CENTRO UNIVERSITÁRIO FEI

GUILHERME CARDOSO COELHO

MODELAGEM DE ROBÔ HUMANOIDE EM SOFTWARE DE SIMULAÇÃO

São Bernardo do Campo

2021

GUILHERME CARDOSO COELHO

MODELAGEM DE ROBÔ HUMANOIDE EM SOFTWARE DE SIMULAÇÃO

Relatório Final de Iniciação Científica apresentado ao Centro Universitário FEI, como parte dos requisitos do Programa PIBIC-FEI. Orientado pelo Prof. Danilo H. Perico.

São Bernardo do Campo

2021

RESUMO

O intuito deste projeto é reproduzir um robô humanoide com 90 cm de altura em um software de simulação, e com isso, aumentar as possibilidades de criação do processo de controle deste robô, denominado de humanoide TeenSize. Essa aplicação nos permite fazer testes e desenvolver programas de forma virtual para simular toda movimentação e dinâmica, avaliar sensores e atuadores, bem como localização, visão computacional e diversas outras aplicações. A necessidade de desenvolver um controle mais sofisticado para o projeto, nos levou à modelagem 3D, processo pelo qual pode-se replicar robôs reais em virtualizações e revelar situações muito próximas da realidade. Todas essas ferramentas podem ser úteis mesmo antes de possuir o robô físico.

Palavras-chave: Robótica Humanoide. Modelagem em simulador. Controle.

ABSTRACT

The purpose of this project is to reproduce a humanoid robot with 90 cm tall in a simulation software, and thus increase the possibilities of creating the control process of this robot, called TeenSize Humanoid. This aplication allows us to make tests and develop programs in a virtual way to simulate all the moviment and dynamics, evaluate sensor and actuators, as well as localization, computational vision and many other application. The need of develop a sofisticated control for the project, had taken us to 3D modeling, process by which one can replicate real robots in virtualizations and reveal situations very close to reality. All this tools can be useful even before you get the physical robot.
Keywords: Humanoid Robotics. Modeling in simulator. Control.

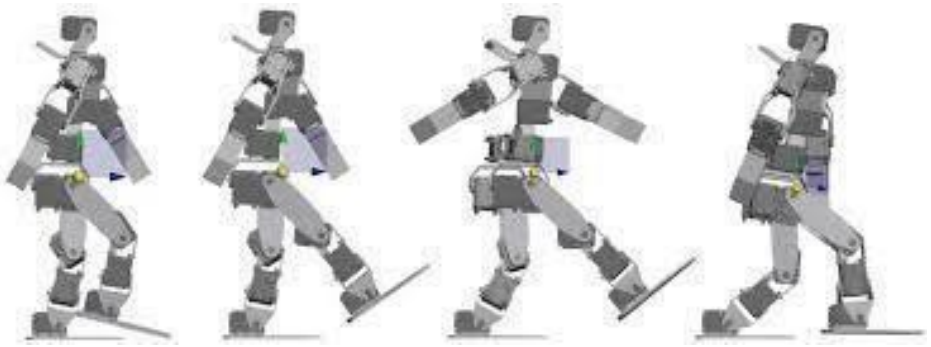
LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Usualmente, quando se trata de robôs as pessoas associam muito com a ficção e conhecimentos superficiais, mas desconhecem o fato de que o mundo da robótica é amplamente diversificado e complexo. Eles podem ser autônomos, ou controlados remotamente, podem nadar, voar, até mesmo caminhar ou jogar futebol. Essas características individuais diferenciam os métodos de desenvolvimento de cada projeto robótico e exigem demasiadamente de uma harmonia com o controle e a movimentação. Baseado nessa ideia de mobilidade a academia têm voltado suas atenções para desenvolvimento de robôs autônomos móveis com cada vez mais graus de liberdade e independência remota (WOLF et al., 2009).

Figura 1 – Robô humanoide em simulação de análise de centro de massa.



Fonte: http://sistemaolimpo.org.

Um ponto de partida na descoberta de tecnologias eficazes para esses problemas de mobilidade é o robô humanoide. Através dele surgem muitas ideias e novas formas de tecnologia em busca de aproximações das características humanas, seja conversar, correr, enxergar, localizar, se equilibrar (PERICO, 2014). Entretanto, tais pesquisas exigem investimento e muitos pesquisadores têm dificuldades em seguir com os projetos. Visto isso, a oportunidade de desenvolver qualquer projeto de robótica se tornou mais cômoda nos últimos anos devido a simulação de robôs em software. Principalmente, o aparelhamento aos engines de física que utilizam conhecimentos adquiridos na área de dinâmica do corpo rígido e cinemática aplicada para trazer veracidade as simulações. Um exemplo é o ODE que é utilizado pelo Webots para essa funcionalidade e que permite a movimentação do robô virtual, visualização de vetores cinemáticos, aceleração linear, quedas, lançamentos, e é bastante utilizado em jogos, simulação 3D, modelagem e mais (SMITH, 2004). O importante é salientar que a virtualização e os processos de simulação apenas exibem o que foram programados para exibir, e existem limitações quanto ao seu uso devido a subjetividade que a física real impõe, entretanto é um ponto de partida

excelente que pode apresentar muitas situações condizentes com a realidade e com o passar dos anos elas se tornam cada vez mais realistas (WEBOTS, 2019e). A Fig. 1 representa um robô humanoide no processo de análise do seu centro de massa.

Baseado na evolução dos mecanismos de inteligência artificial na década de 90 aconteceram algumas evoluções significativas que revolucionaram a forma de pensar e implementar essas tecnologias. O algoritmo IBM Deep Blue venceu o grande jogador de xadrez então campeão mundial Gary Kasparov. No mesmo ano a NASA realizou uma aterrissagem de sucesso em marte com o robô Sojourner, e então, com estes acontecimentos a RoboCup surge com uma iniciativa de criar robôs humanoides capazes de vencer o time campeão da Copa do Mundo de Futebol(ROBOCUP, 2019). A pesquisa envolvida para que este objetivo seja alcançado exige processos de controle decisório, localização e visão computacional. Muitos avanços foram otimizados nos últimos anos em relação a visão computacional e detecção de objetos por parte da equipe de robótica da FEI e outras pesquisas que acontecem simultaneamente ao redor do mundo que contemplam todos os assuntos que permeiam o mundo da robótica em humanoides, e que colocam o objetivo da RoboCup cada vez mais próximo de ser alcançado.(OLIVEIRA et al., 2019)

Dito isto, para contribuir com a maneira de enxergar a robótica, este artigo tem o objetivo de realizar a modelagem de um robô humanoide de 90 cm de altura em um software de simulação. O software escolhido para desenvolvimento deste modelo é o Webots.

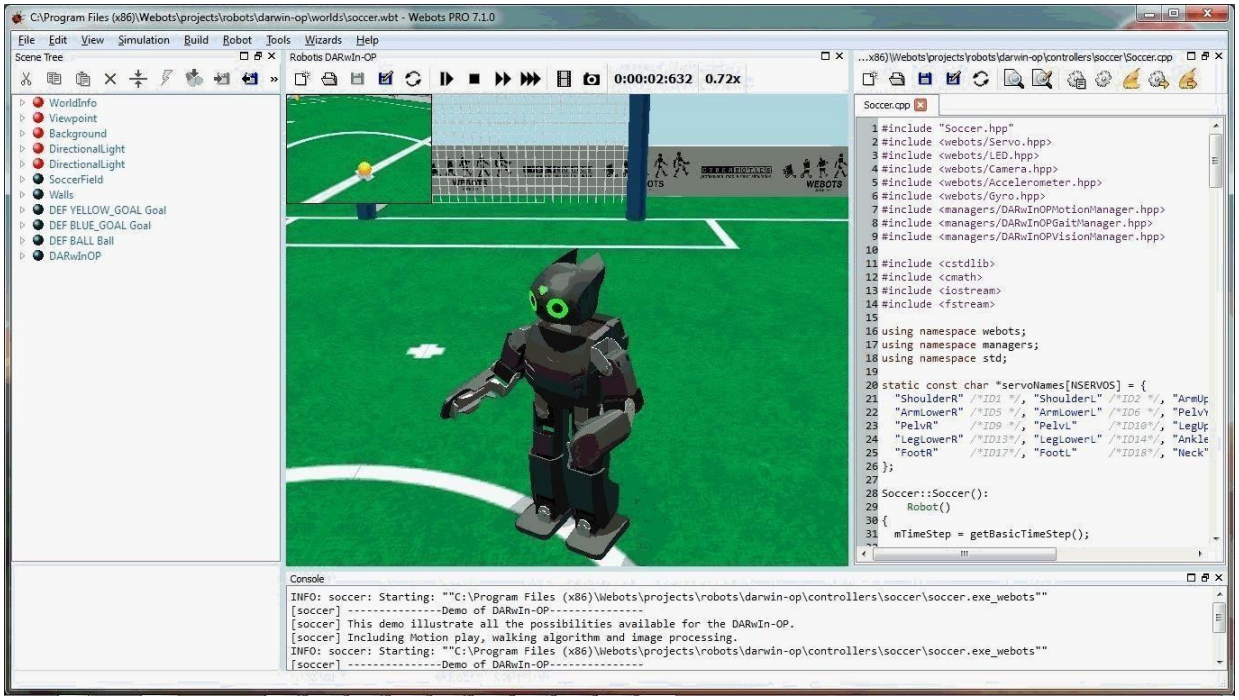
2 REVISÃO BIBLIOGRÁFICA

Existem muitas vantagens em realizar um protótipo a partir de um software de simulação entre elas tornar o projeto exequível e a um custo muito mais baixo, pois toda a parte manual e desgastante pode ser substituída a partir de testes virtuais, a programação responsável por realizar os movimentos são facilmente testadas e alteradas de acordo com as especificações, e quaisquer mudanças durante o desenvolvimento do projeto pode ser alterada sem custos (WEBOTS, 2019e; SILVA et al., 2017). Desta maneira a aplicação pode ser realizada em qualquer lugar não precisando necessariamente de um laboratório, fontes, cabos, e sim apenas, de um notebook e o software instalado. Durante muito tempo as tecnologias foram desenvolvidas na forma de "tentativa e erro" que certamente é um método funcional, e como se sabe esse é o processo natural para uma confirmação teórica em qualquer ramo da ciência, porém às vezes, podemos tomar alguns atalhos menos custosos e mais eficientes na resolução dos nossos problemas. Não se trata de abandonar a ideia das tentativas, pois isso não seria possível, mas sim, adotar uma alternativa à realidade do laboratório e do robô físico, que é a simulação de robótica móvel. A ideia de trazer um simulador vem para diminuir as tentativas manuais e substituir por ambientes virtualmente criados e adaptados, com o intuito de emular uma situação fiel a realidade que possibilita o teste dos códigos, enxergar adaptações mecânicas, a utilização e testes de sensores, ainda que, não os tenha instalado no robô físico, o processo de odometria/localização, interação com outros robôs, testes de visão, e diversas etapas de criação que executados na vida real tornariam o processo de desenvolvimento improdutivo.

2.1 MODELAGEM 3D E WEBOTS

A modelagem 3D é bastante recomendada para aprimorar todas as vias de criação do projeto como um todo, sendo realizado em softwares especializados como Webots, Gazebo, Blender, Microsoft Robotics Developer Studio e muitos outros (BACCARIN, s.d.). O processo se dá na escolha de um robô virtual sendo que o Webots já possui um banco de dados com diversos exemplos. Ainda tem a opção de importar um modelo 3D feito em um software de desenho, então segue a preparação e layout do ambiente virtual selecionando luminosidade, obstáculos, objetos, texturas do solo, mapeamento via computação gráfica, escolha de sensores e todo aparato necessário para a simulação desejada até a implementação do código que pode

Figura 2 – Simulação do robô Darwin.



Fonte: www.software.com.br.

ser escrito nas principais linguagens de programação. A Fig.2 é uma demonstração de como é a interface do programa, e neste teste vemos o robô Darwin.

