mais promissora tendo uma robustez que as demais não possuem, precisando, contudo, melhorar sua flexibilidade. Mubashir, Shao e Seed (2013) destacam que tanto os métodos por câmera quando os de sensores de ambiente encaram uma questão ética de respeitar a confidencialidade e

privacidade.

Diversas tecnologias vêm sendo lançadas e uma das preocupações naquelas que envolvem câmeras, como dito anteriormente, é a privacidade dos que as utilizam. Ao falar sobre um novo equipamento que monitora uma casa — uma espécie de *drone* — anunciado pela Ring, uma empresa de casa inteligente da Amazon, Ackerman (2020) aponta o risco que tal equipamento pode ter em relação à intrusão do dispositivo, podendo ser utilizado para espionar as pessoas que estiverem na residência. A garantia de dignidade e privacidade aos que estão sendo monitorados por tecnologias assistivas é algo que tem sido buscado por desenvolvedores da área juntamente com profissionais da saúde e assistentes sociais (COMISKEY et al., 2018).

A dependência dos usuários da tecnologia é um outro problema em sistemas de detecção de quedas (MUBASHIR; SHAO; SEED, 2013). A interação homem-computador ainda sofre dificuldades já que idosos brasileiros apresentam problemas como atividades básicas e incapacidade física (NEGRINI et al., 2018).

Tendo em vista o cenário atual brasileiro, propõe-se desenvolver um dispositivo com tecnologia assistiva para ambientes fechados capaz de detectar quedas, atenuando os transtornos subsequentes, garantindo a privacidade, e com uma interação de fácil utilização. Ademais, busca-se satisfazer demandas, além das sociais, acadêmicas.

1.2 Objetivos

1.2.1 Objetivo geral

Tem-se como objetivo criar um sistema automático que utiliza o sensor de profundidade do Kinect para detectar quedas com robustez e precisão e rastrear gestos como interação homem-máquina.

1.2.2 Objetivos específicos

Tendo o objetivo geral estabelecido, definiu-se os objetivos específicos desse trabalho. São eles:

- projetar um sistema eficaz em identificação de quedas ou desmaios;
- proteger a privacidade e confidencialidade das pessoas que estão sendo assistidas;
- implementar a detecção de gestos para que o usuário não necessite tocar o dispositivo ao desejar algum comando;
- e ter um sistema com boa qualidade de processamento e independente do usuário para seu funcionamento.

1.3 Requisitos

Dado o problema exposto e sabendo o que é almejado com esse trabalho, definiram-se os seguintes requisitos:

- o período de análise de dados seja inferior ao tempo entre quatros;
- não salve as imagens de profundidade em uma nuvem ou na memória permanente;
- o sistema não utilize a câmera RGB do Kinect;
- seja posicionado adequadamente para que não haja desconforto;

1.4 Organização do trabalho

Esse documento apresenta toda a fundamentação teórica para o sistema de detecção de quedas com interação por gestos. No Capítulo 2, trata-se do embasamento teórico para a tomada de decisões referente ao objetivo e requisitos propostos. No Capítulo 3, há a solução proposta e as etapas seguidas para alcançá-la. No Capítulo 4, são apresentados os resultados obtidos após a implementação do sistema. No Capítulo 5, há as considerações finais sobre o sistema de detecção de quedas implementado. Ao final, há todos os referenciais utilizados para a escrita desse documento e os programas em C++ criados.

2 Revisão Bibliográfica

2.1 Sistema de detecção de quedas

2.1.1 Fall Alert

O *Fall Alert* é um produto da empresa SureSafe. Ele é um pendente que detecta quedas através de sensores de pressão e velocidade (SURESAFE, c2020). A empresa não disponibiliza um *datasheet* do equipamento nem os algoritmos que fazem o sistema funcionar. Tendo detectado, ele envia um sinal de emergência para a base — este trata-se de um aparelho instalado na residência do usuário conectado a uma rede telefônica — e então a empresa entra em contato com um responsável e liga para a emergência mais próxima (IRWIN, 2020).

A distância máxima do pendente à base é de 50 metros. Ele possui um botão de emergência, ou seja, um botão para que a empresa entre em contato com um responsável, e é à prova d'água, sendo que o botão também pode ser utilizado para cancelar o sinal emergencial (SURESAFE, c2020). A Figura 1 apresenta o equipamento.

Figura 1 – Imagem ilustrativa do equipamento *Fall Alert* da SureSafe (SURESAFE, c2020).

2.1.2 Apple Watch

A partir da série 4 em diante, o *Apple Watch* tem a função de detectar quedas do usuário que o utiliza. Quando ele detecta, sinaliza dando toques no pulso, emitindo um alarme e exibindo um sinal de alerta em sua tela (APPLE, c2020). Caso não haja uma queda ou o usuário esteja bem, ele pode apertar o *Digital Crown* (botão superior direito) ou tocar em *estou bem* ou em *fechar*.

Contudo, caso esteja consciente, pode selecionar o botão de emergência que o sistema entrará em contato com o serviço de emergência ou ambulatório, dependendo do

país, que esteja na ficha médica do usuário cadastrado em seu aparelho celular. Já se o usuário estiver inconsciente — ou seja, não se mover no intervalo de um minuto — o dispositivo entrará em contato automaticamente com um número já salvo. A Figura 2 apresenta uma ilustração do *Apple Watch*.

Figura 2 – Imagem ilustrativa do equipamento *Apple Watch* da empresa Apple (APPLE, c2020).

Para que isso seja realizado, o *Apple Watch* faz uso de um acelerômetro e um giroscópio para identificar mudanças em um movimento e a taxa de rotação, respectivamente, em três eixos diferentes (VERGER, 2018). O acelerômetro da nova série 4 consegue medir até $32g$ (g sendo a aceleração da gravidade), com isso, o relógio é capaz de identificar um pico grande de impacto quando, ao usuário cair, se apoia com a mão.

Com a alta sensibilidade de medição da taxa de rotação, o relógio pode determinar a orientação do pulso e, com os dados do acelerômetro, perceber uma possível queda (VERGER, 2018). Para os algoritmos de detecção de quedas no equipamento, são utilizados os dados dos usuários, tornando-o mais preciso.

O sistema usado pela Apple é o FallCall Now da empresa Fall Call Solutions (GREGG, 2016). Essa desenvolveu algoritmos para detectar tais quedas, tornando-se capaz de diferenciar quedas de mecanismos altos e baixos. Uma queda de baixo mecanismo refere-se a uma queda de uma altitude baixa como, por exemplo, de uma posição sentada. Enquanto o de alto mecanismo, em pé ou alta altitude (GREGG, 2016). Os testes feitos pelo primeiro protótipo apresentaram uma precisão de 90% para mecanismos altos, enquanto 70% para os baixos.

2.1.3 Walabot Home

A empresa Vayyar desenvolveu um equipamento com foco em acidentes com idosos no banheiro, que faz uso de imagens em três dimensões baseado em radares (COMSTOCK, 2019). *Walabot Home* é instalado na parede e entra em contato com um responsável registrado no sistema em caso de detecção de quedas (WALABOT, c2020). O equipamento e suas partes estão ilustradas na Figura 3.

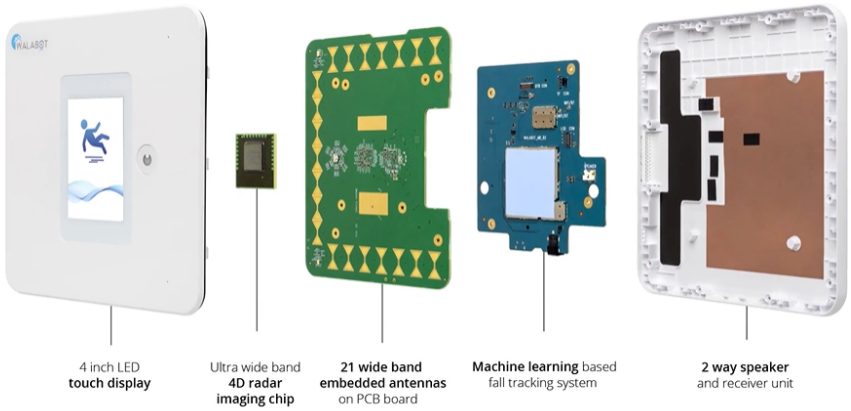


Figura 3 – Imagem ilustrativa do equipamento *Walabot Home* e seus componentes (WALABOT, c2020).

Em caso de o número cadastrado não responder, o equipamento entrará em contato com mais dois contatos registrados, e o usuário poderá conversar, se estiver consciente, com a pessoa no outro lado da linha pelo microfone do próprio equipamento. Esse processo só é executado se o usuário estiver deitado no chão, precisando de ajuda. O equipamento possui um botão que, caso apertado pelo usuário, liga para o contato de emergência semelhante ao processo após a detecção da queda (WALABOT, c2020).

Em sua arquitetura, *Walabot Home* faz uso de cinco sistemas diferentes. O primeiro, e o que o usuário tem contato, é a tela *touch* de LED com quatro polegadas. O segundo é o sistema chamado pela empresa de 4D *radar imaging chip*. Ele administra a placa PCB com 21 antenas embarcadas de banda-larga — terceiro sistema, responsável pela emissão e recepção das ondas de radiofrequência — tornando possível avaliar o ambiente e mandar para o quarto sistema fazer a decisão necessária. A placa com Aprendizagem de Máquina possui algoritmos, que não são informados pela empresa quais seriam, baseados em sistemas de detecção de quedas. Com os dados enviados pelo *chip*, o sistema processa as informações e decide se houve ou não uma queda e, então, faz a ligação caso detectado. O último sistema é a unidade com falante e receptor de voz que permite a comunicação de dois caminhos.

O equipamento possui uma cobertura de 3 metros à sua frente e 1,5 metros de cada lado, uma área total de 9 m^2(OWANO, 2018). Uma de suas vantagens é a diversidade de condições de ambientes nas quais ele consegue funcionar. A empresa afirma que *Walabot Home* trabalha bem em ambientes com fumaças, vapor e escuridão (OWANO, 2018).

2.2 Sensor Kinect

Kinect, ilustrado na Figura 4, é um dispositivo desenvolvido pela Microsoft para detectar, inicialmente, movimentos dos usuários do *console* XBox 360 (ROUSE, 2011).

Nele há o processador, um vetor de microfones, uma câmera RGB e um emissor e receptor NIR (*near-infrared*).

Figura 4 – Ilustração do Kinect com o *console* Xbox 360 (CARMODY, 2010).

Sua câmera RGB-D captura movimentos e identifica pessoas (CARMODY, 2010). Uma luz infravermelho é emitida e o sensor calcula o tempo de deslocamento para saber a profundidade dos objetos. Alguns do raios de luz são desviados o que permite identificar texturas, tornando-se possível diferenciar os objetos (CARMODY, 2010).

2.2.1 Funcionamento dos sensores de profundidade

O sensor de profundidade do Kinect é formado por dois componentes: o emissor de infravermelho (IR) — comumente chamando também de projetor IR — e o sensor IR (SHARMA, 2015) — chamado nesse documento também de receptor IR. A luz IR é emitida de acordo com um padrão de pontos e refletida pelos objetos e pessoas em sua frente. O processo de obtenção de informações do sensor está ilustrado na Figura 5. A câmera IR capta a luz daquele ponto e, com os dados de todos os pontos, determina a distância entre o equipamento e os objetos (SHARMA, 2015).

O *depth stream* é uma sequência de quadros nos quais seus *pixels* referem-se a uma distância — branco sendo o mais distante (BISWAS; BASU, 2011). As resoluções de saída suportadas são: 640x480p, 320x240p e 80x60p (SHARMA, 2015). O sensor, com esses dados e os disponibilizados pela câmera-RGB, capta a informação 3D de um objeto posto em sua frente.

O projetor IR, como mencionado anteriormente, emite um padrão de luz codificada para que seja estimada a profundidade da cena e sua geometria em três dimensões (MUTTO; GUIDO; CORTELAZZO, 2012). Na Figura 6 há um exemplo de como é esse funcionamento. O sistema de coordenadas da câmera (CCS) é chamado também de sistema referencial. A imagem obtida I_K é gerada por meio da emissão do padrão Λ_K respeitando o sistema referencial 2D K. Dentro do *console*, faz-se o processo de gerar um

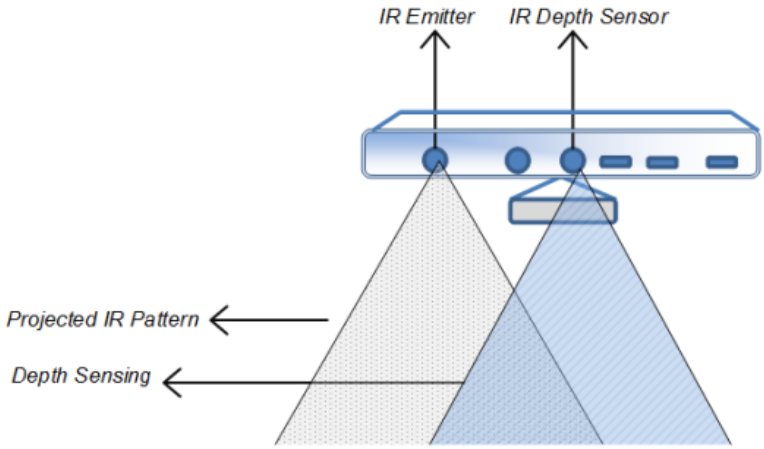


Figura 5 – Processo de obtenção das informações de profundidade através do emissor de IR e o sensor de profundidade IR (SHARMA, 2015).

mapa de disparidade estimada $\hat{D}_K$. Cada componente de $\hat{D}_K$ é um valor de disparidade d no qual é definido como a diferença entre o *pixel* do receptor IR pl_R com o do emissor pl_E, quer dizer, $d = pl_R - pl_E$.

