**UNIVERSIDADE REGIONAL DE BLUMENAU**

**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**

**CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO**

# HEFESTO – FRAMEWORK PARA SIMULAÇÕES UTILIZANDO UM MOTOR DE FÍSICA EM 3D

**THIAGO HENRIQUE TEIXEIRA**

**BLUMENAU**
**2015**

**2015/1-27**

**THIAGO HENRIQUE TEIXEIRA**

# HEFESTO – FRAMEWORK PARA SIMULAÇÕES UTILIZANDO UM MOTOR DE FÍSICA EM 3D

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Dalton Solano dos Reis, M. Sc. – Orientador

**BLUMENAU**
**2015**

**2015/1-27**

# HEFESTO – FRAMEWORK PARA SIMULAÇÕES UTILIZANDO UM MOTOR DE FÍSICA EM 3D

Por

**THIAGO HENRIQUE TEIXEIRA**

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

| | |
|---|---|
| Presidente: | Prof. Dalton Solano dos Santos, M. Sc – Orientador, FURB |
| Membro: | Prof. Antonio Carlos Tavares, M. Sc – FURB |
| Membro: | Prof. Aurélio Faustino Hoppe, M. Sc – FURB |

Blumenau, 02 de julho de 2015

Dedico este trabalho a todo aquele que busca excelência no que faz, e o faz por amor.

## AGRADECIMENTOS

A Ele, pois Ele é digno de toda gratidão.

Aos meus pais. Que me deram parâmetros e atributos. Que trataram de muitos bugs. Por acreditar que possuo a capacidade de aprender e ser melhor a cada dia; e que acima de tudo, se orgulham do que entregaram ao mundo.

A minha amada esposa. Pois me olha e sorri, e é impossível que eu não faça o mesmo.

Aos amigos. Pela paciência e pelos ouvidos, que de maneira muito repetitiva foram usados.

À Milka, por ter o melhor chocolate do mundo.

Aos fones de ouvido. Os piores problemas enfrentados neste trabalho não teriam sido resolvidos sem eles.

Por criar o Kindle, à Amazon.

Ao Dalton; que acreditou neste trabalho mais do que eu mesmo, e que não me cedeu apenas seu tempo, mas sim, sua amizade.

A verdade é que não há nada de digno em ser superior a outra pessoa. A única nobreza genuína é ser superior ao seu antigo eu.

Whitney M. Young Jr

# RESUMO

Este trabalho apresenta o desenvolvimento de um *framework* para simulações utilizando um motor de física em 3D. Para tanto, foi realizado o desenvolvimento de um Motor de Física, uma Biblioteca Remota e um Cliente Remoto – esta estrutura sendo chamada Hefesto. No Motor de Física foi realizada a migração e refatoração da engine Cyclone de Millington (2007), para a linguagem Java; a Biblioteca Remota é um mecanismo para publicar as funcionalidades do Motor através de WebSockets; o cliente Remoto é uma biblioteca JavaScript que estabelece comunicação com a Biblioteca Remota e integra com a biblioteca ThreeJS para desenho de cenas 3D. Na criação da Biblioteca Remota, optou-se por WebSockets dado seu baixo custo para manter a conexão, além da inexistência de protocolo enrijecido – permitindo a criação de um protocolo próprio, viabilizando a construção e utilização de clientes para a Biblioteca através de qualquer plataforma. No decorrer da implementação desse trabalho, a integração entre Cliente e Biblioteca Remota através de WebSockets mostrou eficiência e performance, garantindo que a integridade das simulações realizadas com o Motor de Física fossem mantidas mesmo com a simulação operando remotamente. Com o uso do Hefesto construíu-se algumas aplicações, dentre elas a aplicação chamada Ballistic que simula a lei de Hooke, voltada para uso didático na disciplina de física do ensino médio.

Palavras-chave: Motor de física. Mecânica clássica. Biblioteca remota.

# ABSTRACT

This paper presents the development of a simulation's framework using a 3D physics engine. For such, a Physics Engine, a Remote Library and a Remote Client were developed – this structure being named Hefesto. On the Physics Engine, Millington's (2007) Cyclone engine was migrated and refactored to the Java language; the Remote Library is a mechanism for publishing the Engine features through WebSockets; the Remote Client is a JavaScript library that communicates with the Remote Library, and integrates with the ThreeJS library for 3D scene drawing. In the Remote Library design, the choice was to use WebSockets, given it's low cost to keep connection, besides the absence of a rigid protocol - allowing the creation of a custom protocol, and Library clients development through any platform. During this paper's development, the integration between Client and Remote Library through WebSockets has shown efficiency and performance, assuring the integrity of the Physics Engine simulations were kept even with remote simulation execution. With Hefesto, a few applications were developed, among them an application called Ballistic was built to simulate Hooke's law, focused on high school physics didactic teaching.

Key-words: Physics engine. Classical mechanics. Remote library.

# LISTA DE FIGURAS

# LISTA DE QUADROS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

2D - Duas dimensões

3D - Três dimensões

BR - Biblioteca Remota

CR - Cliente Remoto

EA - *Enterprise Architect*

FPS - *Frames* Por Segundo

FURB - Fundação da Universidade Regional de Blumenau

HTML5 - *Hypertext Markup Language 5*

HTTP - *Hypertext Transfer Protocol*

IDE - *Integrated Development Environment*

JSON - *JavaScript Object Notation*

MEC - Ministério da Educação

MF - Motor de Física

RF - Requisito Funcional

RNF - Requisito Não Funcional

UC - *Use Case*

UML - *Unified Modeling Language*

USP - Universidade de São Paulo

# SUMÁRIO

# 1 INTRODUÇÃO

A palavra física vem do latim *physike*, que significa "natureza"; no grego, *phusikē epistēmē* refere-se ao conhecimento da natureza (OXFORD, 2014). A física como tema de estudo, foi introduzida no Brasil nas primeiras décadas do século XIX, porém, foi delimitada como um conjunto de ciências e não como uma ciência autônoma. Seu ensino propriamente dito somente passou a ocorrer em 1.832, não para formar professores ou físicos, mas sim, médicos e militares (BASTOS, 2010, p. 82-84). A partir de 1.934, com a criação da Faculdade de Ciências e Letras da Universidade de São Paulo (USP), a física passou a ser tratada de forma autônoma. E somente na década de 50 passa a surgir a disciplina correspondente nas escolas de ensino básico até o superior (ROSA; ROSA, 2005, p. 4).

Contudo, seu ensino com qualidade requer constante atualização por parte dos professores. É necessário articular entre a teoria e a prática, contextualizando o ensino para que se torne significativo para o estudante (PEREIRA; AGUIAR, 2006, p. 69). O ambiente escolar é considerado o local apropriado para adquirir conceitos como os da física. Mas, estes conceitos mostram-se diretamente ligados ao cotidiano do estudante, ao que é expontâneo, que é de convívio de cada um e ao que o cerca (ROSA; ROSA, 2005, p. 10).

Observa-se que os professores constumam reproduzir o próprio processo de aprendizagem, e percebe-se a tendência a abordar a Física como apenas uma aplicação de fórmulas e resolução de problemas, limitando o estudante a questionar suas dificuldades com cálculo. Com esta didática o aluno perde a riqueza de explorar os fenômenos da natureza e a essência de seus conceitos (BASTOS, 2010, p. 86).

Segundo Pereira e Aguiar (2006, p. 69), “são evidentes as dificuldades dos professores da área de ciências da natureza, em particular da física, para se atualizarem, tanto em sua área de conhecimento quanto em questões gerais, relativas a educação acadêmica.” Em 2003, o Ministério da Educação (MEC) já tomava ações para desenvolver a didática de professores da educação básica. Neste mesmo período já se mencionava o uso de simulação em computadores como medida para revitalizar o processo de ensino desta disciplina nas escolas (PEREIRA; AGUIAR, 2006, p. 69).

Aplicações que simulem experimentos reais possuem distinções relevantes quando comparadas ao experimento em sua forma natural, e essa distinções precisam ser analisadas para verificar a validade do modelo computacional. Contudo, mesmo com as diferenças compreende-se que a utilização do modelo é válida, devido as facilitações que o mesmo proporciona (SOUZA FILHO, 2010, p. 28-29).

Em um modelo computacional é possível - e normalmente fácil, alterar variáveis de entrada das simulações. Essa facilidade torna a observação dos fenômenos mais simplificada e imediata, permitindo exibir diversas facetas de uma mesma situação, chamado de realidade-tal-como-é-imaginada (SOUZA FILHO, 2010, p. 29).

Ao estudar o movimento de um corpo em queda vertical com resistência do ar poderia-se utilizar apenas um vídeo para exibição do problema. Porém, ao utilizar um simulador é possível ajustar variáveis de entrada como: velocidade, aceleração e peso, tornando a demonstração mais interativa e dinâmica (SOUZA FILHO, 2010, p. 29-30).

Frente a esse contexto, este trabalho descreve a criação de um *framework* para simulações utilizando um motor de física em 3D. O motor desenvolvido é disponibilizado como biblioteca remota para viabilizar a construção de aplicações de uso didático na disciplina de física.

## 1.1 OBJETIVOS

O objetivo deste trabalho é disponibilizar um *framework* para simulações utilizando um motor de física em 3D.

Os objetivos específicos do trabalho são:

a) migrar e refatorar um motor de física 3D;
b) criar uma biblioteca remota que permita a utilização do motor através da web;
c) implementar uma aplicação *web* de teste para utilizar a biblioteca criada.

## 1.2 ESTRUTURA

O trabalho desenvolvido está organizado em quatro capítulos. O capítulo 2 apresenta a fundamentação teórica, proporcionando embasamento teórico para compreensão do trabalho. O capítulo 3 apresenta o desenvolvimento do Motor de Simulações Físicas, descrevendo as implementações do motor de física, da biblioteca remota e do cliente remoto utilizando diagramas de pacotes, diagramas de classes, diagrama de sequência e diagrama de arquitetura. No capítulo 4 são apresentadas as conclusões e as extensões sugeridas.

# 2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 trata sobre a física e a mecânica clássica. A seção 2.2 e 2.3 tratam dos conceitos relacionados a didática do ensino da disciplina de física na edução básica e simuladores, respectivamente. A seção 2.4 explana os conceitos de biblioteca remota. Por fim, a seção 2.5 descreve os trabalhos correlatos ao trabalho desenvolvido.

## 2.1 MECÂNICA CLÁSSICA

As obras de Isaac Newton foram um marco para a história da ciência por diversos motivos. Dentre eles, pode-se citar o "pensar na natura matematicamente" como um dos mais significativos, pois levou ao que chamados hoje de Física (MENEZES, 2005, p. 18-21).

No início do século XVIII as obras de Newton se restringiam aos estudos do movimento e suas causas, o que se desenrolou na Mecânica Clássica (MENEZES, 2005, p. 18-21). Segundo Barreto (2002, p. 21-26), a mecânica de Newton é um sistema lógico que explica os movimentos dos corpos celestes e dos corpos próximo à superfície terrestre. Ainda de acordo com o autor, as três leis do movimento de Newton, são:

a) primeira lei: todo corpo permanece em seu estado de repouso ou de movimento uniforme em linha reta, a menos que seja forçado a mudar aquele estado por forças aplicadas sobre ele;
b) segunda lei: a mudança de movimento é proporcional à força aplicada, e é produzida na direção de linha reta na qual aquela força é exercida;
c) terceira lei: para toda ação há sempre uma reação oposta de igual intensidade; ou as ações mútuas de dois corpos, um sobre o outro, são sempre iguais e dirigem-se para partes opostas.

Newton ainda descreve que "a gravidade deve ser causada por um agente agindo de acordo com certas leis [...]", e este princípio deu origem a lei da atração gravitacional (BARRETO, 2002, p. 39).

Com as três leis do movimento de Newton e a lei da atração gravitacional forma-se a base da física mecânica clássica (BARRETO, 2002, p. 21).

## 2.2 ENSINO DA FÍSICA

A Física é uma das mais básicas de todas as ciências naturais, abrangendo desde a estrutura elementar da matéria até a evolução do universo. Utilizando os princípios físicos é possível explicar uma grande quantidade de fenômenos do cotidiano e resolver diversos problemas. No currículo escolar brasileiro, a Física é uma disciplina padrão do ensino médio,

que possibilita aos alunos a oportunidade de compreender melhor os fenômenos e a natureza que os rodeia (AGUIAR; GAMA; COSTA, 2005, p. 3).

Contudo, há uma grande dificuldade de conseguir-se professores capacitados para lecionar a disciplina, visto que, segundo Bastos (2010, p. 6), "os cursos de Física exibem números muito baixos de formandos a cada turma que ingressa nas universidades." Sendo assim, a didática aplicada no ensino de Física nas salas de aula do ensino médio, resume-se ao estímulo das soluções algébricas (BASTOS, 2010, p. 6).

Ensinando a Física de maneira sistemática e experimental – e não somente como meio de aplicar fórmulas, o aluno poderá desenvolver a capacidade de observação e análise dos fenômenos cotidianos relacionados a Física. Há uma grande gama de desafios para fazer experimentações nas salas de aula devido ao funcionamento de equipamentos, espaços e material necessário além da precariedade de recursos (PEREIRA; AGUIAR, 2006, p. 2).

A utilização da aplicação tecnológica no processo de ensino-aprendizagem permite criar novos projetos metodológicos que levem a produção do conhecimento (FERREIRA, 2008, p. 10). Com acesso a computadores, os professores poderiam articular entre a teoria e a prática, demonstrando os conceitos explanados através da utilização de simuladores de física. Com o auxílio dos simuladores para realizar experimentações, o professor aproximaria os estudantes de seus cotidianos removendo-os do papel de simples ouvintes (PEREIRA; AGUIAR, 2006, p. 2-8).

## 2.3 SIMULADORES

Segundo Oxford (2014), simular é produzir um modelo computacional a partir da aparência ou o caráter de algo. Representar o comportamento de objetos do mundo real utilizando computadores, é algo quase tão antigo quanto os próprios computadores, e nas duas últimas décadas houve uma explosão na quantidade de pesquisas sobre simulações na área da física (CLINE, 1999, p.1).

As simulações das leis da física têm importantes aplicações em muitos campos. O tipo de fenômeno e o destino da simulação costumam definir a importância da precisão e estabilidade da simulação (SCHROEDER, 2011, p.1). As engenharias aplicadas são um dos extremos do espectro de usuários de simulações, que são comumente utilizadas para realizar previsões do que acontecerá no mundo real. Visto que o objeto de estudo é o mundo real, a linha de tempo das simulações pode ser bem limitada para alcançar a maior precisão possível (SCHROEDER, 2011, p. 1).

Do outro lado do espectro está a indústria do entretenimento e educação, onde as principais necessidades são resultados visivelmente plausíveis. Uma vez que a precisão deixa de ser o principal, é possível alcançar uma linha de tempo maior para a simulação - sendo que a mesma dependerá das técnicas empregadas em sua construção. Nesse tipo de simulação costumam estar presentes elementos como: corpos rígidos, sólidos deformáveis e fluídos, além do tratamento de colisão (SCHROEDER, 2011, p. 1).

Simuladores de física são comumente implementados em motores de jogos (UNITY3D, 2013). Dentre os motores de jogos existentes, pode-se citar o motor para jogos 2D criado por Harbs (2013, seção 2.5.1). Nos motores de jogos, os chamados motores de física se responsabilizam por trazer mais realismo ao comportamento dos elementos do jogo (UNITY3D, 2013).

## 2.4 BIBLIOTECA REMOTA

Bibliotecas são coleções de rotinas. Essas rotinas são unidades de programação para executar tarefas como a exibição de informações na tela ou até mesmo calcular a raiz quadrada de um número. O produto final da união entre biblioteca e as rotinas de um programa é chamado de código executável (PRATA, 2012, p. 18).

A ideia de biblioteca remota não é complexa ou extensa. É uma extensão do funcionamento padrão, onde são transferidos o controle e os dados através de uma comunicação de rede. Quando um procedimento remoto é invocado, o ambiente de chamada é suspenso e os parâmetros são transmitidos através da rede para o ambiente onde o procedimento é executado. Ao término da execução, os resultados são transmitidos para o ambiente de chamada, onde a execução continua como se uma simples chamada de procedimento tivesse sido executada (BIRRELL; NELSON, 1984, p.1).

Dentre as tecnologias que viabizam a construção de uma biblioteca remota, pode-se citar *WebSocket*. Sua eficiência se aplica em reduzir o tráfego e latência de dados na rede. Com *WebSocket* é definido um canal de comunicação *full-duplex,* que através da web opera por meio de uma única *socket*. Representa um grande avanço especialmente para aplicações *web* de tempo real, dado que não é apenas mais um aprimoramento gradual para comunicações convencionais sob *Hypertext Transfer Protocol* (HTTP) e sim uma transformação do modo de comunicação de rede tradicional (LUBBERS, 2011, p.1).

## 2.5 TRABALHOS CORRELATOS

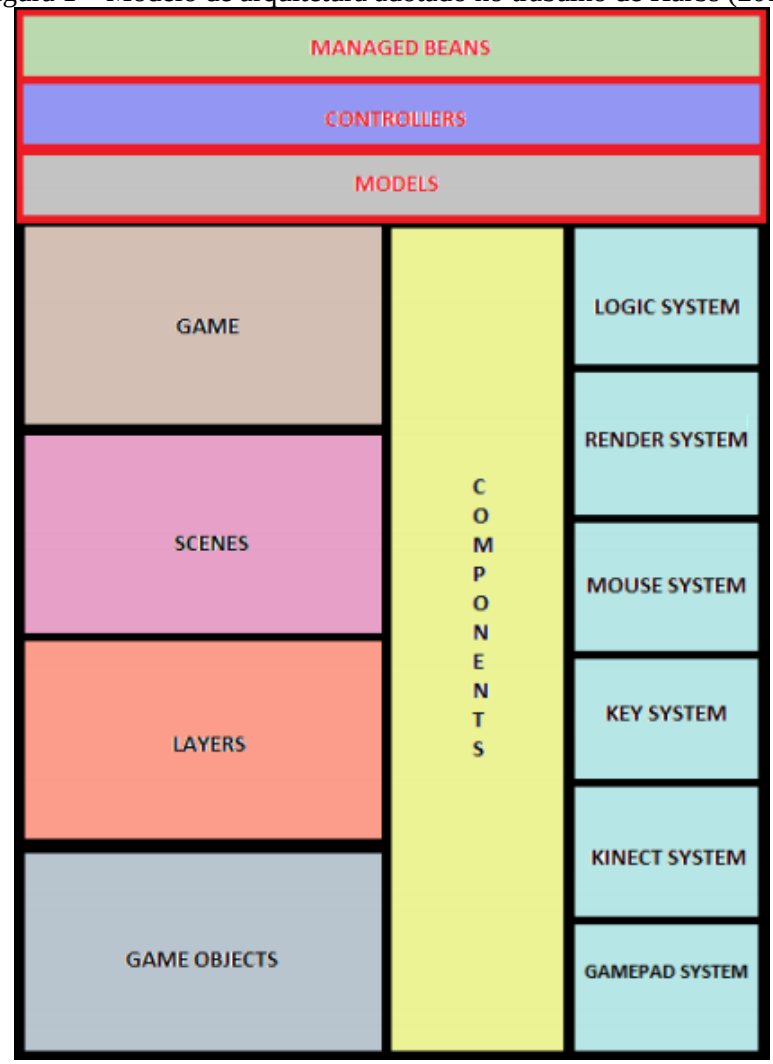
A seguir são apresentados os trabalhos e as ferramentas que utilizam motores de física. São eles: Harbs (2013), Silva (2012) e Algodoo (2013).

### 2.5.1 Motor para jogos 2D utilizando HTML5

O trabalho desenvolvido por Harbs (2013), resultou em um motor de cenas 2D utilizando HTML5 e uma ferramenta para edição de jogos. O motor de cenas 2D foi construído sob uma arquitetura baseada em componentes. O motor originalmente dispõe desde funcionalidades para eventos de controladores como *mouse* e teclado, além da capacidade de simular propriedades físicas.

A arquitetura baseada em componentes permite que novas funcionalidades, elementos gráficos ou comportamentos sejam adicionados ao motor de maneira organizada e sem necessidade de reestruturação. A Figura 1 apresenta o modelo de arquitetura adotado no trabalho de Harbs (2013).

Figura 1 – Modelo de arquitetura adotado no trabalho de Harbs (2013)




Fonte: Harbs (2014, p. 62).

Dentre os componentes construídos podem ser mencionados simuladores de corpos rígidos e colisões. Estes utilizam a biblioteca BOX2D JS (2008) que é um motor de física construído em JavaScript e baseia-se nas leis da mecânica clássica.

Na ferramenta de edição deste motor para jogos é possível editar cenas novas ou anteriomente criadas. Além disto, permite gerenciar *assets*[1] e criar novos componentes que serão facilmente integrados ao ciclo de vida do motor. Segundo Harbs (2013), a ferramenta de edição permite que jogos em duas dimensões sejam desenvolvidos com uma curva de aprendizagem pequena.

### 2.5.2 Protótipo de uma ferramenta gráfica para criação de simulações na área da física em dispositivos móveis

Silva (2012, p. 7) construiu um motor de simulações físicas baseado nas leis da mecânica clássica newtoniana (seção 2.3) e um aplicativo para experimentação das simulações. As funcionalidades do motor consistem em cálculo das informações físicas de um objeto e tratamento de colisão, enquanto no aplicativo é possível criar novas cenas, manipular os objetos criados e reproduzir simulações.

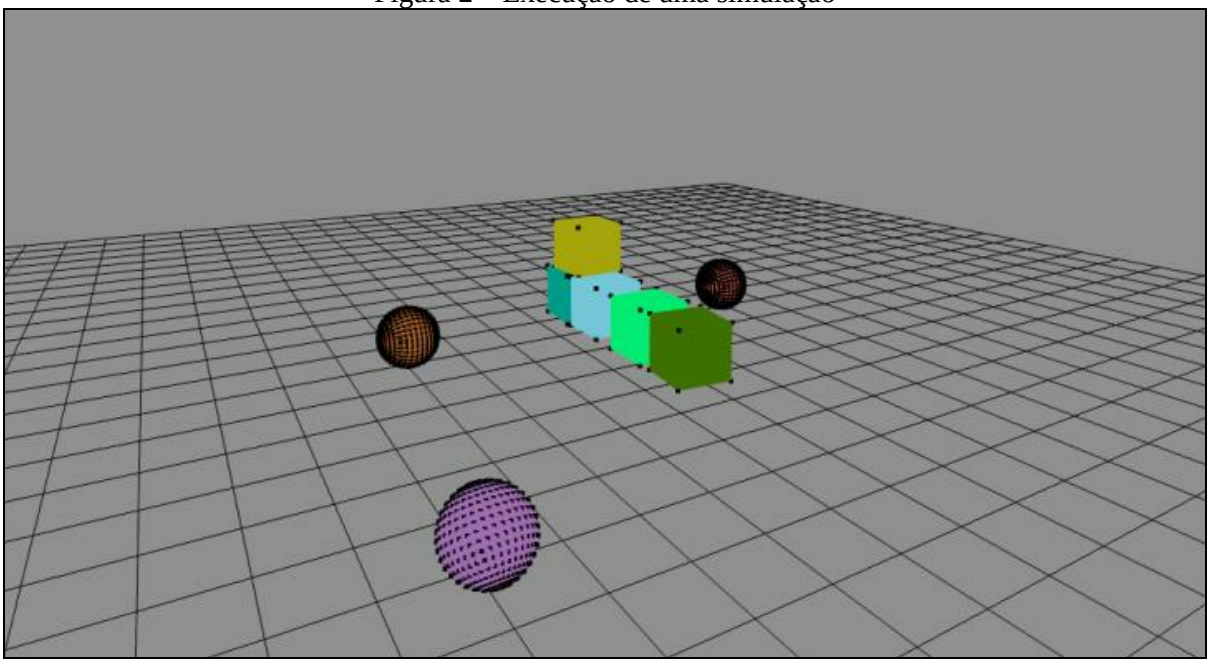
As informações físicas calculadas são baseadas na força aplicada sobre os objetos. Dentre essas informações calculadas pode-se citar a gravidade, aceleração linear e angular, velocidade linear e angular e as forças atuantes (SILVA, 2012, p. 46).