A escolha do Webots foi uma recomendação da equipe por ser um software de prototipagem muito versátil e de fácil manuseio, open source que pode ser utilizado na criação de ambientes simples para máquinas de menor dimensão, e também ambientes mais complexos com robôs maiores (WEBOTS, 2019d). O programa ocupa menos memória e também roda em computadores não muito potentes se comparado aos outros, e como já mencionado possui um engine de física bem consolidado, pode-se importar modelos 3D e cenários que corroborem num ambiente verossímil. O software tem boa aceitação tanto da indústria quanto da academia com grandes empresas e universidades de renome, além de que pode ser adquirido por qualquer um.

O software Webots tem uma subdivisão de estrutura: VRML, componentes e controladores. A extensão VRML(Virtual Reality Modeling Language) que é um formato de arquivo responsável por conter diversos objetos de virtualização 3D sendo compatível com o formato ".wrl", essa linguagem permite a leitura de cenários virtuais criados no programa que contém formas geométricas e propriedades físicas de diversas ordens de complexidade, os componentes representam os fatores que compõem o ambiente como exemplo de um jogo de futebol:

obstáculos, campo, robô, bola etc. Os controladores são arquivos binários responsáveis por dar vida, movimento e animação aos objetos virtualizados, estes podem ser escritos em Java, C++, Python e MATLAB (ZANNATHA et al., 2011). Além de tudo o software possui um repositório no Github com documentação compartilhável de diversas ferramentas que podem ser acrescentadas na própria simulação.

2.2 REAL X VIRTUAL

Em inúmeras situações as pessoas podem ter dúvidas a respeito da simulação e a quão próxima ela pode chegar da realidade. Existem muitos testes que podem ser feitos para demonstrar isso, estes que foram realizados em outras pesquisas. Velocidade máxima, amortecimento, o efeito do controlador P, o máximo torque, e a resposta do torque, controle dos servos pelo torque, para cada um destes é feito um levantamento do comportamento de alguns mecanismos em simulação e na vida real. O teste de velocidade consiste em mover um dos servos do robô, de forma que esteja livre de colisões e rotacione livremente sem resistências mecânicas. O teste utiliza um movimento como padrão, e é repetido por algumas vezes com velocidades diferentes, o mesmo acontece no simulador, e assim, são levantados valores de ambos os testes (físico e virtual) a serem comparados. É claro que alguns responderão de forma mais parecida do que outros atuadores, e por isso não se pode generalizar tudo em apenas um motor. No amortecimento o mesmo acontece, é possível obter o levantamento real dos valores de amortecimento e o quanto pode alterar nos valores de damping para ficar mais próximo dos ideais. Numa visão geral, de tudo o que pode ser feito para melhorar o resultado da simulação chega-se à conclusão de que o robô real possui algumas particularidades que a diferem da simulação, e alguns mecanismos podem ser modificados para se aproximar mais da realidade, e outros não (DAVID; OLIVIER; IJSPEERT, 2013).

2.3 CONTROLADOR

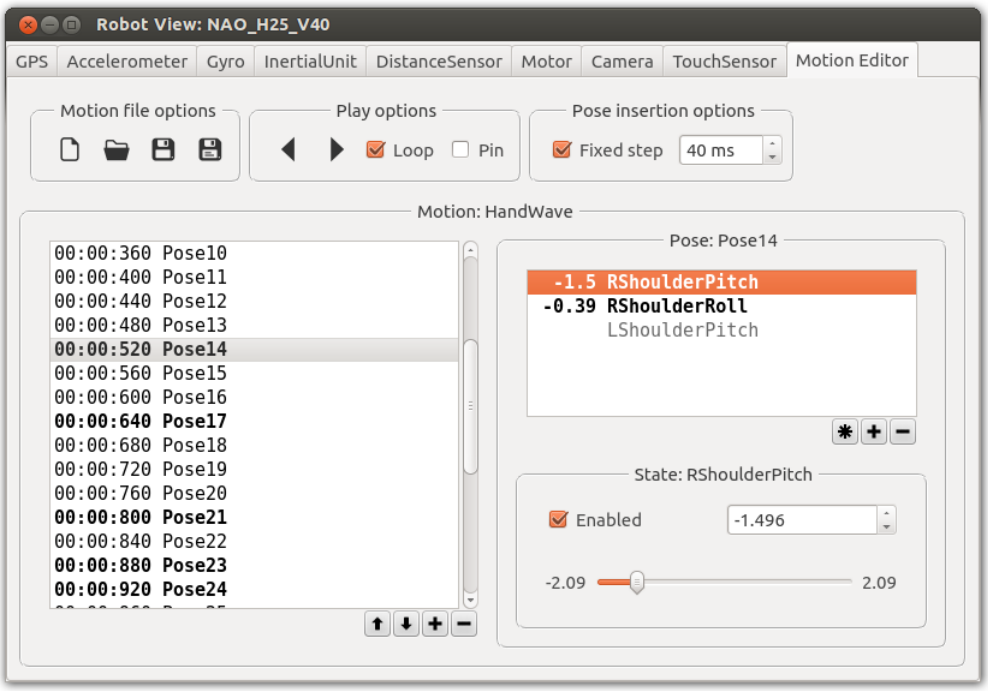
Existe um sistema disponível nesta plataforma que possibilita o uso de códigos realizados no Webots serem testados no robô real. Esta aplicação funciona como uma ponte que comunica com a API (Application Programming Interface) do Webots e o sistema do controlador responsável pelo robô físico. Para isso é utilizado um compilador-cruzado, que transforma o código desenvolvido no IDE do Webots em um executável no sistema de controle do robô

(real). Este compilador-cruzado funciona adequadamente, porém seu processamento é de certa forma lento. A comunicação entre controladores pode afetar seu desempenho. Cada leitura ou escrita de uma palavra leva 1 ms para ser processada, a configuração da posição de servomotores pode levar cerca de 20 ms, e quanto mais detalhado um parâmetro de motor ou atuadores como acelerômetro e giroscópio, mais eles podem diminuir o tempo de processamento do controlador. Algo que pode contribuir para melhorar neste processamento é deixar o código mais inteligente, por exemplo: ao invés de enviar os comandos de todos os parâmetros de controle diretamente para o servo, pode-se armazenar em uma variável, criando uma função que envia os comandos dos 19 motores em um pacote diretamente para um controlador, ao invés de, um pacote para cada comando e para cada servo. Simplificadamente se o tempo de processamento de dados para cada servo for 2 ms, com 19 servos o tempo seria de 38 ms, porém com alguns ajustes é possível que o controlador leia todos os valores em 2 ms, como se fizesse uma leitura paralela de todos ao mesmo tempo (DAVID; OLIVIER; IJSPEERT, 2013). Para um controlador Webots em sua IDE, geralmente divide-se em 3 partes. A primeira inicia a comunicação entre o controlador e a simulação, isto acontece com a função *wb_robot_init* e a terceira finaliza esta comunicação com a função *wb_robot_cleanup*. A parte intermediária acontece geralmente dentro de um loop e deve acompanhar a uma função que sincroniza os dados do controlador e da simulação, esta função é *wb_robot_step*, que possui um argumento onde é passado o valor do passo, este valor vai ser o tempo de retorno da simulação para o controlador. Sempre quiser controlar sensores e atuadores deve-se acrescentar bibliotecas compatíveis com os dispositivos para utilizar as funções próprias dos dispositivos. Para acrescentar bibliotecas no Webots com a linguagem C *#include <webots/library.h>*, *library* é substituída pela biblioteca que for necessária (WEBOTS, 2019a).

2.4 MOTION EDITOR

Uma ferramenta muito utilizada para ler valores das posições dos motores em cada etapa do movimento, serve para configurar movimentações específicas como: levantar e, chutar. O chute, por exemplo, é dividido em partes, e cada uma delas é um frame do movimento, esta ferramenta configura todas as posições dos motores em cada frame, sendo assim, na dinâmica do movimento cada etapa contribue para que ele aconteça de maneira precisa. A Fig.3 mostra um exemplo do layout da ferramenta, as poses são as etapas do movimento. Este exemplo é realizado com o protótipo do NAO disponível no software (WEBOTS, 2019c).

Figura 3 – Motion Manager.



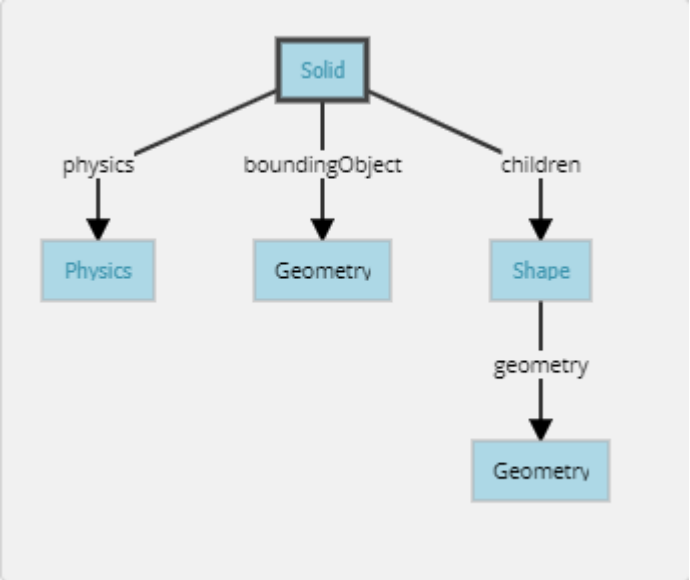
Fonte: www.cyberbotics.com

2.5 SIMULAÇÃO WEBOTS

Quando inicia o processo de criação de uma nova simulação, os arquivos são tratados como worlds. No caso esses arquivos “mundos”, como são chamados, armazenam todas as informações inseridas no ambiente. A maneira de realizar inserções é baseada num sistema de nós advindo da linguagem VRML97. Ao criar um novo projeto, ele abre um diretório world com algumas características padrão, que são alguns nós responsáveis por dar um ambiente inicial para sua simulação. Esses nós iniciais criam parâmetros modificáveis a respeito de toda a simulação, controlam os principais pontos de vistas do que será enxergado na tela, definem o plano de fundo (no caso padrão é uma paisagem com montanhas) e as luzes direcionais que simulam a iluminação do ambiente. O novo projeto por padrão estará sem nenhum robô, com um plano de fundo de montanhas e haverá uma arena quadriculada com paredes, sendo que a arena está disponível assim como diversos outros objetos na biblioteca dos nós. É interessante notar que o plugin de física está programado para simular apenas corpos rígidos. A definição de corpo rígido é a seguinte: imaginando 2 pontos quaisquer na superfície ou internamente ao objeto, diz-se que a distância entre estes pontos deverá permanecer a mesma independente da força externa exercida para tensionar o objeto. O nó responsável por simular um corpo

rígido é o Solid Node. As colisões são definidas por um objeto delimitador que acompanha os parâmetros da física (WEBOTS, 2019b). Em muitas ocasiões os nós são representados como fluxogramas, a Fig.4 mostra como criar um objeto simples definido como corpo rígido e com o formato de esfera, cilindro ou cubo, por exemplo. Onde está adotado como Geometry na figura é o lugar onde pode ser alterado para obter qualquer tipo de formato que deseja, com primitivos básicos já é possível criar muitas simulações. Para objetos mais complexos existe uma API desenvolvida pelo Webots que se comunica com outro software de modelagem, o Blender. Este software realiza modelagens mais complexas e depois existe a possibilidade de exportar o modelo para o Webots, realizando é claro alguns ajustes. Esta é uma opção excelente para quem tem familiaridade com softwares de desenho.

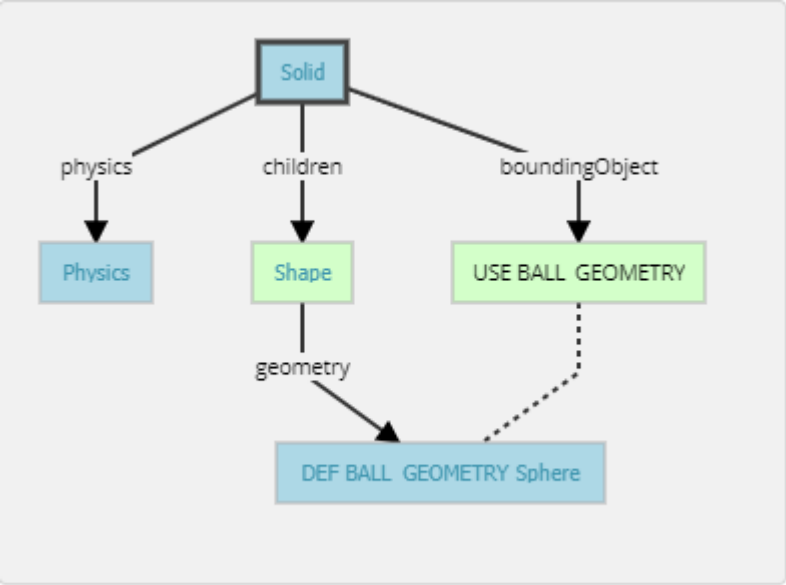
Figura 4 – Representação dos nós em fluxograma.