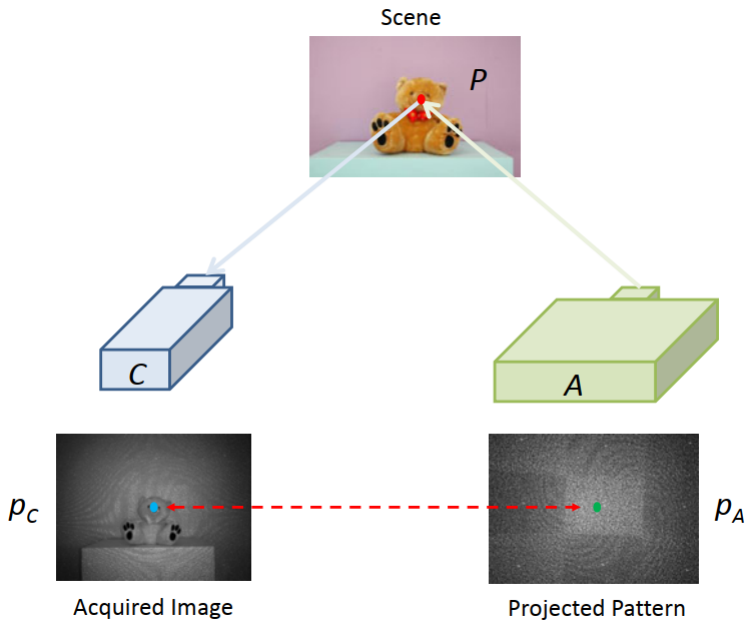


Figura 6 – Padrão de luz emitida e um exemplo de imagem recebida pelo sensor IR do Kinect sendo que p_A é o *pixel* (ponto verde) do padrão emitido pelo emissor A, p_C é o *pixel* (ponto azul) recebido pelo sensor C e P (ponto vermelho) representa a profundidade associada ao *pixel* p_A (MUTTO; GUIDO; CORTE-LAZZO, 2012).

O resultado $\hat{Z}_K$ apresenta a profundidade estimada z de cada *pixel*. Esse valor de profundidade é dado por

$$\frac{b|f|}{d},$$

sendo que b é a linha de base, ou seja, a distância entre o emissor e a câmera IR, e $|f|$ é a

distância focal[1] de ambos sensores (MUTTO; GUIDO; CORTELAZZO, 2012). Todas as imagens I_K, $\hat{D}_K$ e $\hat{Z}_K$ possuem resolução de 640x480. A profundidade mínima de medida é de 0,5 m e a máxima de 15 m. Outra informação relevante é o fato da resolução da profundidade estimada decresce quadraticamente com o aumento da distância do objeto.

2.2.2 Software Develop's Kit (SDK)

O *Software Developer's Kit* do Kinect foi lançado pela Microsoft para que fosse usado por desenvolvedores em suas aplicações de tempo real. O SDK suporta *color stream*, *depth stream* e *skeleton stream* para que seja reconhecido a voz de cada usuário detectado, para aplicações de comando de voz, e rastreado seus movimentos ou de outros objetos (SHARMA, 2015).

2.2.2.1 Detecção de Esqueletos padrão

O processo de detecção de esqueletos utilizado no *firmware* do Xbox 360 está exposto em (SHOTTON et al., 2013). Foi feito um processo de treino no qual capturaram-se 500 mil quadros e considerou-se cada quadro como uma posição aleatória; depois foram definidas partes padrões do corpo para os modelos, que podem ser vistos na Figura 7, por meios das malhas 3D desses — outras variáveis aleatórias também foram consideradas como a posição, roupas, ruídos, formato do cabelo, peso.

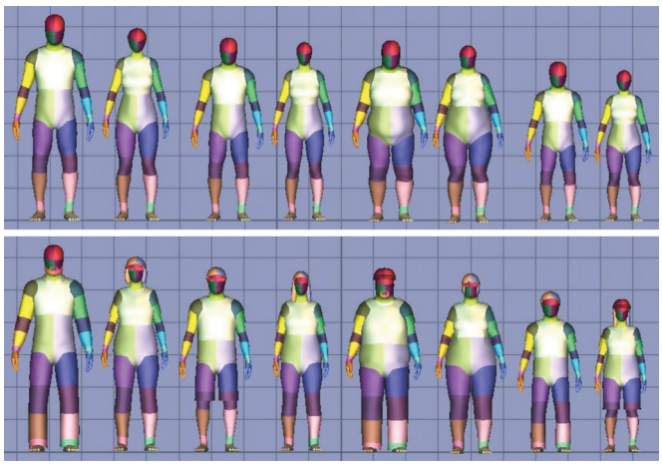


Figura 7 – Renderização de diversos modelos bases, em cima sem roupas e, em baixo, com roupas e cabelo (SHOTTON et al., 2013).

Para a etapa de aprendizagem, foi necessária a função das características comparativas, pois foi utilizado o método *randomized florest classifier* (classificador de floresta aleatória em português). Essa função compara um *pixel* a outros *pixels* nas proximidades. Essa etapa foi definida como etapa de sintetização.

[1] É definido como a distância entre a lente de um sensor e o ponto focal de imagem infinitamente distante; é dada em milímetros (mm).

A característica comparativa f_ϕ para um *pixel* x de duas coordenadas é dada por

$$f_\phi(I_K, x) = d_{I_K}\left(x + \frac{u}{d_{I_K}(x)}\right) - d_{I_K}\left(x + \frac{v}{d_{I_K}(x)}\right), \tag{2.1}$$

no qual $d_I(x)$ é o valor de profundidade do *pixel* x na imagem recebida I e o parâmetro $\phi = (u, v)$ equivale aos *offsets* u e v. Para garantir a invariância perante a alteração de profundidade, o *offset* é normalizado. Esse, quando fica fora dos limites da imagem ou no fundo da imagem, faz com que d_{I_K} retorne um valor elevado (SHOTTON et al., 2013).

A subtração dos valores de profundidade de dois *pixels* próximos de x encontra a variação de profundidade e, posteriormente, é usado em uma Arvore de Decisão Aleatória (RDT). Essa operação, apesar de simples, mostrou-se ser eficiente e sem uma diferença significante nos resultados em uma árvore aleatória com testes a base de somas ponderadas de intensidades com AdaBoost, de gradientes ou de ondas de *wavalet* de Haar (LEPETIT; LAGGER; FUA, 2005).

Cada *pixel* foi para uma floresta de decisão (ou *decision florest* em inglês) formada, no caso do kinect, por três árvores de decisão aleatória (*Randomized Decision Trees* em inglês). Estas são formadas por *split nodes* e *leaf nodes*, ver Figura 8. Dado um limiar τ, as características do *pixel* são comparadas nos *split nodes* até chegar em um *leaf node*. Com os dados de sintetização, nos quais já sabia-se a que parte do corpo cada *pixel* pertencia, foram, então, definidas funções de distribuição P para cada *leaf node*.

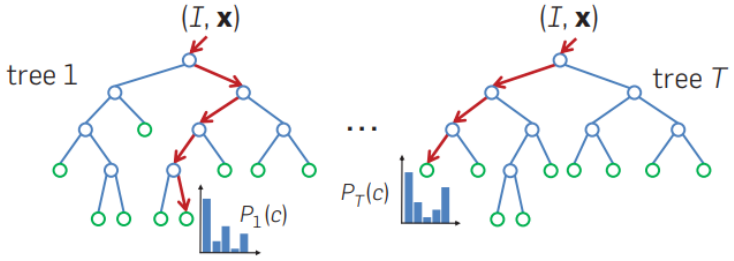


Figura 8 – Exemplificação de uma RDT sendo que os pontos azuls são *split nodes* e os verdes, *leaf nodes* (SHOTTON et al., 2013).

O uso de dois *offsets*, u e v, e uma floresta de decisão serviram para aumentar a precisão na detecção de partes mais finas e que exigem mais detalhes, como, por exemplo, o braço. O modo de operação das características proposto faz uso de 3 *pixels* de uma imagem e, no máximo, cinco operações aritméticas, tornando, portanto, um processo eficiente computacionalmente (SHOTTON et al., 2013).

A última etapa e, consequentemente, a que entrega o resultado é a de determinação das posições das juntas. Para isso, foram considerados dois métodos: baseado em *simple bottom-up clustering*, algoritmo mais rápido, e *mean shift*, mais preciso; o escolhido foi o segundo com um kernel Gaussiano com pesos. A densidade estimada da parte c do corpo

é definida como

$$J_c(\hat{x}) \propto \sum_{i=1}^{N} w_{ic} \exp\left(-\left\|\frac{\hat{x} - \hat{x}_i}{b_c}\right\|^2\right), \tag{2.2}$$

na qual N é o número total de *pixels* da imagem I, $\hat{x}$ é a coordenada no espaço tridimensional e $\hat{x}_i$ do *pixel* i dado $d_I(x_i)$, b_c a largura de banda da parte c e

$$w_{ic} = P(c|(x_i, y_i), I) \cdot z(x_i, y_i)^2, \tag{2.3}$$

sendo definido como peso do *pixel*. Esta faz com que J_c seja invariante à profundidade e melhora a detecção das juntas (SHOTTON et al., 2013).

O *mean shift* é usado, através da densidade, para encontrar os modos de forma mais eficiente (SHOTTON et al., 2013). Um limiar λ_c de probabilidade aprendido é usado para encontrar os pontos iniciais da parte c. Por fim, uma última estimativa é feita somando os pesos de cada *pixel* alcançar cada modo. Esse processo é realizado com dados bidimensionais, porém, para que torne a posição das juntas em 3D, os modos são colocados por meio de um *offset* ζ_c na coordenada z. O resultado para esses valores otimizados foram de b_c de 0,065m, λ_c de 0,14 e ζ_c de 0,039m (SHOTTON et al., 2013).

Para validar seu método, Shotton et al. (2013) definiram os parâmetros como 3 árvores com 20 de profundidade, 300 mil imagens de aprendizagem para cada árvore, 2 mil *pixels* para cada uma dessas imagens, 2 mil candidatos ϕ e 50 candidatos de limites τ por *feature* ϕ. Para a etapa de testes, 5000 imagens sintetizadas foram utilizadas e, para o teste no mundo real, 8808 imagens.

Na etapa de treino, os resultados mostraram que por volta de 100 mil imagens em diante a precisão não sofreu grandes alterações devido a limitação de árvores e profundidade. Usando o método *mean shift* inferindo as partes do corpo, tem-se uma precisão de 73,1%, porém, seu tempo de processamento (classificação das parte e a estimação das juntas) em um computador moderno de octa-core é de 20 ms (SHOTTON et al., 2013).

2.2.3 Detecção de gestos

Existem na literatura corrente diversas propostas de algoritmos para a detecção de gestos com sensores de profundidade. Biswas e Basu (2011) propuseram um método que faz uso de três etapas: pré-processamento; caracterização da região de interesse ROI (do inglês *region of interest*); e treinamento e testes. Através das informações de variação de profundidade e de movimento, é detectado, usando um classificador SVM (Máquina de Vetores de Suporte) de múltiplas classes, qual gesto foi realizado. Esse método não possui muitas operações, não exigindo muito esforço computacional. O artigo não informa a precisão de acertos, porém é apresentada a Matriz de Confusão que se encontra no Anexo B.

Um SVM de múltiplas classes gera uma matriz R de tamanho $M \times N$, sendo que M é a quantidade de classes, N é a quantidade de tarefas ou amostras e $R_{ij} \in \{-1, 0, 1\}$. Há alguns tipo de códigos das saídas, dois desses se destacam: *One-vs-All* (OVA) e *All-vs-All* (AVA). Neste são necessárias $\binom{M}{2}$ classificadores, um para cada par, e

$$R_{mn}(i) = \begin{cases} 1 & \text{se } n = 1, \\ -1 & \text{c.c.}, \end{cases},$$

enquanto naquele são M classificadores e

$$R_{mn}(i,j) = \begin{cases} 1 & \text{se } n = i, \\ -1 & \text{se } n = j, \\ 0 & \text{c.c.} \end{cases},$$

Por fim, os classificadores f^k é calculado separadamente com $(x_1, R_{1k}), ..., (x_l, R_{lk})$ e então f^{*k} — esses cálculos, assim como a função de perda V, dependerão do modelo especificado pelo desenvolvedor.

Biswas e Basu (2011) fizeram testes com detecção de oito gestos diferentes tendo $M = 2156$ (10 partes da escala cinza de 196 células e mais 196 de dados de movimento normalizados) e N a quantidade de quadros. É apontado que, caso seja usado a câmera RGB, seriam obtidos melhores resultados. O resultado apresentado foi de 83% de acerto com os dados testados, tem um gesto com 54% de acerto e outro de 93%. Os cálculos estão no Apêndice A.

Outro método de detecção de gestos é baseado em modelos de Markov ocultos (JURAFSKY; MARTIN, 2019). Cadeias de Markov (ou *Markov Chains* em inglês) são como máquina de estados finitos[2] nas quais as condições para passar de um estado a outro são as probabilidades de transações $(a_{11}...a_{ij}...a_{NN})$. Os estados $q_1q_2...q_N$ podem ser quaisquer coisas desde palavras até *tags* (JURAFSKY; MARTIN, 2019). A probabilidade de uma transição depende unicamente do estado atual. Além disso, a soma de todas as probabilidades de transição de um estado q_i é igual a 1.

O HMM (*hidden Markov model* em inglês) faz uso de cadeias de Markov para dizer se os eventos observados equivalem aos eventos omitidos, aqueles que não estão explícitos (JURAFSKY; MARTIN, 2019). Esse Método de Markov Oculto pode ser utilizado para detecção de gestos (RAHEJA et al., 2015). Um gesto capturado a 30 quadros por segundo (fps) durante 2s gera um grupo de 60 eventos observados. É preciso, então, definir a probabilidade de as mãos passarem pelos oito estados do gesto canônico em uma sequência desejada. Para facilitar o procedimento, usa-se o algoritmo de agrupamento *kmeans* para dividir os pontos em 3D de todos os dados de aprendizagem em N grupos.

[2] Método matemático utilizado para representar programas computacionais ou sistemas lógicos, também chamado de autômato.

