UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

HERON DE SOUZA MARQUES
PRISCILA PEREIRA DE CAMARGO

SIMULADOR DE BRAÇO MECÂNICO PARA ROBÔS EM TEMPO REAL

NITERÓI
2010

HERON DE SOUZA MARQUES
PRISCILA PEREIRA DE CAMARGO

SIMULADOR DE BRAÇO MECÂNICO PARA ROBÔS EM TEMPO REAL

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação. Área de concentraçao: Computação Gráfica.

Orientador: Prof. Dr. ESTEBAN WALTER GONZALEZ CLUA

Niterói
2010

HERON DE SOUZA MARQUES
PRISCILA PEREIRA DE CAMARGO

SIMULADOR DE BRAÇO MECÂNICO PARA ROBÔS EM TEMPO REAL

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação. Área de concentraçao: Computação Gráfica.

Aprovada em julho de 2010.

BANCA EXAMINADORA

Prof. Dr. ESTEBAN WALTER GONZALEZ CLUA - Orientador
UFF

Prof. Dr. ANSELMO ANTUNES MONTENEGRO
UFF

M.Sc. ROMULO CURTY CERQUEIRA
UFF

Niterói
2010

Agradecimentos

Ao único e eterno Deus, que nos concedeu saúde e inteligência para que concluíssemos este trabalho.

Aos nossos amados familiares e amigos, que nos apoiaram incondicionalmente em todas as etapas da vida.

Aos nossos colegas de universidade Carlos André de Souza Augusto, monitor do curso de Mecânica Geral do Instituto de Física, Giancarlo Vasconcelos Taveira do Nascimento, do curso de Ciência da Computação, e a todos da equipe do Medialab pelo apoio.

Ao engenheiro de equipamentos sênior do CENTRO DE PESQUISA E DESENVOLVIMENTO LEOPOLDO A. MIGUEZ DE MELLO (CENPES/PDP/TS) da Petrobras, Ney Robinson Salvi dos Reis, pela liberação para este estudo dos documentos técnicos do braço robótico e do modelo gráfico.

Aos nossos colaboradores do TecGraf/PUC-Rio, pelo desenvolvimento do modelo gráfico utilizado nesse trabalho, liberado pela Petrobras.

Ao mestre e participante desse projeto Romulo Curty Cerqueira, pelas orientações técnicas.

Ao nosso orientador Professor Doutor Esteban Clua, pelo apoio e confiança concedidos.

E a todos aqueles não mencionados nesse espaço que tenham nos ajudado direta ou indiretamente.

Resumo

Com a expansão do setor petrolífero, surge a necessidade de fazer uso de novas ferramentas tecnológicas que auxiliem no processo de exploração de petróleo. Algumas empresas petrolíferas investem em robôs para levarem a cabo tarefas inapropriadas ou impossíveis ao ser humano, como prospecção em grandes profundidades, desobstrução de dutos e exploração de ambientes inóspitos. Alguns desses robôs são tripulados e outros são operados remotamente. De qualquer forma, é preciso fornecer treinamento ao profissional que os irá controlar, para que o mau uso da máquina não cause prejuízos à sociedade e ao meio ambiente. Este trabalho tem como objetivo simular virtualmente um braço robótico destes robôs. Para modelar tal simulação, será utilizado o game engine Unity, que é uma plataforma 3D focada em animações interativas.

Palavras-chave: Braço robótico. Unity. Física. Realidade Virtual. Games.

Abstract

With the expansion of the oil sector, it is necessary to make use of new technological tools to help the process of oil exploration. Some oil companies invests in robots to carry out inappropriate or impossible tasks to humans, as deep mining, clearing of pipelines and exploration of harsh environments. Some of these robots are manned and others are operated remotely. Anyway, it is necessary to provide professional training to the person that will control it, so that the misuse of the machine does not cause damage to society and to the environment. This work aims to simulate virtually a robotic arm of these robots. To model such simulation, will be used the Unity game engine, which is a plattaform focused on 3D interactive animations.

Keywords: Robotic arm. Unity. Physics. Virtual Reality. Games.

Lista de Figuras

1.1 Área de influência do gasoduto [10] 1
1.2 Primeiro protótipo do RAH [11] 2
1.3 Segundo protótipo do RAH [11] 2
1.4 Terceiro protótipo do RAH [10] 2
1.5 Coletor de larvas [10] 3

2.1 SMTPG 6
2.2 Graus de liberdade do SMTPG 7
2.3 Junta de Rotação [2] 8
2.4 Junta prismática [2] 8
2.5 Exemplo de um espaço de trabalho [2] 9
2.6 Espaço de trabalho do SMTPG 9
2.7 Tipos de espaços de trabalho [2] 10
2.8 Robô articulado verticalmente [2] 11

3.1 Exemplos de scripts no Unity 13
3.2 Imagem extraída da cena de demonstração Islands do Unity [12] 14

4.1 Exemplo de torque [9] 16
4.2 Exemplo de overlaping testing [9] 18

5.1 Hierarquia 20
5.2 Partes do cilindro hidráulico 20
5.3 Propriedade Spring do HingeJoint 22
5.4 Eixo de rotação da garra 22
5.5 ConfigurableJoint 23
5.6 Configuração em que os torques são máximos 26

5.7 Atuação das forças na mão 26
5.8 Atuação das forças no antebraço 27
5.9 Atuação das forças no braço 27
5.10 Function OnGUI [12] 29
5.11 GUI da simulação 30

6.1 Janela Statistics da simulação 31
6.2 Ligação entre as partes do braço robótico 33
6.3 Problema da sobreposição de meshes - antes do início da execução 33
6.4 Problema da sobreposição de meshes - depois do início da execução 34

Sumário

Agradecimentos **iv**

Resumo **v**

Abstract **vi**

Lista de Figuras **viii**

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivos 3

1.3 Visão geral 3

2 Apresentação do problema **5**

2.1 Definições 5

2.2 Graus de liberdade 6

2.3 Tipos de juntas 7

2.3.1 Junta de rotação (Rotacional) 8

2.3.2 Junta prismática (Linear) 8

2.4 Espaço de trabalho 8

2.5 Envelope de trabalho 9

2.6 Classificação de estruturas mecânicas 10

2.7 Rigidez mecânica e carga máxima 11

3 Unity **12**

4 Simulaçao física em tempo real **15**

4.1 Corpos rígidos 15

4.2 Torque 15
4.3 Juntas 16
4.4 Colisão 17
4.5 Materiais físicos 18

5 Solução proposta **19**

6 Resultados obtidos **31**

7 Conclusão e Sugestões de trabalhos futuros **35**

Referências Bibliográficas **36**

A Apêndice **38**
A.1 Script Garra.js 38
A.2 Script Mao.js 40
A.3 Script Suspensor.js 42
A.4 Script Suspensor1.js 43
A.5 Script Suspensor2.js 43
A.6 Script Suspensor3.js 44
A.7 Script Suspensor4.js 44
A.8 Script Antebraco.js 45
A.9 Script Braco.js 47
A.10 Script BracoRobotico.js 50
A.11 Script ControleBraco.js 53

Capítulo 1

Introdução

1.1 Motivação

Em 2009, a Petrobrás concluiu a construção de um gasoduto de 400 quilômetros de extensão interligando as cidades de Coari e Manaus no estado do Amazonas, no Norte do Brasil. O gasoduto cruza, além das cidades de Coari e Manaus, os municípios de Codajas, Anori, Anamã, Caapiranga,Manacapuru e Iranduba, conforme a figura 1.1.

Figura 1.1: Área de influência do gasoduto [10]

As características hidrológicas e climáticas dominantes na região determinam diferentes cenários e complexidades peculiares ao longo do trajeto. Para monitorar essa região de difícil acesso e evitar desastres ambientais, a Petrobrás, através do projeto Cognitus, braço tecnológico do projeto Piatam (Potenciais Impactos e Riscos Ambientais da Indústria de Óleo e Gás na Amazônia), está desenvolvendo uma série de robôs, apelidados de AmazonBots.

Esses robôs têm o objetivo de detectar quaisquer anomalias ambientais, especialmente aquelas provocadas por vazamento de gás ou óleo, além de coletar dados sobre o meio ambiente e disponibilizá-los em tempo real via satélite para uma base de dados em Coari, de onde serão,

da mesma forma, enviados ao Rio de Janeiro.

O Robô Ambiental Híbrido (RAH) tem a capacidade de se deslocar tanto na terra quanto na água e é utilizado para dar acesso a áreas alagadas da floresta nas quais o homem não pode chegar por meios normais de transporte. Dentre os modelos já construídos, estão os robôs móveis que podem ser operados via satélite. As figuras 1.2 e 1.3 mostram o primeiro e o segundo protótipos, respectivamente, do RAH desenvolvido pelo Laboratório de Robótica do Centro de Pesquisas e Desenvolvimento Leopoldo Américo M. de Melo (CENPES) da Petrobrás.

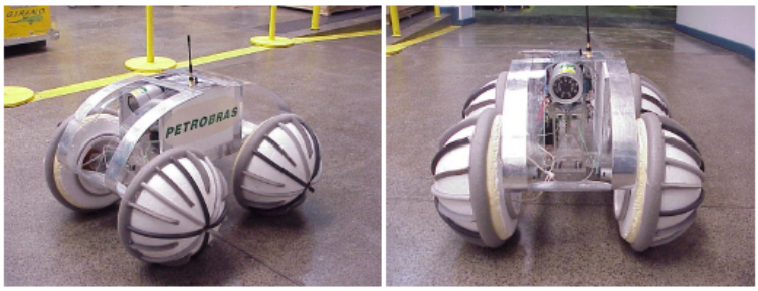

Figura 1.2: Primeiro protótipo do RAH [11]

Figura 1.3: Segundo protótipo do RAH [11]

Além dos protótipos citados acima, existe ainda um terceiro modelo (figura 1.4), este tripulado, cujo projeto está em andamento. Ele servirá para casos em que a presença de um profissional no campo de trabalho seja indispensável, ou ainda para remover alguma pessoa de local de difícil acesso.

Figura 1.4: Terceiro protótipo do RAH [10]

Das missões até agora destacadas para o robô ambiental híbrido, o monitoramento ambiental da Amazônia, principalmente na região do gasoduto Coari-Manaus é, sem dúvida, uma das mais necessárias. Para possibilitar ações de contingência ou mitigação, é necessário que sejam continuadas as atividades de medição e coleta de amostras. Através da análise de mudanças nos dados coletados, é possível determinar se a construção e a utilização do duto estaria causando modificações no meio ambiente. A coleta é realizada por um coletor que está anexado ao manipulador, conforme podemos observar na figura 1.5

Figura 1.5: Coletor de larvas [10]

1.2 Objetivos

Como descrito anteriormente, já está em andamento um terceiro protótipo que será um modelo tripulado. Ele será fundamental nas situações em que a presença de um profissional em campo seja indispensável.

Uma das principais funções do robô híbrido ambiental é coletar dados sobre o meio ambiente em questão a fim de identificar possíveis impactos nessa região, portanto é necessário treinar de forma adequada o profissional que estará na cabine do robô pilotando-o.

Este projeto tem como objetivo desenvolver um simulador virtual de braço mecânico em tempo real fazendo uso de elementos físicos para que o produto final seja o mais realístico possível. Para isso, serão utilizados as especificações disponíveis de um braço robótico e o game engine Unity, que é uma ferramenta integrada para criação conteúdos interativos em 3D.

1.3 Visão geral

Este trabalho foi dividido da seguinte forma. No capítulo 2 são detalhados alguns conceitos relacionados à robótica. O capítulo 3 apresenta a arquitetura da ferramenta Unity. O capítulo 4

descreve as simulações físicas em tempo real, como dinâmica dos corpos rígidos e colisão. No capítulo 5 é mostrada em detalhes a implementação da simulação. O capítulo 6 apresenta os resultados obtidos. No capítulo 7, tiram-se conclusões sobre os resultados obtidos. E, finalmente, o capítulo 8 sugere a expansão do trabalho produzido.