Fonte: www.cyberbotics.com

Para utilizar uma função específica mais de uma vez no decorrer da associação de nós, pode-se utilizar um mecanismo DEF-USE, no qual você define uma funcionalidade com o DEF e sempre que for reutilizado essa mesma operação basta utilizar USE e selecionar a DEF necessária (WEBOTS, 2019d). Desta forma todas as peças modeladas estarão conectadas e quando uma for alterada as outras também serão. Para selecionar os contornos de colisão geralmente utiliza-se USE sobre a DEF que define a peça a ser colidida. Assim como mostra a Fig.5

Figura 5 – Utilizando a função DEF-USE.



Fonte: www.cyberbotics.com

Após finalizar um projeto é possível transformar o seu modelo em um PROTO. Um protótipo que fica disponível, assim como outros robôs, na plataforma e pode ser usado em outras simulações, e se for disponibilizado por uma rede de compartilhamento como o Github, poderá ser usado por qualquer um, e até mesmo, tendo suas características melhoradas por pessoas que pode oferecer sugestões (WEBOTS, 2019d). Este nó PROTO pode ser configurado para receber alterações, basta adicionar a ele alguns campos que configuram suas características, por exemplo: massa, rotação e translação.

3 METODOLOGIA

Nesta seção serão descritos os recursos e métodos que estão sendo utilizados para produzir o conteúdo deste artigo.

3.1 RECURSOS HUMANOS E MATERIAIS

No decorrer do projeto serão usados todos os materiais disponíveis nos laboratórios direcionados à equipe de robótica da FEI, bem como, auxílio dos professores e pessoas ligadas ao projeto e principalmente os computadores do laboratório com Webots e outros softwares.

3.2 MÉTODOS

A prática consiste em reunir os desenhos das peças do robô humanoide TeenSize, que já foram desenvolvidas, fazer a importação no Webots e reproduzir todos os detalhes, inclusive todos os motores e sensores que podemos eventualmente trabalhar, a fim de criar a sua virtualização e trabalhar com o ambiente deixando - o parecido com uma situação de jogo. Então, serão utilizadas as funções do software para extrair dados de movimentação utilizando a ferramenta motion editor que configura as posições dos servos motores a cada etapa do movimento. Realizar testes de códigos que serão utilizados no cenário real. Esses testes serão de suma importância para obter uma ideia precisa de como estará o robô depois de fisicamente pronto e de como ele vai reagir as adversidades impostas numa situação real desde que sejam devidamente simuladas em softwares, o que não for simulado pode apresentar situações que não foram previstas anteriormente.

4 DESENVOLVIMENTO

O software escolhido para modelagem deste robô humanoide foi o Webots, portanto toda explicação teórica a seguir será para um desenvolvimento nesta plataforma. O programa é dividido em três partes principais: a Scene Tree, o ambiente de simulação e o controlador. A Scene Tree tem este nome pois cada etapa de modelagem é realizada através de um sistema de nodes (ou nós) que se ramificam como uma espécie de fluxograma. Estes nodes possuem uma hierarquia, sendo que um é interligado com outro. O primeiro node chamamos de "*parent*", e o nó subsequente é chamado de "*children*". Um *children* pode ser um *parent*, ou seja, pode ser uma base para construção de outra cadeia de nós. Existem vários tipos destes chamados nós (ou nodes) que podem ser utilizados para diversas finalidades, como por exemplo: *Fluid Node* para simular água, *Shape Node* para determinar os primitivos básicos que compõem o Sólido (cubo, prisma, esfera), *Solid Node*, que determina um objeto com características físicas entre muitos outros. O ambiente de simulação é a tela onde poderá ser visualizado todo processo de modelagem. E por fim, a tela mais à direita é um arquivo em branco na qual poderá ser implementado um algoritmo para comandar as ações do robô. Ao selecionar uma linguagem de programação já aparece um *template* (exemplo) para inserir seu código.

Para modelar o robô foi utilizado alguns nodes, estes que serão brevemente detalhados a seguir:

Solid

São aqueles que podem ser movimentados no ambiente de simulação e é possível configurar suas propriedades físicas para que atuem como seriam fora da simulação.

Transform

É um node de agrupamento que se baseia em relacionar o sistema de coordenadas do *children* com o *parent*.

Group

Group node é utilizado para organizar em um único nó uma quantidade de partes de um mesmo objeto, por exemplo pode – se adotar as peças de um braço ou uma perna em um *group*.

HingeJoint

Este node serve para adicionar uma junta, ou articulação. Existem algumas características deste nó como: *hingejoint parameters*, *end point* e *device*, que serão abordados mais adiante.

Robot

Esse nó é próprio para criação de quaisquer tipos de robôs, é uma extensão do *Solid*.

Com base nestas ferramentas (nós) e suas funcionalidades, muitas pessoas utilizam a modelagem para realizar experimentos e testes, seja em projetos pessoais, empresas, e inclusive em pesquisas. Pode-se criar ambientes de robôs com rodas, manipuladores, humanoides, simulações com carros autônomos, com pessoas entre outros. Esta diversidade de nós e funções permitem criar e moldar um ambiente da maneira mais apropriada sem muitos limites para a sua criatividade.

Para iniciar o processo de modelagem da maneira como foi realizado o robô humanoide de 90 cm, espera -se que o desenvolvedor possua os desenhos técnicos de cada peça que fará parte do robô, sendo estes desenhos realizados em algum software de modelagem como: Blender, Solid Works, NX-Siemens ou qualquer outro que possa exportar o desenho para o formato VRML2. Cada peça do robô deste artigo foi desenhada pelo software NX-Siemens. Com as peças no software de modelagem será necessário exportar para o formato VRML2. O VRML2 é uma linguagem de descrição de ambiente para criação de figuras tridimensionais. Na Fig.6 segue um exemplo de como utiliza-se essa linguagem para criar figuras tridimensionais com comandos programáveis, nela você pode adicionar objetos que serão agrupados formando um sólido, localização na simulação, formato, altura, raio tipo de material, como ele vai aparentar, pode ser metálico ou fosco. Cada node e características intrínsecas são colocados entre chaves, e as cores são mescladas com três campos, sendo cada um uma porcentagem das cores primárias, seguindo o código RGB, consegue ajustar as outras tonalidades.

Este tipo de arquivo utiliza tal linguagem para construir visualmente os mundos virtuais compostos por definições cartesianas dos eixos geométricos, a partir da leitura de alguns parâmetros passados para caracterizar tamanhos, formatos aparência entre muitas outras características. A forma como se realiza o processo de modelagem com todos os nós e funções é baseada nesta linguagem.

4.1 PROCESSO DE MODELAGEM DE UM ROBÔ HUMANOIDE

Um robô articulado com 19 servomotores é dividido em peças, juntas e alguns dispositivos. Para inserir articulações com atuadores, no caso servomotores, deve-se adicionar na *Scene Tree* um Robot node, pois apenas este tipo permite acoplar os dispositivos, logicamente chamados de *devices*. Estes podem ser três: *rotational motor*, *position sensor* e *brake*. O *rotational motor* pode simular diversos tipos de motores, basta alterar nos seus parâmetros conforme sua necessidade, o *positional sensor* serve para ler a posição dos motores, e geralmente é inserido

Figura 6 – Exemplo de construção de arquivo VRML.

Figura 2.1 - Primeiro arquivo VRML

```
#VRML V2.0 utf8
# The VRML 2.0 Sourcebook
# Copyright (c) 1997
# Andrea L. Ames, David R. Nadeau, and John L. Moreland
# A brown hut
Group {
    children [
        # Draw the hut walls
        Shape {
            appearance DEF Brown Appearance {
                material Material {
                    diffuseColor 0.6 0.4 0.0
                }
            }
            geometry Cylinder {
                height 2.0
                radius 2.0
            }
        },
        # Draw the hut roof
        Transform {
            translation 0.0 2.0 0.0
            children Shape {
                appearance USE Brown
                geometry Cone {
                    height 2.0
                    bottomRadius 2.5
                }
            }
        }
    ]
}
```

Fonte: www.inf.pucrs.br.

junto com o *rotational motor*. O *brake* age como um sensor que lê a intensidade das forças aplicadas nas juntas.

Para saber como escolher por qual parte do robô começar, precisa pensar no projeto a ser realizado. No caso de um robô com 4 rodas, a melhor opção será o chassi, no qual as rodas são acopladas. A recomendação do próprio portifólio do Webots com relação a robôs humanoides é de começar pelo peito, pois é a partir do peito que irão conectar as principais juntas dos braços das pernas e do pescoço. Portanto este é o primeiro passo, criar um nó Robot.

O processo de montagem é bem simples, na *children* do *Robot Node* deve ser criado um *Solid Node*, este representará todo o peito do robô. Na *children* deste *Solid Node* foi criado um *Group Node*, neste momento deve-se importar todas as peças referentes ao peito do robô, depois basta montá-las utilizando o mouse no ambiente de simulação para alocá-las como deve estar no modelo. Na mesma *children* onde foi criado este *Group* deve-se definir também 5 juntas criando *HingeJoints*, essas juntas serão ambas as pernas, os braços e o pescoço. Uma *HingeJoint* permite inserir uma articulação com um grau de liberdade e possui algumas características para se definir onde será colocada, este *subnode* se chama *HingeJoint Parameters*.

Nele deve ser definida a *anchor*, que seria o ponto exato de conexão da peça anterior e a peça subsequente, no caso o peito e uma perna por exemplo. Este parâmetro assim como outros são definidos por posições nas coordenadas xyz observáveis no ambiente de simulação, são pontos no espaço, e esses valores são obtidos por referência da translação de outras peças. Na prática será possível obter os valores de anchor utilizando o campo de translação das peças que se encontram no mesmo ponto onde será inserida a articulação.

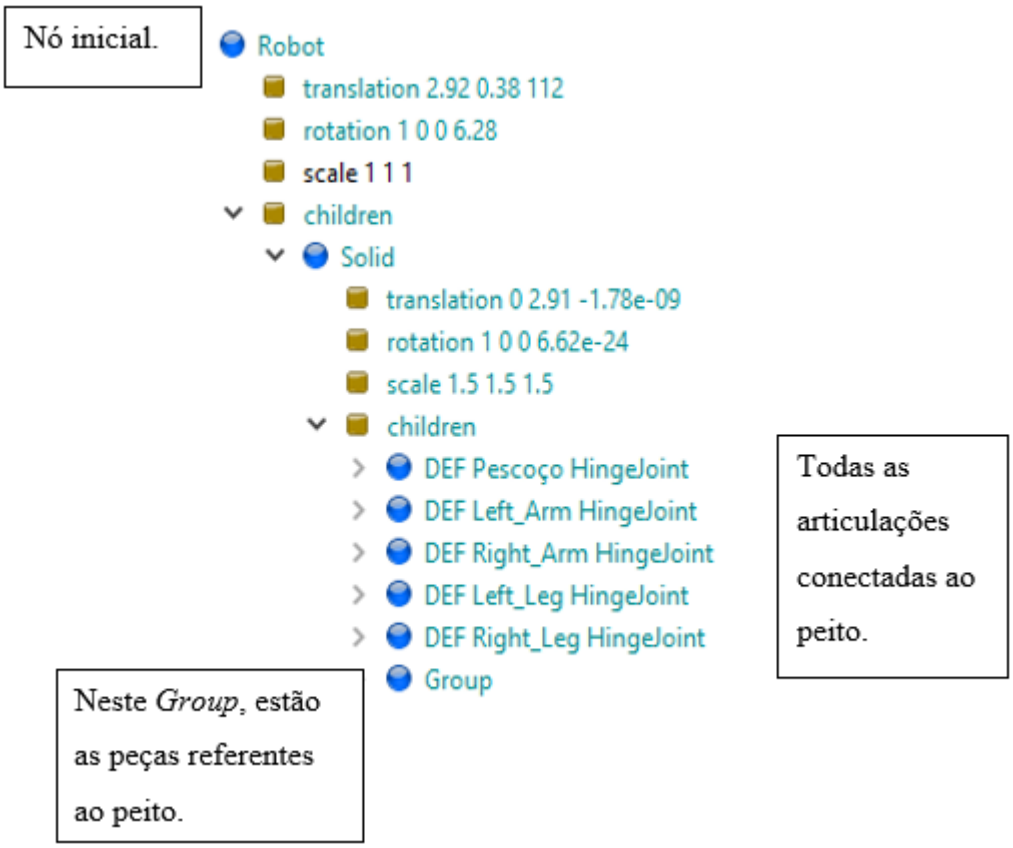
Além da anchor ainda há o axis, que é o campo responsável por determinar o eixo de rotação da peça. A *Damping constant* citada anteriormente, se trata de um fator multiplicativo da força do movimento que é aplicada sobre a junta. Este valor pode ser 0, ou seja, não há forças aplicadas, valores maiores significam que alguma parte da força será exercida diretamente na junta. No caso deste humanoide foi ajustado em 0,1 essa constante. No caso de um modelo que utilize o *brake* ele somará as forças obtidas com o sensor e a *damping constant*. Para o robô em questão foi escolhido o *Rotational Motor* com limites mínimos e máximos para simular um servomotor, atrelado a este motor foi inserido um *Positional Sensor*, um dispositivo que lê a posição do motor conforme ele rotaciona, isso é muito útil para buscar a posição do servo em algum possível algoritmo de controle.