Dentro do tratamento de colisões Silva (2012, p.43) utilizou uma técnica chamada *Coarse Collision*, que através do uso de uma árvore do tipo *octree* tenta identificar possíveis colisões. Uma vez que uma possível colisão seja identificada, são resolvidos os conflitos e realizados os devidos cálculos físicos.

A aplicação que utiliza o motor desenvolvido por Silva (2012, p. 26) executa somente em dispositivos móveis da Apple. Através das funcionalidades da aplicação é possível realizar a experimentação das funcionalidades do motor. Criar novas simulações, criar objetos e alterar suas propriedades, são algumas das funcionalidades disponíveis. Além destas, é possível reproduzir, editar, salvar e carregar as simulações criadas. A Figura 2 apresenta a execução de uma simulação.

[1] *Assets* são ativos digitais. Ativo digital é qualquer item de texto ou arquivo de mídia enriquecido que foi formatado dentro de um código binário que leva embutido seu direito de uso.

Figura 2 – Execução de uma simulação



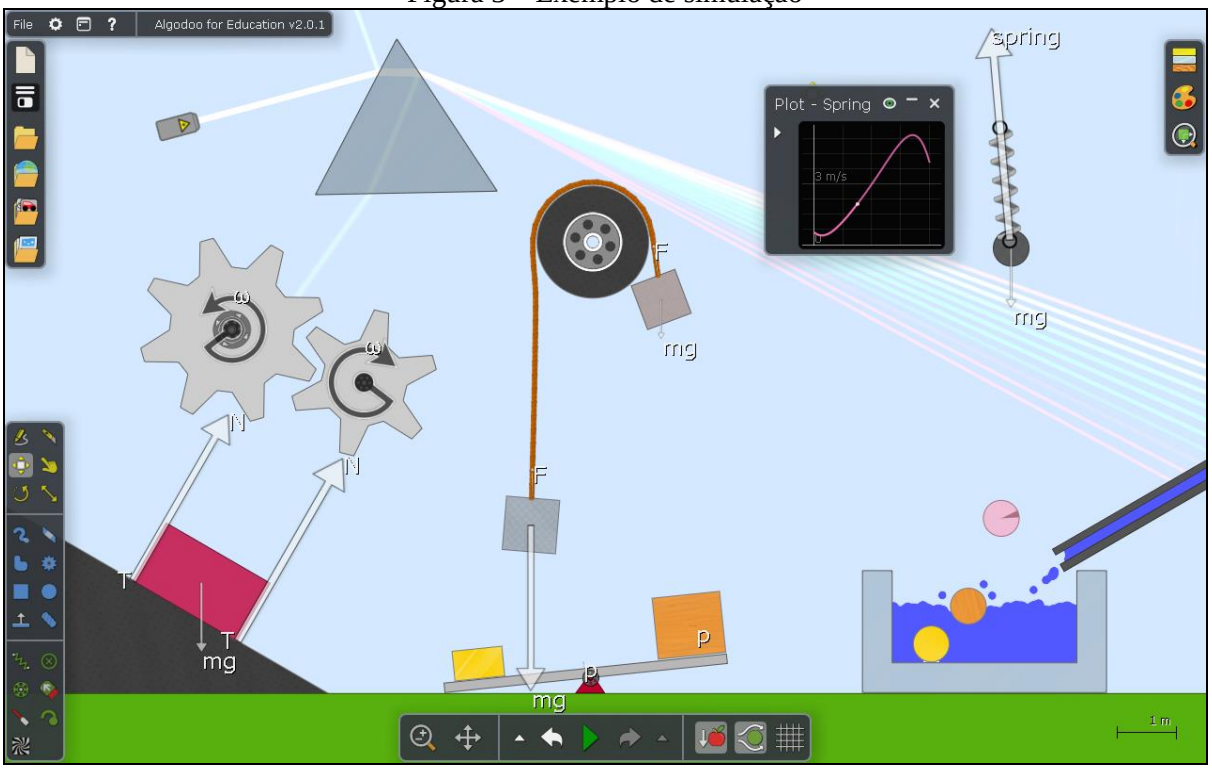
Fonte: Silva (2012, p. 54).

### 2.5.3 Algodoo

Criada pela empresa Algoryx, Algodoo possui uma identidade visual cartunizada e simples para que seja possível explorar a física com maior amigabilidade ao usuário (ALGODOO, 2013). Com Algodoo é possível criar cenas de duas dimensões, que podem ser compostas de formas geométricas, engrenagens, cordas e fluídos. Também é possível realizar diversas alterações nos objetos e dentre elas pode-se citar: girar, dimensionar e mover (ALGODOO, 2013).

Além de construir e reproduzir cenas, a ferramenta possibilita adicionar elementos de física ao cenário. Algumas das funcionalidades dispostas são: realizar simulações com fluídos, molas e motores. Ainda é possível explorar diversos parâmetros como gravidade, atrito, refrassão e atração (ALGODOO, 2013), conforme pode ser observado na Figura 3.

Figura 3 – Exemplo de simulação




Fonte: Algodoo (2013).

Junto do Algodoo, a empresa Algoryx disponibiliza o Algobox, que é uma biblioteca de cenas que podem ser reproduzidas através da ferramenta. Por meio do Algobox é possível salvar e compartilhar as criações, bem como utilizar cenas criadas por outros usuários no Algodoo (2013).

# 3 DESENVOLVIMENTO DO SISTEMA

Nesse capítulo são abordadas as etapas do desenvolvimento do Motor de Física (MF), da Biblioteca Remota (BR) e do Cliente Remoto (CR), assim como as simulações de testes e de demonstração. São abordados os principais requisitos, a especificação, a implementação e os resultados e discussões.

## 3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O motor de física deverá:

a) criar corpo rígido e permitir configurar (Requisito Funcional – RF):
   - aceleração,
   - velocidade,
   - rotação,
   - massa;
b) realizar os cálculos de integração dos corpos seguindo as leis da mecânica clássica (RF);
c) ser desenvolvido de forma que permita auto reuso e extensão (Requisito Não Funcional – RNF);
d) utilizar a linguagem de programação Java (RNF).

A biblioteca remota deverá:

a) utilizar o motor de física desenvolvido (RF);
b) permitir que as funcionalidade do motor de física sejam acessadas remotamente (RF);
c) utilizar *WebSocket* para disponibilizar acesso às funcionalidades do motor (RNF);
d) permitir conexão à partir de qualquer plataforma *WebSocket* (RNF);
e) utilizar a linguagem de programação Java (RNF).

A aplicação de testes deverá:

a) utilizar JavaScript para se comunicar com a biblioteca remota (RNF);
b) demonstrar os conceitos da mecânica clássica disponibilizados no motor através da biblioteca remota (RNF);
c) desenhar a simulação utilizando HTML5 e JavaScript (RNF).

## 3.2 ESPECIFICAÇÃO

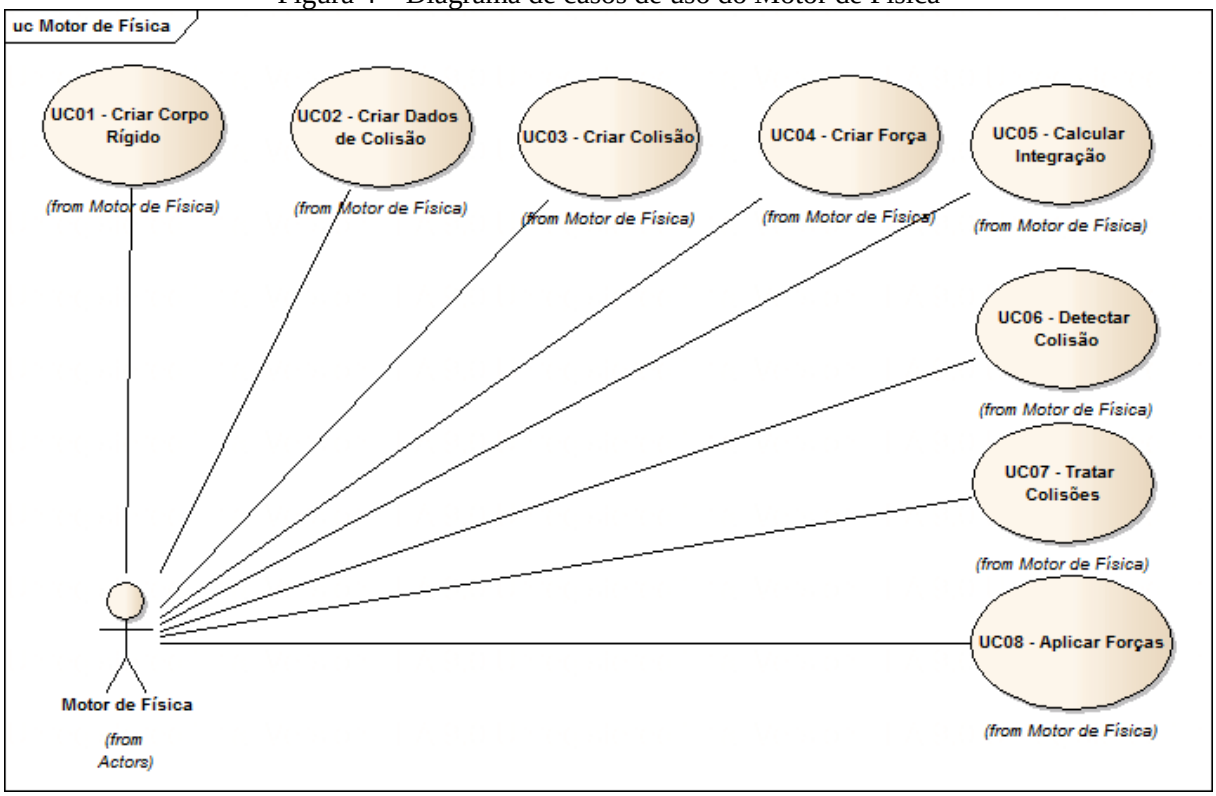
Para realizar a especificação do desenvolvimento do trabalho, foram criados diagramas de caso de uso, de pacote, de classe e de sequência, com a ferramenta *Enterprise Architect*

(EA) e utilizado diagramas da *Unified Modeling Language* (UML). Nas próximas seções é apresentado o desenvolvimento do MF, da BR, do CR, além de um exemplo de aplicação desenvolvida neste trabalho. O MF é uma biblioteca com o propósito de simplificar o desenvolvimento de aplicações que simulem o comportamento físico no mundo real. A BR visa permitir que as rotinas do MF sejam utilizadas remotamente e o CR se comunica com a mesma viabilizando o uso do MF. A seguir será apresentada a especificação dessas três partes que compõem este trabalho.

### 3.2.1 Casos de uso do Motor de Física

Nesta seção são descritos os casos de uso das funcionalidades disponibilizadas pelo MF, ilustrados pela Figura 4. O detalhamento de cada caso de uso pode ser visto no Apêndice A.

Figura 4 – Diagrama de casos de uso do Motor de Física

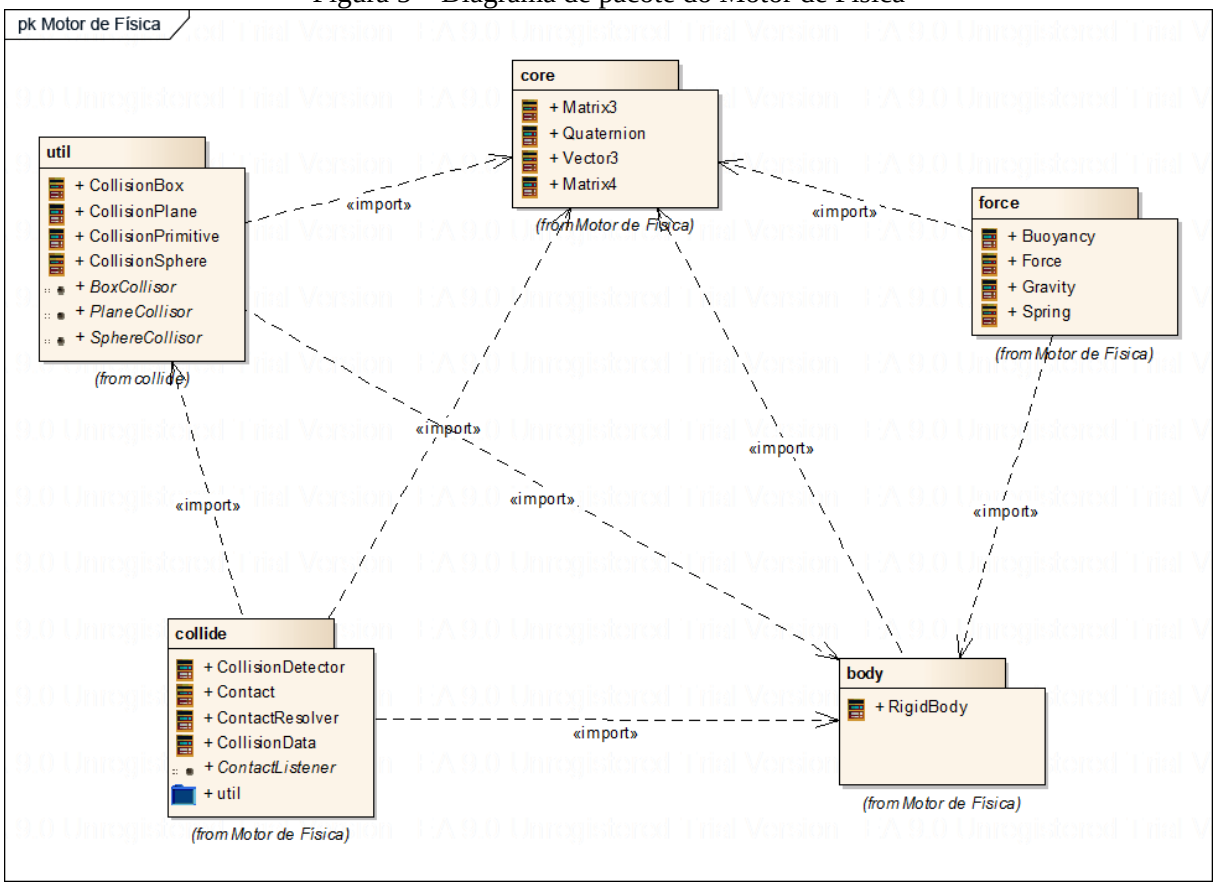



Os casos de uso do Motor de Física apresentam as funcionalidades que permitem a realização de uma simulação. Pode-se criar corpos rígidos, dados de colisão, colisões e forças. Ainda é possível realizar os cálculos de integração, detecção e resolução de colisões e a aplicação das forças criadas.

### 3.2.2 Diagrama de classe do Motor de Física

Nesta seção são descritas as classes e estruturas que constituem o MF e na Figura 5 são exibidas as dependências entre os pacotes do mesmo. As classes destes pacotes serão detalhadas a seguir.

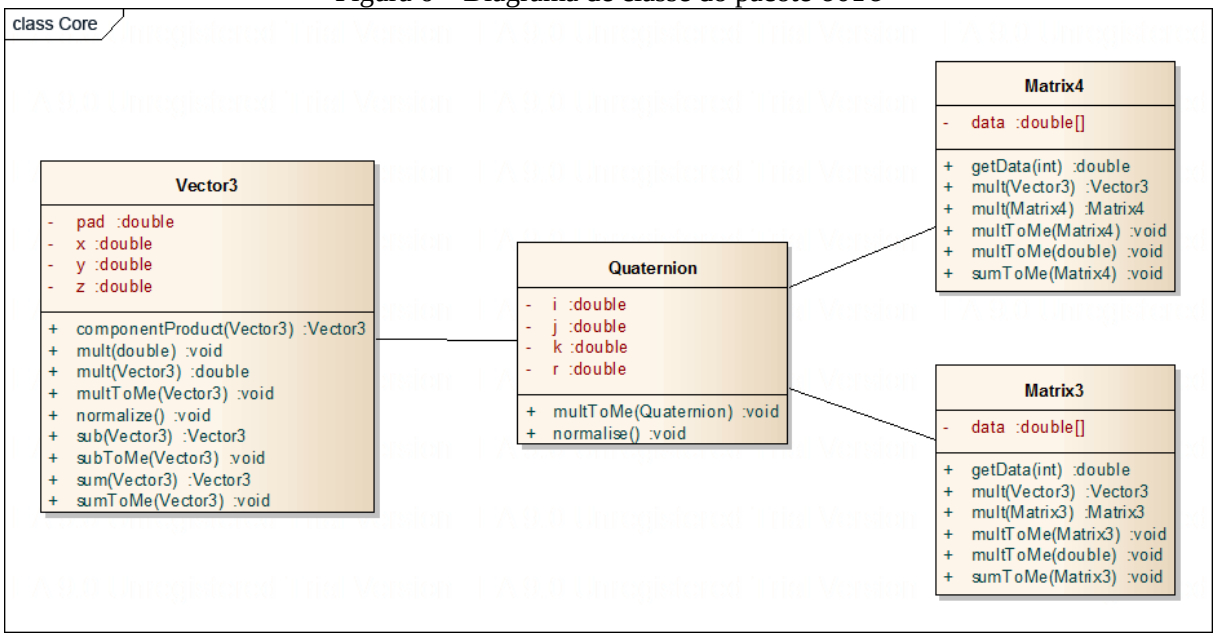
Figura 5 – Diagrama de pacote do Motor de Física




#### 3.2.2.1 Pacote `core`

O pacote denominado `core` possui as classes que representam as unidades de medidas necessárias para definir propriedades, comportamentos e realizar cálculos dentro do motor de física, conforme pode ser observado na Figura 6.

Figura 6 – Diagrama de classe do pacote `core`

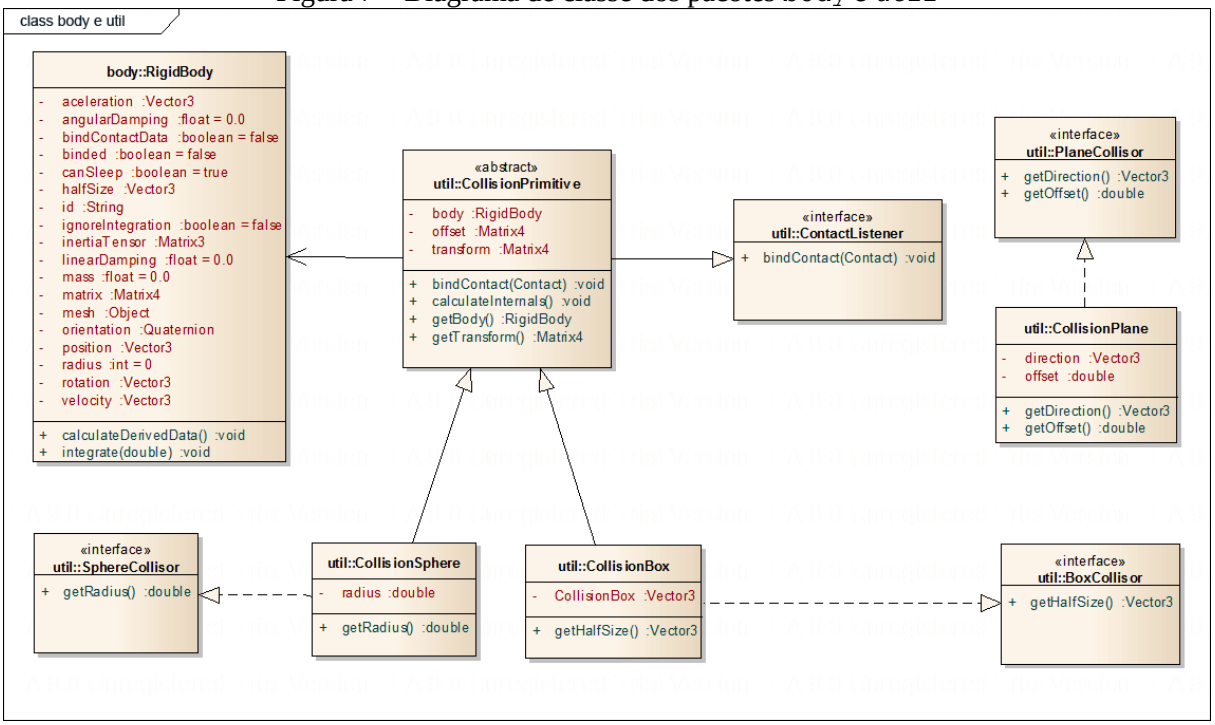



A classe `Vector3` representa um vetor tridimensional para a Física, utilizando esta classe pode-se definir vetores de força, velocidade, aceleração e assim por diante. A classe `Matrix3` define os dados e operabilidade de uma matrix 3x3, e é utilizada para a definição de um tensor de inercia. A classe `Matrix4` representa uma matriz de transformação, que consiste na matriz de rotação e posição de um elemento.

#### 3.2.2.2 Pacotes `body` e `util`

O pacote denominado `body` possui a classe que representa um corpo rígido e o pacote `util` possui classes que especializam a definição de um corpo rígido, conforme Figura 7.

