DM

Definição de Funções Primitivas em *Behavior Markup Language*

DISSERTAÇÃO DE MESTRADO

Mariana Patrícia Barros Teixeira
MESTRADO EM MATEMÁTICA

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

dezembro | 2016

Definição de Funções Primitivas em *Behavior Markup Language*

DISSERTAÇÃO DE MESTRADO

Mariana Patrícia Barros Teixeira
MESTRADO EM MATEMÁTICA

ORIENTADOR
Eduardo Leopoldo Fermé

" My advice to other disabled people would be, concentrate on things your disability doesn't prevent you doing well, and don't regret the things it interferes with. Don't be disabled in spirit as well as physically. "

Stephen Hawking

Agradecimentos

É com muita satisfação que termino esta etapa da minha vida. Foi um caminho longo, com alguns obstáculos, mas, com muito trabalho, determinação e persistência, e também graças à preciosa ajuda de algumas pessoas que não hesitaram em apoiar-me e em apontar-me alguns erros cometidos, foi possível, finalmente, vislumbrar a meta.

Gostaria de começar por agradecer ao meu orientador, Professor Doutor Eduardo Fermé, que sempre me estimulou no aprofundamento das minhas pesquisas e me mostrou o caminho a seguir, principalmente nos momentos de maior aflição. Não foi, porém, o único, pelo que endereço um sincero obrigada ao Professor Doutor Sergi Bermúdez i Badia que me proporcionou uma ajuda essencial na parte técnica desta dissertação.

Em segundo lugar, estou muito grata aos meus pais que me ajudaram sempre, por tudo o que me proporcionaram para poder alcançar este objetivo tão ambicioso.

Logo, a uma amiga de sempre, pela paciência e apoio incondicional, por todas as horas que despendeu na leitura desta dissertação, algumas vezes mesmo sem perceber os seus conteúdos, mas, mesmo assim, dando sempre dicas para melhorar a minha expressão escrita.

Por fim, agradeço a todos os meus professores e colegas da Universidade da Madeira que me acompanharam na minha vida académica e que contribuíram também para a minha formação como pessoa que sou hoje e a que poderei vir a melhorar no futuro.

Resumo

O principal fundamento desta dissertação centra-se num estudo sobre a incorporação de agentes virtuais, com deficiências motoras, num determinado ambiente, utilizando a linguagem *Behavior Markup Language*. Numa primeira fase, efetuámos a seleção de algumas linguagens que se adequavam melhor ao objetivo pretendido e, em simultâneo, estudámos os ambientes em que se podiam utilizar. Em seguida, tornou-se necessário averiguar quais seriam as deficiências que se adequavam melhor a essa incorporação. Logo, adotámos algumas estratégias utilizando funções já existentes no *software Smartbody*. Para finalizar, foi elaborada uma análise de todos os prós e contras das estratégias abordadas, onde o foco foi sempre colocado em deficiências motoras reais.

Palavras-chave: Ciências da Computação, Linguagens formais, *Behavior Markup Language*, Agentes virtuais, Deficiências motoras.

Abstract

The main goal of this dissertation was based on a study about the incorporation of virtual agents with motor disabilities in a given environment using the Behavior Markup Language. Initially, we selected some languages that were better suited for the intended purpose and, simultaneously, we studied the environments in which they could be used. Then it became necessary to find out which were the disabilities that best suited this incorporation. Therefore, we have adopted some strategies using functions that already exist in the Smartbody software. Finally, it was elaborated an analysis of all the pros and cons of the strategies discussed, where the focus was always placed on realistic motor disabilities.

Key words: Computer Science, Formal Languages, Behavior Markup Language, Virtual agents, Disabilities.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Na atualidade, os seres humanos em geral travam uma luta constante pela materialização do princípio da igualdade. Para isso é necessário compreender todos os comportamentos existentes. Um passo importante nesta luta é a plena integração, na sociedade, de indivíduos com deficiências motoras. Neste âmbito, a simulação da interação de indivíduos com incapacidades físicas em ambientes urbanos permite avaliar as adaptações e modificações que estes espaços urbanos devem sofrer para integrar corretamente estes indivíduos. Como todos nós temos conhecimento, a matemática é uma área imensa e vasta, cujo estudo abrange inúmeras componentes envolventes, como a análise, a verificação através de testes, o estabelecimento de paralelismos, a comprovação de resultados, tendo-se sempre em linha de conta os resultados que queremos obter, ou aqueles que obtemos, dos quais nem sempre estávamos à espera. Adaptamo-nos paulatinamente aos resultados como se comprova ao longo desta dissertação. Houve, assim, uma constante modelação ao rumo a seguir para poder chegar àquilo que se pretendia, dado que, do mesmo modo que para obtermos um triângulo precisamos de um conjunto de retas para formar arestas, que terminam em vértices, quais pontos de ligação, mostraremos que, ao longo da minha exposição, existem diversos conjuntos de elementos de ligação que são necessários para obtermos uma analise mais pormenorizada, mais completa na medida em que ser matemático é ser completo.

A simulação de um corpo humano é feita através de animações num esqueleto virtual. Existem diversos métodos de animação tais como: captura de movimentos reais, sequências de *frames* que transmitam uma sensação de movimento, movimentação com um destino definido à priori, controlo do movimento através de parâmetros relativos à orientação bem como à geometria do esqueleto, entre outros. Neste contexto, foram estudadas algumas

estratégias para implementar incapacidades num esqueleto virtual. Isso implicou ter algum conhecimento da biomecânica do corpo humano pois um esqueleto virtual tem que ter uma estrutura articulada e organizada hierarquicamente para poder funcionar corretamente. Desta estrutura faz parte um conjunto de segmentos, ligados por articulações, que têm uma determinada posição e limites. Em síntese, os objetivos principais desta dissertação são: Utilizar, na linguagem BML (*Behavior Markup Language*), modificadores e funções já existentes que permitam descrever as diferentes incapacidades, de forma a permitir modelar facilmente indivíduos com diferentes graus de dificuldade (indivíduos que coxeiam, paralíticos, entre outros). E, ainda, incorporar e simular, em ambientes virtuais, indivíduos com incapacidades, a partir da descrição em BML.

Esta análise está estruturada em 6 capítulos. Numa pequena abordagem, no 1° capítulo, deparamo-nos com o início do contexto e os objetivos. No 2° são descritos conceitos básicos de linguagens formais e de linguagens comportamentais. No 3° são introduzidas algumas linguagens formais baseadas em XML. No 4° iniciamos algumas definições de agentes e ambientes virtuais e, ainda, é exemplificada a plataforma *Smartbody*. No penúltimo estão definidas algumas deficiências motoras e também a implementação de estratégias a utilizar para minorar os seus efeitos. E, finalmente, no último capítulo, surge a conclusão na qual se incluem algumas sugestões para possíveis trabalhos futuros.

A partir dos conteúdos das unidades curriculares Otimização e Complementos de Lógica, lecionados durante o meu Mestrado, tivemos a pretensão de aprofundar conteúdos cujo conhecimento desencadeará, estamos certos, vontades de estudo efetivo mais alargado numa investigação futura. Juntar estes complementos à minha formação académica de Licenciatura em Engenharia Informática, no âmbito das unidades curriculares de Lógica Computacional, Métodos Numéricos e Teoria das Linguagens e Compiladores, acabou por ser uma mais valia inegável pelo facto de concentrar todas as matérias.

A simbiose equilibrada das diferentes aprendizagens nas unidades curriculares acabou por ser um aspeto vantajoso nas análises, nas definições das estratégias, nos testes, na programação, onde podemos verificar que é possível também aqui ter uma espécie de visão holística pelo facto de podermos detetar interligações evidentes. E, com as aprendizagens adquiridas, fomos conduzidos a novas experiências, a novas descobertas, a novos conhecimentos. A escolha baseou-se muito na observação da diferença de cada comportamento humano, das dificuldades diárias bem como na vontade de querer compreender uma vida diferente daquela que se designa na gíria como 'normal'.

Após o acompanhamento de alguns casos que se enquadram na vivência

destes seres humanos com dificuldades motoras, ficámos a conhecer a índole das lutas diárias com que eles se deparam. E, dado que o mundo, aos olhos de um matemático, baseia-se em números, em lógicas, em interpretações, em adaptações, em desafios, em vontade de ir mais além, foi deste modo que abordámos estes conteúdos nesta dissertação. Pretendemos abordar as diversas restrições de que são alvo no seu quotidiano todos aqueles que possuem algum tipo de incapacidade motora.

Como forma de sintetizar todo o processo, segue-se um esquema da dissertação.

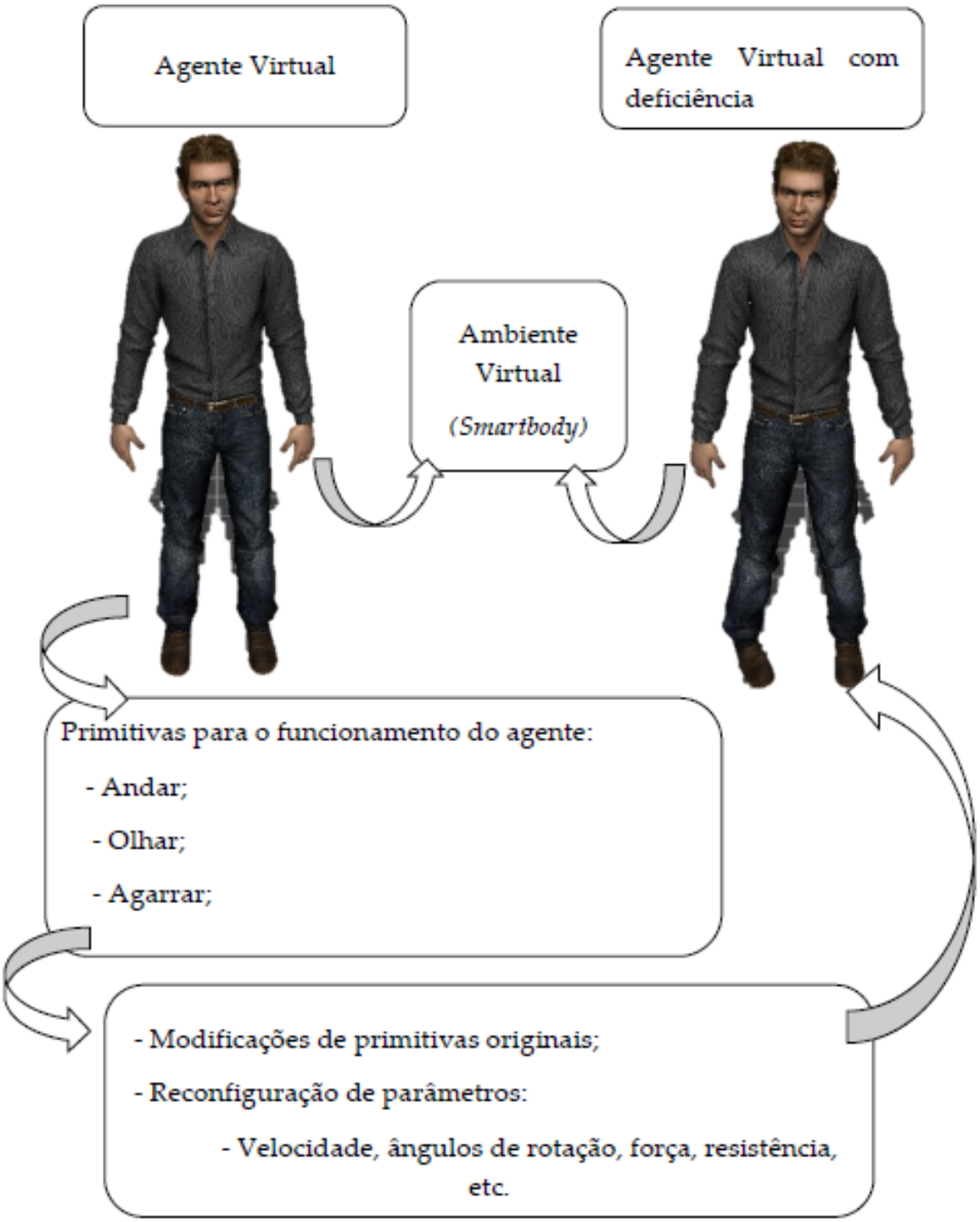


Figura 1.1: Esquema da dissertação.

Capítulo 2

Teoria das linguagens formais

2.1 Introdução

Neste capítulo começámos por introduzir alguns conceitos básicos e notações da teoria de linguagens formais.

Com efeito, nos dias de hoje, para podermos interagir com o próximo, utilizamos uma determinada linguagem comum, por forma a tornar possível o entendimento entre as partes. Assim, o mundo está repleto de inúmeras linguagens, todas diferentes, mas com um traço em comum, o ato de comunicar. E, neste caso específico, verifica-se uma interação entre o 'eu', utilizador, e o computador, que é o recetor. Esta linguagem materializa-se, pois, através de instruções dadas pelo utilizador que serão interpretadas pelo recetor.

As linguagens formais têm, indiscutivelmente, um papel essencial nas ciências da computação. Antes do aparecimento da tecnologia e eletrónica dos computadores, já havia grandes desenvolvimentos de notações formais de matemática, demonstrações, lógica e outras ciências relacionadas. Todas estas notações formais foram evoluindo, de forma a serem selecionados padrões que melhor correspondiam às necessidades dos utilizadores. A revolução dos computadores foi feita a partir destas linguagens que se passaram posteriormente a chamar linguagens artificiais [11].

2.2 Linguagem

Uma linguagem formal, L, é normalmente definida por um conjunto, finito ou infinito, de palavras ou símbolos - um alfabeto, normalmente designado pela letra grega $\sum$. Os conjuntos podem ser constituídos por números, letras, símbolos, entre outros, e estes ainda têm de ser separados por um carater, podendo tomar os seguintes aspetos:

$\{0,1,2\}, \{a,b,c\}, \{\beta,\mu,\Omega\}, \{0,1,+,-,*,/,\%\}$.

O conjunto também pode ser vazio, sem elementos, e pode ser representado por $\emptyset$. Para identificar o conjunto sobre todas as palavras de um alfabeto utilizamos a notação $\sum *$. A notação $|x|$ aplica-se ao comprimento de elementos de um conjunto x, ou seja, representa quantos elementos fazem partem do mesmo. Posto isto, seguem-se vários exemplos para uma melhor compreensão desta linguagem.

Exemplo 2.1 *Sendo o alfabeto* $\sum = \{0,4\}$ *e* x *um conjunto.*
Seguem-se os conjuntos possíveis de construir:
$x = \emptyset; x = 0; x = 4; x = 04; x = 40; x = 004$.

Exemplo 2.2 *Sendo* x *um conjunto e* $|x|$ *o comprimento do mesmo.*
$x = \emptyset, |x| = 0$;
$x = 0, |x| = 1$;
$x = 04, |x| = 2$;
$x = 004, |x| = 3$.

As linguagens formais têm duas caraterísticas essenciais:

- A sintaxe tem de estar muito bem definida, pois é graças às regras que regem que será possível saber se um conjunto de cadeias pertence ou não à linguagem;
- A semântica não deverá conter ambiguidades.

Exemplo 2.3 *Dada uma linguagem* L *sobre o alfabeto* $\sum = \{1,2,3,4,*,=\}$ *e as seguintes regras:*

1. *Cada cadeia diferente de vazio que não contenha* $'*'$ *ou* $' ='$ *pertence a* L.
2. *Uma sequência que contenha* $' ='$ *pertence a* L *se, e só se, houver apenas um* $' ='$ *e que este separe duas cadeias válidas de* L.
3. *Uma sequência que contenha* $'*'$ *e que não contenha '=' pertence a* L*, se, e só se, todos os '*' separam duas cadeias válidas de* L.
4. *Nada mais é cadeia em* L.

Seguindo as regras, podemos afirmar que a cadeia $'23 * 1 = 44' \in L$. Já a cadeia $'12 = *' \notin L$. Estas regras apenas exigem o respeito ao aspeto, ou seja, exigem que se respeite a sintaxe. Em relação ao sentido, ou à semântica,

nada está especificado. Não existe nenhuma indicação de que $'1, 2, 3, 4'$ sejam números naturais nem que $'*'$ corresponda à multiplicação.

É também possível efetuar operações entre as linguagens, tais como: concatenação, união ($\cup$), interseção ($\cap$) e complemento ($\neg$). Sendo duas linguagens L_1 e L_2 sobre os alfabetos $\sum_1$ e $\sum_2$, respetivamente, as que se seguem também são linguagens:

- Concatenação: L_1L_2 é uma linguagem da forma ab, em que a é uma sequência de L_1 e b uma de L_2;
- União: $L_1 \cup L_2$ é uma linguagem sobre $\sum_1 \cup \sum_2$;
- Interseção: $L_1 \cap L_2$ é uma linguagem sobre $\sum_1 \cap \sum_2$ que contém todas as cadeias de L_1 e L_2;
- Complemento: $\neg L$ é uma linguagem que contém todos os elementos que L não contém.

2.3 Gramática

É através de uma gramática que se constrói uma linguagem. Uma gramática, G, é um quádruplo $G = (N, \sum, P, S)$ onde:

- N é um conjunto finito de *símbolos não terminais* (normalmente são representados por letras maiúsculas), carateres lexicais que não podem ser alterados na utilização das regras;
- $\sum$ é um conjunto finito de *símbolos terminais* (normalmente são representados por letras minúsculas), sendo este disjunto de N;
- P é o conjunto das produções. Cada produção é da forma:

$$(N \cup \sum)^* N (N \cup \sum)^* \rightarrow (N \cup \sum)^*,$$

ou da forma:

$$\alpha \rightarrow \beta,$$

significando que α deriva β ;

- S é o símbolo inicial de $G, S \in N$ e S é ainda um símbolo não terminal.

Note-se que o fecho de Kleene $\sum^*$ de um conjunto $\sum$ é definido indutivamente por:

- $\epsilon \in \sum^*$;
- $a.b \in \sum^*$ se $a \in \sum$ e$b \in \sum^*$.

2.3.1 Esquema fraseológico

Tendo em conta que uma gramática G é denotada por $G = (N, \sum, P, S)$, um conjunto de esquemas fraseológicos é definido por indução.

- S, símbolo inicial, é esquema fraseológico de G;
- $\alpha\beta\lambda$ é esquema fraseológico de G se $\alpha\beta\lambda$ é esquema fraseológico de G e $\beta \to \lambda$ é produção de G.

2.3.2 Frase

Uma frase b gerada pela gramática G é um esquema fraseológico constituído apenas por símbolos não terminais de G, ou seja, $b \in \sum^*$. Uma linguagem é o conjunto das frases de uma gramática G e tem como notação $L(G)$.

2.3.3 Derivação

Sendo $\alpha, \beta \in (N \cup \sum)^*$ e $k > 0$, o número de passos numa gramática G de β a partir de α, ou seja, $\alpha \Rightarrow^k \beta$. Uma derivação da gramática $G = (N, \sum, P, S)$ é também definida por indução.

- $\alpha \Rightarrow^0 \alpha$ (0 passos);
- $\alpha \Rightarrow^{k+1} \beta_1\beta_2\beta_3$ se $\alpha \Rightarrow^k \beta_1\alpha_2\beta_3$ e $\alpha_2 \to \beta_2 \in P$.

2.4 Classificação de Gramáticas

Uma gramática $G = (N, \sum, P, S)$ pode ser: *linear à direita (regular)* quando todas as produções são da forma $B \to bC$, onde $B, C \in N$ e $b \in \sum^*$ (0 ou mais símbolos terminais ou 1 símbolo não terminal no fim) ou $B \to b$, onde $B \in N$ e $b \in \sum^*$ (0 ou mais símbolos terminais).

Uma gramática ainda poderá ser classificada por ser *Independente do Contexto (algébrica)* se for composta por produções da forma $B \to \beta$, onde $B \in N$ e $\beta \in (N \cup \sum)^*$.

Segue-se um exemplo de aplicação dos conceitos introduzidos anteriormente.

Exemplo 2.4 *Seja* $G = (\{S, A, B\}, \{a, b\}, P, S)$ *e* $P = \{S \to aSa|A|B,$ $A \to \epsilon|aS,$ $B \to \epsilon|bS\}$ *vamos indicar os passos para chegar à frase ababa. Em primeiro lugar, é necessário identificar a gramática. Como as produções são da forma* $B \to \beta$ *, estamos perante uma gramática independente do contexto.*

Em seguida, começamos a demonstração pelo símbolo inicial.

$S \Rightarrow aSa \Rightarrow aBa \Rightarrow abSa \Rightarrow abAa \Rightarrow abaSa \Rightarrow abaBa \Rightarrow ababSa \Rightarrow ababBa \Rightarrow ababa$

A frase obtida apenas envolve símbolos não terminais e também pode ser designada por esquema fraseológico, que é conseguido através dos esquemas anteriores tendo por ponto de partida o símbolo inicial. Podemos ainda dizer que a linguagem gerada por esta gramática é o conjunto de todas as frases obtidas por S*.*

Em lógica, depois das regras da linguagem estarem definidas, é necessário atribui-lhes um significado, uma interpretação. Normalmente, a interpretação identifica se é verdadeiro ou falso. Segue-se uma explicação mais informal de todo o processo de compilação e interpretação de uma linguagem. Um compilador interpreta a linguagem com auxílio de dois componentes: um analisador, que identifica os símbolos da gramática, e um *parser* que decide se o código é ou não válido, ou, por outras palavras, decide se o código pertence à linguagem para que o compilador foi construído. O analisador gera ainda o executável com o código máquina (código binário) do respetivo computador. Este processo difere de máquina para máquina, pois tem em conta a capacidade do registo de cada processador. É necessário considerar alguns aspetos relacionados com o *parser*. Existem diversos modelos de computação que tratam da verificação e otimização do código.

Irá ser introduzido o conceito de autómato, começando pelos autómatos mais elementares (autómatos finitos e deterministas) até os mais gerais (Autómatos de Pilha).

2.5 Linguagens Regulares

Existem vários tipos de autómatos. Há autómatos com caraterísticas específicas para diferentes tipos de linguagens. Estes também podem ser utilizados no processamento das linguagens baseadas em XML, que irão ser abordadas mais à frente.

2.5.1 Autómato finito não determinista

Um autómato finito não determinista é um dos modelos mais simples de computação. Este modelo serve, essencialmente, para caraterizar as linguagens por ele aceites e também para simplificar algoritmos. É definido por um quíntuplo $M = (Q, \sum, \delta, q_0, F)$, onde:

- Q representa o conjunto finito de estados;
- $\sum$ é um conjunto finito de terminais, mais especificamente, símbolos de entrada;
- δ é uma função de transição direta da forma $\delta : Q \times \sum \rightarrow P(Q)$, em que $P(Q)$, significam as partes do conjunto Q, ou seja, todos os conjuntos de possível construção;
- O estado inicial é representado por q_0, e $q_0 \in Q$;
- O conjunto dos estados finais é representado por F, e $F \subseteq Q$.

Função de transição

Temos a informação de quantos estados existem, do estado inicial e dos símbolos de entrada a utilizar, temos, então, de definir a função de transição. Normalmente, esta função é representada através de diagramas de transição, onde os estados são representados por círculos, as transições por setas legendadas pelos símbolos. O estado inicial é identificado por uma seta e o estado final por dois círculos. Esta função generaliza-se a configurações. Uma configuração de autómato finito é um par $\{q, w\}$, onde $q \in Q$ e $w \in \sum^*$.Define-se, então, a função de transição por indução.

- $\delta^*(q, \epsilon) = \{q\}$;
- $\delta^*(q, wa) = \cup_{q' \in \delta^*(q,w)} \delta(q', a), a \in \sum$.

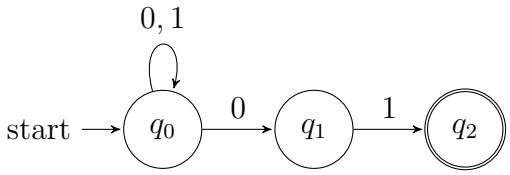


Figura 2.1: Exemplo de Autómato não determinista.

Exemplo 2.5 *Seja o autómato* $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, q_1)$ *representado pela Figura 2.1.*

As transições correspondentes serão:

$$\delta(q_0, 0) = \{q_0, q_1\}$$

$$\delta(q_0, 1) = \{q_0\}$$

$$\delta(q_1, 1) = \{q_2\}$$

Assim, temos como função de transição generalizada:

$$\delta^*(q_0, 101) = \cup_{q' \in \delta^*(q_0,10)} \delta(q^{'}, 1) = \cdots = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_2\}$$

$$\delta^*(q_0, 10) = \cup_{q' \in \delta^*(q_0,1)} \delta(q^{'}, 0) = \cdots = \{q_0, q_1\}$$

$$\delta^*(q_0, 1) = \cup_{q' \in \delta^*(q_0,\epsilon)} \delta(q^{'}, 1) = \{q_0\}$$

Se seguirmos a sequência 101 *vamos para os estados* q_0 *e* q_2*. Utilizando a função de transição generalizada a partir do estado inicial, chegaremos ao estado final.*

Relação de transição

Visto que os autómatos finitos não deterministas podem ter mais do que um estado inicial e transições a partir do mesmo estado com a mesma letra, o uso de uma função de transição não é suficiente. Pois tal implica que alguns estados possam ter que ser executados simultaneamente. Sendo um autómato finito $M = (Q, \sum, \delta, q_0, F)$, a relação de transição é representada por $\vdash_M^*$. Entre configurações a relação é definida por $(q, w) \vdash_M^k (q^{'}, w^{'})$ se existir $k \in \mathbb{N}_0$. A definição por indução será:

- $(q, w) \vdash_M^0 (q, w)$;
- $(q, aw) \vdash_M^{k+1} (q^{''}, w^{''})$ se $q' \in \delta(q, a)$ e $(q^{'}, w) \vdash_M^k (q^{''}, w^{''})$.

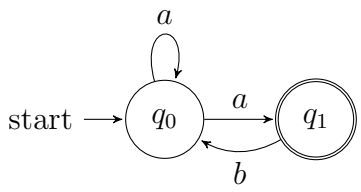


Figura 2.2: Exemplo de um Autómato finito não determinita.

Linguagem aceite

Dado um autómato finito $M = (Q, \sum, \delta, q_0, F)$, a linguagem aceite por este é dada por:

$$L(M) = \{w \in \sum^{*} : \delta^{*}(q_0, w) \vdash_M^{*} (q^{''}, \epsilon) e q^{''} \in F\}.$$

2.5.2 Autómato finito determinista

Um autómato finito diz-se determinista se para todo o estado e para todo símbolo de entrada existe, no máximo, uma transição. Ou seja, um autómato finito M é determinista se para todo o estado q e para todo o símbolo de entrada a de M:

$$\#(\delta(q, a)) \leq 1.$$

Um autómato finito diz-se completamente determinado se para todo estado q e para todo o símbolo de entrada a de M:

$$\#(\delta(q, a)) = 1.$$

Se tivermos um autómato finito é possível obter um autómato determinista e completamente determinado com a mesma linguagem.

Exemplo 2.6 *Vamos obter um autómato determinista e completamente determinado a partir do seguinte autómato.*

A partir do autómato representado na Figura 2.2, retiramos as funções de transição desde o estado inicial:

$$\delta^{'}(\{q_0\}, a) = \{q_0, q_1\}$$

$$\delta^{'}(\{q_0\}, b) = \emptyset$$

Em seguida, construímos os novos estados com os novos conjuntos que obtivemos.

$$\delta^{'}(\{q_0, q_1\}, a) = \{q_0, q_1\}$$

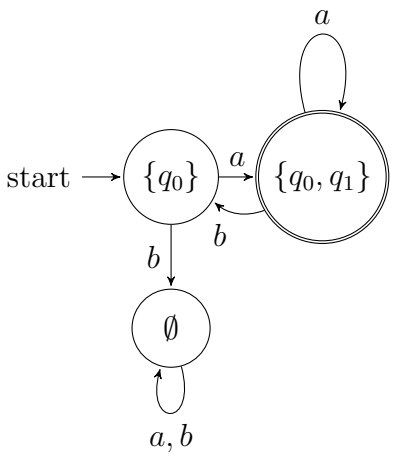


Figura 2.3: Transformação do Autómato da Figura 2.2 em Autómato determinista e completamente determinado.

$$\delta^{'}(\{q_0, q_1\}, b) = \{q_0\}$$
$$\delta^{'}(\emptyset, a) = \emptyset$$
$$\delta^{'}(\emptyset, b) = \emptyset$$

Este processo termina quando não existem mais estados novos. Neste caso, estamos em condições de construir o autómato determinista e completamente determinado. O novo estado final é o conjunto em que estiver q_1, estado final do autómato inicial. É de notar que não é necessário representar o estado vazio, $\emptyset$.

Assim, temos dois autómatos com a mesma linguagem (Figuras 2.2 e 2.3). Em ambos é possível construir as palavras: aba e aaa. Mas, para este procedimento ser possível, o autómato determinista completamente determinado tem de ser tal que:

- $Q^{'} = P(Q)$, *todos os estados têm que ser subconjuntos do autómato original;*
- $\sum^{'} = \sum$, *tem que ter os mesmos símbolos de entrada;*
- $\delta^{'}$ *é definida por* $\delta^{'}(S, a) = \cup_{s \in S} \delta(s, a)$, *onde* $S \in Q^{'}$ *e* $a \in \sum^{'}$;
- $q_0^{'} = \{q_0\}$, *o estado inicial é um conjunto que contém o estado inicial do autómato original;*
- $F^{'} = \{S \subseteq Q : S \cap F \neq \emptyset\}$, *tudo o que envolva o estado final do autómato original é estado final no novo autómato.*

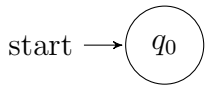


Figura 2.4: Exemplo de Autómato que aceita a linguagem $L = \emptyset$.

2.5.3 Conjunto Regular

Um conjunto regular de alfabeto $\sum$ é definido por indução como se segue:

- $\emptyset$ é um conjunto regular;
- $\{\epsilon\}$ é um conjunto regular;
- Para cada $a \in \sum$, $\{a\}$ é um conjunto regular;
- Se P e Q são conjuntos regulares, $P \cup Q, PQ$ e P^* também o são.

2.5.4 Expressão Regular

As expressões regulares de alfabeto $\sum$ que denotam os conjuntos regulares antes definidos, são definidas indutivamente:

- $\emptyset$ é uma expressão regular que denota o conjunto regular $\emptyset$;
- ϵ é uma expressão regular que denota o conjunto $\{\epsilon\}$;
- Para cada $a \in \sum$, a é uma expressão regular que denota o conjunto $\{a\}$;
- Se p e q são expressões regulares, $p+q$, pq e p^* também o são. Denotam os conjuntos $P \cup Q$, PQ e P^*, respetivamente, onde P e Q são os conjuntos regulares denotados por p e q, respetivamente.

2.5.5 Linguagens regulares e autómatos finitos

As linguagens regulares são aceites por autómatos finitos. Uma linguagem é aceite por um autómato quando existe pelo menos um autómato que a represente.

Vejamos os exemplos (7 até 12) que provam a frase anterior. Sendo L um conjunto regular.

Exemplo 2.7 *$L = \emptyset$ é a linguagem aceite pelo autómato $M = (\{q\}, \sum, \emptyset, q, \emptyset)$.*

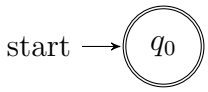


Figura 2.5: Exemplo de Autómato que aceita a linguagem $L = \{\epsilon\}$.