Os autores implementaram os algoritmos em Matlab e em Python. O Matlab, devido à alocação alta de memória, deixou o processamento mais lento além de possuir algumas falhas significantes; seu tempo de processamento foi de aproximadamente 0,6s. Já em Python, obteve-se um desempenho satisfatório, levando 0,2s de processamento (RAHEJA et al., 2015).

Dynamic Time Warping é um algoritmo usado para calcular ou comparar a similaridade entre dois vetores nos quais possuem tamanhos diferentes (ZHANG, 2020). Essa técnica deforma um dos vetores para que ambos tenham o mesmo tamanho.

O objetivo do DTW é, ao comparar dois vetores $X = (x_1, ..., x_n)$ e $Y = (y_1, ..., y_m)$ de tamanhos N e M, respectivamente, encontrar o alinhamento que possui o menor custo. Este trata-se da semelhança entre os valores dos vetores dado como $C = c(x_n, y_m)$, sendo o menor custo aquele com maior semelhança (MüLLER, 2007).

Esse alinhamento é denominado como percurso de deformação $p = (p_1, ..., p_L)$ (em inglês *warping path*), tendo que $p_l = (n_l, m_l)$. Ele deve satisfazer as seguintes condições: de fronteira na qual $p_1 = (1, 1)$ e $p_L = (N, M)$; de monotonicidade, ou seja, $n_1 \leq n_2 \leq ... \leq n_L$ e $m_1 \leq m_2 \leq ... \leq m_L$; e a de que o tamanho do passo deve seguir a seguinte regra, $p_{l+1} - p_l \in (1, 0), (0, 1), (1, 1)$ para $l \in [1 : L - 1]$. Essas condições garantem que o tempo seja contínuo, não retrocedendo. DTW é definido como

$$\begin{aligned} \mathrm{DTW}(X, Y) &= c_{p^*}(X, Y) \\ &= min\{c_p(X, Y) \quad | \quad p \text{ é um } (N, M)\text{-}warping\ path\}, \end{aligned} \tag{2.4}$$

ou seja, o custo $c_p(X, Y)$ do percurso de deformação ótimo p^*, tendo em vista que $c_p(X, Y) := \sum_{l=1}^{L} c(x_{n_l}, y_{m_l})$.

Porém, para encontrar o melhor caminho $p*$ pelo modo tradicional, testando todos os *warping path* possíveis e escolhendo aquele com menor custo, é gerada uma complexidade que aumenta exponencialmente de acordo com N e M (MüLLER, 2007). Então, foi desenvolvido o algoritmo $O(NM)$ para *dynamic time warping* baseado em programação dinâmica (MüLLER, 2007). Primeiramente, é calculada a matriz de custo acumulado $D(n, m)$ que é definido como $\mathrm{DTW}(X(1 : n), Y(1 : m))$, porém, para uma maior eficiência, segue-se a definição

$$D(n, m) = min\{D(n - 1, m - 1), D(n - 1, m), D(n, m - 1)\} + c(x_n, y_n) \tag{2.5}$$

(MüLLER, 2007). Para encontrar, por fim, o percurso de deformação ótimo é feito uma rotina na qual os $p_1, ..., p_L$ são definidos de ordem reversa — começando por $p_L = (N, M)$ e, quando encontrado um $p_l = (1, 1)$, l será 1 — e seguindo a formula

$$p_{l-1} = \begin{cases} (1, m-1) & \text{se } n = 1, \\ (n-1, 1) & \text{se } m = 1, \\ \arg\min\{D(n-1, m-1), D(n-1, m), D(n, m-1)\} & \text{c.c.} \end{cases} \tag{2.6}$$

A fim de fornecer um algoritmo mais rápido de detecção de gestos, Raheja et al. (2015) aplicaram o método de *Dynamic Time Warping*. O vetor X possuiria as posições de uma junta após a fase de aprendizagem e o vetor recebido Y possuiria as posições da mesma junta, mas da fase de teste. O custo entre duas posições é dado por

$$c(x_n, y_m) = |x_n - y_m|,$$

ou seja, é o módulo da diferença entre as profundidades de dois pontos, um de X e outro de Y.

O resultado final, após algoritmo do DTW, é o *warping path* ótimo. Caso o DTW esteja dentro do limite do DTW dos dados de aprendizagem em pelo menos duas das dimensões, então os dados que estão sendo testados correspondem com os dados de aprendizagem. O programa feito por Raheja et al. (2015) em C++ — na qual tem uma boa interação com C# — levou 30 ms para comparar cada gesto testado.

No artigo de Raheja et al. (2015) são apresentados dois modelos para reconhecimento de gestos das mãos, são eles: HMM e DTW. Ambas as técnicas conseguiram uma taxa de classificação satisfatória (em torno de 90%), porém o HMM conseguiu ter uma maior precisão acrescentando o número de estados, símbolos e interações com o modelo. Para aplicações em tempo real, o que apresentou melhor resultado foi o DTW. É destacado também que o HMM é invariante ao tamanho da pessoa que está sendo assistida.

Ao comparar o que utiliza SVM com os mencionados, percebe-se a desvantagem de acurácia. A taxa de acerto médio do SVM é de 83%, valor abaixo do HMM e DTW. Além do que, nos testes com o SVM, obteve-se uma taxa de acerto para um dos gestos de 54% (BISWAS; BASU, 2011). Uma vantagem, contudo, é o processamento; não há necessidade de outro dado além da imagem de profundidade, não exigindo, assim, a etapa de determinação de pessoa ou esqueleto.

2.2.4 Detecção de quedas

O método proposto por Xu e Zhou (2018) baseia-se na teoria do balanço biomecânico de que uma queda é resultado de um desbalanço sem controle. Primeiramente, cria-se um modelo de massa dinâmica e biônica de um ser humano utilizando as juntas do esqueleto fornecido pelo Kinect e, também, o Centro de Massa (COM) (em inglês *center of mass*) calculado pela distribuição de massa do usuário. Então é calculado a Base de

Apoio (BOS) e a Linha de Gravidade (LOG) — em inglês *base of support* e *line of gravity*, respectivamente.

Em seguida, são determinadas as características biomecânicas: as velocidades do COM das cinco cadeias cinemáticas — perna esquerda e direita, tronco, e braço esquerdo e direito — e do dado esqueleto. Para identificar uma queda tendo esses parâmetros, é utilizada um rede Long Short-Term Memory (LSTM), que é uma Rede Neutral Recorrente (RNN) com capacidade de aprender dependências ordenadas de longo período para evitar problemas de predição sequencial (BROWNLEE, 2017).

Por meio de testes, Xu e Zhou (2018) destacam que a cadeia deve possuir 3 camadas de LSTM, o qual é um *trade-off* entre habilidade de modelar e prevenção de *overfitting*[3]. A acurácia do processo de predição apresentado por Xu e Zhou (2018) é de 97,41%. Ao realizar testes não executando a etapa de determinação das cadeias cinemáticas, ou seja, utilizando apenas juntas do esqueleto, obteve-se uma acurácia de 95,84%, um resultado satisfatório para um processo com menor necessidade computacional.

Outro método para detectar quedas através de detecção de esqueletos é usando o algoritmo RDT (*Randomized Decision Tree* em inglês) e SVM (BIAN et al., 2015). Primeiramente, são encontradas as juntas do esqueleto por meio do algoritmo de árvore de decisão aleatória RDT. Com os dados extraídos, é utilizada a SVM, tendo como classificadores o trajeto das juntas, para identificar se houve queda ou não.

Duas ou três características são suficientes para que uma RDT distingua objetos, sendo capaz de lidar com ruídos de imagem e precisando apenas de formatos em 2D (BIAN et al., 2015). A característica comparativa f é usado para reconhecer o *pixel* de teste através da superfície geométrica. Ela é definida como

$$f((x_0, y_0)|(\Delta x, \Delta y)) = z(x_0, y_0) - z\left((x_0, y_0) + \frac{(\Delta x, \Delta y)}{z(x_0, y_0)}\right), \tag{2.7}$$

na qual (x_0, y_0) é o *pixel* de teste, $z(x_i, y_i)$ é o valor do *pixel* i, $(\Delta x, \Delta y)$ é o valor de *offset* relacionado ao *pixel* (x_0, y_0), e o $\frac{1}{z(x,y)}$ é utilizado para normalizar o valor de *offset*. O valor da equação apresentada tem uma alta eficiência computacional já que envolve cálculos simples, dando espaço para a detecção de quedas em tempo real (BIAN et al., 2015). Os demais processos são os descritos em (SHOTTON et al., 2013). Com o algoritmo RDT, há uma redução de tempo de treinamento comparado aos modos tradicionais, chegando até 83 vezes com as mesmas condições (BIAN et al., 2015).

O processo de detecção de esqueleto do Kinect, que utiliza *mean shift*, levou 20 ms para detectar as juntas de um quadro em um sistema octa-core (SHOTTON et al., 2013). Bian et al. (2015) simularam, além de seu método, o proposto por Shotton et al. (2013)

[3] Análise tão próxima ou exatamente igual aos dados utilizados que pode falhar ao adicionar novos dados ou prever novas observações.

utilizando os mesmos parâmetros — a diferença foi que, em vez de fórmula proposta, usou-se a 2.1, porém mantendo os demais métodos. O experimento resultou em 5 ms o tempo por quadro da etapa de teste, enquanto o proposto resultou em 2,8 ms.

O primeiro método apresentado, que faz uso do algoritmo próprio do *firmware* do Kinect, chegou a 95,84%. Enquanto o segundo, que faz da SVM e usa como característica a Equação 2.7, alcançou uma precisão de 97,6%, uma diferença razoável comparado com o outro método.

3 Metodologia

Esse trabalho propõe um sistema que detecta uma queda quando o sensor de profundidade — instalado no centro do teto e com visão para o piso — identifica uma pessoa deitada. Se a posição da mão não variar muito durante 5s, o sistema considera que houve uma possível queda a qual é confirmada caso o usuário não faça o gesto de aceno nos próximos 2s.

A princípio, planejava-se usar o próprio sistema de detecção de esqueleto do Kinect, no entanto, ao testar com o aparelho no teto e uma pessoa deitada no chão, o sistema não conseguiu fazer a detecção. Isso ocorria sempre que uma pessoa estava rente a uma superfície, seja ela uma parede ou o piso. Decidiu-se, por fim, reimplementar o algoritmo da Microsoft apresentado por Shotton et al. (2013), porém, na fase de aprendizagem, fazê-lo funcionar nas condições da solução idealizada.

Portanto, a detecção da mão usa a Árvore de Decisão Aleatória (RDT) e *mean shift*, enquanto para a detecção do gesto, utiliza-se o *dynamic time warping* (DTW), por se tratar de uma aplicação em tempo real. O diagrama do sistema proposto encontra-se na Figura 9.

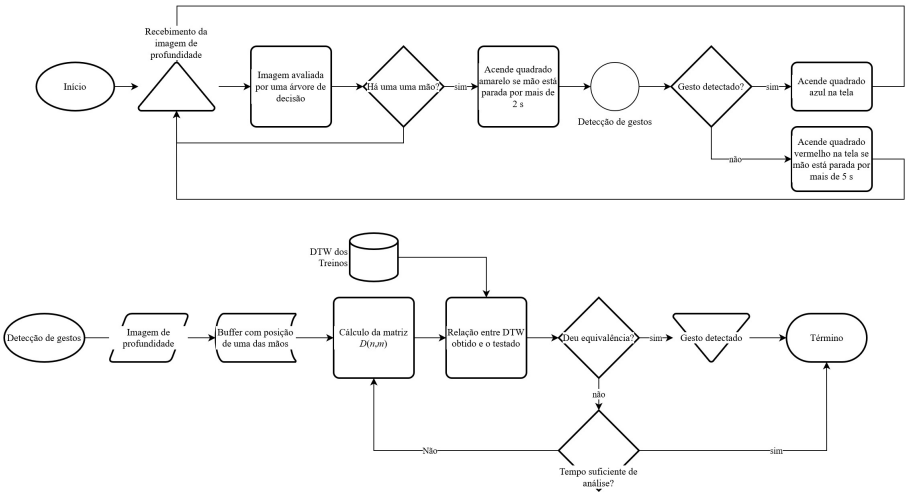


Figura 9 – Diagrama lógico do sistema proposto.

3.1 Aquivos de aprendizagem

Pirmeiro, foram gravados vídeos com as quedas e gestos para as fases de aprendizagem do sistema. Para essa etapa, usou-se o aplicativo Kinect Studio, disponibilizado pela Microsoft. Posicionou-se o Kinect no teto, com o sensor de profundidade direcionado

para o piso, como proposto. Em seguida, um usuário caiu em diversas posições e, em uns casos, fazia-se o gesto de aceno, e em outros, permanecia imóvel.

Foram gravados quadro vídeos com o usuário caindo e permanecendo parado, sendo dois desses em posição dorsal e os outros dois em pronação. E outros cinco vídeos foram gravados, porém, o usuário se deitava no chão e, em seguida, fazia o gesto de aceno. Foram três em posição dorsal e dois em pronação.