Capítulo 2

Apresentação do problema

2.1 Definições

O objeto de estudo desta monografia pode ser descrito como um sistema manipulador teleoperado de propósito geral, projetado para operações nos ambientes inóspitos do fundo do mar e outros ambientes hostis. Por motivos de confidencialidade, não é permitido informar nesse trabalho dados específicos sobre o robô que permitam sua identificação, portanto será usado para referências futuras nesse texto o acrônimo SMTPG, que significa Sistema Manipulador Teleoperado de Propósito Geral (figura 2.1).

O SMTPG, que poderá ser usado pelo projeto Cognitus da Petrobrás, é definido como um robô manipulador. Trata-se de um braço mecânico que move uma ferramenta para a posição desejada no espaço. Este, assim como qualquer outro robô manipulador, consiste de juntas que interligam uma série de corpos rígidos e permitem um movimento relativo entre eles. Esse movimento é controlado por pistões, de modo a posicionar a extremidade livre da cadeia (órgão terminal, efetuador, garra ou ferramenta) em relação à outra extremidade, que é fixa (base do manipulador). Assemelha-se a um braço humano, tanto na forma quanto nas funções e nas possibilidades de movimento. Essa similaridade é importante para o robô ambiental híbrido, citado no capítulo introdutório, visto que um de seus objetivos é a coleta de amostras do ambiente através da utilização de um braço robótico integrado.

Todo robô manipulador tem em algum ponto da sua estrutura física um dispositivo chamado de efetuador. Este dispositivo tem como função operar sobre o objeto a ser manipulado, e pode ser uma ferramenta, como uma ponta de solda, por exemplo, destinada a soldar uma superfície. Pode ser algum dispositivo especial, como uma câmera de vídeo, mas, em

geral, trata-se de algum tipo de garra capaz de segurar uma peça com o intuito de deslocá-la pelo espaço de trabalho do robô. Em particular, os braços mecânicos, como é caso do SMTPG, costumam ter uma garra como efetuador, e a maioria dos braços industriais permitam trocar esse dispositivo efetuador com facilidade, adequando o robô para diferentes tarefas que exigem diferentes tipos de efetuadores.

O SMTPG pode ser caracterizado por conceitos fundamentais da robótica, como graus de liberdade, classificação de estruturas mecânicas, rigidez, carga máxima, espaço e envelope de trabalho e juntas. Nas seções a seguir, são apresentados esses conceitos e, em seguida, relacionados ao braço robótico da simulação.

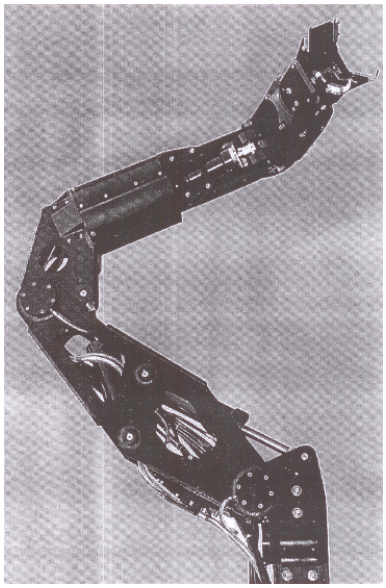

Figura 2.1: SMTPG

2.2 Graus de liberdade

O número total de juntas (articulações) do manipulador é conhecido com o nome de graus de liberdade (ou DOF, segundo as iniciais em inglês). Define-se como grau de liberdade o número de parâmetros (ou variáveis independentes) que terão que ser especificados a fim de definir qualquer configuração (translação e rotação) do mecanismo robótico, ou, mais especificamente, do efetuador. Um manipulador típico tem 6 graus de liberdade, sendo três para posicionamento do efetuador dentro do volume de trabalho, e três para obter uma orientação do efetuador adequada para segurar o objeto. Com menos de 6 graus de liberdade o manipulador poderia não atingir uma posição arbitrária com uma orientação arbitrária dentro do volume de trabalho. Para certas

aplicações por exemplo, manipular objetos num espaço que não se encontra livre de obstáculos, poderiam ser necessários mais de 6 graus de liberdade. A dificuldade de controlar o movimento aumenta com o número de elos do braço.

O SMTPG possui 6 graus de liberdade:

Shoulder Slew: 100 degrees
Shoulder Pitch: 90 degrees
Elbow Pitch: 130 degrees
Lower Arm Rotate: +-180 degrees
Wrist Yaw: 100 degrees
Wrist Rotate: Continuous

A figura 2.2 ajuda a identificar os graus de liberdade descritos acima.

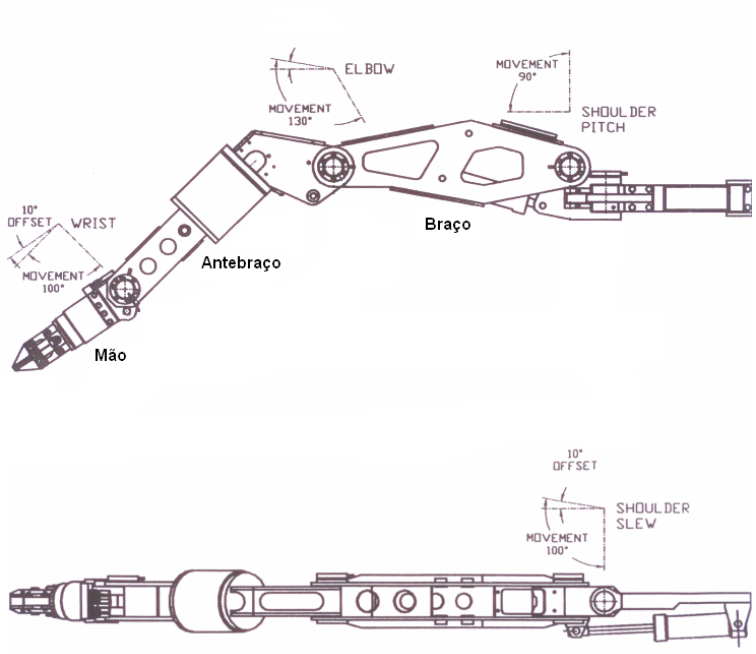

Figura 2.2: Graus de liberdade do SMTPG

2.3 Tipos de juntas

Define-se junta como a interligação entre duas hastes (considere-se haste cada objeto no segundo nível da hierarquia mostrada na figura 5.1). Ela permite o movimento relativo entre os

mesmos numa única dimensão ou grau de liberdade. Existem dois tipos básicos de juntas em robôs:

2.3.1 Junta de rotação (Rotacional)

Quando o movimento entre duas hastes adjacentes é de rotação, como mostrado na figura 2.3. Permite a mudança de orientação relativa entre duas hastes.

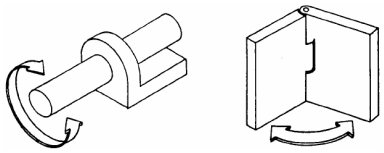

Figura 2.3: Junta de Rotação [2]

2.3.2 Junta prismática (Linear)

Quando o movimento entre duas hastes adjacentes é linear, como mostrado na figura 2.4. Não permite a mudança de orientação relativa entre duas hastes.

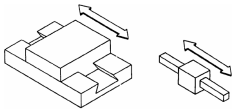

Figura 2.4: Junta prismática [2]

2.4 Espaço de trabalho

O espaço (ou volume) de trabalho do manipulador é o termo que se refere ao espaço dentro do qual este pode movimentar o efetuador; é definido como o volume total conformado pelo percurso do extremo do último elo, o punho, quando o manipulador efetua todas as trajetórias possíveis. Em geral, não é considerada a presença do efetuador para definir este volume de trabalho, pois se fosse assim este volume ficaria determinado pelo seu tamanho, o qual depende do dispositivo terminal utilizado. Observe que este volume dependerá da anatomia do robô, do tamanho das hastes, assim também como dos limites dos movimentos das juntas. A figura 2.5 mostra um espaço de trabalho genérico, e a figura 2.6 mostra o espaço de trabalho do SMTPG.

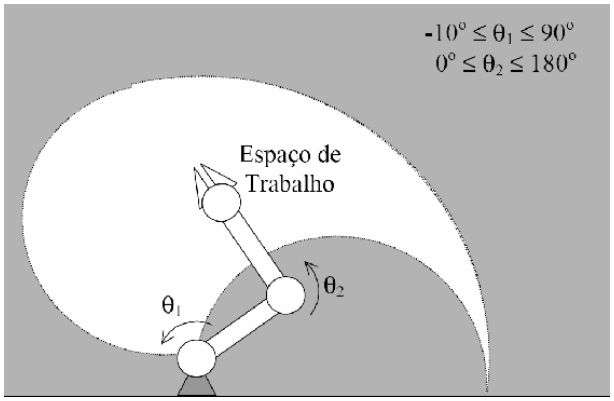

Figura 2.5: Exemplo de um espaço de trabalho [2]

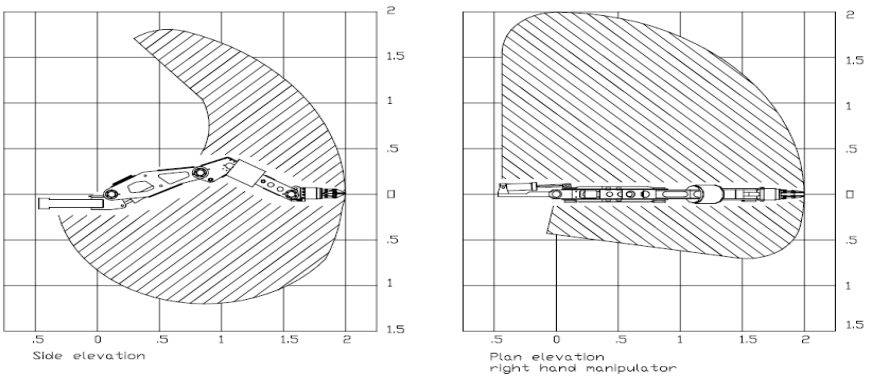

Figura 2.6: Espaço de trabalho do SMTPG

Comparando-se os volumes de trabalho dos vários tipos de robôs (veja a seção 2.6), percebe-se que quanto mais juntas de rotação tiver o robô, maior será seu volume de trabalho. O robô com maior volume de trabalho é o robô articulado verticalmente.

2.5 Envelope de trabalho

O envelope de trabalho é a forma geométrica que envolve o volume de trabalho. Qualquer objeto ou operação a ser alcançado ou realizada pelo robô deve estar localizado dentro do envelope de trabalho do robô, caso contrário o robô não irá ser capaz de alcançar este objeto ou realizar esta operação.

2.6 Classificação de estruturas mecânicas

A estrutura cinemática de um robô influencia a definição de suas características, dentre elas, a precisão, a complexidade de controle e o espaço de trabalho. Existem diferentes configurações físicas, ou diferentes anatomias nos robôs manipuladores. Essas configurações estão determinadas pelos movimentos relativos das três primeiras juntas, as destinadas ao posicionamento do efetuador. Estas juntas podem ser prismáticas, de rotação, ou combinação de ambas. Para cada combinação possível existirá uma configuração física ou anatomia diferente. Observe que a configuração física independe do tamanho dos elos, pois estes determinarão em todo caso o tamanho do espaço de trabalho, mas não sua forma. As configurações físicas estão caracterizadas pelas coordenadas de movimento das três primeiras juntas, ou pelas três primeiras coordenadas generalizadas, que são as variáveis que representam o movimento destas juntas. Os robôs pertencem a uma das 5 categorias a seguir: cartesiano, cilíndrico, esférico, articulado verticalmente e articulado horizontalmente. A figura 2.7 representa o espaço de trabalho definido por todos esses tipos, respectivamente. O SMTPG é articulado verticalmente (Figura 2.8).

