Universidade de São Paulo
Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos
Bacharelado em Engenharia Mecatrônica

Gianlucci B. Minarelli e Renan R. do A. Gurgel

Biblioteca de funções de avaliação de pose e movimento para o Microsoft Kinect

São Paulo
2015

Gianlucci B. Minarelli e Renan R. do A. Gurgel

Biblioteca de funções de avaliação de pose e movimento para o Microsoft Kinect

Monografia apresentada no Departamento de Engenharia Mecatrônica e Sistemas Mecânicos da Escola Politécnica da Universidade de São Paulo para obtenção do título de Engenheiro. Área de Concentração: Engenharia Mecatrônica

Orientador: Fabrício Junqueira
Doutor em Engenharia Mecatrônica - USP

São Paulo
2015

Catalogação-na-publicação

Minarelli, Gianlucci
Biblioteca de funções de avaliação de pose e movimento para o Microsoft Kinect / G. Minarelli, R. Gurgel -- São Paulo, 2015.
128 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.Microsoft Kinect 2.Avaliação de Pose e Movimento Humanos I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos II.t. III.Gurgel, Renan

Termo de Originalidade

Este relatório é apresentado como requisito parcial para obtenção do título de Engenheiro na Escola Politécnica da Universidade de São Paulo. É o produto do nosso próprio trabalho, exceto onde indicado no texto. O relatório pode ser livremente copiado e distribuído desde que a fonte seja citada.

Resumo

A possibilidade de aplicação do *Kinect* para avaliação de movimento em áreas diversas como fisioterapia ou treinamento gera a necessidade de uma biblioteca de funções de comparação de movimentos e poses. O código aberto permite padronização e evita retrabalho. Um estudo sobre funcionamento e fontes de erro do *Kinect* foi realizado. Foram analisadas as diferentes opções para *SDK* e os tipos de dados. Usaram-se os diagramas da linguagem UML para auxiliar a apresentação das especificações do projeto. Uma biblioteca que realiza comparações de poses e movimentos genéricos foi implementada. Bibliotecas *wrappers* auxiliares foram implementadas para realizar a interface entre a biblioteca de avaliação e as duas versões da *Microsoft Kinect SDK* contempladas.

Palavras-chaves: Kinect. Biblioteca de Funções. Avaliação de Movimento Humano.

Abstract

The possibility of applying the *Kinect* for evaluating human movement in activities such as rehabilitation or training creates the need of a movement and pose comparison library. Open-source code allows standardization and avoids unnecessary work. The *Kinect* was analyzed along with is error sources. The different available *SDK*s and data structures were compared. UML diagrams were used to help the project and documentation of the library. A pose and movement comparison library was implemented. *Wrapper* libraries were implemented between the evaluation library and each version of the *Microsoft Kinect SDK*.

Keywords: Kinect. Library. Human Movement Evaluation.

Sumário

Lista de ilustrações

Lista de tabelas

1 Introdução

A necessidade de analisar o movimento humano e avaliá-lo seguindo algum critério existe em diferentes áreas, como fisioterapia, treinamentos industriais, esporte, dentre outras. A utilização do sensor MS Kinect para essa tarefa é interessante devido ao seu baixo custo e fácil integração ao sistema operacional Windows, amplamente disseminado.

O presente projeto propõe o desenvolvimento de uma biblioteca que, utilizando os dados obtidos pelo *Kinect*, será capaz de avaliar de forma quantitativa a precisão de um movimento captado pelo sensor em relação a um movimento de referência previamente registrado. A proposta é motivada pela possibilidade de desenvolvimento de projetos de engenharia voltados para a reabilitação ou treinamento e que se valham de ambientes de realidade virtual utilizando o *Kinect*.

Tal biblioteca possibilita não só uma padronização no modo como a avaliação de movimentos é realizada em diversos aplicativos que utilizam o *Kinect*, através do reaproveitamento do código desenvolvido, como também uma evolução contínua das técnicas de avaliação implementadas (projeto *Open Source*).

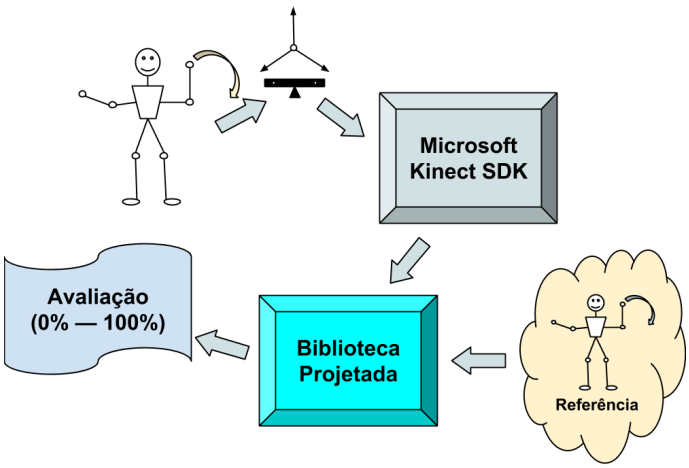

Figura 1 – Posição da biblioteca no fluxo (simplificado) de dados

A reutilização do código da biblioteca aqui desenvolvida em outros projetos:

- Evitará a repetição do esforço de projeto e implementação de algoritmos de avaliação durante o desenvolvimento de aplicativos para o Kinect, cujos desenvolvedores nem sempre estão aptos a desenvolver funções de avaliação de movimentos confiáveis.
- Garantirá a homogeneidade dos resultados obtidos por esses aplicativos, possibilitando, por exemplo, a comparação de resultados obtidos por diferentes softwares.
- Garantirá a qualidade de tais resultados.

A homogenização das técnicas de avaliação dos movimentos captados pelo sensor promoverá, como mencionado, o aperfeiçoamento contínuo dessas técnicas e a otimização, com o passar do tempo, dos resultados obtidos, uma vez que a biblioteca desenvolvida estará sujeita a constantes aprimoramentos por meio de atualizações realizadas pela própria comunidade de desenvolvedores que se valham de tais resultados.

Apesar do sistema descrito apresentar uma grande abrangência em diferentes áreas (no desenvolvimento de jogos, aplicativos voltados para o setor de serviços, na indústria, etc), o projeto aqui proposto irá se limitar à análise de movimento de pessoas, focando-se assim, de uma forma geral, em aplicações nas áreas médicas (reabilitação), de entretenimento (jogos) e voltadas ao treinamento de pessoal especializado.

A validade da avaliação de movimentos, usando o Kinect, para fins médicos, como análise de movimentos típicos do mal de Parkinson (GALNA, 2014), de crises emocionais em crianças (YU, 2011) ou de exercícios de reabilitação (BO, 2011), mostra o potencial do projeto na área médica. As funções implementadas poderiam ser utilizadas, por exemplo, no desenvolvimento de jogos voltados à reabilitação e com uma dificuldade (intensidade do exercício) auto-regulável, de acordo com o *feedback* obtido com o auxílio da biblioteca.

1.1 Objetivos

O objetivo desse trabalho é projetar e implementar uma biblioteca de funções de avaliação de movimentos capturados pelo *Kinect*, para ser utilizada durante o desenvolvimento de aplicativos que se valham de tais métodos.

Deseja-se um código que apresente eficiência de tempo de processamento e manutenibilidade, para que a publicação da biblioteca em código aberto possibilite o maior número de aplicações das funções de avaliação de movimento, evitando retrabalho e fomentando o desenvolvimento dessa classe de algoritmos. Para atingir esses objetivos, as seguintes metas foram estabelecidas:

- Familiarização com a linguagem C#.
- Familiarização com as classes e métodos oferecidos pela SDK utilizada.
- Familiarização com as ferramentas de desenvolvimento de aplicativos oferecidas pelo software *Visual Studio*.
- Projeto e implementação da biblioteca.
- Implementação de um algoritmo para o teste de precisão da biblioteca desenvolvida.
- Desenvolvimento de um aplicativo demonstrativo das funcionalidades da biblioteca implementada.

- Implementação de funções de processamento de imagens (poses) e vídeos (movimentos) no aplicativo desenvolvido, de modo a simular funções passíveis de serem implementadas em softwares que possam utilizar a biblioteca desenvolvida.
- Implementação e integração entre a biblioteca e o aplicativo projetados.
- Realização de testes demonstrativos da integração entre as funções do aplicativo desenvolvido (tratamento de poses e movimentos) e funções de avaliação de poses e movimentos da biblioteca.

2 Revisão Bibliográfica

2.1 Funcionamento do Kinect e fontes de erro

Primeiramente, para prever qual precisão poderá ser atingida usando-se o *Kinect*, é necessário compreender as limitações dos sensores do *Kinect* e, portanto, seu funcionamento. O *Kinect V1* dispõe de duas câmeras: uma *RGB* (do inglês, vermelho, verde e azul) e uma "câmera de profundidade" (baseada em infravermelho), além de um microfone (MICROSOFT, 2015a). O *Kinect V2* tem a câmera de cores HD 1080p e microfones aprimorados (MICROSOFT, 2015c).

Como mostrado na Figura 2, o campo de visão do *Kinect V1* para *Windows* tem o valor de seus parâmetros dependentes da configuração escolhida (modos pré-programados): no modo "próximo" de configuração, tem-se $r1 = 0,4m$, $r2 = 0,8m$, $R1 = 2,5m$ e $R2 = 3m$; no modo "padrão", tem-se $r1 = 0,8m$, $r2 = 1,2m$, $R1 = 3,5m$ e $R2 = 4m$. Na zona de leitura ótima, representada na Figura 2, tem-se melhor precisão e reatividade (MICROSOFT, 2015a).

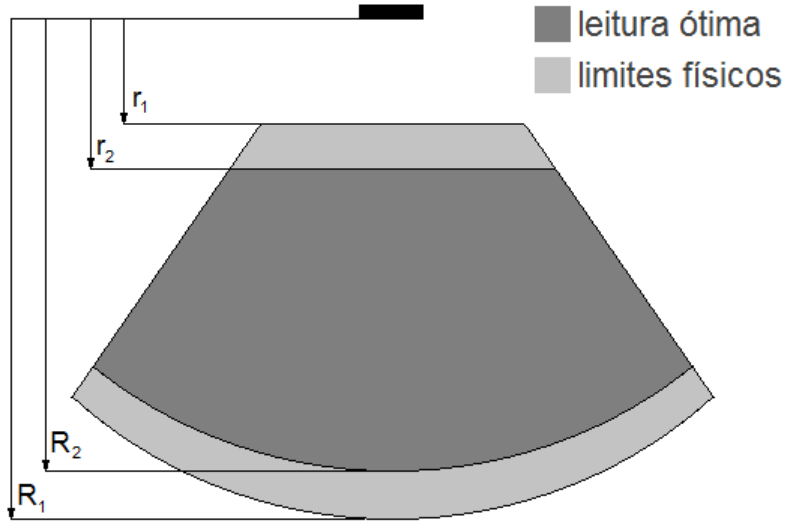

Figura 2 – Aspecto geométrico do campo de visão do Kinect para Windows; o valor dos parâmetros depende da configuração escolhida

Os limites de ângulo de visão do Kinect V1 para Windows (ver Figura 3) são $\alpha = 57,5^o$ (horizontal) e $\beta = 43,5^o$ (vertical, podendo ser rotacionado de 27^o para cima ou para baixo graças ao suporte articulado).

O *Kinect V2* apresenta um campo de visão mais amplo (tanto na horizontal quanto na vertical) (MICROSOFT, 2015c).

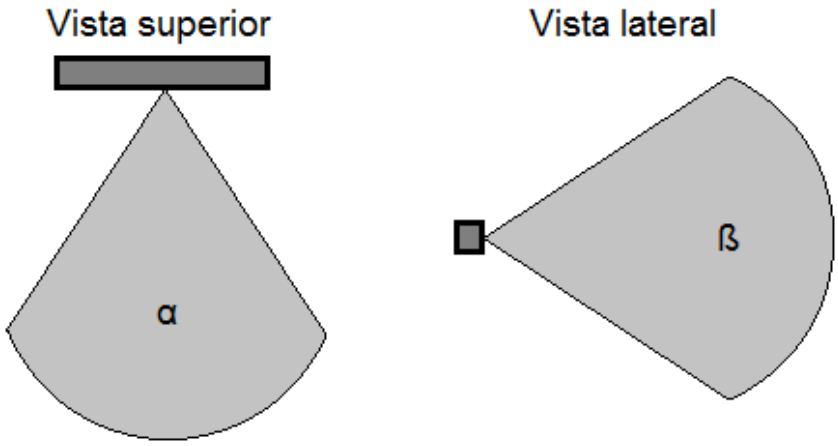

Figura 3 – Ângulos horizontal e vertical do campo de visão do Kinect para Windows

A captura de imagens em cores e em infravermelho se faz de forma simultânea, com uma taxa de aquisição de aproximadamente 30 fps (do inglês, quadros por segundo) para ambos os sensores (KHOSHELHAM, 2011) (MICROSOFT, 2015c).

Para fazer a medição de profundidade, a fonte do laser infravermelho emite um único feixe, que é então difratado para projetar uma rede de pontos no volume do campo de visão do Kinect, para finalmente ser captada pela câmera de infravermelho. Esses pontos são então comparados a uma projeção num plano de referência (informações provenientes da calibração) e, através de uma triangulação, obtém-se os valores de profundidade. Há, neste processo, fontes de incerteza: as distorções provocadas pelas lentes e o desalinhamento com os eixos da câmera RGB (KHOSHELHAM, 2011). A acurácia de profundidade do sensor *V2* é três vezes maior que a do *V1* (MICROSOFT, 2015c).

Uma outra fonte de incertezas é a refletância dos objetos analisados pelo Kinect. O usuário deverá evitar roupas pretas ou com detalhes ou acessórios refletivos, pois isso poderá interferir com a leitura do infravermelho (MICROSOFT, 2015a). A iluminação do ambiente também influencia as medições, mas o sensor *V2* é bem mais robusto nesse aspecto do que o sensor *V1* (MICROSOFT, 2015c).

O fluxo das informações de profundidade lidas se faz usando inteiros de 11 bits (um inteiro desses para cada ponto da rede e para cada instante de tempo em que houve uma leitura), sendo que um deles indica apenas se aquele ponto foi encontrado na imagem analisada ou não (ver Figura 4). Ou seja, a disparidade constatada é discretizada em 1024 níveis, incorrendo numa perda de informação (KHOSHELHAM, 2011).

Uma alternativa que melhora sensivelmente a qualidade das informações obtidas é o uso de múltiplos Kinects em vez de apenas um (desde que não haja sobreposição dos feixes, o que gera interferência entre as malhas lidas) (TONG, 2012) (RAKPRAYOON, 2011). Isso potencialmente evitaria que o Kinect interpretasse objetos distintos como

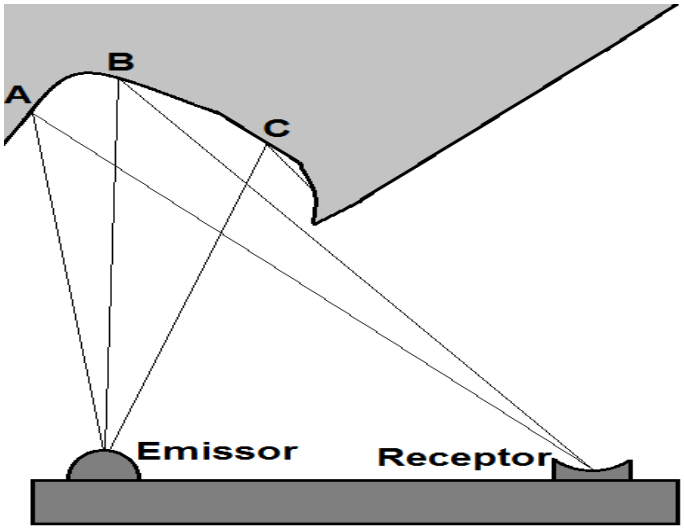

Figura 4 – Dependendo do formato dos objetos analisados, alguns pontos (como o indicado pela letra C) podem não ser recebidos pela câmera de infravermelho

constituindo um corpo único, erro que de fato ocorre quando eles apresentam pontos próximos e com profundidades semelhantes (BO, 2011).

3 Levantamento das Alternativas

Nesse capítulo serão descritas as alternativas tecnológicas (tipos de pacotes de desenvolvimento) e estratégicas (tipos de análise de posicionamento) levantadas durante a fase inicial de projeto e em relação as quais a biblioteca e aplicativos desenvolvidos nesse trabalho são embasados.

3.1 SDK (*Software Development Kit*)

A primeira decisão foi em relação ao tipo de SDK que seria utilizado. Na Tabela 1 são apresentados os tipos considerados e alguns critérios utilizados na escolha da solução final.

Tabela 1 – Comparação entre os SDKs considerados para o projeto (OPENKINECT, 2014) (MICROSOFT, 2015b) (OPENNI, 2015)

Caracterísicas	**Software Development Kit (SDK)**		
	Kinect for Windows V1.8 e *V2.0*	*OpenKinect*	*OpenNI*
Comunidade	Ativa	Inativa (*V1*) e Ativa (*V2*)	Ativa
Fonte	Oficial	Não oficial	Não oficial
Integração com o Kinect	Total	Total	Parcial
Código aberto	Não	Sim	Sim
Rastreamento de esqueleto	Sim	Não	Sim
Documentação	Ampla e organizada	Escassa	Ampla mas pouco organizada

A escolha pelas Microsoft Kinect SDK V1.8 e V2.0 foi feita baseada não só no fato de que essas são as alternativas mais populares entre os desenvolvedores que utilizam o Kinect mas também pois, sendo produtos desenvolvidos e distribuídos pela própria Microsoft, essa é a solução que apresenta o maior número de manuais e outras publicações de cunho didático – focadas na descrição do seu funcionamento, solução de problemas típicos e sugestões de aplicação das funções disponíveis pela ferramenta.

A opção OpenKinect apresenta um nível baixo demais de tratamento de dados para os objetivos deste trabalho, não apresentando a funcionalidade de rastreamento de esqueleto (OPENKINECT, 2014), fundamental para o que se deseja.

3.2 Tipos de dados de posição

A segunda decisão tomada diz respeito aos tipos de dados que seriam utilizados no processo de comparação de poses e movimentos. As alternativas de solução e alguns critérios de seleção considerados estão descritos na Tabela 2.

Tabela 2 – Comparação entre os tipos de dados de posição para o projeto

Caracterísicas	**Tipos de dados (coordenadas)**	
	Ângulos das juntas	*Coordenadas cartesianas das juntas*
Obtenção	Indireta	Direta
Invariância a biotipo	Sim	Não
Invariância à posição relativa ao sensor	Sim	Não
Quantidade de variáveis	Menor ou igual a 45	60

A solução escolhida foi a representação da posição das juntas que compõem o esqueleto captado pelo sensor pelos seus ângulos característicos. Essa solução, comumente utilizada em projetos de modelagem biomecânica do corpo humano (BOGERT, 2013), não só diminui a quantidade de informação necessária para a realização da comparação, como também elimina problemas associados a diferenças fisiológicas entre o modelo que gerou a referência armazenada pelo sistema e o usuário final, cuja imagem é captada e avaliada em relação a essa referência.

4 Projeto

4.1 Requisitos de projeto

Diagramas UML foram utilizados para mostrar, de maneira clara, as funções que o sistema projetado deverá ser capaz de realizar (requisitos funcionais) bem como o modo como as mesmas serão realizadas.

Alguns requisitos de projeto importantes, no que diz respeito ao modo como o sistema deve ser capaz de garantir a sua flexibilidade a diferentes tipos de comparações e critérios de seleção, são:

- A biblioteca deve suportar o uso dos dois modelos do sensor Kinect atualmente disponíveis no mercado (i.e. Kinect V1 e Kinect V2).
- As funções de comparação utilizadas devem ser projetadas de modo a possibilitar a avaliação de esqueletos com N juntas, onde N não é necessariamente igual ao número de juntas definidas nas classes *Skeleton* dos Kinect SDKs v1.8 (Kinect V1) ou 2.0 (Kinect V2). Ou seja, o algoritmo deve ser capaz de avaliar, de uma forma genérica, dois conjuntos de pontos, sem considerar as restrições impostas por um modelo de esqueleto específico.
- A biblioteca deve suportar a utilização de parâmetros genéricos de avaliação associados às poses comparadas, de modo a não limitar os possíveis algoritmos de avaliação suportados pelo sistema. Ou seja, uma pose deve ser definida como um conjunto de parâmetros genéricos (coordenadas cartesianas ou ângulos característicos das juntas que compõem a pose analisada, por exemplo) e as possíveis comparações realizadas devem, então, ser adaptadas ao conjunto de parâmetros genéricos utilizado.

4.2 Diagrama de Casos de Uso

A seguir, encontram-se os diagramas UML que especificam o projeto da biblioteca. Para elaboração do diagrama de casos de uso, considerou-se uma aplicação possível para a biblioteca (reabilitação fisioterápica), então evidenciaram-se os casos de uso que dizem respeito à biblioteca de fato. Os casos de uso que não estão no sistema *Biblioteca de funções de avaliação de movimento* não serão contemplados neste projeto.