3.2 Implementação do sistema de detecção de quedas

Na Figura 10 é apresentado o diagrama de classes UML do programa desenvolvido em C++ para ser integrado com aquele que a Microsoft disponibiliza para leitura do sensor

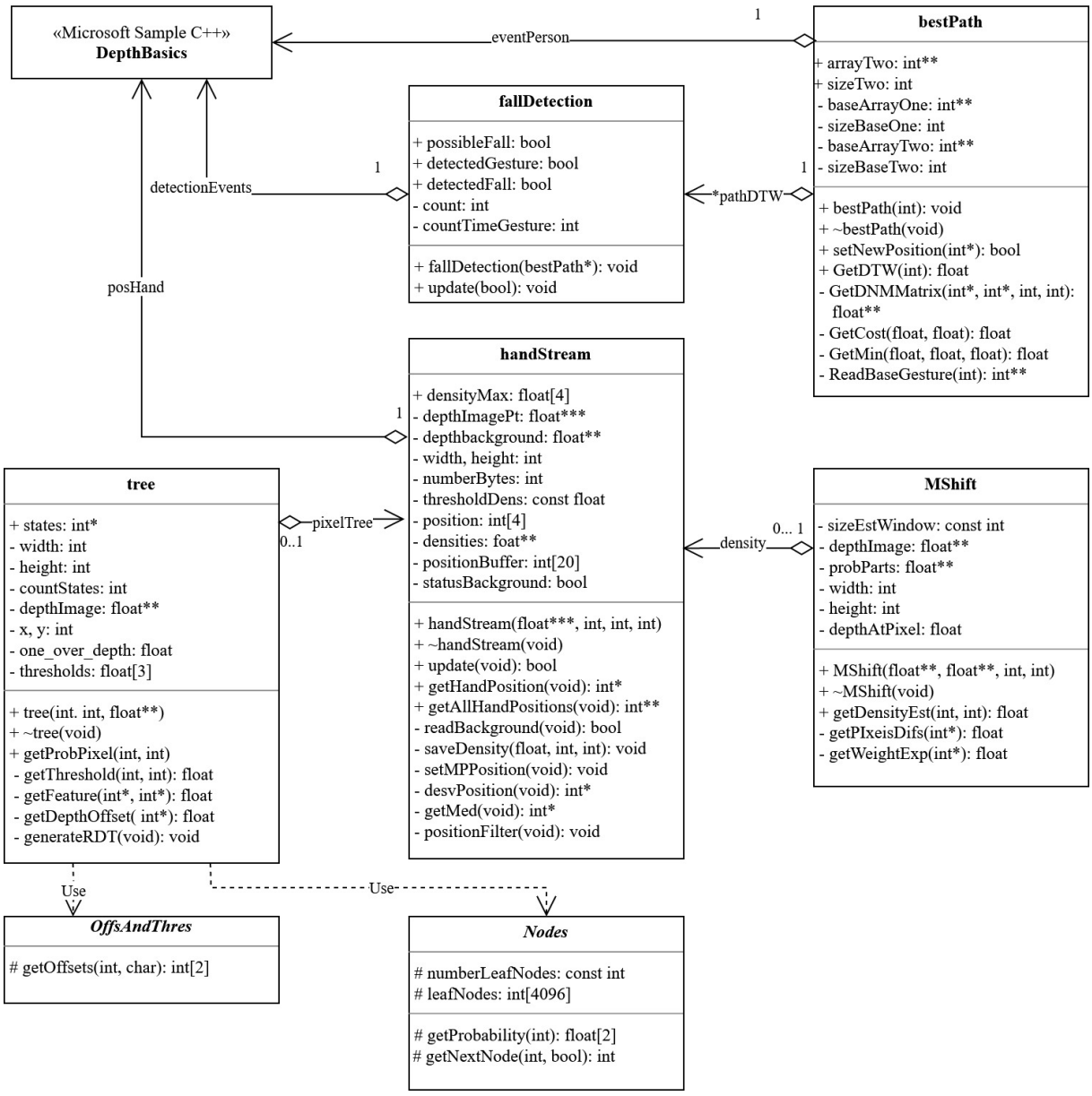


Figura 10 – Diagrama de Classes UML do sistema de detecção de quedas e gestos.

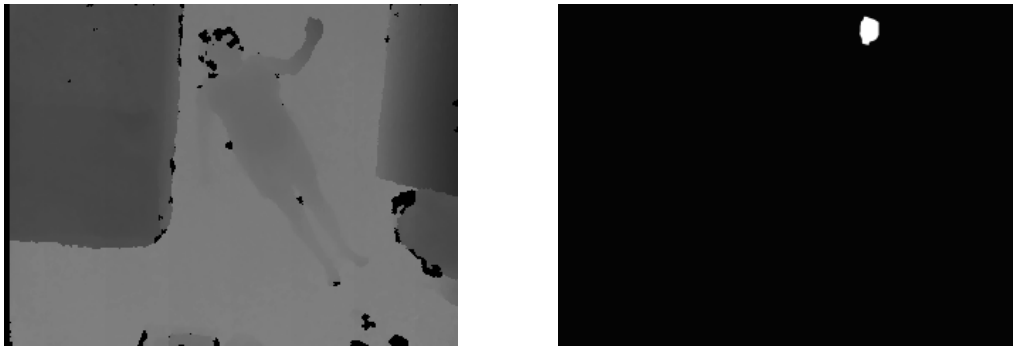
de profundidade do Kinect. Foi escolhida a linguagem C++ para facilitar a integração.

A classe principal da detecção da mão é o *handStream*, que faz uso primeiramente da *tree* para encontrar a probabilidade do *pixel* ser uma mão e, depois, usa a *MShift* para fazer a comparação com os demais *pixels* e dizer qual a posição mais provável de ser uma mão. Já as classes *bestPath* e *fallDetection* servem para informar se ouve ou não uma queda e, inclusive, se houve ou não o gesto. Todos os programas feitos estão no Apêndice B em diante.

3.2.1 Detecção da posição da mão

Para a detecção da posição, primeiramente, desenvolveu-se um programa que criou as características comparativas da RDT de forma aleatória, porém, de acordo com que ia avançando de camada, o tamanho dos vetores ia reduzindo. Desenvolveu-se, então, o programa com a RDT, sendo essa definida como a classe *tree* da Figura 10.

Desenvolveu-se um outro programa que salvava imagens de profundidade em um arquivo .xdr (que mantém os dados) e em .png (com perda de dados); 49 imagens foram salvas para serem usadas na etapa de aprendizagem da RDT. Os arquivos .png foram editadas para informar qual *pixel* pertencia a mão que deveria ser detectada. Para isso, fez-se uso do Photophop para indicar a área da imagem que estava a mão (parte branca da imagem) e a que não era (parte preta). Um exemplo dessas imagens está na Figura 11. Outro programa foi desenvolvido e que, com as imagens mapas (as imagens .png editadas) e os arquivos .xdr, definiu as distribuições de cada *leaf node*.



(a) (b)

Figura 11 – Imagem png (a) salva diretamente usada para a fase de aprendizagem e (b) editada para indicar a mão que deve encontrar.

Após teste, desenvolveram-se métodos para a classe *handStream* para ter uma melhor precisão e estabilidade na detecção da mão, esses fizeram uso de um filtro de média móvel e um estimador de melhor posição com base nos quatro *pixels* com maior densidade estimada obtida por 2.2 alterada, como é exposto posteriormente.

Para melhorar o tempo de processamento de cada quadro, a etapa de definição de probabilidade foi feita *pixel* sim, *pixel* não, para linha sim e outra linha não. Aqueles *pixels* que tiveram, ao seu redor, uma probabilidade maior que o limiar, foi também calculado sua probabilidade. A RDT, ao fim, ficou com 13 níveis, 1365 características e 3 limiares, enquanto a literatura possuía 3 RDTs com 20 de profundidade, 2000 características e 50 limiares para cada árvore (SHOTTON et al., 2013).

Para a etapa da definição das densidades estimadas, foram feitas algumas alterações. Na Equação 2.3, tirou-se a profundidade z, pois a mão sempre estará próximo do chão. E na Equação 2.2, a exponencial foi substituído por

$$\frac{1}{\|\hat{x} - \hat{x}_i\|^2 b_c + 1},$$

por ter operações mais simples e, logo, exigindo um menor tempo de processamento. Esse cálculo de densidade apenas é feito naqueles *pixels* que tem sua probabilidade calculada e está acima de um limiar.

3.2.2 Detecção do gesto de aceno

Desenvolveram-se uma biblioteca com as operações de um DTW e um programa para validá-la. Foram usados os cinco vídeos que o usuário caiu e, em seguida, fez o gesto, para definir o limiar de decisão para cada eixo. Todos os vídeos foram comparados com o primeiro, sendo que somente foram usados os DTWs da faixa de 80 quadros que teve o menor valor. Após isso, notou-se que a mediana era menor significamente que a média, então, usou-se a distribuição log-normal, o que fez o sistema apresentar uma maior precisão da detecção do gesto.

O valor de custo foi definido como

$$c_p = (dp_1 - dp_2)^2,$$

sendo que dp é a diferença entra a posição atual da mão hp_i menos a posição anterior dp_{i-1}, quer dizer, $dp = hp_i - hp_{i-1}$. Essa subtração entre o atual *pixel* e o *pixel* anterior foi utilizada para retirar o *offset*, pois, sem retirá-la, dependendo da posição na imagem, o DTW possuía um valor totalmente diferente.

3.2.3 Integração com o programa da Microsoft

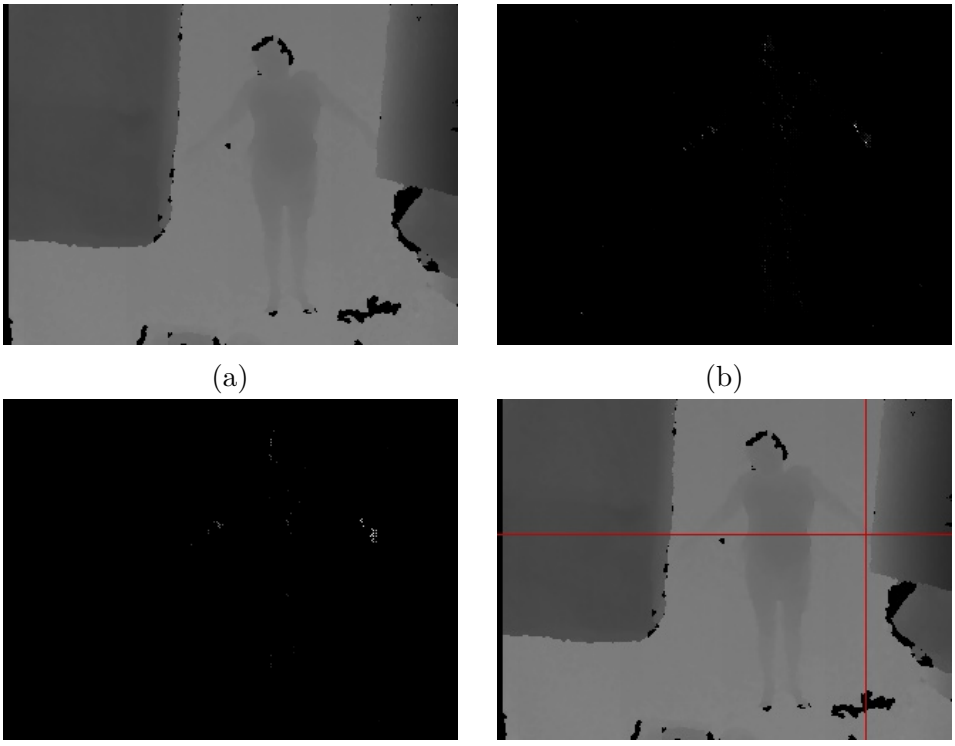
Criou-se uma classe (fallDetection na Figura 10) que, com a detecção da mão e com as operações DTW, informava se houve uma possível queda, o gesto de aceno e/ou a confirmação da queda. No arquivo disponibilizado pela Microsoft, no método que processa a imagem de profundidade, foi colocado a detecção de quedas (ver Apêndice G).

A imagem apresentada do computador foi editada para facilitar a compreensão do comportamento do sistema e facilitar a validação. No canto inferior direito, aparecem quadrados amarelo, azul e vermelho que indicam se houve uma possível queda, o gesto de aceno e confirmação de queda, respectivamente.

4 Resultados

A faixa de densidade estimada aceita da RDT, após a realização de testes com as imagens de aprendizagem, foi de 6.56261 ± 3.9416 (sem grandeza definida), o que equivale a 68% de probabilidade de ser uma mão. Já o limiar de decisão do DTW foi de 3042 (sem grandeza definida) para o eixo x e 3984 para o y, todo valor abaixo desse limiar foi considerado que houve o gesto desejado. Esses dois limiares representam 71% de chance de ser um gesto na distribuição logarítmica para cada eixo.

Na Figura 12, apresenta-se um exemplo de detecção de uma mão. Nota-se uma diferença leve, porém de extrema importância, entre 12b e 12c. A posição da mão direita apresenta uma maior densidade comparada com as das demais parte do corpo. Isso foi devido às imagens de aprendizagem indicarem apenas a mão direita quando não estava em movimento. Na Figura 12d, a mão direita estimada é indicada no cruzamento entre as duas linhas vermelhas perpendiculares. O Kinect salva a imagem invertida, por isso a mão direita está no lado esquerdo do corpo da pessoa.



(a) (b)

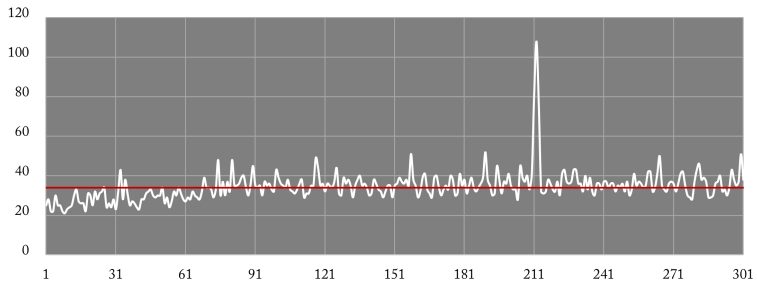
(c) (d)

Figura 12 – Processo de detecção de uma mão, sendo que (a) é a imagem com as probabilidades de cada *pixel* ser uma mão retiradas da RDT, (b) é apresentado a densidade ponderada calculada pelo método *mean shift* que compara a probabilidade com as dos *pixels* ao redor e (d) apresenta a posição da mão encontrada.