A divisão do humanoide como já foi citada é definida pela criação de um Robot, sendo este um *Solid* que irá agrupar 5 Juntas e o *Group*, com as peças do peito, abaixo têm uma demonstração de como ficaria esta configuração inicial e em seguida como configurar uma articulação, basicamente estes são pontos cruciais para a modelagem e depois o processo se repete pra cada detalhe do robô.

Basicamente a *HingeJoint* liga um nó *parent* com um *End Point*, este último poderá ser uma peça interligada por um eixo e uma anchor. Apesar de apenas ser citado um tipo de articulação neste artigo, existem ainda alguns outros tipos para muitas outras finalidades como: *SliderJoint*, *BallJoint* e *Hinge2Joint*. O método utilizado para conectar as peças do robô é:

criar uma articulação (*HingeJoint*) adotando os parâmetros de junta com base em referências de outras peças e depois criar no *endpoint* um *Solid Node*. Em sua *children*, deverá ser inserido um *Group* que irá organizar em apenas um nó, todas as peças que vão fazer parte da sequência da modelagem até a próxima junta. Nesta mesma *children* cria-se mais uma articulação que terminará com outro agrupamento de peças até a próxima *hingejoint*. Nas Figuras 7, 8, 9 são descritos os processos já citados, de uma maneira mais visual, utilizando o próprio software.

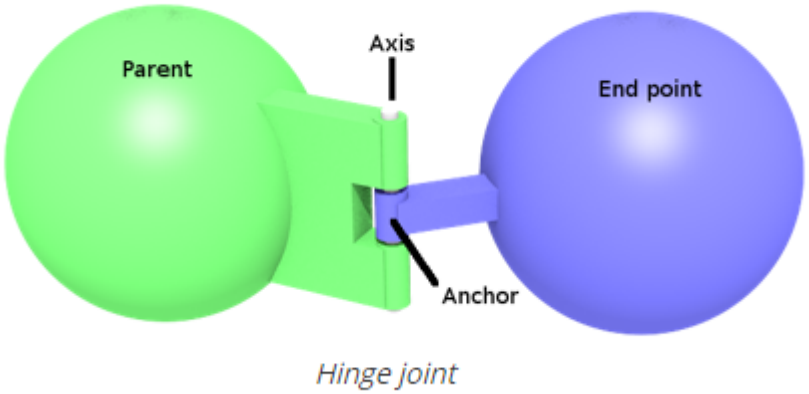
Figura 7 – Descrição de como organizar os primeiros nós.



Fonte: Autor.

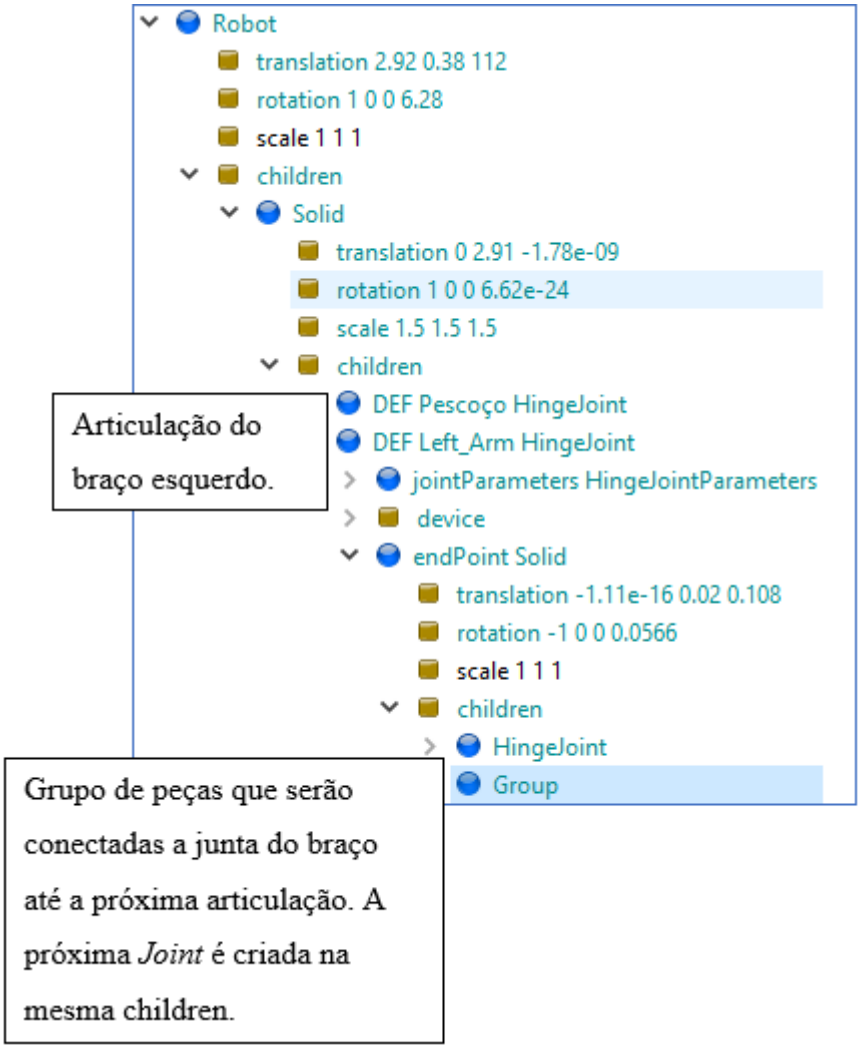
Vale salientar que no campo *device* na Fig.10, o nome adotado para o motor acoplado na junta deve ser escolhido de forma que, este seja o nome utilizado no código de programação implementado para controlar este motor. Este procedimento serve para todos os servos. Os parâmetros de posição máxima, mínima, velocidade e torque podem ser configuradas com qualquer valor, o ideal é que se faça o teste com um controlador para que se consiga obter os melhores resultados de simulação. O importante deste processo é saber configurar uma articulação, pois todo o restante é realizado da mesma maneira. Basicamente é uma junta configurada e um grupo de peças que vai conectar no *endpoint* de outro grupo de peças.

Figura 8 – Imagem didática de uma HingeJoint.



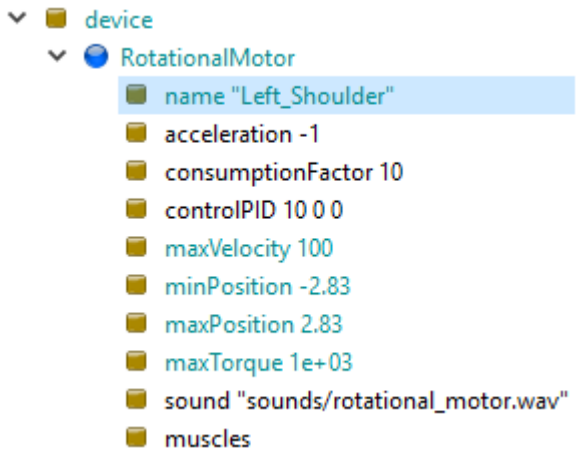
Fonte: www.cyberbotics.com.

Figura 9 – Como configurar uma articulação



Fonte: Autor.

Figura 10 – Parâmetros do dispositivo *Rotational Motor*.

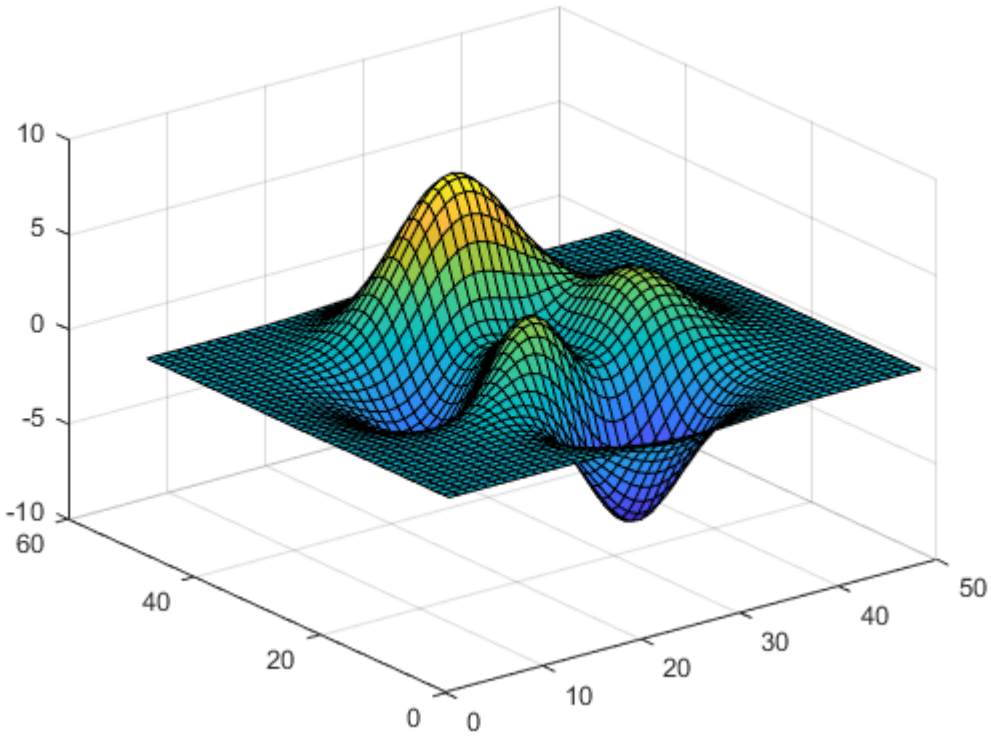


Fonte: Autor.

4.2 BOUNDING OBJECT

Este tópico aborda outro assunto importante relacionado a colisões em ambiente virtual e a física da simulação. A função *BoundingObject* determina uma delimitação no contorno das peças agrupadas, para que elas não se choquem com outros objetos da simulação ou outras peças também delimitadas. Geralmente utiliza-se primitivos básicos para delimitar objetos, pois a renderização das colisões exige menos da sua placa de vídeo. Outra maneira de fazer isso é utilizar uma função, o *IndexedFaceSet*. Esse nó acaba fazendo um contorno na peça exatamente como ela foi desenhada, sua função é fazer extrusões e criar uma peça com formatos mais complexos do que cubos e cilindros por exemplo. Em contrapartida exige-se muito mais da sua máquina para utilizar essa função no *BoundingObject*. A Fig.11 mostra eixos de coordenadas tridimensionais feitas no MATLAB como exemplo de uma deformação característica do *IndexedFaceSet*.