Figura 7 – Diagrama de classe dos pacotes `body` e `util`

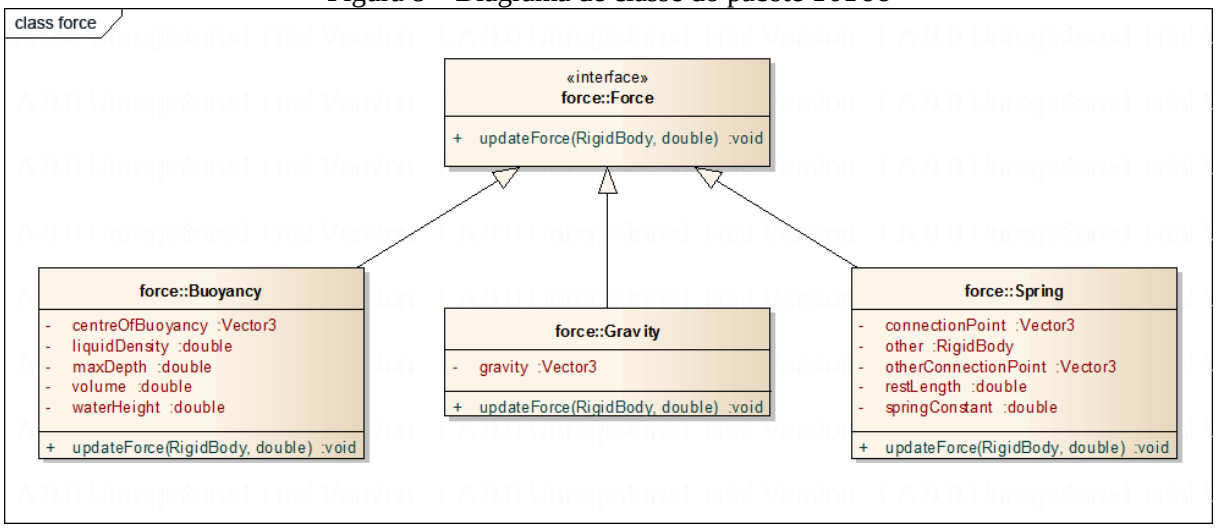



A classe `RigidBody` define um corpo rígido para o Motor de Física. Ela possui os atributos e métodos comuns a todos os corpos rígidos. A classe `CollisionPrimitive` especializa a definição de um corpo rígido para corpo rígido e colidível. As classes `CollisionSphere` e `CollisionBox` definem a implementação concretas para corpos do tipo esfera e caixa. As classes `SphereCollisor` e `BoxCollisor` delimitam acesso as propriedades necessárias para identificar colisões de esferas e caixas, respectivamente. A classe `CollisionPlane` é a implementação concreta da classe `PlaneCollisor`, que delimita acesso às propriedades necessárias para identificar colisão corpos e planos. A interface `ContactListener` define a ação a ser invocada quando uma colisão é detectada.

### 3.2.2.3 Pacote `force`

O pacote denominado `force` possui a interface que define a representação de uma força que pode ser aplicada sobre corpos rígidos, assim como algumas forças disponibilizadas previamente, conforme pode ser observado na Figura 8.

Figura 8 – Diagrama de classe do pacote `force`

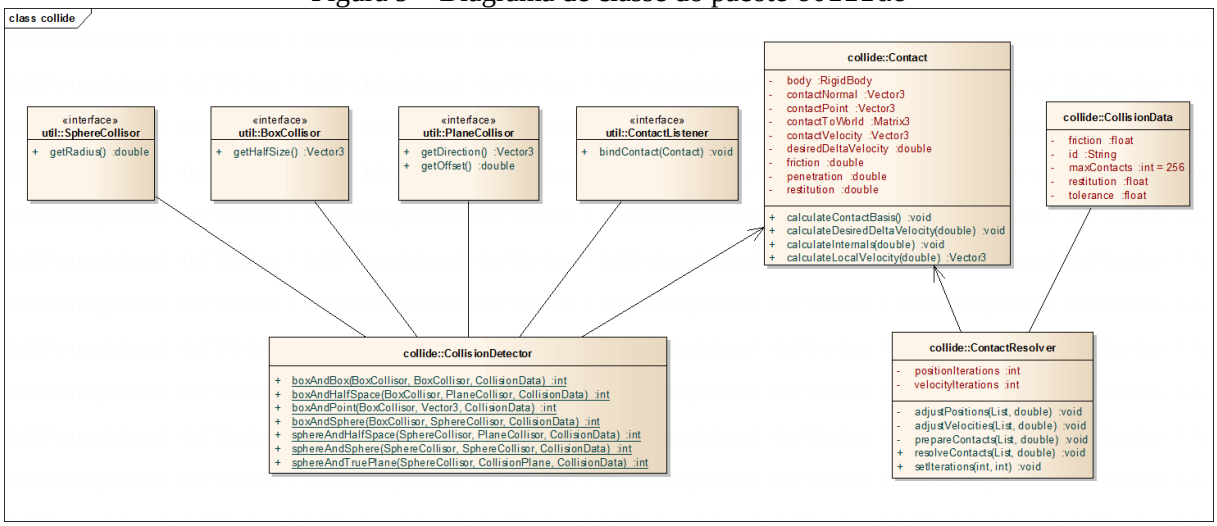



A interface `Force` define a representação de uma força que pode ser aplicada sobre corpos rígidos. A classe `Gravity` é a implementação concreta para a força da gravidade, onde é possível definir o vetor de aceleração. A classe `Spring` define a aplicação de força de mola, já a classe `Buoyancy` define a força de flutuação, e ambas podem ser aplicadas sobre corpos rígidos.

### 3.2.2.4 Pacote `collide`

O pacote denominado `collide` possui as classes necessárias para detectar e tratar colisões envolvendo corpos rígido, conforme pode ser observado na Figura 9.

Figura 9 – Diagrama de classe do pacote `collide`



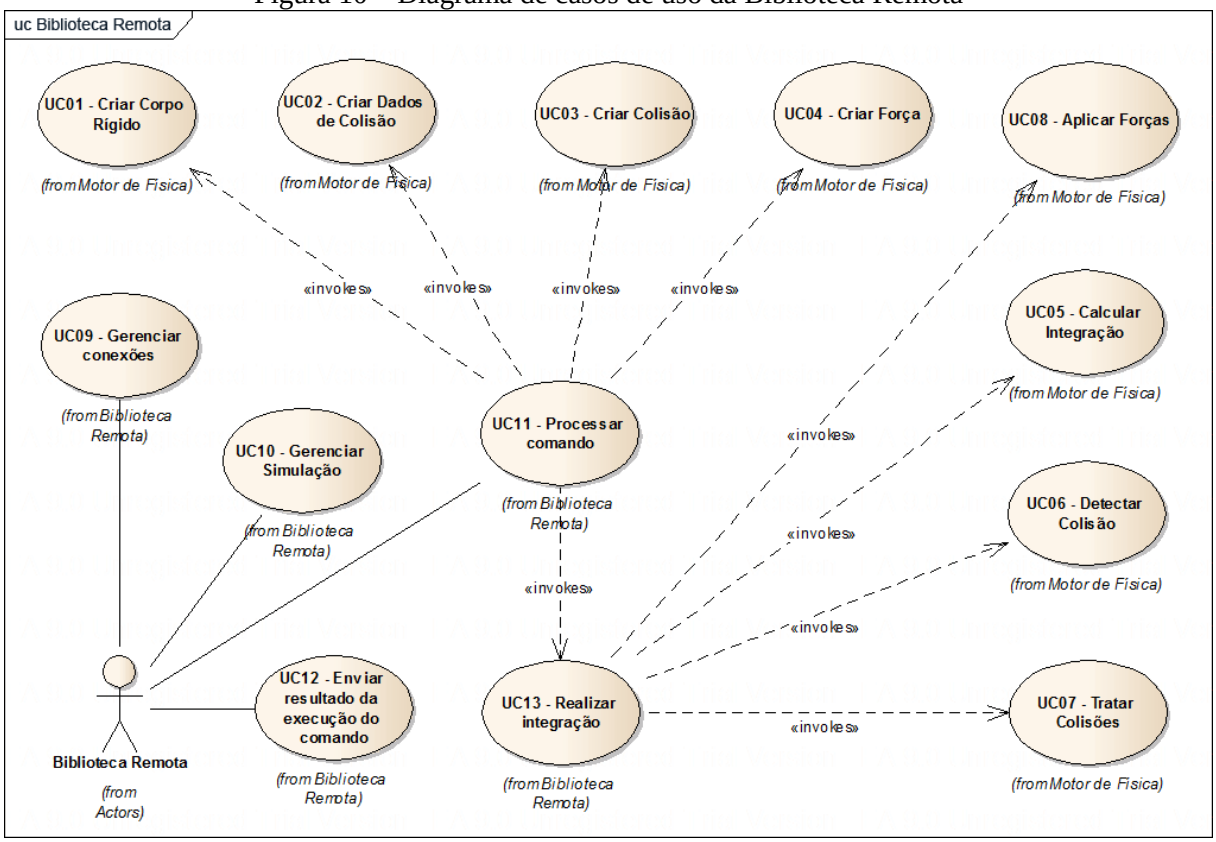

A classe `CollisionData` define as propriedades de colisão entre dois materiais. A classe `Contact` define as propriedades pertinentes à colisão entre de dois corpos, e as funcionalidades dispostas para auxiliar os cálculos de resolução do contato definido. A classe `CollisionDetector` utiliza as interfaces `SphereCollisor`, `BoxCollisor` e `PlaneCollisor`

para identificar contato entre estes tipos de corpos, e a interfaces `ContactListener` para notificar as colisões encontradas. A classe `ContactResolver` é responsável por resolver matematicamente os contatos encontrados, aplicando sobre os corpos envolvidos as respectivas tratativas.

### 3.2.3 Casos de uso da Biblioteca Remota

Nesta seção são descritos os casos de uso das funcionalidades disponibilizadas pela BR, ilustrados pela Figura 10. A seguir estão detalhados os casos de uso 11 e 13. Os demais podem ser vistos no Apêndice B.

Figura 10 – Diagrama de casos de uso da Biblioteca Remota




No Quadro 1 é apresentado o detalhamento do décimo primeiro caso de uso operado pelo ator `Biblioteca Remota`.

Quadro 1 – Caso de uso UC11: Processar comando

| UC11 - Processar comando | |
|---|---|
| Descrição | Permite que a `Biblioteca Remota` realize o processamento de comando recebidos e proporcione a integração com uma simulação remotamente. |
| Cenário Principal | 1. `Biblioteca Remota` recebe mensagem.<br>2. `Biblioteca Remota` tranforma mensagem em comando.<br>3. `Biblioteca Remota` despacha o comando para ser processado pelo Motor de Física.<br>4. `Biblioteca Remota` envia resultado do processamento para a origem da mensagem. |
| Pós-condição | O estado da simulação é alterado e a `Biblioteca Remota` notificou a origem da mensagem com o novo estado da simulação. |

No Quadro 2 é apresentado o detalhamento do 13º caso de uso operado pelo ator Usuário do Motor em virtude do módulo desenvolvido.

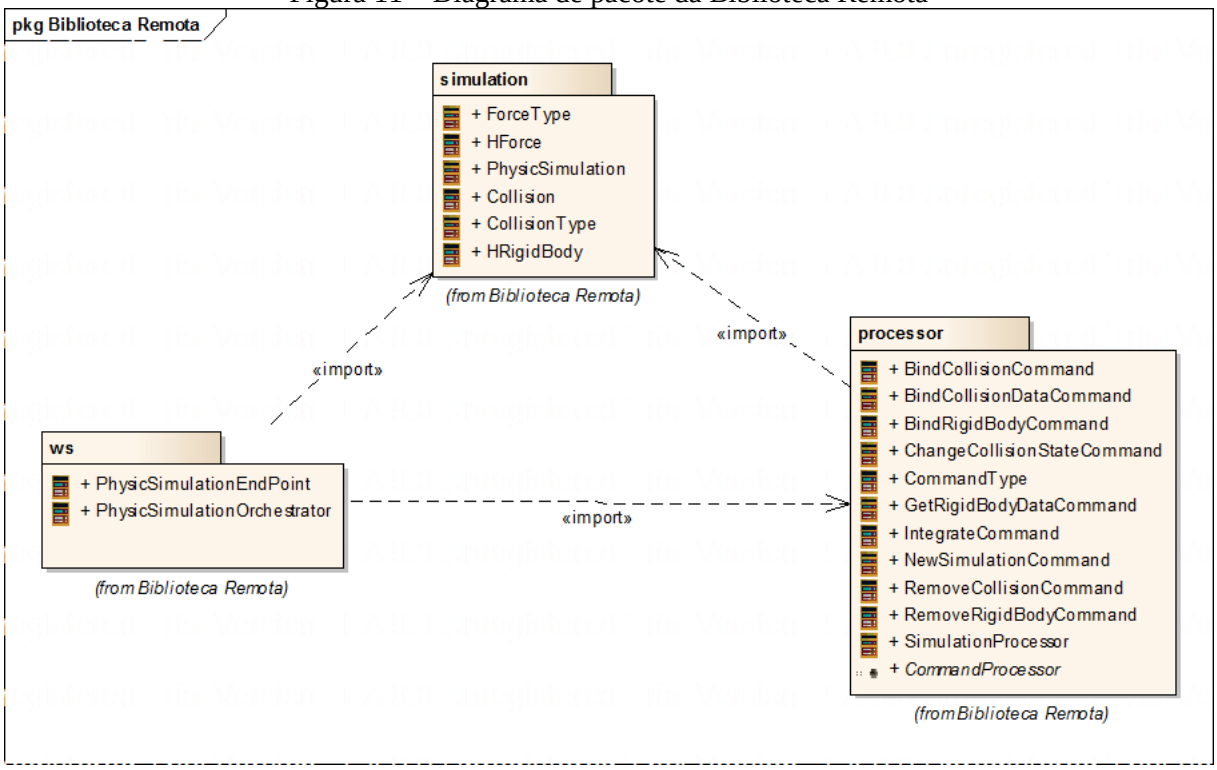
Quadro 2 – Caso de uso UC13: Realizar integração

| UC13 – Realizar integração | |
|---|---|
| Descrição | Permite que a `Biblioteca Remota` realize o processo de integração da simulação e atualize o status de todos os elementos que a compoem. |
| Cenário Principal | 1. `Biblioteca Remota` recebe comandos de integração e invoca ação de integração da simulação.<br>2. Simulação aplica as forças atuantes sobre os corpos rígidos respectivamente.<br>3. Simulação solicita que cada corpo rígido realize seus cálculos de integração e atualize as propriedades envolvidas.<br>4. Motor de Física detecta as colisões entre corpos rígidos.<br>5. Motor de Física trata os contatos encontrados atualizando propriedades dos corpos envolvidos.<br>6. `Biblioteca Remota` obtém lista de novas posições e matrizes de tranformação e envia para origem do comando. |
| Pós-condição | O estado dos elementos da simulação é alterado e a `Biblioteca Remota` notificou a origem da mensagem com o novo estado dos elementos da simulação. |

### 3.2.4 Diagrama de classe da biblioteca remota

Nesta seção são descritas as classes e estruturas que constituem a BR e na Figura 11 são exibidas as dependências entre os pacotes da mesma. As classes destes pacotes são detalhadas a seguir.

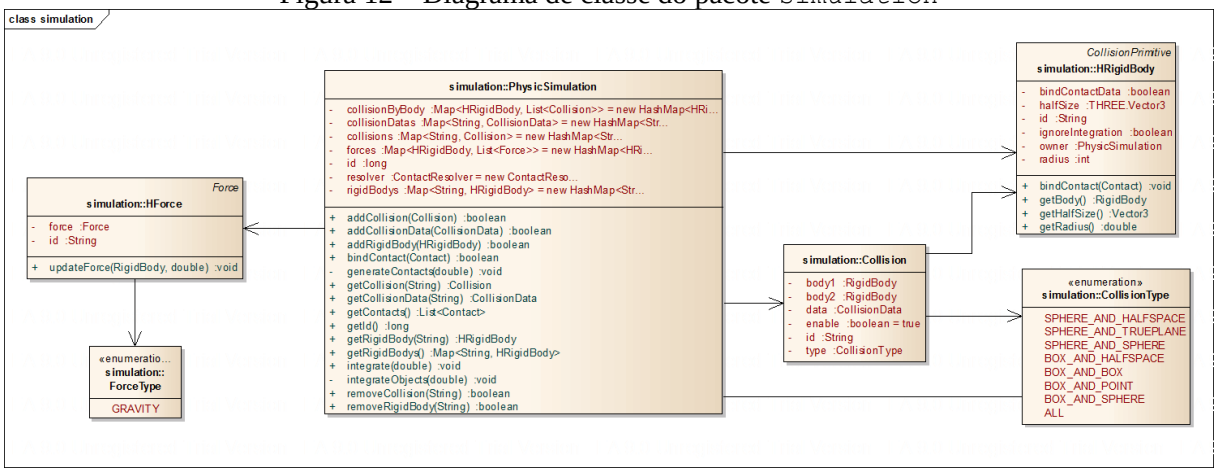
Figura 11 – Diagrama de pacote da Biblioteca Remota




### 3.2.4.1 Pacote `simulation`

O pacote denominado `simulation` possui as classes necessárias para definir e executar uma simulação, composta por corpos rígidos, forças, dados de colisões e colisões, conforme pode ser observado na Figura 12.

Figura 12 – Diagrama de classe do pacote `simulation`



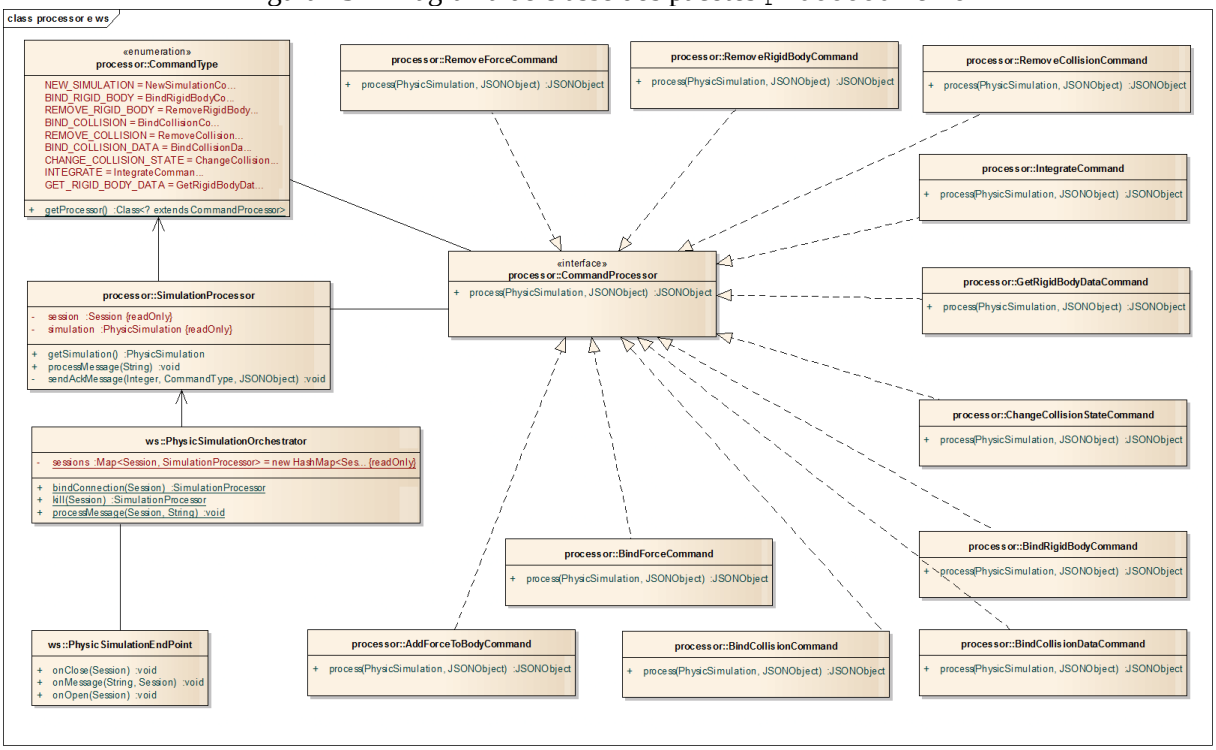

As enumerações `CollisionType` e `ForceType` definem respectivamente os tipos de força e colisões que podem ser empregados em uma simulação. A classe `HForce` representa uma força para a simulação e encapsula uma definição concisa de força para o MF. A classe `HRigidBody` representa um corpo rígido genérico para a simulação, especializando todos os tipos de corpo pré-disponibilizados pelo MF e encapsulando sua implementação concreta,

além de suas propriedades. A classes `Collision` representa uma possível colisão para a simulação, que em cada ciclo de integração deve ser detectada e posteriormente tratada. Por fim `PhysicSimulation` representa a simulação para a BR, encapsulando e gerenciando todos os elementos e rotinas pertinentes ao funcionamento de uma simulação.

#### 3.2.4.2 Pacotes `processor` e `ws`

O pacote denominado `processor` possui a interface que representa um comando, suas definições concretas e classes para que o processamento do comando seja realizado. O pacote `ws` possui as classes responsáveis por estabelecer e manter as conexões *WebSocket*, sendo o ponto inicial para viabilizar o uso do motor remotamente, conforme pode ser observado na Figura 13.

Figura 13 – Diagrama de classe dos pacotes `processor` e `ws`




A interface `CommandProcessor` do pacote `processor`, define um processador de comando, suas especializações são:

a) `BindRigidBodyCommand`: transforma os dados em um novo corpo rígido para a simulação;

b) `RemoveRigidBodyCommand`: descarta um corpo rígido da simulação;

c) `BindForceCommand`: transforma os dados recebidos em uma nova força para a simulação;

d) `AddForceToBodyCommand`: relaciona um corpo rígido à uma força;

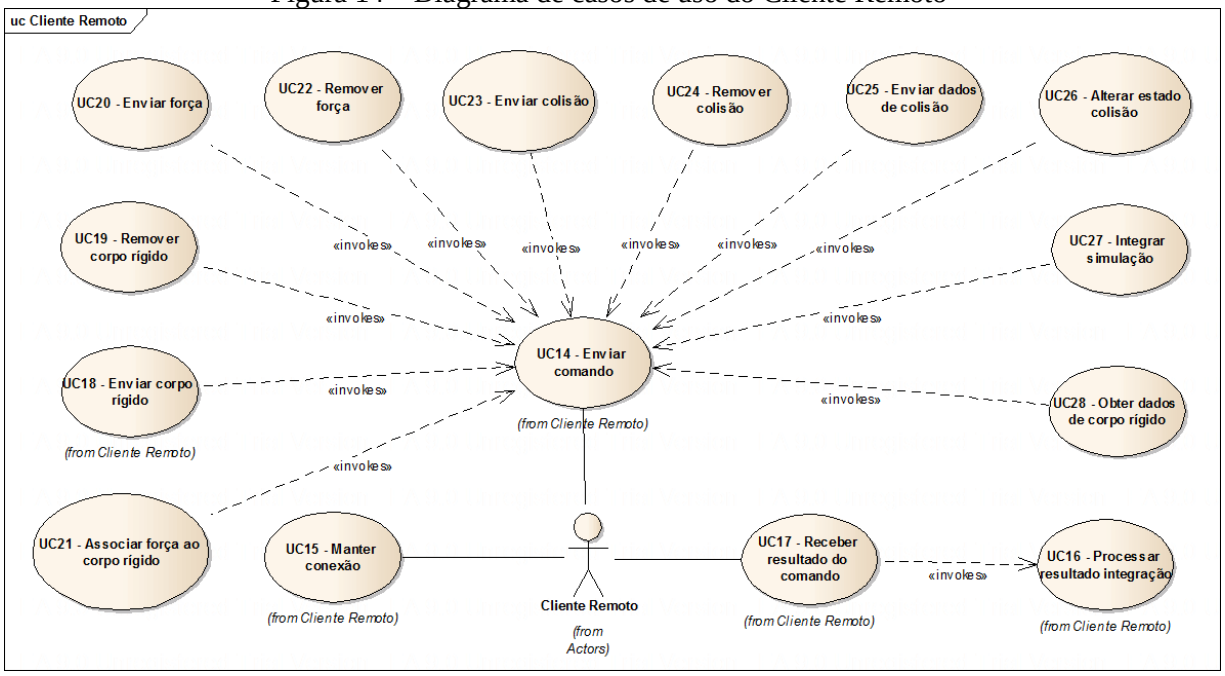
e) `RemoveForceCommand`: descarta uma força da simulação;
f) `BindCollisionCommand`: transforma os dados recebidos em uma nova colisão para a simulação;
g) `RemoveCollisionCommand`: descarta uma colisão da simulação;
h) `BindCollisionDataCommand`: transforma os dados recebidos em dados de colisão para a simulação;
i) `ChangeCollisionStateCommand`: habilita ou desabilita uma possível colisão;
j) `IntegrateCommand`: realiza o processo de integração da simulação atualizando o estados de cada elemento que a compõe e envia as novas informações para a origem do comando;
k) `GetRigidBodyDataCommand`: obtém os valores de cada propriedade do corpo e envia para a origem do comando.