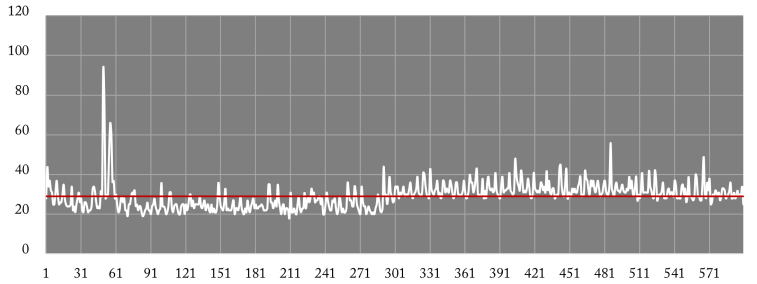
As médias do tempo de processamento dos quadros não divergiram muito entre os vídeos em que o usuário caía e, em seguida, fazia o gesto de aceno. Na figura 13, são apresentados os tempos de processamento dos quadros para cada vídeo. Os três primeiros gráficos, o usuário caiu em posição dorsal, enquanto os demais, em pronação.

A Figura 13a apresenta o tempo de processamento em ms para cada quadro de imagem de profundidade. Nota-se um pico logo após o quadro 241, esse é o instante no qual o usuário começa a fazer o gesto. Até o quadro 61, a maioria dos tempos ficam abaixo da média (reta vermelha) e, a partir dele, ele fica próximo ou acima dela. Este é momento que o usuário já está deitado no chão. Essa diferença fica mais acentuada nos outros gráficos da Figura 13. Com o usuário deitado no chão, há mais *pixels* a serem analisados, com mais deles com probabilidades acima do limiar, o que exige um maior tempo para o processamento.

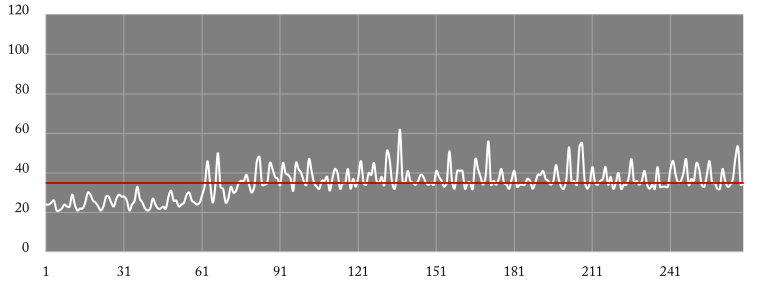
Na Figura 13b, entre as amostras 31 e 91, ocorre um pico. Nesse momento, o usuário passa um tempo considerável sentado no chão antes de se deitar. A posição da mão fica incerta e passar a ter uma variação de distância maior, pois fica alternando entre a mão esquerda, a direita, o joelho esquerdo e o direito. Uma possível razão é do sistema ter que fazer mais cálculos para estimar uma posição mais adequada.



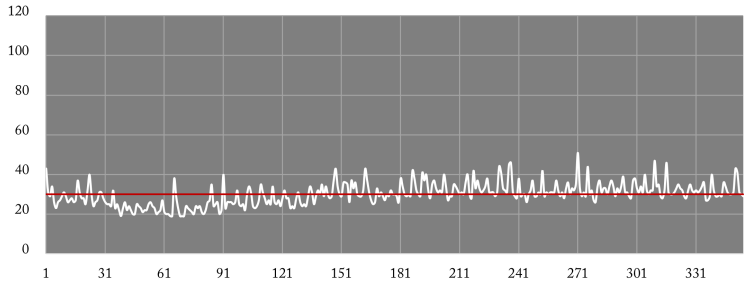
(a)



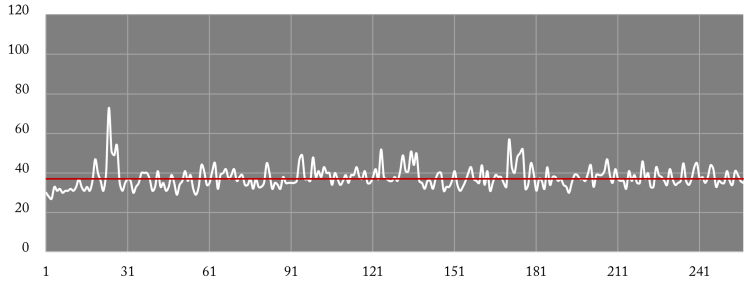
(b)



(c)



(d)

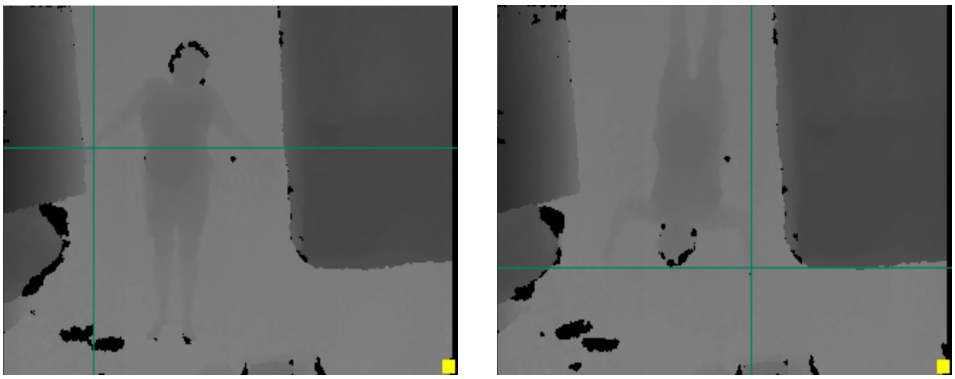


(e)

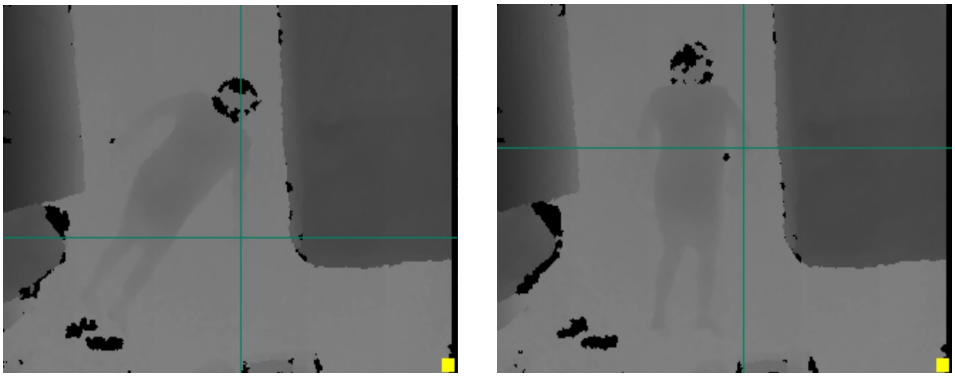
Figura 13 – Tempo de processamento (ms) do sistema para cada quadro na detecção da mão direita, da queda e/ou do gesto quando o usuário caiu e acenou para o sensor em diversas posições, tendo o eixo horizontal o número do quadro processado e a linha vermelha a média do tempo de processamento de cada quadro do vídeo.

A média de tempo de processamento de todos os vídeos da Figura 13 foi de 32ms com um desvio padrão de 7 ms, sendo que o sistema foi executado em um sistema quad-core. A média ficou abaixo do tempo entre quadros padrão, que é de 33 ms. Mesmo havendo tempo acima do padrão, a diferença não interferiu na fluidez nem nas detecções.

Na Figura 14, são apresentados alguns exemplos de posições testadas na detecção de quedas. Foram testadas as posições dorsal e em pronação, em pé referente a visão do sensor (ver Figura 14b e 14d), de cabeça para baixo (Figura 14b) e com o corpo inclinado (Figura 14c). Em nenhuma dessas, houve muita dificuldade na detecção, porém a pronação, ao rodar os vídeos testes, apresentou mais oscilações e dificuldades de detectar a mão, comparado com a dorsal.



(a) (b)

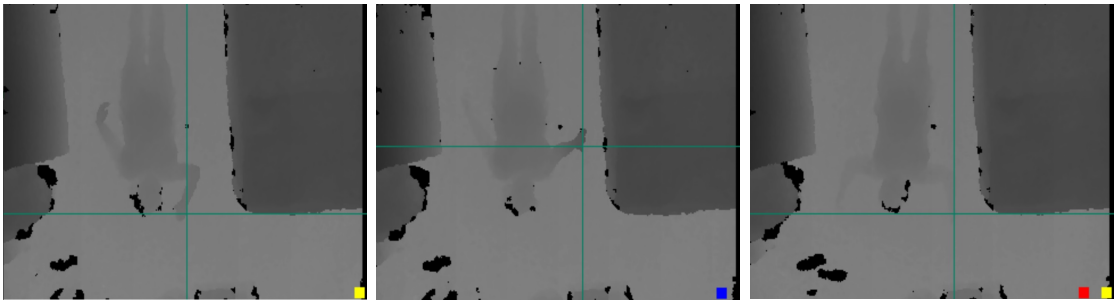


(c) (d)

Figura 14 – Detecção de uma possível queda em (a) e (b) em posição dorsal e (c) e (d) em pronação.

A detecção do gesto de aceno está na Figura 15b. Nela, há um sinal azul, que na Figura 15a não existe, isso porque o gesto foi detectado e que não houve uma queda, então a luz amarela desapareceu (não há uma possível queda) e ativou a azul (houve o gesto). Na Figura 15c mostra o que aconteceria se não houvesse o gesto. Um sinal vermelho seria aceso para confirmar a queda. A Figura 15c pertence àlguns quadros mais à frente da Figura 14b, onde a pessoa ficou imóvel por 5 segundos.

O sistema apresentou uma precisão de 72% na detecção de uma mão, próximo da precisão de (SHOTTON et al., 2013). Já na detecção de uma possível queda, em todos os vídeos testados foram detectadas, assim como na detecção do gesto. No entanto, apesar de não ter interferido nos resultados, houve falsas detecções do gesto desejado que ocorreram enquanto não havia ninguém deitado no chão.



(a) (b) (c)

Figura 15 – Em (a) houve a detecção de uma possível queda e a pessoa no chão começou a fazer o gesto para informar que está tudo certo, (b) o gesto foi detectado e o sinal de possível queda desativado e (c) caso não houvesse feito o gesto um sinal vermelho foi apresentado para confirmar a queda.

5 Conclusão

A cada ano, a proporção da população idosa tem crescido no Brasil, aumentando a necessidade de estudos e pesquisas que auxilie a vida dessas pessoas. A queda é a principal causa que leva idosos a um hospital. Decidiu-se desenvolver um sistemas de detecção de quedas com interação de gestos fazendo uso do sensor de profundidade do Kinect para garantir a uma fácil utilização e privacidade do usuário.

O sistema foi capaz de detectar as quedas e os gestos como esperado. A precisão na detecção da mão usando o método padrão do Kinect ficou perto daquela apresentada na literatura, sendo que o método para uma precisão mais correta seria retirar novas imagens de mais vídeos de teste e refazer o procedimento da identificação da precisão na detecção da mão.

Durante todo o processo, a câmera RGB do Kinect esteve desativada, para garantir que não houvesse invasão e perda da privacidade do usuário. O processamento das imagens de profundidade, algumas vezes, levou mais tempo que a intervalo entre quadro, no entanto, não gerou problemas nas detecções e na fluidez do sistema.

Para projetos futuros, pode-se utilizar mais imagens de aprendizagem para melhorar a capacidade de detecção da mão e diminuir as incertezas. Além disso, podem-se fazer mais amostras de quedas e gestos para evitar falsos positivos e saber a correta precisão do sistema. É aconselhável, também, fazer uso de uma linguagem mais adequada para operações aritméticas recursivas, como o Python, para a Árvore de Decisão Aleatória (RDT).

Referências

ACKERMAN, E. *Why You Should Be Very Skeptical of Ring's Indoor Security Drone*. 2020. Postado no IEEE Spectrum. Acesso em: 26 de outubro de 2020. Disponível em: <https://spectrum.ieee.org/automaton/robotics/drones/ring-indoor-security-drone>. Citado na página 15.

APPLE. *Usar a detecção de queda no Apple Watch*. c2020. Acesso em: 19 de novembro de 2020. Disponível em: <https://support.apple.com/pt-br/HT208944>. Citado 3 vezes nas páginas 9, 17 e 18.

BIAN, Z. et al. Fall detection based on body part tracking using a depth camera. *IEEE Journal of Biomedical and Health Informatics*, v. 19, n. 2, p. 430–439, 2015. Citado na página 28.

BISWAS, K. K.; BASU, S. K. *Gesture recognition using Microsoft Kinect®*. 2011. 100-103 p. Citado 6 vezes nas páginas 9, 20, 24, 25, 27 e 74.

BROWNLEE, J. *A Gentle Introduction to Long Short-Term Memory Networks by the Experts*. 2017. Publicado pelo site Machine Learning Mastery. Acesso em: 12 de novembro de 2020. Disponível em: <https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/>. Citado na página 28.

CARMODY, T. *How Motion Detection Works in Xbox Kinect*. 2010. Acesso em: 21 de outubro de 2020. Disponível em: <https://www.wired.com/2010/11/tonights-release-xbox-kinect-how-does-it-work/>. Citado 2 vezes nas páginas 9 e 20.

CHASE, C. A. et al. Systematic review of the effect of home moditication and fall prevation programs on falls and the performance of community-dwelling older adults. *American Journal of Occupational Therapy*, v. 66, n. 3, p. 284–291, 2012. Citado na página 14.

CIRE, B. *World's older population grows dramatically: NIH-funded Census Bureau report offers details of global aging phenomenon*. 2016. Acesso em: 24 de outubro de 2020. Disponível em: <https://www.nih.gov/news-events/news-releases/worlds-older-population-grows-dramatically#:~:text=The%20world's%20older%20population%20continues,by%202050%20(1.6%20billion).> Citado na página 14.

COMISKEY, C. M. et al. The breathe project, a mobile application, video-monitoring system in family homes as an aid to the caring role: Needs, acceptability and concerns of informal carers. *Digital Health*, v. 4, p. 1–8, 2018. Citado na página 15.

COMSTOCK, J. *How fall detection is moving beyond the pendant*. 2019. Acesso em: 20 de novembro de 2020. Disponível em: <https://www.mobihealthnews.com/content/how-fall-detection-moving-beyond-pendant>. Citado na página 18.