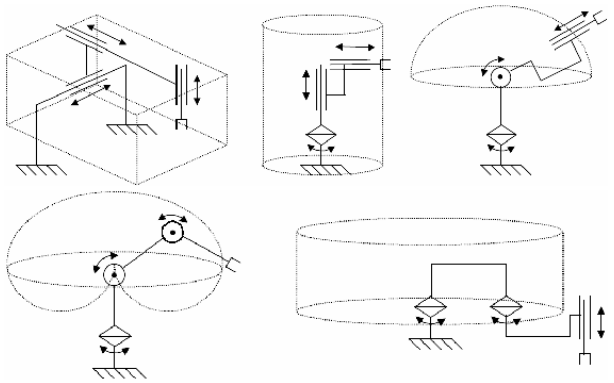

Figura 2.7: Tipos de espaços de trabalho [2]

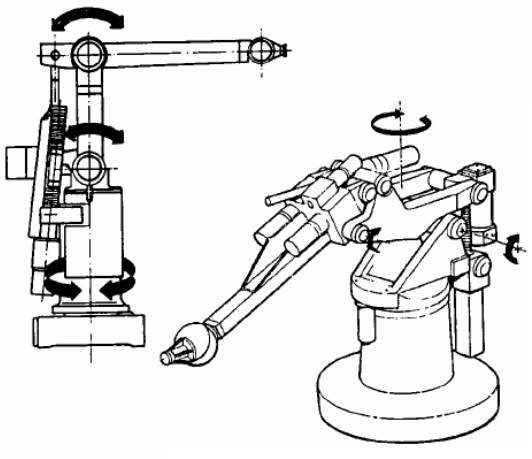

Figura 2.8: Robô articulado verticalmente [2]

2.7 Rigidez mecânica e carga máxima

A rigidez mecânica de uma estrutura é a medida de como uma estrutura distorce pela ação de cargas atuando sobre ela. Quanto mais rígida for uma estrutura, menos ela irá ser distorcida. Naturalmente, uma alta rigidez mecânica é desejável em robôs, pois a posição do braço irá ser menos sensível às cargas impostas sobre ele.

Uma estrutura mecânica (como as de um robô manipulador) apresenta uma limitação quanto ao peso que podem transportar. Essa limitação é a carga máxima. O SMTPG possui capacidade de trasportar no máximo 210 kg, segundo suas especificações técnicas.

Capítulo 3

Unity

O Unity é um game engine. Costuma-se traduzir esse termo como motor de jogo. Trata-se de um programa com um conjunto de bibliotecas que simplificam e abstraem o desenvolvimento de jogos ou outras aplicações gráficas em tempo real, tais como demonstrações, visualizações arquiteturais e simulações de treinamento (pilotagem de aeronaves, manuseio de ferramentas cirúrgicas, etc).

Tipicamente, um motor de jogo inclui um motor gráfico para renderizar gráficos 2D e/ou 3D, um motor de física para simular a física ou simplesmente para fazer detecção de colisão, suporte a grafos de cena e entidades e suporte a uma linguagem de script. Esses recursos são oferecidos pelo Unity e são usados pela simulação realizada nesta monografia.

Os motores gráficos são os responsáveis por processar dados abstraídos de alto nível (os objetos de jogo, ou Game Objects) e gerar dados de baixo nível entendíveis pelo hardware (placa gráfica). Já os motores de física são responsáveis por simular ações físicas reais, através de variáveis como gravidade, massa, velocidade, resistência do ar, atrito, força e elasticidade.

Para efetuar a simulação física de corpos rígidos e o tratamento de colisões o Unity utiliza o motor de física PhysX da NVidia. Considerado um dos mais completos do mercado, inclui a possibilidade de ser executado em GPU, o que pode trazer à simulação um maior desempenho. Com essa integração, tem-se acesso simplificado a um sistema de simulação física sofisticado. Várias das funcionalidades oferecidas pela PhysX são manipuladas graficamente através da interface do Unity ou através de scripts, permitindo que simulações físicas complexas sejam desenvolvidas.

O Unity também possui suporte a grafo de cenas. Este consiste em uma coleção de nós em um grafo ou em uma árvore. Um nó pode ter vários filhos, mas terá apenas um pai e

qualquer efeito produzido no pai será propagado para todos os seus filhos. Desta forma, quando uma característica é aplicada a um determinado grupo, todos os seus membros estarão sob o efeito da mesma característica. Isso possibilita manipular um grupo de objetos tão facilmente quanto um objeto simples.

Além disso, o Unity fornece abstração de hardware, permitindo que um programador desenvolva aplicativos sem a necessidade de conhecer a arquitetura da plataforma-alvo. Por esse motivo, ele foi desenvolvido utilizando-se de API's existentes e bem difundidas, como OpenGL e DirectX. Dessa forma, torna-se posssível desenvolver jogos para diferentes plataformas, como Windows, MAC, Web, iPhone e Wii.

O sistema de scripting do Unity é abrangente e flexível. Internamente, os scripts são executados através de uma versão modificada da biblioteca Mono, uma implementação de código aberto para o sistema DotNet. Através desta biblioteca, o Unity permite que os scripts sejam implementados em qualquer uma das seguintes linguagens: Javascript, C# ou Boo (um dialeto de Python). Não existe penalidade por se escolher uma linguagem ou outra, sendo inclusive possível usar mais de uma delas em uma mesma aplicação. Além das linguagens citadas, a Mono também possibilita trabalhar com PHP, VB.Net, Lua entre outras. Para este trabalho será utilizado o Javascript. De forma consistente à arquitetura desenvolvida, scripts no Unity são acoplados como componentes de game objects.

```
// javaScript
function myfunc() : String {

    return;
}
```

```
// C#
string myfunc(){
    return;
}
```

```
# Boo (explicit return type)
def myfunc() as string:
    return "hello world"
```

Figura 3.1: Exemplos de scripts no Unity

No Unity, é simples importar modelos 3D, pois ele aceita diversos formatos e dá suporte a texturas e animações. Ele lê arquivos FBX, dae, 3DS, dxf e obj gerados pelos softwares de desing gráfico Maya, Cinema 4D, 3ds Max, Cheetah3D, Modo, Lightwave e Blender. Para alguns, como acontece com Maya, Cinema 4D, Cheetah3D e Blender, o Unity importa nativamente os formatos-padrão gerados por tais softwares. Para os demais, é preciso fazer a exportação manualmente, diretamente do software ou através de algum programa conversor de formatos.

O Unity apresenta uma facilidade ímpar para as atividades de teste e depuração. Todos os parâmetros de qualquer objeto, modelo ou até mesmo script podem ser alterados em tempo

de execução no próprio editor, além de existir a possibilidade de pausar a execução para verificar parâmetros a qualquer momento e executar a simulação frame a frame. Essa característica traz ao desenvolvedor uma grande produtividade, pois evita que ele precise, por exemplo, parar a simulação, editar um parâmetro no script, recompilar o código e rodar a simulação novamente. A compilação dos scripts é igualmente prática, pois é realizada toda vez que ele é salvo, e novos efeitos incluídos no decorrer da execução de uma simulação podem ser prontamente observados.

Figura 3.2: Imagem extraída da cena de demonstração Islands do Unity [12]

Capítulo 4

Simulaçao física em tempo real

Visto que a proposta deste trabalho é simular um braço robótico de maneira mais precisa e realista possível, serão abordados neste capítulo alguns conceitos físicos, bem como as ferramentas do Unity utilizadas para implementar tais conceitos em uma simulação.

4.1 Corpos rígidos

O corpo rígido é um sistema constituído de partículas agregadas de tal modo que a distância entre as várias partes que constituem o corpo não varia com o tempo. Ele pode realizar movimento de translação que pode ser analisado observando-se exclusivamente o centro de massa do corpo. O corpo executa movimento de translação se o seu centro de massa se desloca à medida que o tempo passa. Outro movimento do corpo rígido é o movimento de rotação que se observa sempre que um torque (conceito que será abordado na seção 4.2) é aplicado ao corpo rígido.

No Unity, existe um componente chamado Rigidbody, o qual permite que o objeto possa ser controlado através do motor físico. Ao se adicionar este componente a um objeto, este passa a ser influenciado pela força da gravidade (quando ativada) e a reagir a colisões de forma mais precisa. Com o Rigidbody é possível manipular parâmetros como resistência do ar, massa, centro de massa, gravidade, e outros.

4.2 Torque

O movimento de rotação é caracterizado quando cada ponto do corpo se move sobre um círculo cujo centro fica no eixo de rotação, e cada ponto tem o mesmo deslocamento angular durante

um intervalo de tempo.

O torque, também chamado de momento ou momento estático, é definido a partir da componente perpendicular ao eixo de rotação da força aplicada sobre um corpo que é efetivamente utilizada para fazer ele girar em torno de um eixo ou ponto central. A distância do eixo ao ponto onde atua uma força $\vec{F}$ é denotada por $\vec{r}$ que também é um vetor. Sendo assim, o torque é definido pela relação: $\vec{T} = \vec{r}$ x $\vec{F}$

A figura 4.1 mostra a relação entre força e torque, indicando que a força F causou uma rotação no sentido anti-horário.

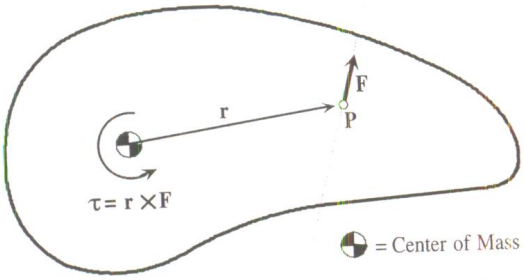

Figura 4.1: Exemplo de torque [9]

O torque será diferente de zero quando a força for aplicada em qualquer ponto do corpo desde que não seja aplicada no eixo de rotação.

Para simular o torque no Unity, serão modelados cilindros hidráulicos (pistões), os quais são próprios do SMTPG. Ao todo serão utilizados 4 pistões e cada um deles será responsável por aplicar uma força capaz de girar o braço, o antebraço, a mão e todo o braço robótico em torno de suas respectivas juntas.

4.3 Juntas

As juntas têm a função de interligar dois corpos rígidos permitindo que sejam realizados movimentos em uma ou mais dimensões (graus de liberdade).

Para representar uma junta no Unity, utiliza-se um componente chamado Joint. Existem vários tipos, como o HingeJoint, o FixedJoint e o ConfigurableJoint.

O HingeJoint permite conectar dois Rigidbodies, limitando-os a moverem-se como se estivessem conectados por uma dobradiça. Representa um tipo de junta de revolução. É útil para modelar portas, correntes, pêndulos, etc. Uma vez aplicada a um GameObject (GO), a

dobradiça irá rotacionar em um ponto especificado pelo usuário, ao redor de um determinado eixo.

O FixedJoint é o Joint mais simples do Unity. Ele torna o movimento de um Rigidbody dependente do movimento de outro. Seu efeito é equivalente ao de filiação de objetos, porém é realizado pela física, e não pela hierarquização de Transforms. O melhor cenário para sua utilização é quando se necessita conectar objetos sem torná-los hierarquicamente dependentes, o que possibilita simular a ruptura física de estruturas.

ConfigurableJoints são extremamente customizáveis. Eles expõem todas as propriedades da PhysX relacionadas às juntas, portanto são capazes de criar comportamentos similares ou até diferentes dos demais tipos de juntas.

4.4 Colisão

A colisão ocorre quando dois corpos aproximam-se o suficiente e interagem durante um pequeno intervalo de tempo. Muitas vezes, as forças exercidas por um corpo sobre o outro são muito mais fortes que quaisquer outras forças externas presentes, e a duração da colisão é tão curta que os corpos não se movem apreciavelmente durante a interação. Podemos definir a colisão como um evento isolado em que uma força relativamente intensa age em cada um de dois ou mais corpos que interagem por um tempo relativamente curto.

Determinar quando dois objetos colidem não é tão simples quanto pode parecer, pois alguns objetos se movimentam muito rapidamente ou então possuem uma geometria complicada. Para se detectar a colisão, existem basicamente duas técnicas que podem ser empregadas: *overlap testing* e *intersection testing*. A principal diferença entre elas é que a técnica *overlap testing* detecta se a colisão já aconteceu enquanto a *intersection testing* tenta prever se uma colisão acontecerá no futuro.

A técnica mais comum é a *overlap testing*. A ideia consiste em testar os pares de objetos a cada passo da simulação para determinar se eles irão se sobrepor. Para isso, os dois objetos envolvidos devem voltar para o último passo em que eles não estão colidindo. Usando a técnica de bisseção, a simulação será avançada ou retardada pela metade do tempo com o intuito de convergir para o exato momento da colisão. Por exemplo, uma vez que a colisão é detectada, deve-se voltar ao tempo da última simulação. Então, ela deve ser avançada pela metade. Se houver colisão novamente, deve-se voltar e avançar por $\frac{1}{4}$ de tempo. Se não houver colisão, ela

deve avançar por $\frac{1}{8}$ de tempo e assim por diante. Isto é demonstrado na figura abaixo.