A enumeração `CommandType` associa as especializações de `CommandProcessor` aos seus respectivos identificadores de comando. A classe `SimulationProcessor` é responsável pelo despacho de cada comando para seu respectivo processador. A classe `PhysicsSimulationOrchestrator` é responsável por gerenciar o ciclo de vida de cada associação e mantê-la associada à uma sessão *WebSocket* conectada a BR. A classe `PhysicSimulationEndPoint` é responsável pela implementação do *WebSocket*, mantendo o ponto de comunicação da BR com o Cliente Remoto.

### 3.2.5 Casos de uso do Cliente Remoto

Nesta seção são descritos os casos de uso das funcionalidades disponibilizadas pelo CR, ilustrados pela Figura 14. O detalhamento de cada caso de uso pode ser visto no Apêndice C.

Figura 14 – Diagrama de casos de uso do Cliente Remoto

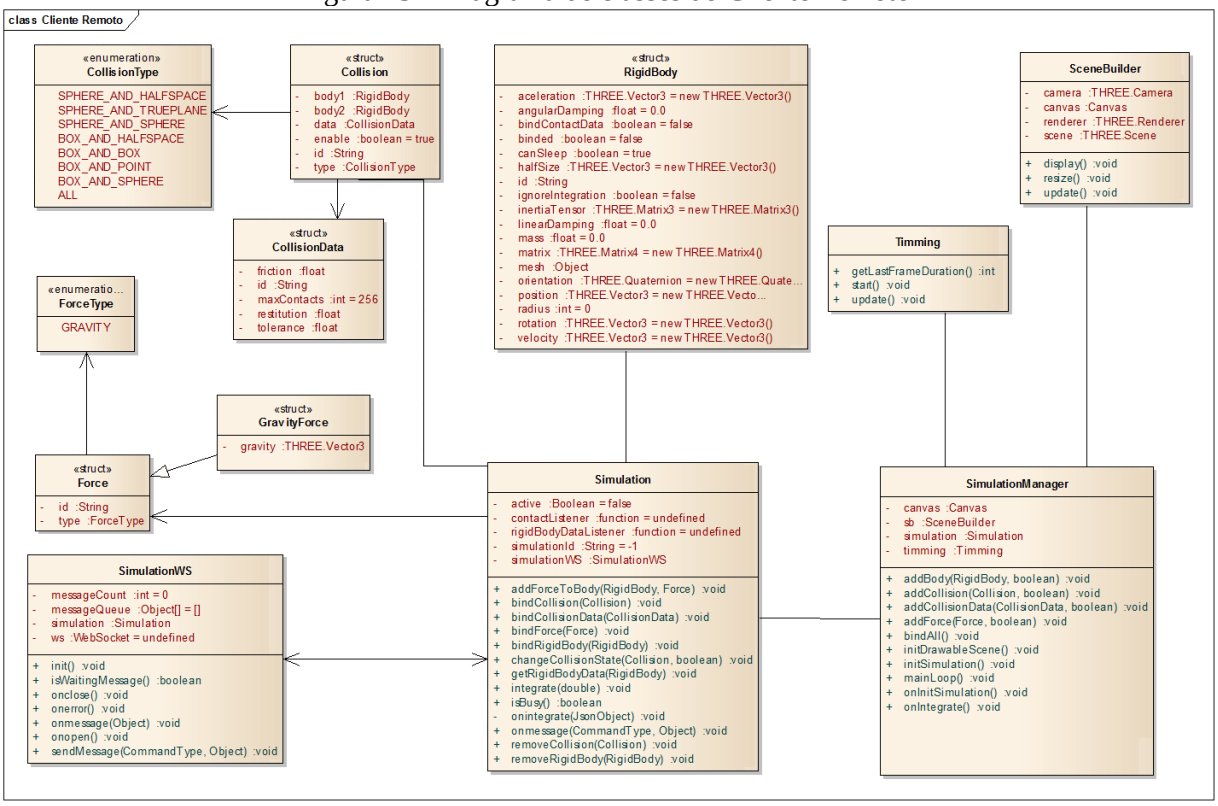



Os casos de uso do Cliente Remoto apresentam as funcionalidades para executar uma simulação remotamente. Entre os casos uso há o de controle da conexão e o envio e recebimento de comandos. Os comandos de preparação da simulação, são os de envio de corpo rígido, dados de colisão, colisões e força. Já os de alteração são para o estado de colisão e associação de força ao corpo rígido. Os de remoção são para corpo rígido, colisão ou forças. Ainda há comandos para obter dados de corpo rígido e realizar a integração da simulação.

### 3.2.6 Diagrama de classes do Cliente Remoto

Nesta seção são descritas as classes e estruturas que constituem o CR e na Figura 15 são exibidas as dependências entre as classes do mesmo, sendo as mesmas detalhadas a seguir.

Figura 15 – Diagrama de classes do Cliente Remoto




A classe `RigidBody` encapsula todas as propriedades configuráveis de um corpo rígido em uma simulação com a BR e armazena a propriedade `mesh`, que é a referência para o objeto gráfico do ThreeJS correspondente a este corpo rígido. As enumerações `CollistionType` e `ForceType` definem os tipos de possíveis de colisão e força respectivamente. A classe `CollisionData` define as propriedades de colisão entre dois materiais. A classe `Collision` define uma possível colisão entre dois corpos rígidos. A classe `Force` define a estrutura base para definição de forças e `GravityForce` por sua vez é a especialização de força da gravidade. A classe `Simulation` encapsula todos os elementos presentes na simulação, os gerencia e monta os comandos a serem enviados para a BR através da classe `SimulationWS`, além de realizar as integrações com o ThreeJS. `SimulationWS` por sua vez mantem a conexão com o *WebSocket*, envia os comandos solicitados e despacha os comandos recebidos para a `Simulation`. A classe `Timming` é responsável por guardar o tempo de intervalo entre *frames*, que será utilizado para as rotinas de integração da simulação. A classe `SceneBuilder` realiza a integração do canvas do HTML5 com o ThreeJS e a simulação. A classe `SimulationManager` mantém todos os elementos necessários para definição e operacionalidade de uma simulação em uma cena gráfica.

### 3.2.7 Diagrama de sequência

Nessa seção serão apresentados os diagramas de sequência que descrevem o *pipeline* de execução do comando de integração. Sendo que o primeiro diagrama descreve como o comando é originado no Cliente Remoto (CR) e enviado à Biblioteca Remota (BR), já o segundo diagrama descreve como a BR utiliza o Motor de Física (MF) para realizar o processamento. Os comandos e sua execução não estão associados a qualquer *loop* de execução ou estado da aplicação, havendo conexão ativa entre o Cliente e a Biblioteca Remota é possível que o usuário do CR realize o envio de comandos. O primeiro e o segundo diagrama são apresentados nas Figuras 16 e 17, respectivamente.

Figura 16 – Sequência do comando de integração (Cliente Remoto e Biblioteca Remota)

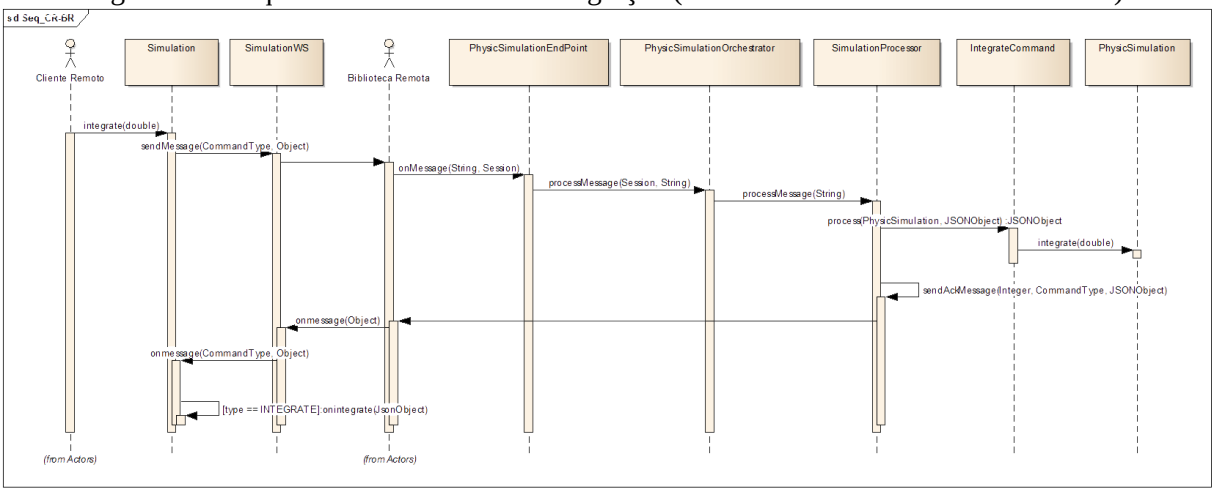
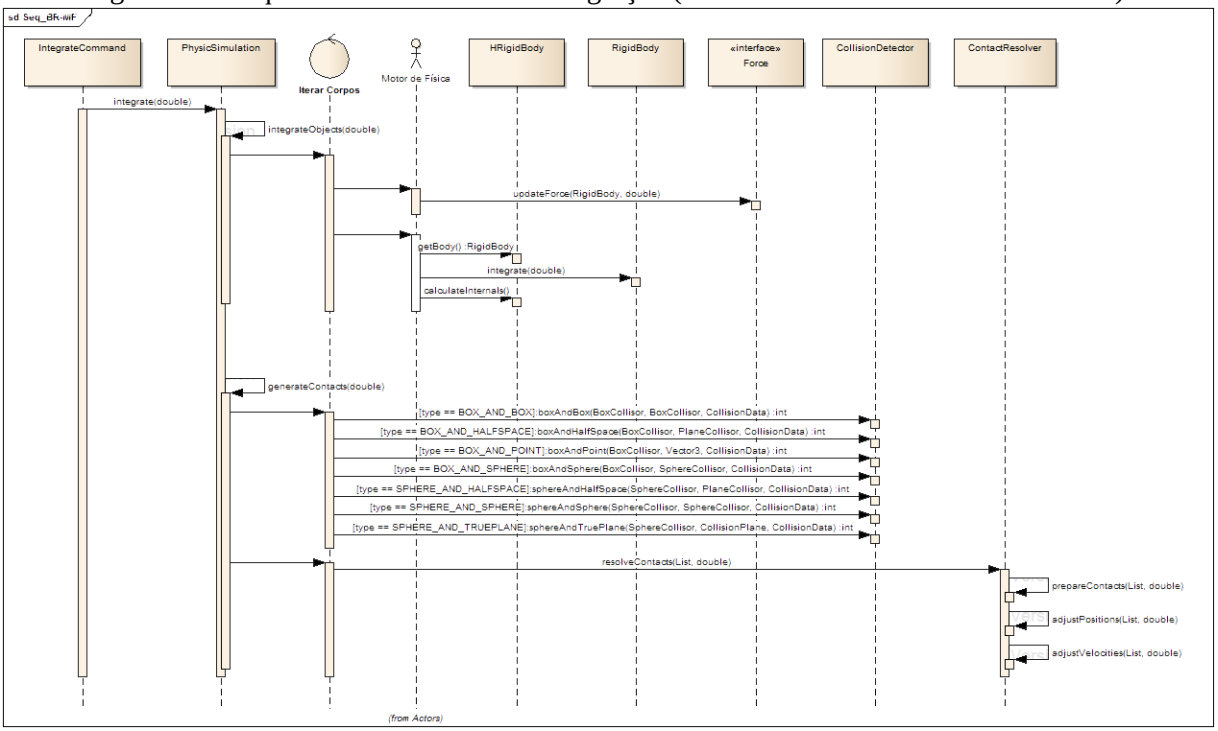



Figura 17 – Sequência do comando de integração (Biblioteca Remota e Motor de Física)

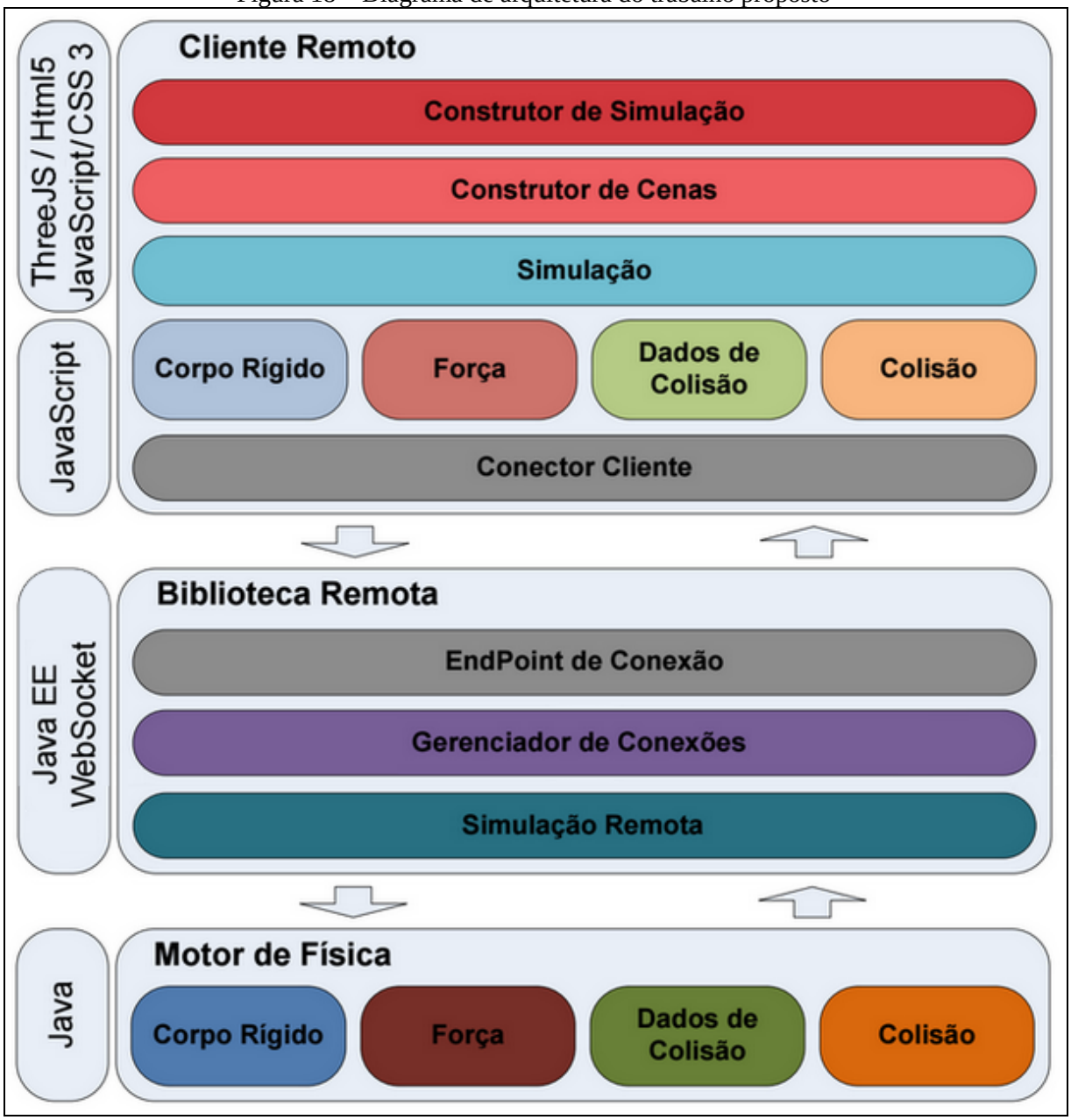


No diagrama apresentado o usuário do CR invoca a ação de integração (`integrate`) através da `Simulation` – o mesmo comportamento é apresentado se a invocação partir do *loop* do `SimulationManager`, dado mesmo propósito e funcionamento. Este método primeiramente monta a mensagem do comando a ser enviado para a BR, e em seguida envia (`sendMessage`). A mensagem é recebida na BR (`onMessage`) e é encaminhada para o processamento no orquestrador de simulações, na classe `SimulationOrchestrator`. Assim que a mensagem é convertida em comando (`processMessage`) e seu tipo é detectado, a mesma é processada (`process`). Na execução do comando de integração é realizada a ação de integrar objetos (`integrateObjects`) e de gerar contatos (`generateContacts`). Para cada objeto integrado são aplicadas as forças existentes (`updateForce`), realizados os cálculos de integração (`integrate`) e recalculada a matriz de transformação (`calculateInternals`). Já na detecção de contatos, é verificada cada possível colisão anteriormente cadastradas (`boxAndBox`, `boxAndHalfSpace`, `boxAndPoint`, `boxAndSphere`, `sphereAndHalfSpace`, `sphereAndSphere` e `sphereAndTruePlane`) e as que ocorreram são separadas para posterior resolução (`resolveContacts`). Na resolução de colisões os dados são calculados (`prepareContacts`) e as posições e velocidades dos corpos são ajustadas (`adjustPositions` e `adjustVelocities`). Ao finalizar o processo de integração é gerado uma mensagem com as novas posições e matrizes de transformação dos corpos, e a mesma é enviada (`sendAckMessage`) para a origem do comando, ou seja, o CR. A mensagem será recebida pelo *WebSocket* cliente (`onMessage`) e despachada para a simulação - que irá processá-la e atualizar os corpos e seus respectivos objetos gráficos.

### 3.2.8 Diagrama de arquitetura

Esta seção apresenta o diagrama das camadas da arquitetura do MF, da BR e do CR. A Figura 18 apresenta este diagrama.

Figura 18 – Diagrama de arquitetura do trabalho proposto




A arquitetura do Motor de Física foi especificada seguindo a modelagem orientada à objetos, com alto reuso e extensibilidade. O modelo adotado visa que à utilização do mesmo através da Biblioteca Remota ou de maneira independente como biblioteca externa em outros projetos.

A BR também foi projetada segundo a orientação à objetos e seguindo os mesmos princípios adotados no MF, para que facilitar o acompanhamento do MF e agregar as novas funcionalidades adicionadas ao motor. Na BR somente a simulação utiliza o MF, e a mesma é encapsulada e manipulada somente através do gerenciador de conexões pelo *EndPoint* do *WebSocket*.

O Cliente Remoto foi concebido para permitir que as funcionalidades do motor fossem utilizadas remotamente, sendo assim, parte das suas camadas equivalem-se às do MF. Utilizando o CR os protocolos de comunicação necessários para realizar a comunicação com a BR tornam-se transparentes para o usuário, visto que a única camada com essa responsabilidade se concentra na simulação.

O construtor de cenas integra os elementos da simulação com a biblioteca de cenas ThreeJS. Toda a estrutura é gerenciada pelo construtor de simulação, que é responsável por encapsular a simulação, o ciclo de integração dos elementos e a integração entre elementos gráficos e lógicos do CR.

## 3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas na implementação do Motor de Física, da Biblioteca Remota e do Cliente Remoto, bem como, detalhes das principais classes e rotinas implementadas durante o desenvolvimento deste trabalho.

### 3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento do Motor de Física, foi utilizada a linguagem de programação Java, na versão 7. Como ambiente de desenvolvimento foi utilizado o Eclipse IDE for Java EE Developers.

Na criação da Biblioteca Remota também foram utilizados a linguagem de programação Java e o Eclipse IDE 4.4.0 for Java EE Developers como ambiente de desenvolvimento. Como servidor de aplicação fui usado o Tomcat 8.0.18 (TOMCAT, 2015).

No desenvolvimento do Cliente Remoto utilizou-se a linguagem JavaScript e elementos do HTML5, principalmente o canvas. Como ambiente de desenvolvimento para o Cliente Remoto foi utilizado o editor de texto Sublime Text 3, na build 3065 (SUBLIME, 2015). Para a representação gráfica e desenho de cenas foi utilizado a biblioteca conhecida como TREEJS.

Os navegadores utilizados durante o desenvolvimento foram o GoogleChrome 32.0.1678.0 dev-m Aura, Internet Explorer 10.0.9200.16721, Opera 17.0 e Mozilla Firefox 25.0.

Para execução e depuração de todo o desenvolvimento do trabalho, utilizou-se um notebook Avell B153 com sistema operacional Windows 7 64 bits, processador Intel Core I7 3630QM 2.4 Giga Hertz (Ghz), memória de 16 GB DDR3 e placa de vídeo Nvidia GeForce GT 640M.

Todo o gerenciamento dos códigos fonte foi realizado através do repositório remoto Git do GitHub e como ferramenta para realização das submissões foi utilizado o próprio ambiente de desenvolvimento IDE for Java EE Developers.

### 3.3.2 O Motor de Física

Essa seção descreve os conceitos utilizados no Motor de Física (MF) e sua arquitetura. Também apresenta como foi desenvolvida a abstração da arquitetura orientada a objetos, viabilizando futuras extensões.

#### 3.3.2.1 Arquitetura do Motor de Física

O MF é constituído por conjuntos de corpos rígidos, de dados de colisão, de colisões e de forças. No MF um corpo rígido representa a estrutura e comportamento de um corpo rígido segundo a mecânica clássica. Os dados de colisão representam os coeficientes de fricção, restituição e tolerância provenientes do contato entre materiais de dois tipos. As colisões são a associação entre os dados de colisão e dois corpos rígidos, sendo a mesma utilizada para indicar ao MF que é necessário analisar se há contato entre os corpos indicados - e no caso de contato, quais os coeficientes que devem ser aplicados para a resolução do mesmo.

Cada representação de força possui cálculos e dados de entrada diferentes. A generalização de força é realizada através de uma interface, o que viabiliza a criação de novas forças que podem ser aplicadas sobre os corpos. Da mesma maneira, as rotinas de detecção e resolução de contatos são específicas para as formas geométricas caixas, bolas e planos. Respeitando os dados de entrada e saída, é possível criar novas forças e novas tratativas de contato para outras formas, viabilizando a expansão das funcionalidades do motor.

#### 3.3.2.2 Estrutura do Motor de Física

O MF foi criado para facilitar o desenvolvimento de aplicações que utilizam os conceitos de física Newtoniana, abstraindo as partes comuns no desenvolvimento dessas aplicações e permitindo que desenvolvedores com pouco conhecimento na área de Física desenvolvam aplicações de maneira simplificada.