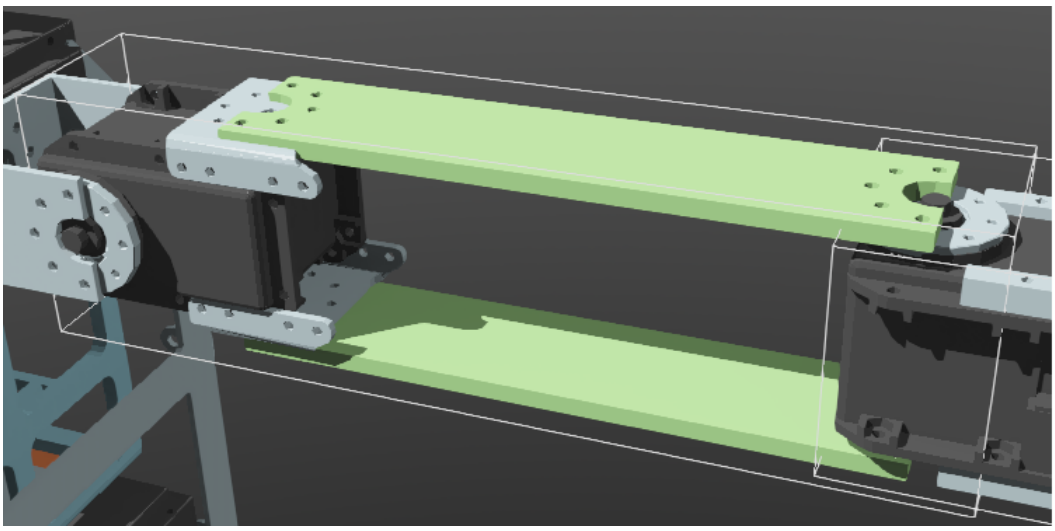
Figura 11 – Deformação de um plano em coordenadas tridimensionais.



Fonte: www.mathworks.com

Podemos observar na Fig.12, que ao redor de um agrupamento de peças temos um paralelepípedo delimitando esse trecho para que ao configurar a física do modelo este braço não colida com outros sólidos. Isso acontece com todos os agrupamentos de peças do robô.

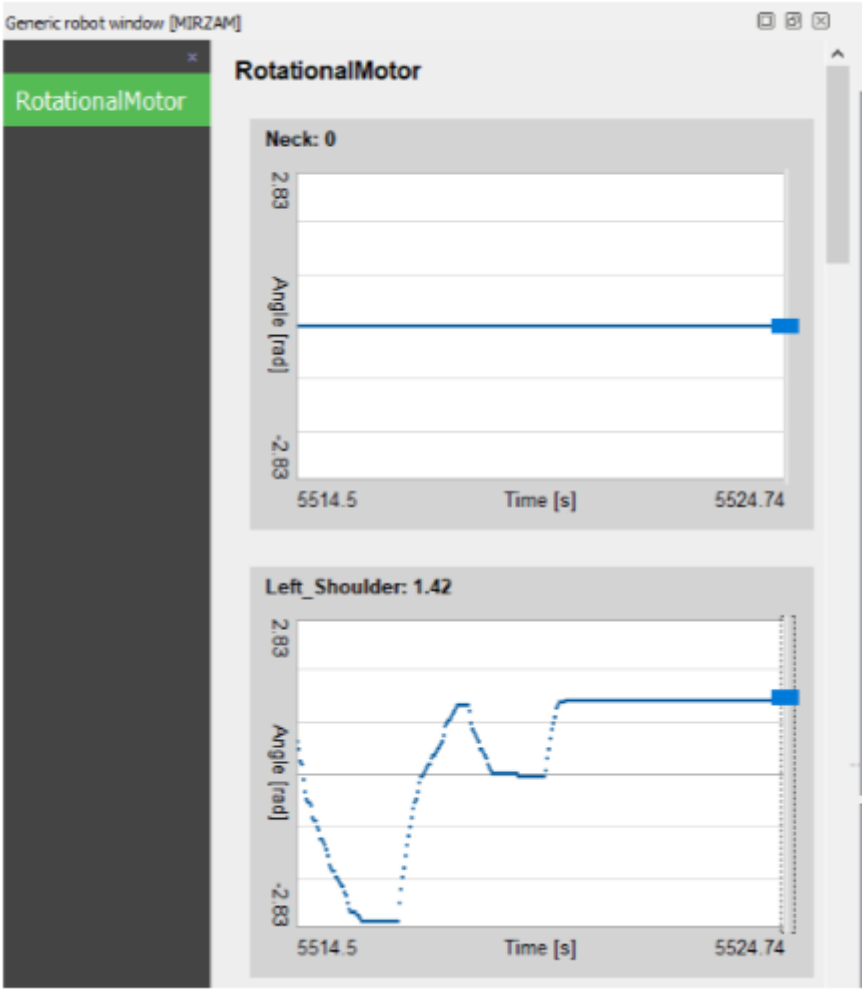
Figura 12 – *Bounding Object* destacado no paralelepípedo que delimita a peça.



Fonte: Autor.

Na Fig.13 vemos a scene tree o ambiente de simulação e uma outra tela ainda não citada, a *Generic Window*, que é muito útil para testar se a junta definida está no lugar correto. Geralmente podem ocorrer erros e a peça girar em torno do eixo errado, ou estar em alguma outra coordenada com relação a onde ela deveria estar de fato na articulação. Nesta tela conseguimos alterar a posição 0 do motor para qualquer posição desejada dentro dos limites estabelecidos nos parâmetros do dispositivo, e por isso conseguimos fazer ajustes finos alterando os valores das coordenadas da *anchor* e *axis*. Nesta janela de controle na Fig.13 nos mostra um motor em sua posição original e outro com valores alterados para visualizarmos no ambiente de simulação como a articulação está reagindo aos parâmetros pré-configurados.

Figura 13 – *Generic Window* para análise dos motores e sensores.



Fonte: Autor.

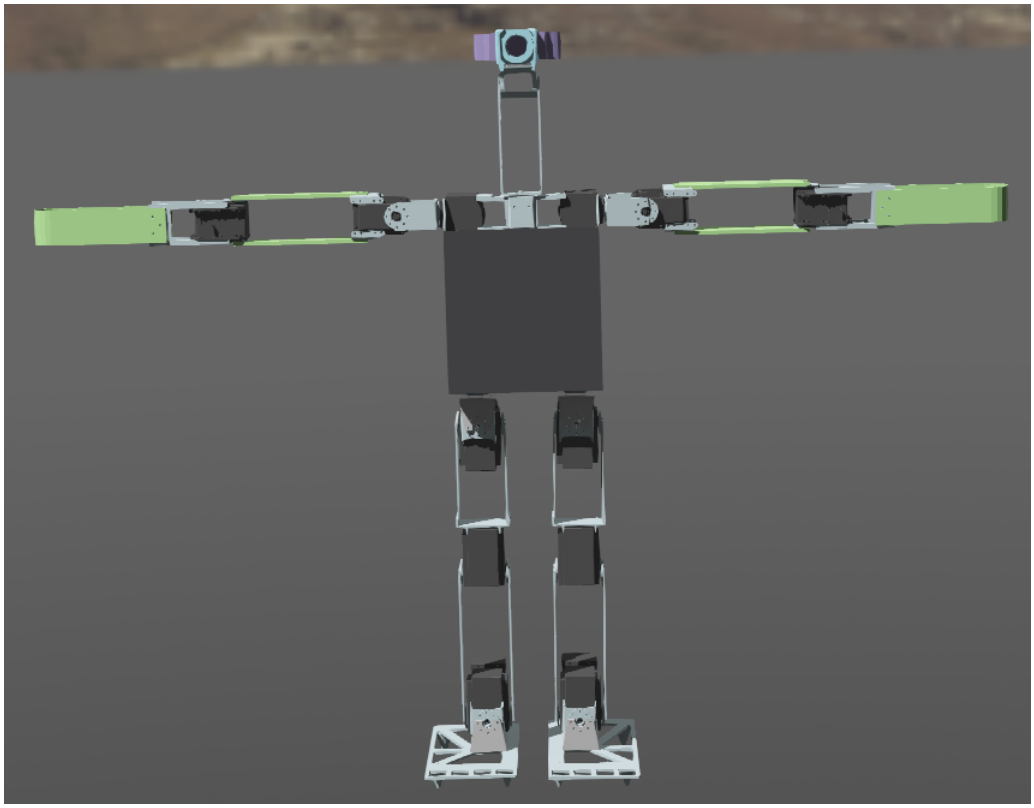
4.3 FÍSICA

Um tópico muito importante a ser tratado em todo este processo é fazer com que o seu modelo seja capaz de responder de forma semelhante à realidade e por isso é fundamental que sejam configurados valores reais. Cada *Group* que foi criado já foi modelado e delimitado com o *bounding object*, agora cada agrupamento vai ser submetido a física da simulação desempenhada pelo ODE. Para isso precisa-se atentar a 2 fatores: a massa e o centro de massa da configuração de peças. Isso significa que para obter uma simulação fundamentada, é necessário obter as massas de cada peça, muitos softwares de modelagem fornecem os valores de massa e centro de massa de acordo com o material utilizado. Ainda é possível ajustar a densidade do material, porém não é preciso configurar a massa e a densidade, quando um valor é fornecido não precisa passar ou outro, sendo assim o que não for passado ficará com -1, ou seja desativado. Neste caso existe uma certa confusão para configurar o centro de massa, pois os valores dados pelo software onde o desenho foi realizado são com referência ao plano criado para modelar, por esta razão se estiver com dúvidas a respeito dos valores de centro de massa ou matriz inercial, optar por não colocar nada é melhor do que colocar valores errados, e também não haverá grandes complicações, pois o ODE possui uma configuração padrão caso não seja fornecido estes dados.

5 RESULTADOS E DISCUSSÃO

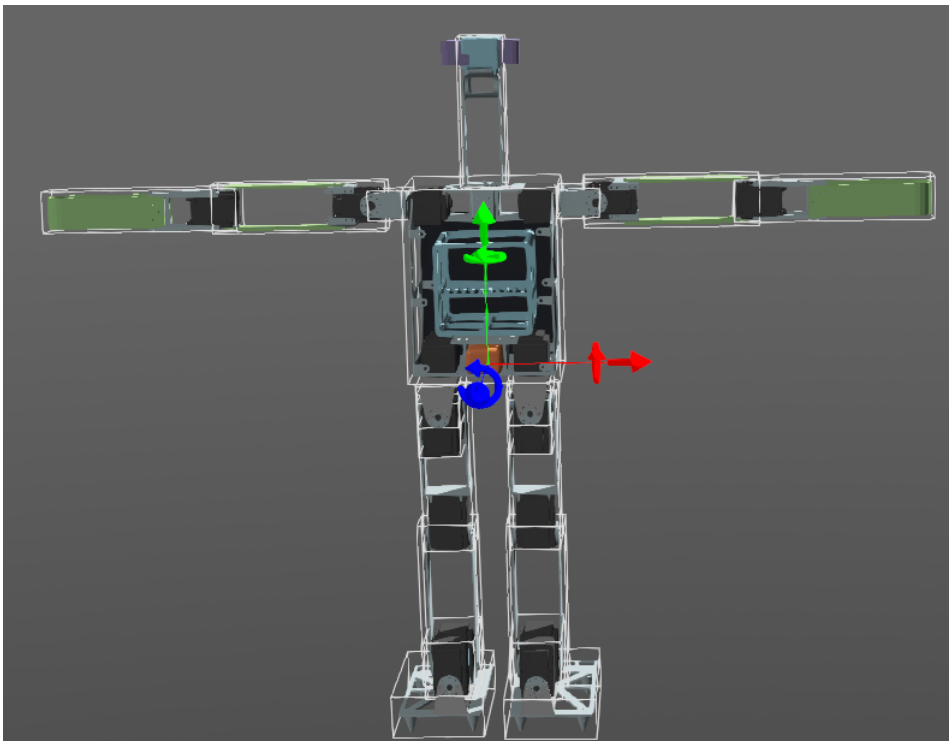
O resultado do trabalho, é um modelo completo, e funcional. Foram configuradas física conforme os dados que tínhamos, além dos testes com controladores que demonstraram resultado satisfatório. A Fig.14 mostra o estado atual da modelagem, é uma visão frontal. Essa posição do robô com os braços abertos é a posição padrão a qual foi modelada, porém é possível alterar a rotação de cada servo para ficar na posição desejada. Já a Fig.15 é a visão de costas do modelo dando destaque para os detalhes das costas, bateria, caixa onde armazena o NUC, e também os retângulos delimitadores do *bounding object*. As setas no robô podem ser usadas para mover e rotacionar em qualquer um dos eixos, ela está localizada no centro de massa aproximado do modelo. A Fig.16 é uma simulação de um modo de jogo com mais de um robô e a Fig.17 ele está direcionado à bola no campo numa posição habitual de jogo.