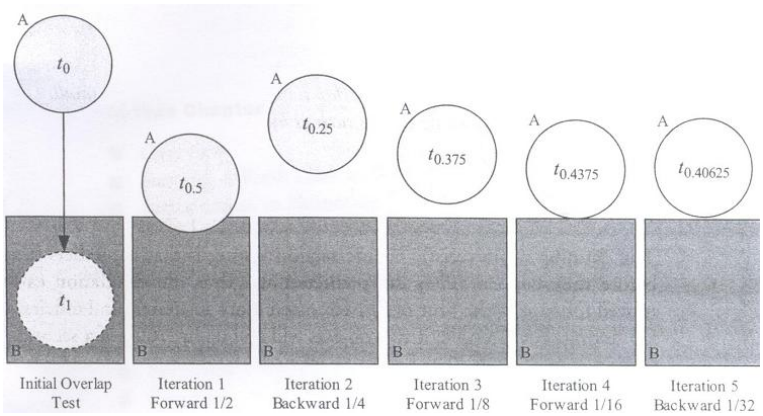

Figura 4.2: Exemplo de overlaping testing [9]

Na prática, boas soluções serão encontradas dentro de cinco iterações. Depois de calculado o tempo exato anterior a colisão, saberemos as posições corretas onde os objetos devem ser posicionados.

Como o Unity não disponibiliza a sua implementação, não conseguimos identificar que técnica é aplicada para resolver a questão da colisão dos objetos. Para o tratamento de colisão, ele dispõe dos Colliders. Eles envolvem o objeto com uma determinada forma geométrica (esfera, cubo, cápsula ou outras mais específicas) com o objetivo de detectar uma colisão e impedir que os objetos se interpenetrem. Alternativamente, eles podem ser usados como triggers, ou seja, ativam o processamento de um trecho de código caso ocorra uma colisão com os objetos.

4.5 Materiais físicos

Outra funcionalidade atribuída ao Unity é a modelagem de materiais físicos (Physics Material). As reações geradas pela colisão entre colliders são determinadas parcialmente através das propriedades individuais de cada superfície. O Unity possibilita o ajuste dos efeitos do atrito e da elasticidade na colisão.

Capítulo 5

Solução proposta

O presente trabalho implementa o SMTPG utilizando o game engine Unity. Não estava em seu escopo o design gráfico da simulação, portanto as malhas foram obtidas de um trabalho elaborado por alunos do TecGraf/PUC-Rio para a Petrobras. O modelo gráfico, desenhado com o software Autodesk 3ds Max, foi importado para o Unity sob o formato fbx. Feito isso, o modelo fbx foi inserido na pasta de Assets do projeto Unity, sendo em seguida automaticamente por ele incorporado.

O modelo gráfico elaborado pela PUC-Rio possuía algumas malhas que não caberiam a esse trabalho. Havia sido construído não só o braço mecânico, porém alguns dos elementos do ambiente que o circunda, como a caixa de energia e objetos para a realização de operações experimentais. Assim, o primeiro trabalho foi editar o modelo gráfico, removendo os GameObjects desnecessários.

Após essa edição, o que restou eram os GO's que seriam usadas para esse projeto, porém não havia qualquer hierarquia entre eles. Os GO's foram, então, separados em 4 grandes grupos, que são: Mão, Antebraço, Braço e Suporte. Essa divisão foi baseada na organização do braço humano, onde o suporte simboliza o ombro e os demais grupos equivalem ao que o seu nome representa. A figura 5.1 mostra essa organização.

Outros agrupamentos, internos aos citados anteriormente, merecem ser destacados, como os cilindros hidráulicos (os suspensores na hierarquia), os esqueletos e a garra. Os cilindros hidráulicos são os atuadores do sistema, isto é, produzem movimento, atendendo a comandos. O esqueleto é a proteção dos componentes internos do sistema (como ligações elétricas e os próprios suspensores). A garra, como descrito no capítulo 2, é o dispositivo efetuador.

As juntas (ou joints) são elementos importantes para a simulação. No Unity, existem

Figura 5.1: Hierarquia

vários tipos, conforme descrito no capítulo 4. O tipo HingeJoint, por exemplo, é o que une o suporte ao braço, o braço ao antebraço e o antebraço à mão. Possui funcionalidade semelhante às juntas de revolução explicadas no capítulo 2. Para unir as duas partes dos cilindros hidráulicos (veja a figura 5.2), utilizou-se o ConfigurableJoint, já que os tipos pré-definidos não conseguiam simular a dependência cinética entre elas. Este representa as juntas prismáticas explicadas no capítulo 2. Por último, destacam-se os FixedJoints, usados para unir os Rigidbodies que não possuem movimento relativo entre si. Por exemplo, os FixedJoints foram usados para unir os cilindros hidráulicos ao braço e ao antebraço. Os FixedJoints não possuem relação com os tipos de juntas do capítulo 2.

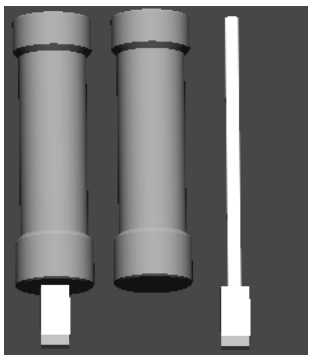

Figura 5.2: Partes do cilindro hidráulico

Na maioria das juntas implementadas por HingeJoints, somente os parâmetros relacionados à orientação e posicionamento do eixo de rotação foram ajustados. Em outros casos, como na ligação entre a Mão e a Garra (figura 5.3), os parâmetros relacionados a forças foram usados para gerar os movimentos de rotação no corpo dependente. A figura 5.4 ilustra bem os dois casos (as setas laranjas são os eixos). O corpo dependente, mencionado anteriormente, está

definido no campo “Connected body”. Na aba Anchor, definiu-se o posicionamento do eixo de rotação para (-3.45, 0.26, 0.04), relativo à origem do GameObject que possui o componente HingeJoint. Em Axis, definiu-se a orientação do eixo de rotação, o qual, nesse caso, deveria estar alinhado ao eixo X para simular o giro da garra (Wrist Rotate), na direção do vetor (-1, 0, 0). Em outras situações, o vetor direção foi o (0, 0, 1), como ilustrado também na figura 5.4.

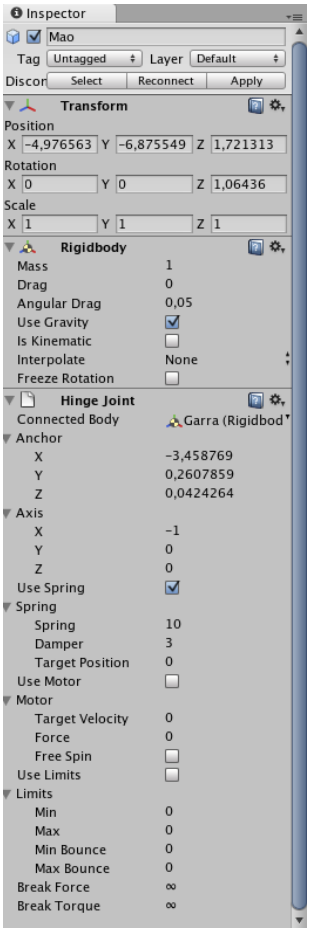

Figura 5.3: Propriedade Spring do HingeJoint

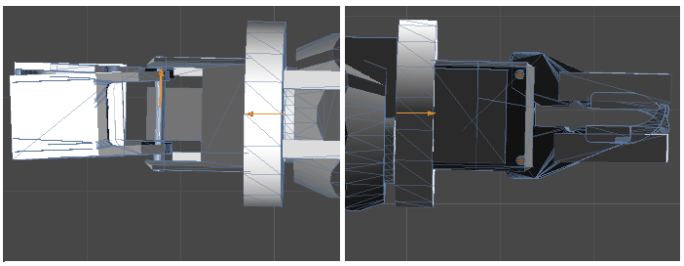

Figura 5.4: Eixo de rotação da garra

Para os cilindros hidráulicos, a configuração dos ConfigurableJoints foi mais trabalhosa, já que muitos parâmetros precisaram ser definidos e ajustados. A figura 5.5 mostra, em resumo, os parâmetros ajustados para o cilindro hidráulico. Logo abaixo de "Connected body", podem-se ver as propriedades XMotion, YMotion, ZMotion, AngularXMotion, AngularYMotion e AngularZMotion. Elas possuem 3 valores possíveis: Locked, Limited e Free. O valor Locked trava o movimento linear (Motion) ou angular (Angular motion). O valor Free, ao contrário, libera o movimento, ainda mantendo o corpo dependente conectado. O valor Limited limita

os movimentos do corpo dependente às propriedades da aba “Linear limits”, como “Limit” (distância máxima que o corpo dependente pode se afastar de sua origem, seguindo qualquer direção), “Bounyness”, “Spring” e “Damper”. A aba YDrive, bem como XDrive e ZDrive, funcionam em conjunto com as abas “Target Position” e “Target Velocity”. O Drive representa a aceleração com a qual o corpo dependente irá se movimentar para atingir um alvo (target).

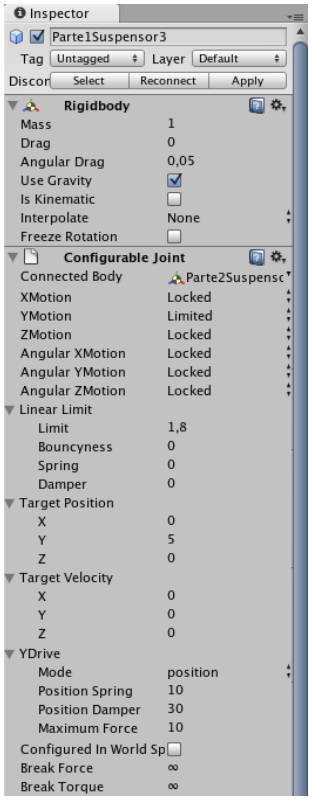

Figura 5.5: ConfigurableJoint

Alguns dos valores, não só dos Rigidbodies e Joints, mas também de outros componentes, foram retirados dos documentos técnicos do braço robótico real, como a carga máxima, a localização das juntas (que já estavam corretamente colocadas no modelo gráfico recebido da Petrobras) e a limitação de seus movimentos. As informações que não eram acessíveis foram estimadas ou inferidas. Casos de valores estimados são as massas de cada Rigidbody e as distâncias. A medida das distâncias foi obtida por inferência, através de um método próprio, que será descrito mais adiante. Aquelas que não se aplicavam ao problema foram mantidas com seus valores originais, como “Drag” e “Angular Drag” de Rigidbody e “break force” e “break torque” dos Joints.

Com exceção do suporte, no restante do braço mecânico os Rigidbodies foram aplicados a partir do terceiro nível da hierarquia, isto é, estão, por exemplo, nos GO's Parte1Antebraco, Parte2Antebraco, EsqueletoBraco, Parte1Suspensor3, Parte2Suspensor3 (figura 5.1). Se um nó possui um Rigidbody, seus filhos não possuirão, e vice-versa. Isso foi feito devido à característica de semi-independência entre GO's, que faz com que nós-pais e nós-filhos ajam independentemente uns dos outros quando estão sob o controle da engine de física. Em outras palavras, se fossem colocados Rigidbodies em um nó-pai e também em seus filhos, esses não mais acompanhariam o posicionamento (linear e angular) do ancestral em todo momento (somente no instante em que o nó-pai fosse movimentado). Além disso, seria gerada uma queda na performance devido à necessidade de inclusão de FixedJoints para uní-los.

Inicialmente, foi adotada a estratégia de inserir Rigidbodies em todos os GO's de mais baixo nível da hierarquia, no intuito de criar efeitos físicos mais próximos do real. Dessa forma, cada peça do braço robótico da simulação seria processada pela engine de física, podendo representar mais fielmente a ruptura física de peças, por exemplo. No entanto, obteve-se uma taxa de atualização de frames muito baixa, na ordem de 2 fps, já que a cada frame era preciso calcular dados físicos de mais de 100 Rigidbodies, sem contar cálculos de aproximadamente o mesmo número de juntas. Portanto, viu-se que era uma estratégia inadequada para o problema. Além disso, ultrapassava o nível de aprofundamento requerido. Deve-se acrescentar também que o efeito de ruptura pretendido inicialmente pode ser alcançado através da adição, por script, de Rigidbodies a todos ou alguns filhos de um GO.