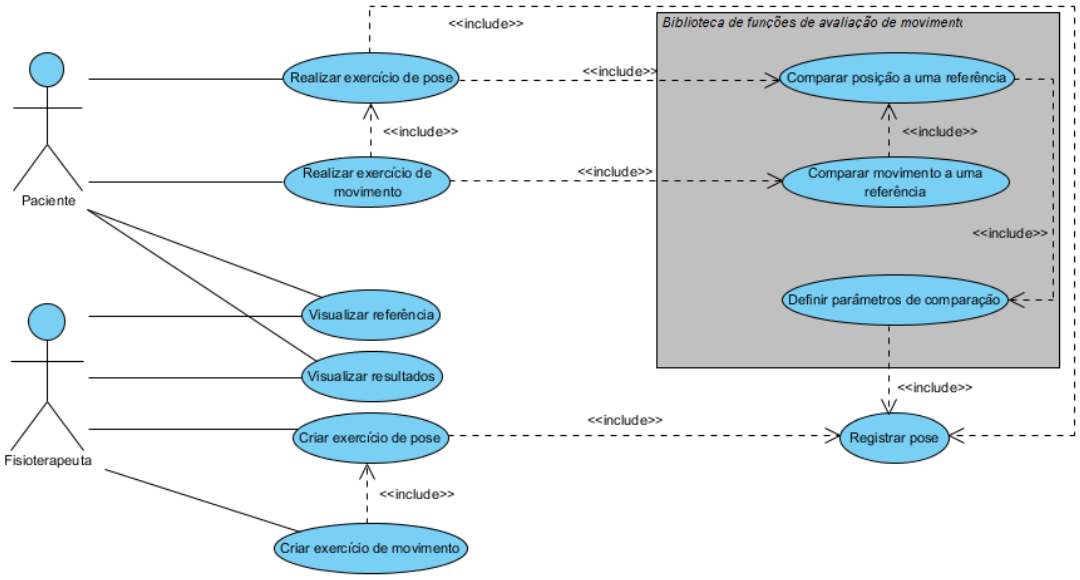

Figura 5 – Diagrama de casos de uso

- Comparar Posição a uma Referência (CDU1): uma pose de entrada (a ser comparada) é avaliada em relação a uma pose de referência.
- Comparar Movimento a uma Referência (CDU2): as poses de um determinado movimento são comparadas a um conjunto de poses de referência.
- Definir Parâmetros de Comparação (CDU3): os parâmetros que serão utilizados nas comparações de poses e movimentos (conjunto de poses) são definidos de acordo com dados fornecidos pelo usuário (*inputs*) e as características das poses analisadas.

4.3 Diagramas de Atividade

4.3.1 Comparar Posição a uma Referência (CDU01)

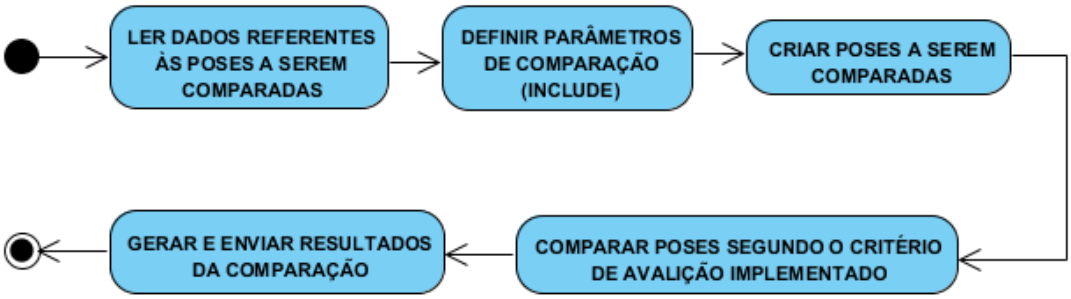

Figura 6 – Diagrama de atividades de Comparar Posição a uma Referência

Esse diagrama mostra as relações entre as atividades que compõem o caso de uso *Comparar Posição a uma Referência*. O objetivo desse caso de uso é a análise dos erros

de posicionamento entre uma pose (conjunto de parâmetros genéricos representativos de um *Skeleton*) de entrada (a ser comparada) e uma pose de referência, de acordo com um determinado critério de avaliação. Informações sobre o conjunto de parâmetros de comparação e o critério de avaliação utilizados podem ser vistas com mais detalhes nas seções 4.5 e 4.6.1, respectivamente.

4.3.2 Comparar Movimento a uma Referência (CDU02)

Figura 7 – Diagrama de atividades de Comparar Movimento a uma Referência

Esse diagrama mostra as relações entre as atividades que compõem o caso de uso *Comparar Movimento a uma Referência*. O objetivo desse caso de uso é a análise dos erros de posicionamento entre um movimento (conjunto de poses) de entrada (a ser comparado) e um movimento de referência, de acordo com um determinado critério de avaliação. Informações sobre o critério de avaliação utilizado podem ser vistas com mais detalhes na seção 4.6.2

4.3.3 Definir Parâmetros de Comparação (CDU03)

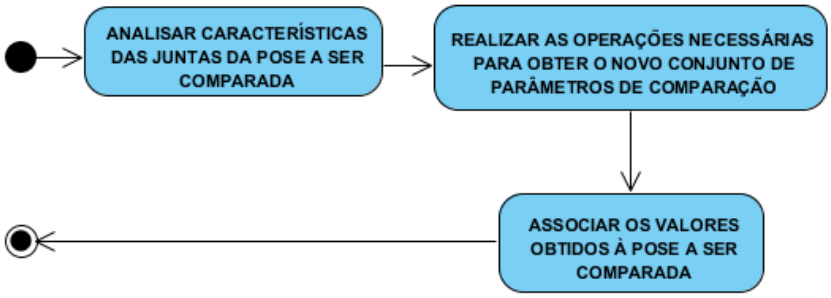

Figura 8 – Diagrama de atividades de Definir Parâmetros de Comparação

Esse diagrama mostra as relações entre as atividades que compõem o caso de uso *Definir Parâmetros de Comparação*. O objetivo desse caso de uso é gerar o conjunto de

parâmetros de comparação que serão associados às poses avaliadas. Informações sobre as características desses parâmetros podem ser vistas com mais detalhes na seção 4.5.

4.4 Diagrama de Classes

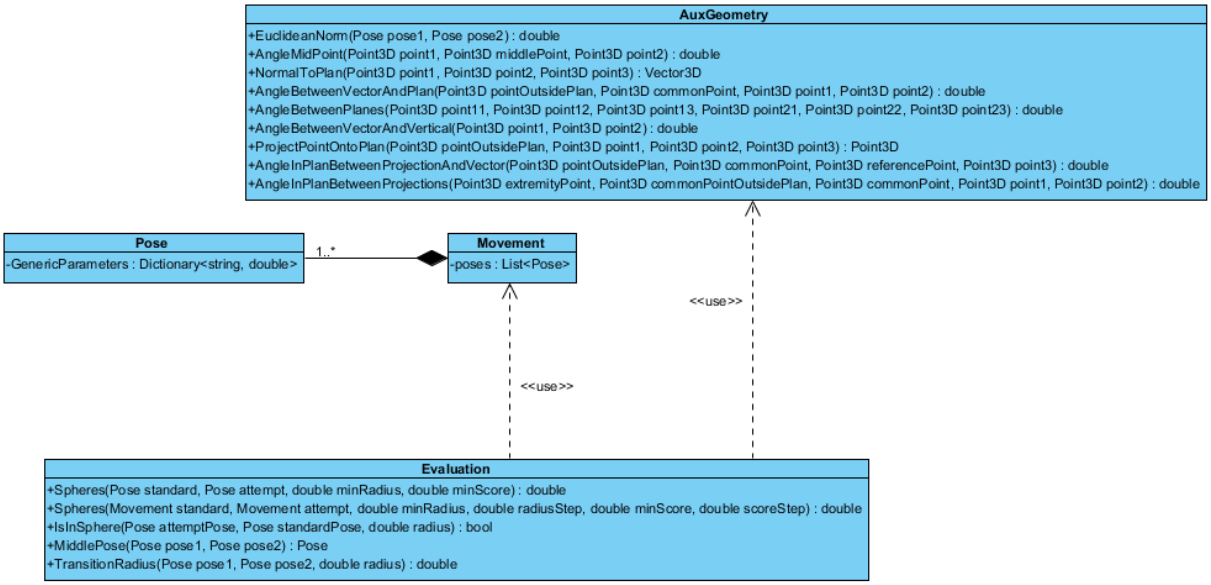

Figura 9 – Diagrama de classes

As classes foram construídas tendo-se sempre em mente o objetivo de produzir uma biblioteca abrangente e genérica, que pudesse avaliar poses e movimentos para diferentes definições de bases de coordenadas e métricas.

Pose

- GenericParameters: uma estrutura *Dictionary<string, double>* de *C#* que guarda os nomes e valores que cada coordenada genérica escolhida.

Movement

- Poses: uma sequência temporalmente ordenada de Poses.

Evaluation: a classe estática que contém todos os métodos do cerne da biblioteca de avaliação.

- Spheres (Pose): método de avaliação de poses. Recebe duas poses (uma *referência* e uma *tentativa*) e as compara com base em dois outros parâmetros: raio mínimo e nota mínima. Uma descrição mais detalhada do algoritmo pode ser encontrada na seção 4.6.1;

- Spheres (Movement): método de avaliação de movimentos. Recebe dois movimentos (um *referência* e um *tentativa*) e os compara com base em quatro outros parâmetros: raio mínimo, passo do raio, nota mínima e passo da nota. Uma descrição mais detalhada do algoritmo pode ser encontrada na seção 4.6.2;
- IsInSphere: um método auxiliar, utilizado pelo método *Spheres*, que verifica se uma dada pose se encontra na esfera definida por outra pose e um raio;
- MiddlePose: um método auxiliar, utilizado pelo método *Spheres*, que retorna a pose média entre duas poses dadas;
- TransitionRadius: um método auxiliar, utilizado pelo método *Spheres*, que calcula o raio de uma esfera de transição entre duas esferas. Mais detalhes sobre o algoritmo completo podem ser encontrados na seção 4.6.2.

AuxGeometry: uma biblioteca auxiliar para realização dos cálculos de geometria analítica necessários. Antes de se realizarem os cálculos, ocorre sempre a verificação de singularidades geométricas.

- EuclideanNorm: calcula a norma euclideana do vetor definido entre duas poses;
- AngleMidPoint: calcula o ângulo definido entre dois pontos de extremidade e um intermediário, recebendo como entrada os três pontos;
- NormalToPlan: calcula o vetor normal a um plano dado, recebendo como entrada apenas os pontos que definem as entidades;
- AngleBetweenVectorAndPlan: calcula o ângulo entre um vetor e um plano, recebendo como entrada apenas os pontos que definem as entidades;
- AngleBetweenPlanes: calcula o ângulo entre planos, recebendo como entrada apenas os pontos que definem as entidades;
- AngleBetweenVectorAndVertical: calcula o ângulo entre um vetor e a vertical (suposta como a terceira coordenada), recebendo como entrada apenas os pontos que definem as entidades;
- ProjectPointOntoPlan: calcula o ponto que representa a projeção ortogonal de um ponto em um plano, recebendo como entrada apenas os pontos que definem as entidades;
- AngleInPlanBetweenProjectionAndVector: calcula o ângulo entre a projeção de um vetor num plano e um outro vetor do plano, recebendo como entrada apenas os pontos que definem as entidades;
- AngleInPlanBetweenProjections: calcula o ângulo entre as projeções de dois vetores num plano, recebendo como entrada apenas os pontos que definem as entidades.

4.5 Definição das Coordenadas

As estruturas de dados que representam um corpo humano rastreado pelo sensor são diferentes entre as versões 1.8 e 2.0 da *Microsoft Kinect SDK*. Criaram-se, portanto, duas bibliotecas *wrappers* auxiliares (ver apêndices A.1 e A.2, uma para cada versão de *SDK* trabalhada. Essas bibliotecas servem de interface entre a biblioteca de avaliação de movimento (apêndice A.3) e as diferentes *SDK*s, além de modularizarem as referências necessárias (evitando múltiplas referências a uma mesma *SDK* em que a única distinção é a versão).

A partir das estruturas de dados recebidas pelas bibliotecas *wrappers*, podem se definir conjuntos de coordenadas que sejam representativas para o tipo de movimento ou pose que se deseja analisar. Foram definidos dois conjuntos, um para cada versão de *SDK*, que usam ângulos calculados a partir das posições das juntas, para representar os ângulos das articulações do corpo humano. Esse tipo de modelo é utilizado na fisioterapia (BOGERT, 2013).

4.5.1 Ângulos para a *SDK* 1.8

A base de coordenadas é escolhida considerando-se a hipótese heurística de que uma rotação em torno do eixo vertical não tenha importância.

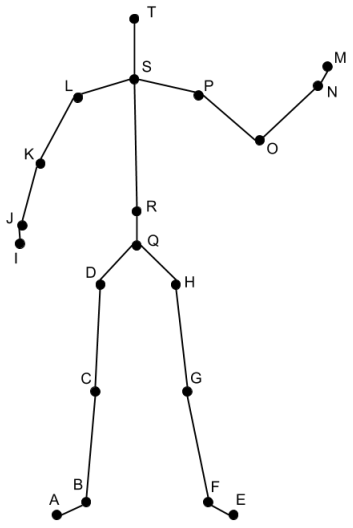

Figura 10 – Representação gráfica da estrutura de dados *skeleton*, da *SDK* 1.8

Na Figura 10, os nomes dos pontos como constam na *SDK* são:

A: *FootRight* (**E**: *FootLeft*);

B: *AnkleRight* (**F**: *AnkleLeft*);

C: *KneeRight* (**G**: *KneeLeft*);

D: *HipRight* (**H**: *HipLeft*);

I: *HandRight* (**M**: *HandLeft*);

J: *WristRight* (**N**: *WristLeft*);

K: *ElbowRight* (**O**: *ElbowLeft*);

L: *ShoulderRight* (**P**: *ShoulderLeft*);

Q: *HipCenter*;

R: *Spine*;

S: *ShoulderCenter*;

T: *Head*;

Definição das coordenadas, usando os pontos indicados na Figura 10:

– Joelhos:

$$\theta_{JD} = \angle BCD,\ \theta_{JE} = \angle FGH$$

– Cotovelos:

$$\theta_{CD} = \angle JKL,\ \theta_{CE} = \angle NOP$$

– Mãos:

$$\theta_{M1D} = \angle IJK,\ \theta_{M1E} = \angle MNO$$

$$\theta_{M2D} = \angle(\overrightarrow{IJ},\ plano(J,K,L)),\ \theta_{M2E} = \angle(\overrightarrow{MN},\ plano(N,O,P))$$

– Pés:

$$\theta_{P1D} = \angle ABC,\ \theta_{M1E} = \angle EFG$$

$$\theta_{P2D} = \angle(\overrightarrow{AB},\ plano(B,C,D)),\ \theta_{P2E} = \angle(\overrightarrow{EF},\ plano(F,G,H))$$

– Pernas:

$$\theta_{Pr1D} = \angle(\overrightarrow{CD},\ plano(D,H,Q)),\ \theta_{Pr1E} = \angle(\overrightarrow{GH},\ plano(D,H,Q))$$

$$\theta_{Pr2D} = \angle(proj_{pLano(D,H,Q)}\overrightarrow{CD},\ \overrightarrow{DQ}),\ \theta_{Pr2E} = \angle(proj_{plano(D,H,Q)}\overrightarrow{GH},\ \overrightarrow{HQ})$$

– Braços:

$$\theta_{B1D} = \angle KLS,\ \theta_{B1E} = \angle OPS$$

$$\theta_{B2D} = \angle(\overrightarrow{KL},\ plano(L,P,S)),\ \theta_{B2E} = \angle(\overrightarrow{PS},\ plano(L,P,S))$$

$$\theta_{B3D} = \angle(\overrightarrow{JK},\ plano(K,L,S)),\ \theta_{B3E} = \angle(\overrightarrow{NO},\ plano(O,P,S))$$

– Torso:

$$\theta_{T1} = \angle QRS$$

$$\theta_{T2} = \angle(plano(D,H,Q)),\ plano(L,P,R))$$

$$\theta_{T3} = \angle(proj_{plano(D,H,Q)}\overrightarrow{QR},\ proj_{plano(D,H,Q)}\overrightarrow{RS})$$

– Cabeça:

$$\theta_{Cb1} = \angle(proj_{plano(L,P,S)}\overrightarrow{ST},\ \overrightarrow{RS})$$

$$\theta_{Cb2} = \angle RST$$

– Inclinação:

$$\theta_{Cb2} = \angle(\overrightarrow{RS},\ \vec{g})$$

4.5.2 Ângulos para a *SDK* 2.0

Novamente, a base de coordenadas é escolhida considerando-se a hipótese heurística de que uma rotação em torno do eixo vertical não tenha importância.

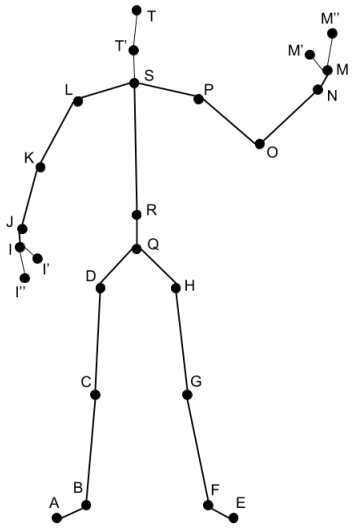

Figura 11 – Representação gráfica da estrutura de dados *skeleton*, da *SDK* 2.0

Na Figura 11, os nomes dos pontos como constam na *SDK* são:

A: *FootRight* (**E**: *FootLeft*);

B: *AnkleRight* (**F**: *AnkleLeft*);

C: *KneeRight* (**G**: *KneeLeft*);

D: *HipRight* (**H**: *HipLeft*);

I: *HandRight* (**M**: *HandLeft*);

I': *HandTipRight* (**M'**: *HandTipLeft*);

I": *ThumbRight* (**M"**: *ThumbLeft*);

J: *WristRight* (**N**: *WristLeft*);

K: *ElbowRight* (**O**: *ElbowLeft*);

L: *ShoulderRight* (**P**: *ShoulderLeft*);

Q: *SpineBase*;

R: *SpineMid*;

S: *SpineShoulder*;

T': *Neck*;

T: *Head*;

Definição das coordenadas, usando os pontos indicados na Figura 11:

– Joelhos:

$$\theta_{JD} = \angle BCD,\ \theta_{JE} = \angle FGH$$

– Cotovelos:

$$\theta_{CD} = \angle JKL,\ \theta_{CE} = \angle NOP$$

– Mãos:

$$\theta_{M1D} = \angle IJK,\ \theta_{M1E} = \angle MNO$$

$$\theta_{M2D} = \angle(\overrightarrow{IJ},\ plano(J,K,L)),\ \theta_{M2E} = \angle(\overrightarrow{MN},\ plano(N,O,P))$$

– Pés:

$$\theta_{P1D} = \angle ABC,\ \theta_{M1E} = \angle EFG$$

$$\theta_{P2D} = \angle(\overrightarrow{AB},\ plano(B,C,D)),\ \theta_{P2E} = \angle(\overrightarrow{EF},\ plano(F,G,H))$$

– Pernas:

$$\theta_{Pr1D} = \angle(\overrightarrow{CD},\ plano(D,H,Q)),\ \theta_{Pr1E} = \angle(\overrightarrow{GH},\ plano(D,H,Q))$$

$$\theta_{Pr2D} = \angle(proj_{pLano(D,H,Q)}\overrightarrow{CD},\ \overrightarrow{DQ}),\ \theta_{Pr2E} = \angle(proj_{plano(D,H,Q)}\overrightarrow{GH},\ \overrightarrow{HQ})$$

– Braços:

$$\theta_{B1D} = \angle KLS,\ \theta_{B1E} = \angle OPS$$

$$\theta_{B2D} = \angle(\overrightarrow{KL},\ plano(L,P,S)),\ \theta_{B2E} = \angle(\overrightarrow{PS},\ plano(L,P,S))$$

$$\theta_{B3D} = \angle(\overrightarrow{JK},\ plano(K,L,S)),\ \theta_{B3E} = \angle(\overrightarrow{NO},\ plano(O,P,S))$$

– Torso:

$$\theta_{T1} = \angle QRS$$

$$\theta_{T2} = \angle(plano(D,H,Q)),\ plano(L,P,R))$$

$$\theta_{T3} = \angle(proj_{plano(D,H,Q)}\overrightarrow{QR},\ proj_{plano(D,H,Q)}\overrightarrow{RS})$$

– Pescoço:

$$\theta_{Pc} = \angle RST'$$

– Cabeça:

$$\theta_{Cb1} = \angle(proj_{plano(L,P,S)}\overrightarrow{ST},\ \overrightarrow{RS})$$

$$\theta_{Cb2} = \angle T'ST$$

– Inclinação:

$$\theta_{Cb2} = \angle(\overrightarrow{RS},\ \vec{g})$$

4.6 Algoritmos de comparação

Definem-se alguns termos que representam as entidades utilizadas no algoritmo:

- **Pose**: um conjunto de coordenadas genéricas (seus nomes e seus valores) que define uma posição de um corpo humano (ou parte dele) para um instante de tempo. Pode ser representado como um vetor $\vec{p}$.
- **Movimento**: um conjunto de poses de mesmas coordenadas genéricas, de um mesmo corpo humano (ou parte dele), referentes a instantes de tempo sequenciais: $M = \{\vec{p}_i\}$
- **Referência**: um objeto (pose ou movimento) considerado como padrão (base de uma comparação).
- **Tentativa**: um objeto (pose ou movimento) que tenta aproximar uma referência.

