UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

SIMULAÇÃO DE FÍSICA CLIENT-SIDE APLICADA A SIMULAÇÃO DE PROJÉTEIS

SÉRGIO LUIZ TOMIO JUNIOR

BLUMENAU
2017

SÉRGIO LUIZ TOMIO JUNIOR

SIMULAÇÃO DE FÍSICA CLIENT-SIDE APLICADA A SIMULAÇÃO DE PROJÉTEIS

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(a). Dalton Solano dos Reis, M.Sc. - Orientador

BLUMENAU
2017

SIMULAÇÃO DE FÍSICA CLIENT-SIDE APLICADA A SIMULAÇÃO DE PROJÉTEIS

Por

SÉRGIO LUIZ TOMIO JUNIOR

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 Reis, M.Sc – Orientador, FURB
Membro:	Prof. Gilvan Justino, M.Sc – FURB
Membro:	Prof. Robson Zacarelli Denke, Doutor – FURB

Blumenau, 04 de julho de 2017

Dedico este trabalho a minha família, amigos e colegas de trabalho que me ajudaram durante o desenvolvimento do mesmo.

AGRADECIMENTOS

À minha família, por me aturar durante todo o período de desenvolvimento desse trabalho.

Aos meus amigos, por terem me ouvido e ajudado durante todo o desenvolvimento.

Ao meu orientador, por todo o apoio e incentivo durante este trabalho.

Conhecimento não é aquilo que você sabe, mas o que você faz com aquilo que você sabe.

Aldous Huxley

RESUMO

Este trabalho apresenta o desenvolvimento de uma melhoria no motor de física Hefesto (TEIXEIRA, 2015). Este motor foi implementado de forma que a simulação de física era executada em um servidor web separado da aplicação cliente. Essa arquitetura posteriormente apresentou problemas de latência e dependência de uma conexão de rede estável para executar qualquer simulação. Para resolver esses problemas a simulação de física foi movida para o *client-side* junto com a aplicação cliente do motor. Usando as rotinas de integração de objetos do motor de física *open source* CannonJS (HEDMAN, 2013a) em conjunto com as rotinas de detecção e tratamento de colisões do Hefesto, convertidas para Javascript, foi possível remover a necessidade de um servidor web para executar as simulações de física. As aplicações web Ballistic (ZANLUCA, 2015) e o editor Hefesto (TEIXEIRA, 2015) foram adaptadas para utilizar o novo motor e usadas para realizar os testes de performance. Estes apresentaram resultados superiores ao motor antigo tanto em testes validando tempo de processamento nos ciclos de integração quanto consumo de memória RAM durante as simulações.

Palavras-chave: Simulação. Física. Web. *Client-side*.

ABSTRACT

This paper presents the development of an improvement in the Hefesto physics engine (TEIXEIRA, 2015). This engine was implemented in a way that the physics simulations was executed in a server separated from the client application, this architecture later presented latency issues and a dependency with a stable network connection to execute any simulation. To solve these problems the physics simulation has been moved to the client-side together with the client application of the engine. Using the object integration routines from the open source physics engine CannonJS (HEDMAN, 2013a) together with the collision detection and treatment routines from Hefesto, converted to Javascript, it was possible to remove the necessity of a web server to run the physics simulations. The web application Ballistic (ZANLUCA, 2015) and the Hefesto editor (TEIXEIRA, 2015) were adapted to use the new engine and were used to execute the performance tests, those presented superior results when compared to the previous engine both in tests validating the processing time of the integration cycles and in RAM memory usage during the simulations.

Key-words: Simulation. Physics. Web. Client-side.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Segundo Carneiro (2007, p. 35), a metodologia utilizada no ensino de física no Brasil, o ensino intensivo de fórmulas matemáticas desvinculadas da realidade e o processo repetitivo de memorização de conceitos tornam a disciplina de física algo muito distante do mundo vivido tanto pelos professores quanto pelos alunos dessa disciplina. No ano 1999, foram publicados os Parâmetros Curriculares Nacionais para o Ensino Médio (PCNEM) que tem como conteúdo "[...] uma explicitação das habilidades básicas, das competências específicas, que se espera que sejam desenvolvidas pelos alunos em Biologia, Física, Química e Matemática [...]" (BRASIL, 1999, p. 4). Uma das competências esperadas na área de física é a de "Compreender a Física presente no mundo vivencial e nos equipamentos e procedimentos tecnológicos. Descobrir o 'como funciona' de aparelhos." (BRASIL, 1999, p. 29).

Para Carneiro (2007, p. 39), "A compreensão dos conhecimentos físicos deve ser desenvolvida por passos, onde os elementos devem ser práticos, próximos da realidade dos alunos", ou seja, os alunos devem aprender conceitos físicos por meio do uso de experimentos palpáveis em situações reais que podem ser vivídas por eles. Um dos métodos citados por Carneiro (2007, p. 47) para estimular a aprendizagem dos alunos durante esses experimentos são as "simulações computacionais", que são softwares desenvolvidos com o intuito de emular algum comportamento observado no mundo real.

Uma simulação computacional para demonstrar o funcionamento da Lei de Hooke foi desenvolvida por Zanluca (2015). Essa simulação levou o nome de Ballistic. Para desenvolver essa simulação o autor utilizou uma biblioteca para simulação de física chamada Hefesto (TEIXEIRA, 2015). Essa biblioteca tem a característica de realizar a simulação de física no lado do servidor e abrir a comunicação para o lado cliente (browser) usando um cliente remoto. Essa abordagem funcionou mas apresentou o problema de ter uma grande dependência entre o cliente e o servidor, o que quer dizer que se existirem problemas de conexão entre um e outro ou se o servidor parar de funcionar por algum motivo a simulação também deixa de funcionar (REIS, 2016).

Diante desse contexto, o objetivo deste trabalho é adaptar uma biblioteca de física desenvolvida em Javascript para funcionar com a mesma interface do Hefesto (TEIXEIRA, 2015). Além disso também foi ajustada a aplicação web Ballistic (ZANLUCA, 2015) para passar a usar o motor de física adaptado.

1.1 OBJETIVOS

O objetivo deste trabalho é construir um adaptador entre a interface do motor de física CannonJS (HEDMAN, 2013a) e a interface do motor de física Hefesto (TEIXEIRA, 2015).

Os objetivos específicos são:

a) integrar o motor adaptado com a aplicação web Ballistic (ZANLUCA, 2015);
b) comparar a performance do motor adaptado com o motor Hefesto, desenvolvido em Java.

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, assim como uma explicação da arquitetura da biblioteca Hefesto (TEIXEIRA, 2015) atual. O capítulo 3 apresenta o desenvolvimento e adaptação do motor de física 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 conceitos de mecânica clássica. As seções 2.2 e 2.3 tratam sobre os principais conceitos relacionados a simulação de física e desenvolviemento web, respectivamente. A seção 2.4 oferece uma visão geral sobre padrões de projeto e aprofunda os padrões que podem ser melhor utilizados no desenvolvimento desse projeto. A seção 2.5 descreve os principais conceitos relacionados ao motor de física CannonJS, utilizado nesse trabalho. Por fim a seção 2.6 descreve os principais trabalhos correlatos identificados.

2.1 MECÂNICA CLÁSSICA

A mecânica clássica, ou mecânica newtoniana, recebe este nome por ter sido desenvolvida por Isaac Newton, por meio dela se define “A relação que existe entre uma força e aceleração produzida por essa força [...]” (HALLIDAY et al., 2012, p. 91). É uma teoria que pode ser aplicada ao estudo dos mais diversos objetos, desde corpos muito pequenos, até corpos muito grandes.

Essa teoria define três leis básicas que regem a forma como objetos se movimentam. A primeira lei é conhecida como princípio da inércia. A segunda é conhecida como princípio fundamental da dinâmica e a terceira é conhecida como princípio da ação e reação. Cada lei é apresentada individualmente nos próximos parágrafos.

De acordo com Halliday et al. (2012, p. 92) a primeira lei de Newton define que “Se nenhuma força atua sobre um corpo, sua velocidade não pode mudar, ou seja, o corpo não pode sofrer uma aceleração”. Em outras palavras, um objeto em repouso (inerte) continuará em repouso, e um objeto em movimento continuará em movimento até que uma força seja aplicada sobre ele.

Já a segunda lei de Newton define que “A força resultante que age sobre um corpo é igual ao produto da massa do corpo pela aceleração” (HALLIDAY et al., 2012, p. 95). Essa lei é melhor explicada com um exemplo. Considerar um jogador de futebol chutando uma bola de futebol normal com uma força de intensidade “y”. A bola será arremessada a uma certa distância, ao trocar a bola de futebol por uma bola de boliche e o jogador chutar a bola de boliche com a mesma força “y” ela será arremessada a uma distância menor, já que sua massa é maior que a da bola de futebol.

A terceira lei de Newton define que “Quando dois corpos interagem, as forças que cada corpo exerce um sobre o outro são iguais em módulo e têm sentidos opostos .” (HALLIDAY et al., 2012, p. 102). Um exemplo é a força que é aplicada pelas pessoas

constantemente sobre o solo, o peso de uma pessoa cria uma força aplicada sobre o solo e o solo faz uma força oposta para manter as pessoas em equilíbrio.

Além das leis básicas de Newton também foi realizado um estudo sobre a lei de Hooke, que é o princípio da aplicação Ballistic (ZANLUCA, 2015). A lei de Hooke determina qual a força elástica de uma mola associada a sua deformação. A força necessária para essa mola ir de um estado alongado ou comprimido de volta para o estado relaxado (HALLIDAY et al., 2012, p. 154). A equação da lei de Hooke é apresentada no Quadro 1.

Quadro 1 - Equação da Lei de Hooke

$$F = -kX$$

Fonte: Halliday et al. (2012).

A equação define que a força elástica F de uma mola é igual ao inverso do produto de uma constante elástica (`k`) e do deslocamento da mola com relação ao seu estado relaxado (`X`). A constante elástica é um valor que define o quanto uma determinada mola é rígida. Quanto maior o valor de `k` mais rígida esta mola é. O valor da deformação `X` é 0 quando a mola está no seu estado relaxado e aumenta conforme a mola é alongada ou diminui conforme a mola é comprimida.

2.2 SIMULAÇÃO DE FÍSICA

De acordo com Millington (2010, p. 3), um motor de física é um pedaço de código que conhece sobre física em geral, mas não é programado especificamente para nenhuma aplicação. Também de acordo com Millington, os motores de física são grandes calculadoras que realizam as operações matemáticas necessárias para simular a física dos objetos em um determinado “mundo”. Para que um motor de física realize a simulação de um objeto ele precisa saber as propriedades desse objeto, tais como posição, velocidade, aceleração, massa, etc. (MILLINGTON, 2010, p. 48-52)

Millington (2010, p. 17) faz uma revisão do que ele chama dos blocos de construção fundamentais dos motores de física: o cálculo e matemática de vetores 3D. Para Millington (2010, p. 18) vetores representam o conceito matemático de uma estrutura que guarda valores que são usados para realizar operações. Os vetores podem representar várias características do espaço, como por exemplo posição, velocidade, direção, aceleração, etc.

Além de operações com vetores, Millington (2010, p. 38) também reforça a importância do estudo sobre como valores mudam ao longo do tempo, pois é necessário para compreender situações, tal como a implementação da mudança de posição de um objeto ao longo de um período de tempo, ou a força ou velocidade de uma mola. Existem duas formas

de realizar essa análise, usando o cálculo diferencial e/ou o cálculo integral. O cálculo diferencial descreve e analisa como uma alteração ocorre, define quais fatores estão influenciando um determinado objeto para que ele esteja se comportando de alguma forma (aceleração, força, velocidade, etc.). Já o cálculo integral ignora os fatores envolvidos em um comportamento e define qual é o resultado desse comportamento (posição final de um determinado objeto que se deslocou no mundo).

Esses são os pilares da implementação dos motores de física.

2.3 DESENVOLVIMENTO WEB

Nesta seção será apresentada uma revisão sobre as três principais linguagens que serão usadas para o desenvolvimento desse projeto. Nos próximos parágrafos serão discutidas as linguagens HTML, CSS e Javascript.