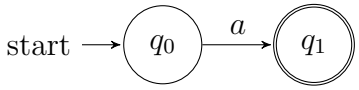


Figura 2.6: Exemplo de Autómato que aceita a linguagem $L = \{a\}$.

Exemplo 2.8 *$L = \{\epsilon\}$ é a linguagem aceite pelo autómato $M = (\{q\}, \sum, \emptyset, q, \{q\})$.*

Exemplo 2.9 *$L = \{a\}$ é a linguagem aceite pelo autómato $M = (\{q_0, q_1\}, \sum, \delta, q_0, \{q_1\})$, onde a função de transição δ é definida por: $\delta(q_0, a) = q_1$.*

Exemplo 2.10 *$L = L_1 \cup L_2$ é a linguagem aceite pelo autómato $M = (Q, \sum, \delta, q_0, F)$, onde M é construído por $M_1 = (Q_1, \sum, \delta_1, q_0^1, F_1)$ e $M_2 = (Q_2, \sum, \delta_2, q_0^2, F_2)$. Suponhamos que M_1 aceita a palavra ab^*a e M_2 a palavra aa^*b. (a^* significa que o símbolo a pode aparecer 0 ou mais vezes). A reunião corresponderá a: $ab^*a + aa^*b = a(b^*a + a^*b)$. O autómato M será, então, representado da seguinte forma:*

Onde os estados de M_1 e M_2 têm de fazer parte do autómato M.

Exemplo 2.11 *$L = L_1L_2$ é a linguagem aceite pelo autómato $M = (Q, \sum, \delta, q_0, F)$, onde as condições de M_1 e M_2 são as mesmas do Exemplo 10. Segue-se, assim, o autómato M.*

Onde:

- *o estado inicial é o estado inicial de M_1;*
- *o estado final é o estado final de M_2.*

Exemplo 2.12 *$L = L_1^*$ (autómato-fecho) é a linguagem aceite pelo autómato $M = (Q, \sum, \delta, q_0, F)$.*

Em relação ao autómato-fecho, é necessário:

- *Acrescentar um novo estado inicial que é também estado final;*
- *O novo estado inicial tem as mesmas transições que o estado inicial antigo;*
- *Adicionar a transição a partir do estado final já existente para o estado que recebe transições do estado inicial.*

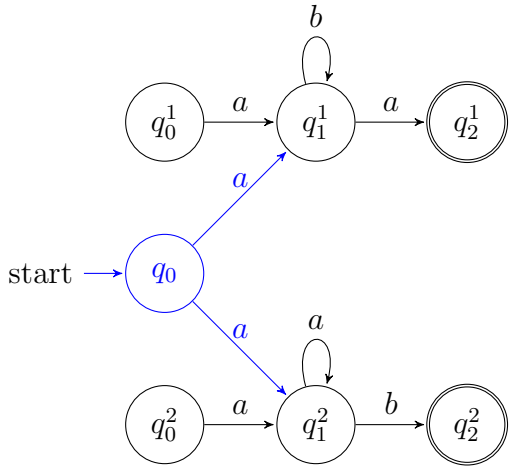


Figura 2.7: Exemplo de Autómato que aceita a linguagem $L = L_1 \cup L_2$.

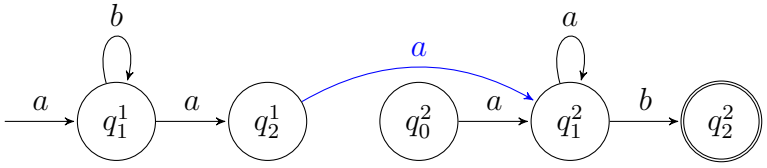


Figura 2.8: Exemplo de Autómato que aceita a linguagem $L = L_1 L_2$.

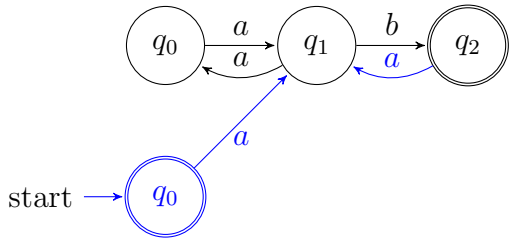


Figura 2.9: Exemplo de Autómato que aceita a linguagem $L = L_1^*$.

2.6 Linguagens independentes do contexto

As linguagens independentes de contexto e os autómatos de pilha são utilizados, normalmente, no desenvolvimento de analisadores sintáticos, parte do compilador. [12]

2.6.1 Árvores de derivação

A derivação de palavras nos compiladores é, por vezes, feita sob a forma de árvore. A árvore terá como raiz o símbolo inicial e os símbolos terminais serão as folhas.

Para uma gramática independente do contexto $G = (N, \sum, P, A)$, uma árvore de derivação D tem como raiz o símbolo inicial A e satisfaz uma das seguintes condições:

- A árvore D só tem uma subárvore D_1 com um único nó ϵ e a produção $A \to \epsilon \in P$;
- Sendo $D_1, \ldots, D_k$ com $K \geq 1$ as subárvores de D e $X_1, \ldots, X_k$ as raízes de $D_1, \ldots, D_k$. Então existe uma produção $A \to X_1, \ldots, X_k \in P$ e para cada $n = 1, \ldots, k$:
  - se $X_n \in \sum$ então D_n tem apenas um nó, X_n;
  - se $X_n \in N$ então D_n é uma árvore de derivação para a gramática $G = (N, \sum, P, A)$.

Exemplo 2.13 *A expressão $id + (m \times id)$ é gerada pela árvore ilustrada na Figura 2.10. Foi utilizada uma gramática independente do contexto com o símbolo inicial E e com o seguinte conjunto de produções:*
$P = \{E \to +T|T, T \to T \times F|F, F \to id|m|(E)\}$.

Segue-se a demonstração para chegar à expressão $id + (m \times id)$ utilizando P.

$E \to E + T \to T + T \to F + T \to id + T \to id + F \to id + (E) \to id + (T) \to id + (T \times F) \to id + (F \times F) \to id + (mxF) \to id + (m \times id)$

2.6.2 Autómatos de pilha

Os autómatos de pilha têm algumas caraterísticas dos autómatos finitos. A diferença entre os dois é que os autómatos de pilha utilizam uma memória auxiliar em forma de pilha. A principal caraterística deste tipo de autómatos

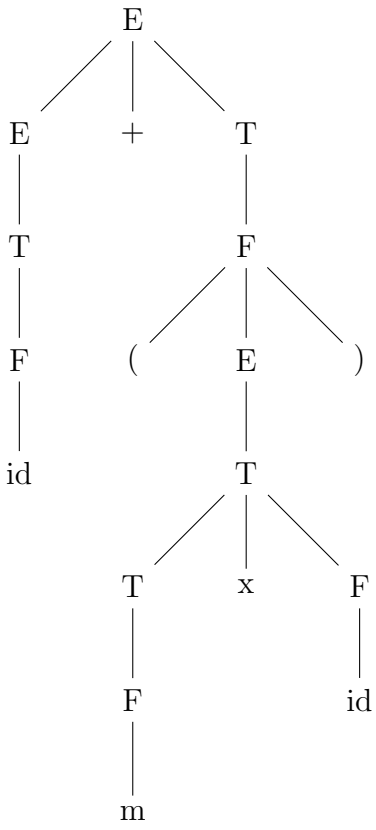


Figura 2.10: Árvore de derivação.

é o aumento do seu poder computacional. Por outro lado, as linguagens utilizadas não são tão simples, dado que, nas suas caraterísticas, são utilizadas linguagens livres de contexto.

Num autómato de pilha definido por séptuplo $P = (Q, \sum, \Gamma, \delta, q_0, Z_0, F)$, onde:

- Q é um conjunto finito de estados;
- $\sum$ trata-se do alfabeto de entrada da pilha;
- Γ é utilizado para representar os símbolos da pilha e também se trata de um conjunto finito;
- Quanto á função de transição, é utilizado o símbolo δ e é definido da seguinte forma:

  $\delta : Q \times (\sum) \cup \{\epsilon\}) \times \Gamma^* \rightarrow (Q \times \Gamma^*)$, onde Γ^*é o conjunto se palavras sobre o alfabeto Γ.
- q_0 é o estado inicial que pertence a Q;
- Z_0 é o símbolo inicial da pilha;
- $F \subseteq Q$ representa o conjunto dos estados finais.

Linguagem aceite pelos autómatos de pilha

Uma linguagem aceite por um autómato de pilha é o conjunto de frases tais que, se processarmos as frases em zero ou mais passos, chegamos a uma configuração. Uma configuração é um triplo $(q, w, \alpha) \in Q \times \sum^* \times \Gamma^*$. Uma configuração representa a posição atual em que o autómato se encontra ao ler uma frase, onde:

- q representa o estado atual do autómato;
- w é a frase que é apresentada para ler, isto é, o que ainda falta ler do autómato;
- α refere-se ao conteúdo da pilha.

Para verificarmos se uma frase pode ou não ser aceite por um autómato temos de ter em conta dois passos importantes: chegar ao estado final e consumir todos os símbolos, não sendo obrigatório esvaziar a pilha.

A linguagem aceite por um autómato pilha é representada por:

$L(P) = w \in \sum^* : (q_0, w, Z_0) \vdash_P^* (q^{'}, \epsilon, \alpha)$, onde $\vdash_P^*$, podem ser feitas transições de 0 ou mais passos;

Uma configuração (q_0, w, Z_0) é uma configuração inicial do autómato de pilha; por outro lado, $(q^{'}, \epsilon, \alpha)$ com $q^{'} \in F, \alpha \in \Gamma^*$ trata-se da configuração final do autómato de pilha.

2.6.3 Autómatos de pilha deterministas

Os autómatos de pilha têm que satisfazer uma das condições seguintes se $\forall_{q \in Q} \forall_{Z \in \Gamma}$:

- $\delta(q, \epsilon, Z) = \emptyset$ e $\forall_{a \in \sum} \#(\delta(q, a, Z)) \leq 1$;
- $\#((q, \epsilon, Z)) \leq 1$ e $\forall_{a \in \sum} \delta(q, a, Z) = \emptyset$.

Toda a linguagem livre do contexto pode ser reconhecida por um autómato de pilha apenas com um ou três estados, como constataremos no próximo capítulo. Por outras palavras, só a estrutura da pilha funciona como memória e todos os estados dos autómatos de pilha poderiam não ser utilizados e tal não reduziria o poder computacional.

Exemplo 2.14 *O autómato de pilha:*
$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$ *aceita a linguagem* $L_{wwr} = \{ww^R : w$ *é a frase constituida por* $0^{'}s e 1^{'}s\}$.

Seguem-se as funções de transição:

$\delta(q_0, 0, Z_0) = \{q_0, 0Z_0)\}$, *onde* q_0 *é o estado da pilha, 0 é o símbolo que vai ser lido e* Z_0 *é o topo da pilha (o mesmo para as funções transições seguintes);*

$\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$;
$\delta(q_0, 0, 0) = \{(q_0, 00)\}$;
$\delta(q_0, 0, 1) = \{(q_0, 01)\}$;
$\delta(q_0, 1, 0) = \{(q_0, 10)\}$;
$\delta(q_0, 1, 1) = \{(q_0, 11)\}$;
$\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$;
$\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$;
$\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$;
$\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$;
$\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$;
$\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$.

Na Figura 2.11 está exemplificada a representação deste autómato.

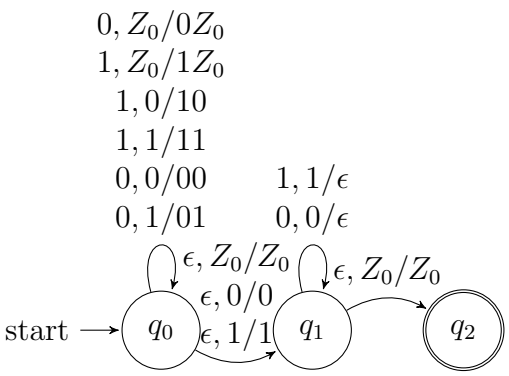


Figura 2.11: Exemplo de Autómato pilha

2.6.4 Autómatos de pilha e linguagens livres do contexto

Tal como foi referido em subseções anteriores, as linguagens livres de contexto são geradas a partir das gramáticas livres de contexto. A partir daí, a construção de um *parser* é imediata. A estrutura da pilha tem que ser adaptada ao computador utilizado.

O autómato de pilha é aquele que melhor se adequa ás linguagens baseadas em XML, porque existem dependências de abertura e fecho de *tags*, como vai ser demonstrado no próximo capítulo.

Capítulo 3

Linguagens formais baseadas em XML

3.1 Introdução

Neste capítulo enuciamos as especificações de uma diversidade de linguagens *markup*.

Todas as linguagens aqui apresentadas são linguagens *markup* baseadas em XML (*Extensible Markup Language*). São simples, guardam e transferem informação para diferentes bases de dados, são auto descritivas, ou seja, a estrutura dos dados está incorporada nos mesmos e, portanto, quando os dados são transferidos, não é necessário voltar a estruturá-los para o armazenamento ser possível. A construção de um bloco de código tem de se iniciar e terminar com uma *tag*. E ainda é permitido a existência de uma hierarquia nas estruturas. Um documento em XML é composto por uma raiz e por vários subelementos e, por isso, é pertinente pensar numa estrutura em árvore.

É de notar que o conceito de linguagens formais está presente em vários aspetos da linguagem XML, tais como:

- Utilização de árvores de derivação;
- Gramáticas livres de contexto com a utilização de DTDs (*Document Type Definition*), documento que descreve um documento XML;
- Expressões regulares com a utilização de DTDs;
- Expressões regulares com a utilização de *XPath*.

É possível relacionar autómatos com a linguagem XML através de quatro aspetos de processamento:

- Validação - verificação de um documento XML com DTDs;
- Navegação - seleção de um conjunto de posições num documento XML com *Xpath*;
- *Querying* - extração de informação a partir de um documento XML (*XQuery*);
- Transformação - construção de um novo documento XML, tendo como referência um outro documento XML (*XSLT*).

Todos estes conceitos serão exemplificados ao longo dos próximos tópicos. Para a representação da estrutura de documentos baseados em XML vamos utilizar árvores.

Em relação às linguagens markup, enumeraremos múltiplos fatores, nomeadamente:

- Em primeiro lugar deverão descrever todo o comportamento de um agente virtual através de modelos matemáticos que levam à sua especificação. A especificação é um conjunto de primitivas que definem emoções, discurso, comportamento não verbal entre outros;
- Em segundo, as primitivas devem poder ser combinadas o que nos permite obter uma maior diversidade de resultados;
- Em terceiro lugar, dever-se-á poder elaborar extensões de uma linguagem;
- Por fim, terá de existir, ainda, um compilador para verificar se a especificação está sintática e semanticamente correta.

Existe um leque de variedades de linguagens para descrever o comportamento de um agente virtual.

3.2 *Behavior Markup Language* (BML)

A linguagem *Behavior Markup Language* (BML) é baseada na *Extensible Markup Language* (XML), como já foi referido anteriormente, e descreve o comportamento humano [19]. Cada comportamento irá ser executado por agentes, num determinado ambiente. Em apenas um bloco de código é possível coordenar os movimentos dos agentes, tais como, o discurso, os gestos, os movimentos da cabeça, o rosto, os lábios, os braços, as pernas e ainda

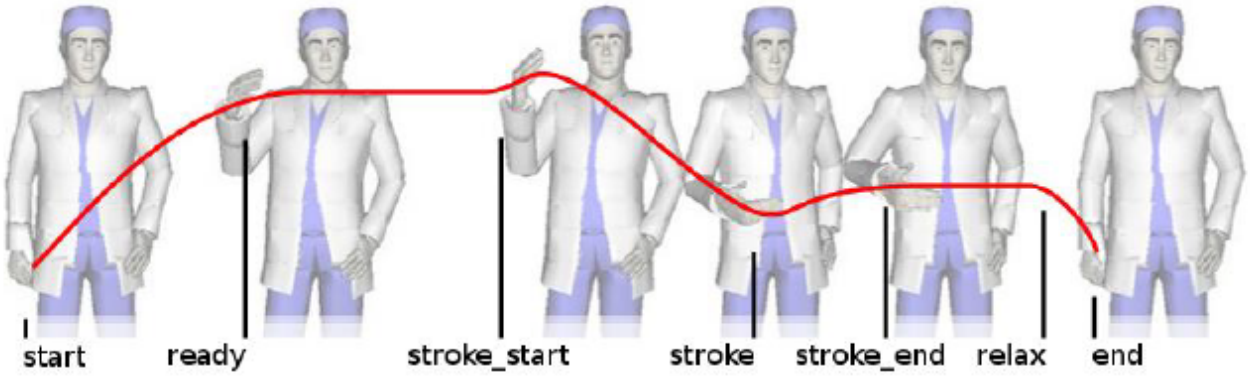


Figura 3.1: Pontos de sincronização para um gesto. Fonte: [5].

alguns comportamentos de espera, por exemplo, aqueles que se manifestam quando se aguarda pelo fim do discurso de um outro agente.

Cada comportamento está ainda divido em seis fases, no fim das quais existe um ponto com o nome da transição do movimento. Esses pontos são: *start*, *ready*, *stroke-start*, *stroke*, *stroke-end*, *relax* e *end*. E, por fim, estabelecemos uma ligação entre os pontos de modo a termos a sincronização dos movimentos [7].

Na Figura 3.1 temos um exemplo do uso dos pontos de sincronização. Neste caso, estes descrevem as diferentes fases de um simples gesto com a mão. Como os nomes indicam, tanto a preparação como a finalização para o movimento, ocorrem entre *start* e *ready*, e entre o *relax* e *end*, respetivamente. O comportamento surge entre o *start* e o *relax*, havendo mais enfâse durante a fase do *stroke*. Esta última fase nem sempre é utilizada, existindo comportamentos em que não é exigida a sua especificação [5].

Especificação de BML:

No primeiro passo são necessários *tags* $< bml >< /bml >$ para iniciar o BML.

Existe uma série de elementos BML para controlar o agente. Começando pela cabeça, o elemento $< head >$, como o próprio nome indica, é utilizado para movimentos da cabeça que podem ser orientados por um ângulo. $< torso >$, para controlar a coluna e os ombros. $< face >$, para expressões faciais. $< gaze >$, para movimentos coordenados dos olhos, pescoço e cabeça. $< body >$, controla todo o movimento do corpo, tal como, orientação, postura e posição. $< legs >$, para coordenar movimentos dos joelhos, tornozelos e pés. $< gesture >$, movimentos simultâneos das mãos e braços. $< speech >$, para discurso corrido e com pausas. E, finalmente $< lips >$, para controlar

os lábios.

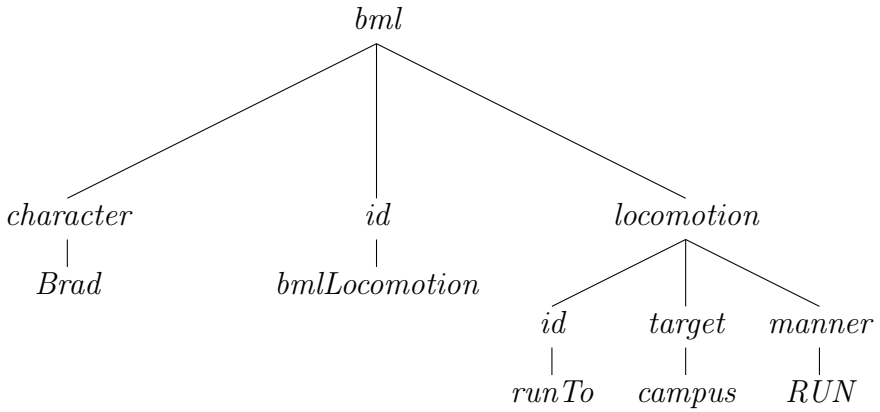
A *tag* que pode, por exemplo, colocar o agente em movimento, é a tag $< locomotion >$. Esta é constituída pelos atributos descritos na Tabela 3.1.

Tabela 3.1: Elemento *Locomotion* BML.

Atributo	**Descrição**
id	Identificação única do elemento BML.
target	Localização final.
manner	O modo da deslocação. Exemplos: WALK, RUN, STRAFE, ...

Exemplo 3.1 *Tendo em conta os atributos da tag locomotion, vai ser apresentada, neste exemplo, a construção de uma árvore. Esta árvore representa o bloco de código que se segue. A árvore terá como raiz o elemento bml. Este elemento é constituído pela tag inicial e tag final e terá como subelemento locomotion. Aqui poderão existir elementos dentro de elementos. A árvore terá como nós os atributos: character, id e locomotion. Locomotion; por sua vez, também terá como filhos: id, target e manner.*

```
<bml character="Brad"
     id="bml_locomotion">
    <locomotion id="run_to"
                target="campus"
                manner="RUN"/>
</bml>
```



Para este bloco de código estar bem definido precisa de:

- *Ter tag de início e fim correspondente à linguagem;*

- *Apresentar tag de início e fim de cada atributo utilizado;*
- *Respeitar a ordem das tags;*
- *Verificar se todos os atributos obrigatórios estão a ser utilizados.*

Relativamente, aos comportamentos face à postura, temos dois tipos diferentes:

- Um temporário, terá o seu termino $< posture >$;
- E outro permanente $< postureShift >$.

O agente poderá, por exemplo, se baixar com os braços abertos ou se sentar com os braços cruzados e voltar à sua pose inicial.

Apenas um atributo é necessário:

Tabela 3.2: Elemento *Posture* BML.

Atributo	**Descrição**
id	Identificação única do elemento BML.

Para além disso, temos a *tag* $< stance >$ (elemento filho das *tags* acima referidas), em exemplo, deitar e sentar:

Tabela 3.3: Elemento *Stance* BML.

Atributo	**Descrição**
type	Postura do corpo. Exemplos: SITTING, CROUCHING, STANDING, LYING.

E, na Tabela 3.4, como complemento à postura do agente, temos $<$ *pose* $>$, (também elemento 'filho'):

Exemplo 3.2 *O DTD que se encontra na Figura 3.3 corresponde à estrutura do bloco de código representado na Figura 3.2.*

E ainda vai ser feita a validação do DTD descrito na Figura 3.3 com um autómato de pilha. Mas, para uma melhor compreensão e tendo o mesmo significado que os autómatos pilha, será feito em fluxograma como se segue na Figura 3.4.

Tabela 3.4: Elemento *Pose* BML.

Atributo	**Descrição**
part	Parte do corpo. Exemplos: ARMS, LEFT-ARM, RIGHT-ARM, LEGS, LEFT-LEG, RIGHT-LEG, HEAD, WHOLEBODY, ...
lexeme	O que fazer com a parte do corpo que foi selecionada. Exemplos: ARMS-CROSSED, ARMS-NEUTRAL, ARMS-OPEN, LEGS-CROSSED, LEGS-NEUTRAL, LEGS-OPEN, LEANING-FORWARD, LEANING-BACKWARD, ...

```
<bml character="Brad"
     id="bml_posture">
     <posture
       id="behavior_posture"
       start="2"
       end="12">
       <stance
         type="STANDING"/>
       <pose
         type="LEGS"
         lexeme="LEGS_NEUTRAL"/>
     </posture>
</bml>
```

Figura 3.2: Exemplo de código em BML *posture*

```
<!DOCTYPE bml [
   <!ELEMENT bml (character, id, posture)>
   <!ELEMENT posture (id, start, end, stance, pose)>
   <!ELEMENT stance (type)>
   <!ELEMENT pose (type, lexeme)>
]>
```

Figura 3.3: DTD que descreve a estrutura da Figura 3.2.

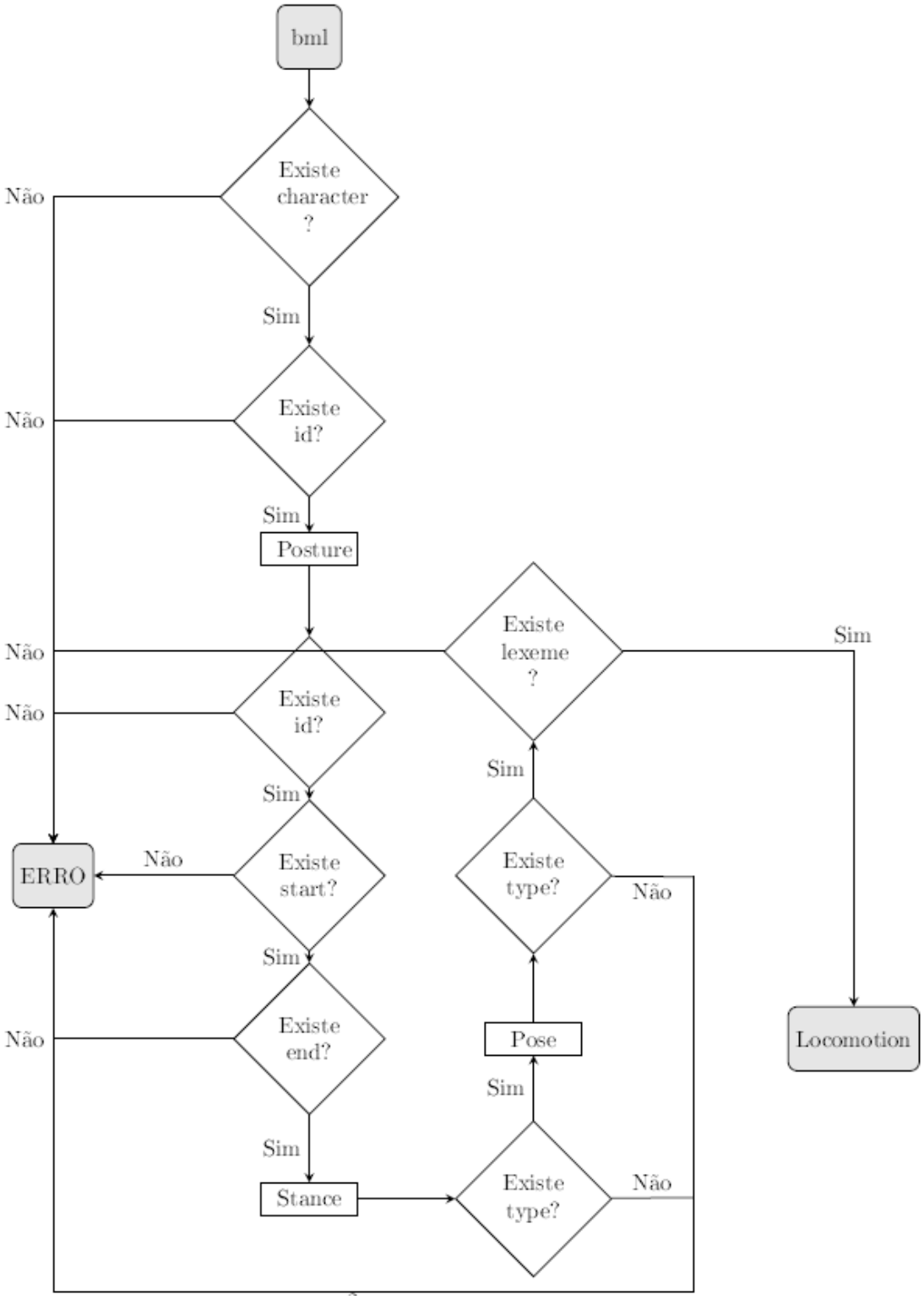


Figura 3.4: Validação em fluxograma do código descrito na Figura 3.2.

3.3 *Functional Markup Language* (FML)

Esta linguagem vai para além dos comportamentos físicos de um agente [9]. Representa as intenções, os objetivos e planos do mesmo. Apesar da mesma continuar a ser estudada, as características que poderão ser utilizadas nos agentes, são: o nome, o género, o tipo (se é um agente ou alguma espécie do ambiente), a apresentação física, a voz, a personalidade, as emoções, o humor, entre outros.

Quanto à comunicação, vamos ter diferentes tipos, o que irá depender das reações e emoções que o agente tiver. Este tipo de comunicação é semelhante a uma conversa entre humanos e todos os sentimentos que estejam envolvidos nesta; deste modo, temos uma comunicação realista.

Conforme já foi referenciado, esta linguagem engloba outros aspetos para além de comportamentos. A especificação da mesma contém características de um agente, (nome, género, tipo, voz, personalidade, emoção, humor, entre outros) e elementos que estes podem usar para comunicar [13].

Nestes, existe um agente que entrega a mensagem e outro que recebe e ainda uma série de funções conjugadas com emoções.

Especificação de FML:

O elemento $< gaze >$ é caraterizado pelos atributos:

Tabela 3.5: Elemento *Gaze* FML.

Atributo	**Descrição**
gaze-type	Um símbolo que descreve os diferentes tipos de olhar para um objeto. Exemplos: *avert*, *cursory*, *look*, *focus*, *weak-focus*.
target	Nome do objeto no qual o agente irá fixar-se.
priority	Um símbolo que vai ser utilizado para determinar a prioridade do comando *gaze*.
reason	O porquê de estar a utilizar o comando *gaze*.

Já $tag < affect >$ é utilizada para exprimir as emoções dos agentes, como está exemplificado na Tabela 3.6.

A $tag < appraisal >$ é mais específica que a $< affect >$ e é constituída por dois atributos, os mesmos estão referênciados na Tabela 3.7.

Tabela 3.6: Elemento *Affect* FML.

Atributo	**Descrição**
type	Indica o tipo de afeto. Exemplos: alegria, medo, raiva.
target	Agente a quem vai ser destinado a emoção.
stance	Identifica se a emoção é voluntária ou involuntária.
intensity	Intensidade da emoção.

Tabela 3.7: Elemento *Appraisal* FML.

Atributo	**Descrição**
type	Uma variável de estimação única. Exemplos: conveniência, controlo.
value	Intensidade da variável de estimação.

3.4 *Virtual Human Markup Language* (VHML)

É baseada também em XML. Aqui são utilizados diversos subsistemas para a descrição de cada componente do comportamento:

- *Dialogue Manager Markup Language* (DMML) para o controlo do diálogo;
- *Facial Animation Markup Language* (FAML) para expressões faciais;
- *Body Animation Markup Language* (BAML) para movimentos corporais;
- *Speech Markup Language* (SML) para o discurso;
- *Emotion Markup Language* (EML) para as diversas emoções;
- E ainda, *HyperText Markup Language* (HTML), para a estrutura da linguagem.

Especificação de VHML:

É uma linguagem que está mais focada na utilização do discurso e movimentos complementares ao mesmo [15]. As *tags* a utilizar para as restantes partes do corpo são as seguintes:

Tabela 3.8: Partes do corpo e respetivas *tags* da linguagem VHML.

Partes corporais	**TAGS**
Braços	*left-arm-front, right-arm-front, left-arm-back, right-arm-back, left-arm-abduct, right-arm-abduct, left-arm-adduct, right-arm-adduct, left-forearm-flex, right-forearm-flex, left-forearm-turn-right, left-forearm-turn-left, right-forearm-turn-right, right-forearm-turn-left*
Mãos	*left-hand-forward, right-hand-forward, left-hand-back, right-hand-back, left-hand-abduct, right-hand-abduct, left-hand-adduct, right-hand-adduct, left-hand-open, right-hand-open*
Tronco	*trunk-turn-left, trunk-turn-right, trunk-front, trunk-back, trunk-bend-left, trunk-bend-right*
Bacia	*pelvis-turn-left, pelvis-turn-right, pelvis-left, pelvis-right*
Pernas	*left-leg-front, right-leg-front, left-leg-back, right-leg-back, left-leg-abduct, right-leg-abduct, left-leg-adduct, right-leg-adduct, left-leg-turn-right, left-leg-turn-left, right-leg-turn-right, right-leg-turn-left, left-calf-flex, right-calf-flex*
Pés	*left-foot-up, right-foot-up, left-foot-down, right-foot-down, left-foot-abduct, right-foot-abduct, left-foot-adduct, right-foot-adduct, left-forefoot-up, right-forefoot-up, left-forefoot-down, right-forefoot-down*

3.5 *Multi-modal Presentation Markup Language* (MPML)

Esta é outra das linguagens que é baseada em XML. A criação desta linguagem surgiu para facilitar a apresentação multimodal em espaços 3D, por forma a obter vários modelos de representação [6]. É mais utilizada para expressar emoções conjugadas com o uso expressões faciais, gestos com as mãos e movimentos com a cabeça. É de fácil utilização e oferece um editor visual (*MPML 3.0 Visual Editor*). Relativamente aos possíveis ambientes a utilizar, estes são baseados em *web*. (exemplo: interação com o público).