GREGG, S. *Fall Call Now: Prototype Build-Out*. 2016. Acesso em: 19 de novembro de 2020. Disponível em: <https://www.medstartr.com/project/detail/1146-Prototype-Build-Out>. Citado na página 18.

IBGE. *Projeção da população do Brasil e Unidades da Federação por sexo e idade para o período 2010-2060*. [S.l.], 2018. Obtido pelo link: <https://www.ibge.gov.br/estatisticas/sociais/populacao/9109-projecao-da-populacao.html?=&t=downloads>, acesso em 25 de outubro de 2020. Citado na página 14.

IRWIN, W. *SureSafe Alarms Reviews*. 2020. Acesso em: 21 de outubro de 2020. Disponível em: <https://www.reviews.co.uk/company-reviews/store/sure-safe-?utm_source=sure-safe-&utm_medium=widget&utm_campaign=carousel-inline>. Citado na página 17.

JURAFSKY, D.; MARTIN, J. H. *Hidden Markov Models*. 2019. Acesso em: 06 de novembro de 2020. Disponível em: <https://web.stanford.edu/~jurafsky/slp3/A.pdf>. Citado na página 25.

LEPETIT, V.; LAGGER, P.; FUA, P. Randomized trees for real-time keypoint recognition. In: . [S.l.: s.n.], 2005. v. 2, p. 775– 781 vol. 2. ISBN 0-7695-2372-2. Citado na página 23.

MüLLER, M. Dynamic time warping. *Information retrieval for music and motion*, 2007. Citado na página 26.

MUBASHIR, M.; SHAO, L.; SEED, L. A survey on fall detection: Principles and aproaches. *Neurocomputing*, v. 100, p. 144–152, 2013. Citado 2 vezes nas páginas 14 e 15.

MUTTO, C.; GUIDO, P.; CORTELAZZO, M. Time-of-flight cameras and microsoft kinect™. *SpringerBriefs in Electrical and Computer Engineering*, 01 2012. Citado 3 vezes nas páginas 20, 21 e 22.

NEGRINI, E. L. D. et al. Quem são e como vivem os idosos que moram sozinhos no brasil. *Rev. Bras. Geriatr. Georontol.*, v. 21, n. 5, p. 542–550, 2018. Citado 2 vezes nas páginas 14 e 15.

OWANO, N. *Automatic fall detector attaches to wall, no wearable needed*. 2018. Acesso em: 18 de novembro de 2020. Disponível em: <https://techxplore.com/news/2018-10-automatic-fall-detector-wall-wearable.html>. Citado na página 19.

PELLIZON, J. *Acidentes dentro de casa podem ser graves: 75% das lesões em idosos atendidos na rede pública são resultado desse tipo de ocorrência*. c2020. Postado em um website da USP. Acesso em: 24 de outubro de 2020. Disponível em: <http://www2.eca.usp.br/njsaoremo/?p=3893>. Citado na página 14.

RAHEJA, J. et al. Robust gesture recognition using kinect: A comparison between dtw and hmm. *Optik*, v. 126, n. 11, p. 1098 – 1104, 2015. ISSN 0030-4026. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0030402615000819>. Citado 3 vezes nas páginas 25, 26 e 27.

ROUSE, M. *Kinect - Definition*. 2011. Acesso em: 21 de outubro de 2020. Disponível em: <https://searchhealthit.techtarget.com/definition/Kinect>. Citado na página 19.

SHARMA, K. Kinect sensor based object feature estimation in depth images. *International Journal of Signal Processing, Image Processing and Pattern Recognition*, v. 8, n. 12, p. 237–256, 2015. Citado 4 vezes nas páginas 9, 20, 21 e 22.

SHOTTON, J. et al. Real-time human pose recognition in parts from single depth images. *Communications of the ACM*, v. 56, n. 1, p. 116 – 124, 2013. ISSN 00010782. Disponível em: <http://search-ebscohost-com.ez54.periodicos.capes.gov.br/login.aspx?direct=true&db=iih&AN=84631519&lang=pt-br&site=ehost-live>. Citado 8 vezes nas páginas 9, 22, 23, 24, 28, 30, 33 e 38.

SURESAFE. *Fall Alert*. c2020. Acesso em: 21 de outubro de 2020. Disponível em: <https://www.personalalarms.org/product/fall-alert-pendant-from-suresafe/>. Citado 2 vezes nas páginas 9 e 17.

VERGER, R. *The Apple Watch learned to detect falls using data from real human mishaps*. 2018. Acesso em: 19 de novembro de 2020. Disponível em: <https://www.popsci.com/apple-watch-fall-detection/>. Citado na página 18.

WALABOT. *How Our Fall Deection System Works*. c2020. Acesso em: 20 de novembro de 2020. Disponível em: <https://walabot.com/walabot-home/how-it-works>. Citado 3 vezes nas páginas 9, 18 e 19.

WANG, X.; ELLUL, J.; AZZOPARDI, G. Elderly fall detection systems: A literature survey. *Frontiers in Robotics and AI*, v. 7, n. 71, 2020. Citado 2 vezes nas páginas 14 e 73.

XU, T.; ZHOU, Y. Elders' fall detection based on biomechanical features using depth camera. *International Journal of Wavelets, Multiresolution and Information Processing*, World Scientific, v. 16, n. 2, 2018. Citado 2 vezes nas páginas 27 e 28.

ZHANG, J. *Dynamic Time Warping: Explanation and Code Implementation*. 2020. Acesso em: 06 de novembro de 2020. Disponível em: <https://towardsdatascience.com/dynamic-time-warping-3933f25fcdd>. Citado na página 26.

Apêndices

APÊNDICE A – Cálculo de precisão do detector de gestos que faz uso de uma SVM

Palma:

$$\frac{449}{449+29+19+2} = \frac{449}{499} = 90\%$$

Ligando:

$$\frac{516}{35+516+4+3+2+4+22+17} = \frac{516}{603} = 86\%$$

Saudação:

$$\frac{464}{28+464+8+8+3+1} = \frac{464}{512} = 91\%$$

Acenar:

$$\frac{302}{2+12+302} = \frac{302}{316} = 96\%$$

Assentindo negativamente:

$$\frac{195}{85+46+195+9=1+1} = \frac{195}{337} = 58\%$$

Assentindo positivamente:

$$\frac{186}{7+98+7+14+186+25+6} = \frac{186}{343} = 54\%$$

Clasp:

$$\frac{516}{7+19+6+516+1} = \frac{516}{549} = 94\%$$

Descansar:

$$\frac{484}{1+8+1+19+3+2+484} = \frac{484}{518} = 93\%$$

APÊNDICE B – Arquivo RDT.h que contém a classe *tree*

```
1  #pragma once
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include<stdbool.h>
6
7  #include <iostream>
8  using namespace std;
9
10 #include"offsetsAndThresholds.h"
11 #include"leafNodes.h"
12
13 class tree : private OffsAndThres,private Nodes{
14
15 public:
16     int* states;
17
18     tree(int, int,float**);
19     ~tree(void);
20     float getProbPixel(int, int);
21 private:
22     int width;
23     int height;
24     int countStates;
25     float** depthImage;
26     int x, y;
27     float one_over_depth;
28     float thresholds[3] = { 0.01f, 0.001f, 0.1f };
29
30     float getThreshold(int, int);
31     float getFeature(int*, int*);
32     float getDepthOffset( int*);
33     void generateRDT();
34 };
35
```

```
36 tree::tree(int widthImage, int heightImage,float** depthImageVector)
   :
37     width(widthImage),
38     height(heightImage),
39     depthImage(depthImageVector),
40     x(0),
41     y(0),
42     countStates(0)
43 {
44     states = new int[12];
45 };
46 tree::~tree() {
47     delete[] states;
48 };
49
50 float tree::getProbPixel(int newX, int newY) {
51
52     x = newX;
53     y = newY;
54     generateRDT();
55     float* probsAllParts = getProbability(states[countStates]);
56     float probPHand = probsAllParts[0];
57     delete[] probsAllParts;
58     if (depthImage[x][y] == -1) {
59         return -1;
60     } else if (states[countStates] > 4095)
61         return probPHand;
62     else
63         return -1;
64 };
65
66 float tree::getThreshold(int node, int layer) {
67     float threshold = 0;
68     if (layer % 2 == 0)
69         threshold = thresholds[0];
70     else
71         if (node % 2 == 0)
72             threshold = thresholds[1];
73         else
74             threshold = thresholds[2];
75     return threshold;
76 }
```

```
77
78 float tree::getFeature(int* vectorU, int* vectorV) {
79     if (depthImage[x][y] == 0)
80         return 0;
81     else if (depthImage[x][y] == -1)
82         return -1;
83     else {
84         float depthU = getDepthOffset(vectorU);
85         float depthV = getDepthOffset(vectorV);
86         return (depthU - depthV);
87     }
88 }
89
90 float tree::getDepthOffset( int* offset) {
91     float offsetDepth = 1;
92     int pixelOffsetX = x + one_over_depth*offset[0];
93     int pixelOffsetY = y + one_over_depth*offset[1];
94
95     if ((pixelOffsetX > width - 1) || pixelOffsetX < 0
96             ||(pixelOffsetY > height - 1) || pixelOffsetY < 0)
97         offsetDepth = 0;
98     else if (depthImage[pixelOffsetX][pixelOffsetY] < 0)
99         offsetDepth = 0;
100    else
101        offsetDepth = depthImage[pixelOffsetX][pixelOffsetY];
102
103    return offsetDepth;
104 }
105
106 void tree::generateRDT() {
107    delete[] states;
108    states = new int[13];
109    countStates = 0;
110    if (depthImage[x][y] != -1) {
111        int currentState = 1;
112        int nextState = 1;
113        one_over_depth = 1/depthImage[x][y];
114        int* U = NULL;
115        int* V = NULL;
116        float feature = -1;
117        float threshold = -1;
118        while (nextState / 10000 == 0) {
```

```
119             currentState = nextState;
120             bool splitState = false;
121             U = getOffsets(currentState, 'U');
122             V = getOffsets(currentState, 'V');
123             feature = getFeature(U, V);
124             for (int e = 0; e < 2; e++) {
125                 splitState = false;
126                 threshold = getThreshold(currentState, countStates);
127                 if (feature > threshold || feature < -threshold)
128                     splitState = true;
129                 nextState = getNextNode(currentState, splitState);
130                 states[countStates] = currentState;
131                 currentState = nextState;
132                 ++countStates;
133             }
134             free(U);
135             free(V);
136         }
137         states[countStates] = nextState - 10000;
138         states[12] = nextState - 10000;
139     }
140     else
141         states[countStates] = -1;
142 }
```

APÊNDICE C – Arquivo MShift.h que contém a classe *MShift*

```
1  #pragma once
2  #include<math.h>
3  #include<iostream>
4
5  using namespace std;
6
7  class MShift {
8      const int sizeEstWindow = 10;
9  public:
10     MShift(float**, float**, int, int);
11     ~MShift();
12     float getDensityEst(int, int);
13 private:
14     float** depthImage;
15     float** probParts;
16     int width;
17     int height;
18     int xX, yY;
19     float depthAtPixel;
20
21     float getPixeisDifs(int*);
22     float getWeightExp(int*);
23 };
24
25 MShift::MShift(float** idepthImage, float** iprobParts, int iwidth,
       int iheight):
26     depthImage(idepthImage),
27     probParts(iprobParts),
28     width(iwidth),
29     height(iheight),
30     xX(0),
31     yY(0),
32     depthAtPixel(0)
33 {
34 };
```

```
35
36 MShift::~MShift() {
37 };
38
39 float MShift::getPixeisDifs(int* otherPixel) {
40     float sizeDif = (xX - otherPixel[0]) * (xX - otherPixel[0]);
41     sizeDif += (yY - otherPixel[1])* (yY - otherPixel[1]);
42     sizeDif = 1 / (sizeDif * 0.05 + 1);
43     return sizeDif;
44 };
45
46 float MShift::getWeightExp(int* otherPixel) {
47     float depthAtOtherPixel = depthImage[otherPixel[0]][otherPixel
       [1]];
48     float weightExp = probParts[otherPixel[0]][otherPixel[1]];
49     weightExp *= (weightExp != 0 ? getPixeisDifs(otherPixel):0);
50     return weightExp;
51 };
52
53 float MShift::getDensityEst(int coordX, int coordY) {
54     xX = coordX;
55     yY = coordY;
56     float densityEstimator = 0;
57     int numberPixeis = 0;
58     depthAtPixel = depthImage[xX][yY];
59     if (depthAtPixel > 0) {
60         for (int x = (xX-sizeEstWindow); x < (xX+sizeEstWindow); x++)
61             for (int y = (yY-sizeEstWindow); y < (yY+sizeEstWindow);
       y++) {
62                 if (x < width && x >= 0 && y < height && y >= 0) {
63                     int otherPixel[2] = { x,y };
64                     densityEstimator += (depthImage[x][y] > 0 &&
       probParts[x][y] > 0 ? getWeightExp(otherPixel) : 0);
65                     numberPixeis++;
66                 }
67             }
68     }
69     else
70         densityEstimator = -1;
71     return densityEstimator;
72 };
```