De acordo com Henick (2010, p. 7), documentos HTML contém grande parte do conteúdo que usuários interagem na web. Esses documentos apresentam a estrutura de um site e conectam os usuários a qualquer tipo de conteúdo como scripts, imagens, vídeos, sons, etc. Esses documentos são estruturados com marcações, também conhecidas como `tags`. Essas `tags` podem conter atributos que modificam tamanho, posicionamento, cor, etc. Como apontado por Henick (2010, p. 7) estruturas HTML podem comportar diferentes camadas de estilo e comportamento. O estilo de uma página é definido pela segunda linguagem revisada, CSS.

Henick (2010, p. 18) faz uma analogia da interação entre HTML e CSS com a construção de uma casa. O HTML define a estrutura da casa, suas paredes, chão e telhado. Já o CSS começa a atuar depois que a estrutura está montada, adicionando coisas como quadros, tapetes e pintura. Em termos práticos, o CSS adiciona estilo a estrutura de um website, permitindo que o programador utilize `selectors` para selecionar que elementos do site serão afetados pelos estilos definidos. Estilos são definidos por pares chave/valor de atributos CSS. Com a estrutura e estilo do site montados, o último passo é definir o comportamento, trabalho realizado pela última linguagem revisada, o Javascript.

Javascript é a única linguagem implementada e suportada em todos os browsers usados hoje em dia, sendo a linguagem de programação usada para implementar comportamento em páginas web. De acordo com Crockford (2008, p. 3) Javascript tem muito em comum com linguagens funcionais como Lisp e Scheme. É uma linguagem de scripting, com tipagem dinâmica e um sistema de herança prototipal onde um objeto herda propriedades diretamente de outros objetos, sem um sistema de classes explícito.

2.4 PADRÕES DE PROJETO

Este trabalho visa adaptar um motor de física que segue uma determinada interface para que ele possa trabalhar com uma interface diferente. Para permitir que esse trabalho seja realizado com maior facilidade e para melhorar o código desenvolvido será apresentado um panorama sobre padrões de projeto e uma explicação mais focada nos padrões que poderão ser utilizados no projeto.

De acordo com Gamma et al. (2000, p. 19), um padrão de projeto descreve um problema e o princípio de sua solução, de forma que essa solução possa ser reutilizada um milhão de vezes sem nunca fazê-la da mesma forma. Os padrões de projeto têm quatro elementos essenciais, o nome do padrão, o problema onde o padrão deve ser aplicado, a solução ao problema e as consequências em utilizá-lo. O nome é importante para descrever brevemente o problema, possível solução e consequências de usar um padrão. O problema descreve em detalhes em que situação um padrão deve ser usado, o contexto e possivelmente uma lista de condições para viabilizar a aplicação do mesmo. A solução descreve uma visão em alto nível de como resolver o problema. Essa solução é reaplicável para resolver problemas em projetos diferentes e não tem um domínio de aplicação específico. Por fim as consequências são as vantagens e desvantagens de aplicar esse padrão. Nos parágrafos seguintes serão descritos em mais detalhes os padrões `adapter` e `abstract factory`, e onde eles podem ser usados na implementação do projeto.

O padrão `adapter` (GAMMA et al., 2000, p. 140) tem a intenção de converter a interface de uma classe para a interface de outra. Esse problema pode aparecer quando em um projeto existir uma classe que espera, por exemplo, um motor de física que segue uma interface, e o desenvolvedor desse projeto quiser usar um motor de física já existente mas que não segue essa interface. Essa é a situação atual desse projeto e é por este motivo que o `adapter` é um padrão que se encaixa como uma possível solução. Para solucionar esse problema Gamma et al. (2000, p. 142) apresentam duas opções, uma delas consiste em utilizar herança múltipla no objeto adaptador para permitir que ele repasse chamadas da interface original para a interface adaptada e a outra envolve utilizar composição para passar ao adaptador uma instância do objeto que segue a interface adaptada e repassar as chamadas da interface original para esse objeto. Como o adaptador para este projeto será construído em Javascript a única opção possível é a segunda, já que a linguagem Javascript não tem uma implementação de herança múltipla, que seria necessária para a primeira opção.

Sobre o padrão `abstract factory`, sua intenção é “Fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.” (GAMMA et al., 2000, p. 95). No projeto será ajustada a aplicação Ballistic (ZANLUCA, 2015) para deixar de utilizar o motor de física Hefesto (TEIXEIRA, 2015) e passar a utilizar o motor de física adaptado. Como esses dois motores seguirão a mesma interface, utilizando apenas um `adapter` para realizar as chamadas do CannonJS na mesma interface do Hefesto, é possível viabilizar a utilização dos dois motores trocando entre um e outro com a alteração de um parâmetro utilizando uma factory. A única alteração nas aplicações que usam o Hefesto é no momento de instânciar o motor de física, onde deve ser ajustado para passar a usar a factory construída ao invés de instanciar um dos dois motores manualmente.

2.5 CANNONJS

CannonJS é uma biblioteca que simula corpos rígidos e pode ser usada, por exemplo, para o desenvolvimento de jogos. Programadores podem usá-la para fazer os objetos de seus jogos se movimentarem e interagirem de forma realística (HEDMAN, 2013a).

O CannonJS é um motor para simulação de física com objetos 3D. Por ser um motor de física que está no mercado há alguns anos e também trabalhar com física 3D como o Hefesto (TEIXEIRA, 2015) ele foi selecionado para ser o motor adaptado para a interface do Hefesto. Na wiki do projeto no Github Hedman (2013b) apresenta um “Hello World” do CannonJS que explica alguns conceitos importantes para o entendimento do motor. Esses conceitos são o *time step*, as *constraints* e o *constraint solver*.

Um *time step* para um motor de física é um passo da simulação no mundo, esse passo simula uma certa quantidade de tempo determinada no exemplo de Hedman pelo parâmetro `timeStep` informado para o método `step`. Esse “passo” irá atualizar propriedades dos objetos como velocidade e posição de acordo com as forças aplicadas sobre esses objetos. Além dessa atualização também é realizado o tratamento das colisões que ocorrem no mundo durante esse período.

Uma *constraint* é uma “restrição” que é colocada sobre um objeto, limitando os seus graus de liberdade. Hedman (2013b) cita em seu exemplo as dobradiças de uma porta que limitam o seu movimento em um único eixo. O *constraint solver* é a entidade dentro do motor de física que irá iterar todas as *constraints* do mundo e resolvê-las. Hedman (2013b) aponta também que a resolução de uma *constraint* pode afetar outras *constraints*, então é necessário definir um número de vezes que o *constraint solver* irá iterar as *constraints* do mundo. Nesse

contexto, menos iterações querem dizer uma simulação menos precisa, porém mais performática. Mais iterações querem dizer uma simulação mais precisa e menos performática.

2.6 TRABALHOS CORRELATOS

Serão apresentados a seguir trabalhos com características semelhantes ao trabalho proposto. O primeiro é um motor de física também desenvolvido em Javascript chamado MatterJS (BRUMMIT, 2014a). O segundo é um projeto que combina simulação de física com realidade aumentada chamado SimulAR (FERREIRA et al., 2011) e o terceiro é um protótipo de uma ferramenta para geração de simulações de física em dispositivos móveis (SILVA, 2012).

2.6.1 MatterJS

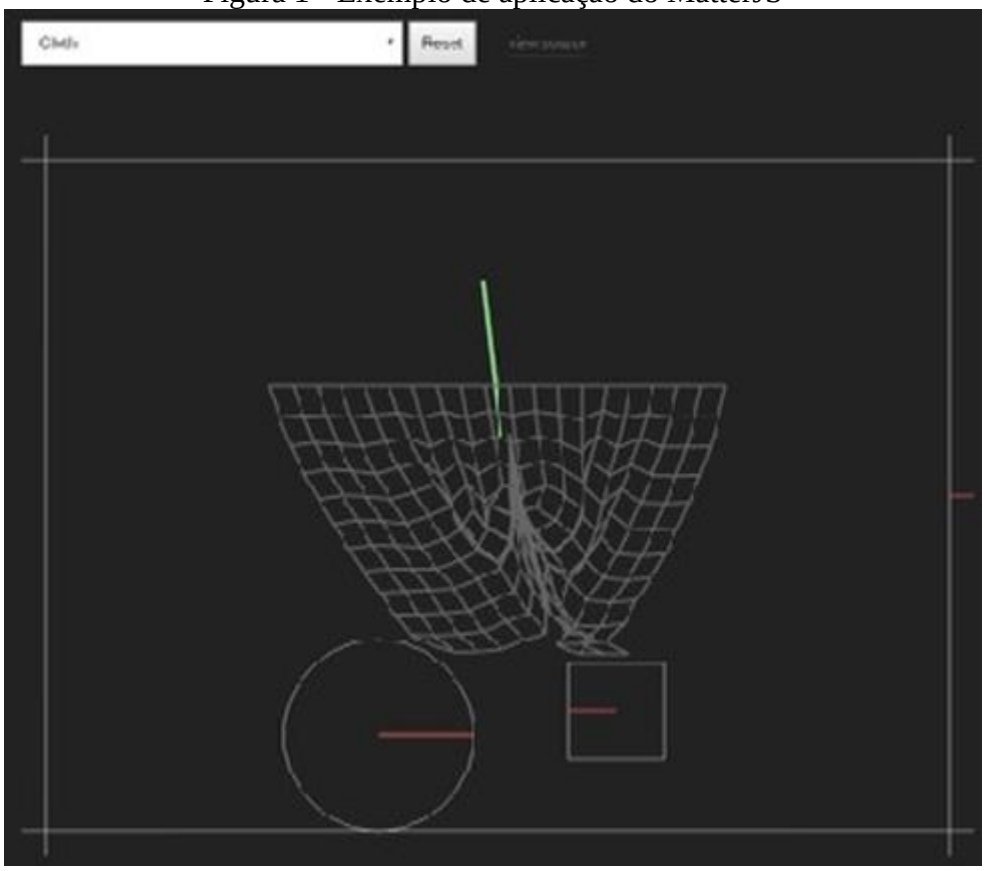
O motor de física MatterJS é um motor 2D desenvolvido para web que permite a simulação de vários tipos de objetos como corpos rígidos, compostos, coberturas côncavas e convexas, assim como a simulação de colisões com controle de propriedades físicas como a fricção, gravidade e elasticidade de objetos (BRUMMIT, 2014a).

A maior diferença do MatterJS para o motor proposto neste trabalho é que o MatterJS só consegue realizar a simulação de corpos 2D, enquanto o motor proposto será uma adaptação do CannonJS, que por si só já é um motor 3D.

Outra característica do MatterJS é ter a capacidade de simular muitos tipos de objetos e propriedades. O MatterJS foi concebido em Javascript, diferente de alguns outros projetos que surgiram a partir da conversão de outras linguagens de programação para o Javascript (YASUSHI, 2008). Além disso o MatterJS ainda disponibiliza uma série de exemplos no site do projeto. Um desses exemplos pode ser visualizado na Figura 1.

Este exemplo mostra a simulação de objetos tipo “tecido” pelo MatterJS representado pela grade retangular. O círculo e quadrado servem como obstáculos mostrando como o tecido se comporta colidindo com outros objetos. Os exemplos são acompanhados de código fonte.

Figura 1 - Exemplo de aplicação do MatterJS



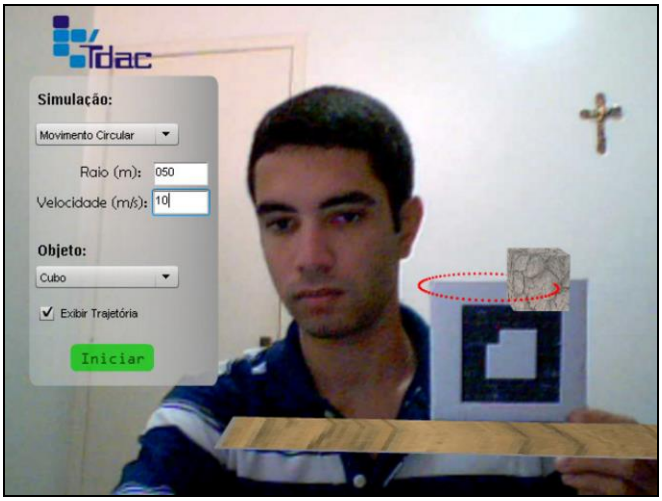
Fonte: Brummit (2014b).