Especificação de MPML:

São necessárias as *tags* $< mpml >< /mpml >$ para se poder iniciar o bloco de código. O primeiro passo é criar um agente. A *tag* $< Agent >$ tem os seguintes atributos:

Tabela 3.9: Elemento *Agent* pertence à linguagem MPML.

Atributo	**Descrição**
id	Identificação do agente.
name	Nome do agente.
description	Pequena descrição do agente.
system	Sistema utilizado pelo agente.
spot	Posição inicial do agente.
location	Localização com as coordenas x e y.
x	Coordenada x do agente.
y	Coordenada y do agente.
voice	Voz usada no discurso.
agreeableness	Valores entre 0 e 100 de quanto é simpático.
activity	Valores entre 0 e 100 referente à atividade da personalidade do agente.

A *tag* $< Page >$ define um *background*. Para cada ação poder ser executada é necessário estar definido um *background*. Com os atributos mencionados na Tabela 3.10.

A *tag* $< Seq >$ serve para definir um conjunto de ações, podendo ser executadas uma de cada vez. É constituída pelos atributos da Tabela 3.11.

Tabela 3.10: Elemento *Page* da linguagem MPML.

Atributo	**Descrição**
id	Identificação da página.
name	Nome da página.
description	Pequena descrição da página.
ref	URL da página que contém o *background*.

Tabela 3.11: Elemento *Seq* inserido na linguagem MPML.

Atributo	**Descrição**
id	Identificação da sequência.
name	Nome da sequência.
description	Pequena descrição da sequência.
agents	Agentes utilizados na sequência com algumas restrições.

A $tag < Par >$ define um conjunto paralelo de ações que serão executadas ao mesmo tempo demonstradas pela Tabela 3.12.

Tabela 3.12: Elemento *Par* que faz parte da linguagem MPML.

Atributo	**Descrição**
id	Identificação do conjunto paralelo de ações.
name	Nome do conjunto paralelo de ações.
description	Pequena descrição do conjunto paralelo de ações.

A $tag < Move >$ é a responsável pela movimentação do agente e é constituída pelos atributos descritos na Tablela 3.13.

Na Tabela 3.14, a $tag < Play >$ representa os gestos ou algum tipo de animação que vão ser executados por um agente.

A $tag < Speak >$ serve para o agente poder comunicar. Os seus respetivos atributos estão representados na Tabela 3.15.

Tabela 3.13: Elemento *Move* referente à linguagem MPML.

Atributo	**Descrição**
id	Identificação do movimento.
name	Nome do movimento.
description	Pequena descrição do movimento.
agent	*id* do agente que se vai movimentar.
spot	*id* do destino final.
location	Localização com as coordenadas x e y.
x	Coordenada x do agente para o destino.
y	Coordenada y do agente para o destino.

Tabela 3.14: Elemento *Play* integrado na linguagem MPML.

Atributo	**Descrição**
id	Identificação da ação.
name	Nome da ação.
description	Pequena descrição da ação.
agent	*id* do agente que vai realizar a ação.
act	Ação em conformidade com o agente ou o sistema.

3.6 *Avatar Markup Language* (AML)

Trata-se de uma linguagem em 3D, de utilização acessível para avatares. É flexível na criação de animações e pode ser estendida facilmente. Fornece expressões, gestos e controlo sincronizado de movimentos [8].

Especificação de AML:

São necessários $< AML >< /AML >$ para iniciar o bloco de código. Este elemento $< AML >$ é constituído pelos atributos abordados na Tabela 3.16.

Na Tabela 3.17, seguem-se outros dois elementos, $< FA >$ para animação facial e $< BA >$ para animação corporal. Ambas as *tags* têm os mesmos atributos.

Tabela 3.15: Elemento *Speak* contido na linguagem MPML.

Atributo	**Descrição**
id	Identificação do discurso que vai ser utilizado.
name	Nome do discurso.
description	Pequena descrição do discurso.
agent	*id* do agente que vai falar.

Tabela 3.16: Elemento $< AML >$.

Atributo	**Descrição**
face-id	Identificação da cara do agente 3D.
body-id	Identificação do corpo do agente 3D.
root-path	Localização dos ficheiros de animação.
name	O nome da animação.

Um outro elemento é $< TTS >$ para o discurso e tem os atributos descritos na Tabela 3.18. Tem ainda um sub-elemento $< Text >$ que define o texto que vai ser utilizado no discurso.

O elemento $< ABML >$ é semelhante ao $< AFML >$, mas só iremos detalhar o $< ABML >$, porque é o que está direcionado para os movimentos corporais. Este tem uma *tag* $< Settings >$ que contém $< BAPLibPath >$ para localizar as animações. Existem animações que já estão predefinidas e em formato BAP e poderão ser encontradas através de $< BodyAnimationTrack >$.

A animação corporal irá usar a *tag* $< PredefinedAnimation >$ na qual cada animação tem os seguintes atributos: *start-time*, *speed* e *priority* da ação. Onde o *speed* pode tomar os valores: lento, normal ou rápido. O atributo *priority*, especifica a animação que deverá ser usada. Temos, ainda, a *tag Intensity* que pode ser utilizada para aumentar ou diminuir o efeito da animação.

Proporciona-nos também uma série de comportamentos definidos: *Facing, Pointing, Walking, Waiting, Resetting.*

Tabela 3.17: Elemento $< FA >$ ou $< BA >$ da linguagem $< AML >$.

Atributo	**Descrição**
start-time	Onde começam os *scripts*.
input-time	O nome do ficheiro AFML (*Avatar Face Markup Language*) ou ABML (*Avatar Body Markup Language*).

Tabela 3.18: Elemento $< TTS >$ da linguagem $< AML >$.

Atributo	**Descrição**
mode	Verifica se o discurso tem *tags* de emoção.
start-time	Início do discurso.
output-fap	O nome do ficheiro gerado por o *output* FAP.
output-wav	O nome do ficheiro quando o som é gerado .

Cada comportamento tem uma localização com coordenadas x, y e z, $< XCoor >$, $< YCoor >$ e $< ZCoor >$. O comportamento *Walking* consegue especificar qualquer número de pontos de controlo com os quais o avatar se poderá cruzar, através da *tag* $< ControlPoint >$.

Não existem limitações quanto a ações simultâneas e apenas é preciso dispensar alguma atenção aos modos existentes.

No Anexo A temos a comparação das linguagens AML, MPML, VHML e BML.

3.7 Conclusão

A análise destas linguagens é muito importante, pois só assim conseguimos um melhor conhecimento face à biomecânica do corpo humano. É necessário realçar que as linguagens formais aqui introduzidas são representadas através de sistemas com base matemática.

Podemos verificar, após um longo estudo, que a linguagem escolhida é *Behavior Markup Language*. Não só porque as suas características, anteriormente definidas, se adequam de uma melhor forma, como também porque se trata da linguagem mais completa, a que tem mais definições a nível de

comportamento não verbal.

Convém acrescentar também que as animações podem ser parametrizáveis. Para além disso, os pontos de sincronização, já referidos na descrição, são uma mais valia em relação às outras linguagens, dando liberdade ao utilizador de aceder a seis pontos de um comportamento.

É uma linguagem que está em constante desenvolvimento e que já foi utilizada em vários tipos de ambientes, inclusivamente em *robots*. Ou seja, esta linguagem já foi, é, e continuará a ser alvo de inúmeros testes, com a possibilidade de diversos resultados.

Um outro aspeto a apontar é que são permitidas extensões, o que nos possibilita criar novas funções. Por todos os aspetos antes referenciados, não só nos pareceu a escolha mais correta, como também a mais adequada para ser implementada nos agentes virtuais.

Capítulo 4

Ambientes e agentes virtuais

4.1 Introdução

Neste capítulo introduz-se a definição de ambientes virtuais, *realizers* e ainda é descrito em detalhe algumas das funcionalidades da plataforma *Smartbody* para que o leitor possa acompanhar as estratégias introduzidas no capítulo seguinte.

Nesta seção também é introduzido parte do funcionamento normal do corpo humano. Os diferentes tipos de articulações e as limitações associadas. Depois é feita uma comparação com os esqueletos de agentes virtuais. Estes limites vão ser importantes para a investigação de estratégias para os agentes com deficiências motoras.

4.2 Agentes virtuais

Um agente virtual, neste caso particular, possui uma aparência humana para que possamos obter uma combinação de inteligência artificial com representação gráfica [13].

Todos os movimentos do corpo são feitos através de juntas que se situam entre os ossos [4]. Quando nos referimos às juntas, é face ao esqueleto virtual, já que, no corpo humano, correspondem às articulações. Estas juntas vão ter diferentes graus de liberdade (DOF's), isto é, vão efetuar diferentes tipos de movimentos. São estes graus de liberdade que vão definir quantos valores podemos parametrizar.

Da mesma forma que para fazermos uma translação de um objeto necessitamos de 3 dimensões (x, y, z) sobre os 3 eixos, para os graus de liberdade também vão ser necessários 3 graus de translação e 3 de rotação. Por outras palavras, uma junta que tenha só um eixo de rotação tem 1 grau de liberdade.

Na Tabela 4.1, podemos ficar a perceber melhor os graus de liberdade das juntas e os respetivos eixos.

Tabela 4.1: Graus de liberdade de juntas e eixos.

DOF's	**Número de eixos de rotação**	**Número de eixos de translação**
1	1	0
6	3	3

Os principais tipos de movimentação permitidos por uma articulação são:

- **Extensão** - ato de dobrar uma articulação aumentando o ângulo, de que é exemplo o esticar do cotovelo;
- **Flexão** - ato de dobrar uma articulação diminuindo o ângulo, demonstrável no dobrar do cotovelo;
- **Abdução** - movimento de um dos membros afastando-se da linha mediana do corpo;
- **Adução** - movimento de um dos membros aproximando-se da linha mediana do corpo;
- **Rotação** - ocorre quando se roda alguma parte do corpo em torno do eixo central, exemplificada através do movimento de abanar a cabeça para os lados.

Tendo em conta que estes tipos de movimentos se encontram em graus, os limites de estudantes do sexo masculino definem-se na Tabela 4.2.

A simulação do movimento das articulações vai ser realizada através de animações. Os métodos de cinemática de captura de movimento são apenas dois, dos muitos que existem para criar animações.

A cinemática apenas considera a pose do esqueleto. Esta pode ser criada através de pontos ou mediante o recurso aos segmentos dos esqueletos numa figura. Este método não tem em conta qualquer tipo de física.

Em relação à captura de movimentos, como o nome indica, são capturados movimentos do corpo humano. Este método tenta ao máximo a aproximação da realidade, tirando os pontos do movimento através da pele e passando os mesmos para os esqueletos virtuais.

É a partir das animações que podemos ver todos estes movimentos enquadrados nos ambientes virtuais.

Tabela 4.2: Limitação, em graus, de movimentos de tipos de articulações. Fonte: [4]

Tipos de movimentos	**Média em graus**
Flexão do ombro	188^o
Extensão do ombro	61^o
Abdução do ombro	134^o
Adução do ombro	48^o
Flexão do cotovelo	142^o
Flexão da bacia	113^o
Abdução da bacia	53^o
Adução da bacia	31^o
Flexão do joelho com a ajuda do braço	$125, 144^o$
Flexão do joelho de pé	$113, 159^o$
Rotação lateral do joelho	43^o
Flexão do tornozelo	35^o
Extensão do tornozelo	38^o

4.3 Ambientes virtuais

A seleção da linguagem teve o propósito da sua utilização num ambiente virtual já que cada ambiente poderá estar programado apenas para uma determinada linguagem.

Um ambiente virtual, de certa forma, retrata um mundo real. Toda a ilusão se deve a três tipos de *hardware*: sensores de posição na cabeça para detetar todos os movimentos corporais, *effectors* para simular o sentido das operações e ainda um *hardware* específico que liga os dois anteriores, produzindo, assim, todas as experiências de um ambiente físico.

Em geral, o *software* para um ambiente virtual tem que ter três funções essenciais:

- A forma e a cinemática para agentes e objetos;
- Interações entre os mesmos e o ambiente, respeitando regras de comportamento;
- A extensão do ambiente envolvente.

4.4 Integração de agentes virtuais em ambientes virtuais

É através do SAIBA (Situação, Agente, Intenção, Comportamento, Animação) *framework* que integramos os agentes virtuais nos ambientes. O objetivo principal é unificar comportamentos que vão ser utilizados pelos agentes e, deste modo, a partilha de recursos entre utilizadores e programadores será mais fácil de realizar.

Existem diferentes níveis de abstração que representam as interfaces entre três estados: o planeamento da intenção de comunicar, o planeamento da realização multimodal dessa intenção e a realização de comportamentos planeados. Entre os dois primeiros estados é utilizada a linguagem já conhecida, FML. E entre o segundo e o último estado, recorremos à linguagem BML que, como também já foi mencionado, descreve o comportamento verbal e não verbal.

4.5 Ambientes que utilizam BML

A seleção dos ambientes e *realizers* aqui apresentados teve em consideração dois aspetos:

- Todos têm de garantir a interpretação da linguagem BML;
- Devem ser de código aberto, ou seja, deve ser possível examinar toda a estrutura dos mesmos.

Os *realizers* são modelos de *software* onde está a implementação de uma interface padrão com a especificação da linguagem.

A ferramenta *Virtual Human Toolkit* [18] é um conjunto de bibliotecas, módulos e ferramentas que trabalham em uníssono para a criação de um agente virtual. Foi desenvolvida com a intenção de que qualquer pessoa possa dela usufruir.

As finalidades desta ferramenta são as seguintes: o agente virtual tem de ser capaz de simular qualquer tipo de comportamento humano, reagir ao comportamento de outros e ainda tem de estar preparado para responder corretamente a questões que lhe possam ser feitas.

Podemos efetuar o *download* a partir do *link* referenciado em [17]. Esta ferramenta ainda está em desenvolvimento e foi criada por investigadores da Universidade da Califórnia do Sul.

Embodied Agents Behavior Realizer (EMBR) e **Elckerlyc** são BML *realizers* utilizados também para gerar comportamentos verbais e não verbais em agentes virtuais.

EMBR destaca-se por ser uma plataforma com um nível diferente no controlo de animações [3]. Apesar de utilizar a linguagem BML, os autores acharam necessário adicionar mais uma camada de abstração *EMRScript* para controlar as animações. Daí ser possível interpolar as animações de maneira que o utilizador considerar mais adequada. Esta camada está localizada entre o planeamento do comportamento e o *realizer*, tornando, assim, possível a implementação de BML através de uma tradução de BML em *EMRScript*.

Elckerlyc distingue-se com o fato de ser possível misturar uma simulação física com outro tipo de animação, como é o caso da captura de movimentos. Não é tão complexo a nível de programação, pois tem mais caraterísticas gráficas.

4.6 *Smartbody*

SmartBody é uma plataforma de criação de animações de humanos virtuais em 3D no tempo real [16]. Esta também foi desenvolvida pela Universidade que criou o *Vhtoolkit*. Foi desenvolvido em *C++* e pode ser controlado através de uma interface utilizando a linguagem *Python*. Trata-se de uma interface muito simples de usar, comparativamente com os softwares anteriormente mencionados.

É baseado numa hierarquia de controladores de movimento, os quais são definidos através de algoritmos de animação, tais como: mistura de *frames*, captura de movimentos, entre outros.

Atualmente suporta oito tipos de comportamentos: postura, olhar, expressões faciais, animações parciais ou de corpo inteiro, eventos, interrupções, notificações e discurso.

Na Figura 4.1, está exemplificado um esquema da plataforma *Smartbody*.

Os componentes de comportamento que fazem parte do *Smartbody* podem ser divididos em duas partes: o gerador de comportamento, que analisa todos os pedidos de BML, e o motor de controladores de movimento que avalia a hierarquia responsável das ações para cada agente virtual, sendo o último o mais importante, pois, para cada *frame* das animações, o motor combina todos os controladores que são invocados e faz gerar as rotações e translações necessárias no esqueleto.

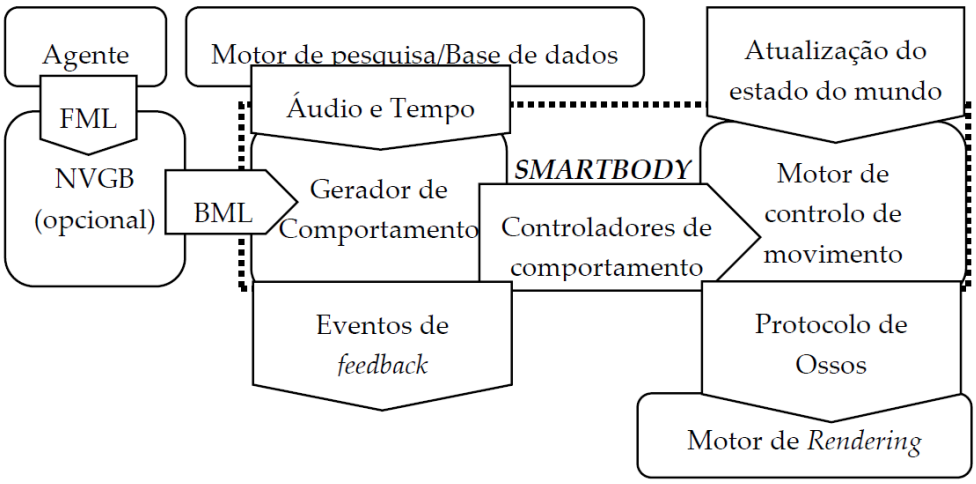


Figura 4.1: Esquema da plataforma *Smartbody*.

O resultado final é enviado através de um protocolo de rede para o sistema de *rendering* (processo de gerar uma imagem através de um modelo), este aplica a *deformable mesh* (deformação) no agente e, finalmente, é projetado para o cenário.

4.6.1 Exemplo de utilização

Existem várias formas de executar esta aplicação que é *standalone*, ou seja, autossuficiente. Descrevemos os primeiros passos no *Smartbody*, para que a possam acompanhar. Depois de o *download* ser feito, esta é a interface que se pode visualizar na Figura 4.2.

São disponibilizados alguns modelos na pasta*/examples* em ficheiros *Python*.

Ao carregarmos, por exemplo, o ficheiro *AddCharacterDemo.py*, é criado um *SmartBody* do sexo masculino, o *Brad*, Figura 4.3, com as capacidades principais, tais como:

- Locomoção;
- Discurso;
- Expressões faciais.

Para podermos controlá-lo não existe uma única forma. Podemos usar a interface que nos é fornecida pela janela BML *creator* ou escrever diretamente na linha de comandos.

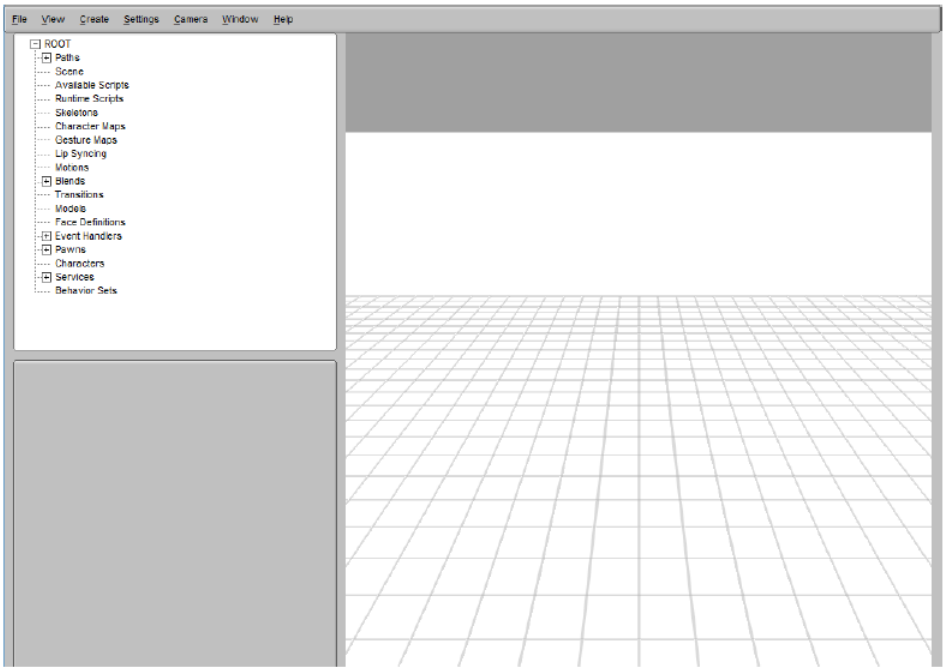


Figura 4.2: Interface *sbgui*.

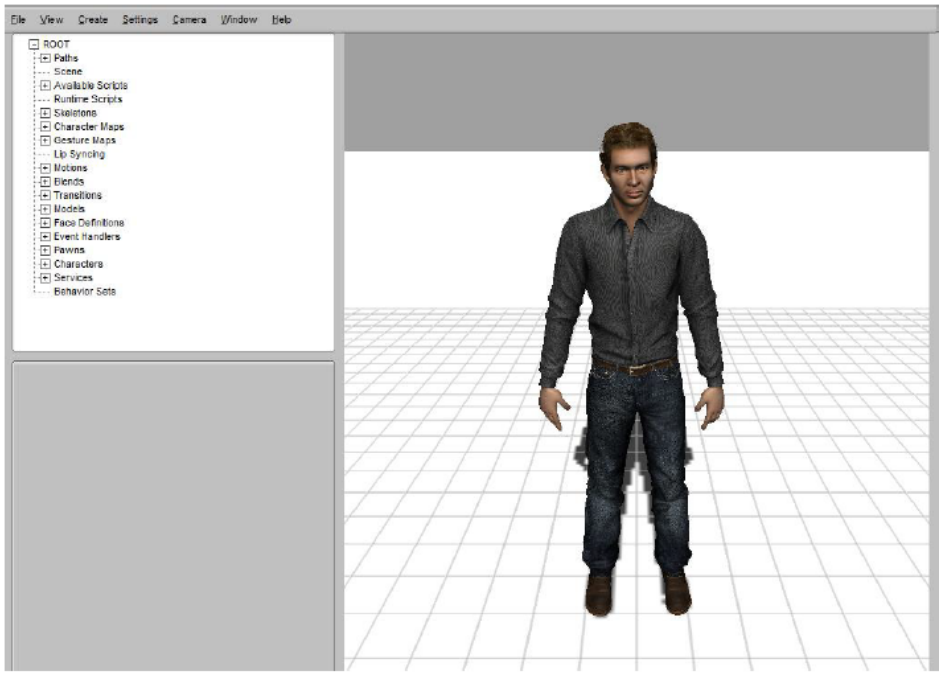


Figura 4.3: *Smartbody Brad*.

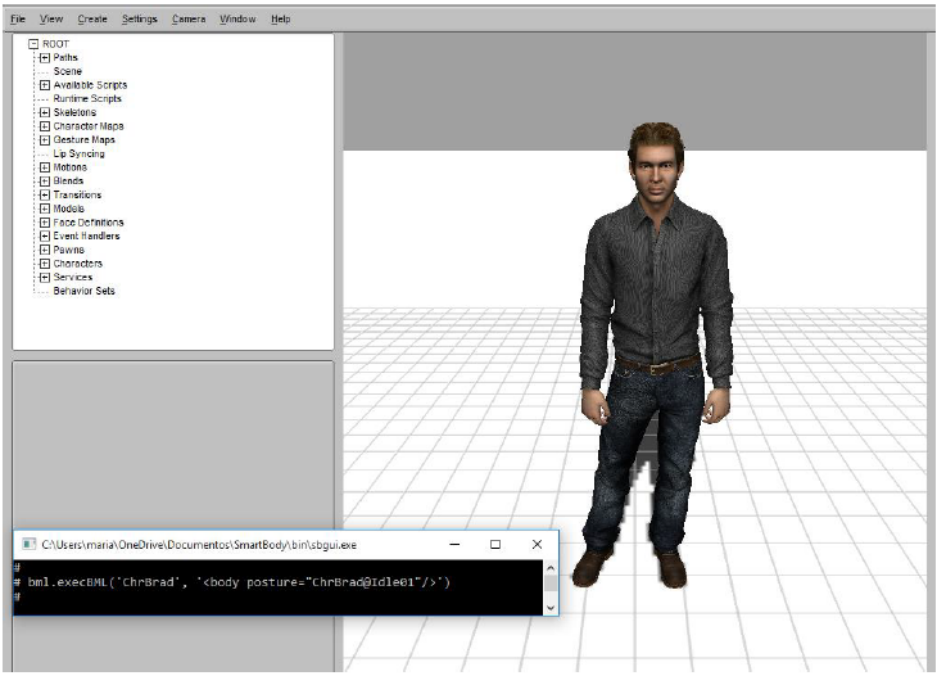


Figura 4.4: Comando BML para postura natural do agente.

Em primeiro lugar, e utilizando a linha de comandos, vamos usar a animação *ChrBrad@Idle01* para a postura natural do agente, Figura 4.4. Para tal escrevemos a seguinte instrução:

```
bml.execBML('ChrBrad', '<body posture="ChrBrad@Idle01"/>')
```

Em seguida, na Figura 4.5, para movimentarmos a cabeça e a coluna, podemos construir um alvo, um objeto. Para este ser construído necessita de uma posição. Logo, é só apontarmos o olhar do agente para esse alvo. Podemos visualizar este movimento com os seguintes comandos:

```
obj = scene.createPawn('obj1')
obj.setPosition(SrVec(0.7,0.8,0))
bml.execBML('ChrBrad','<gaze target="obj1"/>')
```

Também é possível mover o objeto clicando com o rato e pressionando a tecla 'W'. Por outro lado, também podemos usufruir da janela *Motion Editor*, Figura 4.6, na qual é permitido executarmos e examinarmos todas animações existentes.

Para um melhor conhecimento da aplicação, temos uma lista de exemplos de *scripts* em *Python*.

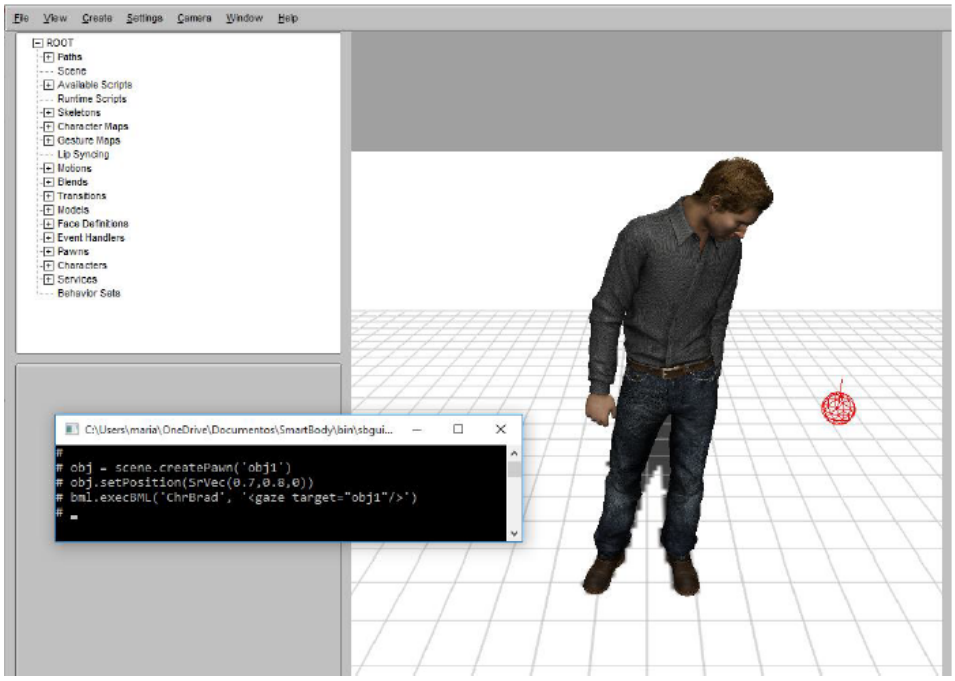


Figura 4.5: Controlo do movimento da cabeça e da coluna.

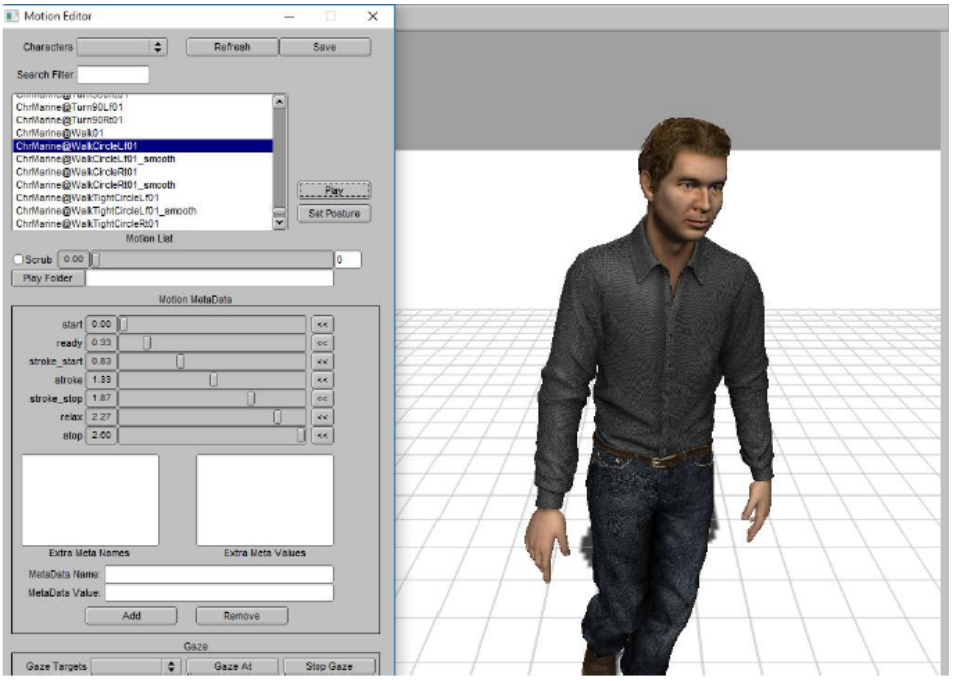


Figura 4.6: Execução da animação de andar aos círculos.