Foram implementados um algoritmo para comparação de movimentos e um para comparação de poses, ambos nomeados *spheres*, aproveitando-se da sobrecarga de métodos possível em *C#*. Observe-se que as implementações são agnósticas à escolha da base de coordenadas (e não somente aplicável àquelas definidas na seção 4.5), produzindo assim algoritmos genéricos que podem ser reutilizados para outras convenções ou modelos.

4.6.1 Poses

O método *spheres* para poses recebe como parâmetros duas poses: uma referência e uma tentativa, e dois valores de ponto flutuante: o raio mínimo e a nota mínima. Como resposta, ele retorna um valor de ponto flutuante representando a avaliação.

Funcionamento: usa-se a norma euclideana. Caso a distância entre a tentativa e a referência seja inferior ao raio mínimo (ou seja, se a tentativa está contida na *esfera mínima* cujo centro é a referência), o método retorna o valor 1, indicando sucesso total. Se não, avalia-se a razão entre o raio mínimo e a distância entre poses; se inferior à nota mínima, o método retorna -1 (indicando falha); se for superior a nota mínima, o método retorna o valor dessa razão como nota da avaliação.

4.6.2 Movimentos

Movimentos podem ser construidos a partir de uma lista de poses ou lidos de um arquivo *.txt* que respeitem o seguinte padrão: cada parâmetro deve estar numa linha diferente, com seus valores temporalmente sequenciais separados por espaços em branco.

O método *spheres* para movimentos recebe como parâmetros dois movimentos: um referência e um tentativa, e quatro valores de ponto flutuante: o raio mínimo, o passo do raio, a nota mínima e o passo da nota. Como resposta, ele retorna um valor de ponto flutuante representando a avaliação.

Funcionamento: são definidas regiões de avaliação $(Z_\phi)_i$ em torno de cada pose $\vec{p_i}$ para cada nível de precisão ϕ estabelecido de maneira discreta, com passo da nota $step_\phi$, da seguinte maneira:

$(Z_\phi)_i : \{\vec{p} \mid \|\vec{p} - \vec{p_i}\| \leq r_\phi\}$, sendo $r_\phi = r_{mín} + j * step_r$, em que $r_{mín}$ é o raio mínimo, j é o inteiro correspondente ao número do passo e $step_r$ é o passo do raio.

São também definidas as regiões de transição $(T_\phi)_{i,i+1}$ entre duas poses consecutivas:

$(T_\phi)_{i,i+1} : \{\vec{p} \mid \|\vec{p} - \vec{p_k}\| \leq R_\phi\}$, com $\vec{p}_k = \frac{1}{2} * (\vec{p_i} + \vec{p}_{i+1})$ e $R_\phi = \sqrt{r_\phi^2 + (\frac{1}{2} * \|\vec{p}_{i+1} - \vec{p_i}\|)^2}$.

A Figura 12 mostra a ideia análoga para duas dimensões, para um dado ϕ, com as circunferências de linha cheia representando $(Z_\phi)_0$ e $(Z_\phi)_1$, e a circunferência de linha pontilhada representando $(T_\phi)_{0,1}$. Há também a representação gráfica de R_ϕ e r_ϕ.

Avalia-se, então, um movimento através do algoritmo ilustrado na figura 13.

Sendo ϕ_i a precisão de cada seção do movimento e n o número de poses contidas no movimento referência, a nota final (o velor de retorno do método) vale:

$$nota = \frac{1}{n-1} \sum_{i=1}^{n-1} \phi_i$$

Em caso de alguma falha, o método retorna imediatamente com valor -1.

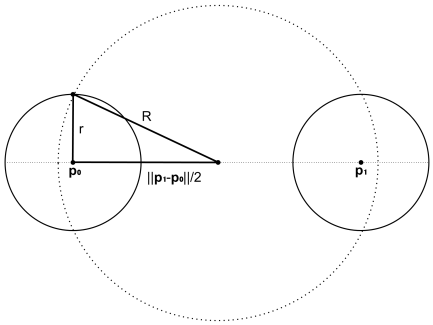

Figura 12 – Representação gráfica bidimensional das regiões de avaliação e de transição

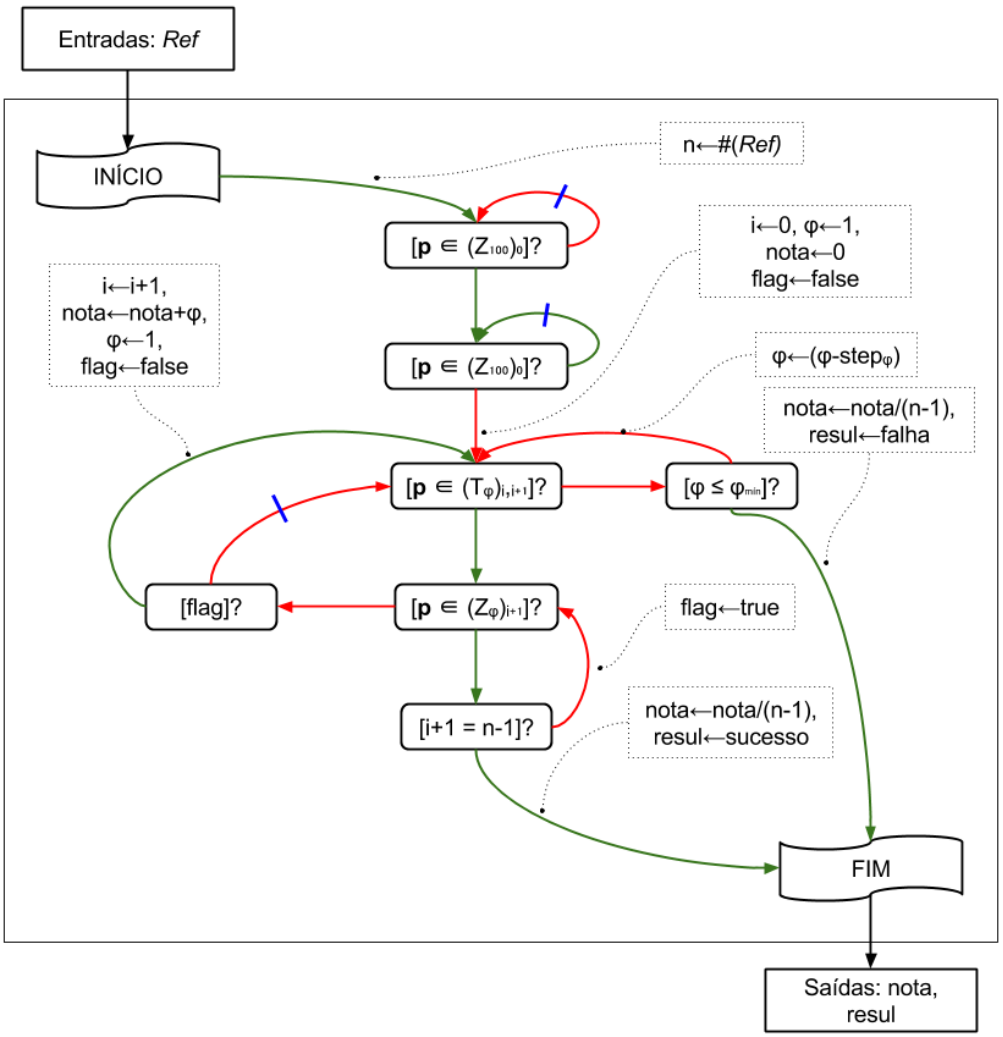

Figura 13 – As flechas verdes são para valores verdadeiros das condições testadas e as vermelhas, para falsos. Os traços azuis indicam leitura da próxima pose do movimento tentativa

4.7 Aplicativo

No decorrer do trabalho um aplicativo para o Kinect V1 foi desenvolvido a fim de demonstrar as funcionalidades da biblioteca projetada e o potencial da mesma como uma importante ferramenta no desenvolvimento de aplicativos que se valham da comparação de poses e movimentos aqui proposta.

O software foi desenvolvido no *Visual Studio Community 2015*, em C# e utilizando o *framework* .NET 4.6 oferecido pela Microsoft.

O código fonte do aplicativo em sua versão final pode ser visto no Apêndice C.

4.7.1 Interface Gráfica e Comandos

Nessa seção serão descritos todos os elementos que compõem a interface gráfica do software desenvolvido, detalhando as suas funções e interdependências dentro do código.

Durante a descrição que será realizada, poses ou movimentos de referência, ou seja, poses ou movimentos que seriam registrados previamente à utilização do aplicativo pelo usuário final do mesmo (registrados por fisioterapeutas para a utilização do software na área de reabilitação, por exemplo), serão referenciados como elementos *de referência*.

Do mesmo modo, poses ou movimentos realizados pelo usuário final do software (um paciente, continuando o exemplo da utilização do aplicativo na área de reabilitação) serão referenciados como elementos *do usuário*.

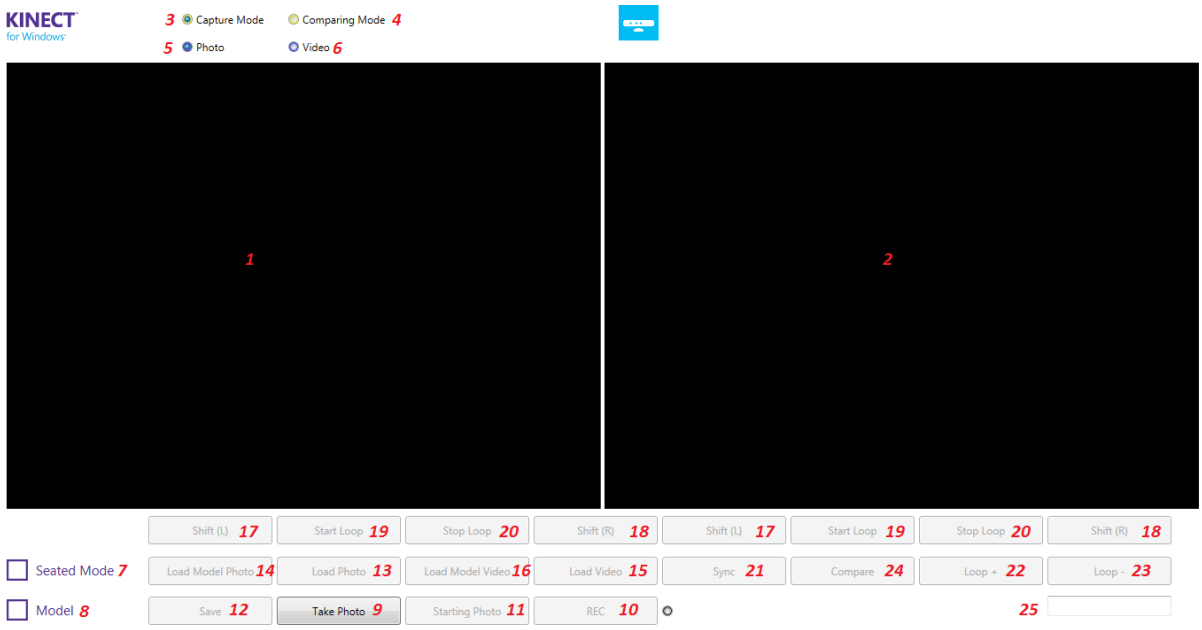

Figura 14 – Interface gráfica do software desenvolvido.

1. Tela esquerda: Nessa tela são plotados poses e movimentos de referência quando

o aplicativo se encontra em modo de comparação (vide itens 3 e 4). Em modo de captura, essa tela plota continuamente os movimentos capturados em tempo real pelo sensor (*feedback* visual).

2. Tela Direita: Nessa tela são plotados poses e movimentos do usuário quando o aplicativo se encontra em modo de comparação (vide itens 3 e 4). Em modo de captura, essa tela plota o movimento que será salvo pelo sistema quando o botão REC (item 10) é pressionado, ou mostra a pose que será salva pelo sistema quando o botão *Take Photo* (item 9) é pressionado, dependendo do que o usuário deseja capturar - movimento ou pose (itens 6 e 5, respectivamente).

3. *Capture Mode*: Quando o aplicativo se encontra nesse modo de operação, botões e demais elementos gráficos que permitem que o usuário realize funções de captura de imagens (poses) ou vídeos (movimentos) são acionados (i.e. podem ser pressionados).

4. *Comparing Mode*: Quando o aplicativo se encontra nesse modo de operação, botões e demais elementos gráficos que permitem que o usuário realize funções de comparação de imagens (poses) ou vídeos (movimentos) são acionados.

5. *Photo*: Quando o aplicativo se encontra nesse modo de operação, botões e demais elementos gráficos que permitem que o usuário realize funções de captura ou comparação de imagens (poses) são acionados.

6. *Video*: Quando o aplicativo se encontra nesse modo de operação, botões e demais elementos gráficos que permitem que o usuário realize funções de captura ou comparação de vídeos (movimentos) são acionados.

7. *Seated Mode*: Quando essa caixa de seleção é marcada, juntas que compõem a parte inferior do esqueleto captado (quadril, joelhos, tornozelos e pés) são ignoradas na construção da figura do esqueleto que será plotado na tela.

8. *Model*: Quando essa caixa de seleção é marcada, qualquer pose ou movimento salvos durante a utilização do software (vide item 12) serão marcados pelo sistema como sendo poses ou movimentos de referência.

9. *Take Photo*: No instante em que esse botão é pressionado, a imagem (pose) mostrada na Tela Esquerda (item 1) é guardada em uma variável global no código (memória volátil).

10. REC: No instante em que esse botão é pressionado, as imagens (poses) que compõem o movimento capturado pelo sensor começam a ser armazenadas pelo software (novamente utilizando variáveis globais, ou seja, a memória ainda é volátil). Pressionando o botão uma segunda vez finaliza o processo de captura do vídeo (movimento).

11. *Starting Photo*: Após o botão REC (item 10) ser pressionado pela segunda vez, é possível selecionar uma nova pose inicial para o movimento capturado utilizando os botões de *Shift* (itens 17 e 18) associados à Tela Direita. Ao pressionar esse botão, a pose mostrada na Tela Direita será a nova pose inicial do movimento capturado.

12. *Save*: Ao pressionar esse botão, poses e movimentos previamente armazenados em variáveis globais (memória volátil) dentro do código (vide itens 9 e 10) são exportados na forma de arquivos de texto (memória não-volátil).

13. *Load Photo*: Ao pressionar esse botão, uma pose será carregada na Tela Direita (item 2).

14. *Load Model Photo*: Ao pressionar esse botão, uma pose será carregada na Tela Esquerda (item 1).

15. *Load Video*: Ao pressionar esse botão, um movimento será carregada na Tela Direita (item 2). Inicialmente a tela mostrará apenas uma pose (pose inicial do movimento), porém botões para a manipulação do movimento carregado estarão ativos (vide itens 17, 18, 19, 20, 21, 22 e 23).

16. *Load Model Video*: Ao pressionar esse botão, um movimento será carregada na Tela Esquerda (item 1). Inicialmente a tela mostrará apenas uma pose (pose inicial do movimento), porém botões para a manipulação do movimento carregado estarão ativos (vide itens 17, 18, 19, 20, 21, 22 e 23).

17. *Shift (L)*: Ao pressionar esse botão, a pose imediatamente anterior à atualmente plotada na tela alvo (esquerda ou direita), e que compõem o movimento carregado pelo sistema (vide itens 15 e 16), é exibida. O botão mais a esquerda está associado à Tela Esquerda (item 1), enquanto que o botão mais a direita está associado à Tela Direita (item 2).

18. *Shift (R)*: Ao pressionar esse botão, a pose imediatamente posterior à atualmente plotada na tela alvo (esquerda ou direita), e que compõem o movimento carregado pelo sistema (vide itens 15 e 16), é exibida. O botão mais a esquerda está associado à Tela Esquerda (item 1), enquanto que o botão mais a direita está associado à Tela Direita (2).

19. *Start Loop*: Ao pressionar esse botão, as poses que compõem o movimento carregado pelo sistema (vide itens 15 e 16) são plotadas na tela alvo (esquerda ou direita) de maneira sequencial crescente e cíclica (i.e. em *loop*). O botão mais a esquerda está associado à Tela Esquerda (item 1), enquanto que o botão mais a direita está associado à Tela Direita (item 2).

20. *Stop Loop*: Ao pressionar esse botão, o plot sequencial crescente e cíclico das poses que compõem o movimento carregado é interrompido na tela alvo. O botão mais a esquerda está associado à Tela Esquerda (item 1), enquanto que o botão mais a direita está associado à Tela Direita (item 2).

21. *Sync*: Ao pressionar esse botão, os movimentos plotados em ambas as telas voltam à sua posição inicial (primeira pose do movimento), sincronizando o índice das poses atualmente plotadas.

22. *Loop +* : Ao pressionar esse botão, o intervalo de tempo entre o plot em *loop* de duas poses consecutivas, em ambos os movimentos, é decrementado. Ou seja, a velocidade com que as imagens são exibidas em sequência aumenta (velocidade do movimento plotado aumenta).

23. *Loop -* : Ao pressionar esse botão, o intervalo de tempo entre o plot em *loop* de duas poses consecutivas, em ambos os movimentos, é incrementado. Ou seja, a velocidade com que as imagens são exibidas em sequência diminui (velocidade do movimento plotado diminui).

24. *Compare*: Ao pressionar esse botão, é realizada a comparação entre as poses ou movimentos de interesse, previamente carregados e exibidos em ambas as telas (1 e 2).

25. *Textbox*: Caixa de texto onde o resultado da comparação (vide item 24) é exibido.

5 Testes e Resultados

Para realizar os testes de todo o código desenvolvido, foram desenvolvidos três pequenos programas auxiliares em *Python*, para que fosse possível avaliar diferentes etapas do fluxo da biblioteca de maneira modular.

Um deles, *testgenerator.py* (ver apêndice B.1) gerou sinais senoidais e imitações com ruido branco para o teste do algoritmo *Spheres*. Os resultados podem ser vistos nas figuras 15, 16 e 17.

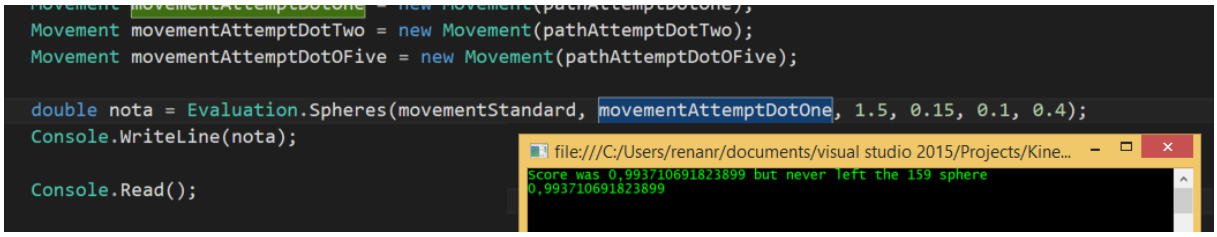

Figura 15 – Sucesso (excetuando-se a última esfera) na avaliação de dois movimentos, o segundo gerado a partir do primeiro com adição de ruido branco de desvio-padrão 0,1.

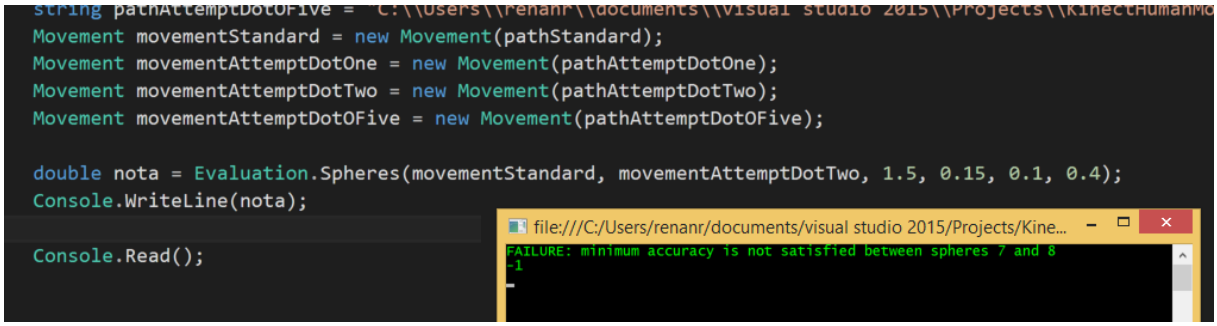

Figura 16 – Falha na avaliação de dois movimentos, mantendo-se os parâmetros mas com o segundo movimento gerado a partir do primeiro com adição de ruido branco de desvio-padrão 0,2.