Para a detecção de colisão, foram escolhidos mesh colliders. Embora demandem de mais poder de processamento, proporcionam uma detecção mais precisa. Essa característica não possui grande peso, para este trabalho, mas pode ser útil para trabalhos futuros, nos quais o braço robótico precisará interagir (i.e., colidir) com objetos do ambiente. Para compensar a demanda de processamento advinda do seu uso, os mesh colliders foram usados somente em algumas partes, preferencialmente em regiões onde a probabilidade de colisão seja maior, como nos esqueletos do braço e do antebraço, nas plataformas de contato da garra. Em alguns GO's, cujas malhas possuiam um formato mais padrão, foram utilizados collider primitivos, como o Box Collider.

Assim como no caso dos Rigidbodies, houve inicialmente uma estratégia também para a utilização de Colliders, que era colocar Mesh Colliders em todos os GO's. Tal estratégia mostrou-se igualmente inadequada, tendo em vista as demandas mencionadas no parágrafo

anterior, resultando em taxas de atualização de frames muito baixas. Houve também efeitos colaterais gerados pela colisão entre meshes, em que cada uma tentava expulsar a outra.

Foram elaborados os seguintes scripts para a simulação:

- Garra: realiza o movimento de abrir e fechar da garra
- Mao: realiza o movimento de girar a garra e delega ao script Garra a requisição de abrir e fechar
- Braco: delega requisições a suspensores (responsáveis pela elevação do Braço e do Antebraço)
- Antebraco: delega requisições ao suspensor que eleva a Mão e realiza o movimento de girar o antebraço
- Suspensor: modela um cilindro hidráulico, realizando o movimento de levantar e abaixar
- BracoRobotico: delega requisições a todos os demais scripts
- ControleBraco: é responsável por capturar a entrada do usuário, interpretá-la e fazer as chamadas dos métodos implementados no script do BracoRobotico

Os scripts da Garra, Mão e Suspensores acessam as instâncias em memória dos componentes dos GO's aos quais estão associados, alterando seus valores durante a execução da simulação. Mais especificamente, acessam os Joints, que são os responsáveis por todo tipo de movimento comandado pelo usuário. No apêndice A.2 é mostrado um exemplo de manipulação das propriedades dos Joints no script da Mão.

O script adicionado ao GO Braço Robótico não só delega as requisições do script ControleBraco, mas também realiza cálculos de física, durante sua inicialização, para determinar as forças que cada Suspensor terá que aplicar sobre a parte do braço robótico em que atua. Esses cálculos foram obtidos pela aplicação de conceitos básicos de física, como torque, forças, decomposição de componentes vetoriais, baseando-se nas leis de Newton.

Cada suspensor aplica o máximo de sua força quando o braço mecânico se encontra em uma configuração na qual o torque sobre a junta que movimenta é máximo. Essa condição é obtida quando se observa o estado apresentado na figura 5.6.

Nessa configuração, o torque aplicado é máximo porque a distância entre o eixo de rotação de cada junta e o ponto de aplicação do peso do objeto manipulado é o mais distante para todas as juntas.

A figuras 5.7, 5.8 e 5.9 mostram as forças, os ângulos e as distâncias que compõem o sistema. Basicamente, trata-se da força peso, aplicada a cada Rigidbody, de forças geradas

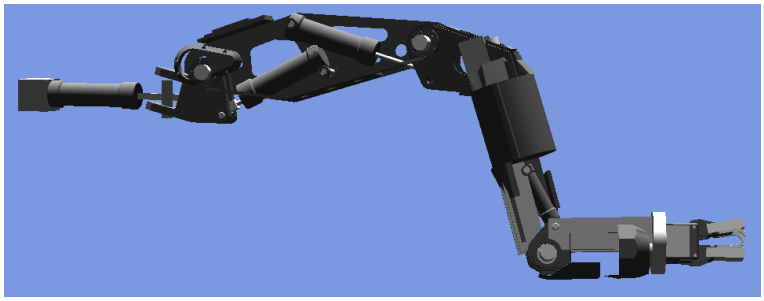

Figura 5.6: Configuração em que os torques são máximos

pelas articulações (F_{art_n}), as quais atuam como reação às forças geradas pelos suspensores, de forças geradas pelos suspensores (F_{P_n}) e da força peso gerada pelo objeto manipulado pelo braço robótico (F_{ext}).

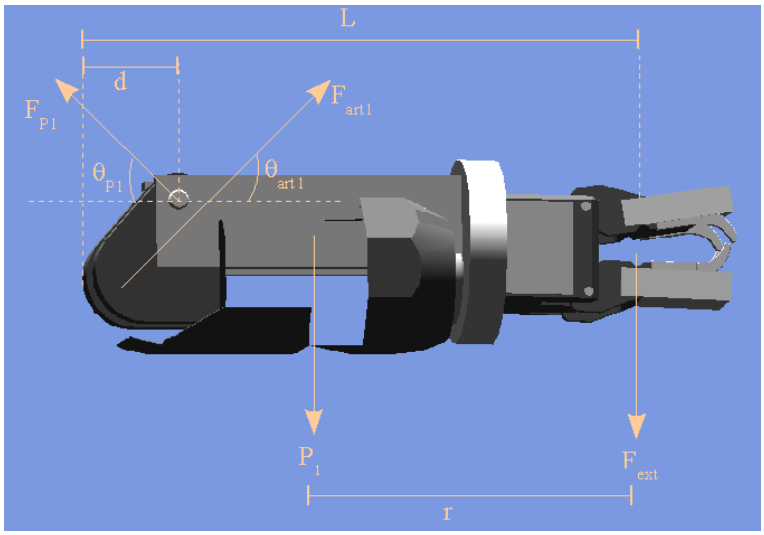

Figura 5.7: Atuação das forças na mão

Nas figuras 5.7, 5.8 e 5.9, os valores exatos das medidas das distâncias foram substituídos por variáveis, a fim de tornar o esquema mais genérico e menos poluído visualmente. Abaixo são mostrados os valores de cada variável utilizada:

- L = 5,3359
- r = 2,3755
- d = 0,3689
- L_2 = 7,76
- r_2 = 1,94
- d_3 = 2,768
- d_2 = 1,113

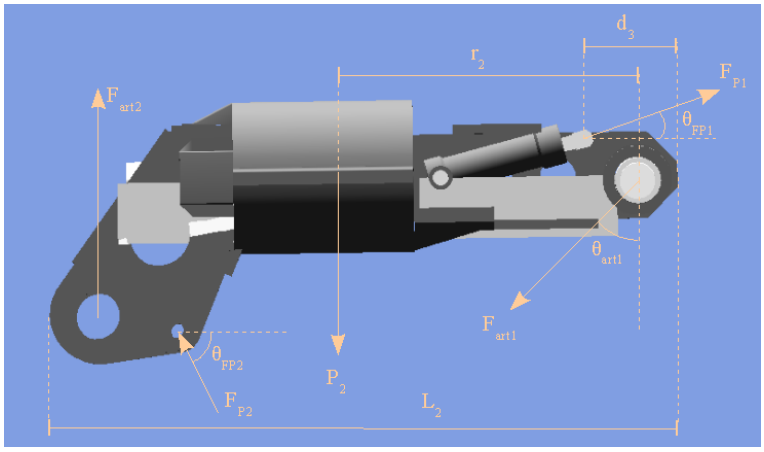

Figura 5.8: Atuação das forças no antebraço

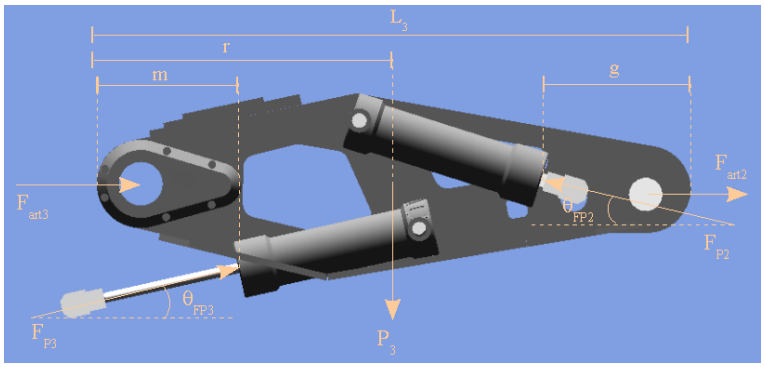

Figura 5.9: Atuação das forças no braço

- L_3 = 7,674
- r = 3,497
- m = 4,36
- g = 4,377

Após realizar a decomposição dos vetores em eixos cartesianos foi possível descrever as relações abaixo:

$$F_{P_1} = \frac{LF_{ext} + (L-r)P_1}{d}$$

$$F_{art_1} = \frac{F_{P_1} cos\theta_{F_{P_1}}}{cos\theta_{F_{art_1}}}$$

$$F_{P_2} = \frac{L_2 F_{art_1} cos\theta_{F_{art_1}} + (L_2 - r_2)P_2 - d_3 F_{P_1} sen\theta_{F_{P_1}}}{d_2 sen\theta_{F_{P_2}}}$$

$$F_{art_2} = P_2 + F_{art_1} cos\theta_{F_{art_1}} - F_{P_1} sen\theta_{F_{P_1}} - F_{P_2} sen\theta_{F_{P_2}}$$

$$F_{P_3} = \frac{rP_3 - (L-g)F_{P_2} sen\theta_{F_{P_2}}}{msen\theta_{F_{P_3}}}$$

$$F_{art_3} = F_{P_2} cos\theta_{F_{P_2}} - F_3 cos\theta_{F_{P_3}} - F_{art_2}$$

Substituindo as variáveis listadas acima pelos seus respectivos valores, tem-se:

$$F_{P_1} = \frac{5,3359F_{ext} + 2,9604P_1}{0,3689}$$

$$F_{art_1} = \frac{F_{P_1} cos\theta_{F_{P_1}}}{cos\theta_{F_{art_1}}}$$

$$F_{P_2} = \frac{7,76F_{art_1} cos\theta_{F_{art_1}} + 5,82P_2 - 2,768F_{P_1} sen\theta_{F_{P_1}}}{1,113sen\theta_{F_{P_2}}}$$

$$F_{art_2} = P_2 + F_{art_1} cos\theta_{F_{art_1}} - F_{P_1} sen\theta_{F_{P_1}} - F_{P_2} sen\theta_{F_{P_2}}$$

$$F_{P_3} = \frac{3,497P_3 - 3,297F_{P_2} sen\theta_{F_{P_2}}}{4,36sen\theta_{F_{P_3}}}$$

$$F_{art_3} = F_{P_2} cos\theta_{F_{P_2}} - F_3 cos\theta_{F_{P_3}} - F_{art_2}$$

Desenvolvendo esse sistema de equações, observa-se que a equação de F_{art_1} pode ser ignorada, pois ao substituir a variável F_{art_1} pela expressão $\frac{F_{P_1} cos\theta_{F_{P_1}}}{cos\theta_{F_{art_1}}}$ na equação de F_{P_2}, os cossenos podem ser anulados, fazendo com que o cálculo de F_{art_1} para a expressão $\frac{L_2 F_{art_1} cos\theta_{F_{art_1}} + (L_2 - r_2)P_2 - d_3 F_{P_1} sen\theta_{F_{P_1}}}{d_2 sen\theta_{F_{P_2}}}$ seja desnecessário. As equações de F_{art_2} e de F_{art_3} são igualmente dispensáveis, pois não figuram nas equações de F_{P_1}, F_{P_2} ou F_{P_3} .