No anexo B, temos todas as ilustrações que demonstram cada exemplo. Passo, então, a enumerá-las e descrevê-las:

- ***AddCharacterDemo.py*** e ***AddCharacterDemoRachel.py*** são exemplos onde são adicionados os agentes *Brad* e *Rachel*, respetivamente. Estes são os agentes do *Smartbody*, por padrão. Aqui estão incluídas todas as suas capacidades, discurso, animação da cara, a locomoção, entre outras;
- No ficheiro ***BlendDemo.py***, é feita uma combinação de animações que estão a ser executadas ao mesmo tempo baseando-se em parâmetro. Aqui temos os *blends* que irão ser descritos mais à frente, a uma, duas e três dimensões;
- ***ConstraintDemo.py*** permite-nos ver como são utilizadas as diferentes restrições de animações com uma combinação de outros comportamentos;
- ***CrowdDemo.py*** é um exemplo de vários agentes a seguir uma direção e a se movimentarem para a mesma;
- ***EventDemo.py*** requer o uso de *Speech relay*, onde é necessário abrir o menu *Window* e depois selecionar *Speech relay*. Trata-se de um exemplo de como conectar eventos em animações e, em seguida, responder a esses mesmos eventos via *script*;
- ***FacialMovementDemo.py***, neste *script* ocupamo-nos das expressões faciais. Estão exemplificadas algumas expressões, como a tristeza, a alegria, o medo e a insatisfação;
- ***GazeDemo.py*** proporciona-nos a demonstração da capacidade de um agente virtual fixar um ponto e olhar para o mesmo, enquanto este se movimenta;
- ***GesturesDemo.py***, neste exemplo podemos visualizar gestos corporais;
- ***HeadDemo.py*** é um modelo de várias animações que permite o controlo da cabeça do *Brad*;
- ***LocomotionDemo.py***, é um exemplar de vários agentes a se movimentarem;
- ***OgreCrowdDemo.py*** é onde está representada uma multidão de carateres do Ogre;

- ***OgreDemo.py***, semelhante ao primeiro *script* do *Brad*, mas com o agente *Sinbad*;
- ***OnLineRetargettingDemo.py*** é um exemplo onde um agente redireciona o movimento de outro em tempo real;
- ***PerlinNoiseDemo.py*** é um modelo que se adiciona *noise* à animação;
- ***ReachDemo.py*** é onde se encontram vários agentes virtuais que tocam, agarram e olham para os objetos;
- ***SaccadeDemo.py*** é onde podemos configurar o movimento nos olhos;
- ***SpeechDemo.py*** requer novamente a utilização do *Speech Relay* e, ainda, a instalação do serviço *ActiveMQ* (instalação já efetuada na fase de instalação do *Virtual Human Toolkit*). Trata-se de um exemplo onde podemos efetuar a sincronização dos lábios e despoletar a capacidade de efetuar um discurso;
- E, finalmente, ***SteeringDemo.py*** é um exemplo de uma combinação da capacidade de se movimentar seguindo uma certa direção e ainda evitando obstáculos.

4.6.2 Construção de um cenário

A entidade principal que trata da criação de agentes virtuais e *querys* é o objeto *scene*.

E, para criar de raiz uma simulação, temos que seguir os seguintes passos que iremos citar:

Smartbody necessita de diferentes tipos de dados: o esqueleto é usado para definir a estrutura do agente, um movimento é usado para fazer animações, uma *skin mesh* é usada para deformar e fazer o *render* do mesmo e ainda existem os *scripts* que contêm comandos em *Python*. Para carregar todos estes dados utilizamos o objeto *asset manager*.

```
assetManager = scene.getAssetManager()
```

Com o comando abaixo, vamos adicionar um caminho de pesquisa *ChrBrad* para o tipo de *motion* de *assets* no *asset Manager*. Quando tentarmos carregar novos movimentos ou esqueletos, o sistema vai procurar o diretório: *ChrBrad*.

```
scene.addAssetPath('motion', 'ChrBrad')
```

O mesmo acontece para os comandos seguintes:

```
scene.addAssetPath("script", "behaviorsets")
scene.addAssetPath('mesh', 'mesh')
scene.addAssetPath('script', 'scripts')
```

Para poder carregar estes ficheiros só falta introduzir o comando seguinte:

```
scene.loadAssets()
```

O cenário tem uma escala por defeito de 0.01 centímetros. Com os comandos seguintes, podemos colocá-la em 1 metro e ainda configurar outros parâmetros do cenário e da câmara.

```
scene.setScale(1.0)
scene.setBoolAttribute('internalAudio', True)
scene.run('default-viewer.py')
camera = getCamera()
camera.setEye(0, 1.71, 1.86)
camera.setCenter(0, 1, 0.01)
camera.setUpVector(SrVec(0, 1, 0))
camera.setScale(1)
camera.setFov(1.0472)
camera.setFarPlane(100)
camera.setNearPlane(0.1)
camera.setAspectRatio(0.966897)
cameraPos = SrVec(0, 1.6, 10)
scene.getPawn('camera').setPosition(cameraPos)
```

O passo seguinte é contruir o esqueleto e os movimentos com os comandos abaixo. Em primeiro lugar, o ficheiro *zebra2-map.py* é executado para definir o tipo de esqueleto e movimentos de *zebra2*.

```
scene.run('zebra2-map.py')
zebra2Map = scene.getJointMapManager().getJointMap('zebra2')
bradSkeleton = scene.getSkeleton('ChrBrad.sk')
zebra2Map.applySkeleton(bradSkeleton)
zebra2Map.applyMotionRecurse('ChrBrad')
```

Em seguida, criamos o agente *ChrBrad* e depois atribuímo-lo ao esqueleto *ChrBrad.sk*, para definir a hierarquia deste. A função *createStandardControllers* serve para criar controladores de padrão que habilitam as capacidades essenciais do agente. Os seguintes comandos exibem o esqueleto como é visível na Figura 4.7.

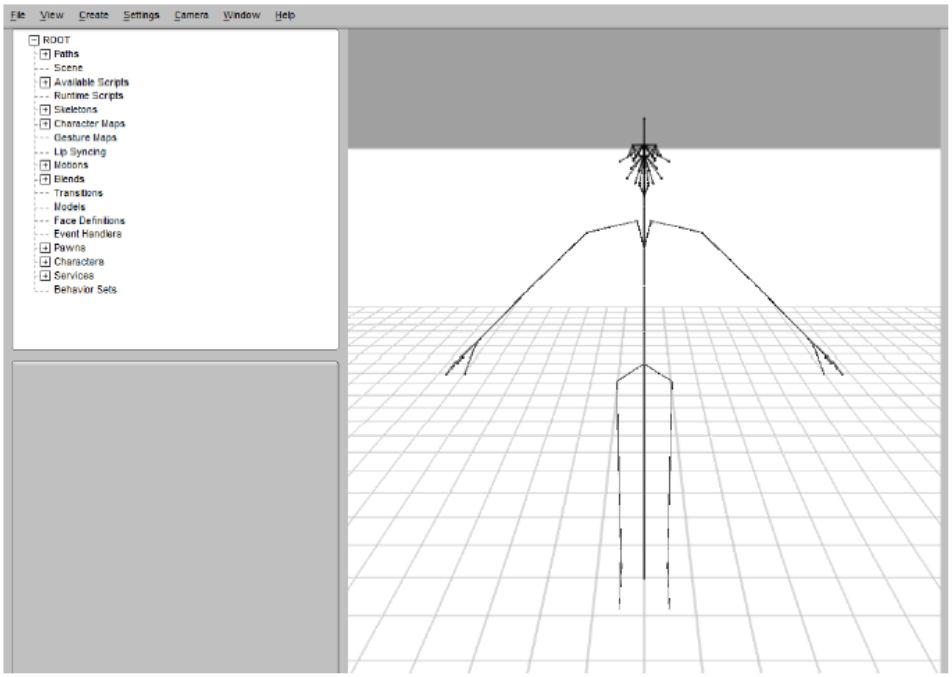


Figura 4.7: Esqueleto de *Brad*.

```
brad = scene.createCharacter('ChrBrad', '')
bradSkeleton = scene.createSkeleton('ChrBrad.sk')
brad.setSkeleton(bradSkeleton)
brad.createStandardControllers()
```

Para carregar o modelo 3D é necessário definir a *skin mesh* para o *ChrBrad*. Tal é possível com os comandos abaixo. Começamos por mudar a escala, em seguida definimos o atributo *deformableMesh* ao *ChrBrad* e, finalmente, para não mostrar apenas os ossos, usamos o atributo *displayType*, representado na Figura 4.8.

```
brad.setDoubleAttribute('deformableMeshScale', .01)
brad.setStringAttribute('deformableMesh', 'ChrBrad.dae')
brad.setStringAttribute("displayType", "GPUmesh")
```

4.6.3 Combinação de comandos BML

É possível combinar todo o tipo de comandos em BML. Só é necessário respeitar a sua sintaxe.

Na Tabela 4.3, podemos ver como se utilizam os atributos.

Tabela 4.3: Atributos pertencentes à BML.

Atributos	**Exemplo**
Gaze	**Olhar para o object1.** *gaze target='object1'*
Locomotion	**Deslocar-se para a posição (10,20).** *locomotion target='10 20'*
Head Movement	**Abanar a cabeça.** *head type='NOD'*
Idle	**Posição neutra.** *body posture='idling-motion-x'*
Animation	**Produz uma animação.** *animation name='motion-2'*
Gesture	**Aponta para o character2.** *gesture type=POINT' target='character2'*
Reach	**Agarra o object2.** *sbm: reach target='object2'*
Constraint	**Restringe o movimento da mão para a bola.** *sbm: constraint target='ball'*
Face	**Levantar as sobrancelhas.** *face type='FACS' au='1' side='both' amount='1'*
Speech	**Dizer: 'Hello, how are you?'** *speech type='text/plain' hello how are you? speech*
Eye Saccade	**Mexer os olhos automaticamente.** *saccade mode='LISTEN'*
Event	**Mandar um evento 3 segundos no futuro.** *sbm:event stroke='3' message='sbm echo hello'*

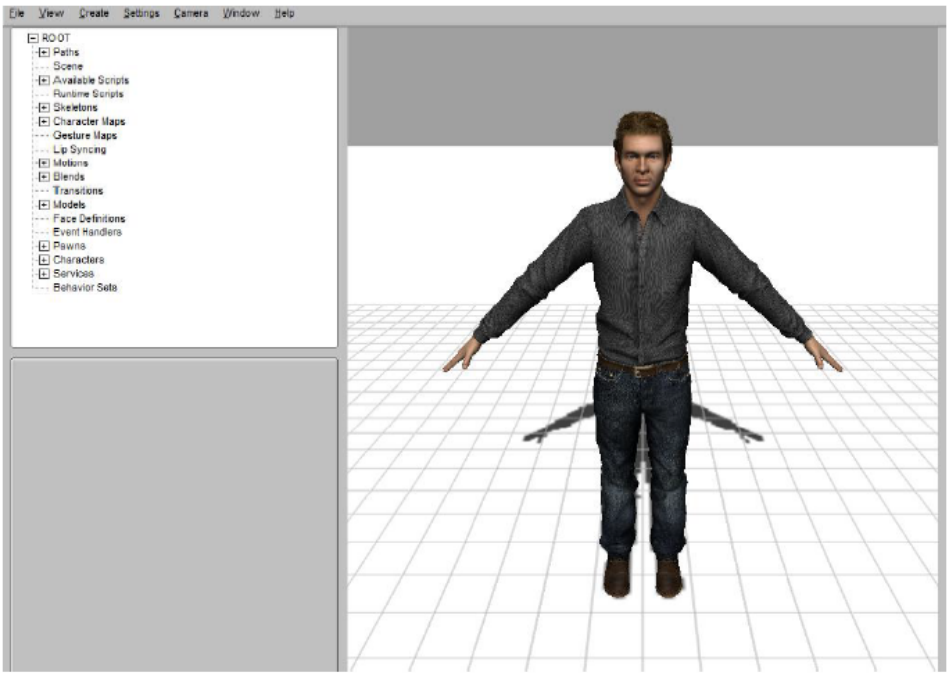


Figura 4.8: Agente *Brad*.

Utilizando os atributos, *face* e *head*, temos um exemplo onde a *Rachel*, Figura 4.9, está a levantar as sobrancelhas e, ao mesmo tempo, a abanar a cabeça.

Na Figura 4.10, podemos ver a combinação de duas animações em instantes diferentes utilizando o atributo *start*. Uma começa no instante 2 e a outra inicia-se no instante 4.

Com o comando assinalado na Figura 4.11, o agente desloca-se para uma determinada posição, tocando guitarra simultaneamente. Como a locomoção e esta animação não têm caraterísticas semelhantes, em vez de vermos a ação de andar conjugada com a animação, apenas é vísivel uma espécie de 'deslizamento' nos pés.

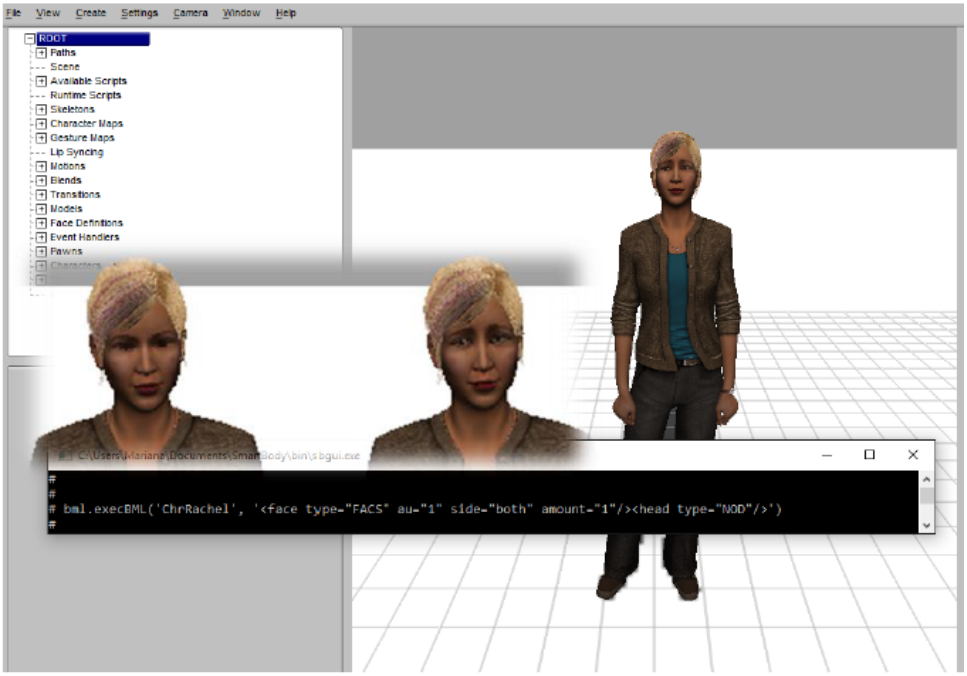


Figura 4.9: *Rachel* a abanar a cabeça e levantar as sobrancelhas.

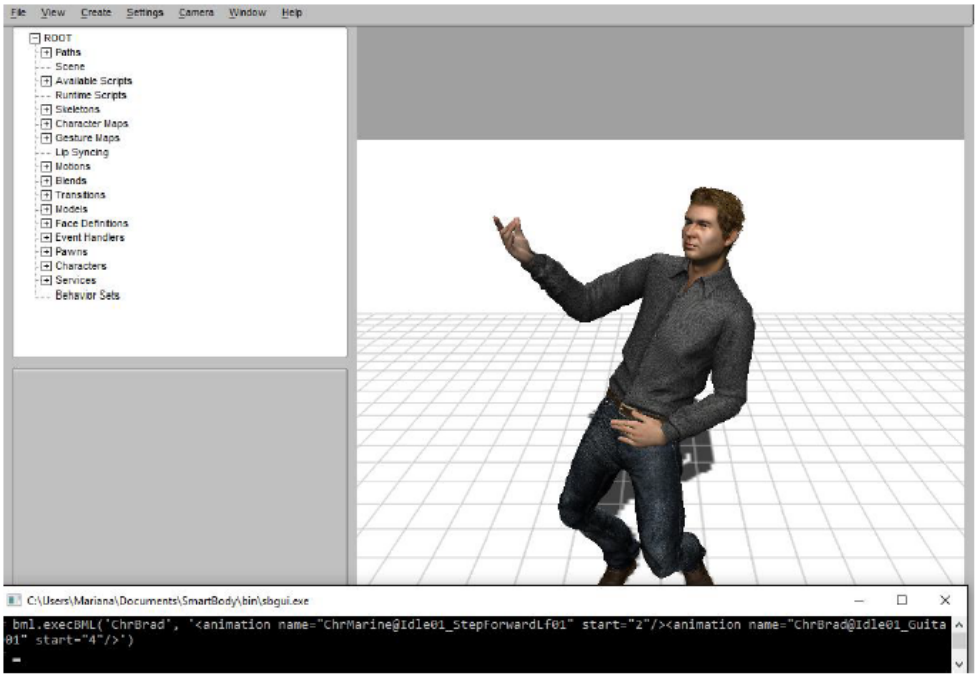


Figura 4.10: *Brad* a andar e tocar guitarra em diferentes instantes.

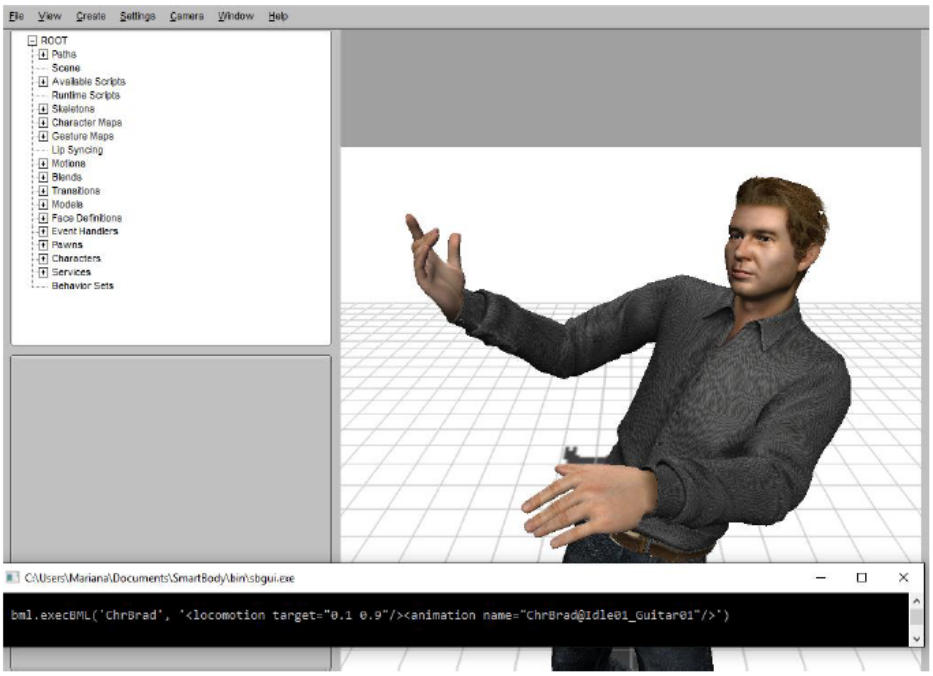


Figura 4.11: *Brad* a deslocar-se e tocar guitarra.

4.6.4 Atributo *locomotion*

A locomoção na aplicação *Smartbody* é feita através de um conjunto de movimentos [1]. São utilizadas 19 animações para uma boa interpolação dos movimentos. Cada animação contém dois ciclos de andar ou correr com movimentos para os lados, para a frente, para trás, a diferentes velocidades.

Cada animação é representada por:

$$M^i = \{P_{t0}i, \ldots, P_{tn}i\},$$

onde:

- $P = \{\tau, \theta\}$ representa a pose de um agente e é determinada pela sua posição de raiz, τ e por um conjunto de ângulos das juntas, $\theta = \{\theta_1, \theta_2, \ldots\}$;
- $(P_t)^i$ é a pose no tempo t para cada i-ésimo movimento;

Um subconjunto da animação M^i é definido no intervalo de tempo $[t_s, t_e]$, por:

$$(M^i(t_s \rightarrow t_e) = (P^i_{ts} \rightarrow P^i_{te})).$$

Note-se que a animação pode ser visualizada em sentido inverso, logo não existem restrições de inferioridade ou superioridade em relação ao tempo.

As animações M^i são configuradas manualmente em subconjuntos da forma:

$$\{M^i(t_1 \rightarrow t_2), \ldots, M^i(t_{n-1} \rightarrow t_n)\},$$

onde $t_k(k = \{1, \ldots, n\})$ indica o tempo da colocação do pé no chão.

Esta segmentação previne a mistura de animações que tenham diferentes ciclos da colocação dos pés e ainda reduz algum desfasamento dos mesmos que possa ocorrer em tempo real. Estas são parametrizadas em 3 dimensões usando a velocidade linear (v_f), velocidade de rotação (w), e velocidade para os lados (v_s). Deste modo, cada animação tem como parâmetros $p^i = (v_f^i, w^i, v_s^i)$ num espaço tridimensional.

Os parâmetros podem ser extraídos automaticamente dos modelos de animação, como a determinação da velocidade média da mesma. E ainda podem ser representados como coordenadas para ser possível efetuar uma interpolação.

Como estes parâmetros podem ser linearmente mapeados com os pesos das misturas, foram construídos tetraedros. Por outras palavras, dado um conjunto de animações M^i, são gerados tetraedros manualmente $V = (T_1, T_2, \ldots, T_v)$ que fazem a ligação com os pontos de parametrização. Por exemplo, uma ligação de um tetraedro com 4 pontos de parametrização é representada por:

$$T_j = (p^{j1}, p^{j2}, p^{j3}, p^{j4}).$$

Durante a execução, é possível obter novos parâmetros a partir dos já descritos anteriormente, $p^{'} = (v_f^{'}, w^{'}, v_s^{'})$. Podemos usar este novo parâmetro $p^{'}$ para averiguar se existe um tetraedro T_j em V que inclua $p^{'}$. Quando um parâmetro está dentro de um tetraedro, as suas coordenadas baricêntricas são todas positivas. Assim, podemos obter T_j calculando as coordenadas baricêntricas de $p^{'}$ para cada tetraedro em V. Assim que obtemos T_j, as animações associadas ao tetraedro $(M^{j1}, M^{j2}, M^{j3}, M^{j4})$, vão ser selecionadas para a interpolação das animações. O cálculo dos pesos das misturas das animações é baseado nas coordenadas baricêntricas de $p^{'}$ em T_j.

A animação resultante $M^{'}$ será a concatenação de:

$$\{M^{'}(t_1 \rightarrow t_2), \ldots, M^{'}(t_{k-1} \rightarrow t_k)\},$$

onde,

$$M^{'}_{t_k \rightarrow t_{k+1}} = \sum_{n=1}^{4} w_{jn} M^{jn}_{t_k \rightarrow t_{k+1}},$$

é o subconjunto das animações $(M^{j1}, M^{j2}, M^{j3}, M^{j4})$ de T_j.

Este método parece muito realista, mas depende sempre das animações inicialmente escolhidas.

Uma outra forma de locomoção é indicar um destino e fazer com que o sistema obtenha um caminho livre de colisões num ambiente dinâmico, ambiente no qual o agente poderá deparar-se com obstáculos. Nesta implementação é utilizado *SteerSuite* como sistema de *steering*. E funciona da seguinte maneira: dado um destino, o sistema calcula um novo trajeto, evitando os possíveis obstáculos e utiliza os parâmetros anteriormente referidos para a locomoção.

Na Tabela 4.4 temos os atributos da *tag locomotion* que estão programados na aplicação *SmartBody*.

4.6.5 Atributo *gaze*

Este atributo oferece várias possibilidades, tais como: a de um agente poder olhar para um objeto, para ele próprio, para outro agente e ainda para alguma parte do corpo em torno de um ambiente [1].

O conjunto das juntas predefinidas para esta função é o seguinte: *spine1*, *spine2*, *spine3*, *spine4* e *spine5*, *skullbase*, *face-top-parent*, *eyeball-left* e *eyeball-righ*.

No anexo C, encontram-se as ilustrações de todos os ossos que fazem parte do esqueleto.

Tal como a locomoção, este atributo *gaze* oferece uma variedade de opções a utilizar, descritas na Tabela 4.5.

Tabela 4.4: Atributo *Locomotion* no *Smartbody*.

Atributos	**Descrição**
type	Tipo de *locomotion*. (Ex: *example* ou *procedural* e por padrão é *basic*)
target	Desloca-se para o local indicado com coordenadas x, y e z ou até um objeto ou ainda com uma direção.
manner	Diferentes formas de *locomotion*. (Ex: *walk*, *jog*, *run* e *step*)
facing	A direção da cabeça, em graus, depois de acabar de andar.
speed	Velocidades em m/s, onde as menores que zero são ignoradas.
sbm:braking	O valor por padrão é 1.2. Se o valor for maior, o agente tende a desacelerar antes de chegar ao *target*.
sbm:follow	Segue o agente indicado.
proximity	O valor por padrão é 1.5. O quão perto o agente deve de ir até ao *target* antes de acabar de andar.
sbm:accel	O valor por padrão é 2. A aceleração do movimento.
sbm:scootaccel	O valor por padrão é 200. A aceleração de lado do movimento.
sbm:angleaccel	O valor por padrão é 450. A aceleração da velocidade angular.
sbm:numsteps	Número de passos.
start	Tempo que começa a *locomotion*.

Tabela 4.5: Atributo *Gaze* no *Smartbody*.

Atributos	**Descrição**
target	O destino para onde o agente tem de olhar.
sbm:target-pos	O local indicado por coordenadas x, y e z para onde olhar.
sbm:joint-range	Indica que parte do corpo é que olha e se vira para o destino.
direction	A direção que adota para olhar para o destino.
angle	Um ângulo através do destino indicado. O valor por padrão é 30 graus.
sbm:priority-joint	Indica qual a parte do corpo que está encarregue. Por padrão estão os olhos.
sbm:joint-speed	Define a velocidade para os olhos ou para o pescoço. Se só está indicado um valor, altera a velocidade do pescoço e se forem dois, altera a velocidade da cabeça e dos olhos. O valor por defeito é 1000.
sbm:handle	Nome dado para ser utilizado mais tarde.
sbm:joint-smooth	Valor médio de suavização.
sbm:fade-in	Intervalo para aparecer.
sbm:fade-out	Intervalo para desaparecer.
start	Em que segundo se inicia.

Capítulo 5

Estratégias para possíveis deficiências motoras

5.1 Introdução

Neste capítulo iniciamos com a descrição de algumas deficiências motoras na locomoção.

Em seguida são descritas algumas estratégias utilizando a aplicação *Smartbody* com o objetivo de assemelhar um agente com deficiências motoras aos tipos de deficiências na vida real inicialmente introduzidas. Estas estratégias vão desde a exploração de métodos já programados na aplicação à alteração direta de ficheiros de animações e esqueletos.

Segue-se a lista das estratégias que foram adotadas:

- Limitações entre um esqueleto virtual e um esqueleto real;
- Utilização de vários métodos implementados na plataforma *Smartbody*;
- Utilização de restrições rotacionais e posicionais;
- Alteração de limites no esqueleto virtual;
- Alteração do código dos ficheiros das animações;
- Utilização de atributo *blend* para juntar várias animações;
- Conversão dos dados de rotação em *Yaw*, *Pitch* e *Roll*.

E, por fim, é feita uma análise da estratégia que se adequa sob o ponto de vista realístico.

5.2 Descrição de deficiências motoras e incapacidades

Uma deficiência motora é uma condição física que priva um ser humano de ter uma vida normal. As deficiências podem ser restrições tanto temporárias como permanentes, mas são, sem dúvida alguma, consideradas como limitativas na rotina diária dos seus portadores.

Não existem deficiências iguais. Tal como existem diferentes formas de caminhar, existe uma diversidade de deficiências motoras. Dentro deste grupo existem as deficiências musco-esqueléticas. Estas revelam alterações do funcionamento e da mecânica da face, cabeça, pescoço, tronco e membros. Foi este grupo de deficiências que aqui foi testado.

Em relação às incapacidades, vamos focar-nos apenas nalguns tipos anormais de locomoção [10].

- **Marcha Antálgica** - Trata-se de uma marcha consequente de uma lesão na bacia, quadril, joelho, tornozelo ou pé. Verifica-se em apenas um dos lados e aquele que está afetado tende a ter uma base de apoio mais curta, na fuga à dor. Relativamente ao equilíbrio, é claro que este vai ser comprometido. A passada vai ser menor, tal como a velocidade e a cadência;

- **Marcha Artrogênica** - Decorre de uma deformidade no quadril ou no joelho. A consequência pode ser visível na elevação da bacia, ao fletir o tornozelo mais que o normal, devido ao joelho ou ao quadril rígido. Com isto, todas as fases de locomoção vão ser menores;

- **Marcha Atáxica** - Surge quando a pessoa não tem uma boa coordenação. Este tipo de locomoção evidencia-se no indivíduo que não tem nenhuma sensibilidade nos pés, o que faz com que os movimentos sejam exagerados;

- **Marcha do Glúteo Máximo** - Como o nome indica, esta é a consequência do glúteo máximo. Este tipo de locomoção resulta na inclinação do corpo para trás;

- **Marcha de Trendelenburg ou Marcha do Glúteo Médio** - A caraterística deste tipo é a inclinação do corpo para os lados, para que o centro de gravidade esteja sobre o apoio. A consequência é a falta de força nos músculos do glúteo médio;

- **Marcha Hemiplégica** - Verifica-se quando existe uma paralisia nos membros inferiores;

- **Marcha Parkinsoniana** - É uma marcha onde o portador da deficiência arrasta os pés. O comprimento da passada é mais pequeno e rápido. Poderá haver uma inclinação do corpo para a frente com a tentativa de andar com mais velocidade do que o normal;

- **Marcha do Pé Caído** - É quando existe uma fraqueza ou até mesmo uma paralisia nos pés. A pessoa tende a levantar o joelho para evitar que o pé se arraste.

5.3 Implementação e estratégias

Os objetivos principais na implementação das deficiências visaram a ideia realista, ou seja, aquela em que o agente virtual tivesse movimentos naturais, adotasse uma postura correta e na qual fosse possível a verificação da dinâmica do centro de gravidade, quando o agente efetuasse os movimentos. Depois das estratégias serem testadas, também foi feita uma avaliação tendo em conta os objetivos descritos.

5.3.1 Física e centro de gravidade

O centro de gravidade de um ser humano pode ser visto como o único ponto de concentração onde todas as partículas estão igualmente distribuídas. Pode variar com o peso, a altura e a estrutura corporal de cada ser humano. Este, deixa de ser estático quando está em movimento. Ou seja, quando os segmentos do corpo tomam outra orientação (andar, correr, saltar, entre outros, ...). Na aplicação *Smartbody* o centro de gravidade do agente é feito pela associação da massa e do momento de inércia de cada junta na hierarquia do esqueleto.

É necessário ter em conta que a densidade do corpo humano na aplicação *Smartbody* é cerca do valor 1.0 kg/m^3, e a partir daí, podemos efetuar o cálculo da seguinte forma: atribuir uma massa às juntas e um ponto médio, somar todos os pontos médios e multiplicar pela massa correspondente e o resultado final será a média.

5.3.2 Limitações esqueleto virtual *vs* esqueleto real

Antes de iniciar todo o processo de testes, foi feita uma questão que poderia à partida ser uma solução para o problema apresentado. Será que é possível inserir um esqueleto real na aplicação? Existe uma série de limitações nesta questão.

Como vamos verificar ao longo deste capítulo, os agentes são definidos inicialmente por um esqueleto padrão. Todas as rotações são feitas invocando uma ou mais juntas em particular. Vejamos um caso particular, se o objetivo fosse ter um agente a dobrar a língua o que teríamos de fazer era adicionar uma junta à língua, porém todos nós sabemos que a língua é uma parte do corpo humano sem ossos. Felizmente, a aplicação *Smartbody* é muito vasta, e é possível utilizar vários tipos de esqueleto. Apenas com uma limitação: o esqueleto tem de ser o mais semelhante possível a um esqueleto humano.