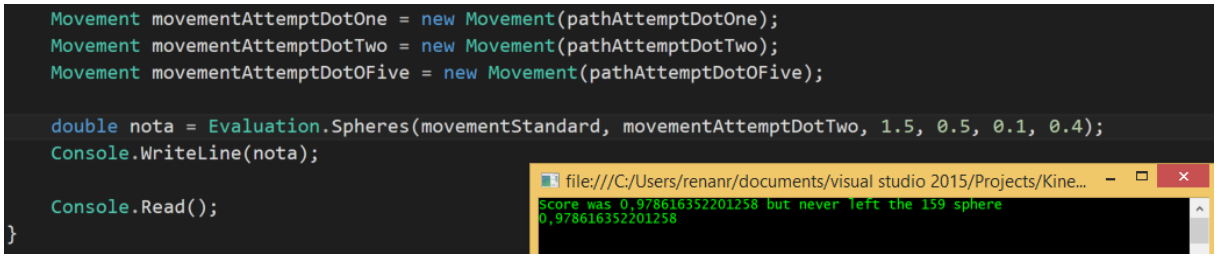

Figura 17 – Sucesso (excetuando-se a última esfera) na avaliação de dois movimentos, o segundo gerado a partir do primeiro com adição de ruido branco de desvio-padrão 0,2, mas com um passo de raio maior e outros parâmetros mantidos.

Os outros dois, *jointsPlotter.py* (apêndice B.2) e *genParmsPlotter.py* (apêndice B.3) receberam arquivos *.txt* contendo, no primeiro caso, as coordenadas cartesianas das juntas lidas diretamente pelo *Kinect V2* e, no segundo caso, os valores dos parâmetros genéricos (ângulos das articulações) para dois movimentos realizados: um considerado "padrão"e o outro sendo uma tentativa de imitar o primeiro (realizados pela mesma pessoa).

O *jointsPlotter.py* confirmou que a captura de dados pelo *Kinect V2* e sua *SDK* funcionaram de maneira satisfatória. O movimento realizado para teste é ilustrado pela figura 18. Os gráficos gerados das coordenadas cartesianas das juntas lidas estão nas figuras 19, 20, 21 e 22. Observou-se que as regiões iniciais e finais apresentam comportamento instável, mas isso deve-se ao tempo que a pessoa levou para entrar e sair do campo de visão do sensor. Exluindo-se esse detalhe, o comportamento dos dados observados reflete satisfatoriamente o movimento realizado.

Figura 18 – Dois momentos do movimento realizado para testar a captura de dados pelo sensor e seu registro na estrutura de dados presente na *SDK*.

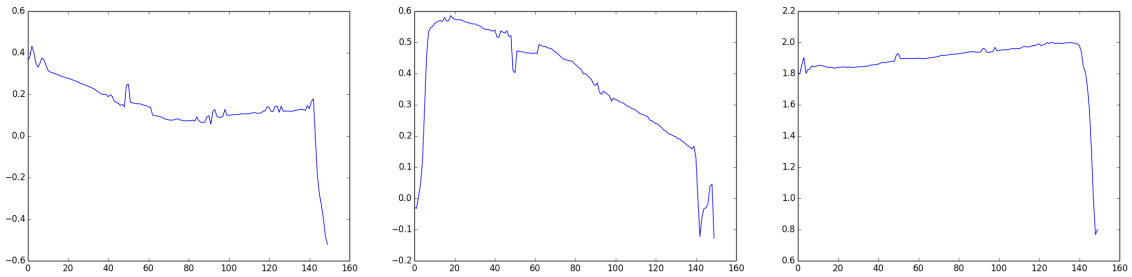

Figura 19 – Coordenadas cartesianas (da esquerda para a direita, X, Y e Z) da junta *mão direita*, com eixos verticais em metros e horizontais em índice da sequência.

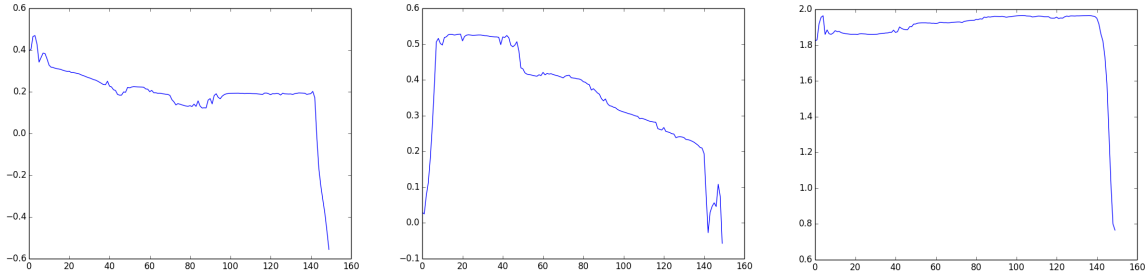

Figura 20 – Coordenadas cartesianas (da esquerda para a direita, X, Y e Z) da junta *pulso direito*, com eixos verticais em metros e horizontais em índice da sequência.

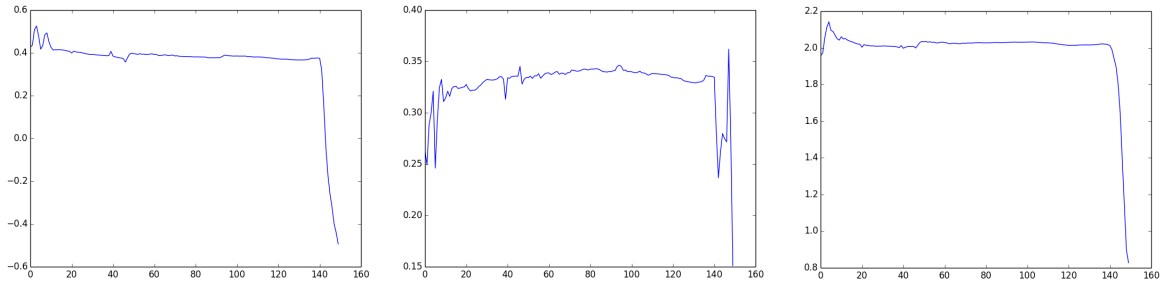

Figura 21 – Coordenadas cartesianas (da esquerda para a direita, X, Y e Z) da junta *cotovelo direito*, com eixos verticais em metros e horizontais em índice da sequência.

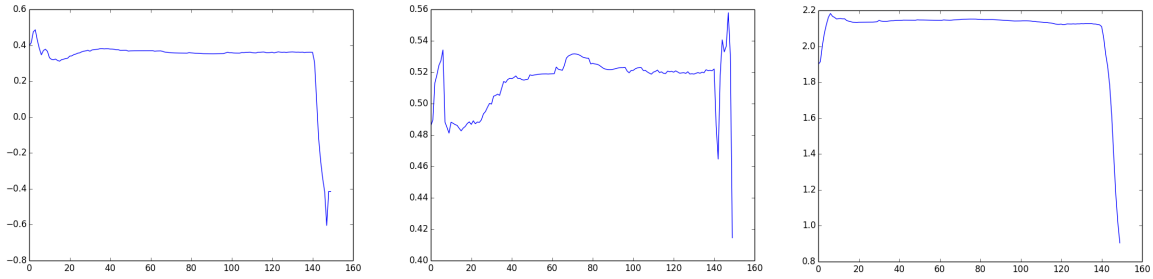

Figura 22 – Coordenadas cartesianas (da esquerda para a direita, X, Y e Z) da junta *ombro direito*, com eixos verticais em metros e horizontais em índice da sequência.

O *genParmsPlotter.py* confirmou que o cálculo dos ângulos seguiu o esperado. Para esse teste, uma pessoa realizou um movimento (numa tentativa de variar apenas um dos ângulos da base, para simplificar a análise) que foi tomado como padrão e, em seguida, tentou repeti-lo. O movimento em questão é ilustrado pela figura 23.

Os dados iniciais e finais foram desprezados,No movimento realizado, tentou-se manter a variação angular do cotovelo constante no tempo, o que foi bem representado pelos resultados, como pode ser visto na figura 24. Além disso, os valores também são condizentes: a figura mostra uma variação de aproximadamente 70 graus (do instante inicial ao final), o que corresponde aos valores calculados (variação de aproximadamente 1,3 radianos). Para validação, ilustrou-se também a comparação dos valores de ângulo para o cotovelo esquerdo, que ficou parado; de fato, como pode ser visto na figura 25, as variações foram inferiores a 1% da variação calculada para o cotovelo direito.

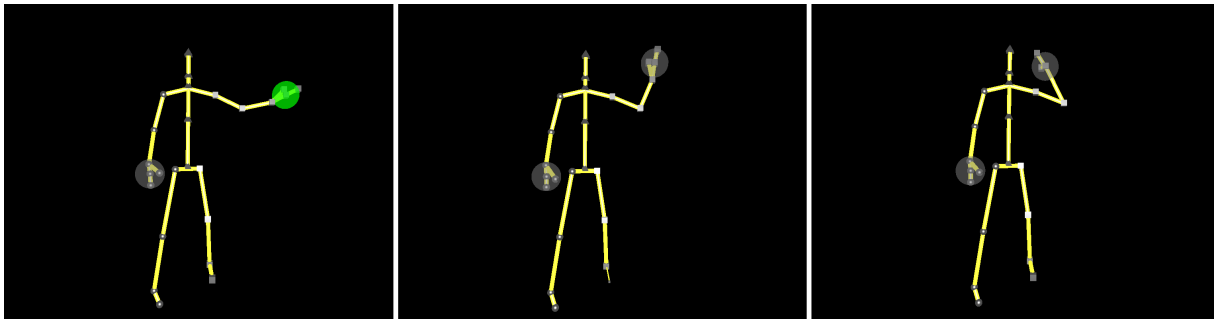

Figura 23 – Três momentos do movimento realizado para testar o cálculo de ângulos como parâmetros genéricos pela biblioteca.

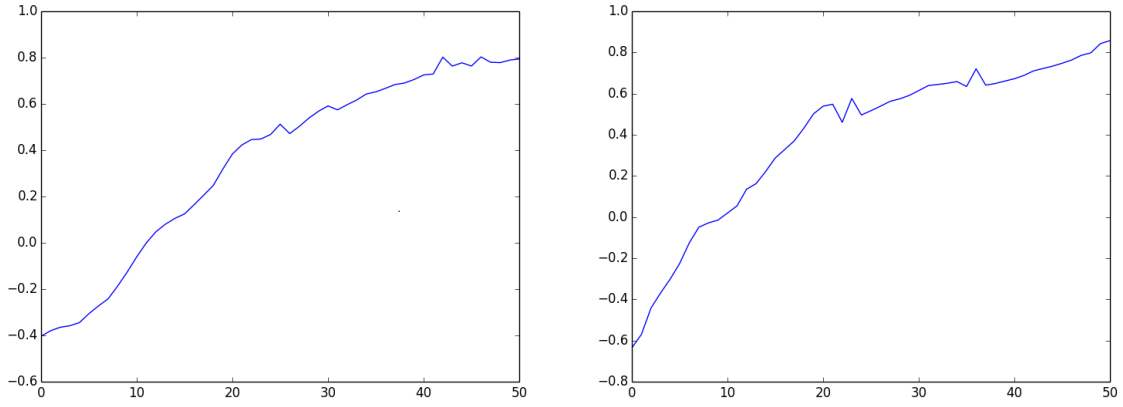

Figura 24 – Ângulo do cotovelo, como definido na seção 4.5.2. À esquerda, valores para o movimento considerado "padrão"; à direita, para a tentativa de reprodução. Eixos verticais em radianos e horizontais em índice da sequência.

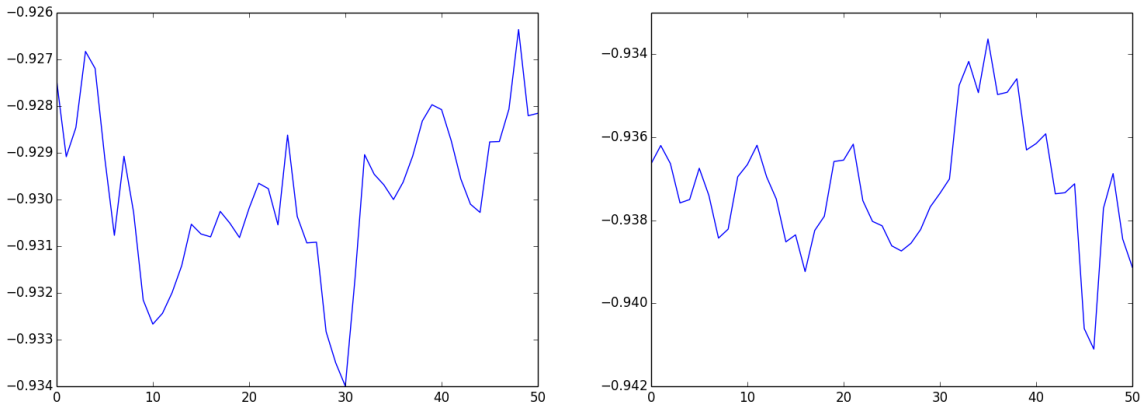

Figura 25 – Ângulo do cotovelo, como definido na seção 4.5.2. À esquerda, valores para o movimento considerado "padrão"; à direita, para a tentativa de reprodução. Eixos verticais em radianos e horizontais em índice da sequência.

6 Conclusões

A biblioteca desenvolvida, como exemplificado pela implementação das funções de comparação de poses e movimentos descritas (vide seção 4.6), possui grande flexibilidade em relação aos tipos de parâmetros de comparação que podem ser associados às poses a serem avaliadas e utilizados em qualquer método de avaliação implementado.

No caso das funções de avaliação aqui projetadas, os parâmetros de comparação das poses a serem avaliadas foram os ângulos característicos das juntas dos esqueletos que representam tais poses, porém, é importante notar, qualquer outro parâmetro (coordenadas cartesianas das juntas, por exemplo) poderia ser utilizado em conjunto com a função de comparação implementada (ou qualquer outra função de comparação), gerando resultados equivalentes.

Isso acontece devido ao modo como uma pose é definida dentro da biblioteca (vide seção 4.4), possuindo um conjunto de parâmetros genéricos (que podem ser ângulos, coordenadas absolutas, pesos, ou quaisquer outros valores atribuídos pelo usuário da biblioteca) como o único descritivo das características dessa pose. Desse modo, uma pose não mais é associada a um número fixo de valores e nem a um tipo fixo associado a esses valores (60 coordenadas cartesianas, por exemplo, como é definido no construtor de um *Skeleton* na Kinect SDK v1.8).

A biblioteca também é flexível em relação ao tipo de sensor utilizado, uma vez que suporta a utilização de ambas as versões do sensor Kinect (Kinect V1 e Kinect V2), desde que a SDK utilizada esteja atualizada (v1.8 para o V1 e v2.0 para o V2).

Os projeto foi realizado utilizando um aplicativo desenvolvido exclusivamente para o Kinect V1, com a SDK v1.8, porém, como a definição de uma pose dentro da biblioteca não está necessariamente associada a nenhuma propriedade previamente estabelecida ao instanciar um objeto *Skeleton* dentro do aplicativo (como citado nos parágrafos anteriores), dados recebidos de aplicativos projetados para qualquer versão do sensor podem ser processados pelos métodos de avaliação de uma maneira única, garantindo resultados consistentes e independentes do sensor utilizado.

Por fim, é importante notar que a possibilidade de crescimento da biblioteca desenvolvida é praticamente ilimitada, uma vez que a implementação de novos métodos de avaliação depende somente da criatividade do desenvolvedor e da sua capacidade em transcrever expressões e métodos matemáticos dentro da biblioteca. A utilização contínua dessa biblioteca como uma ferramenta de avaliação de poses e movimentos promoveria uma evolução natural da mesma pela comunidade de usuários, gerando não só um aumento no número de métodos de avaliação disponíveis (como discutido), como também uma

melhoria significativa (e constante) na eficiência computacional, robustez e precisão do sistema como um todo.

Referências

BELFIORE, P. P.; FáVERO, L. P. L. Técnicas estatísticas multivariadas para análise do comportamento de grupos supermercadistas brasileiros. *IX Seminários em Administração FEA-USP*, 2006. Nenhuma citação no texto.

BO, A. e. a. Joint angle estimation in rehabilitation with inertial sensors and its integration with kinect. *Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, p. 3479–3483, 2011. Citado 2 vezes nas páginas 11 e 15.

BOGERT, A. J. e. a. A real-time system for biomechanical analysis of human movement and muscle function. *PubMed Central*, 2013. Citado 2 vezes nas páginas 17 e 23.

GALNA, B. e. a. Accuracy of the microsoft kinect sensor for measuring movement in people with parkinson's disease. *Gait and Posture*, v. 39, p. 1062–1068, 2014. Citado na página 11.

HAGEGE, R.; FRANCOS, J. M. Parametric estimation of multi-dimensional affine transformations: an exact linear solution. *IEEE International Conference on Acoustics, Speech, and Signal Processing*, v. 2, p. ii/861–ii/864, 2005. Nenhuma citação no texto.

KHOSHELHAM, K. Accuracy analysis of kinect depth data. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, p. 133–138, 2011. Citado na página 14.

MICROSOFT. *Kinect for Windows Human Interface Guidelines v1.8.0*. 2015. Disponível em: <https://msdn.microsoft.com/en-us/library/jj663791.aspx>. Acesso em: 20.5.2015. Citado 2 vezes nas páginas 13 e 14.

MICROSOFT. *Kinect for Windows SDK*. 2015. Disponível em: <https://msdn.microsoft.com/enus/library/hh855347.aspx>. Acesso em: 23.5.2015. Citado 2 vezes nas páginas 9 e 16.

MICROSOFT. *Programming Kinect for Windows v2 Jump Start*. 2015. Disponível em: <https://www.microsoftvirtualacademy.com/en-US/training-courses/programming-kinect-for-windows-v2-jump-start-9088?l=Ju7xHKf4_6604984382>. Acesso em: 18.08.2015. Citado 2 vezes nas páginas 13 e 14.

OPENKINECT. *FAQ*. 2014. Wiki do OpenKinect. Disponível em: <http://openkinect.org/wiki/FAQ>. Acesso em: 23.5.2015. Citado 3 vezes nas páginas 9, 16 e 17.

OPENNI. *Reference Guide*. 2015. Disponível em: <http://openni.ru/reference-guide/index.html?t=index.html>. Acesso em: 25.5.2015. Citado 2 vezes nas páginas 9 e 16.

RAKPRAYOON, P. e. a. Kinect-based obstacle detection for manipulator. *International Symposium on System Integration (SII)*, p. 68–73, 2011. Citado na página 14.

TONG, J. e. a. Scanning 3d full human bodies using kinects. *IEEE Transactions on Visualization and Computer Graphics*, p. 643–650, 2012. Citado na página 14.

YU, X. e. a. Children tantrum behaviour analysis based on kinect sensor. *Third Chinese Conference on Intelligent Visual Surveillance (IVS)*, p. 49–52, 2011. Citado na página 11.