Figura 14 – Modelo em seu padrão *offset*.



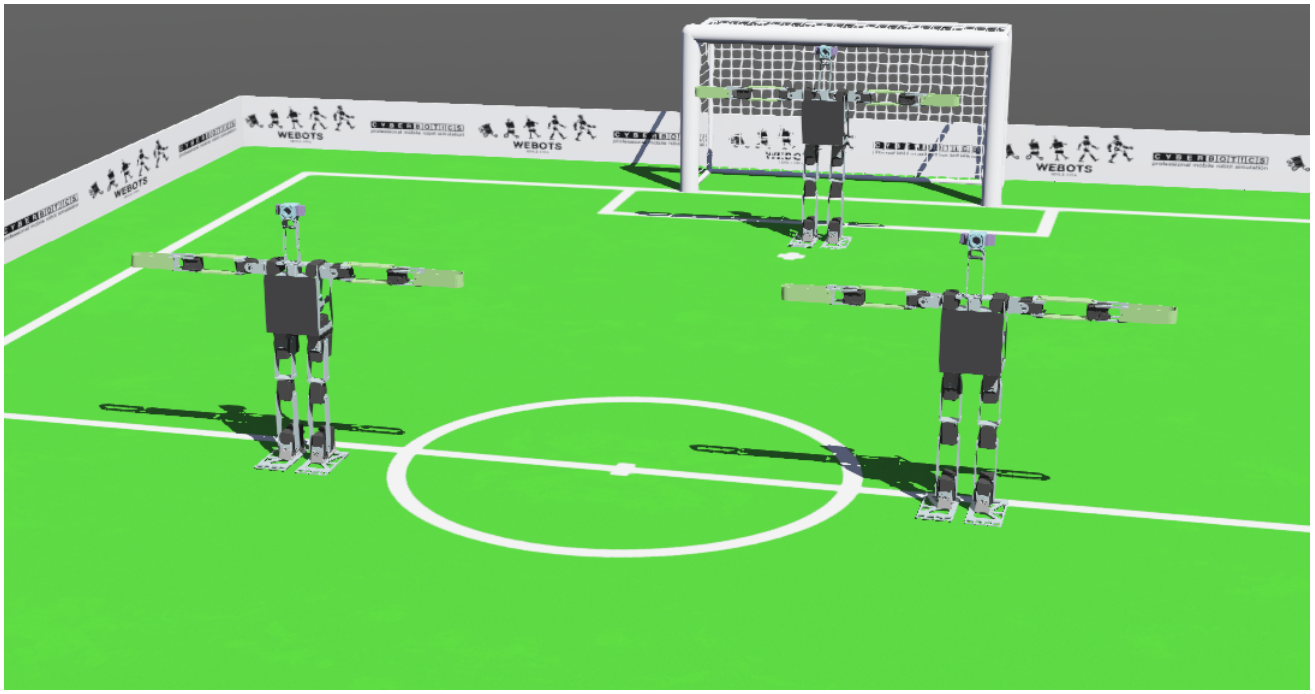
Fonte: Autor.

Figura 15 – Modelo com detalhes em destaque.



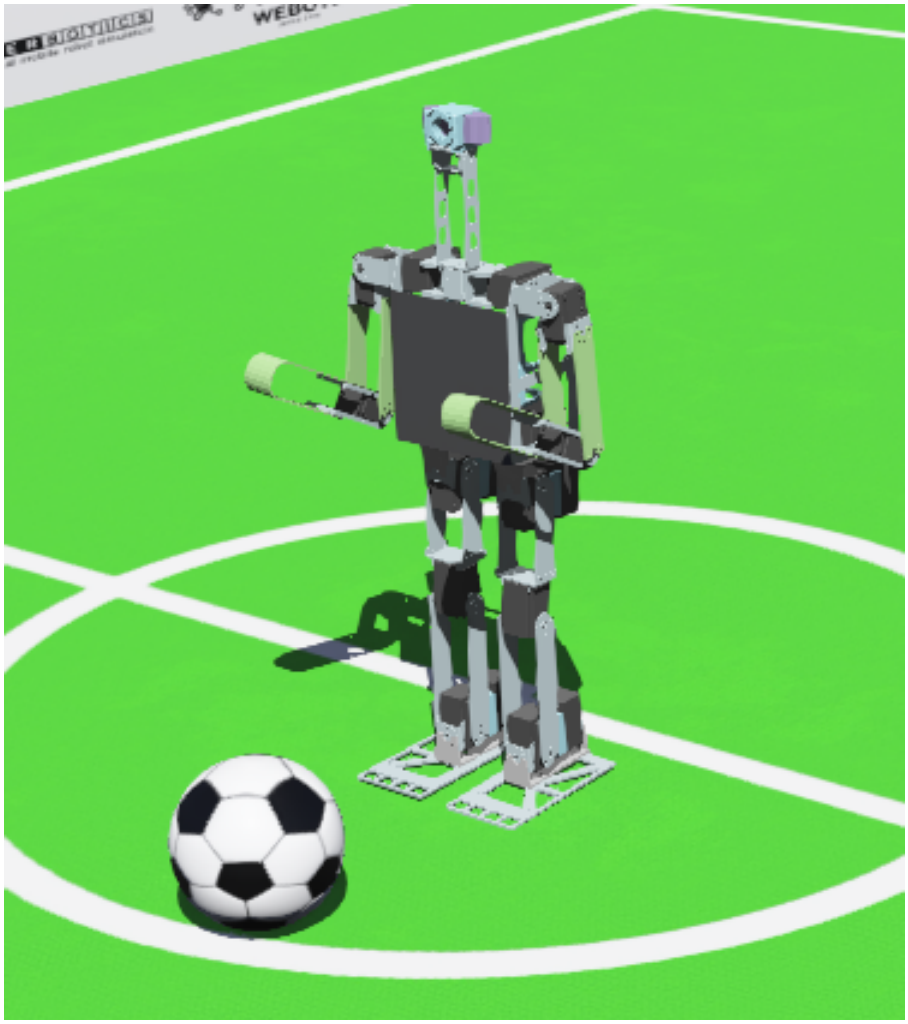
Fonte: Autor.

Figura 16 – Simulação de jogo com 3 robôs.



Fonte: Autor.

Figura 17 – Robô no campo em posição de jogo.



Fonte: Autor.

5.1 CONTROLADOR

Para desenvolver um controlador é necessário primeiramente compreender alguns aspectos da modelagem, e como os nós (nodes) são utilizados na programação. O principal nó na modelagem de um robô como este, é o Robot. Este nó, assim como os outros subsequentes que representam sensores e atuadores (*PositionSensor*, *DistanceSensor*, *RotationalMotor*, etc) são estruturas independentes, na linguagem C++ seriam structs. Como dito anteriormente o nó Robot, é o principal porque ele é visto como uma estrutura que contém outras estruturas, como os já citados sensores e atuadores. Darei um exemplo prático com um algoritmo teste, desenvolvido para rotacionar motores e buscar valores de posição. Também é de suma importância antes de começar um código que se tenha noção do passo da simulação e o passo de controle. O passo de simulação são: divisões em intervalos de tempo entre acontecimentos simulados,

ou seja, são intervalos entre um evento e outro dentro da simulação. Já o passo de controle é utilizado no loop principal, todo controlador deve possuir uma função de *step(TIME_STEP)*, essa função atualiza neste intervalo configurado como *TIME_STEP*, os valores das posições de motores, sensores entre outros, sincronizando-os ao passo de simulação.

Na primeira parte do código são incluídas bibliotecas referentes aos sensores/motores que se deseja trabalhar, todas estas exclusivas do Webots. Também deve ser configurado um *TIME_STEP* em milissegundos como uma constante. Na função principal é alocada memória para trabalhar com um tipo Robot, e atribuir o nome do robô criado, que por padrão é robot, mas pode ser alterado. O loop principal é composto por uma estrutura de repetição que iterativamente atualiza a simulação com o controlador. A maioria desse script é dado automaticamente quando se cria um controlador para o projeto.

Para testar a funcionalidade do modelo criado, foram desenvolvidos dois códigos para testes. Um deles utilizaremos de exemplo para demonstrar como utilizar todos os atuadores na programação de um robô, e o outro foi criado para demonstrar a funcionalidade do modelo, simulando um robô humanoide utilizando diversos motores para levantar-se de uma possível queda. Em toda a programação apresentada a seguir, foi utilizado a IDE do Webots e a linguagem C++. Para declarar sensores, é criado um ponteiro qualquer, do tipo do sensor que vai ser utilizado, no caso de um sensor de posição o tipo do ponteiro declarado será *PositionSensor*, de um sensor de distância o tipo é *DistanceSensor*, e assim por diante. Estes tipos fazem parte das bibliotecas incluídas para cada atuador no início do código. O nó principal, contém outras estruturas dentro dele, isso quer dizer que, através de uma estrutura Robot é possível acessar todos os objetos contidos no modelo. Portanto a linha de código que acessa o atuador nesse caso é: *getPositionSensor("Right_Arm_Shoulder_Position")*, perceba que o argumento passado na função *getPositionSensor()*, é o nome do sensor apontado por este ponteiro, cujo nome deve ser igual ao criado na modelagem. Após isso é necessário também habilitar o sensor para que sofra *updates* iterativos conforme o passo do controle prossegue, essa habilitação ocorre da seguinte maneira *enable(TIME_STEP)*. Para pegar o valor do sensor agora basta utilizar o comando *getValue()* dessa forma *sensor->getValue()*. Para motores não é muito diferente, também é criado um ponteiro que irá apontar para algum motor que foi previamente modelado, e o tipo atribuído ao ponteiro deve ser do tipo Motor. Atribui-se com a função *getMotor("Right_Arm_Shoulder")* seguindo basicamente os mesmos protocolos do sensor de posição, com excessão da necessidade de habilitar, porque o motor mantém seu valor até que seja escrito um novo comando para mudança de posição, ou seja no caso dos motores, percebe -se que eles não serão alterados

com o passo de simulação mas sim com o comando dado previamente pelo controlador. E este comando que altera a posição do motor é *setPosition()*. É passado como parâmetro o grau em radianos referente a posição do motor que se deseja, lembrando que na modelagem do dispositivo motor é configurado um intervalo mínimo e máximo de amplitude do motor. Tendo estes conhecimentos já é possível declarar sensores diversos, motores e enviar esses motores para as posições que quiser.