O processo denomina-se por *Automatic Rigging*. Em primeiro lugar é necessário transformar a estrutura de um esqueleto num modelo 3D. Existem várias alternativas na execução deste passo, pode ser feito manualmente por um animador experiente, ou por via *web* e, ainda, podem ser utilizados *scans* 3D. Em seguida é efetuado um mapeamento das juntas.

5.3.3 Utilização do método Rotate

O método *Rotate* faz a rotação em relação aos eixos x,y e z da junta escolhida pelo utilizador. Para utilizar este método são necessários os seguintes requisitos: escolher uma animação, uma rotação e os ângulos para cada eixo.

Neste caso foi utilizada a animação em que o agente estica os braços à frente do corpo: *ChrBrad@Idle01-ArmStretch01*.

Com os comandos seguintes é feita uma rotação de *spine1* com os valores $x = 45$, $y = 0$ e $z = -30$.

```
motion = scene.getMotion("ChrBrad@Idle01_ArmStretch01")
motion.rotate(45, 0, -30, "spine1")
bml.execBML('ChrBrad', '<animation name="ChrBrad@Idle01_ArmStretch01"/>')
```

Na execução da animação com as alterações podemos observar, na Figura 5.1, uma transformação. Como foi aplicada uma rotação na junta quase central do agente, este não consegue fletir os joelhos de forma natural nem manter a mesma pose da animação original.

Num segundo caso, temos a animação: *ChrMarine@Idle01-StepForwardLf01*, na qual o *Brad* dá um passo para a frente, utilizando a perna esquerda.

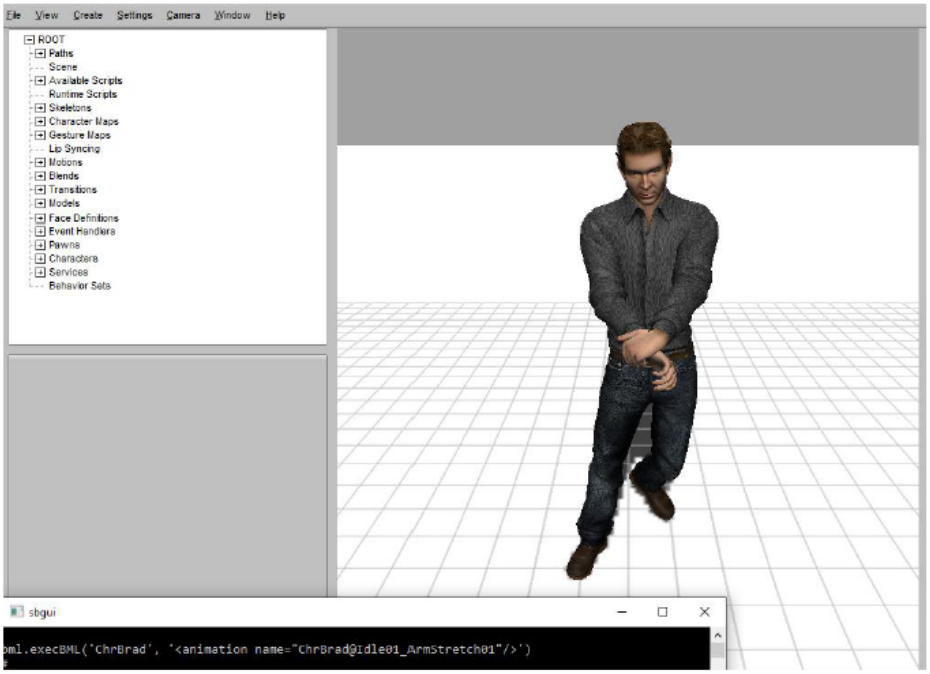


Figura 5.1: *ChrBrad@Idle01-ArmStretch01* com rotação.

Foi aplicada uma rotação da junta *spine1* com $y = 45$, na Figura 5.2, onde podemos ver a transformação no pé direito. Aqui acontece a mesma situação anterior pois a rotação não é aplicada no pé direito diretamente e o que está vísivel é um pequeno desvio no pé e na anca, devido ao fato de a junta *spine1* ser também muitíssimo importante e muito influente na marcha.

Em termos comparativos com os tipos de marcha incialmente descritos, o agente tende a inclinar o corpo para um dos lados, por algum tipo de lesão da bacia. (Marcha Antálgica)

```
motion = scene.getMotion("ChrMarine@Idle01_StepForwardLf01")
motion.rotate(0, 45, 0, "spine1")
bml.execBML('ChrBrad', '<animation name="ChrMarine@Idle01_StepForwardLf01"/>')
```

Num terceiro caso, Figura 5.3, temos a animação *ChrMarine@Idle01-StepSidewaysRt01*, onde o agente dá um passo para o lado, utilizando a perna direita. Aqui temos algumas semelhanças com as características da marcha Artrogênica. A deformidade está presente apenas no joelho. Nota-se uma pequena flexão no tornozelo, mas não é constatada nenhuma diminuição no comprimento da passada nem qualquer tipo de aumento na elevação da bacia.

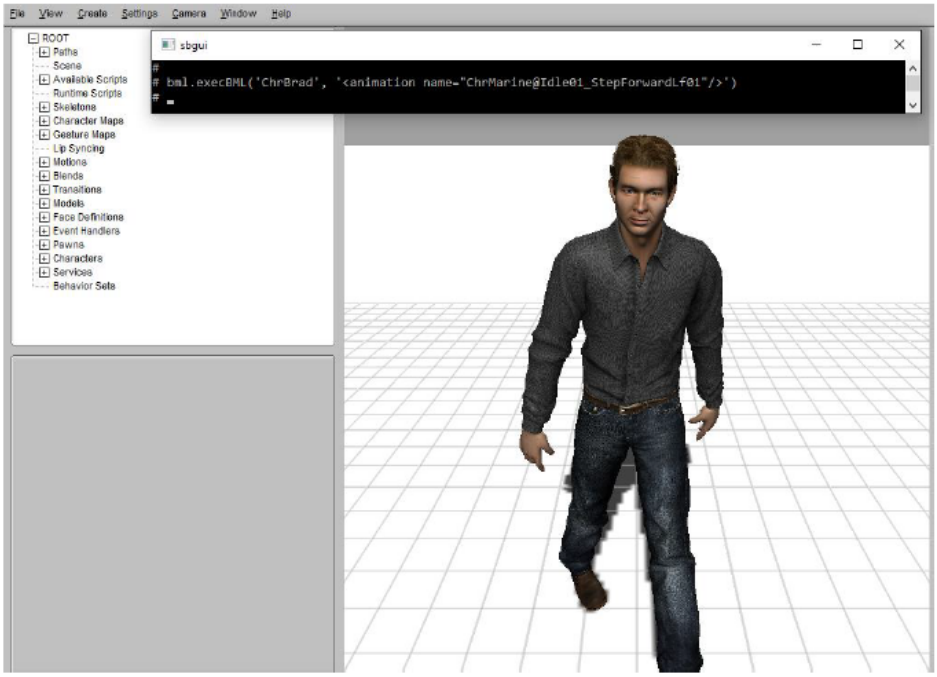


Figura 5.2: *ChrMarine@Idle01-StepForwardLf01* com transformação.

E, com os comandos abaixo, afetamos diretamente o joelho esquerdo como podemos observar na Figura 5.3.

```
motion = scene.getMotion("ChrMarine@Idle01_StepSidewaysRt01")
motion.rotate(0, 45, 0, "l_knee")
bml.execBML('ChrBrad', '<animation name="ChrMarine@Idle01_StepSidewaysRt01"/>')
```

O método Rotate é um método de utilização bastante simples. Contudo, nem sempre reproduz os resultados esperados. Quando aplicado diretamente na junta, parece ligeiramente forçado. É necessário ter em atenção se os ângulos aplicados irão funcionar bem em relação à junta escolhida. Este método não tem física aplicada, apenas faz a rotação indicada.

5.3.4 Utilização de restrições

As restrições aqui implementadas são classificadas como de dois tipos: rotacionais e posicionais. Em geral, é preciso ter em linha de conta que o agente virtual se limita a ser uma cópia de um ser humano e, por essa razão, aqui nas restrições, não se pode exagerar das limitações físicas.

Tendo como referência o ficheiro *ConstraintDemo.py*, foi adicionado ao *Brad*, que é possível ver mais à direita, que, para além da restrição comum a

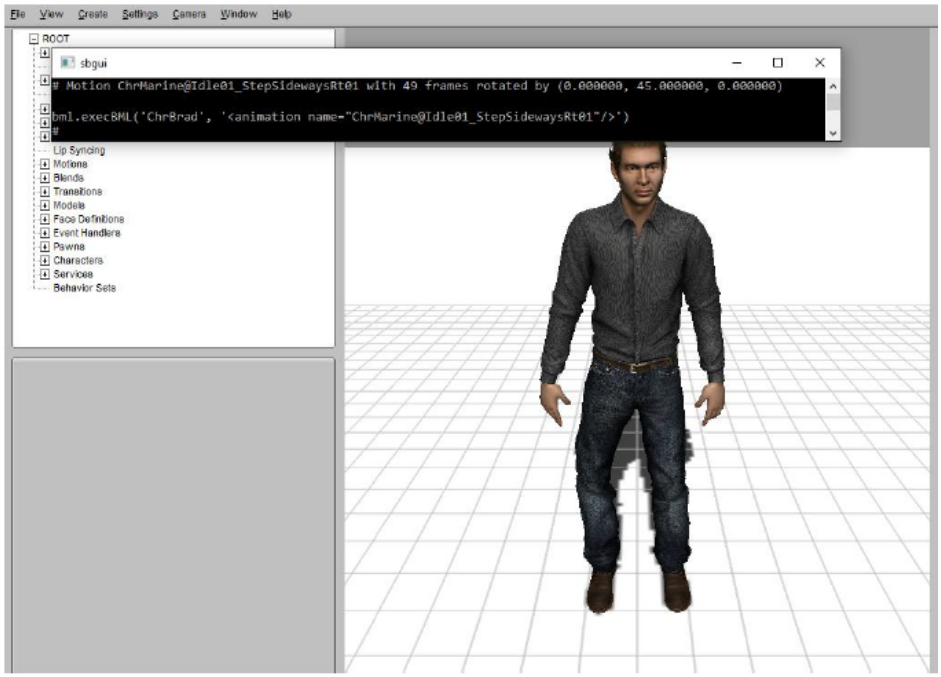


Figura 5.3: *ChrMarine@Idle01-StepSidewaysRt01* com transformação.

todos de olhar sempre para a bola em movimento, tem ainda uma restrição no pé e no pulso.

É possível verificar que ele se desloca até um certo destino sem movimentar um dos pés e o pulso, que estão presos a dois objetos. O código que faz a restrição do *Brad* que se encontra mais à direita é o seguinte:

```
bml.execBMLAt(2, 'ChrBrad3', '<sbm:constraint effector="r_wrist"
sbm:effector-root="r_sternoclavicular" sbm:handle="cbrad3"
target="pawn0" sbm:fade-in="0.5"/>')

bml.execBMLAt(2, 'ChrBrad3', '<sbm:constraint effector="l_toe"
sbm:effector-root="l_hip" sbm:handle="cbrad3" target="pawn1"
sbm:fade-in="0.5"/>')
```

Onde *sbm:constraint effector* é a restrição que vai ser aplicada e *sbm:effector-root* é a junta base que vai ter influência na restrição.

sbm:handle é uma referência que pode ser utilizada mais tarde e *sbm:fade-in* é o tempo em que a restrição se mistura na animação.

Por fim, ainda tem de ter um *target* que, neste caso, são os objetos presos ao pulso e ao pé.

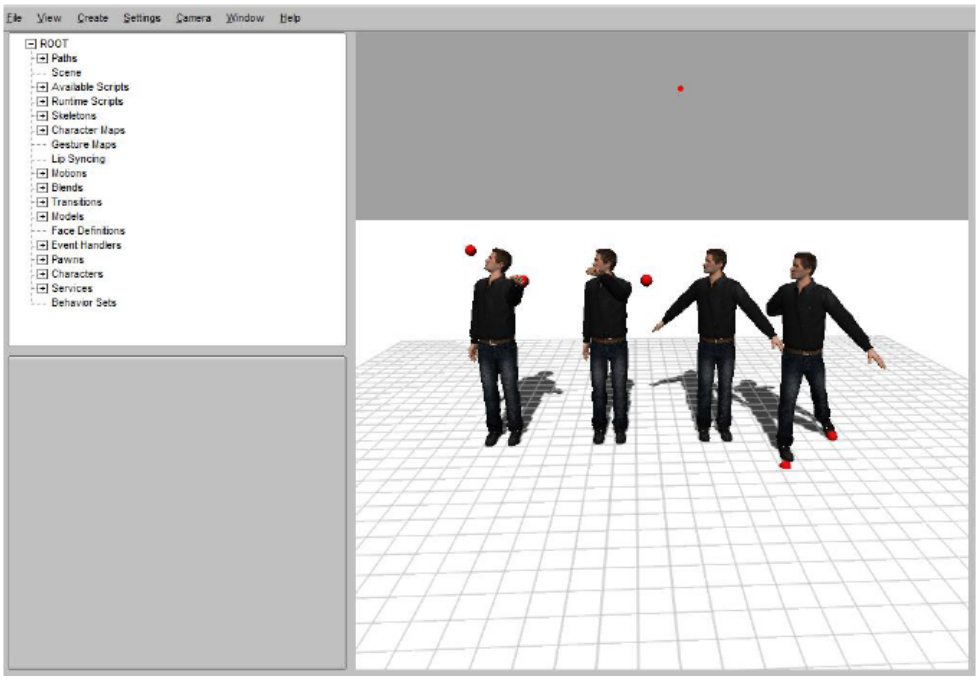


Figura 5.4: Restrição no *Brad* mais à direita.

Para podermos ver apenas um agente com a restrição de não poder andar e ficar preso a um objeto, temos que acrescentar, ao código, o acesso ao ficheiro que dá a possibilidade de o agente andar (*locomotion engine*). Temo-lo exemplificado na Figura 5.5.

É necessário ter em conta vários aspetos na utilização das restrições: não podem ser ultrapassadas as distâncias dos braços nem das pernas. Na colocação dos objetos que delimitam o movimento do agente, não podem se afastar muito da envergadura do mesmo.

Estes objetos, os *targets*, são obrigatórios no uso das restrições. Esta estratégia acaba por ser mais natural do que a anterior, como é possível de ser observado através das imagens.

Foi feita uma análise em pormenor do ficheiro das restrições que pode ser consultado no Anexo D.

5.3.5 Alteração de limites no esqueleto

Após uma análise pormenorizada do ficheiro referente às restrições, linha a linha, propusemo-nos efetuar uma comparação entre um agente com o

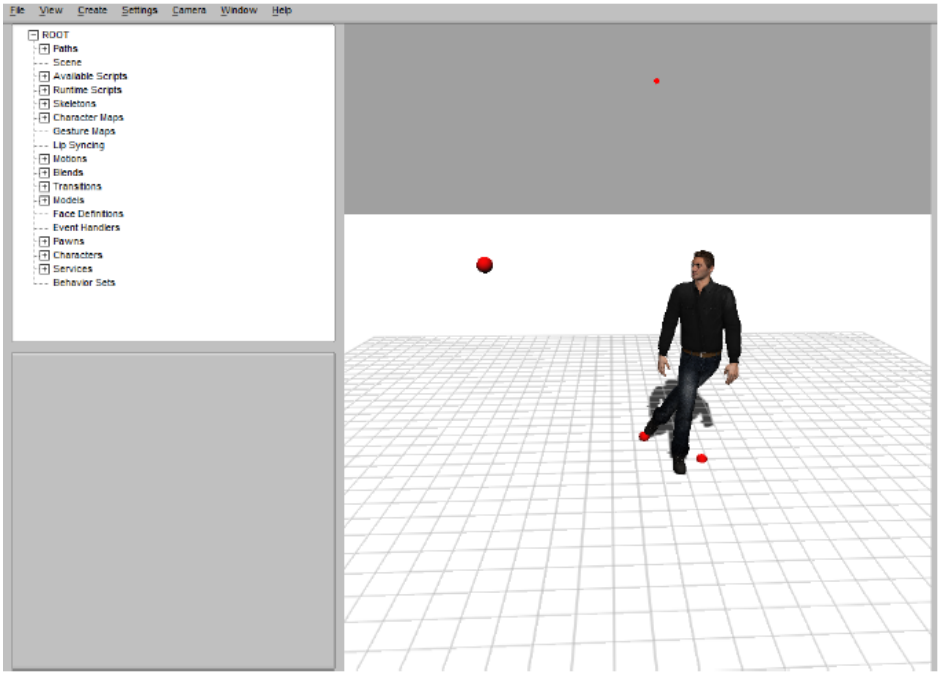


Figura 5.5: Exemplo de restrições modificado.

esqueleto 'normal' e outro com algumas restrições, mais precisamente ao agarrarem um objeto e olharem para uma bola em movimento.

O esqueleto está num formato *.sk* e contém toda a hierarquia dos ossos. É permitido usar diferentes tipos de esqueleto, só que, antes da execução, é necessário haver um mapeamento para o esqueleto padrão *Smartbody*.

No ficheiro *common.sk* podemos encontrar todos os ossos e como estes estão definidos. Este ficheiro é chamado em *BehaviorSetReaching.py* e aqui são chamados todos os movimentos referentes a *reaching* com o esqueleto definido em *common.sk*.

Para elaborar essa alteração e para não perdermos informação original, foi necessário criar novos ficheiros: *common2.sk* e *BehaviorSetReaching2.py*. Ao acrescentarmos mais um *BehaviorSetReaching2.py* tornou-se necessário ir ao ficheiro *default-behavior-sets.py* e adicionar-lhe as seguintes linhas de código:

```
behav = behaviorSetManager.createBehaviorSet("Reaching")
behav.setScript("BehaviorSetReaching2.py")
```

No ficheiro *BehaviorSetReaching2.py* houve a necessidade de alterar todas as instâncias *common.sk* para *common2.sk*. Neste último ficheiro foram feitas as alterações dos limites de alguns ossos.

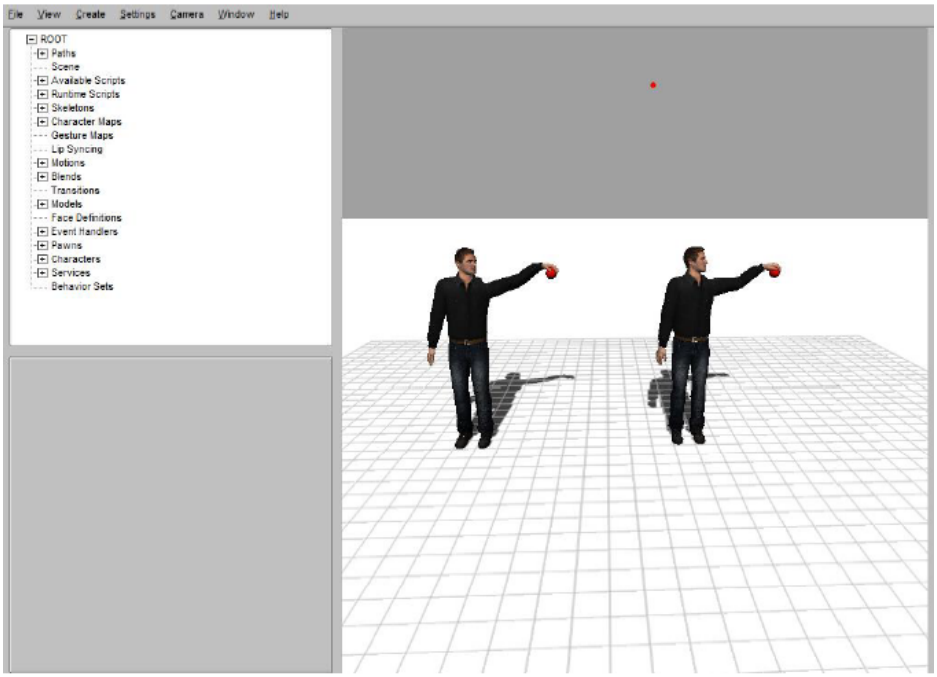


Figura 5.6: Cotovelo esquerdo com limites no eixo do X.

Nos seguintes exemplos, o agente *Brad* da esquerda é aquele que tem algumas restrições, desde os braços até à coluna.

Na Figura 5.6 temos uma extensão do cotovelo esquerdo com limites em relação ao eixo dos X de 0 a -180. Estes limites podem ser encontrados no ficheiro *common.sk*. Abaixo encontramos parte do código correspondente à restrição.

```
joint l_elbow
{ offset 28 0.430660 -0.030480
channel XPos 0 lim 0.000000 -180.000000
channel Quat
```

Na Figura 5.7, para além de ter limites no cotovelo esquerdo em relação ao eixo do X, como anteriormente, também surgem limites em relação ao eixo do Y de 0 a -180. No código que apresentamos a seguir, temos a alteração referenciada:

```
joint l_elbow
{ offset 28 0.430660 -0.030480
channel XPos 0 lim 0.000000 -180.000000
channel YPos 0 lim 0.000000 -180.000000
channel Quat
```

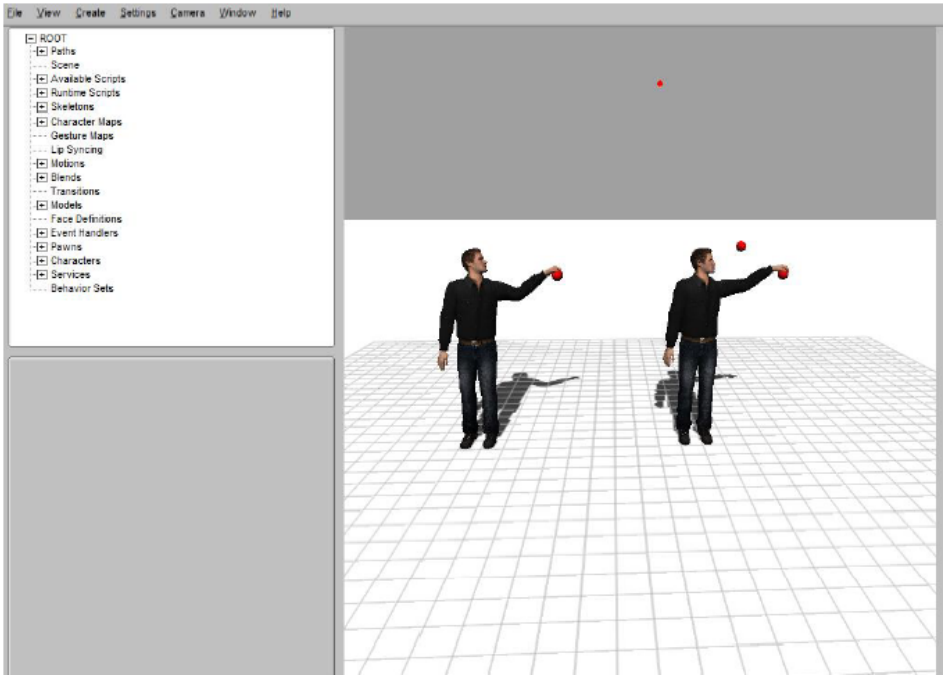


Figura 5.7: Cotovelo esquerdo com limites no eixo do X e do Y.

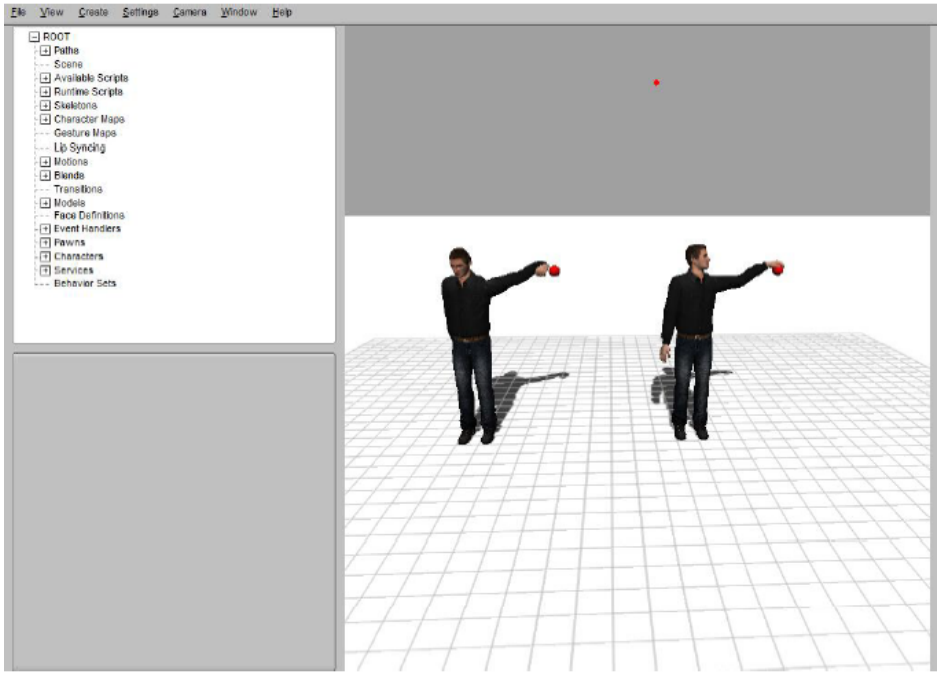


Figura 5.8: Antebraço esquerdo com limites no eixo do X.

Na Figura 5.8, podemos verificar alterações no antebraço. Os limites foram acrescentados em relação ao eixo do X de 0 a -30.

```
joint l_forearm
{ offset 15.998100 0.246060 -0.017420
channel XPos 0 lim 0.000000 -30.000000
channel Quat
```

Não existem muitas diferenças se alterarmos os limites do antebraço e do pulso. Aqui temos limites entre 0 e -10.

```
joint l_wrist
{ offset 11.998570 0.184540 -0.013060
channel XPos 0 lim 0.000000 -30.000000
channel Quat
```

Analisando a alteração dos limites do esqueleto, em algumas situações, fazia sentido pensar que alterar os limites dos canais seria a melhor solução. Mas, como é visível, também em alguns casos não é concluída a ação de *reach*. E, deste modo, não parece muito natural fazê-lo.

5.3.6 Utilização dos métodos: *setOffset*, *setUseRotation*, *setPrerotation* e *setPostrotation*

Como o próprio nome indica, estes quatro métodos atribuem valores às juntas. Em relação ao *setOffset*, trata-se de um método usado na construção da hierarquia do esqueleto. Delimita o *offset* da junta 'pai'.

O *setUseRotation* permite escolher se a junta usa ou não os canais de rotação.

Relativamente ao *setPrerotation* e ao *setPostrotation*, sabemos que colocam valores de rotação de pré e posterior na junta definida, respetivamente. São métodos com rotações adicionais às já existentes por *frame*.

Para a restrição no pescoço, foram criados dois objetos a se movimentarem por cima de cada agente. Como podemos ver na Figura 5.9, o agente com restrição (o da esquerda) tem uma certa dificuldade em acompanhar o objeto quando este se encontra atrás do seu corpo.

Para a restrição do cotovelo, a junta *l-sternoclavicular* é a 'responsável' pelo *reach* de um objeto. Utilizando o método *setOfsset* x,y,z a 0, o agente não é capaz de fechar a mão, como podemos concluir da observação da Figura 5.10.

Este tipo de deficiência, apesar de não ter sido muito explorado, tem semelhanças com uma deficiência neuro-motora, desenvolvida quer por uma

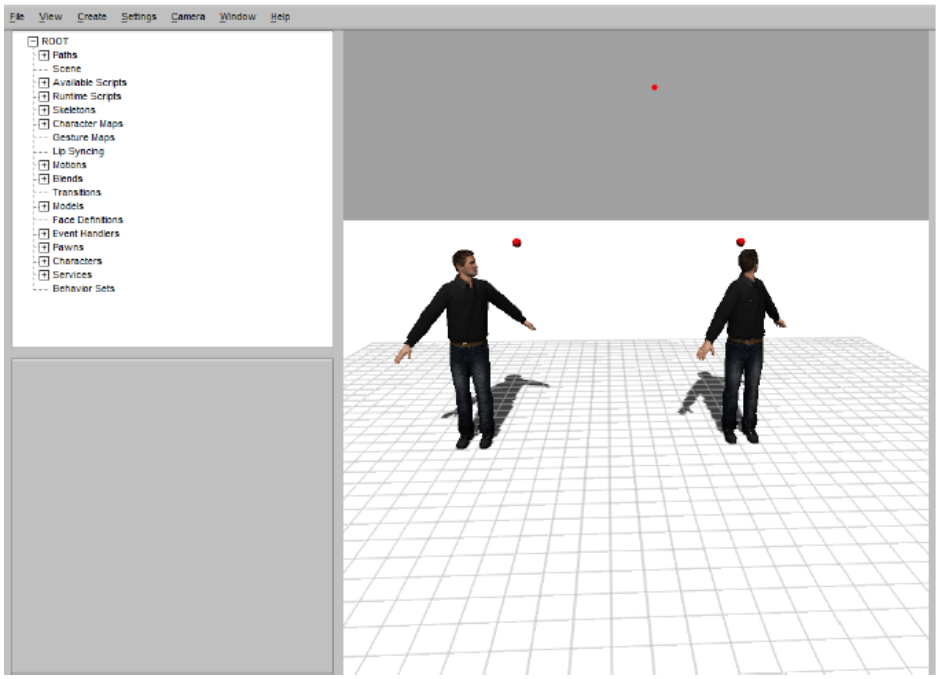


Figura 5.9: Restrição no pescoço.

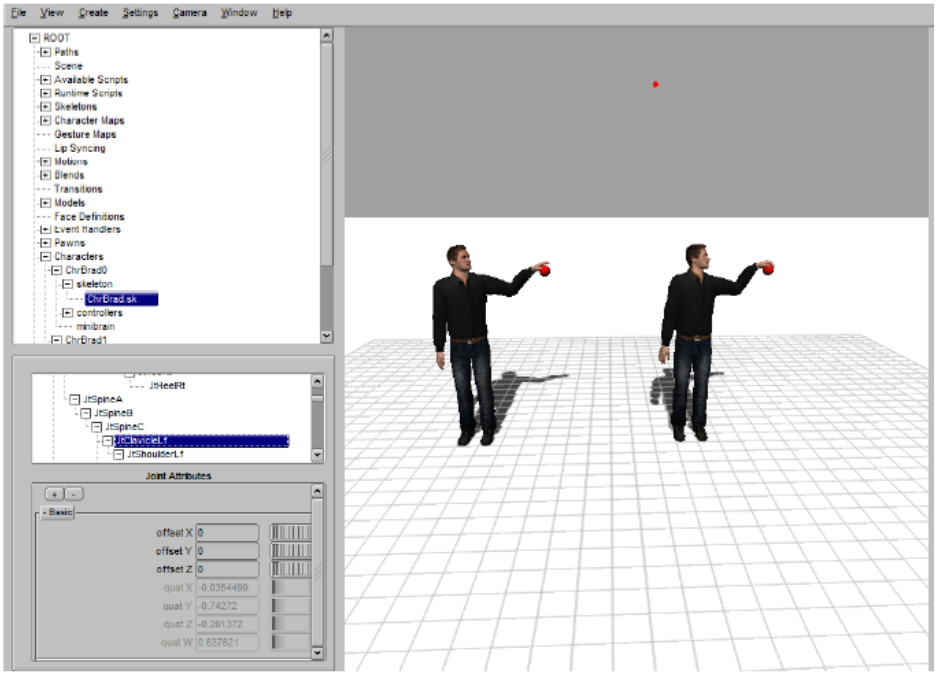


Figura 5.10: Restrição ao agarrar um objeto.

paralisia cerebral, quer pela existência de algum tumor no sistema nervoso central, situado na medula, afetando os membros superiores ou inferiores.