2.6.2 SimulAR

O projeto SimulAR tem o intuito de utilizar a tecnologia de realidade aumentada como uma alternativa de baixo custo para o ensino de conteúdos de física. O software apresenta o comportamento de projéteis, visualizados em três dimensões (3D), com o auxílio de realidade aumentada (FERREIRA et al., 2011).

O software foi desenvolvido em ActionScript 3, e funciona usando marcadores no mundo real para indicar onde o software desenvolvido deve desenhar e simular os projéteis. Foi utilizada uma biblioteca chamada FLARToolKit para realizar o reconhecimento dos marcadores e posicionamento do conteúdo desenhado nas imagens obtidas via webcam. Para desenhar as imagens em 3D foi utilizada a biblioteca PaperVision3D. Todas as bibliotecas citadas são desenvolvidas para a linguagem ActionScript 3 (FERREIRA et al., 2011). Na Figura 2 pode-se ver um exemplo de execução do software.

Figura 2 - Execução do software SimulAR



Fonte: Ferreira et al. (2011).

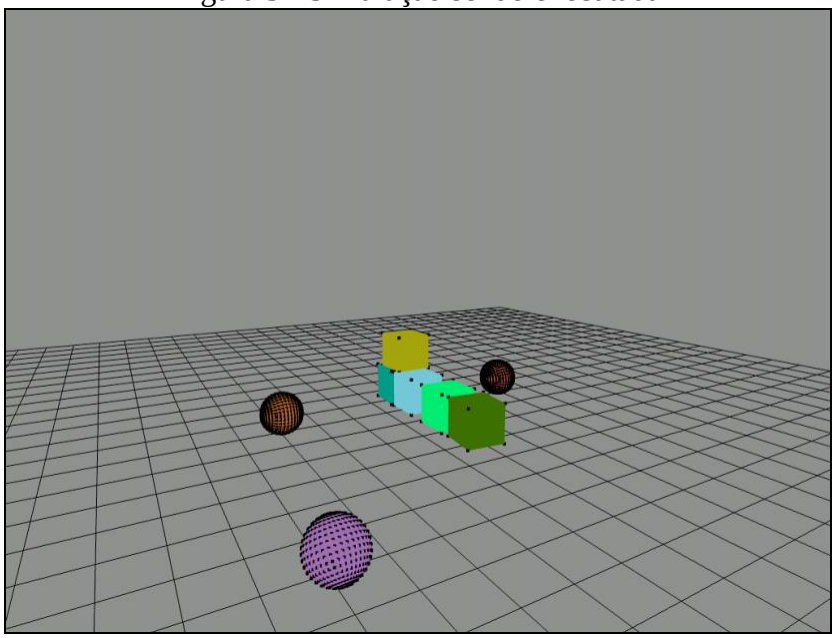
De acordo com Ferreira et al. (2011) o software obteve bons resultados de um ponto de vista pedagógico, pois, de acordo com o experimento aplicado com estudantes, eles concordaram que o software tem sim mérito para ser aplicado no ensino de física, inclusive para estudantes que não tiveram contato prévio com tecnologia.

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

Com um propósito semelhante ao deste projeto, Silva (2012) desenvolveu um software para gerar simulações gráficas na área de física. A grande diferença com relação a este trabalho é que a plataforma focada no trabalho de Silva foi a de dispositivos móveis, especialmente o sistema operacional iOS.

Silva (2012) utilizou a biblioteca gráfica OpenGL ES para realizar o processamento gráfico no software de testes e desenvolveu um motor de física próprio para realizar a simulação de gravidade, geração e tratamento de colisões entre os objetos e o meio, assim como entre os próprios objetos. O software de testes também permite alterar as propriedades físicas dos objetos criados, como a massa, velocidade, aceleração e coeficiente de atrito. A Figura 3 mostra um exemplo de simulação sendo executada.

Figura 3 - Simulação sendo executada



Fonte: Silva (2012).

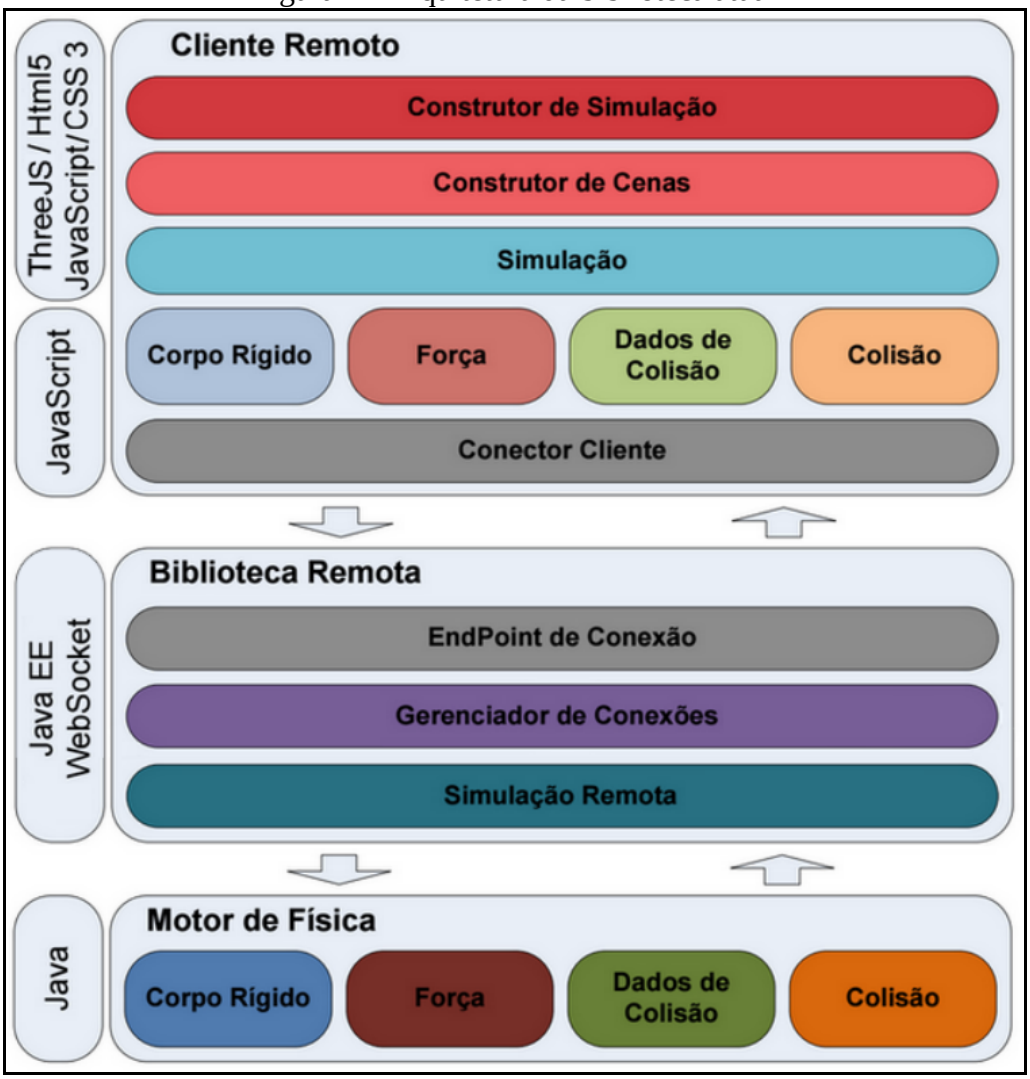
De acordo com Silva (2012, p. 56-57), os resultados obtidos foram satisfatórios. O software apresentou um consumo de memória baixo e conseguiu simular os objetos em uma boa visualização (em torno de 25 FPS) com até 50 objetos na cena.

2.7 BIBLIOTECA ATUAL

O motor desenvolvido por Teixeira (2015) foi dividido em três partes: um motor de física, que realiza a simulação física dos objetos, uma biblioteca remota que publica as funcionalidades do motor através de *WebSockets* e um cliente remoto que é uma biblioteca em Javascript que realiza a comunicação com a biblioteca remota. Um desenho de arquitetura da biblioteca desenvolvida por Teixeira (2015) pode ser observado na Figura 4.

No nível mais baixo tem-se o motor de física desenvolvido em Java. Esse motor realiza as operações matemáticas para a simulação de colisões, gravidade, atrito, etc. Um nível acima do motor há a biblioteca remota, que expõe as funcionalidades do motor de física usando Websockets. É importante observar que a biblioteca remota também está implementada do lado do servidor, para permitir sua comunicação com o motor.

Figura 4 - Arquitetura da biblioteca atual



Fonte: Teixeira (2015).

Acima da biblioteca remota há o lado cliente. No browser está implementado o conector cliente, que realiza a comunicação com a biblioteca e expõe para aplicações cliente as funcionalidades do motor de física. No nível mais alto, acima do conector cliente e da interface do motor de física, tem-se a aplicação cliente, aqui definida como um construtor de cenas e um construtor de simulação. Na Figura 4, a aplicação cliente é o construtor de simulações, mas essa aplicação pode ser qualquer aplicação que precise das funcionalidades do motor de física. Um exemplo é o próprio Ballistic (ZANLUCA, 2015).

3 DESENVOLVIMENTO DA BIBLIOTECA

Nesse capítulo são abordadas as adaptações realizadas tanto no CannonJS quanto no cliente remoto para utilizar o motor de física em Javascript, assim como os testes de validação e carga realizados sobre os dois motores. São abordados os principais requisitos, a especificação, a implementação e os resultados e discussões.

3.1 REQUISITOS

Os principais requisitos do sistema continuam iguais aos descritos por Teixeira (2015), com maiores alterações nos Requisitos Não Funcionais (RNF).

O motor de física deverá:

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

O cliente remoto deverá:

a) utilizar o novo motor de física (RNF);
b) apresentar a mesma interface do cliente remoto desenvolvido por Teixeira (2015) (RNF);
c) ser desenvolvido na linguagem Javascript (RNF).

3.2 ESPECIFICAÇÃO

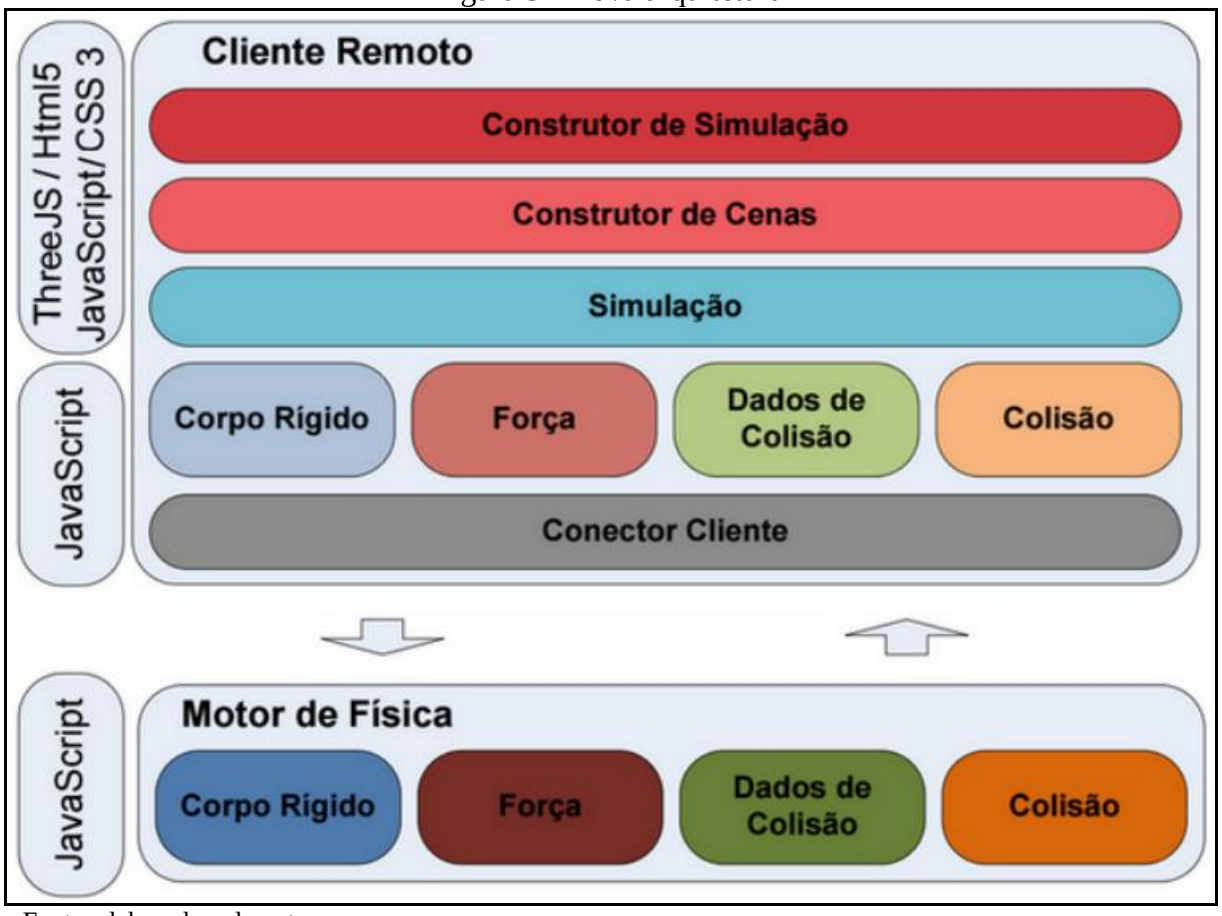
A especificação do trabalho foi desenvolvida utilizando a ferramenta Enterprise Architect (EA). Nela foram desenvolvidos diagramas de classe, sequencia e pacote. Foi optado por não desenvolver diagramas de caso de uso porque esses diagramas serão os mesmos do trabalho de Teixeira (2015), a única diferença foi a exclusão da Biblioteca Remota (TEIXEIRA, 2015, p. 30).