Figura 18 – Código Teste

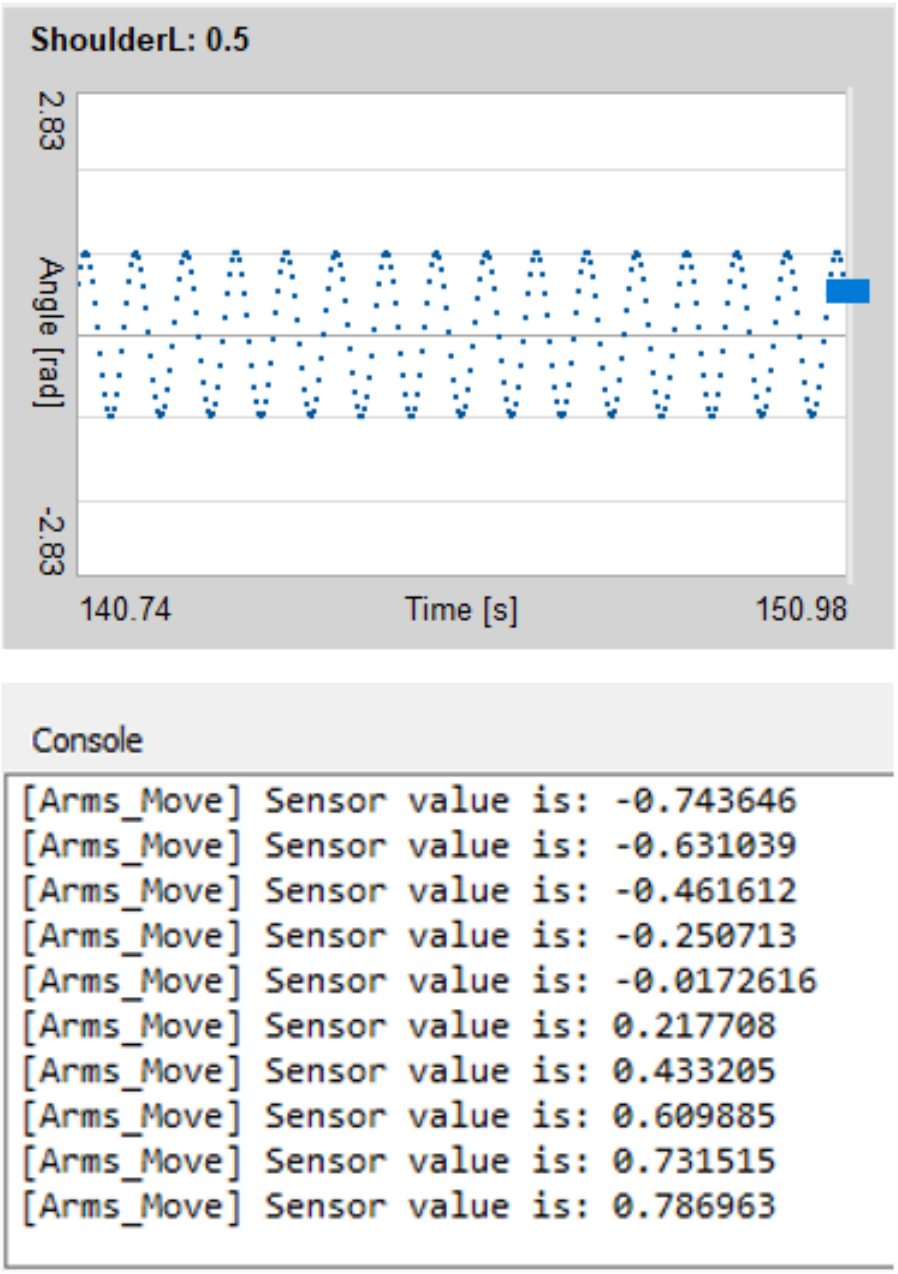
```
// File:          Test.cpp
// Date:15/05/2020
// Description:Arms Movements with a sine wave
// Author:Guilherme Cardoso Coelho
#include <webots/Robot.hpp>
#include <webots/Motor.hpp>
#include <webots/PositionSensor.hpp>
#include <iostream>
#include <cmath>
#define TIME_STEP 32

using namespace webots;  using namespace std;

int main(int argc, char **argv) {
  Robot *robot = new Robot();

  PositionSensor *sensor = robot->getPositionSensor("Right_Arm_Shoulder_Position");
  sensor->enable(TIME_STEP);

  Motor *motor1 = robot->getMotor("Right_Arm_Shoulder");
  Motor *motor2 = robot->getMotor("Left_Arm_Shoulder");

  const double F = 0.5; double t = 0.0;

  while (robot->step(TIME_STEP) != -1) {

    const double position = sin(t*2*M_PI*F);
    motor1->setPosition(position);
    motor2->setPosition(position);
    t += (double)TIME_STEP / 1000.0;

    const double value = sensor->getValue();

    cout << "Sensor value is: " << value << endl;
  };
  delete robot;
  return 0;
}
```

Fonte: Autor.

O código da Fig.18 basicamente movimenta os braços do robô para frente e para trás a partir de uma função senoide, como dito anteriormente a sua função é mais didática do que prática. Acima contém todas as declarações de motores e sensores previamente explicadas. E a

Fig.19 é o resultado do que acontece na simulação, Generic Window e Console. Nesta imagem contém o gráfico gerado com a variação senoidal da posição do motor do ombro, e no console são exibidos os valores do sensor de posição referente a este motor.

Figura 19 – Resultado do Código Teste



Fonte: Autor.

A seguir será apresentado um algoritmo (Fig.20, Fig.21 e Fig.22) responsável por simular um robô humanoide levantando. Na prática um robô humanoide possui alguns movimentos especiais que são configurados separadamente em um arquivo de Motion Editor, porém para simplificar esta aplicação, o algoritmo foi desenvolvido sem um Motion Editor. O código con-

siste em um sequência de eventos, que em cada etapa contribuem de alguma forma para que o robô seja capaz de levantar, o objetivo então é pegar o movimento completo e separá-lo em *frames*.

Figura 20 – Código de levantar Parte I

```
// File:          f_up.cpp
// Date:15/05/2020
// Description: Getting Up
// Author:Guilherme Cardoso Coelho
#include <webots/Robot.hpp>
#include <webots/Motor.hpp>
#include <iostream>

#define timeStep 320
using namespace webots;
using namespace std;

int main(int argc, char **argv) {

  Robot *robot = new Robot();
  Motor *ArmLowerL = robot->getMotor("ArmLowerL");
  Motor *ArmLowerR = robot->getMotor("ArmLowerR");
  Motor *LegLowerL = robot->getMotor("LegLowerL");
  Motor *LegLowerR = robot->getMotor("LegLowerR");
  Motor *LegUpperL = robot->getMotor("LegUpperL");
  Motor *LegUpperR = robot->getMotor("LegUpperR");
  Motor *AnkleL = robot->getMotor("AnkleL");
  Motor *AnkleR = robot->getMotor("AnkleR");
  Motor *ShoulderL = robot->getMotor("ShoulderL");
  Motor *ShoulderR = robot->getMotor("ShoulderR");

  double t = -1;

  while(robot->step(timeStep) != t){
    //G0
    ArmLowerL->setPosition(-2.04);
    ArmLowerR->setPosition(2.04);
    LegLowerL->setPosition(-1.13);
    LegLowerR->setPosition(-1.13);
    AnkleL->setPosition(1.42);
    AnkleR->setPosition(1.42);

   cout << robot->step(timeStep) << endl;
```

Fonte: Autor.

Figura 21 – Código de levantar Parte II

```
39    t = 0;
40   };
41
42   t = -1;
43
44   while(robot->step(timeStep) != t){
45     //G1
46     ShoulderL->setPosition(0.68);
47     ShoulderR->setPosition(0.68);
48      ArmLowerL->setPosition(0);
49     ArmLowerR->setPosition(0);
50     LegLowerL->setPosition(-1.13);
51     LegLowerR->setPosition(-1.13);
52     AnkleL->setPosition(1.42);
53     AnkleR->setPosition(1.42);
54    cout << robot->step(timeStep) << endl;
55    t = 0;
56   };
57     t = -1;
58
59 while(robot->step(timeStep) != t){
60     //G2
61     ShoulderL->setPosition(1.3);
62     ShoulderR->setPosition(1.3);
63     ArmLowerL->setPosition(0);
64     ArmLowerR->setPosition(0);
65     LegLowerL->setPosition(-1.53);
66     LegLowerR->setPosition(-1.53);
67     LegUpperL->setPosition(2.15);
68     LegUpperR->setPosition(2.15);
69     AnkleL->setPosition(0.51);
70     AnkleR->setPosition(0.51);
71    cout << robot->step(timeStep) << endl;
72    t = 0;
73   };
74
75       t = -1;
76
```

Fonte: Autor.

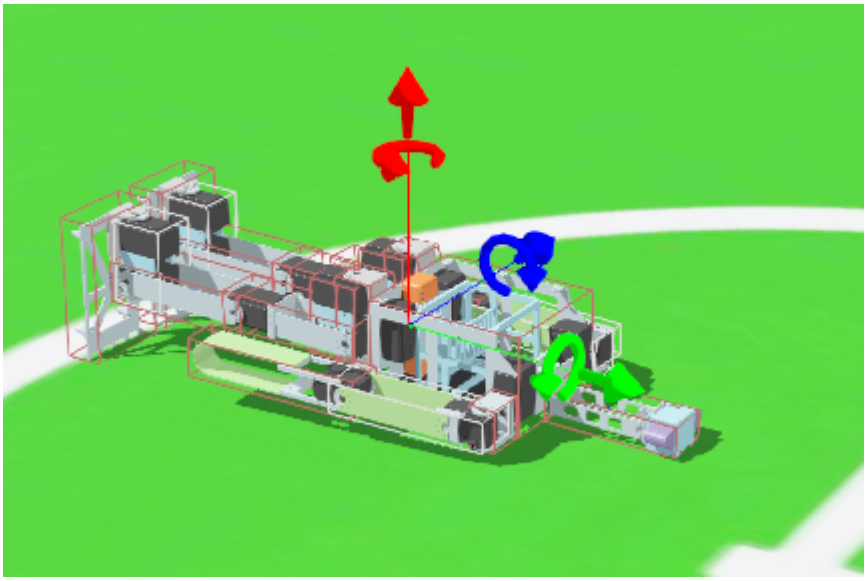
Figura 22 – Código de levantar Parte III

```
77 while(robot->step(timeStep) != t){
78     //G3
79     ShoulderL->setPosition(1.3);
80     ShoulderR->setPosition(1.3);
81     ArmLowerL->setPosition(0);
82     ArmLowerR->setPosition(0);
83     LegLowerL->setPosition(-1.53);
84     LegLowerR->setPosition(-1.53);
85     LegUpperL->setPosition(1.4);
86     LegUpperR->setPosition(1.4);
87     AnkleL->setPosition(0.51);
88     AnkleR->setPosition(0.51);
89    cout << robot->step(timeStep) << endl;
90    t = 0;
91   };
92
93   t = -1;
94
95 while(robot->step(timeStep) != t){
96     //G4
97     ShoulderL->setPosition(1.3);
98     ShoulderR->setPosition(1.3);
99     ArmLowerL->setPosition(0);
100    ArmLowerR->setPosition(0);
101    LegLowerL->setPosition(-1.2);
102    LegLowerR->setPosition(-1.2);
103    LegUpperL->setPosition(0.9);
104    LegUpperR->setPosition(0.9);
105    AnkleL->setPosition(0.51);
106    AnkleR->setPosition(0.51);
107   cout << robot->step(timeStep) << endl;
108   t++;
109  };
110  delete robot;
111  return 0;
112 }
```

Fonte: Autor.

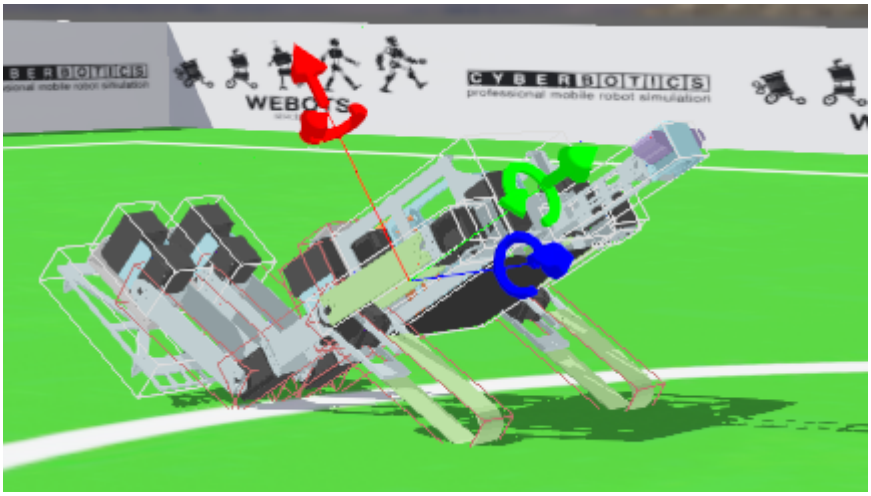
Cada *frame* leva uma sequência de motores para uma determinada posição, são separados por estruturas de repetição *while* e juntos concluem o movimento desejado. Cada etapa do movimento resultante do código acima, seguem nas Fig.23, Fig.24 Fig.25, Fig.26, Fig.27 e Fig.28. O video com os movimentos gerados no robô pelo algoritmo exibido pode ser visto em: https://youtu.be/7uw4iCAzH5Q .