Se utilizarmos o método *setPrerotation* podemos analisar diversas situações. Quando os argumentos têm todos o valor 0, o membro relacionado com junta desaparece.

```
joint1.setPrerotation(SrQuat(0,0,0,0))
```

O método *setUseRotation* aplicado ao cotovelo também não é bem concretizado. O agente fecha a mão antes de agarrar o objeto, ficando esta ao mesmo nível do cotovelo, e por conseguinte, não consegue agarrá-lo.

```
character = scene.getCharacter('ChrBrad0')
jointName1 = 'l_elbow'
joint1 = character.getSkeleton().getJointByName(jointName1)
joint1.setUseRotation(False)
```

Voltando ao método *setPostrotation*, agora é possível verificar uma certa restrição no cotovelo, quando o agente agarra o objeto, Figura 5.11. Esta assemelha-se a uma incapacidade de dobrar o cotovelo (flexão do cotovelo). [14] Trata-se de uma lesão temporária para a qual há cura na maior parte dos casos. Outro exemplo que delimita os movimentos naturais de um cotovelo é a artrite.

Utilizando código abaixo descrito, podemos verificar a restrição abordada anteriormente:

```
character = scene.getCharacter('ChrBrad0')
jointName2 = 'l_shoulder'
joint2 = character.getSkeleton().getJointByName(jointName2)
joint2.setPostrotation(SrQuat(1,0,0,0))
```

Se utilizarmos o mesmo método para o joelho enquanto o agente está a caminhar, podemos verificar uma semelhança com a marcha atáxica. O agente parece não ter nenhuma sensibilidade nos pés e, em consequência, temos um movimento bastante exagerado, a abdução da perna esquerda, onde o pé se aproxima da linha mediana do corpo, como se poderá verificar na Figura 5.12.

Este exemplo também é idêntico à marcha do pé caído, cujo traço distintivo é o arrastamento de um dos pés. O movimento não é, porém, tão natural, dado que o agente não se desequilibra e mantém uma postura semelhante à do agente que não apresenta nenhuma restrição.

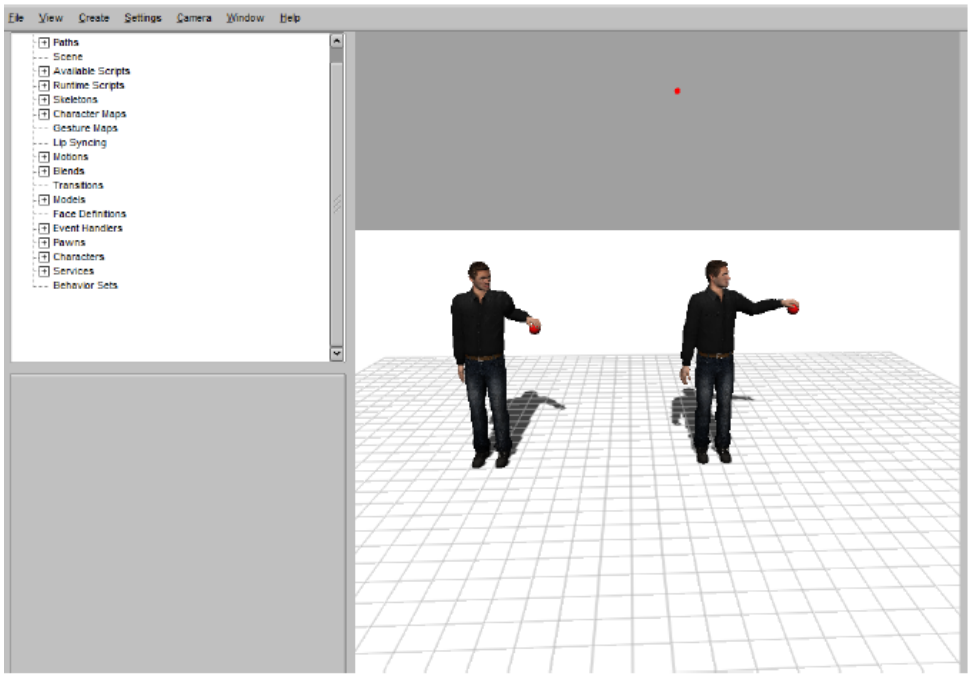


Figura 5.11: Restrição do cotovelo.

Com os comandos seguintes, temos uma pós-rotação do joelho esquerdo em relação aos eixos x e y.

```
character = scene.getCharacter('ChrBrad0')
jointName2 = 'l_knee'
joint2 = character.getSkeleton().getJointByName(jointName2)
joint2.setPostrotation(SrQuat(1,1,0,0))
```

Seguimos para uma restrição no pescoço, Figura 5.13, e observamos que o agente tenta seguir o movimento, mas este é travado, a certa altura. Existe uma série de incapacidades para justificar esta limitação, desde a perda de mobilidade no pescoço, à artrite e à degeneração dos nervos em volta do pescoço, entre outras. Acaba por ser uma restrição natural, pois os ombros e todas as juntas que giram para o objeto também não seguem por completo o movimento. Aqui foi utilizado o método *setOffset*.

```
character = scene.getCharacter('ChrBrad0')
jointName4 = 'skullbase'
joint4 = character.getSkeleton().getJointByName(jointName4)
joint4.setOffset(SrVec(-2,-2,-2))
```

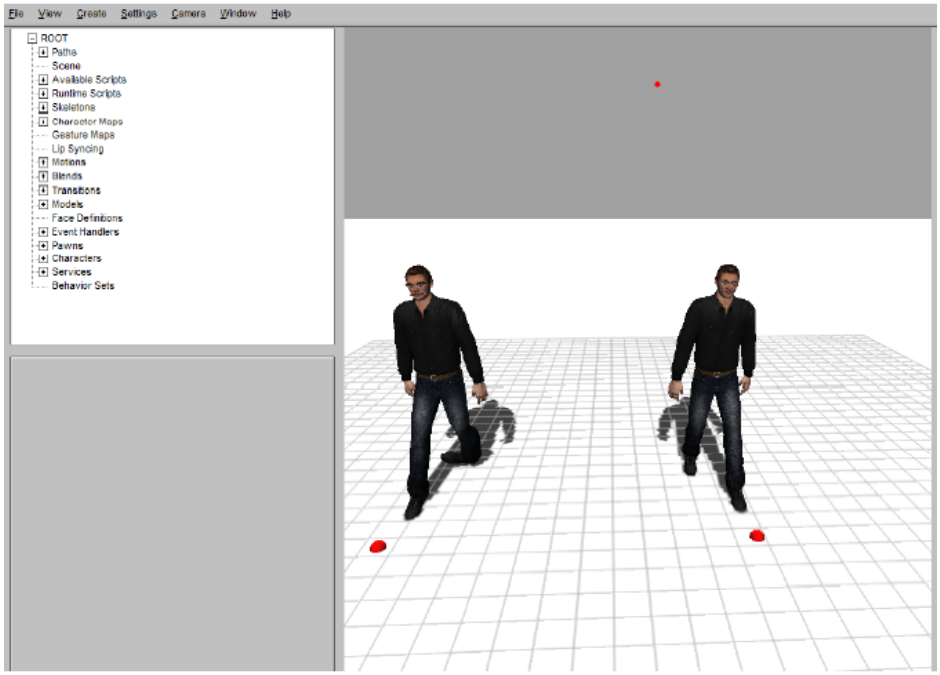


Figura 5.12: Restrição no joelho esquerdo.

Achamos que a incapacidade de dobrar o joelho também poderia ser outra restrição requerendo a utilização do método *setUseRotation*, Figura 5.14. Esta pode ser, de novo, comparada com a marcha artrogênica, pois o agente não consegue dobrar o joelho.

A perna é arrastada durante o movimento e há uma pequena inclinação da bacia para compensar o lado da deformidade.

```
joint2.setUseRotation(False)
```

A próxima restrição está na diferença ao agarrar um objeto que está abaixo da cintura. Aqui também foi utilizado o método *setUseRotation*, mas agora relativamente às juntas *spine1* e *base*. Podemos verificar, assim, que o agente não dobra a cintura na tentativa de o agarrar.

Assim, foram ultrapassados alguns limites, daí o fecho da mão antes de chegar mesmo ao objeto. Para uma postura mais natural, a solução seria colocar o objeto um pouco mais acima até que não seja ultrapassado o comprimento do braço em extensão.

```
joint1.setUseRotation(False)
joint111.setUseRotation(False)
```

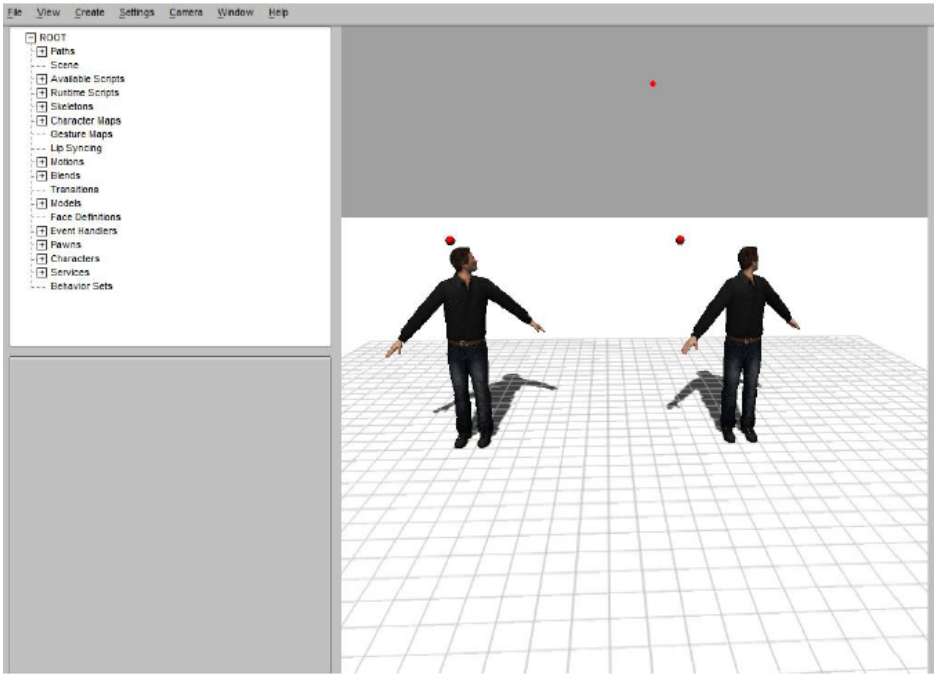


Figura 5.13: Restrição no pescoço.

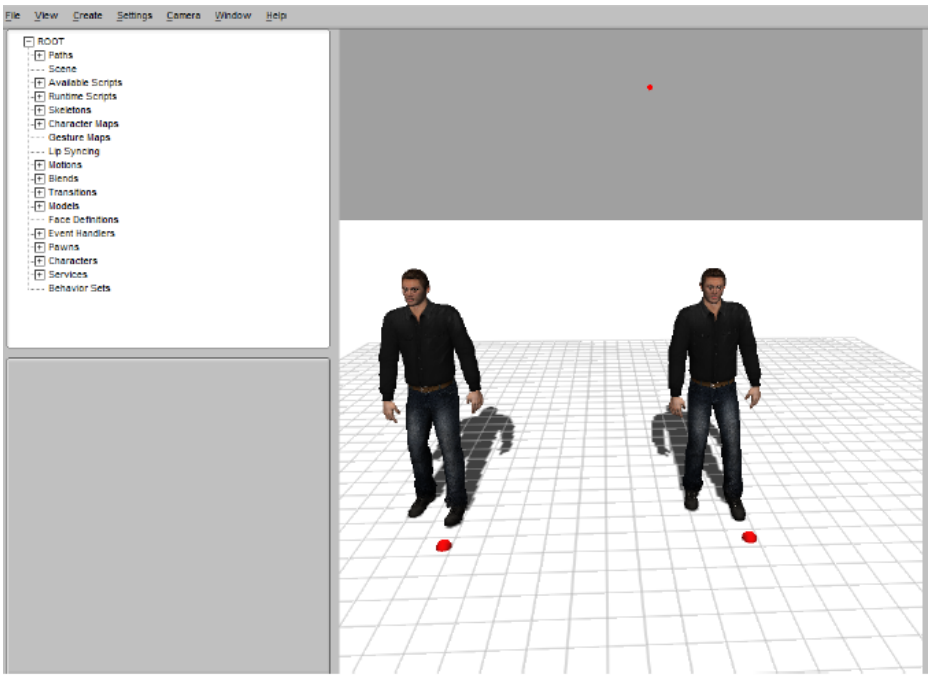


Figura 5.14: Restrição no joelho ao andar.

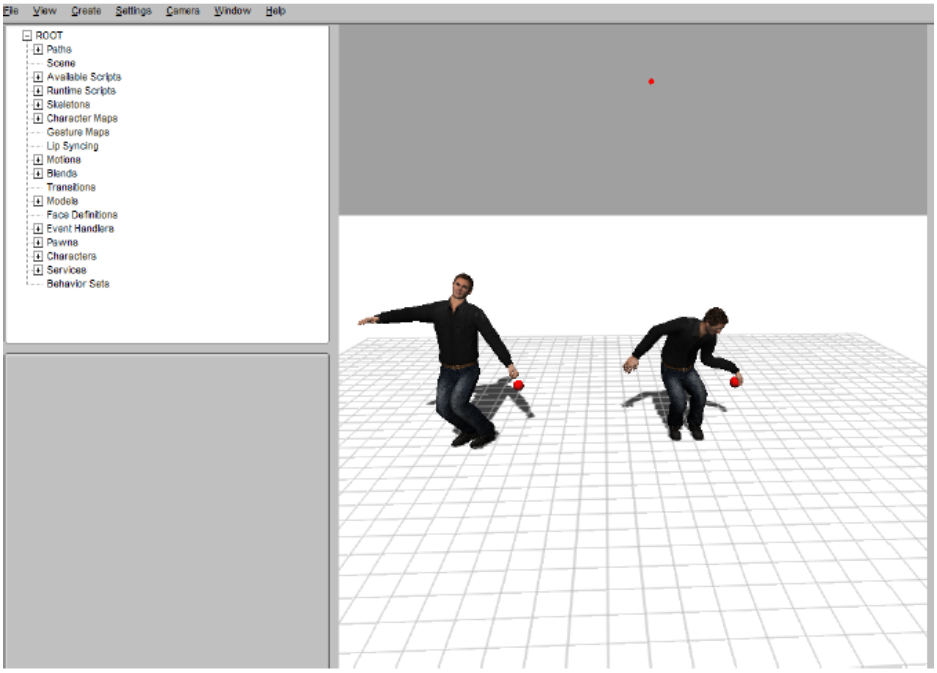


Figura 5.15: Restrição ao agarrar objeto abaixo da cintura.

Em todos os métodos analisados, o efeito produzido depende da junta que foi escolhida e das dependências da mesma. Por isso, não é correto afirmar que um método é melhor do que outro.

Nestes testes, o melhor resultado é o da flexão no cotovelo, Figura 5.11. A justificação é simples dado que o método é aplicado diretamente no cotovelo e este já não interfere com a hierarquia central do esqueleto, o que resulta na incapacidade de agarrar um objeto natural.

5.3.7 Alteração das animações

Uma outra abordagem ao problema consistiu em recorrer à alteração das animações, diretamente. As animações estão em formato *.skm* e, nestas, podemos encontrar os canais e as *frames* correspondentes.

Na Figura 5.16 temos um exemplo de uma animação em formato *.skm*. Nas duas primeiras linhas há comentários que se iniciam com o caracter #, com o intuito de dar alguma informação ao utilizador acerca do conteúdo do ficheiro. Neste caso, trata-se de uma definição de um comportamento em *skm*. Nas linhas seguintes é necessário identificar o nome da animação e os canais correspondentes ao esqueleto do agente.

```
# SKM Motion Definition
# FbxConverter.exe v1.0    Units: m

SkMotion

name "ChrMarine@Idle01_StepBackwardsRt01"

channels 201
JtRoot XPos
JtRoot YPos
JtRoot ZPos
JtRoot Quat

...

frames 41
kt 0.000000 fr -0.005248 -0.009586 -0.042503 0.005765 0.041955 0.012080 0.087830 0.0442
kt 0.033333 fr -0.002412 -0.011462 -0.046332 0.007247 0.046546 0.010951 0.093664 0.0462
kt 0.066667 fr 0.002721 -0.013192 -0.050462 0.015938 0.045175 0.007233 0.097943 0.0488
kt 0.100000 fr 0.008411 -0.015390 -0.055703 0.024684 0.037797 0.002788 0.101519 0.0511
kt 0.133333 fr 0.014467 -0.017693 -0.062162 0.039141 0.025411 -0.002773 0.104298 0.0524
kt 0.166667 fr 0.020889 -0.019393 -0.069720 0.063949 0.005131 -0.011907 0.097059 0.0451

...

ready time: 0.133333
strokeStart time: 0.333333
emphasis time: 0.700000
stroke time: 0.966667
relax time: 1.166667
```

Figura 5.16: Animação em formato *.skm*.

Neste exemplo estão definidos 201 canais. Entenda-se por canais uma forma de indicação da posição e da rotação para cada junta do esqueleto. Nos canais em que não são aplicadas rotações, só são definidas as posições x, y e z, pelos comandos $XPos$, $YPos$ e $ZPos$, respetivamente.

5.3.8 Quaterniões

É necessário ter atenção aos canais de rotação *Quat* (quaterniões). Os quaterniões são uma forma de representação de rotações. Normalmente são representados por 4 valores. Por motivos de otimização dos resultados, na aplicação *Smartbody* é feita uma conversão para 3 valores.

Sir William Hamilton (1843) criou os quaterniões [2] com a motivação de conseguir arranjar uma relação idêntica à dos números complexos com o plano. Inventou os quaterniões, um tipo de números hipercomplexos, para ter uma relação com o espaço tridimensional. É importante referir que estes surgiram antes dos vetores.

Estes foram subdivididos em duas partes. Uma parte real e outra imaginária. Um quaternião é da forma:

$$\hat{q} = q_0 + q_1 i + q_2 j + q_3 k,$$

ou da forma:

$$\hat{q} = q_0 + \vec{q},$$

onde q_0 é um número escalar e $q_1 i + q_2 j + q_3 k$ correspondente à parte imaginária, podendo ser substituída por um vetor $\vec{q}$. O conjunto dos quaterniões é representado por $\mathbb{H}$.

Os quaterniões têm as seguintes propriedades:

- $i^2 = j^2 = k^2 = -1$;
- $ij = k = -ji$;
- $jk = i = -kj$;
- $ki = j = -ik$.

Tendo em conta a notação anterior, seja $\alpha \in \mathbb{R}$ e $\hat{q}$, $\hat{p} \in \mathbb{H}$, seguem-se algumas operações elementares:

- $|\hat{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$;
- $\hat{q}^* = q_0 - \vec{q}$ (complexo conjugado);
- $|\hat{q}| = 1$ (quaternião unitário).

Podemos provar que os quaterniões, no corpo dos números reais, com as operações soma e multiplicação por um escalar, formam um *espaço vetorial*.

Um espaço vetorial sobre um corpo $\mathbb{K}$ é um conjunto V constituído por vetores. Este conjunto está munido das operações de adição e de multiplicação por escalar. A adição tem que satisfazer as seguintes propriedades:

1. Propriedade comutativa: $a + b = b + a, \forall a, b \in V$;
2. Propriedade associativa: $a + (b + c) = (a + b) + c, \forall a, b, c \in V$;
3. Elemento neutro: $a + 0 = a, \exists 0 \in V, \forall a \in V$;
4. Elemento oposto: $a + (-a) = 0, \forall a \in V, \exists (-a) \in V$.

Em relação à multiplicação por um escalar, as propriedades a ser satisfeitas são:

1. $\alpha(\beta a) = (\alpha\beta)a, \forall a \in V, \forall \alpha, \beta \in \mathbb{K}$;

2. $(\alpha + \beta)a = \alpha a + \beta a, \forall a \in V, \forall \alpha, \beta \in \mathbb{K}$;

3. $\alpha(a + b) = \alpha a + \alpha b, \forall a, b \in V, \forall \alpha \in \mathbb{K}$;

4. $1a = a, \forall a \in V$.

Dado o contexto e os objetivos desta dissertação só achamos necessária a demonstração dos axiomas referentes aos quaterniões.

Seja $(\mathbb{H}, \mathbb{R}, +, .)$ um espaço vetorial. $\forall \alpha, \beta, \gamma \in \mathbb{R}$ e $\hat{p}, \hat{q}, \hat{v} \in \mathbb{H}$:

- $\alpha\hat{q} \in \mathbb{H}$

$$\begin{aligned} \alpha\hat{q} &= \alpha(q_0 + \vec{q}) \\ &= \alpha q_0 + \alpha q_1 i + \alpha q_2 j + \alpha q_3 k \\ &\Rightarrow \alpha\hat{q} \in \mathbb{H} \end{aligned}$$

- $\hat{q} + \hat{p} \in \mathbb{H}$

$$\begin{aligned} \hat{q} + \hat{p} &= (q_0 + p_0) + (\vec{q} + \vec{p}) \\ &= (q_0 + p_0) + (q_1 + p_1)i + (q_2 + p_2)j + (q_3 + p_3)k \\ &\Rightarrow \hat{q} + \hat{p} \in \mathbb{H} \end{aligned}$$

- $\hat{q} + \hat{p} = \hat{p} + \hat{q}$

$$\begin{aligned} \hat{p} + \hat{q} &= (p_0 + q_0) + (\vec{p} + \vec{q}) \\ &= (p_0 + q_0) + (p_1 + q_1)i + (p_2 + q_2)j + (p_3 + q_3)k \\ &= \hat{q} + \hat{p} \end{aligned}$$

- $\hat{q} + 0 = \hat{q}$

- $\hat{q}(\alpha + \beta) = \alpha\hat{q} + \beta\hat{q}$

$$\begin{aligned} \hat{q}(\alpha + \beta) &= (q_0 + \vec{q})(\alpha + \beta) \\ &= (q_0 + q_1 i + q_2 j + q_3 k)(\alpha + \beta) \\ &= (\alpha q_0 + \alpha q_1 i + \alpha q_2 j + \alpha q_3 k) + (\beta q_0 + \beta q_1 i + \beta q_2 j + \beta q_3 k \\ &= \alpha\hat{q} + \beta\hat{q} \end{aligned}$$

- $\alpha(\hat{q} + \hat{p}) = \alpha\hat{q} + \alpha\hat{p}$

$$\begin{aligned} \alpha(\hat{q} + \hat{p}) &= \alpha((q_0 + p_0) + (\vec{q} + \vec{p})) \\ &= \alpha(q_0 + p_0) + \alpha(q_1 + p_1)i + \alpha(q_2 + p_2)j + \alpha(q_3 + p_3)k \\ &= \alpha q_0 + \alpha q_1 i + \alpha q_2 j + \alpha q_3 k + \alpha p_0 + \alpha p_1 i + \alpha p_2 j + \alpha p_3 k \\ &= \alpha\hat{q} + \alpha\hat{p} \end{aligned}$$

- $\hat{q}1 = \hat{q}$
- $\alpha(\beta\gamma)\hat{q} = (\alpha\beta)\gamma\hat{q}$

$$\begin{aligned}\alpha(\beta\gamma)\hat{q} &= \alpha(\beta\gamma)(q_0 + \vec{q}) \\ &= \alpha(\beta\gamma)q_0 + \alpha(\beta\gamma)q_1 i + \alpha(\beta\gamma)q_2 j + \alpha(\beta\gamma)q_3 k \\ &= (\alpha\beta)\gamma\hat{q}\end{aligned}$$

- $\hat{q} + (\hat{p} + \hat{v}) = (\hat{q} + \hat{p}) + \hat{v}$

$$\begin{aligned}\hat{q} + (\hat{p} + \hat{v}) &= (q_0 + \vec{q}) + ((p_0 + v_0) + (\vec{p} + \vec{q})) \\ &= (q_0 + p_0 + v_0) + (q_1 + p_1 + v_1)i + (q_2 + p_2 + v_2)j + (q_3 + p_3 + v_3)k \\ &= (\hat{q} + \hat{p}) + \hat{v}\end{aligned}$$

- $\hat{q} - \hat{q} = 0$

Com estes axiomas concluímos que o conjunto dos quaterniões no corpo dos reais formam um espaço vetorial. É, assim, possível construir rotações. Como vimos anteriormente, os quaterniões são números hipercomplexos. Recordando que a multiplicação de números complexos é equivalente a rotações em $\mathbb{R}^2$, a multiplicação entre hipercomplexos vai ser equivalente a rotações em $\mathbb{R}^3$, ou seja, a multiplicação de quaterniões vai gerar rotações em $\mathbb{R}^3$. Como os quaterniões surgiram antes dos vetores, as operações de produto escalar e produto vetorial não poderiam ser utilizadas. Então, Hamilton utilizou a forma matricial para efetuar a multiplicação entre quaterniões, que por sua vez, não é mais do que o produto interno e produto externo entre vetores $\hat{q}\hat{p}$.

$$\hat{r} = \hat{q}\hat{p} = \begin{bmatrix} q_0 & -q_1 & -q_2 & -q_3 \\ q_1 & q_0 & -q_3 & q_2 \\ q_2 & q_3 & q_0 & -q_1 \\ q_3 & -q_2 & q_1 & q_0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

Vejamos o desenvolvimento de $\hat{q}\hat{p}$, tendo em conta as propriedades de i,j e k.

$$\begin{aligned}\hat{r} &= \hat{q}\hat{p} \\ &= (q_0 + q_1 i + q_2 j + q_3 k)(p_0 + p_1 i + p_2 j + p_3 k) \\ &= q_0 p_0 + q_0 p_1 i + q_0 p_2 j + q_0 p_3 k + q_1 i p_0 + q_1 p_1 i^2 + q_1 p_2 ij + q_1 p_3 ik + q_2 j p_0 \\ &+ q_2 p_1 ij + q_2 p_2 j^2 + q_2 p_3 jk + q_3 k p_0 + q_3 p_1 ki + q_3 p_2 kj + q_3 p_3 k^2 \\ &= q_0 p_0 - q_1 p_1 - q_2 p_2 - q_3 p_3 + (q_0 p_1 + q_1 p_0 + q_2 p_3 - q_3 p_2)i + (q_0 p_2 - q_1 p_3 + q_2 p_0 + q_3 p_1)j \\ &+ (q_0 p_3 + q_1 p_2 - q_2 p_1 + q_3 p_0)k \\ &= q_0 p_0 - \vec{q}.\vec{p} + q_0\vec{p} + p_0\vec{q} + \vec{q} \times \vec{p}\end{aligned}$$

Temos, assim, a explicação do resultado de Hamilton. Utilizando a regra de multiplicação de matrizes, podemos observar que o resultado é o mesmo:

$$q_0p_0 - \vec{q}.\vec{p} + q_0\vec{p} + p_0\vec{q} + \vec{q} \times \vec{p}$$

Posto isto, podemos passar à rotação entre quaterniões. É necessário definir um vetor $\vec{v}$, quaternião puro $\hat{w}$, ou seja, um quaternião sem parte escalar e um operador de rotação $\hat{q}$, um quaternião unitário.

$\hat{v} = v_0 + \vec{v}$, com $v_0 = 0$;
$|\hat{q}| = 1$;
$\hat{w} = \hat{q}\hat{v}\hat{q}^* = \hat{q}^*\hat{v}\hat{q}$.
Com os vetores e quaterniões definidos, passamos aos ângulos.

Sabemos que, pela equação fundamental da trigonometria:

$$\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2} = 1.$$

Se $|\hat{q}| = 1$, então $|\hat{q}|^2 = 1$. É equivalente dizer que:

$$\hat{q} = q_0 + \vec{q}$$

$$|\hat{q}|^2 = q_0^2 + |\vec{q}|^2 = 1$$

Seja $\vec{u}$, um vetor diretor para representar o eixo rotação com $|\vec{u}| = 1$, associando o cosseno com q_0 e o seno com $\vec{q}$ e, ainda, utilizando θ como ângulo de rotação, temos:

$$|\hat{q}|^2 = \cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}$$

$$\hat{q} = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\vec{u}$$

Assim, utilizando as relações fundamentais de i,j e k e simplificando ao máximo todos os termos, temos $\hat{w}$,o vetor resultante da rotação de quaterniões na representação usual:

$$\begin{aligned}
\hat{w} &= \hat{q}\hat{v}\hat{q}^* \\
&= \hat{q}^*\hat{v}\hat{q} \\
&= (q_0 - \vec{q})(v_0 q_0 - \vec{v}\vec{q} + v_0\vec{q} + q\vec{v} + \vec{v}\times\vec{q}) \\
&= q_0^2 v_1 i + q_0^2 v_2 j + q_0^2 v_3 k + q_1^2 v_1 i - q_1^2 v_2 j - q_1^2 v_3 k - q_2^2 v_1 i \\
&+ q_2^2 v_2 j - q_2^2 v_3 k - q_3^2 v_1 i - q_3^2 v_2 j + q_3^2 v_3 k + q_1 v_1 q_2 j + q_1 v_1 q_2 j + q_1 v_1 q_3 k \\
&+ q_1 v_1 q_3 k + q_2 v_2 q_1 i + q_2 v_2 q_1 i + q_2 v_2 q_3 k + q_2 v_2 q_3 k + q_3 v_3 q_1 i + q_3 v_3 q_1 i \\
&+ q_3 v_3 q_2 j + q_3 v_3 q_2 j + q_0 v_1 q_2 k - q_0 v_1 q_3 j - q_0 v_2 q_1 k - q_2 q_0 v_3 i + q_0 v_2 q_3 i \\
&+ q_0 v_3 q_1 j - q_0 v_3 q_2 i - q_1 q_0 v_2 k + q_1 q_0 v_3 j + q_2 q_0 v_1 k - q_3 q_0 v_1 j + q_3 q_0 v_2 i \\
&= (q_0^2 - \vec{q}.\vec{q})\vec{v} + 2(\vec{q}.\vec{v})\vec{q} + 2q_0(\vec{q}\times\vec{v})
\end{aligned}$$

Substituindo $\hat{q} = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\vec{u}$ e utilizando as regras que se seguem da trigonometria:

$$\cos 2\theta = \cos^2\theta - \sin^2\theta$$

$$\sin^2\theta = \frac{1}{2}(1 - \cos 2\theta)$$

$$\cos\theta\sin\theta = \frac{1}{2}\sin 2\theta$$

temos a rotação na representação eixo-ângulo:

$$\begin{aligned}
\hat{w} &= (q_0^2 - \vec{q}.\vec{q})\vec{v} + 2(\vec{q}.\vec{v})\vec{q} + 2q_0(\vec{q}\times\vec{v}) \\
&= \left(\cos^2\tfrac{\theta}{2} - \sin\tfrac{\theta}{2}\vec{u}.\sin\tfrac{\theta}{2}\vec{u}\right)\vec{v} + 2\left(\sin\tfrac{\theta}{2}\vec{u}.\vec{v}\right)\sin\tfrac{\theta}{2}\vec{u} + 2\cos\tfrac{\theta}{2}\left(\sin\tfrac{\theta}{2}\vec{u}\times\vec{v}\right) \\
&= \left(\cos^2\tfrac{\theta}{2} - \sin^2\tfrac{\theta}{2}(\vec{u}.\vec{u})\right)\vec{v} + 2\sin^2\tfrac{\theta}{2}(\vec{u}.\vec{v})\vec{u} + 2\cos\tfrac{\theta}{2}\sin\tfrac{\theta}{2}(\vec{u}\times\vec{v}) \\
&= \left(\cos^2\tfrac{\theta}{2} - \sin^2\tfrac{\theta}{2}\right)\vec{v} + 2\sin^2\tfrac{\theta}{2}(\vec{u}.\vec{v})\vec{u} + 2\cos\tfrac{\theta}{2}\sin\tfrac{\theta}{2}(\vec{u}\times\vec{v}) \\
&= \cos\theta\vec{v}(1 - \cos\theta)(\vec{u}.\vec{v})\vec{u} + \sin\theta(\vec{u}\times\vec{v}) \\
&= (1 - \cos\theta)(\vec{u}.\vec{v})\vec{u} + \vec{v}\cos\theta + (\vec{u}\times\vec{v})\sin\theta
\end{aligned}$$

Efetuámos, portanto, demonstrámos uma rotação do vetor $\vec{v}$ em torno do eixo $\vec{u}$ com o quaternião $\hat{q} = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\vec{u}$.