Apêndices

APÊNDICE A – Código Fonte: Algoritmo de Comparação Sequencial

Todo o código está disponível no repositório github.com/renanr/KinectHumanMovementEvaluat

A.1 Wrapper SDK 1.8

```
using System.Collections.Generic;
using K1_8 = Microsoft.Kinect;
using System.Windows.Media.Media3D;
using System;

namespace KHMESDK1_8
{
    public class Skeleton
    {
        public Dictionary<string, Point3D> Joints;

        // Empty Constructor
        public Skeleton() { }

        // Constructor from a Kinect SDK 1.8 Skeleton
        public Skeleton(K1_8.Skeleton skeleton)
        {
            this.Joints = new Dictionary<string, Point3D>();
            foreach(K1_8.Joint joint in skeleton.Joints)
            {
                if (joint.TrackingState != K1_8.JointTrackingState.NotTra
                {
                    this.Joints.Add(joint.JointType.ToString(), KinV1Join
                }
                else
                {
                    Console.WriteLine("WARNING: joint " + joint.JointType
                }
            }
```

```
            /*-----------------------------------------------------------
            Key names for joints are:
            "FootRight", "AnkleRight", "KneeRight", "HipRight"
            "FootLeft", "AnkleLeft", "KneeLeft", "HipLeft"
            "HandRight", "WristRight", "ElbowRight", "ShoulderRight"
            "HandLeft", "WristLeft", "ElbowLeft", "ShoulderLeft"
            "HipCenter", "Spine", "ShoulderCenter", "Head"
            -----------------------------------------------------------*/
        }

        // Method to construct a Point3D from a Kinect V1 Joint
        public Point3D KinV1JointToPoint3D(K1_8.Joint joint)
        {
            Point3D point = new Point3D(joint.Position.X, joint.Position.
            return point;
        }
    }
}
```

A.2 Wrapper SDK 2.0

```
using System.Collections.Generic;
using K2_0 = Microsoft.Kinect;
using System.Windows.Media.Media3D;
using System;

namespace KHMESDK2_0
{
    public class Skeleton
    {
        public Dictionary<string, Point3D> Joints;

        // Empty Constructor
        public Skeleton() { }

        // Constructor from a Kinect SDK 2.0 Skeleton
        public Skeleton(K2_0.Body body)
        {
            this.Joints = new Dictionary<string, Point3D>();
            foreach (K2_0.JointType jointType in body.Joints.Keys)
```

```
            {
                if (body.Joints[jointType].TrackingState != K2_0.Tracking
                {
                    this.Joints.Add(jointType.ToString(), KinV2JointToPoi
                }
                else
                {
                    Console.WriteLine("WARNING: joint " + jointType.ToStr
                }
            }
        }

        // Method to construct a Point3D from a Kinect V1 Joint
        public Point3D KinV2JointToPoint3D(K2_0.Joint joint)
        {
            Point3D point = new Point3D(joint.Position.X, joint.Position.
            return point;
        }
    }
}
```

A.3 KinectHumanMovementEvaluation.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Media.Media3D;
using K1_8 = KHMESDK1_8; // wrapper for Kinect SDK 1.8
                         /*-----------------------------------------------
                         Key names for joints are:
                         "FootRight", "AnkleRight", "KneeRight", "HipRigh
                         "FootLeft", "AnkleLeft", "KneeLeft", "HipLeft",
                         "HandRight", "WristRight", "ElbowRight", "Shoulc
                         "HandLeft", "WristLeft", "ElbowLeft", "ShoulderL
                         "HipCenter", "Spine", "ShoulderCenter", "Head"
                         -----------------------------------------------*/

using K2_0 = KHMESDK2_0; // wrapper for Kinect SDK 2.0
                         /*-----------------------------------------------
                         Key names for joints are:
```

```
"FootRight", "AnkleRight", "KneeRight", "HipRigh
"FootLeft", "AnkleLeft", "KneeLeft", "HipLeft",
"HandTipRight", "ThumbRight",
"HandRight", "WristRight", "ElbowRight", "Shoulc
"HandTipLeft", "ThumbLeft",
"HandLeft", "WristLeft", "ElbowLeft", "ShoulderI
"SpineBase", "SpineMid", "SpineShoulder", "Neck"
```

```
namespace KinectHumanMovementEvaluation
{
    public class Pose
    {
        public Dictionary<string, double> GenericParameters { get; set; }

        // Empty Constructor
        public Pose() { }

        // Complete Constructor: shallow copy of GenericParameters
        public Pose(Dictionary<string, double> genParam)
        {
            this.GenericParameters = genParam;
        }

        // Constructor from a Kinect SDK 1.8 Skeleton,
        // the parameters are defined as some representative joint angles
        public Pose(K1_8.Skeleton s)
        {
            this.GenericParameters = new Dictionary<string, double>();

            //Knees
            if(s.Joints.ContainsKey("AnkleRight") && s.Joints.ContainsKey
            {
                string rkn = "rightKneeAngle";
                double rkv = AuxGeometry.AngleMidPoint(s.Joints["AnkleRig
                this.GenericParameters.Add(rkn, rkv);
            }

            if (s.Joints.ContainsKey("AnkleLeft") && s.Joints.ContainsKey
```

```
{
    string lkn = "leftKneeAngle";
    double lkv = AuxGeometry.AngleMidPoint(s.Joints["AnkleLef
    this.GenericParameters.Add(lkn, lkv);
}

//Elbows
if (s.Joints.ContainsKey("WristRight") && s.Joints.ContainsKe
{
    string ren = "rightElbowAngle";
    double rev = AuxGeometry.AngleMidPoint(s.Joints["WristRig
    this.GenericParameters.Add(ren, rev);
}

if (s.Joints.ContainsKey("WristLeft") && s.Joints.ContainsKey
{
    string len = "leftElbowAngle";
    double lev = AuxGeometry.AngleMidPoint(s.Joints["WristLef
    this.GenericParameters.Add(len, lev);
}

//Hands
if (s.Joints.ContainsKey("HandRight") && s.Joints.ContainsKey
{
    string rhn = "rightHandAngle";
    double rhv = AuxGeometry.AngleMidPoint(s.Joints["HandRigh
    this.GenericParameters.Add(rhn, rhv);
}

if (s.Joints.ContainsKey("HandLeft") && s.Joints.ContainsKey(
{
    string lhn = "leftHandAngle";
    double lhv = AuxGeometry.AngleMidPoint(s.Joints["HandLeft
    this.GenericParameters.Add(lhn, lhv);
}

if (s.Joints.ContainsKey("HandRight") && s.Joints.ContainsKey
{
    string rhin = "rightHandInclination";
```

```
            double rhiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
            this.GenericParameters.Add(rhin, rhiv);
        }

        if (s.Joints.ContainsKey("HandLeft") && s.Joints.ContainsKey(
        {
            string lhin = "leftHandInclination";
            double lhiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
            this.GenericParameters.Add(lhin, lhiv);
        }

        //Feet
        if (s.Joints.ContainsKey("FootRight") && s.Joints.ContainsKey
        {
            string rfn = "rightFootAngle";
            double rfv = AuxGeometry.AngleMidPoint(s.Joints["FootRigh
            this.GenericParameters.Add(rfn, rfv);
        }

        if (s.Joints.ContainsKey("FootLeft") && s.Joints.ContainsKey(
        {
            string lfn = "leftFootAngle";
            double lfv = AuxGeometry.AngleMidPoint(s.Joints["FootLeft
            this.GenericParameters.Add(lfn, lfv);
        }

        if (s.Joints.ContainsKey("FootRight") && s.Joints.ContainsKey
        {
            string rfin = "rightFootInclination";
            double rfiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
            this.GenericParameters.Add(rfin, rfiv);
        }

        if (s.Joints.ContainsKey("FootLeft") && s.Joints.ContainsKey(
        {
            string lfin = "leftFootInclination";
            double lfiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
            this.GenericParameters.Add(lfin, lfiv);
        }
```

```
//Legs
if (s.Joints.ContainsKey("KneeRight") && s.Joints.ContainsKey
{
    string rlin = "rightLegInclination";
    double rliv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(rlin, rliv);
}

if (s.Joints.ContainsKey("KneeLeft") && s.Joints.ContainsKey(
{
    string llin = "leftLegInclination";
    double lliv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(llin, lliv);
}

if (s.Joints.ContainsKey("KneeRight") && s.Joints.ContainsKey
{
    string rlpn = "rightLegPendulum";
    double rlpv = AuxGeometry.AngleInPlanBetweenProjectionAnd
    this.GenericParameters.Add(rlpn, rlpv);
}

if (s.Joints.ContainsKey("KneeLeft") && s.Joints.ContainsKey(
{
    string llpn = "leftLegPendulum";
    double llpv = AuxGeometry.AngleInPlanBetweenProjectionAnd
    this.GenericParameters.Add(llpn, llpv);
}

//Arms
if (s.Joints.ContainsKey("ElbowRight") && s.Joints.ContainsKe
{
    string ran = "rightArmAngle";
    double rav = AuxGeometry.AngleMidPoint(s.Joints["ElbowRig
    this.GenericParameters.Add(ran, rav);
}

if (s.Joints.ContainsKey("ElbowLeft") && s.Joints.ContainsKey
```

```
{
    string lan = "leftArmAngle";
    double lav = AuxGeometry.AngleMidPoint(s.Joints["ElbowLe
    this.GenericParameters.Add(lan, lav);
}

if (s.Joints.ContainsKey("ElbowRight") && s.Joints.ContainsKe
{
    string rain = "rightArmInclination";
    double raiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(rain, raiv);
}

if (s.Joints.ContainsKey("ElbowLeft") && s.Joints.ContainsKey
{
    string lain = "leftArmInclination";
    double laiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(lain, laiv);
}

if (s.Joints.ContainsKey("WristRight") && s.Joints.ContainsKe
{
    string ratn = "rightArmTwist";
    double ratv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(ratn, ratv);
}

if (s.Joints.ContainsKey("WristLeft") && s.Joints.ContainsKey
{
    string latn = "leftArmTwistn";
    double latv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(latn, latv);
}

//Torso
if (s.Joints.ContainsKey("HipCenter") && s.Joints.ContainsKey
{
    string tn = "torsoAngle";
    double tv = AuxGeometry.AngleMidPoint(s.Joints["HipCenter
```

```
        this.GenericParameters.Add(tn, tv);
    }

    if (s.Joints.ContainsKey("HipRight") && s.Joints.ContainsKey(
    {
        string ttn = "torsoTwist";
        double ttv = AuxGeometry.AngleBetweenPlanes(s.Joints["Hip
        this.GenericParameters.Add(ttn, ttv);
    }

    if (s.Joints.ContainsKey("ShoulderCenter") && s.Joints.Contai
    {
        string tpn = "torsoPendulum";
        double tpv = AuxGeometry.AngleInPlanBetweenProjections(s.
        this.GenericParameters.Add(tpn, tpv);
    }

    //Head
    if (s.Joints.ContainsKey("Spine") && s.Joints.ContainsKey("Sh
    {
        string hn = "headAngle";
        double hv = AuxGeometry.AngleMidPoint(s.Joints["Spine"],
        this.GenericParameters.Add(hn, hv);
    }

    if (s.Joints.ContainsKey("Head") && s.Joints.ContainsKey("Sho
    {
        string hpn = "headPendulum";
        Vector3D vec1 = Point3D.Subtract(AuxGeometry.ProjectPoint
        Vector3D vec2 = Point3D.Subtract(s.Joints["Spine"], s.Joi
        double hpv = Vector3D.AngleBetween(vec1, vec2);
        this.GenericParameters.Add(hpn, hpv);
    }

    //Vertical inclination
    if (s.Joints.ContainsKey("Spine") && s.Joints.ContainsKey("Sh
    {
        string vn = "verticalInclination";
        double vv = AuxGeometry.AngleBetweenVectorAndVertical(s.J
```

```
            this.GenericParameters.Add(vn, vv);
        }
    }

    // Constructor from a Kinect SDK 2.0 Skeleton,
    // the parameters are defined as some representative joint angles
    public Pose(K2_0.Skeleton s)
    {
        this.GenericParameters = new Dictionary<string, double>();

        //Knees
        if (s.Joints.ContainsKey("AnkleRight") && s.Joints.ContainsKe
        {
            string rkn = "rightKneeAngle";
            double rkv = AuxGeometry.AngleMidPoint(s.Joints["AnkleRig
            this.GenericParameters.Add(rkn, rkv);
        }

        if (s.Joints.ContainsKey("AnkleLeft") && s.Joints.ContainsKey
        {
            string lkn = "leftKneeAngle";
            double lkv = AuxGeometry.AngleMidPoint(s.Joints["AnkleLef
            this.GenericParameters.Add(lkn, lkv);
        }

        //Elbows
        if (s.Joints.ContainsKey("WristRight") && s.Joints.ContainsKe
        {
            string ren = "rightElbowAngle";
            double rev = AuxGeometry.AngleMidPoint(s.Joints["WristRig
            this.GenericParameters.Add(ren, rev);
        }

        if (s.Joints.ContainsKey("WristLeft") && s.Joints.ContainsKey
        {
            string len = "leftElbowAngle";
            double lev = AuxGeometry.AngleMidPoint(s.Joints["WristLef
            this.GenericParameters.Add(len, lev);
        }
```

```
//Hands
if (s.Joints.ContainsKey("HandRight") && s.Joints.ContainsKey
{
    string rhn = "rightHandAngle";
    double rhv = AuxGeometry.AngleMidPoint(s.Joints["HandRigh
    this.GenericParameters.Add(rhn, rhv);
}

if (s.Joints.ContainsKey("HandLeft") && s.Joints.ContainsKey(
{
    string lhn = "leftHandAngle";
    double lhv = AuxGeometry.AngleMidPoint(s.Joints["HandLeft
    this.GenericParameters.Add(lhn, lhv);
}

if (s.Joints.ContainsKey("HandRight") && s.Joints.ContainsKey
{
    string rhin = "rightHandInclination";
    double rhiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(rhin, rhiv);
}

if (s.Joints.ContainsKey("HandLeft") && s.Joints.ContainsKey(
{
    string lhin = "leftHandInclination";
    double lhiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(lhin, lhiv);
}

//Feet
if (s.Joints.ContainsKey("FootRight") && s.Joints.ContainsKey
{
    string rfn = "rightFootAngle";
    double rfv = AuxGeometry.AngleMidPoint(s.Joints["FootRigh
    this.GenericParameters.Add(rfn, rfv);
}

if (s.Joints.ContainsKey("FootLeft") && s.Joints.ContainsKey(
```

```
{
    string lfn = "leftFootAngle";
    double lfv = AuxGeometry.AngleMidPoint(s.Joints["FootLeft
    this.GenericParameters.Add(lfn, lfv);
}

if (s.Joints.ContainsKey("FootRight") && s.Joints.ContainsKey
{
    string rfin = "rightFootInclination";
    double rfiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(rfin, rfiv);
}

if (s.Joints.ContainsKey("FootLeft") && s.Joints.ContainsKey(
{
    string lfin = "leftFootInclination";
    double lfiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(lfin, lfiv);
}

//Legs
if (s.Joints.ContainsKey("KneeRight") && s.Joints.ContainsKey
{
    string rlin = "rightLegInclination";
    double rliv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(rlin, rliv);
}

if (s.Joints.ContainsKey("KneeLeft") && s.Joints.ContainsKey(
{
    string llin = "leftLegInclination";
    double lliv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(llin, lliv);
}

if (s.Joints.ContainsKey("KneeRight") && s.Joints.ContainsKey
{
    string rlpn = "rightLegPendulum";
    double rlpv = AuxGeometry.AngleInPlanBetweenProjectionAnd
```

```
            this.GenericParameters.Add(rlpn, rlpv);
        }

        if (s.Joints.ContainsKey("KneeLeft") && s.Joints.ContainsKey(
        {
            string llpn = "leftLegPendulum";
            double llpv = AuxGeometry.AngleInPlanBetweenProjectionAnd
            this.GenericParameters.Add(llpn, llpv);
        }

        //Arms
        if (s.Joints.ContainsKey("ElbowRight") && s.Joints.ContainsKe
        {
            string ran = "rightArmAngle";
            double rav = AuxGeometry.AngleMidPoint(s.Joints["ElbowRig
            this.GenericParameters.Add(ran, rav);
        }

        if (s.Joints.ContainsKey("ElbowLeft") && s.Joints.ContainsKey
        {
            string lan = "leftArmAngle";
            double lav = AuxGeometry.AngleMidPoint(s.Joints["ElbowLe
            this.GenericParameters.Add(lan, lav);
        }

        if (s.Joints.ContainsKey("ElbowRight") && s.Joints.ContainsKe
        {
            string rain = "rightArmInclination";
            double raiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
            this.GenericParameters.Add(rain, raiv);
        }

        if (s.Joints.ContainsKey("ElbowLeft") && s.Joints.ContainsKey
        {
            string lain = "leftArmInclination";
            double laiv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
            this.GenericParameters.Add(lain, laiv);
        }
```

```
if (s.Joints.ContainsKey("WristRight") && s.Joints.ContainsKe
{
    string ratn = "rightArmTwist";
    double ratv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(ratn, ratv);
}

if (s.Joints.ContainsKey("WristLeft") && s.Joints.ContainsKey
{
    string latn = "leftArmTwistn";
    double latv = AuxGeometry.AngleBetweenVectorAndPlan(s.Joi
    this.GenericParameters.Add(latn, latv);
}

//Torso
if (s.Joints.ContainsKey("SpineBase") && s.Joints.ContainsKey
{
    string tn = "torsoAngle";
    double tv = AuxGeometry.AngleMidPoint(s.Joints["SpineBase
    this.GenericParameters.Add(tn, tv);
}

if (s.Joints.ContainsKey("HipRight") && s.Joints.ContainsKey(
{
    string ttn = "torsoTwist";
    double ttv = AuxGeometry.AngleBetweenPlanes(s.Joints["Hip
    this.GenericParameters.Add(ttn, ttv);
}

if (s.Joints.ContainsKey("SpineShoulder") && s.Joints.Contain
{
    string tpn = "torsoPendulum";
    double tpv = AuxGeometry.AngleInPlanBetweenProjections(s.
    this.GenericParameters.Add(tpn, tpv);
}

//Neck
if (s.Joints.ContainsKey("SpineMid") && s.Joints.ContainsKey(
{
```

```
                string nn = "neckAngle";
                double nv = AuxGeometry.AngleMidPoint(s.Joints["SpineMid"
                this.GenericParameters.Add(nn, nv);
            }

            //Head
            if (s.Joints.ContainsKey("SpineShoulder") && s.Joints.Contain
            {
                string hn = "headAngle";
                double hv = AuxGeometry.AngleMidPoint(s.Joints["SpineShou
                this.GenericParameters.Add(hn, hv);
            }

            if (s.Joints.ContainsKey("Head") && s.Joints.ContainsKey("Sho
            {
                string hpn = "headPendulum";
                Vector3D vec1 = Point3D.Subtract(AuxGeometry.ProjectPoint
                Vector3D vec2 = Point3D.Subtract(s.Joints["SpineMid"], s.
                double hpv = Vector3D.AngleBetween(vec1, vec2);
                this.GenericParameters.Add(hpn, hpv);
            }

            //Vertical inclination
            if (s.Joints.ContainsKey("SpineMid") && s.Joints.ContainsKey(
            {
                string vn = "verticalInclination";
                double vv = AuxGeometry.AngleBetweenVectorAndVertical(s.J
                this.GenericParameters.Add(vn, vv);
            }
        }
    }

    public class Movement
    {
        public List<Pose> Poses { get; set; }

        // Empty Constructor
        public Movement() { }
```

```
// Constructor that reads from a .txt file
// Each parameter must be on a different line, its different temp
public Movement(string path)
{
    string[] lines = System.IO.File.ReadAllLines(path);
    double[][] doubleValues = new double[lines.Count()][];
    int nbOfParams = lines.Count();
    int nbOfInstants = 0;

    for (int j = 0; j < nbOfParams; j++)
    {
        string[] stringValues = lines[j].Split(' ');
        nbOfInstants = stringValues.Count() - 1;

        doubleValues[j] = new double[nbOfInstants + 1];
        for (int i = 0; i < nbOfInstants; i++) // discards last v
        {
            doubleValues[j][i] = Convert.ToDouble(stringValues[i]
        }
    }

    this.Poses = new List<Pose>();

    for (int i = 0; i < nbOfInstants; i++)
    {
        Pose pose = new Pose();
        pose.GenericParameters = new Dictionary<string, double>()

        for (int j = 0; j < nbOfParams; j++)
        {
            string name = "Parameter " + j;
            pose.GenericParameters.Add(name, doubleValues[j][i]);
        }

        this.Poses.Add(pose);
    }
}

// Constructor that receives a list of Poses
```

```
        public Movement(List<Pose> poses)
        {
            this.Poses = new List<Pose>();

            foreach(Pose pose in poses)
            {
                this.Poses.Add(pose);
            }
        }
    }

    public static class Evaluation
    {
        /* Spheres Method for two given poses.----------------------------
        Spheres Method for two given poses.
        Returns positive or zero float as a score for success or negative
        Score represents ratio dist(attempt, standard)/minDist.
        ---------------------------------------------------------------------

        public static double Spheres(Pose standard, Pose attempt, double
        {
            double score = 0;

           if (standard.GenericParameters.Count != attempt.GenericParame
            {
                score = -1;
                Console.WriteLine("FAILURE: poses do not have same numbe
            }

            foreach(KeyValuePair<string, double> parameter in standard.G
            {
                if(! attempt.GenericParameters.ContainsKey(parameter.Key)
                {
                    score = -1;
                    Console.WriteLine("FAILURE: attempt pose does not hav
                }
            }

            if(score > -1)
            {
```

```
            double distance = AuxGeometry.EuclideanNorm(standard, att
            if (distance < minRadius)
            {
                score = 1;
                Console.WriteLine("TOTAL SUCCESS");
            }
            else
            {
                score = minRadius / distance;
                if (score < minScore)
                {
                    score = -1;
                    Console.WriteLine("FAILURE: minimum distance was
                }
            }
        }

        return score;
    }

    /* Spheres Method for two given movements.----------------------
    Returns positive or zero float as a score for success or negative
    Score represents average of all poses' scores, each one indicatin
    were needed to increase the radius until the sphere contained the
    ------------------------------------------------------------------

    public static double Spheres(Movement standard, Movement attempt,
    {
        double finalScore = 0; // final score for complete movement
        double r_n = minRadius;
        int currentPoseIndex = 0;

        // Reads until 'attempt' enters the first sphere
        for ( ; currentPoseIndex < attempt.Poses.Count &&
            !IsInSphere(attempt.Poses[currentPoseIndex], standard.Pos
        // Checks if 'attempt' still has poses to be used
        if(! (currentPoseIndex < attempt.Poses.Count))
        {
            Console.WriteLine("FAILURE: Never entered the first spher
            return -1.0;
```