Para esses cálculos, surgiu uma dificuldade, que era a obtenção no Unity das medidas das distâncias. Não existe no Unity uma maneira direta de obter o tamanho das meshes utilizadas. Assim, o método escolhido foi obter as medidas das distâncias através da diferença de pontos. Ele pode ser descrito da seguinte forma:

1- Criam-se dois GameObjects vazios (Empty GameObject)

2- Posicionam-se os dois sobre a mesh

3- Calcula-se a distância entre esses dois pontos pela fórmula:

$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}$

Além do tamanho total, era necessário também obter a distância do eixo de rotação até o centro de massa dos Rigidbodies. Para esse caso, apenas um GO foi utilizado (posicionado no eixo de rotação), pois o componente Rigidbody disponibiliza essa informação. Dessa forma, a distância do eixo de rotação até o centro de massa foi calculado utilizando também a fórmula descrita anteriormente.

Os valores dos pesos P_1, P_2 e P_3 foram obtidos dinamicamente dos componentes Rigidbody, acessados no script BracoRobotico pelos métodos getMassa() disponibilizados pelos scripts Antebraco, Braco e Mao. Os valores dos ângulos $\theta_{F_{P_1}}$, $\theta_{F_{P_2}}$ e $\theta_{F_{P_3}}$ foram definidos de modo que gerassem uma configuração em que o torque fosse máximo.

A aplicação dos cálculos mencionados acima, bem como todos os scripts, pode ser consultada no Apêndice A.

Foi criada uma interface gráfica do usuário (GUI) através do uso da própria API (Application Programming Interface) do Unity para realizar a entrada de comandos ao braço mecânico. O sistema de interface gráfica do Unity é chamada UnityGUI. Ela permite a criação fácil e rápida de uma grande variedade de GUI's. Em vez de criar um objeto de interface gráfica, posicioná-lo manualmente e, então, escrever um script que manipula suas funcionalidades, é possível fazer tudo isso em um pequeno trecho de código. Todas essas facilidades são conseguidas através da criação de "GUI Controls" (labels, botões, caixas de texto, etc), que são ao mesmo tempo instanciados, posicionados e definidos. Um exemplo de sua utilização pode ser visto no trecho de código mostrado na figura 5.10.

```
function OnGUI ()
{
    if (GUI.Button (Rect (10,10,150,100), "I am a button"))
    {
        print ("You clicked the button!");
    }
}
```

Figura 5.10: Function OnGUI [12]

A função OnGUI() é executada a cada frame, enquanto o script que o implementa estiver ativo. O comando de seleção é a todo momento avaliado para decidir quanto à execução do

comando print, de modo que um clique no botão “I am a button” retorna ao if uma resposta verdadeira e é impressa no console a mensagem “You clicked the button”.

Para realizar a entrada de comandos para a simulação, foram criados VerticalSliders, os quais enviam aos métodos do script BracoRobotico (como o método "girarGarra") os valores de seus parâmetros de acordo com a movimentação pelo usuário de uma barra de rolagem. Os VerticalSliders são ativados ou desativados através dos Toggles. A figura 5.11 ilustra a utilização desses controles.

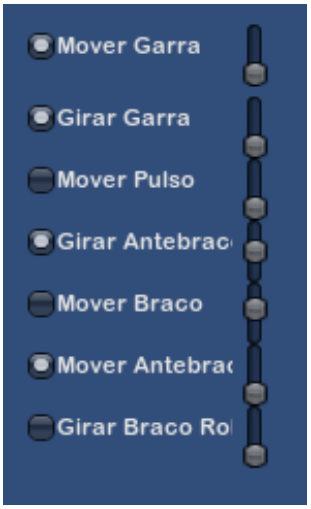

Figura 5.11: GUI da simulação

Capítulo 6

Resultados obtidos

A simulação desenvolvida apresentou uma taxa de atualização de frames aceitável, no valor de, aproximadamente, 400 fps(figura 6.1). Deve-se considerar que tal resultado foi obtido em uma situação na qual não havia outro cenário gráfico ou outros objetos dinâmicos. Pelo mesmo motivo, a quantidade de memória de vídeo utilizada é pouca, sendo 0,5% do total. Os elementos que mais estão influenciando nessa taxa são as chamadas aos métodos dos scripts e os Rigidbodies utilizados, os quais são razoavelmente elevados. Observa-se que o número de Draw Calls (número de objetos renderizados) é elevado; isso deve-se ao fato de o modelo gráfico ser high poly (i.e., possui alto número de polígonos).

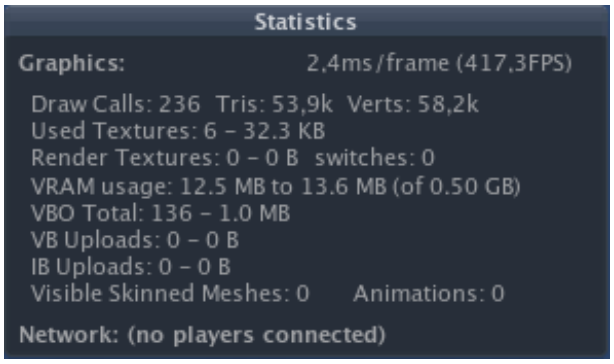

Figura 6.1: Janela Statistics da simulação

Alguns efeitos indesejáveis foram encontrados na simulação, os quais podem ser explicados com base em algumas hipóteses. O fato de não se conhecer a implementação do Unity e principalmente da PhysX é o motivo para que a explicação seja embasada em hipóteses.

É importante notar que a PhysX, assim como qualquer engine de física, não consegue reproduzir todos os efeitos físicos do mundo real com 100% de precisão. Um experimento

foi montado para mostrar tal afirmação. Criaram-se no Unity os GameObjects pré-definidos do tipo esfera e dois do tipo plano. Colocaram-se os componentes Collider e Rigidbody em cada um deles. Criaram-se também um Physic Material, os quais foram aplicados nos colliders. A propriedade *Bouncyness* foi definida com o valor 1, indicando que qualquer colisão que acontecesse no collider que fizesse uso dele fosse totalmente elástica. Os dois planos estavam paralelos e totalmente alinhados um ao outro. A esfera foi colocada entre os dois planos. Após iniciar o experimento, notou-se que a esfera ganhou energia. Trata-se de um efeito anômalo, não encontrado no mundo real, visto que em colisões elásticas, o objeto mantém constante sua energia.

Com o passar do tempo, a esfera vai ganhando energia até começar a colidir com o plano de cima. Essa nova colisão também lhe fornece energia, de modo que sua velocidade vai aumentando até que, após um certo tempo, ela traspassa um dos planos. Desta forma, pode-se perceber que a alta velocidade da esfera na simulação faz com que o passo de processamento da detecção de colisão não perceba sua presença.

Outro problema encontrado foi a falta de colisão (trespasse) entre meshes com collider no modelo gráfico. A figura 6.2 mostra a disposição das meshes antes da execução da simulação. Após o início da execução, elas se interpenetram, conforme se observa na figura 6.3. O Unity não fornece um tipo de collider que delineie espaços vazios (buracos) na mesh, característica desejável para simular o encaixe entre duas partes do braço robótico (por exemplo, a parte inferior do antebraço e o lado oposto à garra na mão). Já que não é oferecido esse recurso, foi preciso utilizar os tipos de collider existentes, deixando-os sobrepostos, conforme mostrado na figura 6.4. Como resultado, os colliders reagiam entre si, um tentando expulsar o outro. Era impossível não utilizá-los, pois além de serem necessários para evitar a interpenetração entre si, a sua ausência produziria um efeito no qual as partes do braço robótico pareceriam desconectadas uma das outras, embora seus movimentos continuassem limitados pelos Joints.

O objetivo do cálculo das forças aplicadas pelos suspensores era deixar de usar valores empíricos e, em substituição, obter um maior embasamento teórico. Tal objetivo foi alcançado. No entanto, não se observou totalmente o comportamento de alta precisão previsto pelas teorias da Física em todos os casos. Em parte, foi devido a imprecisões da engine de física do Unity. Adicionalmente, alguns erros numéricos certamente surgiram a partir do método de obtenção das distâncias descrito no Capítulo 5, já que o posicionamento dos GameObjects vazios era feito visualmente, além dos erros oriundos de seguidos arredondamentos nos resultados dos cálculos,

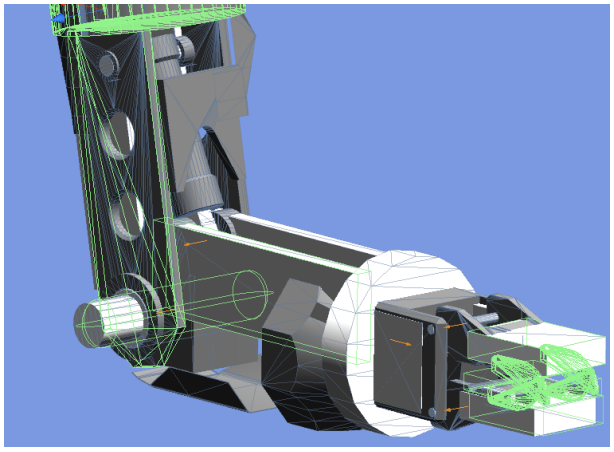

Figura 6.2: Ligação entre as partes do braço robótico

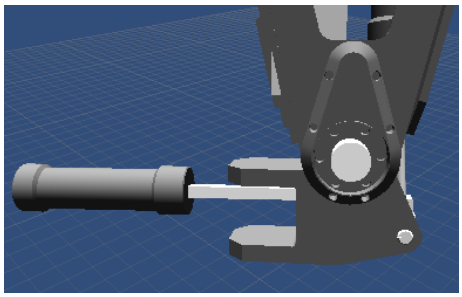

Figura 6.3: Problema da sobreposição de meshes - antes do início da execução

os quais, em muitos casos, eram obtidos após um número de passos.

Apesar dos problemas citados nos parágrafos anteriores desse capítulo, o resultado proposto para este trabalho foi alcançado, pois foi implementada uma simulação de um braço robótico que, embora com pouca precisão, responde corretamente a comandos do usuário. Por responder corretamente, pode-se considerar que o braço robótico realiza o movimento esperado após a manipulação de sua interface gráfica.

Um ponto forte dessa simulação é o fato de ela ter sido totalmente construída com componentes físicos. Não há movimento algum executado pela manipulação direta de posições e ângulos dos GameObjects. Isso lhe trouxe um aspecto mais realístico, além de fornecer mais abstração durante a implementação.

A implementação está bem modularizada. Tal característica está presente graças à hierarquia construída para o braço robótico. A modularidade permitiu que os scripts fossem implementados com mais facilidade (já que a responsabilidade de cada um era bem definida) e que erros de execução pudessem ser isolados e sua causa localizada precisamente.

A interface gráfica do usuário criada foi a mais simples possível, já que a escolha da

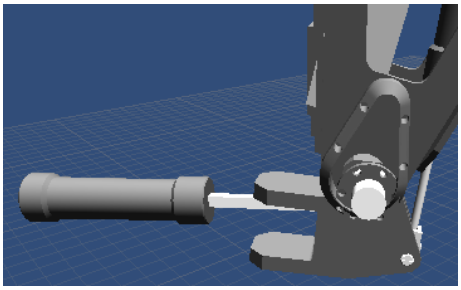

Figura 6.4: Problema da sobreposição de meshes - depois do início da execução

melhor interface não fazia parte do escopo deste trabalho. Ela é de fácil manuseio, bastando ao usuário clicar em Toggles para ativar ou desativar o controle de determinada parte do braço robótico e deslizar uma barra de rolagem para controlar sua movimentação. Assim, a interface cumpre seu objetivo principal, que é o de dar entrada aos comandos do braço robótico.

Capítulo 7

Conclusão e Sugestões de trabalhos futuros

O game engine Unity possibilita a criação de simulações físicas em tempo real, através da utilização de componentes que abstraem a implementação de vários conceitos, como conceitos de Física, de Computação Gráfica, de Design Gráfico, de colisão e de hardware. Essas abstrações trazem produtividade ao desenvolvedor, já que ele não tem a necessidade de se preocupar com todos esses detalhes. A não existência delas o levaria a implementar todos esses conceitos, o que lhe demandaria mais tempo a reservar para o projeto, sem levar em conta os defeitos que precisariam ser consertados até que fosse possível obter alguma confiabilidade.