A estrutura básica de uma aplicação utilizando o motor consiste de um ou mais corpos rígidos (`RigidBody`), uma ou mais forças (`Force`), um ou mais dados de colisão (`CollisionData`) e uma ou mais possíveis colisões (`Collision`).

A integração e orquestração destes objetos viabiliza a construção e execução de uma simulação. Porém, o motor não se responsabiliza por definir um ciclo de vida para a simulação. O ciclo deve ser definido e operado pelo cliente do motor.

As operações que devem ser executadas em cada ciclo são: aplicar as forças sobre os corpos, integrar os corpos, detectar colisões e resolver contatos. Em todas os cálculos realizados nessas operações é necessário informar um valor em comum, que é chamado de "delta T" - normalmente, a duração do último *frame*.

No Quadro 3 é apresentada a implementação da Força da Gravidade e no Quadro 4 é apresentada sua aplicação em um ciclo de vida.

Quadro 3 – Implementação da classe `Gravity`

```
 1  Package br.law123.force;
11  public class Gravity implements Force {
12
13      private Vector3 gravity;
14
27      @Override
28      public void updateForce(RigidBody body, double duration) {
29          if (!body.hasFiniteMass()) {
30              return;
31          }
32          // apply the mass-scaled force to the body
33          body.addForce(gravity.mult(body.getMass()));
34      }
35  }
```

Quadro 4 – Exemplo da aplicação da Força da Gravidade sobre corpos rígidos

```
11  public void applyForce(double duration) {
12        Gravity g = new Gravity(new Vector3(0, -10, 0));
13
14        for (RigidBody rb : getRigidBodies()) {
15              g.updateForce(rb, duration);
16        }
17  }
```

A realização do processo de integração de um corpo rígido é feita em duas etapas, sendo a primeira a atualização das velocidades e posteriormente a atualização da posição do corpo. Na atualização das velocidades os seguintes procedimentos são realizados: cálculo da aceleração linear a partir de entradas de força e angular a partir das entradas de torque; ajuste das velocidades linear e angular para a aceleração e impulso; ajuste do torque. Já na atualização das posições, é feito o ajuste da posição linear e angular; normalização da orientação; atualização das matrizes com a nova posição e orientação e por fim, atualização do armazenamento de energia cinética. A implementação deste processo corresponde ao método `integrate` que pode ser observado no Quadro 5.

Quadro 5 – Implementação da rotina de integração de corpo rígido

```
61  public void integrate(double duration) {
62      if (!isAwake) return;
63
64      // Calculate linear acceleration from force inputs.
65      lastFrameAcceleration = new Vector3(acceleration);
66      lastFrameAcceleration.addScaledVector(forceAccum, inverseMass);
67
68      // Calculate angular acceleration from torque inputs.
69      Vector3 angularAcceleration =
    inverseInertiaTensorWorld.transform(torqueAccum);
70
71      // Adjust velocities
72      // Update linear velocity from both acceleration and impulse.
73      velocity.addScaledVector(lastFrameAcceleration, duration);
74
75      // Update angular velocity from both acceleration and impulse.
76      rotation.addScaledVector(angularAcceleration, duration);
77
78      // Impose drag.
79      velocity.multToMe(Math.pow(linearDamping, duration));
80      rotation.multToMe(Math.pow(angularDamping, duration));
81
82      // Adjust positions
83      // Update linear position.
84      position.addScaledVector(velocity, duration);
85
86      // Update angular position.
87      orientation.addScaledVector(rotation, duration);
88
89      // Normalise the orientation, and update the matrices with the new
90      // position and orientation
91      calculateDerivedData();
92
93      // Clear accumulators.
94      clearAccumulators();
95
96      // Update the kinetic energy store, and possibly put the body to sleep
97      if (canSleep) {
98          double currentMotion = velocity.scalarProduct(velocity) +
    rotation.scalarProduct(rotation);
99
100         double bias = Math.pow(0.5, duration);
101         motion = bias * motion + (1 - bias) * currentMotion;
102
103         double sleepEpsilon = Core.get().getSleepEpsilon();
104         if (motion < sleepEpsilon) setAwake(false);
105         else if (motion > 10 * sleepEpsilon) motion = 10 * sleepEpsilon;
106     }
107 }
```

O Quadro 6 apresenta um exemplo de ciclo de vida para a integração dos corpos rígidos de uma simulação.

Quadro 6 – Exemplo de ciclo de vida para a integração de corpos rígidos

```
 9  public void integrateRigidBodies(double duration) {
10      for (RigidBody rb : getRigidBodies()) {
11          rb.integrate(duration);
12      }
13  }
```

Para realização da detecção e resolução de colisões, é necessário utilizar uma primitiva de colisão (`CollisionPrimitive`), que é uma especialização de um corpo rígido para uma forma geométrica específica. O motor disponibiliza primitivas de colisão para caixa (`CollisionBox`) e bola (`CollisionSphere`). Há também a primitiva para a plano (`CollisionPlane`), contudo, a mesma não é especialização de um corpo, e sim, a representação de um plano colidível. No Quadro 7 é apresentado a invocação do detector de colisões (`CollisionDetector`) para as primitivas do tipo bola e plano, demais tipos de detecção podem ser consultados na Figura 9.

Quadro 7 – Exemplo de detecção e resolução de contato entre bola e plano

```
12  public void detectCollision(double duration) {
13      CollisionData data = getCollisionData();
14
15      CollisionPlane plane = new CollisionPlane();
16      plane.setDirection(new Vector3(0, 1, 0));
17
18      CollisionSphere sphere = getSphere();
19
20      CollisionDetector.sphereAndTruePlane(sphere, plane, data);
21
22      ContactResolver resolver = getContactResolver();
23      resolver.resolveContacts(data.collectContacts(), duration);
24  }
```

### 3.3.3 A Biblioteca Remota

Essa seção descreve os conceitos utilizados na Biblioteca Remota (BR) e sua arquitetura. Também apresenta como foi desenvolvida a abstração da arquitetura orientada a objetos, viabilizando futuras extensões.

#### 3.3.3.1 Arquitetura da Biblioteca Remota

A BR consiste de um mecanismo desenvolvida para viabilizar a construção e execução de simulações utilizando o motor criado, remotamente. A biblioteca necessita do motor, entretanto, o inverso não é verdade. É possível usar o motor criado sem a utilização da Biblioteca Remota.

A BR utiliza o motor criado e oferece um canal de comunicação que torna o uso do motor distribuído, além de gerar e gerenciar um ciclo de vida da simulação. Sendo assim, as funcionalidades do motor anteriormente citadas são suportadas e disponibilizadas pela BR. Para viabilizar o uso remoto, foi utilizada a tecnologia *WebSocket* e uma aplicação prática do *design pattern Command*, possibilitando a criação de novos comandos.

A tecnologia *WebSocket* não é exclusiva do Java EE. Logo, é possível construir clientes que utilizem a BR em qualquer plataforma que implemente *WebSocket*.

#### 3.3.3.2 Estrutura da Biblioteca Remota

Para realizar a comunicação entre a BR e a outra ponta da conexão *WebSocket* - aqui denominado cliente, foi especificado um protocolo de comunicação utilizando o formato JSON (apêndice D) para fácil especificação e interpretação. O *EndPoint* do *WebSocket* (`PhysicSimulationEndPoint`) recebe uma nova mensagem (ver diagrama de sequência) e escalona para o respectivo processador (`SimulationProcessor`). Esse procedimento é apresentado no Quadro 8.

Quadro 8 – Procedimento de escalonamento do processador de mensagens

```
25 public void processMessage(String message) {
26     JSONObject object = new JSONObject(message);
27 
28     Integer _id = object.getInt("id");
29 
30     String strType = object.getString("type");
31     CommandType cmd = CommandType.valueOf(strType);
32 
33     try {
34         CommandProcessor processor = cmd.getProcessor().newInstance();
35         JSONObject data = processor.process(simulation,
   object.getJSONObject("data"));
36         sendAckMessage(_id, cmd, data);
37     } catch (Exception e) {
38         e.printStackTrace();
39     }
40 }
```

O processador irá interpretar a mensagem - dado seu formato, transformando-a em comando e despachando para o respectivo tratador. Assim que a mensagem é tratada, o processador irá responder ao cliente com uma nova mensagem - contendo o resultado do comando processado.

Cada comando possui seus dados de entrada, procedimentos a serem executados e dados de saída. Parte dos comandos são responsáveis pela criação e manutenção da simulação, utilizando as classes do pacote `simulation`. A classe `PhysicSimulation` é responsável por manter e operar a simulação, sendo utilizada por todos os comandos. Entre os

comandos, o principal e de maior complexidade é o "realizar integração" (`IntegrateCommand`), sua implementação é apresentada no Quadro 9.

Quadro 9 – Implementação do comando de integração

```
22  public class IntegrateCommand implements CommandProcessor {
23
24      @Override
25      public JSONObject process(PhysicSimulation simulation, JSONObject data)
    {
26          double duration = data.getDouble("duration");
27          simulation.integrate(duration);
28
29          Map<String, HRigidBody> bodys = simulation.getRigidBodys();
30          JSONArray array = new JSONArray();
31          for (Entry<String, HRigidBody> e : bodys.entrySet()) {
32              JSONObject obj = new JSONObject();
33              obj.put("id", e.getKey());
34
35              RigidBody body = e.getValue().getBody();
36                     obj.put("position", getJSONVector3(body.getPosition()));
37              float[] mat = new float[16];
38              body.getGLTransform(mat);
39              obj.put("transform", mat);
40              array.put(obj);
41          }
42
43          List<Contact> contacts = simulation.getContacts();
44
45          JSONArray conts = new JSONArray();
46          for (Contact c : contacts) {
47               JSONObject cont = new JSONObject();
48               cont.put("friction", c.getFriction());
49               cont.put("restitution", c.getRestitution());
50               cont.put("penetration", c.getPenetration());
51               cont.put("contactNormal",
    getJSONVector3(c.getContactNormal()));
52               cont.put("contactPoint", getJSONVector3(c.getContactPoint()));
53               cont.put("contactToWorld", c.getContactToWorld().data);
54               cont.put("body1", c.getBody()[0]);
55               cont.put("body2", c.getBody()[1]);
56               conts.put(cont);
57          }
58
59          JSONObject _result = new JSONObject();
60          _result.put("_rigidBodys", array);
61          _result.put("_contacts", conts);
62          return _result;
63      }
```

O comando de integração é responsável por invocar o ciclo de atualização da simulação. Para isso, o comando recebe a duração do último *frame* e a passa na invocação do procedimento `integrate` da simulação. Após a integração, o comando irá gerar uma lista com as novas posições e matrizes de transformação de cada corpo, além de uma lista das colisões detectadas e tratadas. O ciclo de integração definido para a representação de uma simulação na BR pode ser observado no Quadro 10.

Quadro 10 – Implementação do comando de integração

```
169 public void integrate(double duration) {
170     // Update the objects
171     integrateObjects(duration);
172     // Perform the contact generation
173     generateContacts(duration);
174 }
175 
176 protected void integrateObjects(double duration) {
177     for (HRigidBody rb : rigidBodys.values()) {
178         Collection<HForce> forces = forcesByBody.get(rb);
179         if (rb.isUseWorldForces()) {
180             forces = this.forces.values();
181         }
182         if (forces != null) {
183             for (Force f : forces) {
184                 f.updateForce(rb.getBody(), duration);
185             }
186         }
187         rb.getBody().integrate(duration);
188         rb.calculateInternals();
189     }
190 }
191 
192 protected void generateContacts(double duration) {
193     // Create the ground plane data
194     CollisionPlane plane = new CollisionPlane(new Vector3(0, 1, 0));
195 
196     for (Collision col : collisions.values()) {
197         CollisionData data = col.getData();
198         if (!data.hasMoreContacts() || !col.isEnable()) {
199             continue;
200         }
201         HRigidBody rb1 = col.getRb1();
202         HRigidBody rb2 = col.getRb2();
203         switch (col.getType()) {
204         case BOX_AND_BOX: CollisionDetector.boxAndBox(rb1, rb2, data); break;
205         case BOX_AND_HALFSPACE: CollisionDetector.boxAndHalfSpace(rb1, plane,
    data); break;
206         case BOX_AND_SPHERE: CollisionDetector.boxAndSphere(rb1, rb2, data);
    break;
207         case SPHERE_AND_HALFSPACE: CollisionDetector.sphereAndHalfSpace(rb1,
    plane, data); break;
208         case SPHERE_AND_SPHERE: CollisionDetector.sphereAndSphere(rb1, rb2,
    data); break;
209         case SPHERE_AND_TRUEPLANE: CollisionDetector.sphereAndTruePlane(rb1,
    plane, data); break;
210         default: System.err.println("Unkown collision type: " + col.getType());
211         }
212     }
213     for (CollisionData cd : collisionDatas.values()) {
214         resolver.resolveContacts(cd.collectContacts(), duration);
215         cd.reset(maxContact);
216     }
217 }
```

O procedimento de integração é executado em duas etapas. A primeira etapa trata da integração dos objetos e a segunda da detecção das colisões.

Na primeira etapa, para cada corpo rígido da simulação são aplicadas as forças, e em seguida o corpo é integrado. Na segunda etapa, as colisões são detectadas conforme os tipos dos corpos envolvido. Caso haja contato, os mesmos são resolvidos de acordo com os dados de colisão dos materiais envolvidos.

### 3.3.4 O Cliente Remoto

Essa seção descreve os conceitos utilizados no Cliente Remoto (CR) e sua arquitetura. Também apresenta como foi desenvolvida a implementação com *WebSocket* e a integração com o ThreeJS.

#### 3.3.4.1 Arquitetura do Cliente Remoto

O CR foi desenvolvido com o propósito de permitir a construção de aplicações que utilizem o Motor de Física e exibam o produto da simulação no navegador. Para tanto, foi implementado um cliente *WebSocket* que utilize os protocolos especificados (Apêndice D) para se comunicar com a Biblioteca Remota. Além disso, foi realiza a integração com a biblioteca ThreeJS, permitindo a construção de cenas 3D para exibição da simulação.

O CR foi projetado exclusivamente para se comunicar com a BR e depende da mesma. A BR por sua vez, não depende do CR. Porém, dada a estrutura criada e a simplicidade da mesma é possível integrar o CR com outros mecanismos, permitindo que elementos além da Física sejam introduzidos à cena.

#### 3.3.4.2 Estrutura do Cliente Remoto

Para que todas as funcionalidades do MF fossem disponibilizadas no CR foi criado um espelhamento de suas definições, ou seja, suas classes de dados - conforme pode ser comparado através dos diagramas das Figuras 7, 8, 9 e 15.

Além das classes `RigidBody`, `Collision`, `CollisionData` e `Force` – que são utilizadas para definição dos elementos que irão compor a simulação e para viabilizar a integração com o ThreeJS, também há correspondência entre as classes `PhysicSimulation` da BR, e `Simulation` do CR.

Ambas as classes são responsáveis por manter os dados da simulação, uma em cada ponto da conexão. Contudo, `PhysicSimulation` é responsável pelo ciclo de vida de integração junto ao MF, enquanto que `Simulation` encapsula as funcionalidades de comunicação (`SimulationWS`) com a BR, além das rotinas de integração com o ThreeJS.

Sendo assim, para adicionar um novo corpo rígido (`RigidBody`) à simulação, é necessário invocar o método `bindRigidBody` da classe `Simulation`, sua implementação é apresentada no Quadro 11.

Quadro 11 – Implementação da função de adicionar corpo rígido a simulação

```
 70 HEFESTO.Simulation.prototype = {
 71     constructor : HEFESTO.Simulation,
 72 
 99     bindRigidBody: function (body) {
100         var mesh = body.mesh;
101         body.mesh = undefined;
102         try {
103 this._simulationWS.sendMessage(HEFESTO.CommandType.BIND_RIGID_BODY, body);
104         } finally {
105             body.mesh = mesh;
106         }
107 
108         this._rigidBodys[body.id] = body;
109         this.rigidBodyCount++;
110     },
110 
328 };
```

O novo corpo rígido recebido como parâmetro é adicionado à lista de corpos da simulação, e um novo comando é enviado para a BR através da invocação de `sendMessage` da classe `SimulationWS`. A implementação do envio de comandos pode ser observada através do Quadro 12.

Quadro 12 – Implementação da função de envio de comando

```
 16 HEFESTO.SimulationWS.prototype = {
 17 construtor: HEFESTO.SimulationWS,
 18 
105 sendMessage: function(type, data) {
106     //se não está ativa, precisa receber a primeira mensagem do servidor.
107     if (!this._simulation._active) {
108         return -1;
109     }
110 
111     var msg = {};
112     msg['id'] = this._messageCount++;
113     msg['type'] = type;
114     msg['data'] = data;
115 
116     // enfilera mensagem
117     this._messageQueue.push(msg);
118 
119     this._ws.send(JSON.prune(msg));
120     log('SND: ' + msg);
121 },
131 
132 };
```

O tipo do comando é recebido através do parâmetro `type`, e o parâmetro `data` contém os dados necessários para garantir a execução do mesmo. Para cada nova mensagem é gerado

um identificador – dado pela quantidade de mensagens já enviadas. O identificador da mensagem é utilizado para removê-la da lista de mensagens enviadas quando o retorno do seu processamento for recebido no método `onMessage`. A implementação do método citado é apresentada no Quadro 13.

Quadro 13 – Implementação da função de recebimento de comando processado

```
 16 HEFESTO.SimulationWS.prototype = {
 17 construtor: HEFESTO.SimulationWS,
 18 
 79 onmessage: function (data) {
 80     log('RCV: ' + data.data);
 81 
 82     var _data = JSON.parse(data.data);
 83     // notifica a simulação com a nova mensagem.
 84     this._simulation.onmessage(_data.type, _data.data);
 85 
 86     // remove mensagem da lista.
 87     if (_data.id > -1) {
 88         var first = this._messageQueue.shift();
 89         if (first.id != _data.id) {
 90             this._messageQueue.push(first);
 91             var actual = this._messageQueue.shift();
 92             while (actual.id != _data.id & actual != first) {
 93                 this._messageQueue.push(actual);
 94                 actual = this._messageQueue.shift();
 95             }
 96         }
 97     }
 98 },
131 
132 };
```

Os resultado do processamento de um comando é recebido através do parâmetro `data` e o mesmo é despachado para processamento de resultado através do método `onmessage` da simulação (`Simulation`). Assim que o resultado é processado, a mensagem é removida da lista de mensagens enviadas.

O processamento do resultado do comando é dado pelo seu tipo, mas não são todos os comandos que precisam de algum processamento. O principal comando que precisa de processamento para obter um resultado é o de integração, e a implementação do método `onintegrate` responsável por este processamento pode ser observada no Quadro 14.

Quadro 14 – Implementação da função onmessage da classe Simulation

```
 70 HEFESTO.Simulation.prototype = {
 71     constructor : HEFESTO.Simulation,
 72 
271     onintegrate: function (data) {
272         for (i = 0; i < data._rigidBodys.length; i++) {
273             var _rb = data._rigidBodys[i];
274 
275             var rb = this._rigidBodys[_rb.id];
276 
277             var p = new THREE.Vector3();
278             p.x =  _rb.position.x;
279             p.y =  _rb.position.y;
280             p.z =  _rb.position.z;
281 
282             rb.position = p;
283 
284         // caso nao possua mesh, nao precisa ajustar valores de exibicao
285             if (rb.mesh == undefined) {
286                 continue;
287             }
288 
289             //rb._mesh.position.set(p);
290             rb.mesh.position.x = p.x;
291             rb.mesh.position.y = p.y;
292             rb.mesh.position.z = p.z;
293 
294             rb.mesh.matrix.identity();
295             var a = _rb.transform;
296             var m = new THREE.Matrix4();
297             m.set(a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8],
    a[9], a[10], a[11], a[12], a[13], a[14], a[15]);
298 
299             rb.mesh.applyMatrix(m);
300             rb.mesh.updateMatrix();
301             rb.mesh.matrix.setPosition(p);
302         }
303         if (this.contactListener != undefined && this.contactListener !=
    null) {
304             var contacts = data._contacts;
305             for (j = 0; j < contacts.length; j++) {
306                 this.contactListener(contacts[0]);
307             }
308         }
309     },
325 };
```

O retorno do comando de integração é a lista de novas posições e matrizes de transformação dos corpos que compõem a simulação. No método `onintegrate` a lista recebida é iterada e para cada elemento é obtido o corpo rígido. Quando a propriedade `matrixAutoUpdate` da `mesh` - contida no corpo rígido - está desligada, a responsabilidade de atualização da mesma não pertence mais ao ThreeJS, permitindo a atualização manual da posição e da matriz de transformação. Ao atualizar manualmente estas propriedades, é

possível visualizar o resultado da integração em cada ciclo, e consequentemente, a execução de uma simulação.

### 3.3.5 Operacionalidade da implementação

Para demonstrar a operacionalidade do Cliente Remoto será descrita a implementação de um exemplo, denominado demo-monografia. O exemplo proposto contém uma esfera e a força da gravidade, dados de colisão e colisão entre esfera e solo.

O primeiro passo para criar o exemplo é definir a estrutura inicial do ThreeJS. Para tanto, é necessário declarar e inicializar o *canvas*, o renderizador, a câmera e a cena. O código correspondente está no Quadro 15.

Quadro 15 – Definição da estrutura inicial do ThreeJS

```
34  <div id="canvas-3d" class="viewer"></div>
35
36  <script>
37      var canvas, renderer, camera, scene;
38      var ms_Width = $(window).width(), ms_Height = $(window).height();
39
40      var simulation = new HEFESTO.Simulation();
41      var timming = new HEFESTO.Timming();
42
50      function initializeTHREEJS() {
51          canvas = document.getElementById('canvas-3d');
52
53          //inicializa renderer, camera e scene
54          renderer = new THREE.WebGLRenderer();
55          renderer.setSize(ms_Width, ms_Height);
56          renderer.setClearColor(0xffffff, 1);
57          canvas.appendChild(renderer.domElement);
58
59          scene = new THREE.Scene();
60
61          camera = new THREE.PerspectiveCamera(55.0, ms_Width / ms_Height,
    0.5, 3000000);
62          camera.position.set(10, 50, 150);
63          camera.lookAt(new THREE.Vector3(0, 0, 0));
64
65          // adiciona luz
66          var directionalLight = new THREE.DirectionalLight(0xffff55, 1);
67          directionalLight.position.set(10, 10, 100);
68          scene.add(directionalLight);
69      }
```

No método `initializeTHREEJS` foram inicializados os componentes `canvas`, `render`, `camera` e `scene`. Todos foram declarados em espoço público para permitir a utilização nos demais métodos que serão apresentados. Também foram declarados e inicializados os recursos `simulation` e `timming`, ambos da biblioteca HEFESTO – o Cliente Remoto. As inicializações apresentadas devem ser a primeira ação executada ao carregar a tela, e em seguida deve-se inicializar a simulação, conforme Quadro 16.