APÊNDICE D – Arquivo BodyPartTracking.h que contém a classe *handStream*

```
1  #pragma once
2  #include <iostream>
3  #include<fstream>
4  #include <opencv2/opencv.hpp>
5
6  using namespace cv;
7
8  #include"RDT.h"
9  #include"MShift.h"
10
11
12 class handStream {
13 public:
14     handStream(float***, int, int, int);
15     ~handStream();
16     bool update();
17     int* getHandPosition();
18     int** getAllHandPositions();
19     float* densityMax = new float[4]{ 0 };
20 private:
21     float*** depthImagePt;
22     float** depthBackground;
23     int width, height, numberBytes;
24     float** probs;
25     float** densities;
26     float** depthImage;
27     int* positionBuffer[20];
28     bool statusBackground;
29     const float thresholdDens = 0.14f;
30     int** position = new int*[4];
31
32     bool readBackground();
33     void saveDensity(float, int, int);
```

```
34      void setMPPosition();
35      int* desvPosition();
36      int* getMed();
37      void positionFilter();
38
39  };
40
41  handStream::handStream(float*** depthStream, int widthS, int heightS,
        int numberOfBytes) :
42      width(widthS),
43      height(heightS),
44      numberBytes(numberOfBytes),
45      depthImagePt(depthStream)
46  {
47      statusBackground = readBackground();
48      probs = new float* [width];
49      for (int i = 0; i < width; i++)
50          probs[i] = new float[height];
51      densities = new float* [width];
52      for (int i = 0; i < width; i++)
53          densities[i] = new float[height];
54      for (int i = 0; i < 20; i++)
55          positionBuffer[i] = new int[2]{ 0};
56      for (int i = 0; i < 4; i++)
57          position[i] = NULL;
58  };
59
60  handStream::~handStream()
61  {
62      for (int i = 0; i < width; i++) {
63          delete[] depthBackground[i];
64          delete[] densities[i];
65          delete[] probs[i];
66      }
67      delete[] probs;
68      delete[] densities;
69      delete[] depthBackground;
70  }
71
72  bool handStream::update() {
73
74      if (!statusBackground)
```

```
75          return false;
76
77      depthImage = depthImagePt[0];
78      unique_ptr<tree> pixelTree(new tree(width, height, depthImage));
79
80      for (int i = 0; i < width; i++)
81          if (i % 2 == 0)
82              for (int j = 0; j < height; j++)
83                  if (depthImage[i][j] < 0.98 * depthBackground[i][j]
    && depthImage[i][j] != -1)
84                      if (j % 2 == 0)
85                          probs[i][j] = pixelTree->getProbPixel(i, j);
86                      else
87                          probs[i][j] = 0;
88
89      for (int i = 1; i < width - 1; i++)
90          if (i % 2 != 0)
91              for (int j = 1; j < height - 1; j++)
92                  if (j % 2 != 0) {
93                      float forwardProb = (probs[i + 1][j + 1] > 0 ?
    probs[i + 1][j + 1] : 0) +
94                          (probs[i][j + 1] > 0 ? probs[i][j + 1] : 0) +
95                          (probs[i + 1][j] > 0 ? probs[i + 1][j] : 0);
96                      if (depthImage[i][j] < depthBackground[i][j] &&
    depthImage[i][j] != -1
97                          && forwardProb > thresholdDens * 0.5)
98                          probs[i][j] = pixelTree->getProbPixel(i, j);
99                  }
100     pixelTree.reset();
101
102     for (int d = 0; d < 4; d++) {
103         densityMax[d] = 0;
104         if (position[d] != NULL) {
105             delete[] position[d];
106             position[d] = NULL;
107         }
108     }
109
110     unique_ptr<MShift> density(new MShift(depthImage, probs, width,
    height));
111     for (int i = 0; i < width; i++)
112         for (int j = 0; j < height; j++)
```

```
113             if (depthImage[i][j] > 0 && depthImage[i][j] < 0.98 *
    depthBackground[i][j] && probs[i][j] > thresholdDens) {
114                 densities[i][j] = density->getDensityEst(i, j);
115                 if (densities[i][j] > 6.56261 - 3.9416 && densities[i
    ][j] < 6.56261 + 3.9416)
116                     saveDensity(densities[i][j], i, j);
117             }
118     density.reset();
119     setMPPosition();
120     positionFilter();
121
122     if (position[0] == NULL)
123         return false;
124
125     if (depthImage[position[0][0]][position[0][1]] < 0.97 *
    depthBackground[position[0][0]][position[0][1]])
126         return false;
127     else if (depthImage[position[0][0]][position[0][1]] > 0.97 *
    depthBackground[position[0][0]][position[0][1]] &&
128         depthImage[position[0][0]][position[0][1]] < depthBackground[
    position[0][0]][position[0][1]]) {
129         return true;
130     }
131     else
132         return false;
133
134 }
135
136 void handStream::saveDensity(float newDensity, int x, int y) {
137     for(int i = 3; i >= 0; i--)
138         if (newDensity > densityMax[i]) {
139             delete[] position[0];
140             for (int c = 0; c < i; c++) {
141                 densityMax[c] = densityMax[c + 1];
142                 position[c] = position[c + 1];
143             }
144             densityMax[i] = newDensity;
145             position[i] = new int[2]{ x,y };
146             break;
147         }
148 }
149
```

```
150 void handStream::setMPPosition() {
151     bool thereIsZero = false;
152     for(int i = 0; i < 4; i++)
153         if (densityMax[i] == 0) {
154             thereIsZero = true;
155             break;
156         }
157     if (thereIsZero == true) {
158         delete[] position[0];
159         position[0] = NULL;
160         return;
161     }
162     else {
163         int* desv = desvPosition();
164         if (desv[0] > 5 || desv[1] > 5) {
165             int* med = getMed();
166             for (int i = 0; i < 4; i++) {
167                 if ((position[i][0] - med[0] > 5 || position[i][0] -
    med[0] < -5) ||
168                     (position[i][1] - med[1] > 5 || position[i][1] -
    med[1] < -5)) {
169                     densityMax[i] = 0;
170                 }
171             }
172         }
173     }
174     position[0] = getMed();
175     return;
176 }
177
178 int* handStream::desvPosition() {
179     int med[2] = { 0 };
180     int desN[2] = { 0 };
181     for (int i = 0; i < 4; i++) {
182         med[0] += position[i][0];
183         med[1] += position[i][1];
184     }
185     med[0] /= 4;
186     med[1] /= 4;
187     for (int i = 0; i < 4; i++) {
188         desN[0] += (position[i][0] - med[0] > 0 ? position[i][0] -
    med[0] : med[0] - position[i][0]);
```

```
189         desN[1] += (position[i][1] - med[1] > 0 ? position[i][1] -
    med[1] : med[1] - position[i][1]);
190     }
191     desN[0] /= 4;
192     desN[1] /= 4;
193 
194     return desN;
195 }
196 
197 int* handStream::getMed() {
198     int* med = new int[2] { 0 };
199     int total = 0;
200     for (int i = 0; i < 4; i++) {
201         if (densityMax[i] != 0) {
202             med[0] += position[i][0];
203             med[1] += position[i][1];
204             total++;
205         }
206     }
207     if (total != 0) {
208         med[0] /= total;
209         med[1] /= total;
210     }
211     else {
212         delete[] med;
213         med = NULL;
214     }
215 
216     return med;
217 }
218 
219 int* handStream::getHandPosition() {
220     return position[0];
221 }
222 int** handStream::getAllHandPositions() {
223     return position;
224 }
225 
226 bool handStream::readBackground()
227 {
228     const int numberOfSamples = width * height;
229 
```

```
230     depthBackground = new float* [width];
231     for (int x = 0; x < width; x++)
232         depthBackground[x] = new float[height];
233     string filename = "C:\\Users\\victo\\OneDrive - unb.br\\TCC - 
    Documentos\\FallAndGestureDetection\\BodyPartTracking\\Images\\
    background.png";
234 
235     Mat depthImageFile = imread(filename);
236 
237     if (!depthImageFile.empty()) {
238         for (int x = 0; x < width; x++)
239             for (int y = 0; y < height; y++) {
240                 float bytePixel = depthImageFile.at<Vec3b>(y, x).val
    [0];
241                 depthBackground[x][y] = bytePixel / 255;
242             }
243     }
244     else{
245         _RPT1(0, "File %s was not found\n", filename.c_str());
246         return false;
247     }
248     return true;
249 }
250 
251 void handStream::positionFilter() {
252     int sumDX = 0;
253     int sumDY = 0;
254     int sumP = 0;
255 
256     delete[] positionBuffer[0];
257     for (int i = 0; i < 19; i++) {
258         positionBuffer[i] = positionBuffer[i + 1];
259     }
260     if (position[0] != NULL)
261         positionBuffer[19] = position[0];
262     else
263         positionBuffer[19] = new int[2]{ 0,0 };
264 
265     int* base = NULL;
266     int endBuffer = 0;
267     for (int i = 19; i >= 10; i--)
268         if (positionBuffer[i][0] != 0) {
```

```
269             base = positionBuffer[i];
270             endBuffer = i;
271             break;
272         }
273
274     for (int i = 0; i < endBuffer; i++) {
275         if (positionBuffer[i][0] != 0 && positionBuffer[i][1] != 0 &&
     base != NULL &&
276             (base[0] - positionBuffer[i][0])*(base[0] -
    positionBuffer[i][0]) < 20 &&
277             (base[1] - positionBuffer[i][1])*(base[1] -
    positionBuffer[i][1]) < 20){
278             sumDX += (i + 1) * positionBuffer[i][0];
279             sumDY += (i + 1) * positionBuffer[i][1];
280             sumP += (i + 1);
281         }
282     }
283     if (base != NULL) {
284         sumDX += (20 - (19 - endBuffer)) * base[0];
285         sumDY += (20 - (19 - endBuffer)) * base[1];
286         sumP += (20 - (19 - endBuffer));
287     }
288     if (sumP == 0)
289         position[0] = NULL;
290     else {
291         sumDX /= sumP;
292         sumDY /= sumP;
293         position[0] = new int[2]{ sumDX,sumDY };
294     }
295 }
```

APÊNDICE E – Arquivo DTW.h que contém a classe *bestPath*

```
1  #ifndef DTW_H_INCLUDED
2  #define DTW_H_INCLUDED
3
4  #include <math.h>
5  #include <stdio.h>
6  #include <stdbool.h>
7  #include <iostream>
8  #include<fstream>
9  using namespace std;
10
11 class bestPath {
12 public:
13     int** arrayTwo;
14     int sizeTwo;
15
16     bestPath(int);
17     ~bestPath();
18     bool setNewPosition(int*);
19     float GetDTW(int, int);
20 private:
21     int** baseArrayOne;
22     int sizeBaseOne;
23     int** baseArrayTwo;
24     int sizeBaseTwo;
25
26     float** GetDNMMatrix(int*, int*, int, int);
27     float GetCost(float, float);
28     float GetMin(float, float, float);
29     int** ReadBaseGesture(int);
30 };
31
32  bestPath::bestPath(int sizeVector) :
33      sizeTwo(sizeVector)
34  {
35      baseArrayTwo = ReadBaseGesture(1);
```

```
36      sizeBaseTwo = sizeBaseOne;
37      baseArrayOne = ReadBaseGesture(0);
38      arrayTwo = new int* [2];
39      for (int i = 0; i < 2; i++)
40          arrayTwo[i] = new int[sizeTwo] {0};
41  }
42
43  bestPath::~bestPath()
44  {
45      delete[] baseArrayOne[0];
46      delete[] baseArrayOne[1];
47      delete[] baseArrayOne;
48      delete[] arrayTwo[0];
49      delete[] arrayTwo[1];
50      delete[] arrayTwo;
51  }
52
53  bool bestPath::setNewPosition(int* position) {
54      for (int i = 0; i < sizeTwo - 1; i++) {
55          arrayTwo[0][i] = arrayTwo[0][i + 1];
56          arrayTwo[1][i] = arrayTwo[1][i + 1];
57      }
58      arrayTwo[0][sizeTwo - 1] = position[0];
59      arrayTwo[1][sizeTwo - 1] = position[1];
60      return true;
61  }
62
63 float bestPath::GetDTW(int whichDimension, int whichBase)
64 {
65     int* newArrayTwo = new int[sizeTwo];
66     newArrayTwo[0] = 0;
67     for (int i = 1; i < sizeTwo; i++)
68         newArrayTwo[i] = arrayTwo[whichDimension][i] - arrayTwo[
               whichDimension][i - 1];
69
70     if (whichBase == 0) {
71         float** DNM = GetDNMMatrix(baseArrayOne[whichDimension],
               newArrayTwo, sizeBaseOne, sizeTwo);
72         float DTW1 = DNM[sizeBaseOne - 1][sizeTwo - 1];
73         for (int i = 0; i < sizeBaseOne; i++)
74             delete[] DNM[i];
75         delete[] DNM;
```

```
76          return DTW1;
77      }
78      else {
79          float** DNM = GetDNMMatrix(baseArrayTwo[whichDimension],
                newArrayTwo, sizeBaseTwo, sizeTwo);
80          float DTW2 = DNM[sizeBaseTwo - 1][sizeTwo - 1];
81          for (int i = 0; i < sizeBaseTwo; i++)
82              delete[] DNM[i];
83          delete[] DNM;
84          return DTW2;
85      }
86  }
87
88  float** bestPath::GetDNMMatrix(int* vectorOfPositions1, int*
        vectorOfPositions2, int length1, int length2)
89  {
90      const int N = length1;
91      const int M = length2;
92
93      float** DNM = new float*[N];
94      for (int i = 0; i < N; i++)
95          DNM[i] = new float[M];
96
97      DNM[0][0] = GetCost(vectorOfPositions1[0], vectorOfPositions2[0])
            ;
98
99      for (int i = 1; i < N; i++)
100         DNM[i][0] = DNM[i-1][0] +
101                     GetCost(vectorOfPositions1[i], vectorOfPositions2
                            [0]);
102
103     for (int j = 1; j < M; j++)
104         DNM[0][j] = DNM[0][j-1] +
105                     GetCost(vectorOfPositions1[0], vectorOfPositions2
                            [j]);
106
107     for (int i = 1; i < N; i++)
108         for (int j = 1; j < M; j++)
109                     DNM[i][j] = GetMin(DNM[i - 1][j - 1], DNM[i - 1][
                            j], DNM[i][j - 1])
110                     + GetCost(vectorOfPositions1[i],
                            vectorOfPositions2[j]);
```

```
111
112     return DNM;
113 }
114
115
116 float bestPath::GetCost(float number1, float number2)
117 {
118     float cost = (number1 - number2)*(number1 - number2);
119     return cost;
120 }
121
122
123 float bestPath::GetMin(float number1, float number2, float number3)
124 {
125     float numberMin = number1;
126
127     if (numberMin > number2)
128         numberMin = number2;
129     if (numberMin > number3)
130         numberMin = number3;
131
132     return numberMin;
133 }
134
135 int** bestPath::ReadBaseGesture(int witchBase)
136 {
137     const int numberOfDimentions = 2;
138     const int numberOfBytes = 4;
139     int** Streams = new int*[numberOfDimentions];
140     ifstream Stream;
141     string fileName;
142     if(witchBase == 0)
143         fileName = "C:/Users/victo/OneDrive - unb.br/TCC - Documentos
            /FallAndGestureDetection/GestureDetection/gestures/
            baseGesture.bin";
144     else
145         fileName = "C:/Users/victo/OneDrive - unb.br/TCC - Documentos
            /FallAndGestureDetection/GestureDetection/gestures/
            baseGesture2.bin";
146     sizeBaseOne = 0;
147
148     Stream.open(fileName, ios::binary);
```

```
149
150     Stream.seekg(0, Stream.end);
151     long int lengthOfFile = Stream.tellg();
152     Stream.seekg(0,Stream.beg);
153     lengthOfFile--;
154     lengthOfFile /= (numberOfBytes * 2 + 1);
155     lengthOfFile--;
156
157     Streams[0] = new int[lengthOfFile];
158     Streams[1] = new int[lengthOfFile];
159     for(int m = 0; m < lengthOfFile; m++){
160         char initData;
161         do{
162             Stream.get(initData);
163         }while(initData != 'D'&& initData != EOF);
164
165         if (initData == EOF)
166         {
167             for (int i = m; i < lengthOfFile; i++) {
168                 Streams[0][m] = Streams[0][m - 1];
169                 Streams[1][m] = Streams[1][m - 1];
170             }
171             break;
172         }
173         Stream.read((char*)&Streams[0][m],numberOfBytes);
174         Stream.read((char*)&Streams[1][m],numberOfBytes);
175
176         bool notCount = true;
177         for(int d = 0; d < numberOfDimentions; d++){
178             if((Streams[d][m] > 640) || (Streams[d][m] < 0)){
179                 m--;
180                 notCount = true;
181                 lengthOfFile--;
182                 break;
183             }else
184                 notCount = false;
185         }
186         if (notCount == false)
187             sizeBaseOne++;
188     }
189     Stream.close();
190
```

```
191     int** newBase = new int* [numberOfDimentions];
192     newBase[0] = new int[sizeBaseOne];
193     newBase[1] = new int[sizeBaseOne];
194     newBase[0][0] = 0;
195     newBase[1][0] = 0;
196     for (int i = 1; i < sizeBaseOne; i++) {
197         newBase[0][i] = Streams[0][i] - Streams[0][i - 1];
198         newBase[1][i] = Streams[1][i] - Streams[1][i - 1];
199     }
200 
201     return newBase;
202 }
203 #endif // DTW_H_INCLUDED
```