3.2.1 Diagrama de arquitetura

O diagrama de arquitetura apresentado na Figura 5 visa mostrar as alterações realizadas na arquitetura proposta por Teixeira (2015). A principal alteração foi a remoção da Biblioteca Remota, que realizava a conexão entre a aplicação cliente e o motor de física (via

cliente remoto). A partir de agora o cliente remoto se comunica diretamente com o motor de física. Somente foi mantido o cliente remoto para garantir a compatibilidade das aplicações já existentes com a nova arquitetura.

Figura 5 - Nova arquitetura



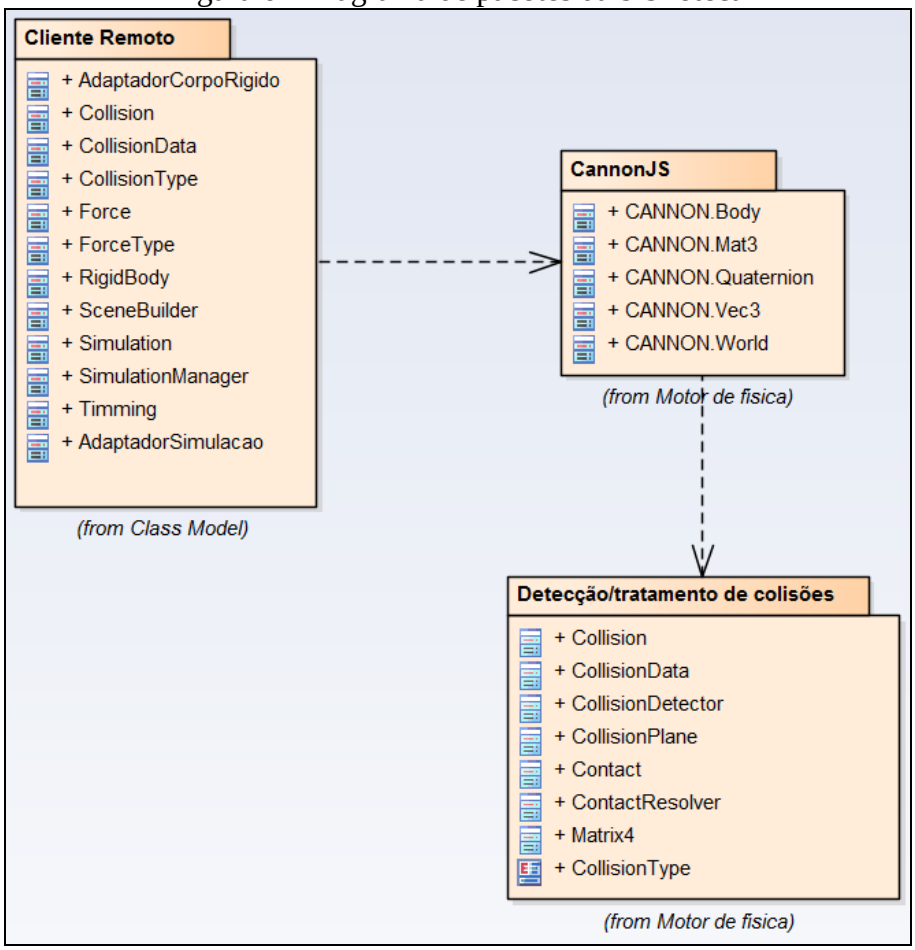
Fonte: elaborado pelo autor.

Além da remoção da Biblioteca Remota, o motor de física agora também funcionará em Javascript. Essa alteração possibilita que a aplicação funcione toda no lado cliente, removendo a necessidade de um servidor e consequentemente removendo a necessidade de uma conexão com a internet.

3.2.2 Diagrama de pacotes

Nesta seção será apresentado o diagrama de pacotes da biblioteca, com o intuito de demonstrar a forma como estão estruturadas suas classes e também demonstrar uma adaptação que foi realizada no CannonJS. O diagrama está exposto na Figura 6.

Figura 6 - Diagrama de pacotes da biblioteca



Fonte: elaborado pelo autor.

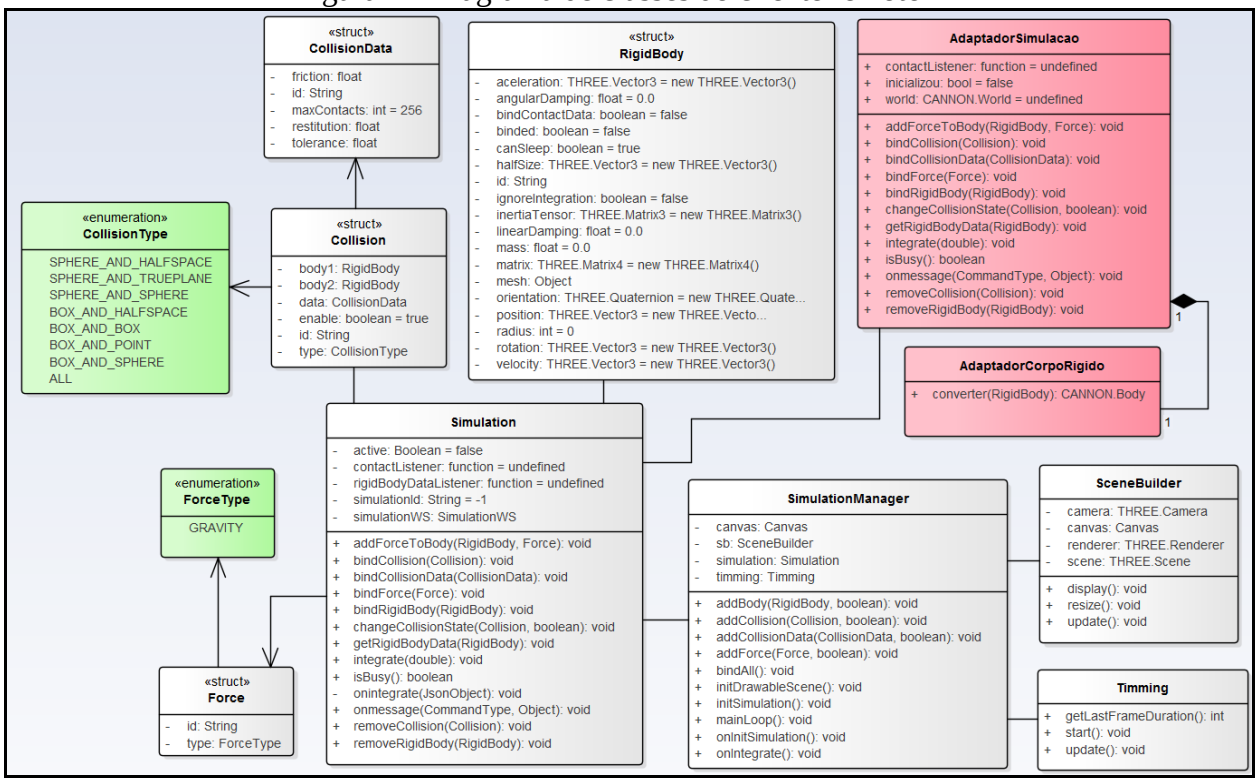
Os pacotes existentes são o cliente remoto, já existente na biblioteca de Teixeira (2015), o CannonJS que é o núcleo do motor de física e o pacote de detecção e tratamento de colisões.

A parte de detecção e tratamento de colisões ainda faz parte do motor de física, mas não foi possível utilizar o CannonJS para realizar esse trabalho porque as aplicações usando essa parte do CannonJS estavam se comportando de uma forma muito diferente das mesmas aplicações usando o Hefesto. Onde no Hefesto um objeto colidia com o chão e quicava diretamente para cima, no CannonJS esse objeto quicava em uma direção diferente, ou sofria uma rotação. Com esse problema foi decidido deixar de lado o CannonJS e converter a parte do Hefesto que realiza esse tratamento. Os pacotes serão discutidos mais a fundo nos diagramas de classes de cada pacote a seguir.

3.2.3 Diagrama de classes do cliente remoto

O cliente remoto expõe a funcionalidade do motor de física para as aplicações cliente e seu respectivo diagrama de classes pode ser visualizado na Figura 7.

Figura 7 - Diagrama de classes do cliente remoto



Fonte: elaborado pelo autor.

Boa parte das classes apresentadas no diagrama da Figura 7 já existiam no trabalho de Teixeira (2015). A aplicação cliente continua acessando as funcionalidades do motor de física usando a classe `Simulation` e os modelos de dados já disponibilizados anteriormente (`RigidBody`, `Force`, etc.). As maiores alterações nesse pacote foram a remoção da classe `SimulationWS` e a adição das classes `AdaptadorSimulacao` e `AdaptadorCorpoRigido`.

A classe `SimulationWS` realizava a comunicação entre o cliente remoto e a biblioteca remota. Como agora a biblioteca remota não existe mais, essa classe foi substituida pela classe `AdaptadorSimulacao`. A classe `AdaptadorSimulacao` é uma implementação do padrão de projeto Adapter, que tem como intuito adaptar a interface do cliente remoto ao novo motor de física utilizado. Ela implementa todos os métodos disponibilizados na classe `Simulation` e aplica os tratamentos necessários para que os parâmetros informados sejam passados corretamente para o motor de física. Esses tratamentos incluem a conversão dos modelos usados no Hefesto para os modelos usados pelo CannonJS, o mapeamento das funções do cliente remoto para funções que realizam essas operações no CannonJS e também a ligação dos objetos do CannonJS com os objetos da aplicação cliente.