No entanto, para o desenvolvimento de simulações com requisitos de grande precisão, como a desse trabalho, não é possível, usando o Unity, atender a todas as demandas de precisão elucidadas nos requisitos de sistema. Em parte, isso é devido à impossibilidade das engines de física de representar as leis físicas com total fidelidade. A falta de conhecimento da implementação do Unity e da PhysX impede uma depuração mais objetiva, levando o desenvolvedor a se basear em hipóteses.

Diante dos resultados apresentados, sugere-se que novos trabalhos sejam elaborados a fim de suprir necessidades pendentes e trazer melhoramentos a este. Abaixo estão listadas algumas sugestões:

- Pesquisar e implementar soluções para os problemas de imprecisão na simulação
- Aplicar os conceitos de cálculo de cinemática direta e/ou inversa para medir a precisão dos movimentos do braço robótico
- Aperfeiçoar o método de interação homem-máquina

Referências Bibliográficas

[1] ALEX NOGUEIRA BRASIL. *Tópicos Especiais em Robótica*. Capítulo 1. Disponível em: <http://www.alexbrasil.eng.br/arquivos/robotica/capitulo1_ introducao.pdf>.

[2] ALEX NOGUEIRA BRASIL. *Tópicos Especiais em Robótica*. Capítulo 2. Disponível em: <http://www.alexbrasil.eng.br/arquivos/robotica/capitulo2_ bracomecanico.pdf>.

[3] *Ambiental Híbrido*. Disponível em: <http://www.redetec.org.br/inventabrasil/ambienthib.htm>.

[4] CARNEIRO, Emanuel; GIRÃO, Frederico. Centro de Massa e Aplicações à Geometria. *Eureka!*, Rio de Janeiro: UFC, n. 21, 2005. Disponível em: <http://www.math.sunysb.edu/ girao/paper_ eureka/eureka.pdf>.

[5] CENTRO DE ENSINO E PESQUISA AVANÇADA. *Mecânica*. Disponível em: <http://efisica.if.usp.br/mecanica/basico/>.

[6] PASSOS, Erick Baptista; JÚNIOR, José Ricardo da Silva; RIBEIRO, Fernando Emiliano Cardoso; MOURÃO, Pedro Thiago. Tutorial: Desenvolvimento de Jogos com Unity3D. In: SIMPÓSIO BRASILEIRO DE GAMES E ENTRETENIMENTO DIGITAL, 8., 2009. Rio de Janeiro. *Anais...* p.1-30. Disponível em: <http://wwwusers.rdc.puc-rio.br/sbgames/09/_ proceedings/dat/_ pdfs/computing/tutorialComputing2.pdf>.

[7] PHILIP CHU. *Game Development with Unity*. Disponível em: <http://www.technicat.com/games/unity.html# Physics>

[8] *Projeto Cognitus*. Disponível em: <http://www.cognitus.org/>.

[9] RABIN, Steve. Game Programming: Math, Collision Detection, and Physics. In: _____. *Introduction to Game Development*. [S.l.]: [S.n.]. 982 p. cap. 4, p. 329-420.

[10] SANTOS, Auderi Vicente; GÓES, Emerson de; MIRANDA, Fernando Pellon; FREITAS, Gustavo Medeiros; SENA, José Almir; REIS, Ney Robinson Salvi dos; PANTA, Pedro Eduardo Gonzáles; FERREIRA, Rodrigo Carvalho; CERQUEIRA, Romulo Curty. Robô Ambiental Híbrido: um novo conceito em locomoção e monitoramento de áreas. In: SIMPÓSIO DE GEOTECNOLOGIAS NO PANTANAL, 2., 2009. Corumbá. *Anais*... Corumbá: Embrapa Informática Agropecuária/INPE, 2009. 11 p. p. 1-11. Disponível em: <http://www.geopantanal2009.cnptia.embrapa.br/cd/pdf/p68.pdf>.

[11] SILVA, Alexandre Francisco Barral. *Modelagem de Sistemas Robóticos Móveis para Controle de Tração em Terrenos*. Rio de Janeiro, 2007. 194f. Dissertação (Pós-Graduação em Engenharia Mecânica) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007. Disponível em: <http://www.maxwell.lambda.ele.puc-rio.br/Busca_etds.php? strSecao=resultado& nrSeq=11030@1>.

[12] UNITY TECHNOLOGIES. Disponível em: <http://unity3d.com>.

Apêndice A

Apêndice

A.1 Script Garra.js

```
static var SENTIDO_HORARIO:int = 1;
static var SENTIDO_ANTI_HORARIO:int = -1;

static var aberturaMaximaPlataformas:float; //em graus

private var plataforma1:HingeJoint;
private var plataforma2:HingeJoint;

private var massaLevantada:float;

function setSpring(plataformaID:int, spring:float, damper:float)
{
        switch(plataformaID)
        {
                case 1:
                        plataforma1.spring.spring = spring;
                        plataforma1.spring.damper = damper;
                break;
                case 2:
                        plataforma2.spring.spring = spring;
                        plataforma2.spring.damper = damper;
                break;
                default:
                        Debug.Log("plataformaID errada");
```

```
25                  break;
            }
    }

    function moverPlataformas(angulo:float)
30  {
            if ((angulo >= 0) && (angulo <= aberturaMaximaPlataformas))
            {
                    plataforma1.spring.targetPosition = angulo/2;
                    plataforma2.spring.targetPosition = -angulo/2;
35          }
    }

    //Retorna a abertura em graus entre as plataformas
    function getAberturaPlataformas():float
40  {
            return Mathf.Abs(plataforma1.spring.targetPosition) +
               Mathf.Abs(plataforma2.spring.targetPosition);
    }

45  function getAberturaMaximaPlataformas():float
    {
            return aberturaMaximaPlataformas;
    }

50  function medirMassa(massa:float)
    {
            massaLevantada = massa;
    }

55  function getMassaLevantada():float
    {
            return massaLevantada;
    }

60  function Start()
    {
            var hingeJoints = gameObject.GetComponents (HingeJoint);
            plataforma1 = hingeJoints[0];
```

```
        plataforma2 = hingeJoints[1];
        plataforma1.useSpring = true;
        plataforma2.useSpring = true;

        aberturaMaximaPlataformas = 90;
}
```

A.2 Script Mao.js

```
var garra:Garra;
private var garraHingeJoint:HingeJoint;

function girarGarra(sentido:int, anguloDestino:float)
{
        if (sentido == Garra.SENTIDO_HORARIO)
                garraHingeJoint.spring.targetPosition = anguloDestino;
        else
                if (sentido == Garra.SENTIDO_ANTI_HORARIO)
                        garraHingeJoint.spring.targetPosition = -anguloDestino;
}

function girarGarraContinuamente(sentido:int, targetVelocity:float, force:float)
{
        garraHingeJoint.useMotor = true;
        garraHingeJoint.motor.force = force;
        if (sentido == Garra.SENTIDO_HORARIO)
                garraHingeJoint.motor.targetVelocity = targetVelocity;
        else
                if (sentido == Garra.SENTIDO_ANTI_HORARIO)
                        garraHingeJoint.motor.targetVelocity = - targetVelocity;
}

function frearGiroContinuo()
{
        garraHingeJoint.useMotor = false;
}

function setSpring(spring:float, damper:float)
{
```

```
        garraHingeJoint.spring.spring = spring;
        garraHingeJoint.spring.damper = damper;
}

function moverPlataformas(angAbertura:float)
{
        garra.moverPlataformas(angAbertura);
}

function getAberturaPlataformas():float
{
        return garra.getAberturaPlataformas();
}

function getAnguloRotacaoGarra():float
{
        return garraHingeJoint.spring.targetPosition;
}

function getAberturaMaximaPlataformas():float
{
        return garra.getAberturaMaximaPlataformas();
}

function getMassaLevantada():float
{
        return garra.getMassaLevantada();
}

function getMassa():float
{
        var rbsGarra:float =
    gameObject.Find("Garra").GetComponent(Rigidbody).mass;
        var rbsSuporte:float =
    gameObject.Find("SuporteSuportePlataforma").GetComponent(Rigidbody).mass;
        return rbsGarra + rbsSuporte;
}

function Start()
```

```
{
        garraHingeJoint =
    gameObject.Find("SuporteSuportePlataforma").GetComponent(HingeJoint);
        garraHingeJoint.useSpring = true;
}
```

A.3 Script Suspensor.js

```
protected var maxPosition:float;
protected var minPosition:float;
protected var suspensor:GameObject;
protected var joint:ConfigurableJoint;
protected var joint2:HingeJoint;
protected var noLimite:boolean;

function mover(tgtPosition:float)
{
        if ((tgtPosition > minPosition) && (tgtPosition < maxPosition))
                joint.targetPosition.y = tgtPosition;
}

function setDrive(posSpring:float, posDamper:float)
{
        joint.yDrive.positionSpring = posSpring;
        joint.yDrive.positionDamper = posDamper;
}

function getAngle():float
{
        return joint2.angle;
}

function getMaxPosition():float
{
        return maxPosition;
}

function getMinPosition():float
{
```

```
        return minPosition;
}

function tratarColisao(tipoColisao:String)
{
        Debug.Log(tipoColisao);
}
```

A.4 Script Suspensor1.js

```
class Suspensor1 extends Suspensor
{
        function Start()
        {
                suspensor = GameObject.Find("Suspensor1");
                joint = suspensor.Find("Parte1Suspensor1").
        GetComponent("ConfigurableJoint");
                joint2 = suspensor.Find("Parte2Suspensor1").
        GetComponent("HingeJoint");
                minPosition = -8;
                maxPosition = 5;
        }

}
```

A.5 Script Suspensor2.js

```
class Suspensor2 extends Suspensor
{
        function Start()
        {
                suspensor = GameObject.Find("Suspensor2");
                joint = suspensor.Find("Parte1Suspensor2").
        GetComponent("ConfigurableJoint");
                joint2 = suspensor.Find("Parte2Suspensor2").
        GetComponent("HingeJoint");
                minPosition = 0;
                maxPosition = 20;
```

```
        }
}
```

A.6 Script Suspensor3.js

```
class Suspensor3 extends Suspensor
{
        function Start()
        {
                suspensor = GameObject.Find("Suspensor3");
                joint = suspensor.Find("Parte1Suspensor3").
        GetComponent("ConfigurableJoint");
                joint2 = suspensor.Find("Parte2Suspensor3").
        GetComponent("HingeJoint");
                minPosition = 0;
                maxPosition = 10;
        }
}
```

A.7 Script Suspensor4.js

```
class Suspensor4 extends Suspensor
{
        function Start()
        {
                suspensor = GameObject.Find("Suspensor4");
                joint = suspensor.Find("Parte1Suspensor4").
        GetComponent("ConfigurableJoint");
                joint2 = suspensor.Find("Parte2Suspensor4").
        GetComponent("HingeJoint");
                minPosition = 0;
                maxPosition = 1.8;
        }

        function mover(tgtPosition:float)
        {
                if ((tgtPosition > minPosition) && (tgtPosition < maxPosition))
                        joint.targetPosition.x = tgtPosition;
```

```
        }

        function setDrive(posSpring:float, posDamper:float)
        {
                joint.xDrive.positionSpring = posSpring;
                joint.xDrive.positionDamper = posDamper;
        }
}
```