```
}

// Reads until 'attempt' leaves first sphere
for ( ; currentPoseIndex < attempt.Poses.Count &&
    IsInSphere(attempt.Poses[currentPoseIndex], standard.Pose
// Checks if 'attempt' still has poses to be used
if (!(currentPoseIndex < attempt.Poses.Count))
{
    Console.WriteLine("FAILURE: Never left the first sphere")
    return -1.0;
}

bool isInNextSphere = false;

double score; // score between spheres
Pose midPose; // auxiliary variable

for (int j = 0;
    currentPoseIndex < attempt.Poses.Count() &&
    j < standard.Poses.Count; j++)
{
    score = 1;
    midPose = MiddlePose(standard.Poses[j], standard.Poses[j

    // Reads until 'attempt' enters next sphere or minimum ac
    for (; currentPoseIndex < attempt.Poses.Count &&
    !IsInSphere(attempt.Poses[currentPoseIndex], standard.Pos
    {
        // Increases radius of transition sphere until 'attem
        for (; !IsInSphere(attempt.Poses[currentPoseIndex], m
            score = score - scoreStep)
        {
            // minimum accuracy is not satisfied
            if (score <= minScore)
            {
                Console.WriteLine("FAILURE: minimum accuracy
                return -1.0;
            }
```

```
            r_n += radiusStep;
        }
    }
    // Checks if 'attempt' still has poses to be used
    if (!(currentPoseIndex < attempt.Poses.Count))
    {
        Console.WriteLine("Score was " + (finalScore / (stand
            " but never entered the " + (j + 1) + " sphere");
        return finalScore / (standard.Poses.Count - 1);
    }

    // Reads until 'attempt' leaves next sphere or reaches th
    for (; currentPoseIndex < attempt.Poses.Count &&
        IsInSphere(attempt.Poses[currentPoseIndex], standard.
    {
        // reaches the end of the movement
        if (j + 1 == standard.Poses.Count)
        {
            Console.WriteLine("SUCCESS");
            return finalScore / (standard.Poses.Count - 1); /
        }
        else
        {
            isInNextSphere = true;
        }
    }
    // Checks if 'attempt' still has poses to be used
    if (!(currentPoseIndex < attempt.Poses.Count))
    {
        Console.WriteLine("Score was " + (finalScore / (stand
            " but never left the " + (j + 1) + " sphere");
        return finalScore / (standard.Poses.Count - 1);
    }

    if (isInNextSphere)
    {
        finalScore += score;
        score = 1;
        r_n = minRadius;
```

```
                isInNextSphere = false;
            }
        }

        Console.WriteLine("Movement didn't end?");
        return -1.0;
    }

    public static bool IsInSphere(Pose attemptPose, Pose standardPose
    {
        return AuxGeometry.EuclideanNorm(attemptPose, standardPose) <
    }

    public static Pose MiddlePose(Pose pose1, Pose pose2)
    {
        bool flag = false;
        Pose middlePose = new Pose(new Dictionary<string, double>());

        if (pose1.GenericParameters.Count != pose2.GenericParameters.
        {
            flag = true;
            Console.WriteLine("FAILURE: poses do not have same number
        }

        foreach (KeyValuePair<string, double> parameter in pose1.Gen
        {
            if (! pose2.GenericParameters.ContainsKey(parameter.Key))
            {
                flag = true;
                Console.WriteLine("FAILURE: attempt pose does not hav
            }
        }

        if (! flag)
        {
            double midValue;

            foreach (KeyValuePair<string, double> parameter in pose1.
            {
```

```
                midValue = 0.5 * (parameter.Value + pose2.GenericPara
                middlePose.GenericParameters.Add(parameter.Key, midV
            }
        }

        return middlePose;
    }

    public static double TransitionRadius(Pose pose1, Pose pose2, dou
    {
        return Math.Sqrt(Math.Pow(radius, 2) + Math.Pow(0.5*AuxGeome
    }
}

public static class AuxGeometry
{
    public static double EuclideanNorm(Pose pose1, Pose pose2)
    {
        double norm = 0;

        if (pose1.GenericParameters.Count != pose2.GenericParameters.
        {
            norm = -1;
            Console.WriteLine("FAILURE: poses do not have same numbe
        }

        foreach (KeyValuePair<string, double> parameter in pose1.Gen
        {
            if (!pose2.GenericParameters.ContainsKey(parameter.Key))
            {
                norm = -1;
                Console.WriteLine("FAILURE: attempt pose does not hav
            }
        }
        if (norm > -1)
        {
            foreach (KeyValuePair<string, double> parameter in pose1.
            {
                norm += Math.Pow((parameter.Value - pose2.GenericPara
```

```
            }
            norm = Math.Sqrt(norm);
        }

        return norm;
    }

    public static double AngleMidPoint(Point3D point1, Point3D middle
    {
        Vector3D vec1 = Point3D.Subtract(point1, middlePoint);
        Vector3D vec2 = Point3D.Subtract(point2, middlePoint);

        // Check for geometric singularities
        Vector3D crProd = Vector3D.CrossProduct(vec1, vec2);
        double triangleArea = 0.5 * Math.Sqrt(Vector3D.DotProduct(crP
        double answer;
        if (triangleArea < 0.01)
        {
            Console.WriteLine("WARNING: Geometric singularity.");
            answer = -1;
        }

        else
        {
            answer = Vector3D.AngleBetween(vec2, vec1);
        }

        return answer;
    }

    public static Vector3D NormalToPlan(Point3D point1, Point3D point
    {
        Vector3D vec1OfPlan = Point3D.Subtract(point1, point3);
        Vector3D vec2OfPlan = Point3D.Subtract(point2, point3);

        // Check for geometric singularities
        Vector3D crProd = Vector3D.CrossProduct(vec1OfPlan, vec2OfPla
        double triangleArea = 0.5 * Math.Sqrt(Vector3D.DotProduct(crP
        Vector3D answer;
```

```
        if (triangleArea < 0.01)
        {
            Console.WriteLine("WARNING: Geometric singularity.");
            answer = new Vector3D();
        }

        else
        {
            answer = Vector3D.CrossProduct(vec1OfPlan, vec2OfPlan);
            answer.Normalize();
        }

        return answer;
    }

    public static double AngleBetweenVectorAndPlan(Point3D pointOutsi
    {
        Vector3D vecOutPlan = Point3D.Subtract(pointOutsidePlan, comm
        Vector3D normal = NormalToPlan(commonPoint, point1, point2);

        // Check for geometric singularities
        Vector3D crProd = Vector3D.CrossProduct(Point3D.Subtract(comm
        double triangleArea = 0.5 * Math.Sqrt(Vector3D.DotProduct(crP
        double vecSize = vecOutPlan.Length;
        double answer;
        if (triangleArea < 0.01 || vecSize < 0.01)
        {
            Console.WriteLine("WARNING: Geometric singularity.");
            answer = -1;
        }

        else
        {
            answer = Vector3D.AngleBetween(vecOutPlan, normal);
            if (answer > 90) answer = 180 - answer;
            answer = 90 - answer;
        }

        return answer;
```

```
        }

        public static double AngleBetweenPlanes(Point3D point11, Point3D
        {
            Vector3D normal1 = NormalToPlan(point11, point12, point13);
            Vector3D normal2 = NormalToPlan(point21, point22, point23);

            return Vector3D.AngleBetween(normal1, normal2);
        }

        public static double AngleBetweenVectorAndVertical(Point3D point1
        {
            Vector3D vector = Point3D.Subtract(point1, point2);

            // Check for geometric singularities
            double answer;
            if (vector.Length < 0.01)
            {
                Console.WriteLine("WARNING: Geometric singularity.");
                answer = -1;
            }

            else
            {
                Vector3D vert = Vector3D.Multiply(vector.Length, new Vect
                answer = Vector3D.AngleBetween(vector, vert);
            }

            return answer;
        }

        public static Point3D ProjectPointOntoPlan(Point3D pointOutsidePl
        {
            // Check for geometric singularities
            Vector3D crProd = Vector3D.CrossProduct(Point3D.Subtract(poin
            double triangleArea = 0.5 * Math.Sqrt(Vector3D.DotProduct(crP
            Point3D answer;
            if (triangleArea < 0.01)
            {
```

```
                Console.WriteLine("WARNING: Geometric singularity.");
                answer = new Point3D();
            }

            else
            {
                // Calculates the projected point
                Vector3D normal = Vector3D.CrossProduct(Point3D.Subtract(
                normal.Normalize();
                double distance = Vector3D.DotProduct(normal, Point3D.Sub
                answer = Point3D.Subtract(pointOutsidePlan, Vector3D.Mul
            }

            return answer;
        }

        public static double AngleInPlanBetweenProjectionAndVector(Point3
        {
            Point3D projP = ProjectPointOntoPlan(pointOutsidePlan, commo

            return AngleMidPoint(projP, commonPoint, referencePoint);
        }

        public static double AngleInPlanBetweenProjections(Point3D extrem
        {
            Point3D extrProj = ProjectPointOntoPlan(extremityPoint, comm
            Point3D commOutProj = ProjectPointOntoPlan(commonPointOutside
            Vector3D vec1 = Point3D.Subtract(extrProj, commOutProj);
            Vector3D vec2 = Point3D.Subtract(commOutProj, commonPoint);

            return Vector3D.AngleBetween(vec2, vec1);
        }
    }
}
```

APÊNDICE B – Código Fonte: Algoritmo para Testes de Precisão

B.1 testgenerator.py

```
import math
from matplotlib import pyplot as plt
import numpy as np

oneLineS = np.linspace(0,125.6637,160)
# defined from 0 to 40*pi, representing 20 repetitions

threeParamsS = np.vstack((oneLineS,oneLineS,oneLineS))

threeParamsS[0,] = math.pi*np.cos(threeParamsS[0,])
threeParamsS[1,] = math.pi*np.sin(threeParamsS[1,])
threeParamsS[2,] = math.pi*np.sin(0.5*threeParamsS[2,])

fileStandard = open("standardMovement3Params.txt", "w")
fileStandard = open("standardMovement3Params.txt", "a+")

for row in threeParamsS:
  for value in row:
    fileStandard.write("%f " % value)
  fileStandard.write("\n")

fileStandard.close()

oneLineA = np.linspace(0,125.6637,15*160)
# sampled 15 times more often than the Standard

threeParamsA = np.vstack((oneLineA,oneLineA,oneLineA))

noise1 = np.random.normal(0,.05,15*160)
noise2 = np.random.normal(0,.05,15*160)
noise3 = np.random.normal(0,.05,15*160)
```

```
# 0 is the mean of the normal distribution
# 1 is the standard deviation of the normal distribution

threeParamsA[0,] = math.pi*np.cos(threeParamsA[0,]) + noise1
threeParamsA[1,] = math.pi*np.sin(threeParamsA[1,]) + noise2
threeParamsA[2,] = math.pi*np.sin(0.5*threeParamsA[2,]) + noise3

fileAttempt = open("attemptMovement3ParamsDotOFive.txt", "w")
fileAttempt = open("attemptMovement3ParamsDotOFive.txt", "a+")

for row in threeParamsA:
  for value in row:
    fileAttempt.write("%f " % value)
  fileAttempt.write("\n")

fileAttempt.close()
```

B.2 jointsPlotter.py

```
from matplotlib import pyplot as plt
import math

file1 = open('laj.txt', 'r')
lines = [ line.split() for line in file1 ]

SpineBase = []
SpineMid = []
Neck = []
Head = []
ShoulderLeft = []
ElbowLeft = []
WristLeft = []
HandLeft = []
ShoulderRight = []
ElbowRight = []
WristRight = []
HandRight = []
HipLeft = []
KneeLeft = []
AnkleLeft = []
```

```
FootLeft = []
HipRight = []
KneeRight = []
AnkleRight = []
FootRight = []
SpineShoulder = []
HandTipLeft = []
ThumbLeft = []
HandTipRight = []
ThumbRight = []

for joint in lines:

  if joint[0] == 'SpineBase':
    SpineBase.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'SpineMid':
    SpineMid.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'Neck':
    Neck.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'Head':
    Head.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'ShoulderLeft':
    ShoulderLeft.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'ElbowLeft':
```

```
    ElbowLeft.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'WristLeft':
    WristLeft.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'HandLeft':
    HandLeft.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'ShoulderRight':
    ShoulderRight.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'ElbowRight':
    ElbowRight.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'WristRight':
    WristRight.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'HandRight':
    HandRight.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])

  if joint[0] == 'HipLeft':
    HipLeft.append([float(joint[1].replace(',','.')),
    float(joint[2].replace(',','.')),
    float(joint[3].replace(',','.'))])
```

```
if joint[0] == 'KneeLeft':
  KneeLeft.append([float(joint[1].replace(',','.')),
  float(joint[2].replace(',','.')),
  float(joint[3].replace(',','.'))])

if joint[0] == 'AnkleLeft':
  AnkleLeft.append([float(joint[1].replace(',','.')),
  float(joint[2].replace(',','.')),
  float(joint[3].replace(',','.'))])

if joint[0] == 'FootLeft':
  FootLeft.append([float(joint[1].replace(',','.')),
  float(joint[2].replace(',','.')),
  float(joint[3].replace(',','.'))])

if joint[0] == 'HipRight':
  HipRight.append([float(joint[1].replace(',','.')),
  float(joint[2].replace(',','.')),
  float(joint[3].replace(',','.'))])

if joint[0] == 'KneeRight':
  KneeRight.append([float(joint[1].replace(',','.')),
  float(joint[2].replace(',','.')),
  float(joint[3].replace(',','.'))])

if joint[0] == 'AnkleRight':
  AnkleRight.append([float(joint[1].replace(',','.')),
  float(joint[2].replace(',','.')),
  float(joint[3].replace(',','.'))])

if joint[0] == 'FootRight':
  FootRight.append([float(joint[1].replace(',','.')),
  float(joint[2].replace(',','.')),
  float(joint[3].replace(',','.'))])

if joint[0] == 'SpineShoulder':
  SpineShoulder.append([float(joint[1].replace(',','.')),
  float(joint[2].replace(',','.')),
  float(joint[3].replace(',','.'))])
```

```
    if joint[0] == 'HandTipLeft':
      HandTipLeft.append([float(joint[1].replace(',','.')),
      float(joint[2].replace(',','.')),
      float(joint[3].replace(',','.'))])

    if joint[0] == 'ThumbLeft':
      ThumbLeft.append([float(joint[1].replace(',','.')),
      float(joint[2].replace(',','.')),
      float(joint[3].replace(',','.'))])

    if joint[0] == 'HandTipRight':
      HandTipRight.append([float(joint[1].replace(',','.')),
      float(joint[2].replace(',','.')),
      float(joint[3].replace(',','.'))])

    if joint[0] == 'ThumbRight':
      ThumbRight.append([float(joint[1].replace(',','.')),
      float(joint[2].replace(',','.')),
      float(joint[3].replace(',','.'))])

WristRightX = []
WristRightY = []
WristRightZ = []

for coords in WristLeft:
    WristRightX.append(coords[0])
    WristRightY.append(coords[1])
    WristRightZ.append(coords[2])

ElbowRightX = []
ElbowRightY = []
ElbowRightZ = []

for coords in ElbowLeft:
    ElbowRightX.append(coords[0])
    ElbowRightY.append(coords[1])
    ElbowRightZ.append(coords[2])
```

```
ShoulderRightX = []
ShoulderRightY = []
ShoulderRightZ = []

for coords in ShoulderLeft:
  ShoulderRightX.append(coords[0])
  ShoulderRightY.append(coords[1])
  ShoulderRightZ.append(coords[2])

# chosen =

plt.plot(chosen)
plt.show()
```

B.3 genParmsPlotter.py

```
from matplotlib import pyplot as plt

file1 = open('lsgp.txt', 'r')
lines = [ line.split() for line in file1 ]

rightKneeAngle = []
leftKneeAngle = []
rightElbowAngle = []
leftElbowAngle = []
rightHandAngle = []
leftHandAngle = []
rightHandInclination = []
leftHandInclination = []
rightFootAngle = []
leftFootAngle = []
rightFootInclination = []
leftFootInclination = []
rightLegInclination = []
leftLegInclination = []
rightLegPendulum = []
leftLegPendulum = []
rightArmAngle = []
leftArmAngle = []
rightArmInclination = []
```

```
leftArmInclination = []
rightArmTwist = []
leftArmTwistn = []
torsoAngle = []
torsoTwist = []
torsoPendulum = []
neckAngle = []
headAngle = []
headPendulum = []
verticalInclination = []

for genericParameter in lines:

  if genericParameter[0] == 'rightKneeAngle':
    rightKneeAngle.append(float(genericParameter[1].replace(',','.')))

  if genericParameter[0] == 'leftKneeAngle':
    leftKneeAngle.append(float(genericParameter[1].replace(',','.')))

  if genericParameter[0] == 'rightElbowAngle':
    rightElbowAngle.append(float(genericParameter[1].replace(',','.')))

  if genericParameter[0] == 'leftElbowAngle':
    leftElbowAngle.append(float(genericParameter[1].replace(',','.')))

  if genericParameter[0] == 'rightHandAngle':
    rightHandAngle.append(float(genericParameter[1].replace(',','.')))

  if genericParameter[0] == 'leftHandAngle':
    leftHandAngle.append(float(genericParameter[1].replace(',','.')))

  if genericParameter[0] == 'rightHandInclination':
    rightHandInclination.append(float(genericParameter[1].replace(',','.'

  if genericParameter[0] == 'leftHandInclination':
    leftHandInclination.append(float(genericParameter[1].replace(',','.')

  if genericParameter[0] == 'rightFootAngle':
    rightFootAngle.append(float(genericParameter[1].replace(',','.')))
```

```
if genericParameter[0] == 'leftFootAngle':
  leftFootAngle.append(float(genericParameter[1].replace(',','.')))

if genericParameter[0] == 'rightFootInclination':
  rightFootInclination.append(float(genericParameter[1].replace(',','.

if genericParameter[0] == 'leftFootInclination':
  leftFootInclination.append(float(genericParameter[1].replace(',','.')

if genericParameter[0] == 'rightLegInclination':
  rightLegInclination.append(float(genericParameter[1].replace(',','.')

if genericParameter[0] == 'leftLegInclination':
  leftLegInclination.append(float(genericParameter[1].replace(',','.')))

if genericParameter[0] == 'rightLegPendulum':
  rightLegPendulum.append(float(genericParameter[1].replace(',','.')))

if genericParameter[0] == 'leftLegPendulum':
  leftLegPendulum.append(float(genericParameter[1].replace(',','.')))

if genericParameter[0] == 'rightArmAngle':
  rightArmAngle.append(float(genericParameter[1].replace(',','.')))

if genericParameter[0] == 'leftArmAngle':
  leftArmAngle.append(float(genericParameter[1].replace(',','.')))

if genericParameter[0] == 'rightArmInclination':
  rightArmInclination.append(float(genericParameter[1].replace(',','.')

if genericParameter[0] == 'leftArmInclination':
  leftArmInclination.append(float(genericParameter[1].replace(',','.')))

if genericParameter[0] == 'rightArmTwist':
  rightArmTwist.append(float(genericParameter[1].replace(',','.')))

if genericParameter[0] == 'leftArmTwistn':
  leftArmTwistn.append(float(genericParameter[1].replace(',','.')))
```

```
    if genericParameter[0] == 'torsoAngle':
      torsoAngle.append(float(genericParameter[1].replace(',','.')))

    if genericParameter[0] == 'torsoTwist':
      torsoTwist.append(float(genericParameter[1].replace(',','.')))

    if genericParameter[0] == 'torsoPendulum':
      torsoPendulum.append(float(genericParameter[1].replace(',','.')))

    if genericParameter[0] == 'neckAngle':
      neckAngle.append(float(genericParameter[1].replace(',','.')))

    if genericParameter[0] == 'headAngle':
      headAngle.append(float(genericParameter[1].replace(',','.')))

    if genericParameter[0] == 'headPendulum':
      headPendulum.append(float(genericParameter[1].replace(',','.')))

    if genericParameter[0] == 'verticalInclination':
      verticalInclination.append(float(genericParameter[1].replace(',','.')