Figura 23 – Posição Inicial



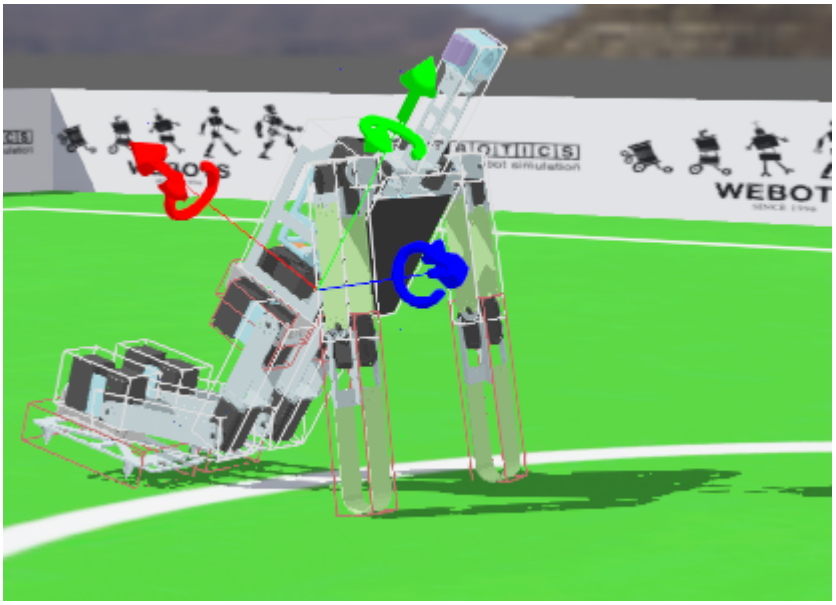
Fonte: Autor.

Figura 24 – G0



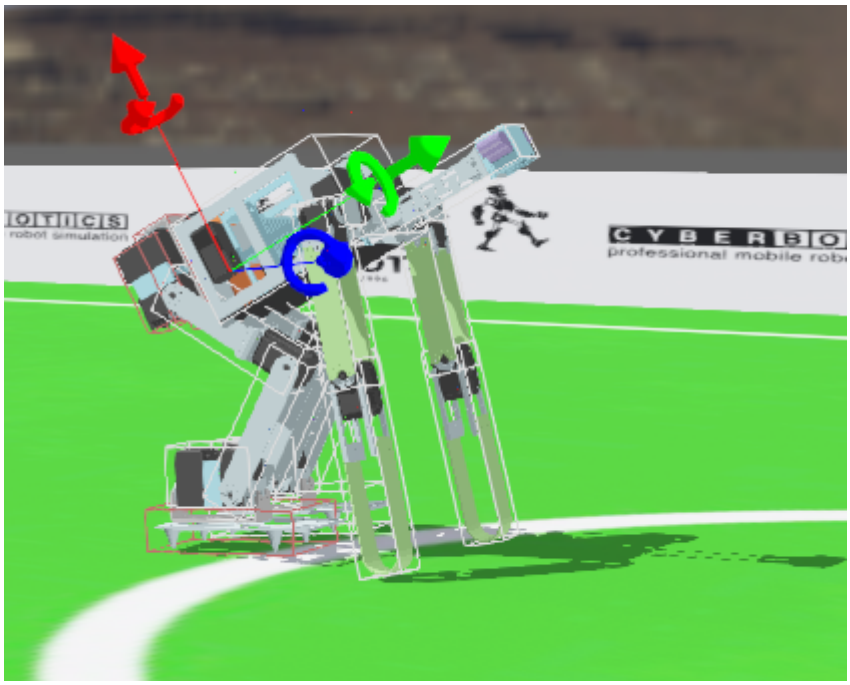
Fonte: Autor.

Figura 25 – G1



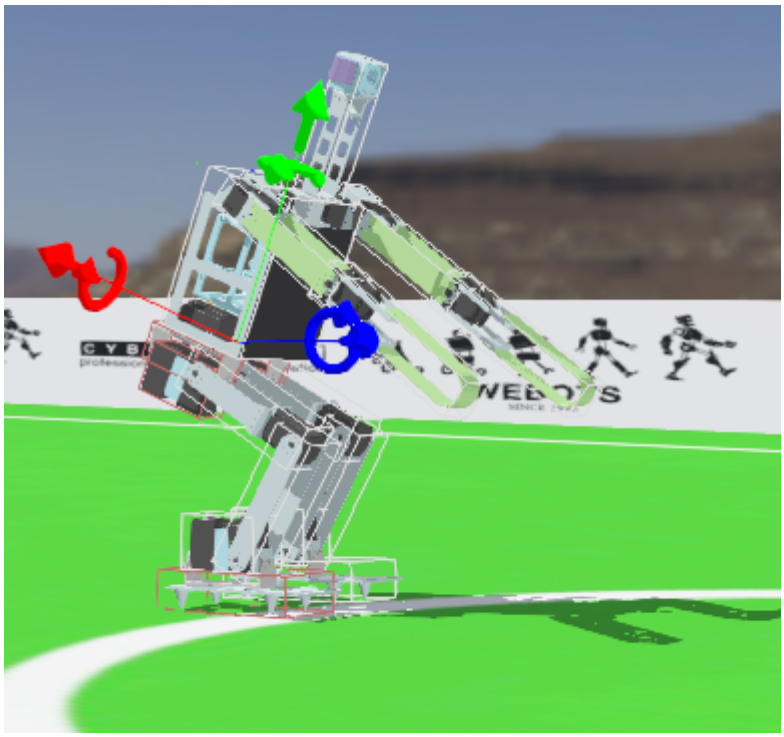
Fonte: Autor.

Figura 26 – G2



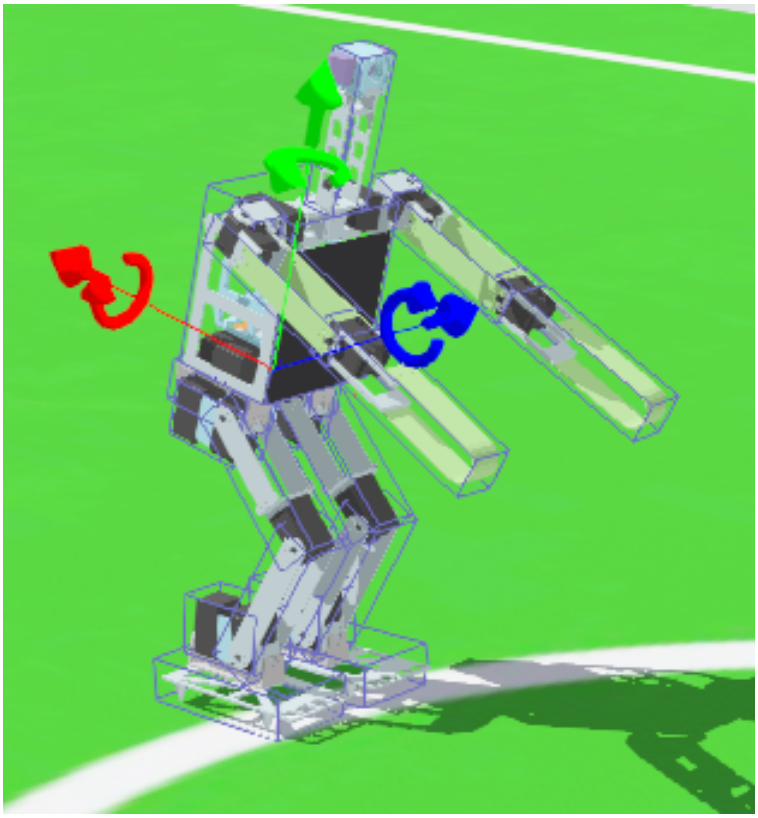
Fonte: Autor.

Figura 27 – G3



Fonte: Autor.

Figura 28 – G4



Fonte: Autor.

6 CONCLUSÕES

O projeto foi realizado com êxito, foram feitos testes para dimensionar o desempenho do robô em software e projetar um comparativo com a realidade. O robô foi modelado com todas as 19 articulações e com delimitações em todos os agrupamentos, bem como foram inseridos valores de massa e centro de massa. Acredito que com este trabalho será muito mais fácil projetar novos robôs, é muito importante possuir seus próprios modelos já que tudo é integralmente desenvolvido pela equipe de robótica.

A própria simulação pode ser utilizada para desenvolver habilidades na programação de algoritmos, é uma ferramenta que pode até mesmo ser utilizada em aulas de robótica. O que mais chamou atenção são as tecnologias de inteligência artificial e aprendizado de máquina que podem ser implementados em modelos próprios na simulação. Todas essas características apontam pontos positivos em relação a didática que esta ferramenta pode trazer.

O comportamento do modelo com os controladores teve um desempenho satisfatório, é perceptível as reações físicas de apoio das partes mecânicas do robô com o solo, com a bola, assim como forças de reação que aparecem quando os motores alteram a posição de forma rápida. Tudo indica que o modelo está pronto para utilização da equipe de robótica em testes e simulações. Para trabalhos futuros, poderiam ser realizadas otimizações na física e no ambiente de simulação para aumentar a veracidade do sistema, e customizá-la com detalhes baseado em situações específicas de jogo.

REFERÊNCIAS

BACCARIN, Bruno. **Modelagem 3D**. Disponível em: <http://fluxoconsultoria.poli.ufrj.br/blog/projetos-mecanicos/modelagem-3d-peca-como-funciona/>. Acesso em: 15.02.2019.

DAVID, Mansolino; OLIVIER, Michel; IJSPEERT, Auke Jan. Mobile Robot Modeling, Simulation and Programming. In: CYBERBOTICS. CONIELECOMP 2011, 21st International Conference on Electrical Communications and Computers. [S.l.: s.n.], 2013.

OLIVEIRA, Jonas Henrique Renolfi de et al. Object detection under constrained hardware scenarios: a comparative study of reduced convolutional network architectures. In: 2019 XVI Latin American Robotics Symposium and VII Brazilian Robotics Symposium (LARS/SBR). [S.l.]: IEEE, 2019.

PERICO, D. H. **Robôs Humanoides - Da Utopia à Realidade. Mundo Robótica.** [S.l.: s.n.], 2014. Disponível em: <http://www.obr.org.br/>. Acesso em: 01.05.2014.

ROBOCUP. **A Brief History of RoboCup.** [S.l.: s.n.], 2019. Disponível em: <https://www.robocup.org/a_brief_history_of_robocup>. Acesso em: 15.10.2019.

SILVA, Isaac J et al. Using reinforcement learning to optimize gait generation parameters of a humanoid robot. **XIII Simpósio Brasileiro de Automação Inteligente**, 2017.

SMITH, R. **Dynamics Simulation**. Disponível em: <http://ode.org/slides/parc/dynamics.pdf>. Acesso em: 19.02.2019. 2004.

WEBOTS. **Cyberbotics Controller**. 2019. Disponível em: <https://cyberbotics.com/doc/guide/controller-programming>.

_____. **Modification of the Enviroment**. 2019. Disponível em: <https://cyberbotics.com/doc/guide/tutorial-2-modification-of-the-environment>.

_____. **Motion Editor**. 2019. Disponível em: <https://cyberbotics.com/doc/guide/robotis-op2#motion-manager>.

_____. **Portifolio Webots**. 2019. Disponível em: <https://cyberbotics.com/#portfolio>.

_____. **Simulation Software with 3D Modeling.** [S.l.: s.n.], 2019. Disponível em: <https://www.intorobotics.com/roboticssimulation-softwares-with-3d-modeling-and-programming-support/>. Acesso em: 15.02.2019.

WOLF, Denis Fernando et al. Robótica móvel inteligente: Da simulação às aplicações no mundo real. In: SN. MINI-CURSO: Jornada de Atualização em Informática (JAI), Congresso da SBC. [S.l.: s.n.], 2009. p. 13.

ZANNATHA, JM Ibarra et al. Behavior control for a humanoid soccer player using Webots. In: IEEE. CONIELECOMP 2011, 21st International Conference on Electrical Communications and Computers. [S.l.: s.n.], 2011. p. 164–170.