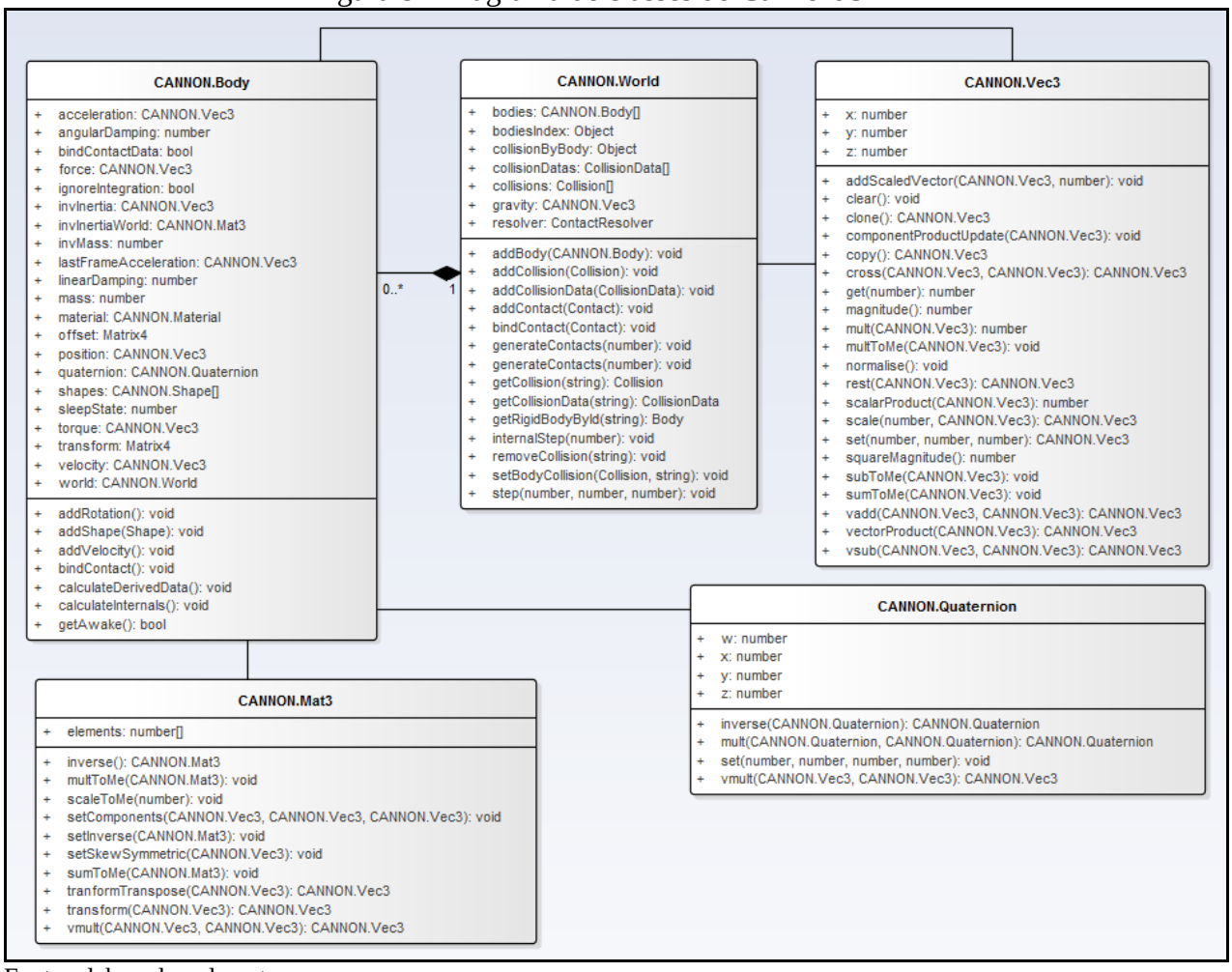
Já a classe `AdaptadorCorpoRigido` foi criada para isolar a operação de converter um `RigidBody` do Hefesto para um `Body` do CannonJS. Essa operação foi separada pela grande quantidade de parâmetros que foi necessário migrar de um modelo para outro. Existiam

inclusive parâmetros que não existiam no CannonJS e tiveram que ser implementados do zero.

3.2.4 Diagrama de classes do CannonJS

Na Figura 8 é apresentado o diagrama de classes do CannonJS. Somente os membros relevantes para esta implementação foram destacados.

Figura 8 - Diagrama de classes do CannonJS



Fonte: elaborado pelo autor.

Todas as classes desse pacote já estão inclusas na versão original do CannonJS, mas foi necessário realizar algumas implementações adicionais para facilitar a conversão do tratamento de colisões do Hefesto e também para substituir esse tratamento no CannonJS pelo tratamento convertido.

A primeira alteração foi na classe `World`. Foram adicionados novos métodos e propriedades referentes ao tratamento de colisões e também foi ajustado o método `internalStep`, substituindo as chamadas para as rotinas de tratamento de colisão antigas por

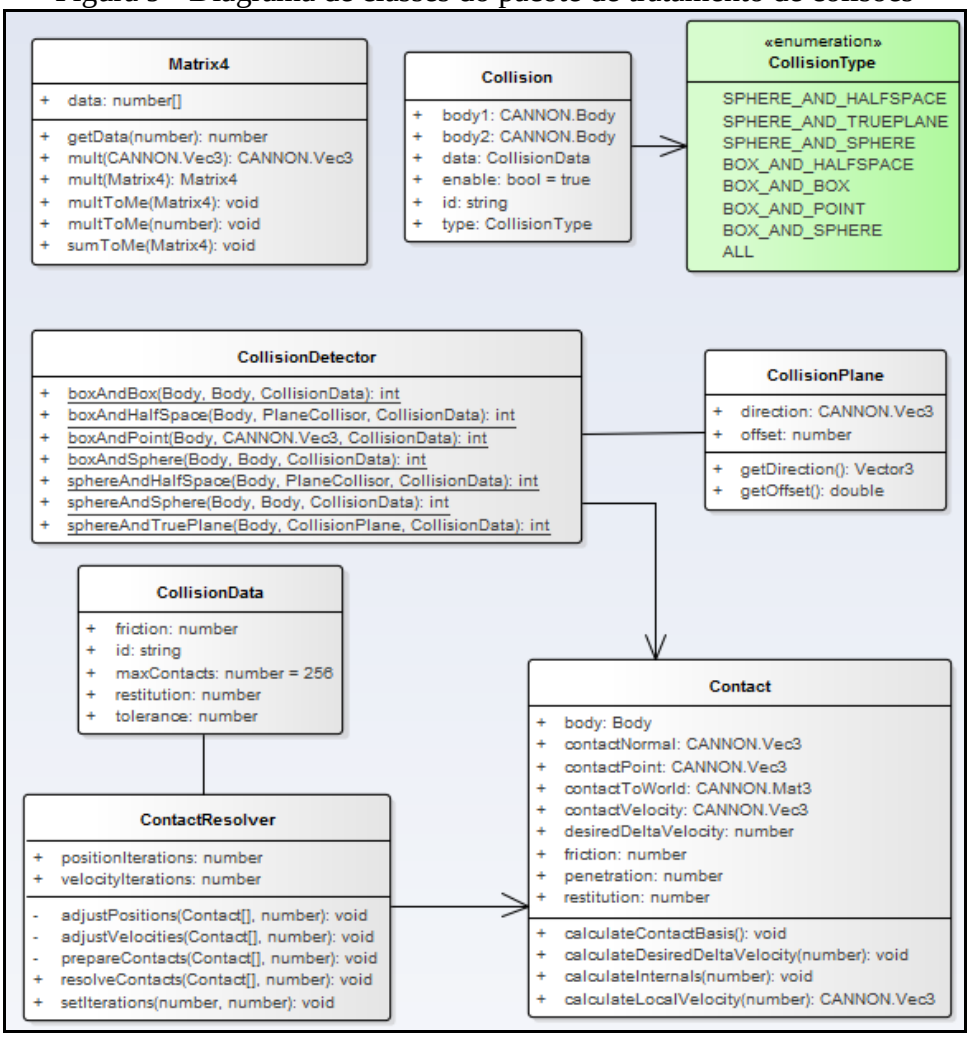
uma chamada ao método `generateContacts`. Tudo que executa a partir desse método é parte do pacote de detecção e tratamento de colisões e é código convertido do Hefesto.

Além dessa alteração também foram implementados uma série de *getters* e *setters* nas classes `Vec3`, `Quaternion`, `Mat3` e `Body`. Essa alteração foi feita porque as rotinas de tratamento de colisões do Hefesto acessavam os dados dos objetos por meio desses métodos, enquanto o CannonJS não os possua. Para facilitar a conversão e diminuir a chance de causar erros foi decidido adicionar estes métodos de acesso nos objetos do CannonJS. Estes métodos métodos foram ocultados no diagrama de classes da Figura 8.

3.2.5 Diagrama de classes do pacote de tratamento de colisões

Nesta seção é discutido o pacote de detecção e tratamento de colisões, convertido do Hefesto para o CannonJS com alguns ajustes. A Figura 9 apresenta o diagrama de classes desse pacote.

Figura 9 - Diagrama de classes do pacote de tratamento de colisões



Fonte: elaborado pelo autor.

Essas classes estão muito parecidas com as classes implementadas por Teixeira (2015), com algumas diferenças. Primeiramente os tipos dos atributos de cada classe foram substituídos por seus respectivos tipos no CannonJS. Esses tipos foram adaptados para atender as necessidades dessa parte do motor do Hefesto. A classe que recebeu a maior alteração foi a `Collision`. Anteriormente os atributos `body1` e `body2` eram do tipo `HRigidBody`, um *wrapper* que contém um `RigidBody` e outras propriedades usadas no tratamento de colisões. Como o CannonJS não tem esse *wrapper* e ao invés disso usa os objetos `Body` no tratamento de colisões, foi decidido alterar a classe `Body` incluindo novos membros e usá-los ao invés de criar um tipo novo.

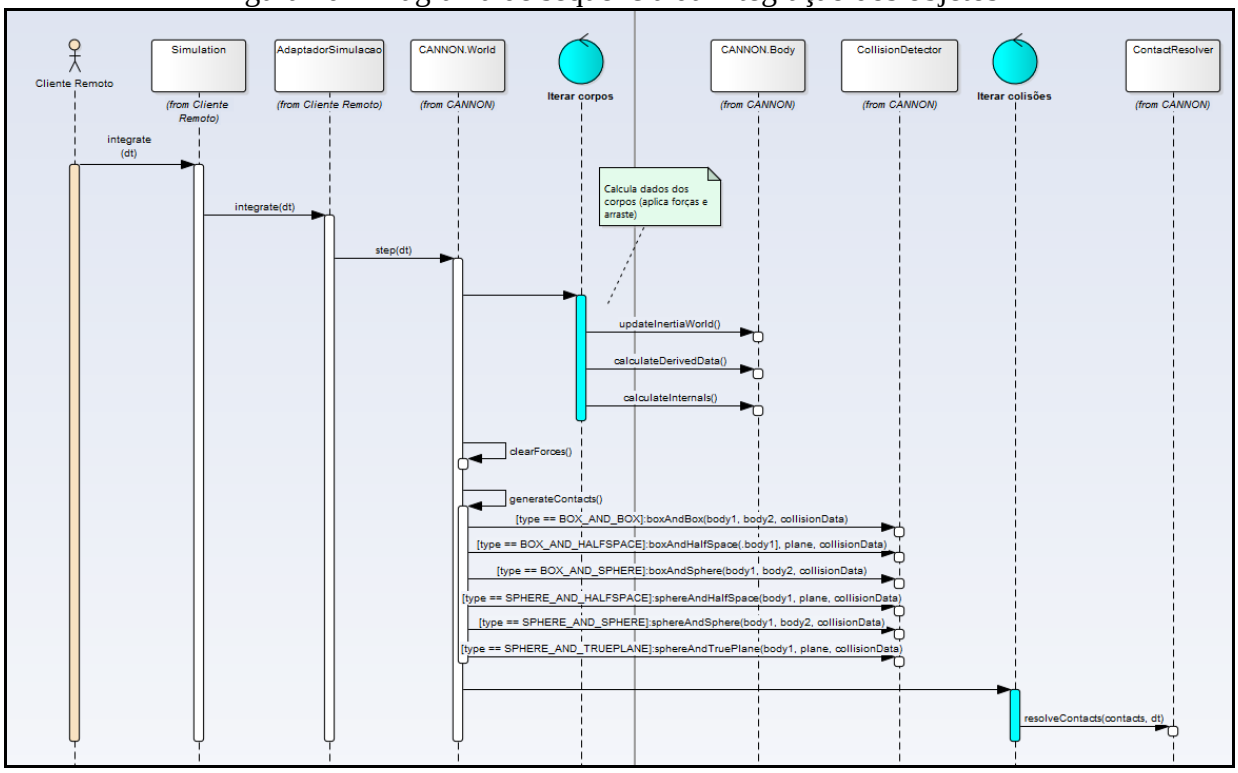
O tipo `CollisionPlane` representa o chão no mundo do Hefesto, ele é sempre criado de uma mesma forma na rotina de tratamento de colisões. Essa forma de tratar o chão impede o usuário do motor de criar terrenos customizados (por exemplo, uma cratera). Quando foi decidido usar o CannonJS também foi cogitado remover o `CollisionPlane` e usar o tipo `Body` do CannonJS para representar o chão, porém foi preciso usar a rotina de tratamento de colisões do Hefesto não foi possível realizar essa substituição.

3.2.6 Diagrama de sequencia da integração dos objetos

No Hefesto é chamada a integração para simular todos os objetos do mundo por um determinado tempo. Nessa simulação são aplicadas as forças que estão agindo sobre um objeto (gravidade, aceleração, rotação, etc.) e as colisões que ocorreram nesse período. Por ser a rotina com o maior processamento foi desenvolvido um diagrama de sequencia para detalhá-la, apresentado na Figura 10.