# chosen =

plt.plot(chosen)
plt.show()
```

APÊNDICE C – Código Fonte: Aplicativo Kinect V1

```
namespace Microsoft.Samples.Kinect.SkeletonBasics
{
using System.IO;
using System.Windows;
using System.Windows.Media;
using Microsoft.Kinect;
using System;
using System.Collections.Generic;
using System.Linq;
using KinectHumanMovementEvaluation;
using KHMESDK1_8 = KHMESDK1_8;
using KHMESDK2_0 = KHMESDK2_0;

/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : Window
{
/// <summary>
/// Width of output drawing
/// </summary>
private const float RenderWidth = 640.0f;

/// <summary>
/// Height of our output drawing
/// </summary>
private const float RenderHeight = 480.0f;

/// <summary>
/// Thickness of drawn joint lines
/// </summary>
private const double JointThickness = 3;
```

```
/// <summary>
/// Thickness of body center ellipse
/// </summary>
private const double BodyCenterThickness = 10;

/// <summary>
/// Thickness of clip edge rectangles
/// </summary>
private const double ClipBoundsThickness = 10;

/// <summary>
/// Brush used to draw skeleton center point
/// </summary>
private readonly Brush centerPointBrush = Brushes.Blue;

/// <summary>
/// Brush used for drawing joints that are currently tracked
/// </summary>
private readonly Brush trackedJointBrush = new SolidColorBrush(Color.From

/// <summary>
/// Brush used for drawing joints that are currently inferred
/// </summary>
private readonly Brush inferredJointBrush = Brushes.Yellow;

/// <summary>
/// Pen used for drawing bones that are currently tracked
/// </summary>
private readonly Pen trackedBonePen = new Pen(Brushes.Green, 6);

/// <summary>
/// Pen used for drawing bones that are currently inferred
/// </summary>
private readonly Pen inferredBonePen = new Pen(Brushes.Gray, 1);

/// <summary>
/// Active Kinect sensor
/// </summary>
```

```
private KinectSensor sensor;

/// <summary>
/// Drawing grousp for skeleton rendering output
/// </summary>
private DrawingGroup drawingGroup;
private DrawingGroup drawingGroup2;

/// <summary>
/// Drawing images that we will display
/// </summary>
private DrawingImage imageSource;
private DrawingImage imageSource2;

//
//
// New variables:
//
//

// Flags:

/// <summary>
/// "Photo" button handling flag
/// </summary>
private bool skelPhoto = false;

/// <summary>
/// "Save" button handling flag
/// </summary>
private bool skelSave = false;

/// <summary>
/// "Capture Mode" button handling flag
/// </summary>
private bool captureModeOn = true;

/// <summary>
/// "Comparing Mode" button handling flag
```

```
/// </summary>
private bool comparingModeOn = false;

/// <summary>
/// "Model" checkbox handling flag
/// </summary>
private bool modelOn = false;

/// <summary>
/// "Load Model Photo" button handling flag
/// </summary>
private bool loadModelPhoto = false;

/// <summary>
/// "Load Photo" button handling flag
/// </summary>
private bool loadPhoto = false;

/// <summary>
/// "Seated Mode" checkbox handling flag
/// </summary>
private bool seatedModeOn = false;

/// <summary>
/// Flags the plotting of photos captured with "Seated Mode" enabled
/// </summary>
private bool plotInSeatedMode = false;

/// <summary>
/// Flags the plotting lock (by looping the same image) on the right scre
/// </summary>
private bool photoScreenLock = false;

/// <summary>
///  "REC" button handling flag
/// </summary>
private bool recOn = false;

/// <summary>
```

```
/// Flags the beginning of the "starting photo" selection process
/// </summary>
private bool startingPhotoModeOn = false;

/// <summary>
/// "Starting Photo" button handling flag
/// </summary>
private bool startingPhotoSelected = false;

/// <summary>
/// "Load Model Video" button handling flag
/// </summary>
private bool loadModelVideo = false;

/// <summary>
/// "Load Video" button handling flag
/// </summary>
private bool loadVideo = false;

/// <summary>
/// "Start Loop" button handling flag (left screen)
/// </summary>
private bool loopOnModel = false;

/// <summary>
/// "Start Loop" button handling flag (right screen)
/// </summary>
private bool loopOnUser = false;

// Skeleton data:

/// <summary>
/// Holds a pose data
/// </summary>
private Skeleton skelData = new Skeleton();

/// <summary>
/// Holds a movement data (set of poses)
/// </summary>
```

```
List<Skeleton> skelsData = new List<Skeleton>();

/// <summary>
/// Holds a skeleton data loaded from a text file (model)
/// </summary>
private Skeleton loadModelSkel = new Skeleton();

/// <summary>
/// Holds a skeleton data loaded from a text file (user)
/// </summary>
private Skeleton loadPhotoSkel = new Skeleton ();

// Indexes:

/// <summary>
/// Holds the currently plotted pose's index (model)
/// </summary>
int shiftValueModel = 0;

/// <summary>
/// Holds the currently plotted pose's index (user)
/// </summary>
int shiftValueUser = 0;

/// <summary>
/// Holds the selected starting photo's index
/// </summary>
int startingPhotoIndex = 0;

// Control variables:

/// <summary>
/// Used for the plotting speed control (model)
/// </summary>
int loopDelayModel = 0;

/// <summary>
/// Used for the plotting speed control (user)
/// </summary>
```

```
int loopDelayUser = 0;

/// <summary>
/// Used for the plotting speed control (setup value)
/// </summary>
int loopDelaySetup = 0;

/// <summary>
/// Holds the current movement's length (modelo)
/// </summary>
int videoLengthCountModel = 0;

/// <summary>
/// Holds the current movement's length (user)
/// </summary>
int videoLengthCountUser = 0;

/// <summary>
/// Used in Sphere method (pose evaluation).
/// Define how the evaluation process will occur, and must be assigned
before the program start.
/// </summary>
double radiusPose = 30;
double minScorePose = 0.3;

/// <summary>
/// Used in Sphere method (movement evaluation).
/// Define how the evaluation process will occur, and must be assigned
before the program start.
/// </summary>
double radiusMovement = 80;
double radiusStep = 30;
double minScoreMovement = 0.3;
double scoreStep = 0.02;

// Paths:

string photoCorePathModel = @"C:\Users\Gian\Desktop\Poli 2015\Segundo Ser
string photoCorePathUser = @"C:\Users\Gian\Desktop\Poli 2015\Segundo Sem
```

```
string videoCorePathModel = @"C:\Users\Gian\Desktop\Poli 2015\Segundo Ser
referencia";
string videoCorePathUser = @"C:\Users\Gian\Desktop\Poli 2015\Segundo Sem
string compareResultPath = @"C:\Users\Gian\Desktop\Poli 2015\Segundo Sem

// File names:

string photoCoreNameModel = "poseTeste_referencia";
string photoCoreNameUser = "poseTeste_usuario";
string videoCoreNameModel = "movimentoTeste_referencia_";
string videoCoreNameUser = "movimentoTeste_usuario_";
string compareResultName = "resultadoTeste";

//
//
// Main:
//
//

/// <summary>
/// Initializes a new instance of the MainWindow class.
/// </summary>
public MainWindow()
{
InitializeComponent();
}

/// <summary>
/// Draws indicators to show which edges are clipping skeleton data
/// </summary>
/// <param name="skeleton">skeleton to draw clipping information
for</param>
/// <param name="drawingContext">drawing context to draw to</param>
private static void RenderClippedEdges(Skeleton skeleton, DrawingContext
drawingContext)
{
if (skeleton.ClippedEdges.HasFlag(FrameEdges.Bottom))
{
drawingContext.DrawRectangle(
```

```
Brushes.Red,
null,
new Rect(0, RenderHeight - ClipBoundsThickness, RenderWidth,
ClipBoundsThickness));
}

if (skeleton.ClippedEdges.HasFlag(FrameEdges.Top))
{
drawingContext.DrawRectangle(
Brushes.Red,
null,
new Rect(0, 0, RenderWidth, ClipBoundsThickness));
}

if (skeleton.ClippedEdges.HasFlag(FrameEdges.Left))
{
drawingContext.DrawRectangle(
Brushes.Red,
null,
new Rect(0, 0, ClipBoundsThickness, RenderHeight));
}

if (skeleton.ClippedEdges.HasFlag(FrameEdges.Right))
{
drawingContext.DrawRectangle(
Brushes.Red,
null,
new Rect(RenderWidth - ClipBoundsThickness, 0, ClipBoundsThickness,
RenderHeight));
}
}

/// <summary>
/// Execute startup tasks
/// </summary>
/// <param name="sender">object sending the event</param>
/// <param name="e">event arguments</param>
private void WindowLoaded(object sender, RoutedEventArgs e)
{
```

```
// Create the drawing groups we'll use for drawing
this.drawingGroup = new DrawingGroup();
this.drawingGroup2 = new DrawingGroup();

// Create the image sources that we can use in our image control
this.imageSource = new DrawingImage(this.drawingGroup);
this.imageSource2 = new DrawingImage(this.drawingGroup2);

// Display the drawing using our image control
Image.Source = this.imageSource;
Image2.Source = this.imageSource2;

// Look through all sensors and start the first connected one.
// This requires that a Kinect is connected at the time of app startup.
// To make your app robust against plug/unplug.

foreach (var potentialSensor in KinectSensor.KinectSensors)
{
if (potentialSensor.Status == KinectStatus.Connected)
{
this.sensor = potentialSensor;
break;
}
}

if (null != this.sensor)
{
// Turn on the skeleton stream to receive skeleton frames
this.sensor.SkeletonStream.Enable();

// Add an event handler to be called whenever there is new color
frame data
this.sensor.SkeletonFrameReady += this.SensorSkeletonFrameReady;

// Start the sensor!
try
{
this.sensor.Start();
}
```

```
catch (IOException)
{
this.sensor = null;
}
}

if (null == this.sensor)
{
// "no Kinect ready" error
}
}

/// <summary>
/// Execute shutdown tasks
/// </summary>
/// <param name="sender">object sending the event</param>
/// <param name="e">event arguments</param>
private void WindowClosing(object sender,
System.ComponentModel.CancelEventArgs e)
{
if (null != this.sensor)
{
this.sensor.Stop();
}
}

/// <summary>
/// Event handler for Kinect sensor's SkeletonFrameReady event
/// </summary>
/// <param name="sender">object sending the event</param>
/// <param name="e">event arguments</param>
private void SensorSkeletonFrameReady(object sender,
SkeletonFrameReadyEventArgs e)
{

Skeleton[] skeletons = new Skeleton[0];

// copying the captured skeleton data
using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())
```

```
{
if (skeletonFrame != null)
{
skeletons = new Skeleton[skeletonFrame.SkeletonArrayLength];
skeletonFrame.CopySkeletonDataTo(skeletons);
}
}

// left screen
using (DrawingContext dc = this.drawingGroup.Open())
{
// draw a transparent background to set the render size
dc.DrawRectangle(Brushes.Black, null, new Rect(0.0, 0.0, RenderWidth,
RenderHeight));

// if capture mode is on
if (skeletons.Length != 0 && captureModeOn)
{
foreach (Skeleton skel in skeletons)
{
RenderClippedEdges(skel, dc);  // show clipping indicators

if (skel.TrackingState == SkeletonTrackingState.Tracked) // if
the skeleton
is being tracked
{
this.DrawBonesAndJoints(skel, dc); // draw skeleton
}

else if (skel.TrackingState == SkeletonTrackingState.PositionOnly)
only the position
{
// draw position
dc.DrawEllipse(
this.centerPointBrush,
null,
this.SkeletonPointToScreen(skel.Position),
BodyCenterThickness,
BodyCenterThickness);
```

```
}
}
}

// else if comparing mode is on
else if (comparingModeOn)
{
if (loadModelPhoto) // if loading photo
{
string[] modelLines = File.ReadAllLines(Path.Combine(photoCorePathModel,
photoCoreNameModel + ".txt"));
loadSkelDataFromText(modelLines, true);
this.DrawBonesAndJoints(loadModelSkel, dc);
}

else if (loadModelVideo) // else if loading video
{
if (loopOnModel == true) // if the video is being looped
{
// setting the plotting speed
if (loopDelayModel > 0) loopDelayModel--; // if a delay is set
(through loopDelaySetup), wait
else // after waiting (or if a delay is not set)
{
    shiftValueModel++; // incrementing index
    loopDelayModel = loopDelaySetup; // (re)setting delay
}
}

// checking current pose and closing loop
if (shiftValueModel < 0) shiftValueModel = videoLengthCountModel;
if (shiftValueModel > videoLengthCountModel) shiftValueModel = 0;

// reading data

string[] modelLines = File.ReadAllLines(Path.Combine(videoCorePathModel,
videoCoreNameModel + shiftValueModel + ".txt"));

loadSkelDataFromText(modelLines, true);
```

```
// plotting skeleton
this.DrawBonesAndJoints(loadModelSkel, dc);
}
}

// prevent drawing outside of our render area
this.drawingGroup.ClipGeometry = new RectangleGeometry(new Rect(0.0, 0.0,
RenderWidth, RenderHeight));
}

// Right screen
using (DrawingContext dc2 = this.drawingGroup2.Open())
{
// draw a transparent background to set the render size
dc2.DrawRectangle(Brushes.Black, null, new Rect(0.0, 0.0, RenderWidth,
RenderHeight));

// if capture mode is on
if (captureModeOn)
{
// if a photo (pose) is being taken
if (skelPhoto)
{
foreach (Skeleton skel in skeletons)
{
if (skel.TrackingState == SkeletonTrackingState.Tracked)
{
    this.DrawBonesAndJoints(skel, dc2); // plotting skeleton
    skelData = skel; // holding the data

}
}

skelsData.Add(skelData); // add the new skeleton to the recording data
(in this case, skelsData.length will always be 1)
skelPhoto = false; // end photo
}
```

```
// locking the screen (by looping the same skeleton), only happens when
a photo is taken ("Take Photo" button)
if (photoScreenLock) this.DrawBonesAndJoints(skelsData[0], dc2);

// if a video (movement) is being recorded
if (recOn)
{
foreach (Skeleton skel in skeletons)
{
if (skel.TrackingState == SkeletonTrackingState.Tracked)
{
    this.DrawBonesAndJoints(skel, dc2); // plotting skeleton
    skelData = skel; // holding the data

}
}

skelsData.Add(skelData); // add the new skeleton to the recording data
}

// selecting the starting photo
else if (startingPhotoModeOn)
{
// checking current pose and closing loop
if (shiftValueUser < 0) shiftValueUser = (skelsData.Count - 1);
if (shiftValueUser > (skelsData.Count - 1)) shiftValueUser = 0;

this.DrawBonesAndJoints(skelsData[shiftValueUser], dc2); // plotting the
currently selected pose

// if the "Starting Photo" button was pressed
if (startingPhotoSelected)
{
startingPhotoIndex = shiftValueUser; // holding starting photo index
startingPhotoSelected = false; // end starting photo selection
}
}
}
```

```
else if (comparingModeOn)
{
if (loadPhoto) // if a photo is being loaded
{
string[] photoLines = File.ReadAllLines(Path.Combine(photoCorePathUser,
photoCoreNameUser + ".txt"));
loadSkelDataFromText(photoLines, false);
this.DrawBonesAndJoints(loadPhotoSkel, dc2);
}

else if (loadVideo) // else if a video is being loaded
{
if (loopOnUser == true) // if the video is being looped
{
// setting the plotting speed
if (loopDelayUser > 0) loopDelayUser--; // if a delay is set
(through loopDelaySetup), wait
else // after waiting (or if a delay is not set)
{
    shiftValueUser++; // incrementing index
    loopDelayUser = loopDelaySetup; // (re)setting delay
}
}

// checking current pose and closing loop
if (shiftValueUser < 0) shiftValueUser = videoLengthCountUser;
if (shiftValueUser > videoLengthCountUser) shiftValueUser = 0;

// reading data
string[] photoLines = File.ReadAllLines(Path.Combine(videoCorePathUser,
videoCoreNameUser +
shiftValueUser + ".txt"));
loadSkelDataFromText(photoLines, false);

// plotting skeleton
this.DrawBonesAndJoints(loadPhotoSkel, dc2);
}
}
}
```

```
// Saving files
if (skelSave)
{
skelsData.RemoveRange(0, startingPhotoIndex); // setting the new starting
pose

for (int j = 0; j < skelsData.Count; j++) {

string stringData = null; // clearing the new line

foreach (Joint joint in skelsData[j].Joints)
{
if (seatedModeOn)
{

stringData += string.Format("{0} {1} {2} {3} SeatedModeOn" + Environment.
joint.JointType, joint.Position.X, joint.Position.Y, joint.Position.Z);
}

else
{
stringData += string.Format("{0} {1} {2} {3} SeatedModeOff" + Environment
joint.JointType, joint.Position.X, joint.Position.Y, joint.Position.Z);
}
}

// if it's a model
if (modelOn)
{
if (skelsData.Count == 1) // pose
{
File.WriteAllText(Path.Combine(photoCorePathModel, photoCoreNameModel + "
stringData);
}

else // movement
{
File.WriteAllText(Path.Combine(videoCorePathModel, videoCoreNameModel + j
```

```
stringData);
}
}

// if not
else if (!modelOn)
{
if (skelsData.Count == 1) // pose
{
File.WriteAllText(Path.Combine(photoCorePathUser, photoCoreNameUser + ".t
stringData);
}
else // movement
{
File.WriteAllText(Path.Combine(videoCorePathUser, videoCoreNameUser + j +
stringData);
}
}
}

skelSave = false; // end save
photoScreenLock = false; // end screen lock
skelsData.Clear(); // clearing the saved data
startingPhotoIndex = 0; // clearing the starting index
}
}

/// <summary>
/// Draws a skeleton's bones and joints
/// </summary>
/// <param name="skeleton">skeleton to draw</param>
/// <param name="drawingContext">drawing context to draw to</param>
private void DrawBonesAndJoints(Skeleton skeleton, DrawingContext drawing
{
// Render Torso
this.DrawBone(skeleton, drawingContext, JointType.Head, JointType.Shoulde
this.DrawBone(skeleton, drawingContext, JointType.ShoulderCenter, JointTy
this.DrawBone(skeleton, drawingContext, JointType.ShoulderCenter, JointTy
this.DrawBone(skeleton, drawingContext, JointType.ShoulderCenter, JointTy
```

```
this.DrawBone(skeleton, drawingContext, JointType.Spine, JointType.HipCe
this.DrawBone(skeleton, drawingContext, JointType.HipCenter, JointType.H
this.DrawBone(skeleton, drawingContext, JointType.HipCenter, JointType.Hi

// Left Arm
this.DrawBone(skeleton, drawingContext, JointType.ShoulderLeft, JointType
this.DrawBone(skeleton, drawingContext, JointType.ElbowLeft, JointType.W
this.DrawBone(skeleton, drawingContext, JointType.WristLeft, JointType.Ha

// Right Arm
this.DrawBone(skeleton, drawingContext, JointType.ShoulderRight, JointTyp
this.DrawBone(skeleton, drawingContext, JointType.ElbowRight, JointType.W
this.DrawBone(skeleton, drawingContext, JointType.WristRight, JointType.H

// Left Leg
this.DrawBone(skeleton, drawingContext, JointType.HipLeft, JointType.Knee
this.DrawBone(skeleton, drawingContext, JointType.KneeLeft, JointType.An
this.DrawBone(skeleton, drawingContext, JointType.AnkleLeft, JointType.Fo

// Right Leg
this.DrawBone(skeleton, drawingContext, JointType.HipRight, JointType.Kne
this.DrawBone(skeleton, drawingContext, JointType.KneeRight, JointType.An
this.DrawBone(skeleton, drawingContext, JointType.AnkleRight, JointType.F

// Render Joints
foreach (Joint joint in skeleton.Joints)
{
Brush drawBrush = null;

if (joint.TrackingState == JointTrackingState.Tracked || comparingModeOn)
{
drawBrush = this.trackedJointBrush;
}
else if (joint.TrackingState == JointTrackingState.Inferred)
{
drawBrush = this.inferredJointBrush;
}

if (drawBrush != null)
```

```
{
drawingContext.DrawEllipse(drawBrush, null, this.SkeletonPointToScreen(jo
JointThickness, JointThickness);
}
}
}

/// <summary>
/// Maps a SkeletonPoint to lie within our render space and converts to D
/// </summary>
/// <param name="skelpoint">point to map</param>
/// <returns>mapped point</returns>
private Point SkeletonPointToScreen(SkeletonPoint skelpoint)
{
// Convert point to depth space.
// We are not using depth directly, but we do want the points in our 640x
output resolution.
DepthImagePoint depthPoint = this.sensor.CoordinateMapper.MapSkeletonPoin
return new Point(depthPoint.X, depthPoint.Y);
}

/// <summary>
/// Draws a bone line between two joints
/// </summary>
/// <param name="skeleton">skeleton to draw bones from</param>
/// <param name="drawingContext">drawing context to draw to</param>
/// <param name="jointType0">joint to start drawing from</param>
/// <param name="jointType1">joint to end drawing at</param>
private void DrawBone(Skeleton skeleton, DrawingContext drawingContext, J
jointType0, JointType jointType1)
{
Joint joint0 = skeleton.Joints[jointType0];
Joint joint1 = skeleton.Joints[jointType1];

// If we can't find either of these joints, exit
if ((joint0.TrackingState == JointTrackingState.NotTracked ||
joint1.TrackingState == JointTrackingState.NotTracked) && captureModeOn)
{
return;
```

```
}

// Don't draw if both points are inferred
if ((joint0.TrackingState == JointTrackingState.Inferred &&
joint1.TrackingState == JointTrackingState.Inferred) && captureModeOn)
{
return;
}