Quadro 16 – Rotina de inicialização do ThreeJS e da simulação

```
43      $(function() {
44          initializeTHREEJS();
45
46          simulation.init();
47          continueInitilization();
48      });
49
71      function continueInitilization() {
72          if (!simulation.isBusy()) {
73              initializeSimulation();
74
75              timming.start();
76              timming.update();
77              mainLoop();
78          } else {
79              requestAnimationFrame(continueInitilization);
80          }
81      }
```

Após a inicialização do ThreeJS deve-se iniciar a simulação, procedimento que ocorre em dois passos: estabelecer conexão e criar cena. Para estabelecer a conexão deve-se invocar a operação `init` e liberar a *thread* principal. Com a liberação da *thread* principal - pode ser realizada através do `requestAnimationFrame`[2] - é possível que a simulação receba a primeira mensagem da Biblioteca Remota, que é considerada o *handshake*, tornando a simulação disponível.

Assim que a simulação está disponível é possível adicionar elementos à mesma (`initializeSimulation`). A preparação da simulação é apresentada no Quadro 17.

[2] `Window.requestAnimationFrame`() informa ao navegador que será realizado uma animação, é solicita uma chamada de função especificada para atualizar uma animação antes da próxima redesenhar de tela.

Quadro 17 – Preparação da simulação

```
 83      function initializeSimulation() {
 84          //base base no threejs (somente desenho)
 85          var geometry = new THREE.BoxGeometry(100, 1, 100);
 86          var material = new THREE.MeshBasicMaterial( {color: 0x09400a,
     transparent: true, opacity: 0.2} );
 87          var baseMesh = new THREE.Mesh( geometry, material );
 88          scene.add(baseMesh);
 89
 90
 91          // cria esfera no threejs
 92          var  material = new THREE.MeshLambertMaterial( { color:
     Math.random() * 0xffffff, shading: THREE.SmoothShading } );
 93          var geometryBall = new THREE.SphereGeometry(5, 32, 16);

 94          var ballMesh = new THREE.Mesh(geometryBall, material);
 95          ballMesh.position.y = 50;
 96          ballMesh.matrixAutoUpdate = false;
 97          scene.add(ballMesh);
 98
 98          //cria dados de colisao entre esfera e solo
 99          var cdGround = new HEFESTO.CollisionData(0.9, 0.9, 0.01);
100          cdGround.maxCollision = 256 * 256;
101          simulation.bindCollisionData(cdGround);
102
103          // cria corpo rigido da esfera
104          var rb = new HEFESTO.RigidBody(ballMesh);
105          rb.radius = 5;
106          rb.mass = 200;
107          rb.inertiaTensor.set(5360957.0, -0.0, -0.0, -0.0, 5360957.0, -0.0,
     -0.0, -0.0, 5360957.0);
108          rb.linearDamping = 0.95;
109          rb.angularDamping = 0.8;
110          rb.useWorldForces = true;
111          simulation.bindRigidBody(rb);
112
113          // cria colisao entre o solo e a esfera
114          var collisionGround = new
     HEFESTO.Collision(HEFESTO.CollisionType.SPHERE_AND_TRUEPLANE, cdGround, rb,
     null);
115          simulation.bindCollision(collisionGround);
116
117          // adiciona força da gravidade
118          var gravity = new HEFESTO.GravityForce(new THREE.Vector3(0, -10,
     0));
119          simulation.bindForce(gravity);
120      };
```

No método `initializeSimulation` são configurados os elementos que irão compor a simulação. Inicialmente é criado uma base representando o solo, e uma esfera que irá representar o corpo rígido no ThreeJS. Então são definidos os dados de colisão (`CollisionData`) entre o solo e qualquer corpo, e enviado (`bindCollisionData`) para a BR. Em seguida é definido o corpo rígido (`RigidBody`) para a esfera e enviado (`bindRigidBody`) à BR, e o mesmo para a declaração de colisão (`Collision`) entre a esfera e o solo que é

adicionada a simulação através de `bindCollision`. Por fim, a invocação de `bindForce` adiciona a força de gravidade (`Gravity`) à simulação.

É possível adicionar novos elementos em qualquer momento – desde que a simulação esteja ativa. Uma vez que a preparação inicial da simulação foi concluída, pode-se iniciar a temporização de *frames* e o *loop* principal (ver Quadro 16). O *loop* principal é apresentado no Quadro 18.

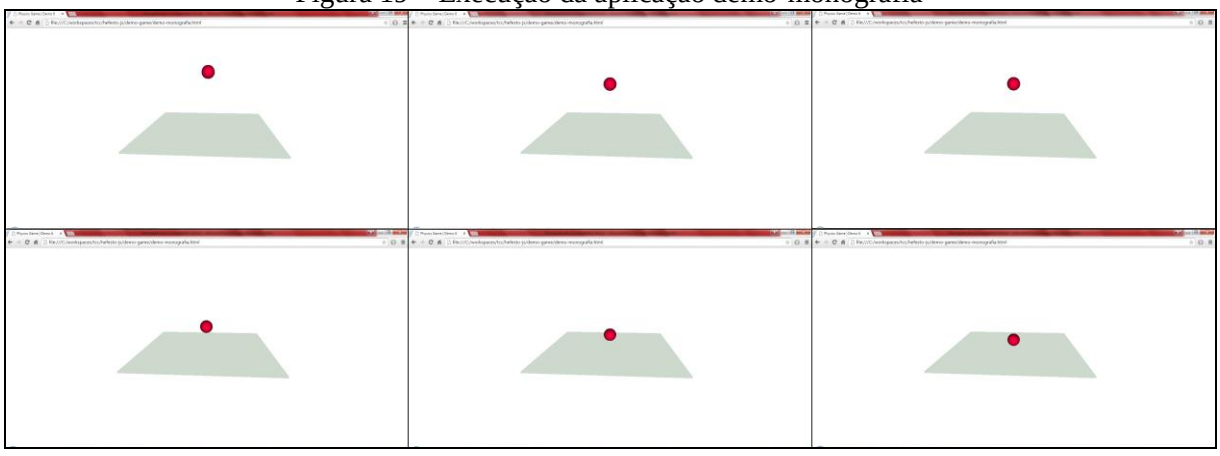
Quadro 18 – Implementação do *loop* principal

```
123     function mainLoop() {
124         requestAnimationFrame(mainLoop);
125         if (!simulation.isBusy()) {
126             simulation.integrate(timming.getLastFrameDuration() * 0.001);
127 
128             renderer.render(scene, camera);
129             timming.update();
130         }
131     };
```

No *loop* principal é verificado o estado da simulação (`isBusy`), ou seja, se não há mensagens aguardando o processamento. Se estiver disponível é solicitado que a BR execute a integração (`integrate`) dos elementos da simulação - passando o tempo do último *frame*. Em seguida é invocada a ação de redesenhar (`render`) do ThreeJS, e por fim é marcado o tempo de duração do último *frame*.

Seguindo os passos descritos acima, será produzido uma aplicação conforme apresentada na Figura 19. Nesta aplicação a esfera irá cair sob o efeito da gravidade e ficar quicando na base definida até que pare.

Figura 19 – Execução da aplicação demo-monografia



## 3.4 RESULTADOS E DISCUSSÃO

Este trabalho apresenta o desenvolvimento de um motor de física e mecanismos que permitam sua utilização remotamente. Foi contemplado o objetivo de desenvolver um motor de física, por meio da construção de um mecanismo que simule as Leis de Newton

computacionalmente. O motor desenvolvido é uma migração e refatoração da *engine* chamada Cyclone de Millington (2007), onde é utilizada a integração Newton-Euler e o Princípio de D'Alembert para os cálculos de física, permitindo que diversas propriedades de um corpo sejam parametrizadas - além de viabilizar a extensão do motor para novas áreas além da Mecânica Clássica. No original a *engine* Cyclone foi construída com a linguagem C++, na migração optou-se pela linguagem Java e foram realizadas alterações para adequá-la as necessidades deste trabalho.

Para atender o objetivo de permitir o uso do motor através da *web*, foi criada uma biblioteca remota. A biblioteca foi desenvolvida baseada em comandos, o que permite que novas funcionalidades sejam adicionadas a qualquer momento. Para a utilização da biblioteca criou-se uma camada WebSocket com um protocolo próprio de mensagens, que permite uma fácil interpretação e extensão. Uma vez que utilize o protocolo especificado, esta implementação de WebSocket pode receber conexão de qualquer origem, oriunda de um navegador ou mesmo de outras plataformas.

Para testar e validar a estrutura desenvolvida, criou-se um cliente para a biblioteca remota. Cliente este, desenvolvido em JavaScript, HTML5 e ThreeJS – que é uma biblioteca para desenho de cenas em 3D. Sendo JavaScript para implementar a estrutura de construção da simulação e mecanismo de comunicação (através de *WebSockets*), HTML5 como o *container* para renderização das cenas 3D e ThreeJS uma biblioteca facilitadora para a construção das cenas.

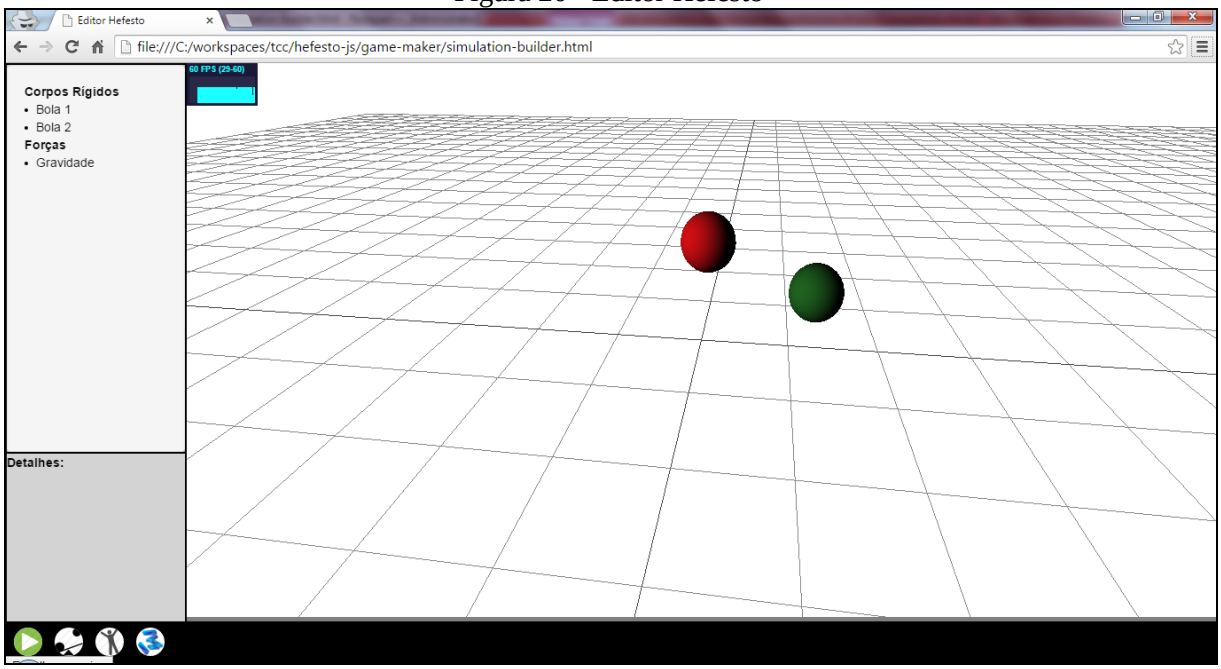
Inicialmente a disponibilzação dos serviços da biblioteca remota através de *WebSockets* foi de carácter experimental. Sua facilidade de uso e flexibilidade em permitir a criação de um protocolo próprio, se mostraram eficazes de acordo com as necessidades da biblioteca. Então decidiu-se que a biblioteca seria disponibilizada por *WebSockets* através de um servidor de aplicação (optou-se pelo Apache Tomcat).

Uma das intenções iniciais do trabalho era disponibilizar algum mecanismo que permitisse auxiliar a construção de aplicações didáticas. Para tanto, é necessário que a estrutura criada seja de fácil compreensão e absorção, possua e permita o uso de recursos com facilidade e atenda as necessidades quanto a performance de execução das simulações. Para validar os itens descritos, a seguir serão apresentados os testes qualitativos e de performance respectivamente, realizados com o intuito de averiguar a viabilidade do uso do trabalho no desenvolvimento de aplicações didáticas.

### 3.4.1 Testes qualitativos

Para viabilizar uma experiência mais amigável com o trabalho desenvolvido, foi construído um editor que permite a criação de uma simulação através de componentes gráficos – denominado Editor Hefesto. Com o mesmo pode-se criar simulações que contenham corpos rígidos, dados de colisão, colisões e forças – conforme Figura 20. O editor também viabilizou uma observação da estrutura criada e consequentemente a correção e aprimoramento de elementos da estrutura.

Figura 20 - Editor Hefesto



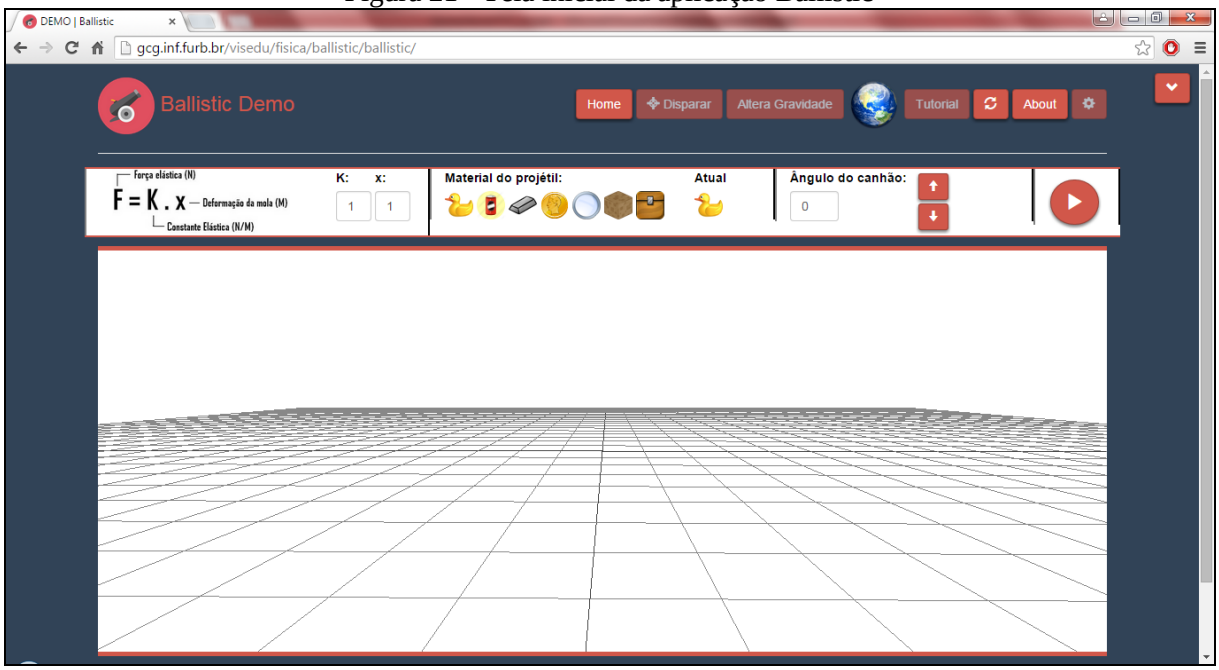

Durante o desenvolvimento deste trabalho, o professor Dalton S. Dos Reis apresentou o projeto Visualizador de material Educacional (VISEDU, 2015a) do Departamento de Sistemas e Computação da Fundação da Universidade Regional de Blumenau (FURB), para a construção de material didático na área de física do ensino médio.

Ao estabelecer contato com Barboza (2015) – que é professor de física do ensino médio, Dalton S. Dos Reis sondou-o a respeito das necessidades que o mesmo possui em relação a realização de demonstrações dos conteúdos abordados em sala de aula. Em parceria com o mesmo, decidiu-se construir uma aplicação que simule as leis de Hooke. Para realizar a construção da aplicação optou-se pelo motor desenvolvido neste trabalho, denominado Hefesto. Zanluca (2015), que é bolsista do mesmo projeto, ficou responsável pela construção da citada aplicação.

Acompanhou-se Zanluca (2015) durante o desenvolvimento da aplicação. Para tanto, participou-se de reunião com o professor Barboza (2015), que forneceu o material necessário

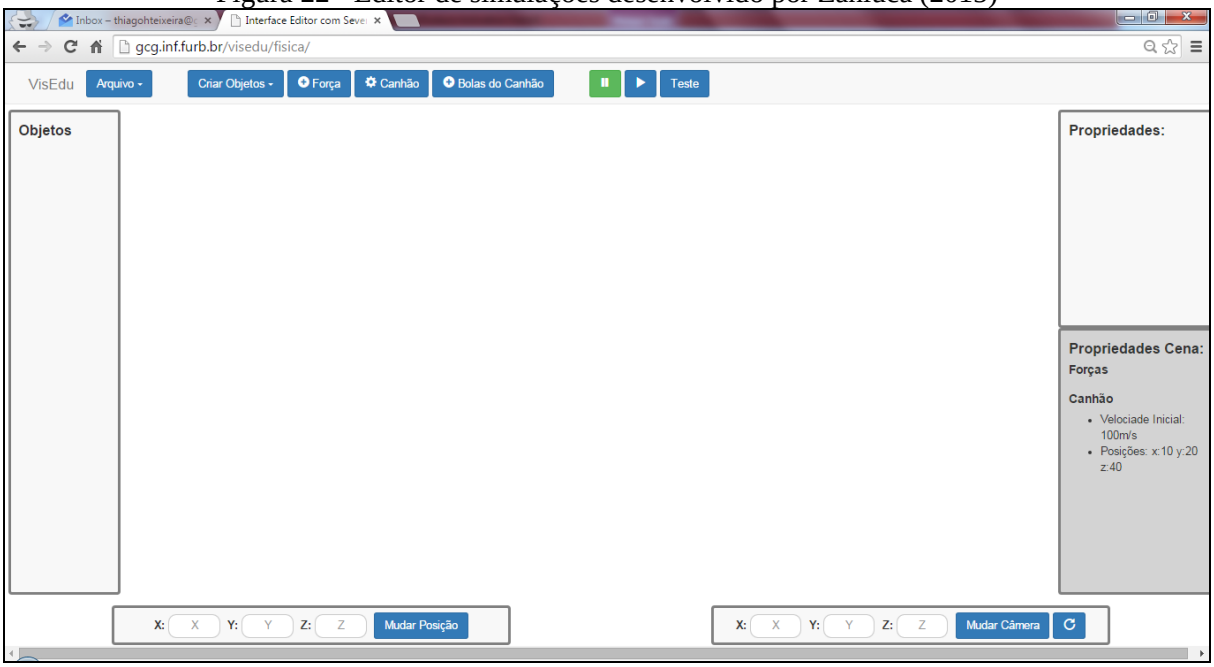
para fundamentação didática da mesma (Anexo A). Também foi realizada a instrução referente ao funcionamento, comportamento e características do Hefesto para que Zanluca (2015) tivesse o conhecimento necessário para construção da aplicação. A aplicação construída está disponível em Ballistic (2015) e pode ser observada na Figura 21.

Figura 21 - Tela inicial da aplicação Ballistic




Zanluca (2015) realizou também o desenvolvimento de uma interface gráfica para construção e execução de simulações com o uso do Hefesto. Da mesma forma que o Editor Hefesto, também é possível criar simulações contendo corpos rígidos, dados de colisão, colisões e forças. Ainda é possível utilizar elementos do tipo canhão e alvo, salvar o estado da execução em um arquivo, que pode ser carregado para realizar a execução novamente. O editor pode ser acessado em VisEdu (2015b) e é apresentado na Figura 22.

Figura 22 - Editor de simulações desenvolvido por Zanluca (2015)




Após o desenvolvimento das aplicações, Zanluca (2015) foi questionado a respeito do funcionamento e características do Hefesto Engine. Zanluca (2015) relatou que sua maior dificuldade foi com a linguagem de programação JavaScript, já que o mesmo nunca havia trabalhado com linguagens não tipadas. Quando questionado a respeito da dificuldade de produzir o material, o mesmo disse que foi simples. Segundo ele "não era necessário o conhecimento anterior de física, uma vez que os dados de entrada são especificados pelas fórmulas, é possível utilizar o motor com certa facilidade". O mesmo também sugeriu que fossem adicionados novos tipos de corpos rígidos ao motor, além de mecanismos que facilitem a descriminação de colisões e rotinas para obter valores de corpos rígidos específicos durante a execução da simulação.

### 3.4.2 Testes de performance

Os resultados obtidos foram coletados a partir do Motor de Física e da Biblioteca Remota. Os mesmos foram medidos através de testes experimentais da sua capacidade de renderizar cenas. Entende-se que a velocidade de renderização dos objetos virtuais é um fator crítico para o sucesso de ambas as soluções, pois o atraso no desenho da cena pode causar a perda da noção de realidade pelo usuário e tornar a simulação ineficaz de acordo com seu propósito.

No caso do Motor de Física, os testes foram realizados através de aplicações Swing com OpenGL, e para a Biblioteca e Cliente Remoto foi utilizado sua implementação com ThreeJS. Para o desenho de cenas, o método utilizado para medir o desempenho foi a

contagem do número de *Frames* Por Segundo (FPS), já no que se refere ao processamento das rotinas utilizou-se o tempo em milisegundos.

Acredita-se que dois fatores principais podem afetar o desempenho de ambas as soluções, sendo eles: a quantidade de elementos presentes na simulação e os tipos dos corpos – dado pelo custo do tratamento de colisões. Para o Cliente Remoto há um terceiro fator, que é a taxa de transmissão de rede e o processamento dos dados recebidos para aplicar as transformações à cena.

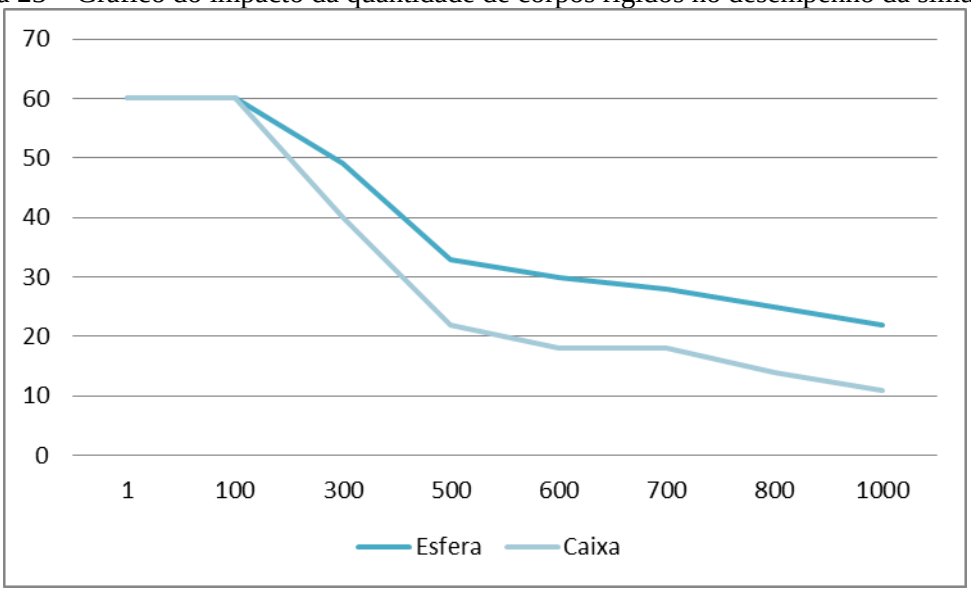
O primeiro conjunto de testes realizado objetivou medir o impacto dos tipos dos corpos rígidos na taxa de FPS da aplicação. Para tanto, foram realizadas medições da taxa de FPS com diferentes quantidades e tipos de corpos sendo desenhados ao mesmo tempo. As medições foram realizadas com a força da gravidade habilitada e o tratamento de colisão habilitados entre os corpos da simulação e o solo, sendo cada simulação com um tipo de corpo. Os resultados obtidos são apresentados na Tabela 1.