Assim como no Hefesto a integração dos objetos é iniciada pelo método `integrate` da classe `Simulation`. O processamento então prossegue para o `AdaptadorSimulacao` que repassa o fluxo de execução para o método `step` da classe `World`. Este fluxo é equivalente ao método `integrate` mas do CannonJS. Dentro do método `step`, o CannonJS realiza a aplicação da gravidade, aceleração e rotação dos corpos e passa o processamento para o `CollisionDetector`. O `CollisionDetector` irá então percorrer as possíveis colisões no mundo e gerar “contatos”, que serão processados pelo `ContacResolver` que atualizará as posições, acelerações e rotações de todos os objetos que sofreram alguma colisão. Quando o processamento retorna para o `AdaptadorSimulacao` este somente atualiza os objetos da aplicação cliente e a integração está finalizada.

Figura 10 - Diagrama de sequencia da integração dos objetos



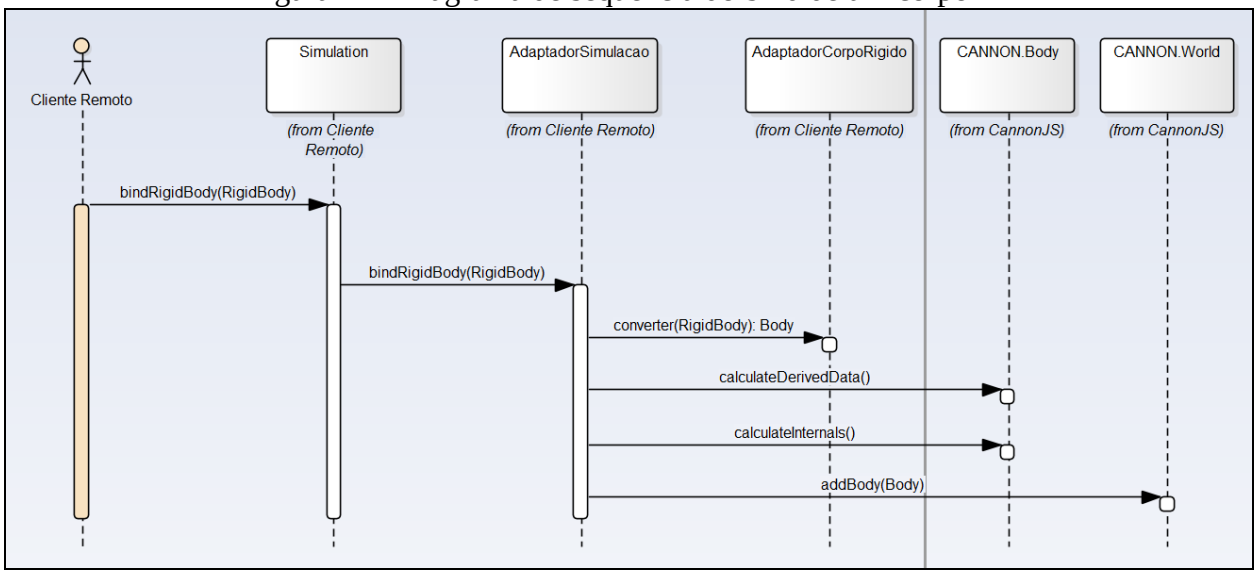
Fonte: elaborado pelo autor.

Uma diferença significativa desse mesmo processo realizado no trabalho de Teixera (2015, p. 37) é que no trabalho anterior esse processamento é assíncrono. A aplicação cliente pedia ao cliente remoto simular um determinado período de tempo, e o cliente remoto repassava esse comando para a biblioteca remota e o processamento acabava ali. A biblioteca remota posteriormente chamava outra função que recuperava os dados dos objetos do mundo e atualizava os objetos da aplicação cliente. Agora o processamento é síncrono, ou seja, uma chamada ao método `integrate` somente irá retornar quando os dados de todos os objetos do mundo estiverem atualizados.

3.2.7 Diagrama de sequencia do *bind* de um corpo

O *bind* de um corpo no Hefesto é o momento em que um corpo é enviado para o servidor pela primeira vez. Nesta rotina é onde ocorre o maior processo de conversão de dados entre o Hefesto e o CannonJS. Além de simples conversão também são adicionadas novas propriedades nos objetos do CannonJS para a rotina de tratamento de colisões. O diagrama de sequencia dessa rotina é mostrado na Figura 11.

Figura 11 - Diagrama de sequencia do bind de um corpo



Fonte: elaborado pelo autor.

O processo é iniciado pelo método `bindRigidBody` da classe `Simulation`. Esse método repassa o processamento para o `AdaptadorSimulacao` que chama o método `converter` de `AdaptadorCorpoRigido`. Esse método trata de criar um `Body` com as mesmas propriedades do `RigidBody` informado. Depois de finalizar a conversão são chamados os métodos `calculateDerivedData` e `calculateInternals` de `Body`. Esses métodos calculam informações usadas pela rotina de tratamento de colisões, tendo sido convertidos do Hefesto, pois não existiam originalmente no CannonJS. Com todos os dados calculados no `Body` é chamado o método `addBody` de `World` que finaliza esse processamento.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas na adaptação do cliente remoto para a utilização do CannonJS e da adaptação do mesmo para utilizar a rotina de tratamento de colisões do Hefesto. A seguir é mostrada a operacionalidade da biblioteca.

3.3.1 Técnicas e ferramentas utilizadas

Para realizar as alterações no cliente remoto foi utilizado o Integrated Development Environment (IDE) Visual Studio Code na versão 1.10.2. Além disso também foi utilizado o Notepad++ na versão 7.3 para realizar alterações mais pontuais no código fonte.

Também foi utilizada a IDE Eclipse na versão 4.6.2 (release Neon.2) para depuração e consulta do motor de física desenvolvido por Teixeira. Foi utilizado como framework uma Java Runtime Engine (JRE) na versão 8 e um servidor Tomcat na versão 9 para realizar o *host* do motor de física.

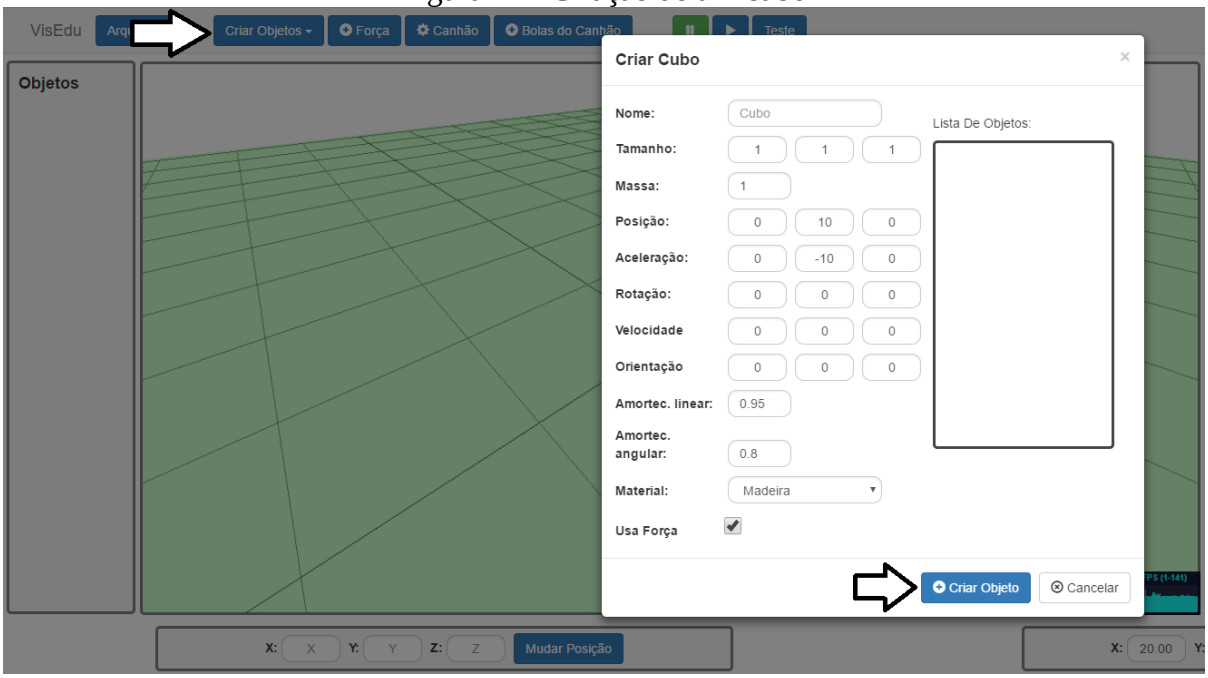
Os testes qualitativos foram realizados no navegador Google Chrome versão 57.0.2987. Para monitorar o consumo de memória RAM nos testes de performance foi utilizado o Process Explorer na versão 16.21.

Para gerenciamento dos fontes foi utilizado um repositório no Github e como ferramenta de interação com esse repositório foi utilizado o SourceTree na versão 2.0.20.1.

3.3.2 Operacionalidade da implementação

Nesta seção será mostrado um caso de uso da nova biblioteca. A aplicação utilizada para a realização desse teste será o editor Hefesto (TEIXEIRA, 2015) executando com a nova bilioteca. Inicialmente é adicionado um novo corpo na simulação, mostrado na Figura 12.

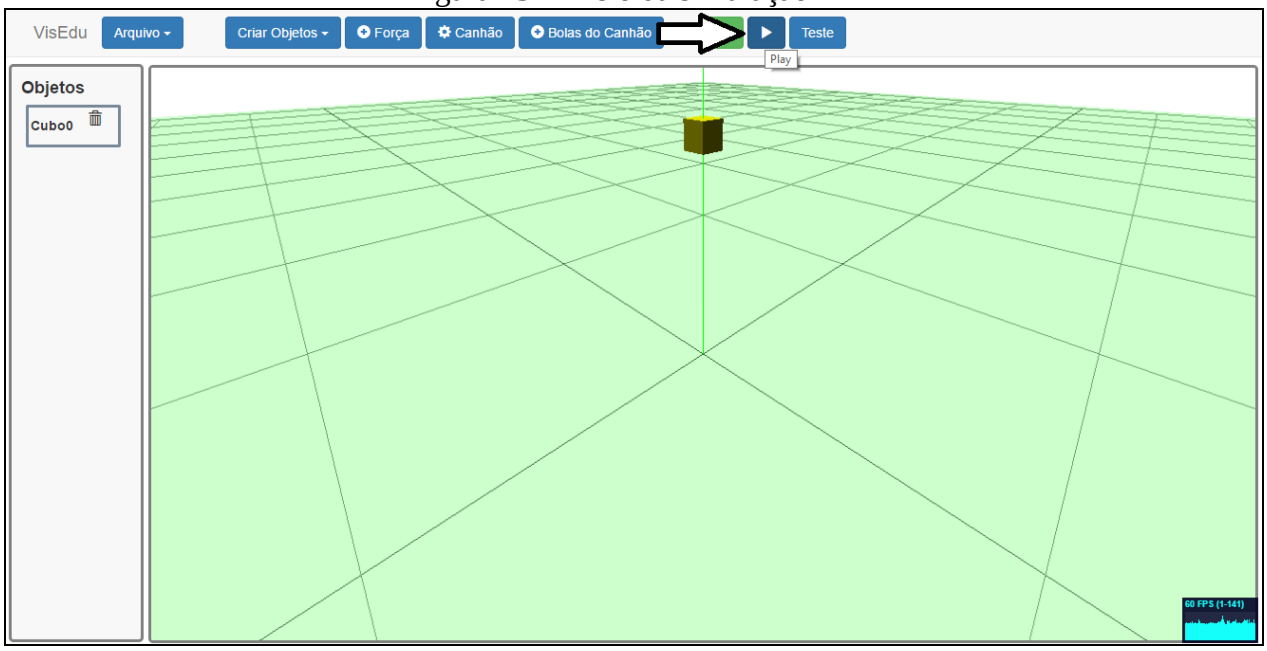
Figura 12 - Criação de um cubo



Fonte: elaborado pelo autor.