// We assume all drawn bones are inferred unless BOTH joints are tracked
Pen drawPen = this.inferredBonePen;
if ((joint0.TrackingState == JointTrackingState.Tracked && joint1.Trackin
JointTrackingState.Tracked) || comparingModeOn)
{
drawPen = this.trackedBonePen;
}

if (comparingModeOn && plotInSeatedMode)
{
if ((jointType0 != JointType.HipCenter) && (jointType1 != JointType.HipC
(jointType0 != JointType.Spine) && (jointType1 != JointType.Spine) &&
(jointType0 != JointType.HipLeft) && (jointType1 != JointType.HipLeft) &&
(jointType0 != JointType.KneeLeft) && (jointType1 != JointType.KneeLeft)
(jointType0 != JointType.AnkleLeft) && (jointType1 != JointType.AnkleLeft
(jointType0 != JointType.FootLeft) && (jointType1 != JointType.FootLeft)
(jointType0 != JointType.HipRight) && (jointType1 != JointType.HipRight)
(jointType0 != JointType.KneeRight) && (jointType1 != JointType.KneeRight
(jointType0 != JointType.AnkleRight) && (jointType1 != JointType.AnkleRig
(jointType0 != JointType.FootRight) && (jointType1 != JointType.FootRight
{
drawingContext.DrawLine(drawPen, this.SkeletonPointToScreen(joint0.Positi
}
}

else drawingContext.DrawLine(drawPen, this.SkeletonPointToScreen(joint0.P
}

/// <summary>
/// Handles the checking or unchecking of the seated mode combo box
```

```
/// </summary>
/// <param name="sender">object sending the event</param>
/// <param name="e">event arguments</param>
private void CheckBoxSeatedModeChanged(object sender, RoutedEventArgs e)
{
if (null != this.sensor)
{
if (this.checkBoxSeatedMode.IsChecked.GetValueOrDefault())
{
this.sensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Seated;
seatedModeOn = true;
}
else
{
this.sensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Default;
seatedModeOn = false;
}
}
}

//
//
// New methods:
//
//

/// <summary>
/// Load a skeleton object from a text file
/// </summary>
/// <param name="dataLines"></param>
private void loadSkelDataFromText (string[] dataLines, bool isModel)
{

if (dataLines[0].Split(null)[4] == "SeatedModeOn") plotInSeatedMode = tru
photo was taken in seated mode
else if (dataLines[0].Split(null)[4] == "SeatedModeOff") plotInSeatedMode

foreach (string line in dataLines)
{
```

```
// creating a new point (holding coordinates)
var newPoint = new SkeletonPoint
{
X = Convert.ToSingle(line.Split(null)[1]),
Y = Convert.ToSingle(line.Split(null)[2]),
Z = Convert.ToSingle(line.Split(null)[3]),

};

// checking the type of the current joint (current line)
if (line.Split(null)[0] == "HipCenter")
{

if (isModel) // checking the type of the data
{
var newJoint = loadModelSkel.Joints[JointType.HipCenter]; // creating new
newJoint.Position = newPoint; // setting the joint's position
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.HipCenter] = newJoint; // updating the ske
}

else // the same is done if the joint is not a "model" type, as shown bel
{
var newJoint = loadPhotoSkel.Joints[JointType.HipCenter];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.HipCenter] = newJoint;
}
}

// the same is done for all the other joints

else if (line.Split(null)[0] == "Spine")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.Spine];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
```

```
loadModelSkel.Joints[JointType.Spine] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.Spine];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.Spine] = newJoint;
}
}

else if (line.Split(null)[0] == "ShoulderCenter")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.ShoulderCenter];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.ShoulderCenter] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.ShoulderCenter];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.ShoulderCenter] = newJoint;
}
}

else if (line.Split(null)[0] == "Head")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.Head];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.Head] = newJoint;
```

```
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.Head];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.Head] = newJoint;
}
}

else if (line.Split(null)[0] == "ShoulderLeft")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.ShoulderLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.ShoulderLeft] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.ShoulderLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.ShoulderLeft] = newJoint;
}
}

else if (line.Split(null)[0] == "ElbowLeft")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.ElbowLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.ElbowLeft] = newJoint;
}
```

```
else
{
var newJoint = loadPhotoSkel.Joints[JointType.ElbowLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.ElbowLeft] = newJoint;
}
}

else if (line.Split(null)[0] == "WristLeft")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.WristLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.WristLeft] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.WristLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.WristLeft] = newJoint;
}
}

else if (line.Split(null)[0] == "HandLeft")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.HandLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.HandLeft] = newJoint;
}
```

```
else
{
var newJoint = loadPhotoSkel.Joints[JointType.HandLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.HandLeft] = newJoint;
}
}

else if (line.Split(null)[0] == "ShoulderRight")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.ShoulderRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.ShoulderRight] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.ShoulderRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.ShoulderRight] = newJoint;
}
}

else if (line.Split(null)[0] == "ElbowRight")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.ElbowRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.ElbowRight] = newJoint;
}

else
```

```
{
var newJoint = loadPhotoSkel.Joints[JointType.ElbowRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.ElbowRight] = newJoint;
}
}

else if (line.Split(null)[0] == "WristRight")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.WristRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.WristRight] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.WristRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.WristRight] = newJoint;
}
}

else if (line.Split(null)[0] == "HandRight")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.HandRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.HandRight] = newJoint;
}

else
{
```

```
var newJoint = loadPhotoSkel.Joints[JointType.HandRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.HandRight] = newJoint;
}
}

else if (line.Split(null)[0] == "HipLeft")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.HipLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.HipLeft] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.HipLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.HipLeft] = newJoint;
}
}

else if (line.Split(null)[0] == "KneeLeft")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.KneeLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.KneeLeft] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.KneeLeft];
```

```
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.KneeLeft] = newJoint;
}
}

else if (line.Split(null)[0] == "AnkleLeft")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.AnkleLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.AnkleLeft] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.AnkleLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.AnkleLeft] = newJoint;
}
}

else if (line.Split(null)[0] == "FootLeft")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.FootLeft];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.FootLeft] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.FootLeft];
newJoint.Position = newPoint;
```

```
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.FootLeft] = newJoint;
}
}

else if (line.Split(null)[0] == "HipRight")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.HipRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.HipRight] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.HipRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.HipRight] = newJoint;
}
}

else if (line.Split(null)[0] == "KneeRight")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.KneeRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.KneeRight] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.KneeRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
```

```
loadPhotoSkel.Joints[JointType.KneeRight] = newJoint;
}
}

else if (line.Split(null)[0] == "AnkleRight")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.AnkleRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.AnkleRight] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.AnkleRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.AnkleRight] = newJoint;
}
}

else if (line.Split(null)[0] == "FootRight")
{
if (isModel)
{
var newJoint = loadModelSkel.Joints[JointType.FootRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadModelSkel.Joints[JointType.FootRight] = newJoint;
}

else
{
var newJoint = loadPhotoSkel.Joints[JointType.FootRight];
newJoint.Position = newPoint;
newJoint.TrackingState = JointTrackingState.Tracked;
loadPhotoSkel.Joints[JointType.FootRight] = newJoint;
```

```
}
}
}
}

//
//
// New handlers:
//
//

// Buttons:

/// <summary>
/// Checks if the "Photo" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void photoButton_Click(object sender, RoutedEventArgs e)
{
skelPhoto = true;
photoScreenLock = true;
skelsData.Clear();

saveButton.IsEnabled = true;
}

/// <summary>
/// Checks if the "Save" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void saveButton_Click(object sender, RoutedEventArgs e)
{
skelSave = true;
startingPhotoModeOn = false;

// controle de botoes:
saveButton.IsEnabled = false;
```

```
startingPhotoButton.IsEnabled = false;
leftShiftUser.IsEnabled = false;
rightShiftUser.IsEnabled = false;
}

/// <summary>
/// Checks if the "Load Model Photo" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void loadModelButton_Click(object sender, RoutedEventArgs e)
{
loadModelPhoto = true;
if (loadPhoto) compareButton.IsEnabled = true;
}

/// <summary>
/// Checks if the "Load Photo" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void loadPhotoButton_Click(object sender, RoutedEventArgs e)
{
loadPhoto = true;
if (loadModelPhoto) compareButton.IsEnabled = true;
}

/// <summary>
/// Checks if the "REC" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void recButton_Click(object sender, RoutedEventArgs e)
{
photoScreenLock = false;

// inicio da gravacao:
if (!recOn)
{
```

```
skelsData.Clear(); // limpando skelsData
sinalREC.IsChecked = true;
recOn = true;
startingPhotoModeOn = false;

//controle dos botoes:
radioButtonCaptureMode.IsEnabled = false;
radioButtonComparingMode.IsEnabled = false;
radioButtonPhoto.IsEnabled = false;
radioButtonVideo.IsEnabled = false;
startingPhotoButton.IsEnabled = false;
leftShiftUser.IsEnabled = false;
rightShiftUser.IsEnabled = false;
}

// fim da gravacao:
else if (recOn)
{

sinalREC.IsChecked = false;
recOn = false;
startingPhotoModeOn = true;

//controle dos botoes:
radioButtonCaptureMode.IsEnabled = true;
radioButtonComparingMode.IsEnabled = true;
radioButtonPhoto.IsEnabled = true;
radioButtonVideo.IsEnabled = true;
saveButton.IsEnabled = true;
startingPhotoButton.IsEnabled = true;
leftShiftUser.IsEnabled = true;
rightShiftUser.IsEnabled = true;


}
}

/// <summary>
/// Checks if the "Load Video" button was pressed
```

```
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void loadVideoButton_Click(object sender, RoutedEventArgs e)
{
loadVideo = true;
videoLengthCountUser = (Directory.GetFiles(videoCorePathUser, "*").Length

// controle de botoes:
leftShiftUser.IsEnabled = true;
rightShiftUser.IsEnabled = true;
startLoopUser.IsEnabled = true;
syncButton.IsEnabled = true;
if (loadModelVideo) compareButton.IsEnabled = true;
loopMinusButton.IsEnabled = true;
}

/// <summary>
/// Checks if the "Shift (L)" button (left screen) was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void LeftShiftModel_Click(object sender, RoutedEventArgs e)
{
shiftValueModel--;
}

/// <summary>
/// Checks if the "Shift (R)" button (left screen) was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void RightShiftModel_Click(object sender, RoutedEventArgs e)
{
shiftValueModel++;
}

/// <summary>
/// Checks if the "Start Loop" button (left screen) was pressed
```

```
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void StartLoopModel_Click(object sender, RoutedEventArgs e)
{
loopOnModel = true;

//controle dos botoes:
startLoopModel.IsEnabled = false;
stopLoopModel.IsEnabled = true;

radioButtonCaptureMode.IsEnabled = false;
radioButtonComparingMode.IsEnabled = false;
radioButtonPhoto.IsEnabled = false;
radioButtonVideo.IsEnabled = false;

leftShiftModel.IsEnabled = false;
rightShiftModel.IsEnabled = false;
loadModelVideoButton.IsEnabled = false;
}

/// <summary>
/// Checks if the "Stop Loop" button (left screen) was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void StopLoopModel_Click(object sender, RoutedEventArgs e)
{
loopOnModel = false;

//controle dos botoes:
startLoopModel.IsEnabled = true;
stopLoopModel.IsEnabled = false;

if (stopLoopUser.IsEnabled == false)
{
radioButtonCaptureMode.IsEnabled = true;
radioButtonComparingMode.IsEnabled = true;
radioButtonPhoto.IsEnabled = true;
```

```
radioButtonVideo.IsEnabled = true;
}

leftShiftModel.IsEnabled = true;
rightShiftModel.IsEnabled = true;
loadModelVideoButton.IsEnabled = true;
}

/// <summary>
/// Checks if the "Shift (L)" button (right screen) was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void LeftShiftUser_Click(object sender, RoutedEventArgs e)
{
shiftValueUser--;
}

/// <summary>
/// Checks if the "Shift (R)" button (right screen) was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void RightShiftUser_Click(object sender, RoutedEventArgs e)
{
shiftValueUser++;
}

/// <summary>
/// Checks if the "Start Loop" button (right screen) was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void StartLoopUser_Click(object sender, RoutedEventArgs e)
{
loopOnUser = true;

//controle dos botoes:
startLoopUser.IsEnabled = false;
```

```
stopLoopUser.IsEnabled = true;

radioButtonCaptureMode.IsEnabled = false;
radioButtonComparingMode.IsEnabled = false;
radioButtonPhoto.IsEnabled = false;
radioButtonVideo.IsEnabled = false;

leftShiftUser.IsEnabled = false;
rightShiftUser.IsEnabled = false;
loadVideoButton.IsEnabled = false;
}

/// <summary>
/// Checks if the "Stop Loop" button (right screen) was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void StopLoopUser_Click(object sender, RoutedEventArgs e)
{
loopOnUser = false;

//controle dos botoes:
startLoopUser.IsEnabled = true;
stopLoopUser.IsEnabled = false;

if (stopLoopModel.IsEnabled == false)
{
radioButtonCaptureMode.IsEnabled = true;
radioButtonComparingMode.IsEnabled = true;
radioButtonPhoto.IsEnabled = true;
radioButtonVideo.IsEnabled = true;
}

leftShiftUser.IsEnabled = true;
rightShiftUser.IsEnabled = true;
loadVideoButton.IsEnabled = true;
}

/// <summary>
```

```
/// Checks if the "Sync" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void SyncButton_Click(object sender, RoutedEventArgs e)
{
shiftValueModel = 0;
shiftValueUser = 0;
}

/// <summary>
/// Checks if the "Loop +" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void loopPlusButton_Click(object sender, RoutedEventArgs e)
{
loopDelaySetup--;
if (loopDelaySetup <= 0) loopPlusButton.IsEnabled = false;
}

/// <summary>
/// Checks if the "Loop -" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void loopMinusButton_Click(object sender, RoutedEventArgs e)
{
loopDelaySetup++;
if (loopDelaySetup > 0) loopPlusButton.IsEnabled = true;
}

/// <summary>
/// Checks if the "Load Model Video" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void loadModelVideoButton_Click(object sender, RoutedEventArgs e)
{
```

```
loadModelVideo = true;
videoLengthCountModel = (Directory.GetFiles(videoCorePathModel).Length) -

// controle de botoes:
leftShiftModel.IsEnabled = true;
rightShiftModel.IsEnabled = true;
startLoopModel.IsEnabled = true;
syncButton.IsEnabled = true;
if (loadVideo) compareButton.IsEnabled = true;
loopMinusButton.IsEnabled = true;
}

/// <summary>
/// Checks if the "Starting Photo" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void startingPhotoButton_Click(object sender, RoutedEventArgs e)
{
startingPhotoSelected = true;
}

/// <summary>
/// Checks if the "Compare" button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void compareButton_Click(object sender, RoutedEventArgs e)
{
List<Pose> compareListUser = new List<Pose>();
List<Pose> compareListModel = new List<Pose>();
double finalScore = 0;
string finalScoreString;

if (this.radioButtonPhoto.IsChecked.GetValueOrDefault()) // if comparing
{
// user photo
KHMESDK1_8.Skeleton compareSkeletonUser = new KHMESDK1_8.Skeleton(loadPho
Pose comparePoseUser = new Pose(compareSkeletonUser);
```

```
// model photo
KHMESDK1_8.Skeleton compareSkeletonModel = new KHMESDK1_8.Skeleton(loadM
Pose comparePoseModel = new Pose(compareSkeletonModel);

// final score
finalScore = Evaluation.Spheres(comparePoseModel, comparePoseUser,
radiusPose, minScorePose);

foreach (Joint joint in loadPhotoSkel.Joints)
{
Console.WriteLine("Joint Type = " + joint.JointType +
"Joint Position X, Y, Z = " + joint.Position.X + joint.Position.Y +
joint.Position.Z);
}
}

else if (this.radioButtonVideo.IsChecked.GetValueOrDefault()) // else
if comparing videos
{
for (int j = 0; j < videoLengthCountUser; j++) // user
{
// reading movement data
string[] photoLines = File.ReadAllLines(Path.Combine(videoCorePathUser,
videoCoreNameUser + j + ".txt"));
loadSkelDataFromText(photoLines, false);

// creating movement list
KHMESDK1_8.Skeleton compareSkeletonUser =
new KHMESDK1_8.Skeleton(loadPhotoSkel);
Pose comparePoseUser = new Pose(compareSkeletonUser);
compareListUser.Add(comparePoseUser);

}

for (int j = 0; j < videoLengthCountModel; j++) // model
{
// reading movement data
```

```
string[] modelLines = File.ReadAllLines(Path.Combine(videoCorePathModel,
videoCoreNameModel
+ j + ".txt"));

loadSkelDataFromText(modelLines, true);

// creating movement list
KHMESDK1_8.Skeleton compareSkeletonModel =
new KHMESDK1_8.Skeleton(loadModelSkel);
Pose comparePoseModel = new Pose(compareSkeletonModel);
compareListModel.Add(comparePoseModel);

}

// creating the movements
Movement compareMovementUser = new Movement(compareListUser);
Movement compareMovementModel = new Movement(compareListModel);

// final score
finalScore = Evaluation.Spheres(compareMovementModel, compareMovementUser
radiusMovement, radiusStep, minScoreMovement, scoreStep);

}

// plotting results
finalScoreString = finalScore.ToString();
File.WriteAllText(Path.Combine(compareResultPath, compareResultName + ".t
finalScoreString);
textBox.Clear();
textBox.Text = finalScoreString;

}

// Radio Buttons:

/// <summary>
/// Checks changes on the "Capture Mode" radio button state
/// </summary>
```

```
/// <param name="sender"></param>
/// <param name="e"></param>
private void RadioButtonCaptureModeChanged(object sender,
RoutedEventArgs e)
{
if (this.radioButtonCaptureMode.IsChecked.GetValueOrDefault())
{
captureModeOn = true;
loadModelPhoto = false;
loadPhoto = false;
loadVideo = false;
loadModelVideo = false;
startingPhotoModeOn = false;
loopDelaySetup = 0;
loopDelayModel = 0;
loopDelayUser = 0;
}

else
{
captureModeOn = false;
}
}

/// <summary>
/// Checks changes on the "Comparing Mode" radio button state
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void RadioButtonComparingModeChanged(object sender,
RoutedEventArgs e)
{
compareButton.IsEnabled = false;

if (this.radioButtonComparingMode.IsChecked.GetValueOrDefault())
{
comparingModeOn = true;
photoScreenLock = false;
```

```
// controle de botoes:
saveButton.IsEnabled = false;
startingPhotoButton.IsEnabled = false;
photoButton.IsEnabled = false;
recButton.IsEnabled = false;
leftShiftUser.IsEnabled = false;
rightShiftUser.IsEnabled = false;

if (radioButtonPhoto.IsChecked == true)
{
loadModelButton.IsEnabled = true;
loadPhotoButton.IsEnabled = true;
}

else if (radioButtonVideo.IsChecked == true)
{
loadModelVideoButton.IsEnabled = true;
loadVideoButton.IsEnabled = true;
}
}

else
{
comparingModeOn = false;

// controle de botoes:
if (radioButtonPhoto.IsChecked == true) photoButton.IsEnabled = true;
else if (radioButtonVideo.IsChecked == true) recButton.IsEnabled = true;
loadModelButton.IsEnabled = false;
loadPhotoButton.IsEnabled = false;
loadModelVideoButton.IsEnabled = false;
loadVideoButton.IsEnabled = false;
leftShiftUser.IsEnabled = false;
rightShiftUser.IsEnabled = false;
startLoopUser.IsEnabled = false;
leftShiftModel.IsEnabled = false;
rightShiftModel.IsEnabled = false;
startLoopModel.IsEnabled = false;
syncButton.IsEnabled = false;
```

```
loopMinusButton.IsEnabled = false;
loopPlusButton.IsEnabled = false;
}
}

/// <summary>
/// Checks changes on the "Photo" radio button state
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void RadioButtonPhotoChanged(object sender, RoutedEventArgs e)
{
startingPhotoModeOn = false;

if (this.radioButtonPhoto.IsChecked.GetValueOrDefault())
{
loadVideo = false;
loadModelVideo = false;
loopDelaySetup = 0;
loopDelayModel = 0;
loopDelayUser = 0;
}

else
{
loadModelPhoto = false;
loadPhoto = false;
}
}

/// <summary>
/// Checks changes on the "Video" radio button state
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void RadioButtonVideoChanged(object sender, RoutedEventArgs e)
{
saveButton.IsEnabled = false;
startingPhotoButton.IsEnabled = false;
```

```
leftShiftUser.IsEnabled = false;
rightShiftUser.IsEnabled = false;
compareButton.IsEnabled = false;

if (this.radioButtonVideo.IsChecked.GetValueOrDefault())
{

// controle de botoes:
if (radioButtonCaptureMode.IsChecked == true) recButton.IsEnabled = true;

else if (radioButtonComparingMode.IsChecked == true)
{
loadVideoButton.IsEnabled = true;
loadModelVideoButton.IsEnabled = true;
}

photoButton.IsEnabled = false;
loadModelButton.IsEnabled = false;
loadPhotoButton.IsEnabled = false;
}

else
{
// controle de botoes:
if (radioButtonCaptureMode.IsChecked == true) photoButton.IsEnabled = tru

else if (radioButtonComparingMode.IsChecked == true)
{
loadModelButton.IsEnabled = true;
loadPhotoButton.IsEnabled = true;
}

loadModelVideoButton.IsEnabled = false;
loadVideoButton.IsEnabled = false;
recButton.IsEnabled = false;
leftShiftUser.IsEnabled = false;
rightShiftUser.IsEnabled = false;
startLoopUser.IsEnabled = false;
leftShiftModel.IsEnabled = false;
```