Tabela 1 – Medição da taxa de FPS na renderização de simulação com esferas e caixas

| **Quantidade de corpos rígidos** | **Média FPS** | |
|---|---|---|
| | **Esfera** | **Caixa** |
| 1 | 60 | 60 |
| 100 | 60 | 60 |
| 300 | 49 | 40 |
| 500 | 33 | 22 |
| 600 | 30 | 18 |
| 700 | 28 | 18 |
| 800 | 25 | 14 |
| 1000 | 22 | 11 |

Com base nos resultados obtidos, é possível observar que o desempenho da simulação mostrou-se sensível a quantidade de corpos rígidos que serão renderizados. A Figura 23 mostra o gráfico gerado com base nas medições, com o desempenho obtido através da medição do FPS na vertical e a quantidade de objetos na horizontal.

Figura 23 – Gráfico do impacto da quantidade de corpos rígidos no desempenho da simulação




O segundo conjunto de testes realizado visou medir o tempo de processamento de cada etapa da integração – com atenção especial para a influência da detecção e resolução de contatos entre os corpos rígidos presentes na cena. Este cenário ocorre quando a detecção de colisão está habilitada para todos os corpos da simulação.

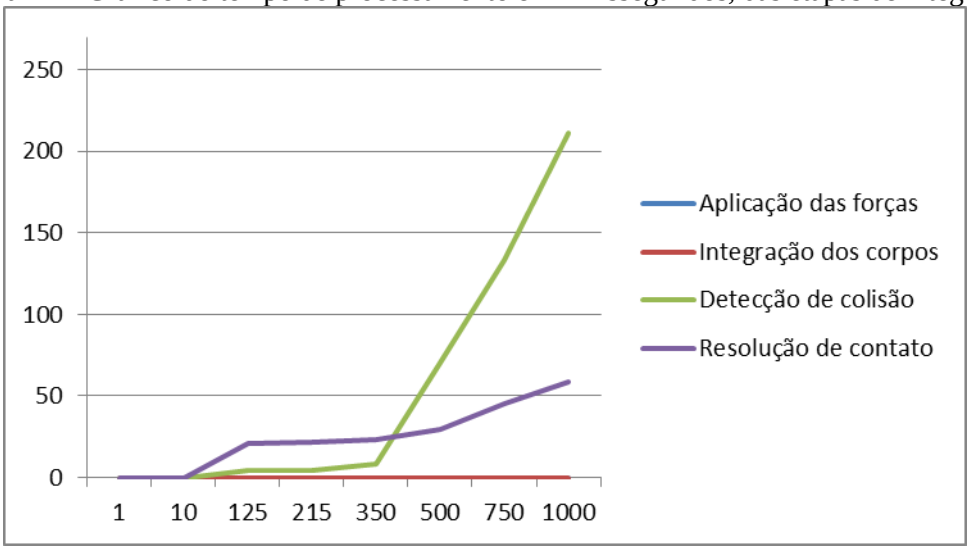
As simulações foram aprimoradas para gerar o máximo de contatos entre todos os corpos. Foi utilizado corpos do tipo caixa - empilhando-os de maneira a gerar um grande cubo - visto que a colisão entre caixas e demais corpos possui um custo de processamento mais acentuado, conforme apresenta o gráfico da Figura 23. As medições foram realizadas na Biblioteca Remota e levam em consideração o tempo de processamento de cada etapa da integração dos corpos, os dados obtidos são apresentados na Tabela 2.

Tabela 2 – Medição do custo operacional das rotinas de integração em milissegundos

| **Quantidade de corpos rígidos** | **Aplicação das forças** | **Integração dos corpos** | **Detecção de colisão** | **Resolução de contato** | **Tempo total** |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 |
| 125 | 0 | 0 | 4.2 | 20.9 | 25.1 |
| 216 | 0 | 0 | 4.4 | 21.3 | 25.7 |
| 343 | 0 | 0 | 8.4 | 23.1 | 31.5 |
| 512 | 0 | 0 | 70.1 | 29.4 | 99.5 |
| 729 | 0 | 0 | 133.6 | 45.5 | 179.1 |
| 1000 | 0 | 0 | 210.9 | 58.8 | 269.7 |

Através do gráfico apresentado na Figura 24 – com o tempo na vertical e a quantidade de objetos na horizontal, é possível observar que o desempenho da integração é extremamente senvível a quantidade de corpos. Sendo que a etapa de detecção de colisão se mostrou a mais custosa, seguida da resolução de contato.

Figura 24 – Gráfico do tempo de processamento em milissegundos, das etapas de integração




O terceiro conjunto de amostras foi obtido através do mesmo cenário do conjunto anterior. O cenário visa apresentar o comparativo do processamento das respostas recebidas pelo CR para viabilizar a renderização do novo estado da cena em distintos navegadores.

As amostras foram coletadas individualmente para os seguintes navegadores: Google Chrome 43.0.2357.81m, Mozilla Firefox 38.0.1, Internet Explorer 11.0.9600.17801 e Opera 29.0.1795.60 e as medições são apresentados na Tabela 3. Além dos navegadores citados, os testes também seriam realizados no navegador Safari 5.1.7 (versão atual do Safari para Windows) porém o navegador possui suporte completo a especificação da *WebSocket* somente a partir das versões 6.1 no iOS Safari e 7 no Safari (CANIUSE, 2014).

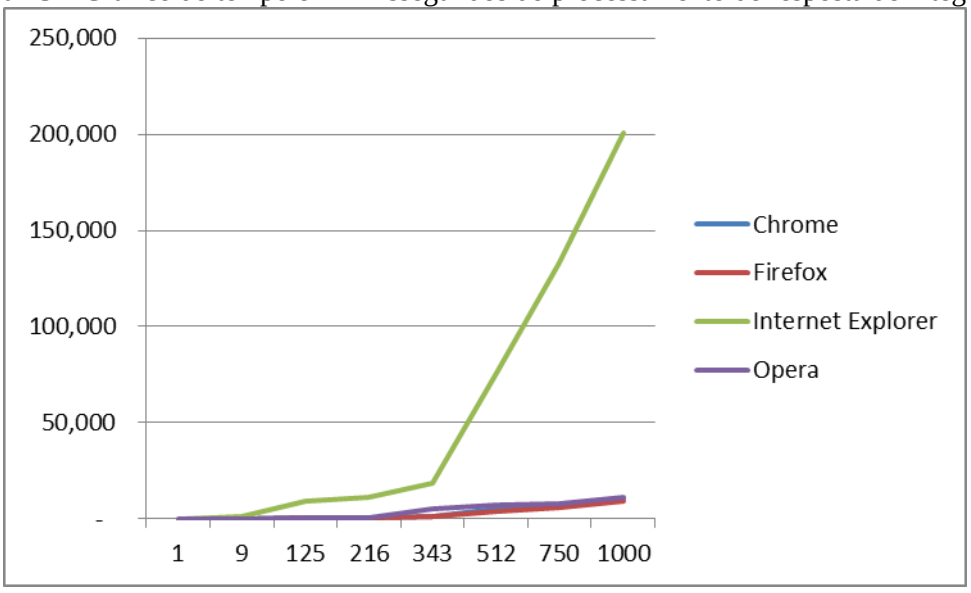
Tabela 3 – Medição do custo operacional das rotinas de processamento de resposta em milissegundos

| **Quantidade de corpos rígidos** | **Tempo de processamento da resposta** | | | |
|---|---|---|---|---|
| | **Chrome** | **Firefox** | **Internet Explorer** | **Opera** |
| 1 | 0,061 | 0,072 | 0,092 | 0,058 |
| 9 | 0,121 | 0,101 | 1,144 | 0,170 |
| 125 | 0,442 | 0,551 | 8,881 | 0,635 |
| 216 | 0,724 | 0,611 | 11,056 | 0,639 |
| 343 | 1,063 | 0,916 | 18,462 | 4,879 |
| 512 | 4,951 | 3,852 | 75,711 | 6,881 |
| 750 | 7,542 | 5,881 | 133,130 | 7,677 |
| 1000 | 10,733 | 9,199 | 200,846 | 11,094 |

Através da tabela é possível observar que o Internet Explorer apresentou o pior desempenho no que se refere ao processamento de resposta das integrações. Apesar do melhor tempo apresentado pelo Firefox, os demais navegadores se mostraram bastante similares

quanto à eficiência de processamento. A Figura 25 apresenta o gráfico das informações acima listadas, tendo o tempo – em milissegundos – na vertical e a quantidade de corpos na horizontal.

Figura 25 – Gráfico do tempo em milissegundos do processamento de resposta de integração




Conforme esclarecido nesse capítulo, a detecção de colisão é o principal agente de impacto para a performance da simulação. O impacto é refletido diretamente na taxa de FPS, o que pode comprometer a fidedignidade da simulação.

A variação na taxa FPS é melhor percebida com corpos do tipo caixa, que possuem um custo operacional mais acentuado para a detecção de colisão. A diferença de custo em relação a detecção de colisão para corpos do tipo esfera - mesmo que somente colidam com corpos planos (solo) - é bastante perceptível, conforme pode-se observar na Figura 23.

### 3.4.3 Comparativo entre o trabalho desenvolvido e seus correlatos

Nessa seção será apresentada uma comparação entre as principais características do trabalho desenvolvido com as características identificadas dos trabalhos definidos como correlatos do mesmo (Quadro 19).

Quadro 19 – Comparação com os trabalhos correlatos

| Característica | Harbs (2013) | Silva (2012) | Algodoo (2013) | Hefesto[3] (2015) |
|---|---|---|---|---|
| Corpos do tipo polígono | X | - | X | - |
| Executa em dispositívos móveis | - | X | - | X* |
| Construção de cenas 3D | - | X | - | X |
| Disponível para acesso Web | X | - | - | X |
| Motor de física próprio | - | X | X* | X |
| Motor de física desacoplado | X | X | X* | X |
| Desenvolvido com HTML5 e JavaScript | X | - | - | X** |

As características marcadas com asterisco (*) não foram propriamente identificadas, porém acredita-se que os correlatos as tenham. Já a característica marcada com dois asteriscos (**), está assim marcada por atender parcialmente ao item. Independente da diferença de propósito dos trabalhos correlatos com o trabalho desenvolvido, o mesmo apresenta a maioria das características comparadas.

O trabalho de Harbs (2013) visa a construção de um motor de jogos em 2D para web. Uma de suas funcionalidades é a simulação física dos corpos, e isso é feito através da Box2DJS. A Box2DJS é um motor de física para cenas de duas dimensões construído em JavaScript que executa em navegadores, permitindo inclusive o uso de corpos do tipo polígono.

Silva (2012) construiu um motor de física em 3D na linguagem C++ que pode ser utilizado para construção de jogos e simulações. Não permite o uso de corpos do tipo polígono e pode ser utilizado somente em dispositivos móveis da Apple.

O Algodoo é uma aplicação que permite a construção de cenas 2D com elementos da física. Corpos rígidos, polígonos ou mesmo molas estão em seu acervo de recursos. Algodoo é uma aplicação da empresa Algoryx e - com seu motor proprietário, pode ser executado em Windows e Mac OS.

[3] Trabalho desenvolvido, sendo a união entre motor de física, biblioteca remota e cliente remoto.

# 4 CONCLUSÕES

Os objetivos principais deste trabalho, que eram a criação de um motor de física e uma biblioteca remota que permitisse o uso do motor remotamente, foram plenamente atendidos. O motor foi desenvolvido através da migração e refatoração do motor Cyclone – contendo as funcionalidades básicas para proporcionar uma experiência mínima de simulações utilizando as leis de Newton.

Para viabilizar a execução das funcionalidades do motor remotamente, criou-se uma biblioteca remota. A arquitetura utilizada na biblioteca remota - com base em comandos, mostrou-se eficiente, visto que conforme adicionava-se funcionalidades ao motor, era possível criar novos comandos sem grandes dificuldades ou impacto nos comandos já existentes. A execução da biblioteca se dá por meio de um servidor Apache Tomcat, que é de fácil instalação, permitindo que o ambiente de execução do motor seja facilmente montado. Ao fazer uso da tecnologia *WebSocket*, viabilizou-se a conexão por múltiplas plataformas. Sendo possível realizá-las simultaneamente através de um protocolo simples, via navegador (JavaScript), outras linguagens ou mesmo dispositivos móveis.

Para validar a estrutura empregada, desenvolveu-se um cliente para a biblioteca remota utilizando JavaScript, *WebSocket* e ThreeJS. JavaScript como linguagem nativa executada nos navegadores, *WebSocket* sendo a tecnologia utilizada para estabelecer a comunicação entre cliente e biblioteca remota, e o desenho das cenas através da biblioteca gráfica ThreeJS.

O uso do cliente desenvolvido – e as tecnologias nele empregadas – se mostraram bastante práticas e eficientes. Visto que o cliente foi utilizado por Zanluca (2015) no desenvolvimento de uma aplicação didática funcional, nos testes de performance e no desenvolvimento de aplicações modelo para experimentos deste trabalho.

A experiência relatada por Zanluca (2015), junto aos testes aplicados na estrutura criada, mostram que a mesma é bastante eficaz – desde que suas limitações sejam respeitadas. Segundo Zanluca (2015), há limitações na variedade de tipos de corpos possível de se utilizar, e também no modelo adotado para criação de colisões. Já os testes de performance mostram a sensibilidade do processamento em relação ao volume de colisões a serem detectadas e consequentemente resolvidas.

Apesar das limitações funcionais e capacitativas do trabalho desenvolvido, acredita-se na possibilidade de uso e evolução do mesmo. Também se crê em seu uso para construção de material educacional complementar que permita uma melhor experiência didática com a disciplina de física.

## 4.1 EXTENSÕES

São sugeridas as seguintes extensões para a continuidade do trabalho:

a) disponibilizar novos tipos de corpos no Motor de Física;
b) disponibilizar funcionalidades que permitam a colisão com novos corpos;
c) adicionar implementação de resistência do ar, através dos mecanismo de força desenvolvidos;
d) adicionar novas forças e propriedades físicas configuráveis;
e) criar funcionalidades para simulação das Leis de Kepler e o movimento planetário utilizando a estrutura criada no motor;
f) criar mecanismos que facilitem a descriminação de colisões;
g) criar mecanismo para obter os valores de um corpos rígidos específico durante a execução da simulação;
h) criar funcionalidades para simulação de mola e junções;
i) estudar a viabilidade do uso do Motor de Física em dispositivos Android;
j) viabilizar a exibição de trajetória de movimento do corpo;
k) viabilizar a utilização de *timeline* e restauração de estado ao executar uma simulação.

## REFERÊNCIAS

ALGODOO: **What is it?**. [S.l.], 2013. Disponível em: <http://www.algodoo.com/what-is-it/>. Acesso em: 25 mar. 2014.

AGUIAR, Carlos E. M. de.; GAMA, Eduardo A.; COSTA, Sandro Monteiro. **Física no ensino médio**. [S.l], 2005. Disponível em: <http://www.if.ufrj.br/~marta/aprendizagememfisica/RC-FISICA.pdf>. Acesso em: 6 abr. 2014.

APACHE TOMCAT. **Apache Tomcat**. [S.l.], 2015. Disponível em: <http://tomcat.apache.org/index.html>. Acesso em: 29 abr. 2015.

BALLISTIC. [S.l.], 2015. Disponível em: <http://www.inf.furb.br/gcg/visedu/fisica/ballistic>. Acesso em: 17 jun. 2015.

BARBOZA, Rui. **Entrevista sobre aplicação didática das leis de Hooke**. Entrevistadores: Thiago Henrique Teixeira, Gabriel Zanluca e Dalton Solano dos Reis. Blumenau. 2015. Entrevista feita através de conversação – não publicada.

BARRETO, Marcio. **Física**: Newton para o ensino médio. Campinas: Papirus, 2002.

BASTOS, Paulo M. S. O ensino de física na rede pública da Bahia. **Caderno de Física da UEFS**, Feira de Santana, v. 8, n. 1-2, p. 81-89, Jan./Dez. 2010. Disponível em: <http://dfis.uefs.br/caderno/vol8n12/PauloEnsino.pdf>. Acesso em: 30 mar. 2014.

BIRRELL, Andrew D.; NELSON, Bruce J. Implementing Remote Procedure Calls. **ACM Transactions on Computer Systems**, [S.l], v. 2, n. 1, p. 39-59, Feb. 1984. Disponível em: <http://www.cs.princeton.edu/courses/archive/fall03/cs518/papers/rpc.pdf>. Acesso em: 5 abr. 2014.

BOX2D JS. [S.l.], 2008. Disponível em: <http://box2d-js.sourceforge.net/>. Acesso em: 7 abr. 2014.

CANIUSE. Web Sockets. [S.l.], 2014. Disponível em: http://caniuse.com/#search=websocket. Acesso em: 13 mar. 2015.

CLINE, Michael B. **Rigid body simulation with contact and constraints**. 1999. 102 f. Thesis (Master of Science) - The Faculty of Graduate Studies, The University of Texas, Austin. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.186.5829&rep=rep1&type=pdf>. Acesso em: 5 abr. 2014.

FERREIRA, Andreia de A. O computador no processo de ensino-aprendizagem: da resistência a sedução. **Trabalho & Educação**, Brasília, v. 17, n. 2, p. 65-76, Maio/Ago. 2008. Disponível em: <http://www.portal.fae.ufmg.br/seer/index.php/trabedu/article/viewFile/330/299>. Acesso em: 6 abr. 2014.

HARBS, Marcos. **Motor para jogos 2d utilizando html5**. 2013. 77 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

LUBBERS, Peter. **HTML5 WebSocket**: full-duplex, real-time web Communication. [S.l.], 2011. Disponível em: <http://refcardz.dzone.com/refcardz/html5-websocket>. Acesso em: 09 mar. 2015.

MENEZES, Luis. C. **A matéria – Uma aventura do espírito**: Fundamentos e fronteiras do conhecimento físico. São Paulo: Editora Livraria da Física, 2005.

MILLINGTON, Ian. **Game physics**: engine development. San Francisco: Morgan Kaufmann, 2007.

OXFORD. **Oxford dictionaries**. [S.l.], [2014?]. Disponível em: <http://oxforddictionaries.com/>. Acesso em: 29 mar. 2014.

PEREIRA, Denis R. de O.; AGUIAR, Oderli. Ensino de física no nível médio: tópicos de física moderna e experimentação. **Revista Ponto de Vista**, Viçosa, v. 3, n. 1, p. 65-81, 2006. Disponível em: <http://www.coluni.ufv.br/revista/docs/volume03/ensinoFisica.pdf>. Acesso em: 29 mar. 2014.

PRATA, Stephen. **C++ Primer plus**. 6 ed. Crawfordsville: Addison-Wesley, 2012.

ROSA, Álvaro B. da; ROSA, Cleci W. da. Ensino de Física: objetivos e imposições no ensino médio. **Revista Electrónica de Enseñanza de las Ciencias**, [S.l], v. 4, n. 1, p. 1-18, 2005. Disponível em: <http://reec.uvigo.es/volumenes/volumen4/ART2_Vol4_N1.pdf>. Acesso em: 30 mar. 2014.

SCHROEDER, Craig. **Coupled simulation of deformable solids, rigid bodies, and fluids with surface tension**. 2011. 204 f. Dissertation (Doctor of Philosophy) - The Department of Computer Science and The Committee On Graduate Studies, Stanford University, Stanford. Disponível em: <http://hydra.math.ucla.edu/~craig/papers/2011-thesis/thesis.pdf>. Acesso em: 5 abr. 2014.

SILVA, Silvio F. da. **Protótipo de uma ferramenta gráfica para criação de simulações na área da física em dispositivos móveis**. 2012. 75 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SOUZA FILHO, Geraldo F. de. **Simuladores computacionais para o ensino de física básica**: uma discussão sobre produção e uso. 2010. 77 f. Dissertação (Mestrado Profissional em Ensino de Física) - Curso de Pós-Graduação em Ensino de Física, Universidade Federal do Rio de Janeiro, Rio de Janeiro. Disponível em: <http://www.if.ufrj.br/~pef/producao_academica/dissertacoes/2010_Geraldo_Felipe/dissertacao_Geraldo_Felipe.pdf>. Acesso em: 5 abr. 2014.

SUBLIME. **Sublime Text**. [S.l.], 2015. Disponível em: <http://www.sublimetext.com/ >. Acesso em: 29 abr. 2015.

UNITY3D: **Physics**. [S.l.], 2013. Disponível em: <https://docs.unity3d.com/Documentation/Manual/Physics.html>. Acesso em: 3 abr. 2014.

VISEDU. **VISEDU: Visualizador de Material Educacional**. [S.l.], 2015a. Disponível em: <http://www.inf.furb.br/gcg/visedu/ >. Acesso em: 18 jun. 2015.

_____. **Interface Editor com Server**. [S.l.], 2015b. Disponível em: <http://www.inf.furb.br/gcg/visedu/fisica >. Acesso em: 18 jun. 2015.

ZANLUCA, Gabriel. **Entrevista sobre experiência com o uso do motor**. Entrevistador: Thiago Henrique Teixeira. Blumenau. 2015. Entrevista feita através de conversação – não publicada.

## APÊNDICE A – Casos de usos do Motor de Física

A seguir, do Quadro 20 ao 27 são apresentados os casos de uso do Motor de Física.