A criação do cubo não envolve comunicação com o servidor que executa o motor de física, por isso o código executado até este momento é o mesmo do trabalho anterior. Com o cubo criado no mundo foi iniciada a simulação. Este processo é mostrado na Figura 13.

Figura 13 - Início da simulação



Fonte: elaborado pelo autor.

Quando o usuário clica no botão indicado na Figura 13, a primeira operação que o sistema realiza é o *bind* de todos os objetos que o motor de física ainda não conhece. No trabalho anterior esse processo envolvia serializar os objetos do mundo e enviá-los ao servidor que estava executando o motor de física. Agora o processo envolve somente comunicar o CannonJS que um novo objeto deve ser adicionado ao mundo. O processo começa na classe `Simulation`, mostrada no Quadro 2.

Quadro 2 - Função `bindRigidBody`

```
102     bindRigidBody: function (body) {
103         this._rigidBodys = this.adaptador.bindRigidBody(body, this._rigidBodys);
104     },
```

Fonte: elaborado pelo autor.

A função repassa o processamento para o `AdaptadorSimulacao`, que tratará de adaptar o `RigidBody` informado para um modelo que o CannonJS entenda e comunicar ao motor de física que um novo corpo deve passar a ser simulado no mundo. A implementação da função `bindRigidBody` do `AdaptadorSimulacao` pode ser vista no Quadro 3.

Quadro 3 - Função `bindRigidBody` de `AdaptadorSimulacao`

```
20          this.bindRigidBody = function (body, corpos) {
21              var corpoCannon = this.adaptadorCorpoRigido.converter(body);
22
23              corpoCannon.calculateDerivedData();
24              corpoCannon.calculateInternals();
25
26              this.world.addBody(corpoCannon);
27              this.world.bodiesIndex[corpoCannon.id] = corpoCannon;
28
29              corpos[body.id] = body;
30              return corpos;
31          },
```

Fonte: elaborado pelo autor.

A função chamará o `AdaptadorCorpoRigido` para gerar um `Body` a partir do `RigidBody` informado e adicionará esse `Body` novo no mundo com a função `addBody`. Essa função não realiza uma grande quantidade de trabalho. A maior carga está sobre a função `converter`, conforme é mostrado no Quadro 4.

Quadro 4 - Função `converter` de `AdaptadorCorpoRigido`

```
3   this.converter = function(corpo)
4   {
5       var c = corpo;
6       var orientacao = new CANNON.Quaternion(-c.orientation.x, -c.orientation.y, -c.orientation.z, c.orientation.w);
7
8       var corpoCannon = new CANNON.Body({
9           mass: c.mass,
10          position: new CANNON.Vec3(parseFloat(c.position.x), parseFloat(c.position.y), parseFloat(c.position.z)),
11          velocity: new CANNON.Vec3(parseFloat(c.velocity.x), parseFloat(c.velocity.y), parseFloat(c.velocity.z)),
12          angularVelocity: new CANNON.Vec3(parseFloat(c.rotation.x), parseFloat(c.rotation.y), parseFloat(c.rotation.z)),
13          linearDamping: c.linearDamping,
14          angularDamping: c.angularDamping,
15          quaternion: orientacao,
16      });
17
18      corpoCannon.setInertiaTensor(c.inertiaTensor);
19      corpoCannon.ignoreIntegration = c.ignoreIntegration;
20      corpoCannon.useWorldForces = c.useWorldForces;
21      corpoCannon.id = c.id;
22
23      if (c.ignoreIntegration) {
24          corpoCannon.type = CANNON.Body.STATIC;
25      }
26
27      if (c.radius > 0) {
28         var forma = new CANNON.Sphere(c.radius);
29          corpoCannon.addShape(forma);
30      } else {
31          var hs = c.halfSize;
32          var forma = new CANNON.Box(new CANNON.Vec3(hs.x, hs.y, hs.z));
33          corpoCannon.addShape(forma);
34      }
35
36      if (c.bindContactData &&
37          c.bindContactData !== null) {
38          corpoCannon.bindContactData = c.bindContactData;
39      }
40
41      aceleracaoX = parseFloat(c.acceleration.x);
42      aceleracaoY = parseFloat(c.acceleration.y);
43      aceleracaoZ = parseFloat(c.acceleration.z);
44
45      corpoCannon.acceleration = new CANNON.Vec3(aceleracaoX, aceleracaoY, aceleracaoZ);
46
47      return corpoCannon;
48  }
```

Fonte: elaborado pelo autor.

Na linha 8 dessa função é chamado o construtor de `Body`. Este construtor disponibiliza uma estrutura chamada `options` para informar os parâmetros do objeto criado. As linhas 10 a 12 convertem a posição, velocidade e rotação para as respectivas propriedades de `Body`. É

necessário realizar o *parse* destas propriedades porque o CannonJS trabalho com valores numéricos nesses campos. No trabalho anterior esses valores eram mandados para o servidor como *strings* e esse *parse* era realizado da mesma forma no servidor. O `if` na linha 27 checa se o `RigidBody` passado para esse método tem um `radius` porque somente esferas tem esse valor informado no Hefesto, assim como somente cubos tem a propriedade `halfSize` informada. Assim, é criado o `Shape` correto do CannonJS e atribuido no `Body`.

Com o corpo criado o motor de física está pronto para iniciar a simulação, a função `integrate` da classe `Simulation` deve ser chamada informando o tempo que deve ser simulado pelo motor. A implementação da função `integrate` pode ser vista no Quadro 5.

Quadro 5 - Função `integrate` de `Simulation`

```
168     integrate: function (duration) {
169         this._rigidBodys = this.adaptador.integrate(duration, this._rigidBodys);
170     },
```

Fonte: elaborado pelo autor.

Da mesma forma que a função `bindRigidBody`, a função `integrate` também somente repassa o processamento para a função de mesmo nome da classe `AdaptadorSimulacao`. A implementação dessa função pode ser vista no Quadro 6.

A primeira etapa desse processamento é informar ao CannonJS que deve ser processado um intervalo de tempo (`duration`) no mundo. Isto é feito chamando o método `step` de `World`. Depois que a simulação foi avançada por esse período é possível atualizar todos os corpos do mundo, para que eles sejam representados corretamente no editor Hefesto (aqui esses corpos são representados pelo parâmetro `corpos)` .

A função `step` em grande parte é implementada pelo CannonJS, ele adiciona as forças que estão sendo aplicadas sobre um deteminado objeto (gravidade, aceleração, rotação, etc.) e atualiza a posição e velocidade destes no mundo. Como grande parte desse código não foi implementado nesse trabalho somente serão mostradas aqui as partes que foram alteradas. Duas grandes alterações foram feitas na rotina de integração do CannonJS. A primeira alteração é permitir que seja configurada uma aceleração nos objetos físicos do mundo, a implementação desse tratamento pode ser visto no Quadro 7.

Quadro 6 - Função `integrate` de `AdaptadorSimulacao`

```
this.integrate = function (duration, corpos) {
    this.world.step(duration);
    var corposCannon = Object.values(this.world.bodiesIndex);

    for (var i = 0; i < corposCannon.length; i++) {
        var _rb = corposCannon[i];
        var rb = corpos[_rb.id];
        var p = new THREE.Vector3();
        p.x = _rb.position.x;
        p.y = _rb.position.y;
        p.z = _rb.position.z;
        rb.position = p;
        // caso nao possua mesh, nao precisa ajustar valores de exibicao
        if (rb.mesh == undefined) {
            continue;
        }
        rb.mesh.position.x = p.x;
        rb.mesh.position.y = p.y;
        rb.mesh.position.z = p.z;
        //tentar setar o orientacao e posicao
        rb.mesh.matrix.identity();
        var utils = new AdaptadorUtils();
        var a = utils.calcularMatrizTransformacao(_rb.position, _rb.quaternion);
        var m = new THREE.Matrix4();
        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]);
        rb.mesh.applyMatrix(m);
        rb.mesh.updateMatrix();
        rb.mesh.matrix.setPosition(p);
        if (this._contactListener != undefined && this._contactListener != null) {
            var contacts = this.world.contacts;
            for (j = 0; j < contacts.length; j++) {
                this._contactListener(contacts[j]);
            }
        }
        log('Integrate simulation: ' + this._simulationId);
    }
    return corpos;
},
```

Fonte: elaborado pelo autor.

Quadro 7 - Tratamento de aceleração no CannonJS

```
b.lastFrameAcceleration = new Vec3(b.acceleration.x, b.acceleration.y, b.acceleration.z);
b.lastFrameAcceleration.vadd(b.force.scale(invMass), b.lastFrameAcceleration);

velo.x += b.lastFrameAcceleration.x * dt;
velo.y += b.lastFrameAcceleration.y * dt;
velo.z += b.lastFrameAcceleration.z * dt;
```

Fonte: elaborado pelo autor.

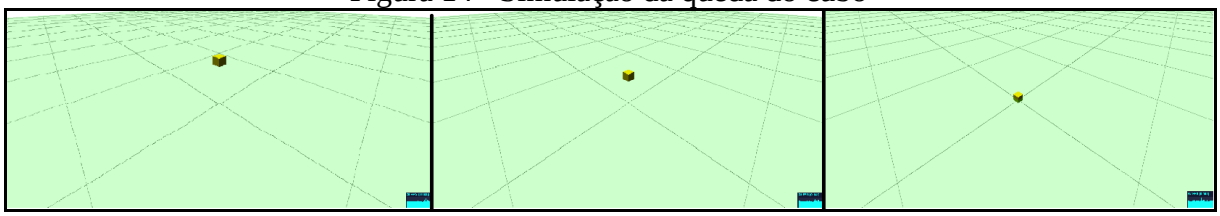
Neste trecho de código `b` é o corpo do mundo e `velo` é sua velocidade. Originalmente essa rotina somente adicionava o valor de `force` de `b` sobre sua velocidade. Foi necessário alterar esse tratamento para passar a considerar a aceleração do corpo, isso é feito utilizando a propriedade `acceleration` do corpo e somando esse valor à propriedade `force` multiplicado pelo inverso da massa do corpo (o mesmo tratamento implementado no Hefesto).

Além dessa alteração, também foi removido o tratamento de colisões implementado pelo CannonJS e implementado o mesmo tratamento realizado pelo Hefesto. No final do método `internalStep` do CannonJS foi adicionado uma chamada ao método

`generateContacts`. O tratamento a partir desse ponto é o mesmo implementado por Teixeira (2015) mas convertido para Javascript.

Prosseguindo com a simulação, o cubo cai até o plano do chão e quica algumas vezes, diminuindo a altura a cada vez, até parar. Esse comportamento está representado na Figura 14.

Figura 14 - Simulação da queda do cubo



Fonte: elaborado pelo autor.

3.4 ANÁLISE DOS RESULTADOS

Foram realizados dois tipos de teste sobre a biblioteca nova, qualitativos e de performance. Os testes qualitativos tinham como intuito validar que a biblioteca continua se comportando de forma similar ao Hefesto depois da conversão. Os testes de performance tinham como objetivo avaliar se a construção do `adapter` do CannonJS e a conversão das rotinas de tratamento de colisões do Hefesto causaram algum impacto em questão de performance, assim como validar se a remoção da comunicação cliente-servidor traria algum ganho no mesmo sentido.

3.4.1 Testes qualitativos

Esses testes foram realizados sem coleta de estatísticas, foram executados o editor Hefesto e o Ballistic lado a lado com os dois motores e avaliado visualmente se a simulação estava ocorrendo de forma similar. Foi decidido não coletar nenhuma estatística aqui pois o objetivo do trabalho nunca foi deixar os dois motores se comportando de forma exatamente igual, apenas o motor convertido deveria se comportar de uma forma próxima ao seu comportamento no mundo real (tendo como parâmetro o motor antigo).