A.8 Script Antebraco.js

```
static var SENTIDO_HORARIO:int = 1;
static var SENTIDO_ANTI_HORARIO:int = -1;
private var anguloMaximoAntebraco:float;
private var anguloMinimoAntebraco:float;

var suspensor3:Suspensor3;
private var parte2AntebracoHingeJoint:HingeJoint;
private var juntaAntebracoMao:HingeJoint;

function setDriveSuspensor3(posSpring:float, posDamper:float)
{
        suspensor3.setDrive(posSpring, posDamper);
}

function getAngleSuspensor3():float
{
        //337.5947 é o ângulo inicial da parte2 do suspensor 3
        return suspensor3.getAngle() + 337.5947;
}

function getAngleJuntaAntebracoMao():float
{
        //349.1764 é o ângulo incial da parte 2 do antebraço
        return juntaAntebracoMao.angle + 349.1764;
}

function getMassa():float
{
```

```
        var rbsP1A:float = gameObject.Find("Parte1Antebraco").
30  GetComponent(Rigidbody).mass;
        var rbsP2A:float = gameObject.Find("Parte2Antebraco").
    GetComponent(Rigidbody).mass;
        var rbsP1S3:float = gameObject.Find("Parte1Suspensor3").
    GetComponent(Rigidbody).mass;
35      var rbsP2S3:float = gameObject.Find("Parte2Suspensor3").
    GetComponent(Rigidbody).mass;
        return rbsP1A + rbsP2A + rbsP1S3 + rbsP2S3;
}

40  function movimentarPulso(tgtPosition:float)
{
        suspensor3.mover(tgtPosition);
}

45  function getMinPositionSuspensor3():float
{
        return suspensor3.getMinPosition();
}

50  function getMaxPositionSuspensor3():float
{
        return suspensor3.getMaxPosition();
}

55  function getAnguloAntebraco():float
{
        return parte2AntebracoHingeJoint.spring.targetPosition;
}

60  function getAnguloMinAntebraco():float
{
        return anguloMinimoAntebraco;
}

65  function getAnguloMaxAntebraco():float
{
        return anguloMaximoAntebraco;
```

```
}

function girarAntebraco (sentido:int, anguloDestino:float)
{
        if ((anguloDestino >= anguloMinimoAntebraco) &&
       (anguloDestino <= anguloMaximoAntebraco))
                if (sentido == Antebraco.SENTIDO_HORARIO)
                        parte2AntebracoHingeJoint.spring.targetPosition =
                                anguloDestino;
                else
                        if (sentido == Antebraco.SENTIDO_ANTI_HORARIO)
                                parte2AntebracoHingeJoint.spring.targetPosition =
                                -anguloDestino;
}

function setSpring(spring:float, damper:float)
{
        parte2AntebracoHingeJoint.spring.spring = spring;
        parte2AntebracoHingeJoint.spring.damper = damper;
}

function Start()
{
        parte2AntebracoHingeJoint = gameObject.Find("Parte1Antebraco").
    GetComponent(HingeJoint);
        parte2AntebracoHingeJoint.useSpring = true;

        juntaAntebracoMao = gameObject.Find("Parte2Antebraco").
    GetComponent(HingeJoint);

        anguloMinimoAntebraco = -90;
        anguloMaximoAntebraco = 90;
}
```

A.9 Script Braco.js

```
var suspensor1:Suspensor1;
var suspensor2:Suspensor2;
```

```
private var juntaBracoAntebraco:HingeJoint;

function setDriveSuspensor1(posSpring:float, posDamper:float)
{
        suspensor1.setDrive(posSpring, posDamper);
}

function setDriveSuspensor2(posSpring:float, posDamper:float)
{
        suspensor2.setDrive(posSpring, posDamper);
}

function getAngleSuspensor1():float
{
        return suspensor1.getAngle();
}

function getAngleSuspensor2():float
{
        //17.44901 é o ângulo inicial da parte 2 do suspensor 2
        return suspensor2.getAngle() + 17.44901;
}

function getMassa():float
{
        var rbsEsqueleto:float = gameObject.Find("EsqueletoBraco").
    GetComponent(Rigidbody).mass;
        var rbsP1S1:float = gameObject.Find("Parte1Suspensor1").
    GetComponent(Rigidbody).mass;
        var rbsP2S1:float = gameObject.Find("Parte2Suspensor1").
    GetComponent(Rigidbody).mass;
        var rbsP1S2:float = gameObject.Find("Parte1Suspensor2").
    GetComponent(Rigidbody).mass;
        var rbsP2S2:float = gameObject.Find("Parte2Suspensor2").
    GetComponent(Rigidbody).mass;
        return rbsEsqueleto + rbsP1S1 + rbsP2S1 + rbsP1S2 + rbsP2S2;
}

function getAngleJuntaBracoAntebraco():float
```

```
{
        return juntaBracoAntebraco.angle;
}

function getMinPositionSuspensor1():float
{
        return suspensor1.getMinPosition();
}

function getMaxPositionSuspensor1():float
{
        return suspensor1.getMaxPosition();
}

function getMinPositionSuspensor2():float
{
        return suspensor2.getMinPosition();
}

function getMaxPositionSuspensor2():float
{
        return suspensor2.getMaxPosition();
}

function movimentarBraco(tgtPosition:float)
{
        suspensor1.mover(tgtPosition);
}

function movimentarAntebraco(tgtPosition:float)
{
        suspensor2.mover(tgtPosition);
}

function Start()
{
        juntaBracoAntebraco = gameObject.Find("EsqueletoBraco").
    GetComponent(HingeJoint);
}
```

A.10 Script BracoRobotico.js

```
var antebraco:Antebraco;
var mao:Mao;
var braco:Braco;
var suspensor4:Suspensor4;

private var fp1:float;
private var fp2:float;
private var fp3:float;

private var g = 9.8;
private var cargaMaxima:float = 210;
private var tetaP1:float = 320 * Mathf.Deg2Rad;
private var tetaP2:float = 117 * Mathf.Deg2Rad;
private var tetaP3:float = 50 * Mathf.Deg2Rad;

function getAberturaPlataformas():float
{
        return mao.getAberturaPlataformas();
}

function getAberturaMaximaPlataformas():float
{
        return mao.getAberturaMaximaPlataformas();
}

function getAnguloRotacaoGarra():float
{
        return mao.getAnguloRotacaoGarra();
}

function getAnguloAntebraco():float
{
        return antebraco.getAnguloAntebraco();
}

function getAnguloMinAntebraco():float
```

```
    {
        return antebraco.getAnguloMinAntebraco();
40  }

    function getAnguloMaxAntebraco():float
    {
        return antebraco.getAnguloMaxAntebraco();
45  }

    function getMinPositionSuspensor1():float
    {
        return braco.getMinPositionSuspensor1();
50  }

    function getMaxPositionSuspensor1():float
    {
        return braco.getMaxPositionSuspensor1();
55  }

    function getMinPositionSuspensor2():float
    {
        return braco.getMinPositionSuspensor2();
60  }

    function getMaxPositionSuspensor2():float
    {
        return braco.getMaxPositionSuspensor2();
65  }

    function getMinPositionSuspensor3():float
    {
        return antebraco.getMinPositionSuspensor3();
70  }

    function getMaxPositionSuspensor3():float
    {
        return antebraco.getMaxPositionSuspensor3();
75  }
```

```
function getMinPositionSuspensor4():float
{
        return suspensor4.getMinPosition();
}

function getMaxPositionSuspensor4():float
{
        return suspensor4.getMaxPosition();
}

function moverPlataformas(angAbert:float)
{
        mao.moverPlataformas(angAbert);
}

function girarGarra(sentGiro:int, angDest:float)
{
        mao.girarGarra(sentGiro, angDest);
}

function girarGarraContinuamente(sentido:int, targetVelocity:float, force:float)
{
        mao.girarGarraContinuamente(sentido, targetVelocity, force);
}

function movimentarPulso(targetPositPulso:float)
{
        antebraco.movimentarPulso(targetPositPulso);
}

function girarAntebraco (sentGiroAntebraco:int, targetAngAntebraco:float)
{
        antebraco.girarAntebraco(sentGiroAntebraco, targetAngAntebraco);
}

function movimentarAntebraco(targetPositAntebraco:float)
{
        braco.movimentarAntebraco(targetPositAntebraco);
}
```

```
function movimentarBraco(targetPositBraco:float)
{
        braco.movimentarBraco(targetPositBraco);
}

function girarBracoRobotico(targetPositBracoRobotico:float)
{
        suspensor4.mover(targetPositBracoRobotico);
}

function Start()
{
        fp1 = (5.3359*cargaMaxima*g + 2.9604 * mao.getMassa()*g) / 0.3689;
        fp2 = (fp1*Mathf.Cos(tetaP1)*7.76 + antebraco.getMassa()*g*5.82
        - fp1*Mathf.Sin(tetaP1)*2.768) / (1.113 * Mathf.Sin(tetaP2));
        fp3 = (3.497*braco.getMassa()*g - fp2*Mathf.Sin(tetaP2)*3.297)
        / (Mathf.Sin(tetaP3)*4.36);
        antebraco.setDriveSuspensor3(fp1, 3*fp1);
        braco.setDriveSuspensor2(fp2, 3*fp2);
        braco.setDriveSuspensor1(fp3, 3*fp3);
}
```

A.11 Script ControleBraco.js

```
var bracoRobotico:BracoRobotico;

var moverGarra:boolean; var anguloAbertura:float;
var girarGarra:boolean; var sentidoHorario:boolean;
var sentidoAntiHorario:boolean;
var anguloDestino:float; var moverPulso:boolean;
var targetPositionPulso:float; var girarAntebraco:boolean;
var sentidoGiroAntebraco:int;
var targetAngleAntebraco:float;
var moverBraco:boolean; var targetPositionBraco:float;
var moverAntebraco:boolean; var targetPositionAntebraco:float;
var girarBracoRobotico:boolean;
var targetPositionBracoRobotico:float;
```

```
15 function OnGUI ()
   {
        moverGarra =
        GUI.Toggle (Rect (25, 25, 100, 30), moverGarra, "Mover␣Garra");
        girarGarra =
20      GUI.Toggle (Rect (25, 60, 100, 30), girarGarra, "Girar␣Garra");
        moverPulso =
        GUI.Toggle (Rect (25, 90, 100, 30), moverPulso, "Mover␣Pulso");
        girarAntebraco =
            GUI.Toggle (Rect (25, 120, 100, 30), girarAntebraco,
25       "Girar␣Antebraco");
        moverBraco =
        GUI.Toggle (Rect (25, 150, 100, 30), moverBraco, "Mover␣Braco");
        moverAntebraco =
            GUI.Toggle (Rect (25, 180, 100, 30), moverAntebraco,
30       "Mover␣Antebraco");
        girarBracoRobotico =
            GUI.Toggle (Rect (25, 210, 100, 30), girarBracoRobotico,
        "Girar␣Braco␣Robotico");
        anguloAbertura =
35           GUI.VerticalSlider (Rect (130, 25, 100, 30), anguloAbertura,
          bracoRobotico.getAberturaMaximaPlataformas(), 0.0);
        anguloDestino =
            GUI.VerticalSlider (Rect (130, 60, 100, 30),
                anguloDestino, 360.0, 0.0);
40      targetPositionPulso =
            GUI.VerticalSlider (Rect (130, 90, 100, 30),
          targetPositionPulso,
                bracoRobotico.getMaxPositionSuspensor3(),
          bracoRobotico.getMinPositionSuspensor3());
45      targetAngleAntebraco =
            GUI.VerticalSlider (Rect (130, 120, 100, 30),
          targetAngleAntebraco,
                bracoRobotico.getAnguloMaxAntebraco(),
          bracoRobotico.getAnguloMinAntebraco());
50      targetPositionBraco =
            GUI.VerticalSlider (Rect (130, 150, 100, 30),
          targetPositionBraco,
                bracoRobotico.getMaxPositionSuspensor1(),
```

```
            bracoRobotico.getMinPositionSuspensor1());
        targetPositionAntebraco =
            GUI.VerticalSlider (Rect (130, 180, 100, 30),
                targetPositionAntebraco,
                bracoRobotico.getMaxPositionSuspensor2(),
                bracoRobotico.getMinPositionSuspensor2());
        targetPositionBracoRobotico =
            GUI.VerticalSlider (Rect (130, 210, 100, 30),
            targetPositionBracoRobotico,
                bracoRobotico.getMaxPositionSuspensor4(),
            bracoRobotico.getMinPositionSuspensor4());

        if (GUI.changed)
        {
            if (moverGarra)
                bracoRobotico.moverPlataformas(anguloAbertura);
            if (girarGarra)
                bracoRobotico.girarGarra(Garra.SENTIDO_HORARIO,
            anguloDestino);
            if (moverPulso)
                bracoRobotico.movimentarPulso(targetPositionPulso);
            if (girarAntebraco)
                bracoRobotico.girarAntebraco(Antebraco.SENTIDO_HORARIO,
            targetAngleAntebraco);
            if (moverAntebraco)
                bracoRobotico
                .movimentarAntebraco(targetPositionAntebraco);
            if (moverBraco)
                bracoRobotico.movimentarBraco(targetPositionBraco);
            if (girarBracoRobotico)
                bracoRobotico.
            girarBracoRobotico(targetPositionBracoRobotico);
        }
    }
```