APÊNDICE F – Arquivo fallDetection.h que contém a classe *fallDetection*

```
1  #pragma once
2  #include "../GestureDetection/DTW.h"
3
4  class fallDetection {
5      bestPath* pathDTW;
6      int count;
7      int countTimeGesture;
8  public:
9      bool possibleFall;
10     bool detectedGesture;
11     bool detectedFall;
12
13     fallDetection(bestPath*);
14     void update(bool);
15 };
16
17 fallDetection::fallDetection(bestPath* bestPathG) :
18     pathDTW(bestPathG),
19     detectedGesture(false),
20     detectedFall(false),
21     count(0),
22     countTimeGesture(0),
23     possibleFall(false)
24 {
25 }
26
27 void fallDetection::update(bool nextToTheGround) {
28     float DTWx = 0;
29     float DTWy = 0;
30     float DTWx1 = pathDTW->GetDTW(0, 0);
31     float DTWy1 = pathDTW->GetDTW(1,0);
32     float DTWx2 = pathDTW->GetDTW(0, 0);
33     float DTWy2 = pathDTW->GetDTW(1, 0);
34     if (DTWx1 + DTWy1 < DTWx2 + DTWy2) {
35         DTWx = DTWx1;
```

```
36          DTWy = DTWy1;
37      }
38      else {
39          DTWx = DTWx2;
40          DTWy = DTWy2;
41      }
42
43      if (DTWx < 3042 && DTWy < 3984) {
44          detectedGesture = true;
45          countTimeGesture = 0;
46      }
47      else if (countTimeGesture > 60 || countTimeGesture == 0)
48          detectedGesture = false;
49
50      if (detectedGesture)
51          countTimeGesture++;
52      else
53          countTimeGesture = 0;
54
55      double sumPosition = 0;
56      int sizeVector = pathDTW->sizeTwo;
57      int countValid = 0;
58      for (int i = sizeVector - 30; i < sizeVector; i++) {
59          if (pathDTW->arrayTwo[0][i] > 0 && pathDTW->arrayTwo[1][i] >
                0) {
60              sumPosition += static_cast<double>(((pathDTW->arrayTwo
                    [0][i] - pathDTW->arrayTwo[0][sizeVector])*
61                  (pathDTW->arrayTwo[0][i] - pathDTW->arrayTwo[0][
                        sizeVector]))/pathDTW->arrayTwo[0][sizeVector]);
62              sumPosition += static_cast<double>(((pathDTW->arrayTwo
                    [1][i] - pathDTW->arrayTwo[1][sizeVector])*
63                  (pathDTW->arrayTwo[1][i] - pathDTW->arrayTwo[1][
                        sizeVector]))/pathDTW->arrayTwo[1][sizeVector]);
64              countValid++;
65          }
66      }
67
68      if (countValid > 25 && !detectedFall) {
69          sumPosition /= 2* static_cast<double>(countValid);
70          if (sumPosition < 10 && nextToTheGround)
71              possibleFall = true;
72      }
```

```
73
74      if (possibleFall)
75          count++;
76      else
77          count = 0;
78
79      if ((count > 210 && detectedFall)||detectedGesture) {
80          possibleFall = false;
81          detectedFall = false;
82          count = 0;
83      }
84      else if (count > 150 && !detectedGesture && possibleFall &&
            nextToTheGround) {
85          detectedFall = true;
86      }
87  }
```

APÊNDICE G – Método *ProcessDepth* do programa da Microsoft editado para funcionar como o detector de quedas com interação por gestos

```
1  void CDepthBasics::ProcessDepth()
2  {
3      HRESULT hr;
4      NUI_IMAGE_FRAME imageFrame;
5
6      // Attempt to get the depth frame
7      hr = m_pNuiSensor->NuiImageStreamGetNextFrame(
           m_pDepthStreamHandle, 0, &imageFrame);
8      if (FAILED(hr))
9      {
10         return;
11     }
12
13     BOOL nearMode;
14     INuiFrameTexture* pTexture;
15
16     // Get the depth image pixel texture
17     hr = m_pNuiSensor->NuiImageFrameGetDepthImagePixelFrameTexture(
18         m_pDepthStreamHandle, &imageFrame, &nearMode, &pTexture);
19     if (FAILED(hr))
20     {
21         goto ReleaseFrame;
22     }
23
24     NUI_LOCKED_RECT LockedRect;
25
26     // Lock the frame data so the Kinect knows not to modify it while
           we're reading it
27     pTexture->LockRect(0, &LockedRect, NULL, 0);
28
29     // Make sure we've received valid data
```

```
30      if (LockedRect.Pitch != 0)
31      {
32          // Get the min and max reliable depth for the current frame
33          int minDepth = (nearMode ? NUI_IMAGE_DEPTH_MINIMUM_NEAR_MODE
                : NUI_IMAGE_DEPTH_MINIMUM) >> NUI_IMAGE_PLAYER_INDEX_SHIFT
                ;
34          int maxDepth = (nearMode ? NUI_IMAGE_DEPTH_MAXIMUM_NEAR_MODE
                : NUI_IMAGE_DEPTH_MAXIMUM) >> NUI_IMAGE_PLAYER_INDEX_SHIFT
                ;
35
36          BYTE* rgbrun = m_depthRGBX;
37          const NUI_DEPTH_IMAGE_PIXEL* pBufferRun = reinterpret_cast<
                const NUI_DEPTH_IMAGE_PIXEL*>(LockedRect.pBits);
38
39          // end pixel is start + width*height - 1
40          const NUI_DEPTH_IMAGE_PIXEL* pBufferEnd = pBufferRun + (
                cDepthWidth * cDepthHeight);
41
42          int widthCount = 0;
43          int heightCount = 0;
44          int* handPosition = NULL;
45          //Get the depth data
46          while (pBufferRun < pBufferEnd)
47          {
48              // discard the portion of the depth that contains only
                    the player index
49              USHORT depthShort = pBufferRun->depth;
50              float numDepth = static_cast<float>(depthShort - minDepth
                    );
51              depthImage[widthCount][heightCount] = static_cast<float>(
                    depthShort >= minDepth && depthShort <= maxDepth ?
                    numDepth / (maxDepth - minDepth) : -1);
52
53              widthCount++;
54              if (widthCount == cDepthWidth) {
55                  heightCount++;
56                  widthCount = 0;
57              }
58              // Increment our index into the Kinect's depth buffer
59              ++pBufferRun;
60          }
61
```

```
62          //Get hand position and if it is next to the ground
63          bool statusHand = posHand.update();
64          handPosition = posHand.getHandPosition();
65          //If position is not NULL then save the new position
66          if(handPosition!= NULL)
67              eventPerson.setNewPosition(handPosition);
68          //Update the detections variables
69          detectionEvents.update(statusHand);
70
71          for (int j = 0; j < cDepthHeight; j++)
72              for (int i = cDepthWidth - 1; i >= 0; i--)
73              {
74                  float depthToBeUsed = depthImage[i][j];
75
76                  USHORT depth_ajust = static_cast <SHORT>(255 * (
                        depthToBeUsed));
77                  BYTE intensity = static_cast<BYTE>(depthToBeUsed >= 0
                        && depthToBeUsed <= 1 ?
78                        depth_ajust % 256 : 0);
79                  if (handPosition != NULL && (i == handPosition[0] ||
    j == handPosition[1])) {
80                      *(rgbrun++) = 100;      //green
81                      *(rgbrun++) = 130;
82                      *(rgbrun++) = 0;
83                  }
84                  else if (detectionEvents.detectedFall && (i > 21 && i
                        < 31 && j < 239 && j >229)) {
85                      *(rgbrun++) = 0;        //red
86                      *(rgbrun++) = 0;
87                      *(rgbrun++) = 255;
88                  }
89                  else if (detectionEvents.detectedGesture && (i > 11
                        && i < 21 && j < 239 && j >229)) {
90                      *(rgbrun++) = 255;      //blue
91                      *(rgbrun++) = 0;
92                      *(rgbrun++) = 0;
93                  }
94                  else if (detectionEvents.possibleFall && (i > 1 && i
                        < 11 && j < 239 && j >229)) {
95                      *(rgbrun++) = 0;      //yellow
96                      *(rgbrun++) = 255;
97                      *(rgbrun++) = 255;
```

```
98                  }
99                  else {
100                     *(rgbrun++) = intensity;//gray scale
101                     *(rgbrun++) = intensity;
102                     *(rgbrun++) = intensity;
103                 }
104                 ++rgbrun;
105             }
106
107         m_pDrawDepth->Draw(m_depthRGBX, cDepthWidth * cDepthHeight *
                cBytesPerPixel);
108     }
109     // We're done with the texture so unlock it
110     pTexture->UnlockRect(0);
111
112     pTexture->Release();
113
114 ReleaseFrame:
115     // Release the frame
116     m_pNuiSensor->NuiImageStreamReleaseFrame(m_pDepthStreamHandle, &
            imageFrame);
117 }
```

Anexos

ANEXO A – Gráfico de interesse por sistema de detecção de quedas

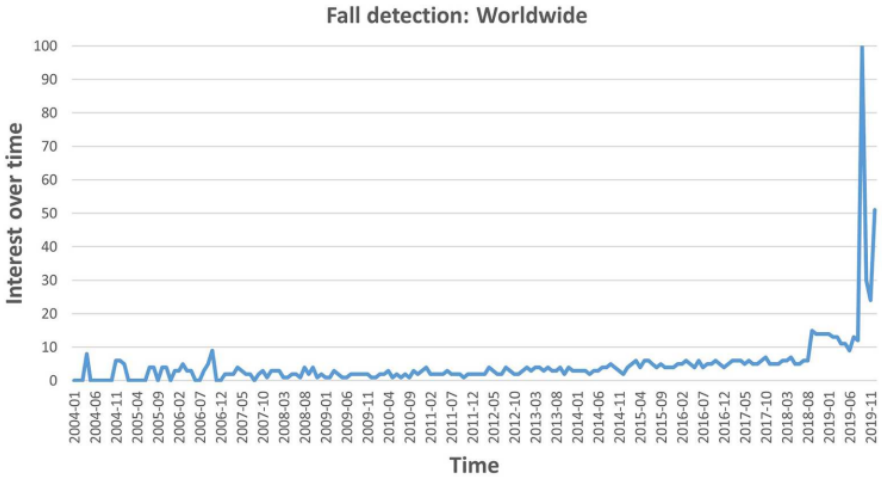


Figura 16 – Interesse por detecção de quedas ao longo do tempo, de janeiro de 2004 a dezembro de 2019, exporto pela Google Trends. Os valores de interesse estão normalizados ao pico de interesse (WANG; ELLUL; AZZOPARDI, 2020).

ANEXO B – Matriz de confusão da detecção de gestos que faz uso de uma SVM de múltipla classes

Gestures	Gestures							
	Clap	*Call*	*Greet*	*Wave*	*No*	*Yes*	*Clasp*	*Rest*
Clap	449	29	19	0	0	0	2	0
Call	35	516	4	3	2	4	22	17
Greet	0	28	464	8	8	3	0	1
Wave	2	12	0	302	0	0	0	0
No	0	85	0	46	195	9	1	1
Yes	7	98	0	7	14	186	25	6
Clasp	7	19	0	6	0	0	516	1
Rest	1	8	1	19	3	0	2	484

Figura 17 – Tabela de confusão (BISWAS; BASU, 2011).