Quadro 20 – Caso de uso UC01: Criar Corpo Rígido

| UC01 – Criar Corpo Rígido | |
|---|---|
| Descrição | Permite a criação e associação de um novo Corpo Rígido com o Motor de Física. Um Corpo Rígido é a representação de um corpo rígido para a Mecânica Clássica. |
| Cenário principal | 1. Um novo Corpo Rígido é instanciado;<br>2. Os valores do Corpo Rígido são informados;<br>3. O Corpo é associado ao Motor de Física. |
| Pós-condição | O novo Corpo Rígido está associado ao Motor de Física e é possível realizar o processo de integração com o mesmo. |

Quadro 21 – Caso de uso UC02: Criar Dados de Colisão

| UC02 – Criar Dados de Colisão | |
|---|---|
| Descrição | Permite a criação e associação de novos Dados de Colisão ao Motor de Física. Dados de Colisão são um conjunto de parâmetro (coeficiente de fricção restuição e tolerância) necessário para realizar a resolução de contatos de corpos baseado no seu material. |
| Cenário principal | 1. Um novos conjunto de Dados de Colisão é instanciado;<br>2. Os valores dos Dados de Colisão são informados;<br>3. O conjunto é associado ao Motor de Física. |
| Pós-condição | O novo conjunto de Dados de Colisão está criado e é possível criar novas Colisões. |

Quadro 22 – Caso de uso UC03: Criar Colisão

| UC03 – Criar Colisão | |
|---|---|
| Descrição | Permite a criação e associação de uma nova Colisão ao Motor de Física. Uma Colisão é a representação computacional dos dados necessário para realizar a detecção e resolução de contato entre dois corpos. |
| Pré-condição | Existir Dados de Colisão associados ao Motor de Física. |
| Cenário principal | 1. Uma nova Colisão é instanciada;<br>2. Os valores da Colisão são informados;<br>3. A Colisão é associada ao Motor de Física. |
| Pós-condição | A nova Colisão está criada e é possível detectar e resolver contato entre os corpos associados no processo de integração. |

Quadro 23 – Caso de uso UC04: Criar Força

| UC04 – Criar Força | |
|---|---|
| Descrição | Permite a criação e associação de uma nova Força ao Motor de Física. Uma Força é a representação computacional das equações matemáticas necessário para realizar a simulação e aplicação da resultante vetorial de uma força a um Corpo Rígido. |
| Cenário principal | 1. Uma nova Força é instanciada;<br>2. Os valores da Força são informados;<br>3. A Força é associada ao Motor de Física. |
| Pós-condição | A nova Força está criada e é possível realizar a aplicação da Força sob um corpo ao realizar o processo de integração. |

Quadro 24 – Caso de uso UC05: Calcular Integração

| UC05 – Calcular Integração | |
|---|---|
| Descrição | Permite realizar o processo de integração de uma simulação baseado no Princípio de d'Alambert. No processo de integração, deve-se realizar os cálculos de integração dos corpos, detectar e resolver colisões e aplicar as forças existentes. |
| Pré-condição | Que os itens do caso de uso 1 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. Calcula a aceleração linear a partir de entradas de força;<br>2. Calcular aceleração angular a partir das entradas de torque;<br>3. Ajuste velocidades, linear e angular de ambos aceleração e de impulso;<br>4. Impõe arrasto;<br>5. Ajuste posição linear e angular;<br>6. Normaliza a orientação, e atualizar as matrizes com a nova posição e orientação;<br>7. Atualizar o armazenamento de energia cinética. |
| Pós-condição | O processo de integração foi realizado e os dados e a posição do corpo estão atualizados. |
| Fluxo alternativo 1 | No passo 7, se o total do coeficienete de movimento do corpo for nulo o estado do mesmo é alterado para dormindo. |

Quadro 25 – Caso de uso UC06: Detectar Colisão

| UC06 – Detectar Colisão | |
|---|---|
| Descrição | Permite realizar a processo de detecção de colisão entre corpos. |
| Pré-condição | Que os itens dos casos de uso 1, 2 e 3 teham sua execução realizada com sucesso. |
| Cenário principal | 1. Realiza os processos de detecção baseado nos tipos dos corpos;<br>2. Armazenametos dos dados obtidos das colisões para posterior resolução. |
| Pós-condição | As colisões detectadas são armazenadas para posterior resolução. |

Quadro 26 – Caso de uso UC07: Tratar Colisões

| UC07 – Tratar Colisões | |
|---|---|
| Descrição | Permite realizar o processamento da resolução de colisão entre corpos. |
| Pré-condição | É necessário que colisões tenham sido detectadas e seus dados devem estar armazenados para coleta e resolução. |
| Cenário principal | 1. Realizar o processo de resolução de colisão baseado nos dados previamente obtidos;<br>2. Aplicar as tranformações geradas nos corpos envolvidos. |
| Pós-condição | A colisão está resolvida e os corpos envolvidos estão com os dados atualizados. |

Quadro 27 – Caso de uso UC08: Aplicar Forças

| UC08 – Aplicar Forças | |
|---|---|
| Descrição | Permite realiza o processamento das forças e aplica as resultantes aos corpos da simulação. |
| Pré-condição | Que os itens dos casos de uso 1 e 4 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. Realizar os calculos das forças;<br>2. Aplicar as resultantes ao corpos da simulação. |
| Pós-condição | As forças da simulação foram aplicadas aos corpos. |

## APÊNDICE B – Casos de usos da Biblioteca Remota

A seguir, do Quadro 28 ao 30 são apresentados os casos de uso da Biblioteca Remota.

Quadro 28 – Caso de uso UC09: Gerenciar conexões

| UC09 – Gerenciar conexões | |
|---|---|
| Descrição | Permite que a Biblioteca Remota realize o gerenciamento do ciclo de vida das conexões. |
| Cenário principal | 1. Ao receber uma nova conexão, gerar uma nova simulação associada a conexão e mantê-la;<br>2. Ao receber um sinal de quebra de conexão, eliminar todos os dados associados à mesma. |
| Pós-condição | A Biblioteca Remota manteve as conexões e seus dados integros. |
| Exceção 01 | Caso ocorre quebra de conexão por fatores externos à execução normal, é realizado os procedimentos do passo 2. |

Quadro 29 – Caso de uso UC10: Gerenciar Simulação

| UC10 – Gerenciar Simulação | |
|---|---|
| Descrição | Permite que a Biblioteca Remota realize o gerenciamento do ciclo de vida de uma simulação. |
| Pré-condição | Ter conexão estabelecida. |
| Cenário principal | 1. A Biblioteca Remota mantém os dados da simulação. |
| Pós-condição | A Biblioteca Remota manteve os dados da simulação. |

Quadro 30 – Caso de uso UC12: Enviar resultado da execução do comando

| UC12 – Enviar resultado da execução do comando | |
|---|---|
| Descrição | Permite que a Biblioteca Remota realize o envio do resultado do processamento do comando ao cliente. |
| Pré-condição | Que os itens do caso de uso 11 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. Recebe o resultado do processamento do comando;<br>2. Formata nova mensagem à ser enviada;<br>3. Envia mensagem para origem do comando com o resultado do processamento. |
| Pós-condição | A Biblioteca Remota enviou o resultado do processamento do comando à origem. |
| Fluxo alternativo 1 | No passo 3, caso ocorra erro no processamento do comando, é uma mensagem à origem com a mensagem de erro e a simulação é abortada. |

## APÊNDICE C – Casos de usos do Cliente Remoto

A seguir, do Quadro 31 ao 45 são apresentados os casos de uso do Cliente Remoto.

Quadro 31 – Caso de uso UC14: Manter conexão

| UC14 – Manter conexão | |
|---|---|
| Descrição | Permite que o Cliente Remoto mantenha a conexão com a Biblioteca Remota. |
| Pré-condição | É necessário que aja conexão entre Cliente e Biblioteca Remota. |
| Cenário principal | 1. O Cliente Remoto mantém a conexão e os dados da simulação. |
| Pós-condição | O Cliente Remoto manteve as conexões e seus dados integros. |

Quadro 32 – Caso de uso UC15: Enviar comando

| UC15 – Enviar comando | |
|---|---|
| Descrição | Permite que o Cliente Remoto envie um novo comando para a Biblioteca Remota |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. O Cliente Remoto recebe novo comando a ser enviado;<br>2. O Cliente Remoto formata nova mensagem à ser enviada;<br>3. O Cliente Remoto armazena mensagem à ser enviada;<br>4. O Cliente Remoto envia a mensagem para a Biblioteca Remota. |
| Pós-condição | O Cliente Remoto enviou a mensagem e libera a execução. |

Quadro 33 – Caso de uso UC16: Receber resultado do comando

| UC16 – Receber resultado do comando | |
|---|---|
| Descrição | Permite que o Cliente Remoto receba o resultado de um comando e o processe. |
| Pré-condição | Que os itens do caso de uso 15 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. O Cliente Remoto recebe nova mensagem;<br>2. O Cliente Remoro converte mensagem em resultado de comando;<br>3. O Cliente Remoto despacha resultado de comando para ser processado. |
| Pós-condição | O Cliente Remoto realizou o processamento do resultado do comando. |

Quadro 34 – Caso de uso UC17: Processar resultado integração

| UC17 – Processar resultado integração | |
|---|---|
| Descrição | Permite que a Simulação processe o resultado da integração. |
| Pré-condição | Que os itens do caso de uso 16 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. A Simulação recebe o comando de resultado da integração através do Cliente Remoto;<br>2. A Simulação aplica as novas matrizes de transformação e vetores de posição;<br>3. A Simulação notifica os contatos, caso aja;<br>4. A Simulação notifica os dados do corpo rígido, caso necessário. |
| Pós-condição | O Cliente Remoto processou o resultado da integração e a Simulação está atualizada. |

Quadro 35 – Caso de uso UC18: Enviar corpo rígido

| UC18 – Enviar corpo rígido | |
|---|---|
| Descrição | Permite o envio de um novo corpo rígido para a Biblioteca Remota. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. O Cliente Remoto recebe a nova instância de corpo rígido.<br>2. O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 36 – Caso de uso UC19: Remover corpo rígido

| UC19 – Remover corpo rígido | |
|---|---|
| Descrição | Permite que o Cliente Remoto remova um corpo rígido da Simulação. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. O Cliente Remoto recebe a instância do corpo rígido a ser removido;<br>2. O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 37 – Caso de uso UC20: Enviar força

| UC20 – Enviar força | |
|---|---|
| Descrição | Permite o envio de uma nova Força para a Biblioteca Remota. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. O Cliente Remoto recebe a nova instância de força.<br>2. O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 38 – Caso de uso UC21: Associar força ao corpo rígido

| UC21 – Associar força ao corpo rígido | |
|---|---|
| Descrição | Permite que uma força seja associada a um corpo rígido na Biblioteca Remota. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso. |
| Cenário principal | 1. O Cliente Remoto recebe a força e o corpo rígido.<br>2. O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 39 – Caso de uso UC22: Remover força

| UC22 – Remover força | |
|---|---|
| Descrição | Permite que uma força seja removida da Simulação na Biblioteca Remota. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso |
| Cenário principal | 1. O Cliente Remoto recebe a força.<br>2. O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 40 – Caso de uso UC23: Enviar colisão

| UC23 – Enviar colisão | |
|---|---|
| Descrição | Permite que uma colisão seja adicionada a Simulação na Biblioteca Remota. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso |
| Cenário principal | 1. O Cliente Remoto recebe a colisão.<br>2. O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 41 – Caso de uso UC24: Remover colisão

| UC24 – Remover colisão | |
|---|---|
| Descrição | Permite que uma colisão seja removida da Simulação na Biblioteca Remota. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso |
| Cenário principal | 1. O Cliente Remoto recebe a colisão.<br>2. O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 42 – Caso de uso UC25: Enviar dados de colisão

| UC25 – Enviar dados de colisão | |
|---|---|
| Descrição | Permite que novos dados de colisão sejam adicionados a Simulação na Biblioteca Remota. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso |
| Cenário principal | 1. O Cliente Remoto recebe os dados de colisão.<br>2. O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 43 – Caso de uso UC26: Alterar estado colisão

| UC26 – Alterar estado colisão | |
|---|---|
| Descrição | Permite que o estado de colisão seja alterado na Simulação na Biblioteca Remota. Permitindo assim, ligar ou desligar a verificação de colisão para os corpos envolvidos. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso |
| Cenário principal | 1. O Cliente Remoto recebe a colisão.<br>2. O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 44 – Caso de uso UC27: Integrar simulação

| UC27 – Integrar simulação | |
|---|---|
| Descrição | Permite que a integração da Simulação seja executada na Biblioteca Remota. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso |
| Cenário principal | O Cliente Remoto recebe a duração do último quadro.<br>O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

Quadro 45 – Caso de uso UC28: Obter dados de corpo rígido

| UC28 – Obter dados de corpo rígido | |
|---|---|
| Descrição | Permite que seja obtido os dados de um corpo através da Biblioteca Remota. |
| Pré-condição | Que os itens do caso de uso 14 tenham sua execução realizada com sucesso |
| Cenário principal | O Cliente Remoto recebe o corpo ao qual se deseja obter os dados.<br>O Cliente Remoto monta e envia o comando para a Biblioteca Remota. |
| Pós-condição | O comando é enviado. |

## APÊNDICE D – Exemplos de comandos trocados entre Cliente e Biblioteca Remota

No Quadros 46 a 57 são apresentados exemplos dos comandos trocados entre Cliente e Biblioteca Remota. Sendo os comandos recebidos iniciados por RCV e os enviado por SND.

Quadro 46 – Exemplo resultado comando Nova Simulação

```
RCV: {"id":-1, "type":"NEW_SIMULATION", "data":{"id":1434336385384}}
```

Quadro 47 – Exemplo comando Adicionar Dados de Colisão

```
SND: { "id":0,
       "type":"BIND_COLLISION_DATA",
       "data":{"id":"3ce67941-1c2a-7300-a6cd-6f66c1d551b5",
                "name":"3ce67941-1c2a-7300-a6cd-6f66c1d551b5",
                "friction":0.9, "restitution":0.9, "tolerance":0.01,
                "maxContacts":65536, "maxCollision":65536}}

RCV: { "id":0, "type":"BIND_COLLISION_DATA",
       "data":{"id":"3ce67941-1c2a-7300-a6cd-6f66c1d551b5", "added":true}}
```

Quadro 48 – Exemplo comando Adicionar Corpo Rígido

```
SND: { "id":1,
       "type":"BIND_RIGID_BODY",
       "data":{"id":"b5699214-de71-584b-ac76-b92a9be6c408",
              "position":{"x":0,"y":5,"z":0},
              "orientation":{"_x":0,"_y":0,"_z":0,"_w":1},
              "velocity":{"x":0,"y":0,"z":0},
              "acceleration":{"x":0,"y":0,"z":0},
              "rotation":{"x":0,"y":0,"z":0},
              "mass":200,
                    "inertiaTensor":{"elements":{"0":5360957,
      "1":0,"2":0,"3":0,"4":5360957,"5":0,"6":0,"7":0,"8":5360957}},
              "linearDamping":0.95, "angularDamping":0.8,
              "canSleep":true,
"matrix":{"elements":{"0":1,"1":0,"2":0,"3":0,"4":0,"5":1,"6":0,"7":0,
      "8":0,"9":0,"10":1,"11":0,"12":0,"13":0,"14":0,"15":1}},
              "radius":5,
              "halfSize":{"x":0,"y":0,"z":0},
              "ignoreIntegration":false,
              "bindContactData":true,
              "useWorldForces":false}}

RCV: { "id":1, "type":"BIND_RIGID_BODY",
       "data":{"id":"b5699214-de71-584b-ac76-b92a9be6c408", "added":true}}
```

Quadro 49 – Exemplo comando Adicionar Colisão

```
SND: { "id":2,
       "type":"BIND_COLLISION",
       "data":{"id":"10464fbb-4cf8-5248-e70a-6c821778cfbc",
              "type":"SPHERE_AND_TRUEPLANE",
              "data":"3ce67941-1c2a-7300-a6cd-6f66c1d551b5",
              "body1":"b5699214-de71-584b-ac76-b92a9be6c408",
              "body2":null, "enable":true}}

RCV: { "id":2, "type":"BIND_COLLISION",
       "data":{"id":"10464fbb-4cf8-5248-e70a-6c821778cfbc", "added":true}}
```

Quadro 50 – Exemplo comando Adicionar Força

```
SND: { "id":3,
       "type":"BIND_FORCE",
       "data":{"id":"25203151-ba35-3f68-8afc-9d7ad3825ace",
              "name":"Force", "type":"GRAVITY","gravity":{"x":0,"y":-10,"z":0}}}

RCV: { "id":3, "type":"BIND_FORCE", "data":{"added":true}}
```

Quadro 51 – Exemplo comando Adicionar Força ao Corpo

```
SND: { "id":4,
       "type":"ADD_FORCE_TO_BODY",
       "data":{"force":"25203151-ba35-3f68-8afc-9d7ad3825ace",
               "body":"b5699214-de71-584b-ac76-b92a9be6c408"}}

RCV: { "id":4, "type":"ADD_FORCE_TO_BODY", "data":{"added":true}}
```

Quadro 52 – Exemplo comando Integrar

```
SND: { "id":5, "type":"INTEGRATE", "data":{"duration":0.017}}



      RCV: { "id":5,
       "type":"INTEGRATE",
       "data":{"_rigidBodys":[{"id":"b5699214-de71-584b-ac76-b92a9be6c408",
                               "position":{"z":0,"y":5,"x":0},
                               "transform":[1,0,0,0,0,1,0,0,0,0,1,0,0,5,0,1]}],
              "_contacts":[{"penetration":0,
                            "friction":0.9,
                            "contactPoint":{"z":0,"y":0,"x":0},
                            "contactNormal":{"z":0,"y":1,"x":0},
                            "body1":"b5699214-de71-584b-ac76-b92a9be6c408",
                            "restitution":0.9,
                            "contactToWorld":[0,0,1,1,-0,-0,0,1,-0]}]}}
```

Quadro 53 – Exemplo comando Remover Força

```
SND: { "id":6, "type":"REMOVE_FORCE",
       "data":{"id":"25203151-ba35-3f68-8afc-9d7ad3825ace"}}

RCV: { "id":6, "type":"REMOVE_FORCE", "data":{"removed":true}}
```

Quadro 54 – Exemplo comando Mudar Estado da Colisão

```
SND: { "id":7, "type":"CHANGE_COLLISION_STATE",
       "data":{"id":"10464fbb-4cf8-5248-e70a-6c821778cfbc", "state":false}}

RCV: { "id":7, "type":"CHANGE_COLLISION_STATE",
       "data":{"id":"10464fbb-4cf8-5248-e70a-6c821778cfbc","enabled":false}}
```

Quadro 55 – Exemplo comando Remover Colisão

```
SND: { "id":8, "type":"REMOVE_COLLISION",
       "data":{"id":"10464fbb-4cf8-5248-e70a-6c821778cfbc"}}

RCV: { "id":8, "type":"REMOVE_COLLISION", "data":{"removed":true}}
```

Quadro 56 – Exemplo comando Obter Dados de Corpo Rigido

```
SND: { "id":9, "type":"GET_RIGID_BODY_DATA",
       "data":{"body":"b5699214-de71-584b-ac76-b92a9be6c408"}}

RCV: { "id":9,
       "type":"GET_RIGID_BODY_DATA",
       "data":{"bodys":[
             {"id":"b5699214-de71-584b-ac76-b92a9be6c408",
              "position":{"z":0,"y":5,"x":0},
       "orientation":{"_w":1,"_x":0,"_y":0,"_z":0},
       "rotation":{"z":0,"y":0,"x":0},
       "aceleration":{"z":0,"y":0,"x":0},
       "velocity":{"z":0,"y":0,"x":0},
       "inverseInertiaTensor":[1.865338595329155E-7,-0,0,-0,1.865338595329155E-
7,-0,0,-0,1.865338595329155E-7]}]}}
```

Quadro 57 – Exemplo comando Remover Corpo Rígido

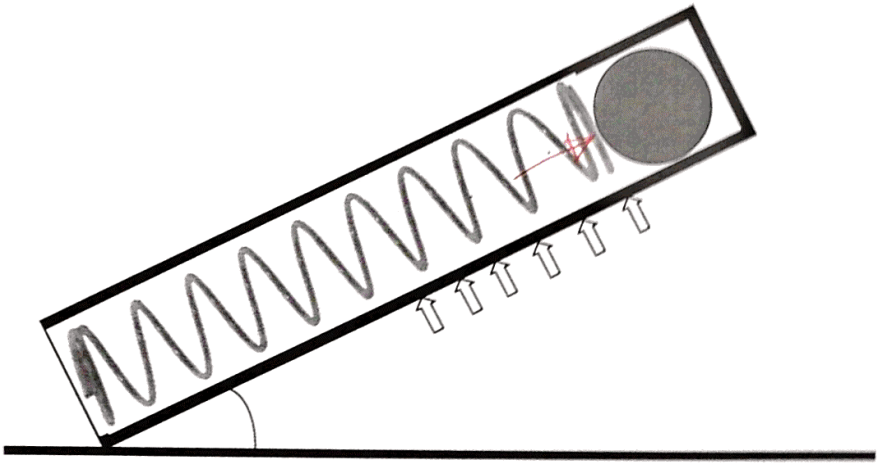
```
SND: { "id":10, "type":"REMOVE_RIGID_BODY",
       "data":{"id":"b5699214-de71-584b-ac76-b92a9be6c408"}}

RCV: { "id":10, "type":"REMOVE_RIGID_BODY",
      "data":{"id":"b5699214-de71-584b-ac76-b92a9be6c408","removed":true}}
```

## ANEXO A – Lei de Hooke

Este anexo contém informações decorrentes da entrevista feita com o professor de física Barboza (2015). As Figuras 26 e 27 apresentam as explicações do professor sobre a lei de Hooke e lançamento oblíquo.

Figura 26 – Lei de Hooke (pg. 1)

**Disparo do canhão:**



**Densidade dos projéteis:** $\mu = \frac{m}{v}$ ⟶ $m = \mu \cdot V$

- Materiais: (variável)
  1. Borracha silicone ..... $\mu = 0{,}95\ g/cm^3$
  2. Alumínio ..... $\mu = 2{,}70\ g/cm^3$
  3. Ferro ..... $\mu = 7{,}87\ g/cm^3$
  4. Ouro ..... $\mu = 19{,}28\ g/cm^3$
  5. Vidro comum ..... $\mu = 2{,}60\ g/cm^3$
  6. Madeira seca (cedro) ..... $\mu = 0{,}485\ g/cm^3$
  7. Madeira seca (ipê) ..... $\mu = 1{,}10\ g/cm^3$

**Lei de Hooke - Força Elástica (mola):** $F_{el} = K \cdot x$

Onde:

x é a deformação da mola (variável), K é a constante elástica da mola (variável) e $F_{el}$ é a força elástica.

Figura 27 – Lei de Hooke (pg. 2)

**2ª Lei de Newton:** $F_R = m \cdot a$

- Onde:
  **m** é a massa do projétil (esfera), **a** é a aceleração desenvolvida no impulso pela mola e $F_R$ é a resultante das forças (considerar apenas a $F_{el}$)

**Velocidade de lançamento:** $V^2 = 2 \cdot a \cdot x \quad \rightarrow \quad V = \sqrt{2 \cdot a \cdot x}$

- Onde:
  **a** é aceleração no interior do canhão, **x** é o deslocamento do projétil durante o impulso pela mola (deformação elástica) e **V** é velocidade de lançamento do projétil.

**Lançamento oblíquo:**

- **Alcance** (distância horizontal) → $d = V \cdot \cos\theta \cdot t$, onde: d é a distância horizontal, V é a velocidade de lançamento, θ é o ângulo de inclinação do canhão e **t** é o instante (tempo).
- **Posição vertical (altura)** → $h = h_o + V \cdot sen\,\theta \cdot t - \frac{g}{2} \cdot t^2$
- **Equação da Trajetória - Posição do projétil no plano vertical (h) em função da distância horizontal (d)** → $h = h_o + d \cdot tg\,\theta - \frac{g}{2} \cdot \frac{d^2}{(V \cdot cos\,\theta)^2}$, onde: **h** é a altura em qualquer instante, $\mathbf{h_o}$ é a altura incial de lançamento, **d** é a distância horizontal, **tg θ** é a tangente do ângulo de inclinação do canhão, **g** é aceleração da gravidade e **cos θ** é o cosseno do ângulo θ.