Exemplo 5.1 *O quaternião que se segue representa uma rotação com o ângulo $\frac{\pi}{3}$ (30) em torno do vetor $(1,0,0) \in \mathbb{R}^3$.*

$$\hat{q} = \cos\frac{\pi}{6} + \sin\frac{\pi}{6}i = \frac{\sqrt{3}}{2} + \frac{1}{2}i$$

.

Estudada a álgebra e a rotação dos quaterniões, estamos em condições de trabalhar com os mesmos. Na aplicação *Smartbody*, os quaterniões estão repartidos em 3 valores e, como já foi visto anteriormente, um quaternião é composto por 4 valores. Para ser possível trabalhar com os mesmos foi necessário fazer uma conversão.

Em primeiro lugar é feita a normalização e extração dos ângulos:

```
void SrQuat::set ( const SrVec& axisangle )
float ang;
x=axisangle.x; y=axisangle.y; z=axisangle.z;
ang = x*x + y*y + z*z;
if ( ang>0 )
{ ang = sqrtf ( ang );
x/=ang; y/=ang; z/=ang;
}
```

E em seguida tratam-se das rotações:

```
ang/=2;
w = cosf ( ang );
ang = sinf ( ang );
x*=ang; y*=ang; z*=ang;
```

Continuando na descrição do ficheiro da animação representado na Figura 5.16, depois dos canais estarem definidos, segue-se a definição dos *frames* (combinações de várias imagens que, em conjunto, formam uma animação). Para além de indicar a quantidade de *frames* (*frames 41*) que vão ser utilizados, também é necessário indicar o tempo em que se vão iniciar (*kt 0.000000*). Os restantes números que se seguem correspondem aos valores dos canais.

Por outras palavras, seria lógico pensar que cada *frame* teria 201 valores, mas o que, na verdade, acontece é que, como já foi descrito anteriormente, cada canal $XPos$, $YPos$ e $ZPos$ tem apenas um único valor. Todos os outros que são de rotação, ou seja, os *Quat* têm 3 valores. No fim do ficheiro da animação temos a indicação do tempo para cada ponto de sincronização.

Depois das animações serem analisadas ao mais ínfimo pormenor, foram feitos vários testes, de onde surgiu a ideia de desfragmentação em seis partes da animação de um agente em marcha.

A desfragmentação anteriormente mencionada foi realizada com o intuito de termos seis animações de caminhadas, mas só a movimentar várias partes do corpo, ficando com algumas bloqueadas. Repartimo-las da seguinte forma: uma animação em que havia movimentação só num pé, outra unicamente no joelho direito e, por fim, uma outra só movimentando um lado da anca do agente.

Na prática isto significou a alteração em todos os *frames* dos valores dos canais de rotação dos pés, joelhos e ancas para o valor 0. Ficamos com uma animação em que o pé esquerdo não tinha rotação, outra em que o pé direito não tinha rotação e assim sucessivamente para os joelhos e ancas.

As animações, só por si, não nos pareceram naturais, como seria de esperar e, por isso, fomos levados a seguir a seguinte estratégia.

5.3.9 Utilização do atributo *blend*

Depois de as animações estarem desfragmentadas, foi utilizada a função *blend*. Um *blend* não é mais do que uma mistura de várias animações, Figura 5.17.

Para utilizarmos este atributo é necessário reunir os seguintes requisitos: ter um vasto conhecimento do conjunto de animações escolhidas, especificar os seus parâmetros, adicionar pontos de correspondência entre as animações e ainda adicionar triângulos (se tratar-se de *blend* a duas dimensões) ou tetraedros (se se tratar de *blend* tridimensionais) com as mesmas.

O objetivo inicial era juntar as animações anteriores de maneira a que se produzisse uma animação igual à animação original. Embora não tivesse o efeito pretendido, foi possível verificar, com a construção de um hexágono, que, no ponto geométrico central, houve uma mistura parcial de todas as animações.

A principal vantagem que este método oferece é a possibilidade de parametrização das animações em diferentes dimensões.

Este método requer uma análise anterior à sua utilização pois é essencial ter em conta alguns aspetos. Deve, assim, ser feita uma escolha cuidada das animações, porque estas têm de ter um objetivo comum. Por exemplo, um *blend* que não funcionaria seria o de um agente a fazer o pino juntamente com um agente a correr. Outro aspeto a termos em consideração é a utilização de animações sequenciais. Em vez de utilizar apenas um *blend*, a melhor estratégia a adotar, neste caso, seria criarmos um grupo de *blends* (onde cada *blend* seria composto por uma única animação) e, em seguida, adicionarmos transições entre os mesmos.

5.3.10 Utilização *Yaw*, *Pitch* e *Roll*

Estes foram utilizados, não no contexto da deslocação do agente, mas sim no de seguir um objeto em constante mudança de direção. Os três componentes são utilizados em diversas situações e giram em torno de um centro de gravidade.

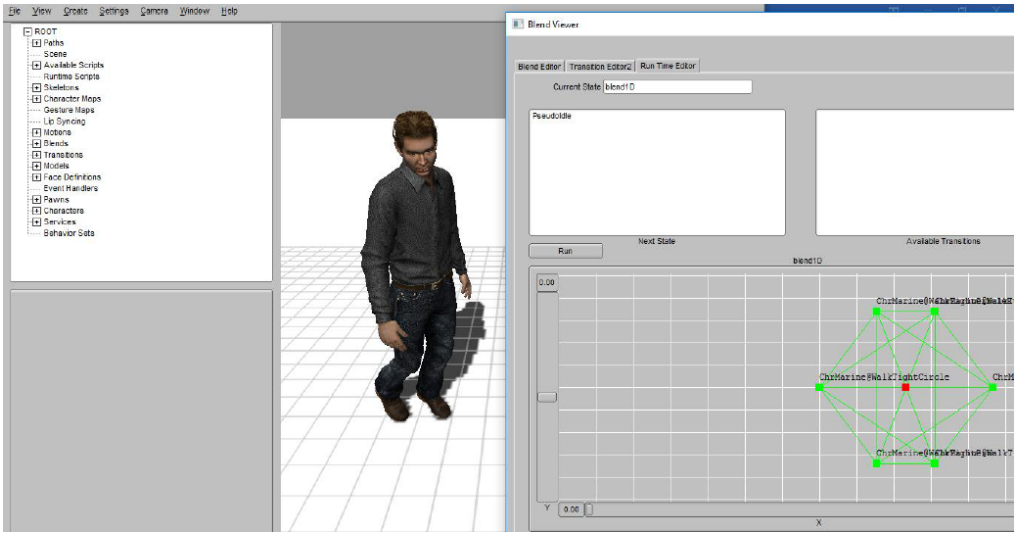


Figura 5.17: *Blend* de animações editadas.

O *yaw* corresponde à rotação que possa existir verticalmente, o *pitch*, à rotação em relação à horizontal (uma rotação lado a lado) e o *roll* corresponde à rotação que possa vir de trás para a frente.

A conversão foi feita num controlador da seguinte forma:

```
def getYPR (quat):
    w = quat.getData(0)
    x = quat.getData(1)
    z = quat.getData(2)
    y = quat.getData(3)

    roll = degrees(math.atan2(2*(x*y+w*z),w*w+x*x-y*y-z*z))
    pitch = degrees(math.asin(-2*(x*z-w*y)))
    yaw = degrees(math.atan2(2*(y*z + w*x),w*w-x*x-y*y+z*z))

    return [yaw, pitch, roll]
```

Feita a conversão, foram definidos novos *yaw*, *pitch* e *roll* sobre diferentes juntas. E, ainda, adicionámos restrições com limites, tanto para valores máximos, como para valores mínimos. No exemplo que utilizámos, foi possível verificar uma restrição quase perfeita de um agente com uma restrição no pescoço a olhar para uma bola.

Esta abordagem, a nível de código, acabou por ser a mais complexa. Foi necessário criar uma classe com o objetivo de que esta funcionasse como uma espécie de novo controlador. Houve também que fazer conversões de ângulos para radianos e vice-versa. Mas, por outro lado, foi das únicas abordagens

em que foi possível haver um controlo na parametrização das restrições sobre qualquer junta.

5.4 Comparação das estratégias

Na primeira estratégia, na utilização do **método *Rotate***, é necessário ter em atenção qual o tipo de junta de que vamos fazer a rotação, pois se a junta tiver filhos, não vai haver uma modificação direta. É também importante ter em conta os limites de flexão e de extensão das juntas, pois não há nenhuma restrição neste método a esse nível.

Para uma deformidade poder ser visível na caminhada do agente, a dica é a de aplicar os ângulos de rotação diretamente nos joelhos, pés ou tornozelos, em animações nas quais o agente virtual esteja a andar, sem cometer excessos.

É o método mais simples, pois só é necessário utilizar a linha de comandos. Os comandos são muito simples, são apenas 3 linhas de código. Na primeira linha é necessário indicar qual a animação em que queremos efetuar a rotação. Na segunda invocamos o método *rotate* que recebe como parâmetros os ângulos e a junta correspondente.

O processo que acontece é o seguinte: na animação escolhida há uma alteração temporária nos canais de rotação da junta. Seguidamente, para ver a rotação, basta utilizar a terceira linha de código, que é um comando em BML. Este faz a correspondência do agente com a animação e aplica a rotação nesse agente.

Relativamente às **restrições**, verificámos que estas acabaram por se revelar muito próximas da realidade. Este método partiu da modificação do *script* de exemplo *ConstraintDemo.py*. Serviu também como base para algumas das estratégias seguintes, tendo sido utilizados objetos para delimitar as ações. Em primeiro lugar extraímos os dados em que as juntas se encontravam, para podermos colocar os objetos nas posições exatas.

Por exemplo, para a Figura 5.5 extraímos os seguintes valores, em metros, da posição do pé esquerdo:

$$x = 0.0988204851747m$$

$$y = 0.02940069139m$$

$$z = 0.16155333817m$$

Em seguida, utilizámos estes valores como um centro de uma esfera e verificámos que havia um perímetro permitido para colocar os objetos. Tendo em conta que colocámos o objeto nestas posições $x = 0.56$, $y = 0$, $z = 0.6$,

a distância mais longa possível de colocar os objetos afastados num dos pés, para poder haver uma restrição correta relativa ao caminhar foi de:

$$x = 0.47m$$

$$y = 0.02m$$

$$z = 0.44m$$

Por outro lado, neste exemplo, o agente virtual desloca-se para uma determinada posição, com 4 passos, em cerca de 5 segundos.

Podemos concluir que o comprimento de passada nesta animação é cerca de $0.1175metros$ e que a velocidade média de avanço é de $0.094m/s$.

Comparando com um agente sem a restrição, foi possível verificar que, tanto o comprimento da passada como a velocidade média, são maiores.

O comprimento da passada aumenta e, em vez de o agente sem restrições dar 4 passos para chegar ao destino, apenas dá 3.

Na **alteração dos limites dos esqueletos** existiam juntas que não possuíam limites e foi preciso acrescentá-los, mas produziram efeito nalguns. Nestes casos, os limites que tinham de ser alterados eram nos canais de rotação e não nos de posição com um limite de posição no cotovelo, sem ultrapassar os graus de extensão e flexão, normalmente, permitidos.

Todos os outros testes levaram-nos a concluir que esta estratégia é muito limitativa, pois só funcionou para pouquíssimos casos. A nível de implementação é preciso ter em atenção alguns pormenores. Todos estes ficheiros têm dependências e a melhor solução é criar um novo esqueleto, colocá-lo no mesmo diretório e chamá-lo em todas as instâncias em que o existente é solicitado.

A exploração destes métodos ***setOffset***, ***setUseRotation***, ***setPrerotation*** e ***setPostrotation***, levou-nos à conclusão de que todos efetuam as alterações durante a construção da hierarquia das juntas. Não foram verificadas grandes diferenças entre pré e pós-rotações, sendo que as pré-rotações normalmente diminuem os graus de rotações de forma ligeira.

Quanto ao método ***setUseRotation***, verificámos que este se revelou ser útil quando queremos indicar qual a junta que sofre ou não uma rotação.

Já a função ***setOffset*** limita o comprimento de extensão e de flexão das juntas indicadas, mas só quando estas não têm dependências que reproduzam o efeito pretendido.

A estratégia de **alterar as animações** foi a que implicou um maior período de tempo deste trabalho dado que foi preciso analisar algumas centenas de colunas de números para perceber qual os canais certos a mudar para cada *frame*.

Todas as animações existentes foram estudadas linha a linha na medida em que era necessário ter em atenção que aquelas têm 60 *frames* por segundo e que, em todos os segundos, para a animação ter um bom funcionamento, os pontos de sincronização têm que estar bem definidos.

Para utilizarmos esta estratégia no futuro teremos de ter várias animações já capturadas e, a partir de então, poderemos proceder à sua modificação.

A **estratégia do *blend*** funciona bem quando o objetivo é termos uma mistura de várias animações numa só. O resultado melhora com a quantidade e complexidade das mesmas. Quanto mais animações forem adicionadas, quanto mais complexa for a figura geométrica, quanto mais pormenorizada, tanto melhor será o resultado.

Aqui, temos apenas um hexágono, seis vértices e um ponto central. Mas, por exemplo, nesta plataforma, no caso concreto da locomoção, esta é realizada utilizando 19 animações diferentes com um tetraedro e o resultado é perfeito.

Para o caso em estudo, esta é a que, sem dúvida, se aproxima mais da realidade. O problema é que teríamos de capturar os movimentos de todas as deficiências, todos os graus a que o pé se levanta do chão, todos os ângulos a que a cabeça gira, todas as vertentes da inclinação que o corpo faz, entre outros. Tal operação levaria imenso tempo, mas, em contrapartida, o realismo conseguido seria bastante satisfatório.

A ***Yaw***, ***pitch*** e ***roll*** foi a mais complexa a nível de programação. Depois de todas as conversões necessárias, definimos um máximo e um mínimo de distância até onde o agente poderia olhar, verificando os valores máximos possíveis, tendo-se realizado inúmeros testes para confirmação.

Na Tabela 5.1 temos uma comparação sintetizada de todos os métodos estudados.

Tabela 5.1: Comparação das estratégias.

Estratégias	**Vantagens**	**Desvantagens**
Método *Rotate*	Simplicidade de código e menor tempo de implementação.	Nem sempre se aproxima da realidade; limitação nos ângulos de extensão e flexões.
Restrições	Aproximação ao realismo e menor tempo de implementação.	Nem sempre se aproxima da realidade; obrigatoriedade do uso de objetos.
Limites no esqueleto	Simplicidade de código e menor tempo de implementação.	Nem sempre se aproxima da realidade; dependência de ficheiros; limitação de ângulos de extensão e flexões.
Métodos *setOffset*, . . . , *setPostrotation*	Simplicidade de código e menor tempo de implementação.	Afastamento da realidade.
Alteração das animações	Aproximação ao realismo.	Complexidade de código; maior tempo de implementação; necessidade de captura movimentos reais e de conversões.
Atributo *blend*	Aproximação ao realismo e possibilidade de animações parametrizáveis.	Necessidade de captura movimentos reais e maior tempo de implementação.
Yaw*, *pitch* e *roll	Aproximação ao realismo.	Complexidade de código; maior tempo de implementação e necessidade de conversões.

Capítulo 6

Conclusão

Nesta etapa final do meu trabalho, faremos, ainda, uma pequena síntese da índole dos conteúdos abordados, das principais dificuldades encontradas, do modo como conseguimos dar resposta aos objetivos delineados inicialmente. Abordaremos ainda as falhas, as conquistas, as frustrações sentidas e, por fim, o sentimento de realização.

À priori, aquilo que pretendíamos era aprofundar as caraterísticas da 'Definição de funções primitivas em *Behavior Markup Language*', como definido no título. Esta temática suscitou-nos interesse devido à nossa vontade de compreender melhor as dificuldades do ser humano que é diferente, quer sob o ponto de vista físico, quer ainda sob outros aspetos, e identificar as suas grandes dificuldades de adaptação ao meio ou ambiente geral e comum. Por outro lado, julgámos que o facto de inserirmos estes agentes específicos e de nos debruçarmos sobre as técnicas de inserção dos mesmos poderia ser um contributo para a expansão da ideia de que todos os seres humanos nos devem merecer o maior respeito e admiração. Merecem que tudo façamos para a descoberta de modos de ajuda na superação de barreiras e outros obstáculos que surgem no seu quotidiano. Na verdade, aquilo que é diferente, para nós, suscita mais interesse, gera uma maior ânsia de conhecimento e representa um maior desafio.

Para esta definição, foi necessário realizar um intenso estudo das linguagens formais tendo como objetivo encontrar aquelas que serviriam melhor o fito de descrever, mais fielmente, o comportamento humano, as que apresentariam melhores qualidades e as que melhor se adaptariam a estes agentes e contextos tão particulares. Destas, a escolhida foi a BML pelas características já enunciadas.

Logo, analisámos os ambientes virtuais para verificar qual linguagem se

integraria melhor segundo o modo pretendido.

Após uma breve definição do agente virtual, selecionámos a linguagem adequada e o ambiente virtual. A aplicação *Smartbody* proporcionou o ambiente virtual e constituiu um desafio significativo pois surgiram então as maiores dificuldades, não só devido à falta de documentação, como também de conhecimento desta plataforma.

Assim, o primeiro passo foi conhecê-la, comunicando com os criadores da mesma, fazendo inúmeros testes, verificando resultados, testando atributos, criando restrições. Já mais confortáveis, após um melhor conhecimento da plataforma *Smartbody*, estudámos as estratégias para possíveis deficiências motoras, que necessitámos de aprofundar nas suas particularidades e especificidades de modo a podermos analisar e propor estratégias a implementar. O pretendido, desde do início, foi obter um esqueleto virtual o mais idêntico possível ao do corpo humano. Utilizámos diversos métodos, alterámos animações, tendo sempre como base aplicações matemáticas.

Concluímos que a melhor solução das estratégias estudadas foi o recurso ao atributo *blend*. Foi uma solução não esperada, visto que não alterava diretamente as juntas e, por tal, acabou por ser um atributo que ficou para últimas análises. Não só porque foi o que se aproximou mais da realidade, mas também porque foi o que se adaptou da melhor forma ao ambiente escolhido o *blend* foi o escolhido. Deste modo, e como pretendido ao longo deste estudo, podemos concluir que o atributo acima mencionado, não só mostrou ser a melhor escolha, como também acabou por surpreender positivamente nos resultados obtidos.

Apesar de não ter sido testado, em futuros trabalhos, a melhor implementação de deficiências motoras, num agente virtual, seria incorporar mais um tipo de locomoção com a captura de vários tipos de movimentos e em seguida juntá-los num *blend*.

Para concluir, este percurso analítico não foi só um desafio para nós, mas sobretudo uma aprendizagem constante e profícua. Tivemos de nos adaptar paulatinamente aos resultados sempre com a visão de atingirmos o nosso objetivo inicialmente definido. Foram as diversas dificuldades e até algumas frustrações, causadas pelo desconhecimento de algumas matérias, mas também o nosso espírito de luta e a ajuda dos nossos orientadores que nos proporcionam nestes momentos um sentimento de orgulho por termos finalizado este ano de estudos. Nem sempre o que planeáramos tomou o rumo mais certo, mas, assim como na matemática, há que saber trabalhar

com os resultados que nos são dados e nos adaptarmos constantemente às circunstâncias, vislumbrando outras possibilidades de prosseguir no nosso caminho em direção às nossas metas.

Anexos

Anexo A

Comparação das linguagens AML, MPML, VHML e BML

Neste Anexo temos uma comparação de vários atributos das linguagens AML, MPML, VHML e BML introduzidas no Capítulo 3.

	BML	**AML**	**MPML**	**VHML**
Objetivos	Animação	Animação	Apresentação	Animação
Elementos de especificação	Definição do agente; Animação; Mundo; Comportamentos; Controlo de voz.	Animação; Comportamentos.	Definição do agente; Animação; Mundo; Comportamentos.	Definição do agente; Animação; Diálogo; Comportamentos; Controlo de voz.
Controlo de animações	Sincronização; Conjugar animações; Permite adicionar parâmetros.	Conjugar animações; Sincronização; Permite adicionar parâmetros; Ações parametrizadas.	Conjugar animações; Sincronização; Ações parametrizadas.	Sincronização; Permite adicionar parâmetros; Ações parametrizadas.
Tipo de agente	Humano	Humano	Humano	Humano e Não humano.
Partes do corpo	Cara, Corpo e Discurso.	Cara, Corpo e Discurso.	Corpo e Discurso.	Cara, Corpo e Discurso.

Anexo B

Lista de exemplos *Smartbody*

Seguem-se ilustrações da lista de exemplos utilizados na aplicação *Smartbody* referidos no Capítulo 4.

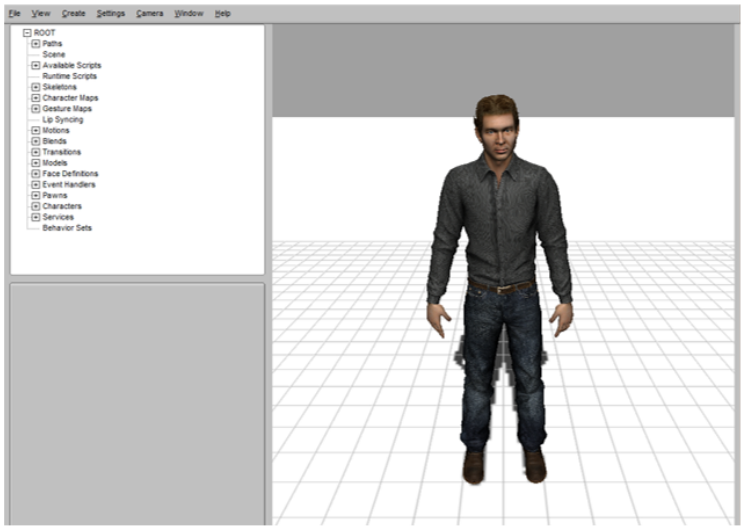


Figura B.1: *AddCharacterDemo.py*.

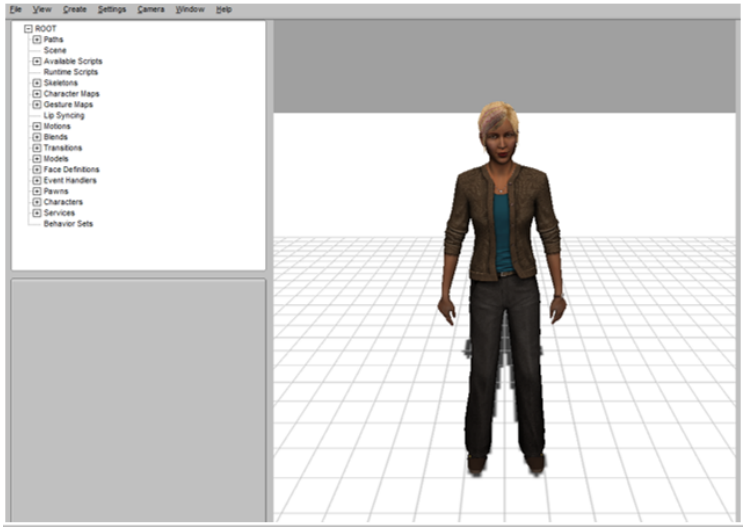


Figura B.2: *AddCharacterDemoRachel.py*.

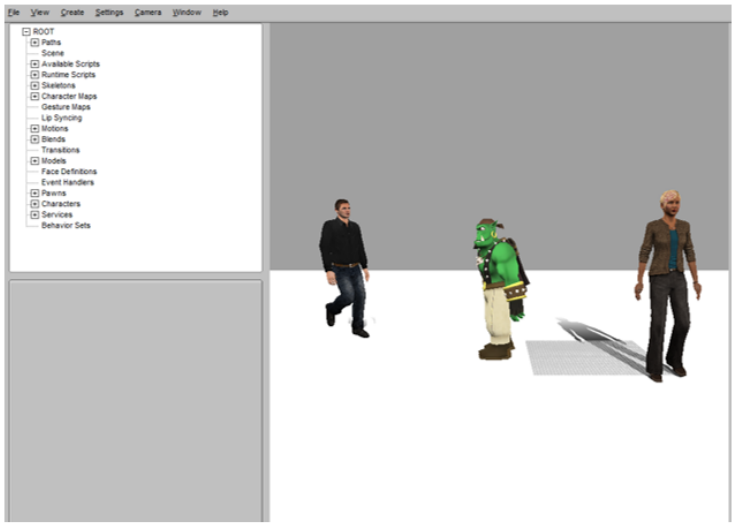


Figura B.3: *BlendDemo.py*.

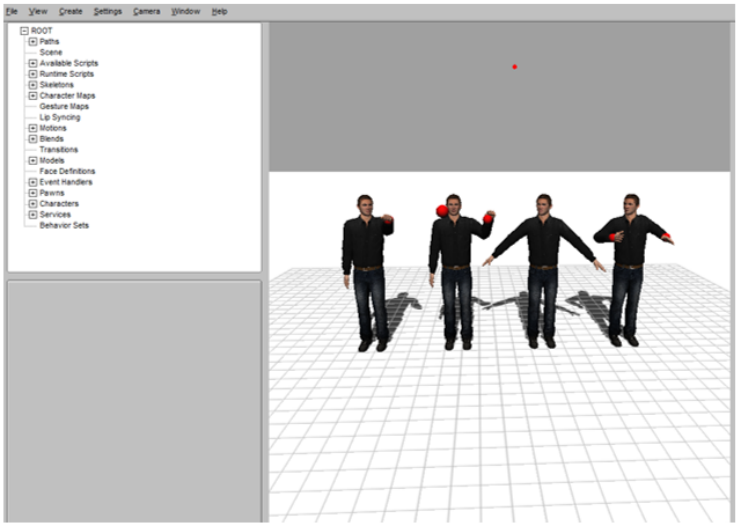


Figura B.4: *ConstraintDemo.py*.

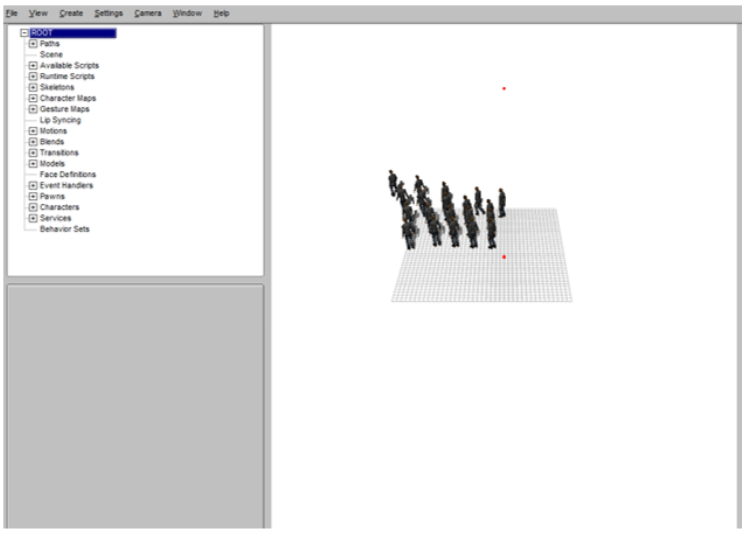


Figura B.5: *CrowdDemo.py*.

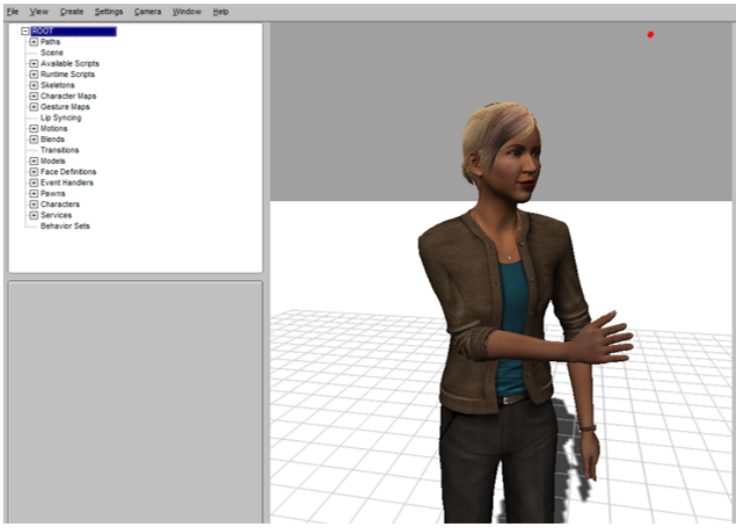


Figura B.6: *EventDemo.py*.

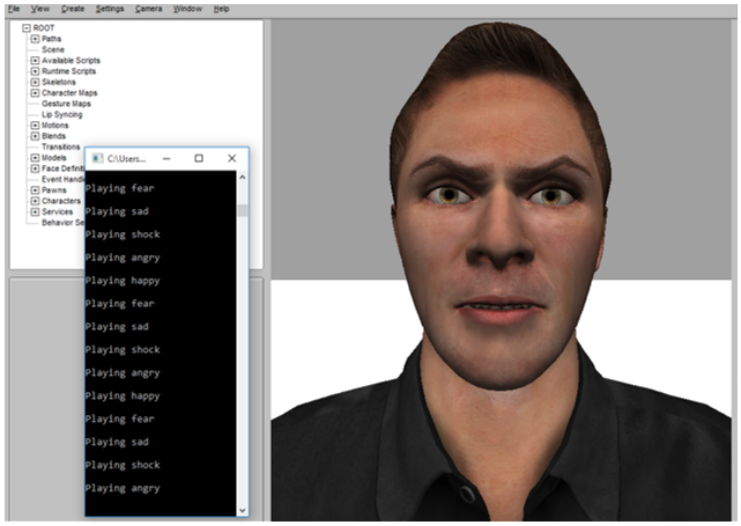


Figura B.7: *FacialMovementDemo.py*.

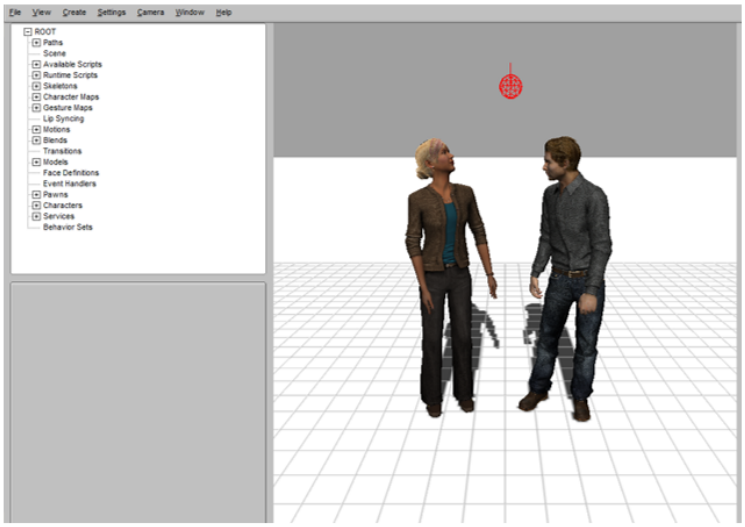


Figura B.8: *GazeDemo.py*.

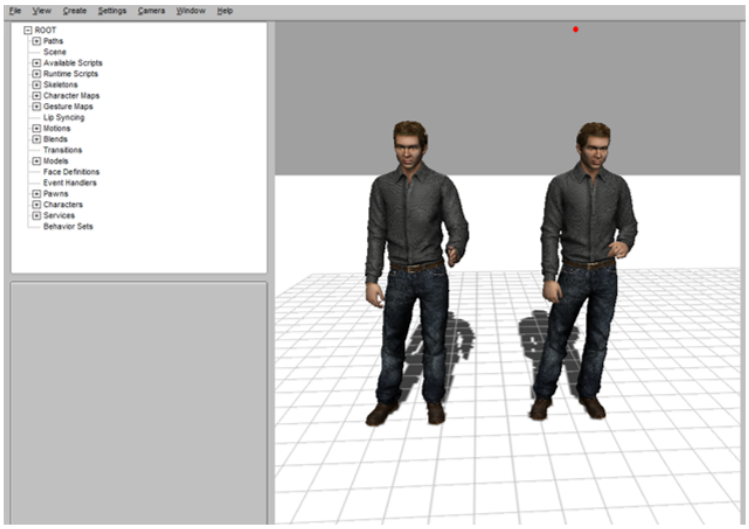


Figura B.9: *GesturesDemo.py*.

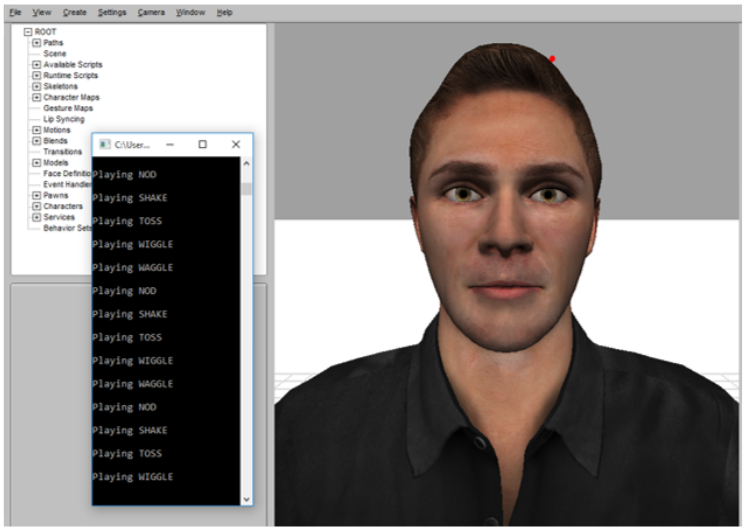


Figura B.10: *HeadDemo.py*.

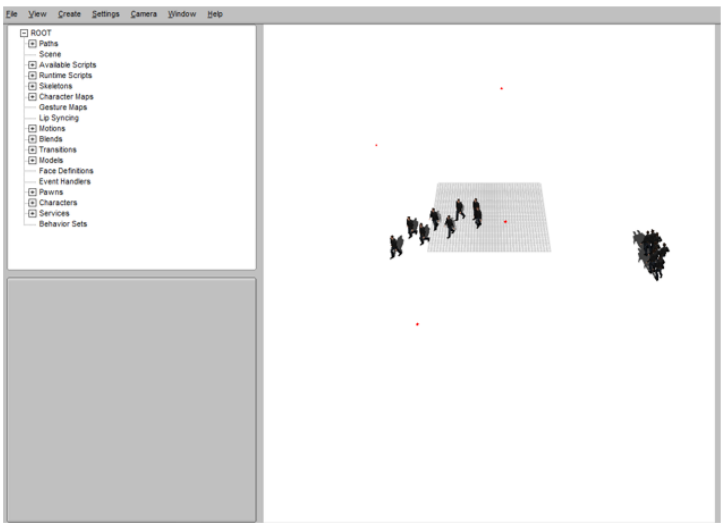


Figura B.11: *LocomotionDemo.py*.

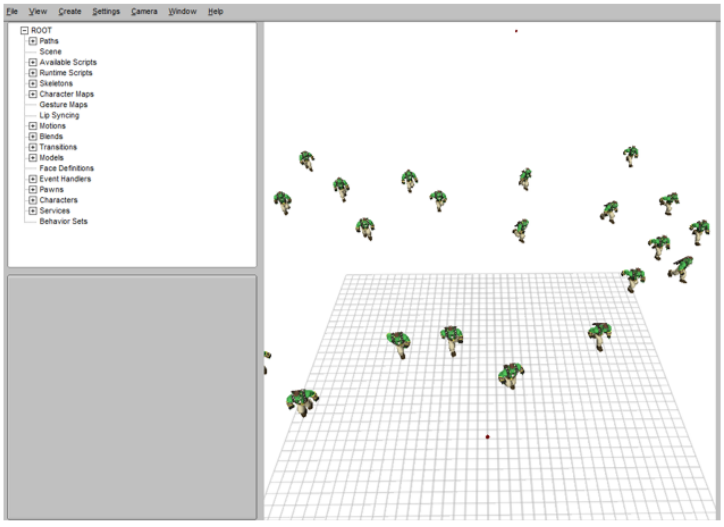


Figura B.12: *OgreCrowdDemo.py*.

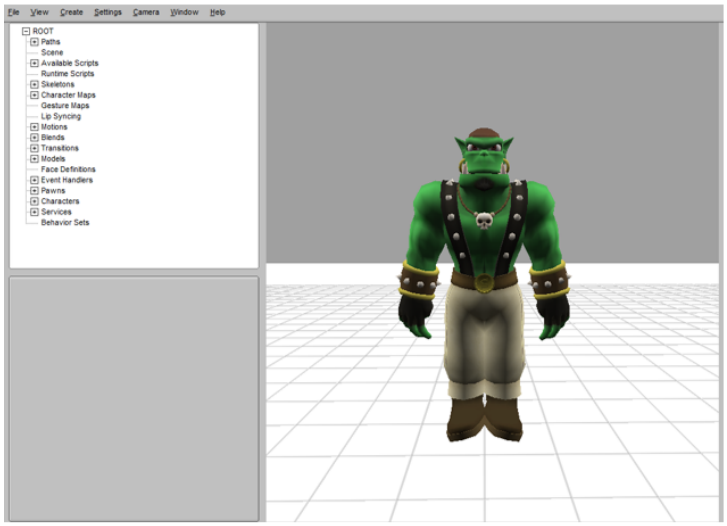


Figura B.13: *OgreDemo.py*.

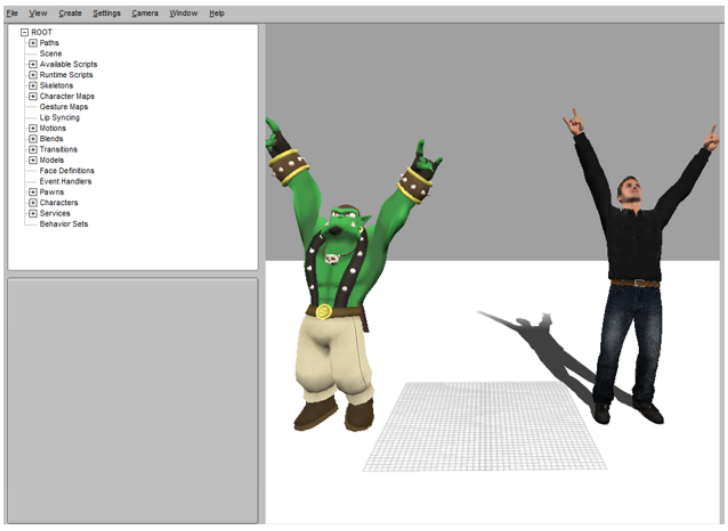


Figura B.14: *HeadDemo.py*.

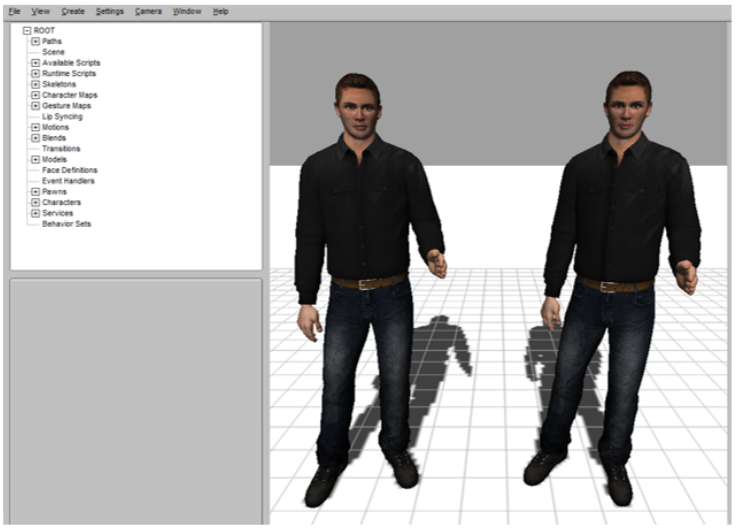


Figura B.15: *PerlinNoiseDemo.py*.

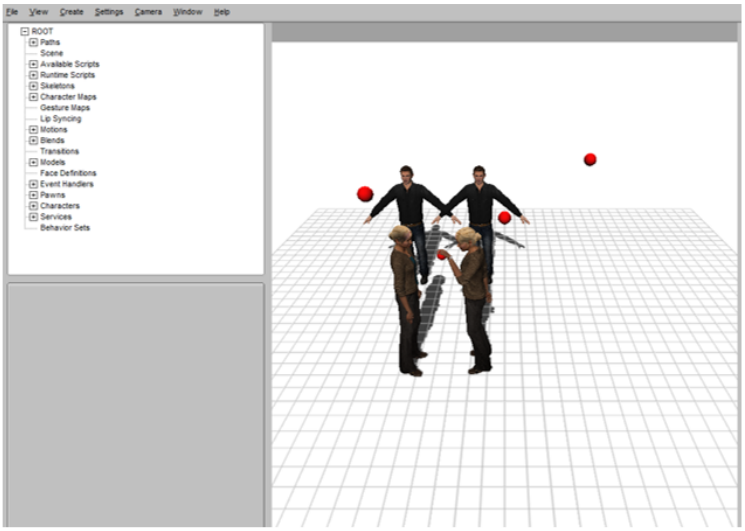


Figura B.16: *ReachDemo.py*.

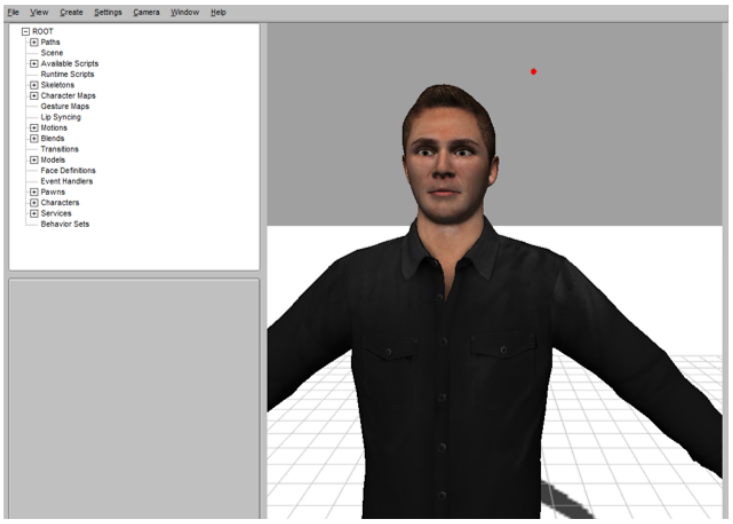


Figura B.17: *SaccadeDemo.py*.

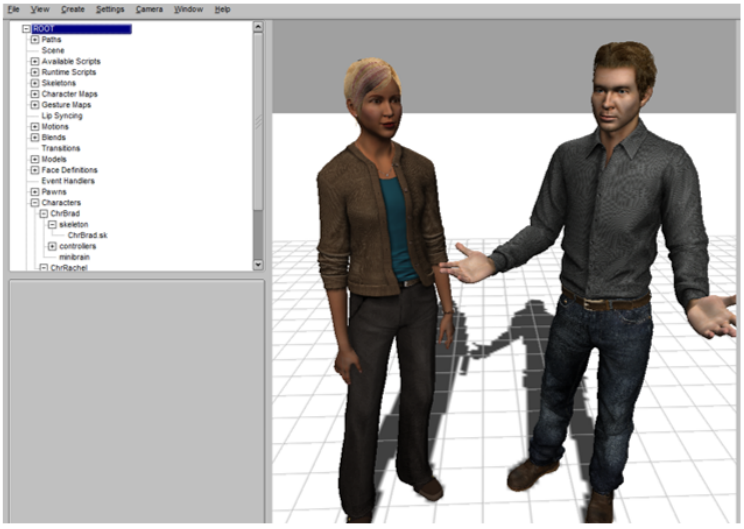
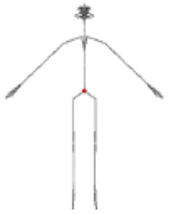
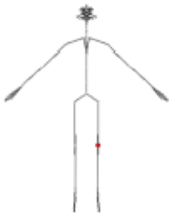
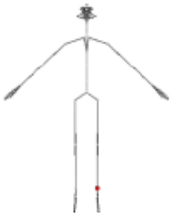


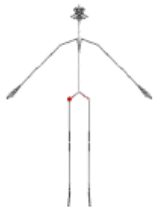
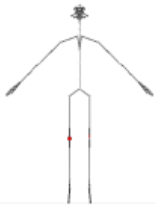
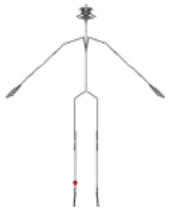
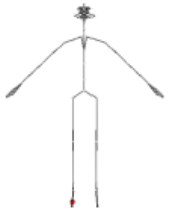
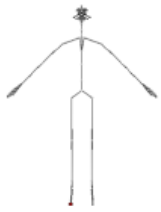
Figura B.18: *SpeechDemo.py*.

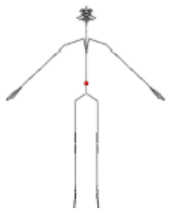
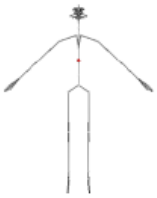
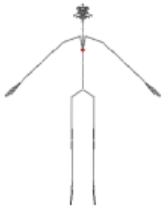
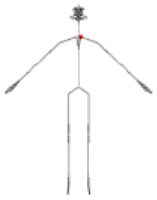
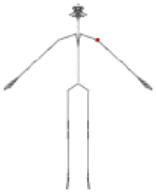
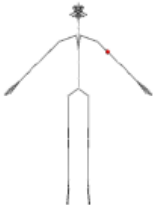
Anexo C

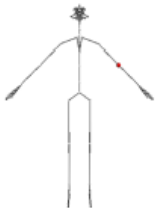
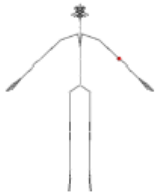
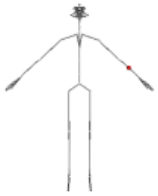
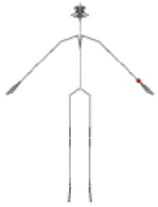
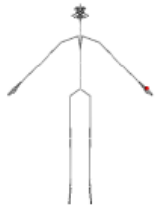
Definição do esqueleto

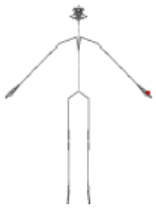
Para uma melhor interpretação, foi feita uma ilustração dos nomes dos ossos existentes no esqueleto. O esqueleto padrão *Smartbody* é composto por um conjunto de nomes e de relações. É utilizada uma hierarquia pois alguns são obrigatórios na criação de esqueletos para que os controladores do *Smartbody* funcionem corretamente.

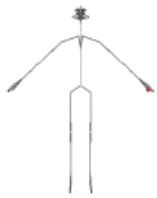
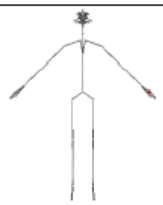
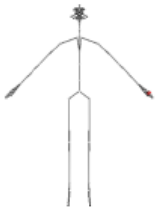
Nome do osso predefinido do *SmartBody*	Nome do osso de *zebra2*	Esqueleto
JtRoot	base	
JtHipLf	l_hip	
JtKneeLf	l_knee	
JtAnklef	l_ankle	
JtBallLf	l_forefoot	

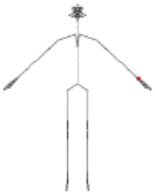
JtToeLf	l_toe	
JtHipRt	l_hip	
JtKneeRt	r_knee	
JtAnklcRt	r_anklc	
JtBallRt	r_forefoot	
JtToeRt	r_toe	

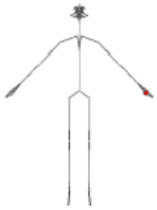
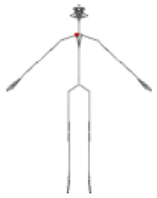
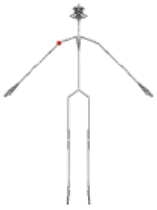
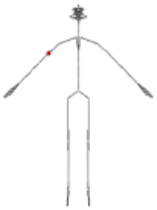
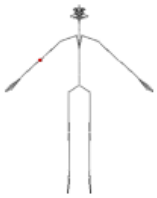
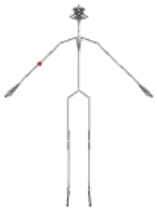
JtSpineA	spine1	
JtSpineB	spine2	
JtSpineC	spine3	
JtClavicleLf	l_sternoclavicular	
JtShoulderLf	l_shoulder	
JtUpperArmTwistALf	l_upperarm1	

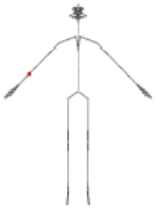
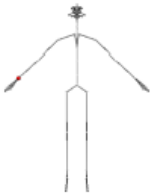
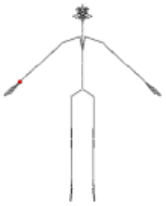
JtUpperArmTwistBLf	l_upperarm2	
JtElbowLf	l_elbow	
JtForearmTwistALf	l_forearm1	
JtForcarmTwistBLf	l_forcarm2	
JtWristLf	l_wrist	
JtIndexALf	l_index1	

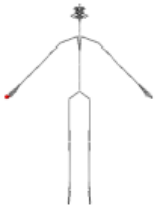
JtIndexBLf	l_index2	
JtIndexCLf	l_index3	
JTIndexDLf	l_index4	
JtLittlcALf	l_pinky1	
JtLittleBLf	l_pinky2	
JtLittleCLf	l_pinky3	

JtLittleDLf	l_pinky4	
JtMiddleALf	l_middle1	
JtMiddleBLf	l_middle2	
JtMiddleCLf	l_middle3	
JtMiddleDLf	l_middle4	
JtRingALf	l_ring1	

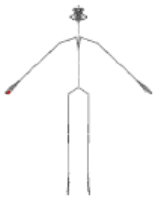
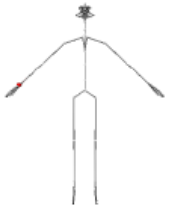
JtRingBLf	l_ring2	
JtRingCLf	l_ring3	
JtRingDLf	l_ring4	
JtThumbALf	l_thumb1	
JtThumbBLf	l_thumb2	
JtThumbCLf	l_thumb3	

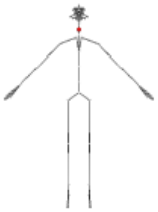
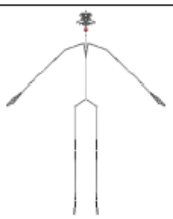
JtThumbDLf	l_thumb4	
JtClavicleRt	r_sternoclavicular	
JtShoulderRt	r_shoulder	
JtUpperArmTwistARt	r_upperarm1	
JtUpperArmTwistBRt	r_upperarm2	
JtElbowRt	r_elbow	

JtForearmTwistARt	r_forearm1	
JtForearmTwistBRt	r_forearm2	
JtWristRt	r_wrist	
JtIndexARt	r_index1	
JtIndexBRt	r_index2	
JtIndexCRt	r_index3	

JtIndexDRt	r_index4	
JtLittleARt	r_pinky1	
JtLittleBRt	r_pinky2	
JtLittlcCRt	r_pinky3	
JtLittleDRt	r_pinky4	
JtMiddleARt	r_middle1	

JtMiddleBRt	r_middle2	
JtMiddleCRt	r_middle3	
JtMiddleDRt	r_middle4	
JtRingARt	r_ring1	
JtRingBRt	r_ring2	
JtRingCRt	r_ring3	

JtRingDRt	r_ring4	
JtThumbARt	r_thumb1	
JtThumbBRt	r_thumb2	
JtThumbCRt	r_thumb3	
JtThumbDRt	r_thumb4	
JtNeckA	spine4	

JtNeckB	spine5	
JtSkullA	skullbase	
JtEyeLf	eyeball_left	
JtEycRf	cycball_right	
JtEyelidLowerLf	lower_eyelid_left	
JtEyelidLowerRf	lower_eyelid_right	

Anexo D

Análise ficheiro e restrições

Dispomos, em seguida, de uma análise em pormenor do ficheiro das restrições enunciado no Capítulo 5.

Instrução	Dependências	Significado
import math		Importação da biblioteca matemática
scene.setScale(1.0)	SmartBody::SBScene::setScale()	Define a escala da cena em metros
scene.addAssetPath('mesh', 'mesh')	SmartBody::SBScene::addAssetPath ()	Acrescenta *asset paths*. É onde pode encontrar o diretório que contem os *scripts* que vão ser executados
scene.addAssetPath("script", "behaviorsets")	SmartBody::SBScene::addAssetPath ()	
scene.addAssetPath('script', 'scripts')	SmartBody::SBScene::addAssetPath ()	
scene.loadAssets()	SmartBody::SBScene::loadAssets()	Carrega os *scripts*
scene.run('default-viewer.py')	run SmartBody::SBScene	Executa o *default viewer* da camara.
camera = getCamera()		
camera.setEye(-0.193661, 2.12169, 4.92749)	setEye SmartBody::Camera	
camera.setCenter(-0.0138556, 1.1562, 1.2083)	setCenter SmartBody::Camera	
camera.setFarPlane(100)		

camera.setNearPlane(0.1)		
scene.run('zebra2-map.py')	run SmartBody::SBScene	*Joint map* para o *Brad*.
zebra2Map = scene.getJointMapManager().getJointMap('zebra2')	getJointMapManager SmartBody::SBScene	
bradSkeleton = scene.getSkeleton('ChrBrad.sk')	SmartBody::SBScene::getSkeleton	
zebra2Map.applySkeleton(bradSkeleton)	applySkeleton SmartBody::SBJointMap	
zebra2Map.applyMotionRecurse('ChrBrad')	applyMotionRecurse SmartBody::SBJointMap	
posX = -1.45;		Cria os vários *Brads*.
for i in range(4):		
baseName = 'ChrBrad%s' % i		
brad = scene.createCharacter(baseName, '')	createCharacter SmartBody::SBScene	
bradSkeleton = scene.createSkeleton('ChrBrad.sk')	SmartBody::SBScene::createSkeleton()	

brad.setSkeleton(bradSkeleton)	SmartBody::SBJoint::setSkeleton()	
bradPos = SrVec(posX + (i * 1.00), 0, 0)		Determina a posição.
brad.setPosition(bradPos)	setPosition SmartBody::SBPawn	
brad.createStandardControllers()	createStandardControllersSbmChara cter	Configuração de controladores standard.
brad.setDoubleAttribute('deformableMeshScale ', .01)	setDoubleAttributeSmartBody::SBO bject	*Deformable mesh*
brad.setStringAttribute('deformableMesh', 'ChrMaarten.dae')	setStringAttributeSmartBody::SBObj ect	
bml.execBML(baseName, '<body posture="ChrBrad@Idle01"/>')	execBMLSmartBody::SBBmlProces sor	Executa animação inicial.
if i== 0 :		*Retarget character*
scene.run('BehaviorSetReaching.py')	run SmartBody::SBScene	
setupBehaviorSet()		
retargetBehaviorSet(baseName)		

scene.getPawn('camera').setPosition(SrVec(0, -.50, 0))	SmartBody::SBScene::getPawn() setPosition SmartBody::SBPawn	Posição da camara.
for name in scene.getCharacterNames():	getCharacterNames SmartBody::SBScene	GPU *deformable geometry* para todos.
scene.getCharacter(name).setStringAttribute("displayType", "GPUmesh")	SmartBody::SBScene::getCharacter( ) setStringAttribute SmartBody::SBObject	
for i in range(5):		Cria vários objetos.
baseName = 'pawn%s' % i		
pawn = scene.createPawn(baseName)	createPawn SmartBody::SBScene	
pawn.setStringAttribute('collisionShape', 'sphere')	setStringAttribute SmartBody::SBObject	
collisionShapeScale = SrVec(.05,.05,.05)		
pawn.getAttribute('collisionShapeScale').setValue(collisionShapeScale)	SmartBody::SBObject::getAttribute SmartBody::Vec3Attribute::setValue	

scene.getPawn('pawn0').setPosition(SrVec(1.30, 1.30, .15))	SmartBody::SBScene::getPawn() setPosition SmartBody::SBPawn	Posição dos objetos e escala da colisão.
scene.getPawn('pawn1').setPosition(SrVec(1.90, 1.30, .15))		
scene.getPawn('pawn2').setPosition(SrVec(-1.00, 1.75, 1.50))		
scene.getPawn('pawn3').setPosition(SrVec(-1.10, 1.55, .43))		
scene.getPawn('pawn4').setPosition(SrVec(-.10, 1.55, .43))		
bml.execBMLAt(1, 'ChrBrad0', '<sbm:reach sbm:action="touch" target="pawn3"/>')		2 *Brads* da esquerda tocam no objeto.
bml.execBMLAt(1, 'ChrBrad1', '<sbm:reach sbm:action="touch" target="pawn4"/>')		2 *Brads* da esquerda tocam no objeto.
bml.execBMLAt(2, 'ChrBrad0', '<sbm:constraint effector="l_wrist" sbm:effector-root="l_sternoclavicular" sbm:handle="cbrad0" target="pawn3" sbm:fade-in="0.5"/>')		Restrição para o primeiro *Brad*.

bml.execBMLAt(2, 'ChrBrad3', '<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:handle="cbrad3" target="pawn0" sbm:fade-in="0.5"/>')		*Brad* da direita com restrições no pulso direito.
bml.execBMLAt(2, 'ChrBrad3', '<sbm:constraint effector="l_wrist" sbm:effector-root="l_sternoclavicular" sbm:handle="cbrad3" target="pawn1" sbm:fade-in="0.5"/>')		*Brad* da direita com restrições no pulso esquerdo.
bml.execBMLAt(5, '*', '<gaze target="pawn2"/>')		Todos os *Brads* seguem o objeto2.
gazeX = -2.00		Objeto 2 a circular.
dir = 1		
speed = .01		
pawn2 = scene.getPawn('pawn2')	SmartBody::SBScene::getPawn()	
class ConstraintDemo(SBScript):	SmartBody::SBScript	
def update(self, time):		

global gazeX, dir, speed						
if gazeX > 2.00:						Objeto muda de direção quando atinge os limites.
dir = -1						
elif gazeX < -2.00:						
dir = 1						
gazeX = gazeX + speed * dir						
pawn2.setPosition(SrVec(gazeX, 1.75, 1.50 * math.sin(time)))						

Bibliografia

[1] Andrew W Feng, Yuyu Xu, and Ari Shapiro. An example-based motion synthesis technique for locomotion and object manipulation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 95–102. ACM, 2012.

[2] William Rowan Hamilton. Ii. on quaternions; or on a new system of imaginaries in algebra. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 25(163):10–13, 1844.

[3] Alexis Heloir and Michael Kipp. Embr–a realtime animation engine for interactive embodied agents. In *International Workshop on Intelligent Virtual Agents*, pages 393–404. Springer, 2009.

[4] Irving P Herman. *Physics of the human body*. Springer, 2007.

[5] IIIM. BML 1.0 standard. `http://www.mindmakers.org/projects/bml-1-0/wiki`, 2016. Consultado: 1/10/2015.

[6] Ishizuka Lab. MPML. `http://www.miv.t.u-tokyo.ac.jp/MPML/mpml.html`, 1998. Consultado: 2/10/2015.

[7] Stefan Kopp, Brigitte Krenn, Stacy Marsella, Andrew N Marshall, Catherine Pelachaud, Hannes Pirker, Kristinn R Thórisson, and Hannes Vilhjálmsson. Towards a common framework for multimodal generation: The behavior markup language. In *International Workshop on Intelligent Virtual Agents*, pages 205–217. Springer, 2006.

[8] Sumedha Kshirsagar, Nadia Magnenat-Thalmann, Anthony Guye-Vuillème, Daniel Thalmann, Kaveh Kamyab, and Ebrahim Mamdani. Avatar markup language. In *ACM International Conference Proceeding Series*, volume 23, pages 169–177, 2002.

[9] Jina Lee, David DeVault, Stacy Marsella, and David Traum. Thoughts on fml: Behavior generation in the virtual human communication architecture. *Proceedings of FML*, pages 83–95, 2008.

[10] David J Magee. *Avaliação musculoesquelética*. Manole, 2010.

[11] John C Martin. *Introduction to Languages and the Theory of Computation*, volume 4. McGraw-Hill NY, 1991.

[12] Paulo Blauth Menezes. *Linguagens formais e autômatos*. Sagra-Dcluzzato, 1998.

[13] Catherine Pelachaud, Jean-Claude Martin, Elisabeth Andre, Gérard Chollet, Kostas Karpouzis, and Danielle Pelé. *Intelligent Virtual Agents: 7th International Working Conference, IVA 2007, Paris, France, September 17-19, 2007, Proceedings*, volume 4722. Springer Science & Business Media, 2007.

[14] Patricia Potter. *Fundamentos de enfermagem*. Elsevier Brasil, 2014.

[15] S Lab. VHML. `http://www.vhml.org/`, 2015. Consultado: 1/10/2015.

[16] Marcus Thiebaux, Stacy Marsella, Andrew N Marshall, and Marcelo Kallmann. Smartbody: Behavior realization for embodied conversational agents. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, pages 151–158. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[17] USC ICT. Smartbody. `http://smartbody.ict.usc.edu/`, 2016. Consultado em 1/10/2015.

[18] USC ICT. Virtual human toolkit. `https://vhtoolkit.ict.usc.edu/`, 2016. Consultado em 2/10/2015.

[19] Hannes Vilhjálmsson, Nathan Cantelmo, Justine Cassell, Nicolas E Chafai, Michael Kipp, Stefan Kopp, Maurizio Mancini, Stacy Marsella, Andrew N Marshall, Catherine Pelachaud, et al. The behavior markup language: Recent developments and challenges. In *International Workshop on Intelligent Virtual Agents*, pages 99–111. Springer, 2007.