Foram validados cenários de cada aplicação separadamente, cobrindo as principais funcionalidades de ambas. Os cenários de teste para a aplicação Hefesto são:

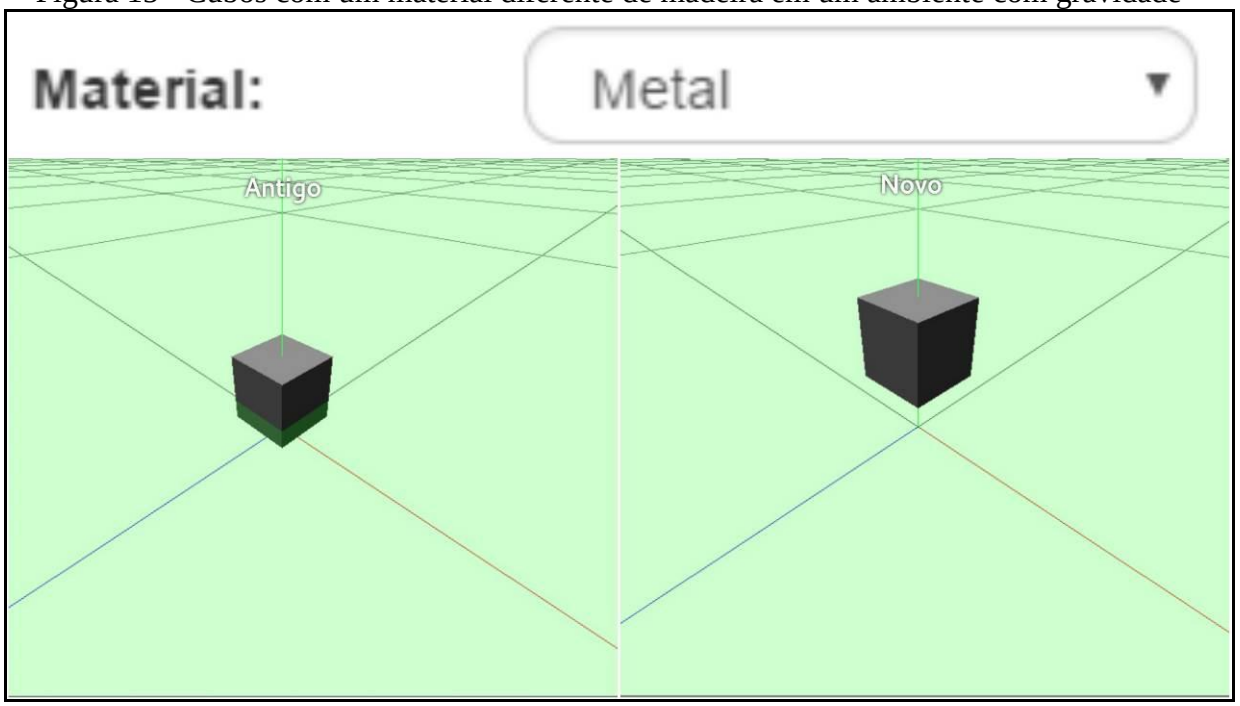
a) validar que os objetos estão se comportando corretamente sob gravidades diferentes;
b) validar que objetos com alterações nas seguintes propriedades se comportam corretamente:
   - aceleração,
   - rotação,

- orientação,
- material,
- amortecimento linear,
- amortecimento angular,
- “usa força” (propriedade booleana indicando se um objeto é afetado pela gravidade ou não);

c) validar que objetos que podem colidir um com outro estão colidindo corretamente.

Um exemplo dessas validações pode ser visto na Figura 15. As outras validações são apresentadas no apêndice A

Figura 15 - Cubos com um material diferente de madeira em um ambiente com gravidade



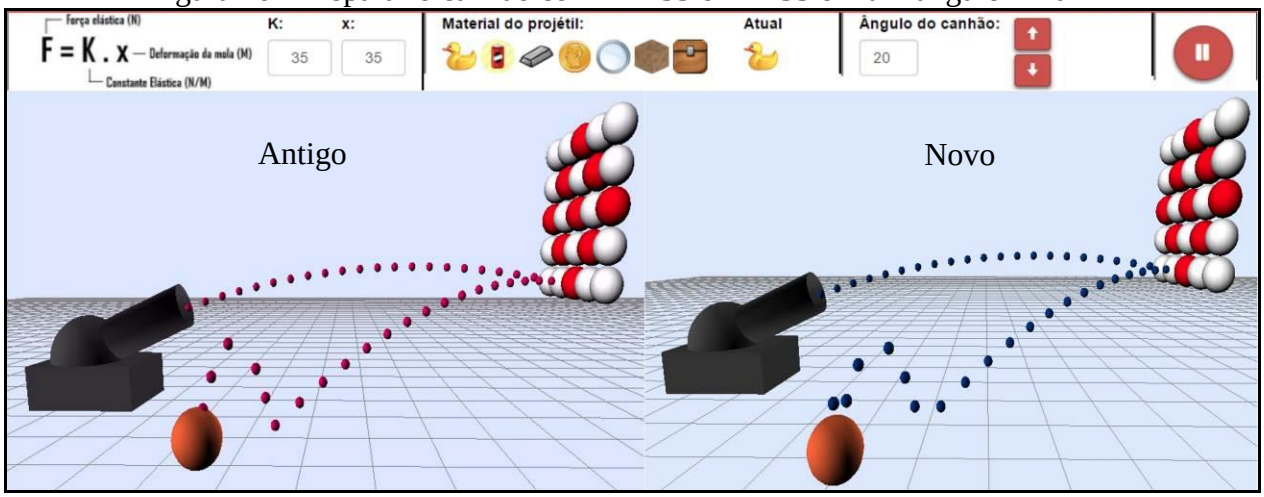
Fonte: elaborado pelo autor

Os cenários de testes para o Ballistic são:

a) validar um disparo do canhão com os valores padrão;
b) validar um disparo do canhão com uma alteração nos parâmetros K, X e ângulo;
c) validar um disparo do canhão na gravidade da Lua;
d) validar um disparo do canhão com a bola sendo do material Ferro;
e) validar que o modo tutorial está funcionando corretamente;
f) validar que o botão “Recarregar simulação” está funcionando corretamente.

Um exemplo dessas validações pode ser visto na Figura 16. As outras validações são apresentadas no apêndice B.

Figura 16 - Disparar o canhão com K = 35 e X = 35 em um ângulo = 20

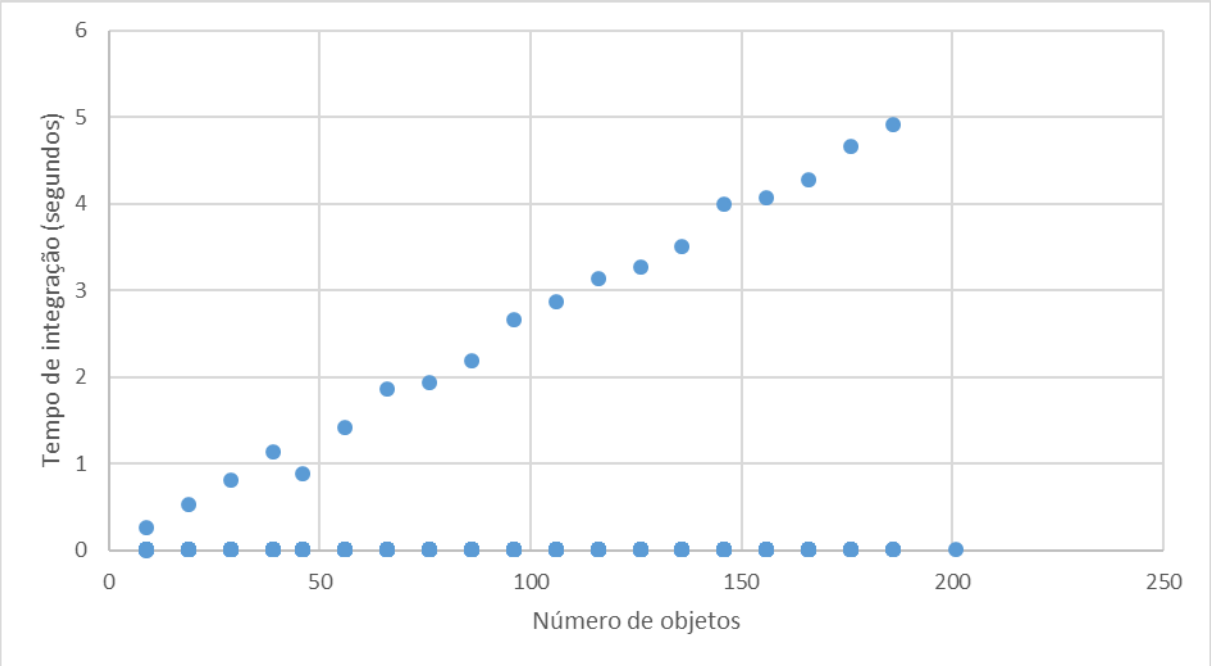


Fonte: elaborado pelo autor.

3.4.2 Testes de performance

Foram realizados dois testes de performance em ambas aplicações. O primeiro teste consiste em executar uma simulação que adiciona gradualmente mais objetos no mundo até chegar em 200 objetos, durante todo o período gravando a cada iteração dos objetos no mundo o tempo desde a última iteração e o número de objetos sendo simulados atualmente no mundo. Para esse teste foi gerado como saída um gráfico de dispersão demonstrando o número de objetos no mundo/tempo de integração. Esse gráfico para o motor antigo é mostrado na Figura 17.

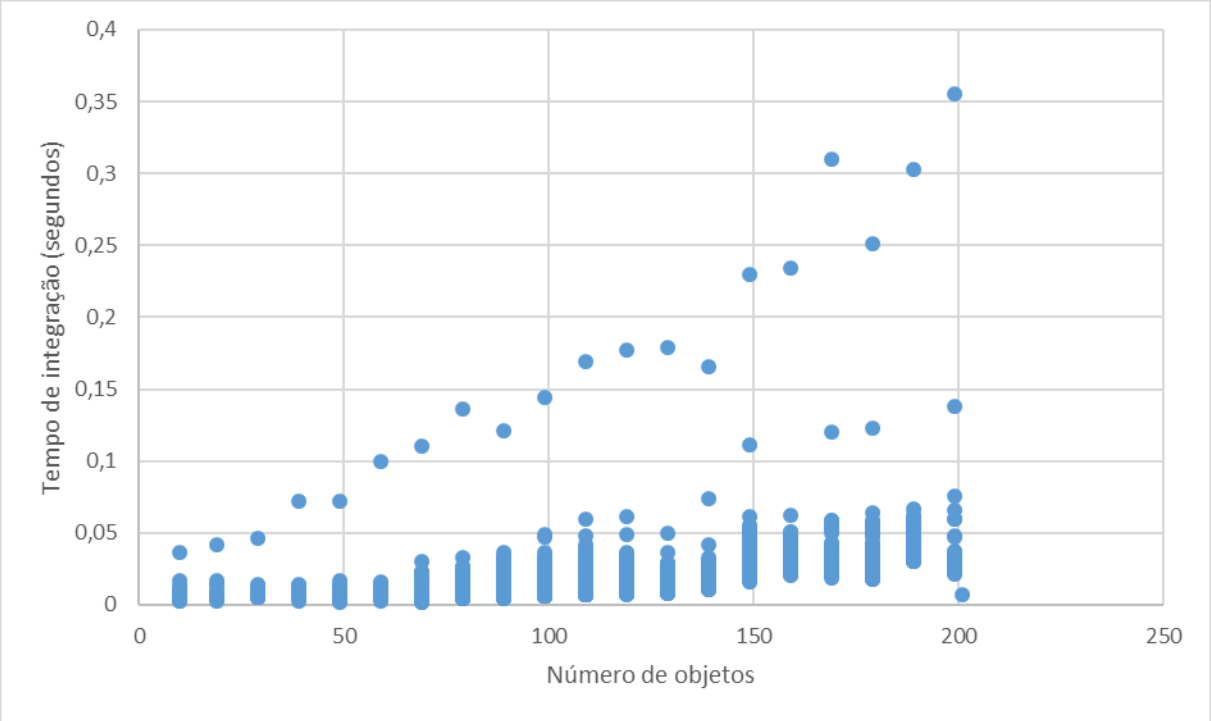
Figura 17 - Gráfico número de objetos/tempo de integração motor antigo



Fonte: elaborado pelo autor.

Pode-se observar um salto maior no tempo de integração em alguns ciclos conforme o número de objetos sobe. Esse salto se dá nos momentos em que o cliente remoto está sincronizando os corpos novos com o servidor. O *bind* de um corpo e de suas colisões causa uma grande transferência de dados entre o cliente e o servidor. Essa transferência deixará o servidor "ocupado" até que o *bind* do corpo e das colisões seja finalizado. A Figura 18 mostra o mesmo gráfico quando utilizado o motor de física novo.

Figura 18 - Gráfico número de objetos/tempo de integração motor novo

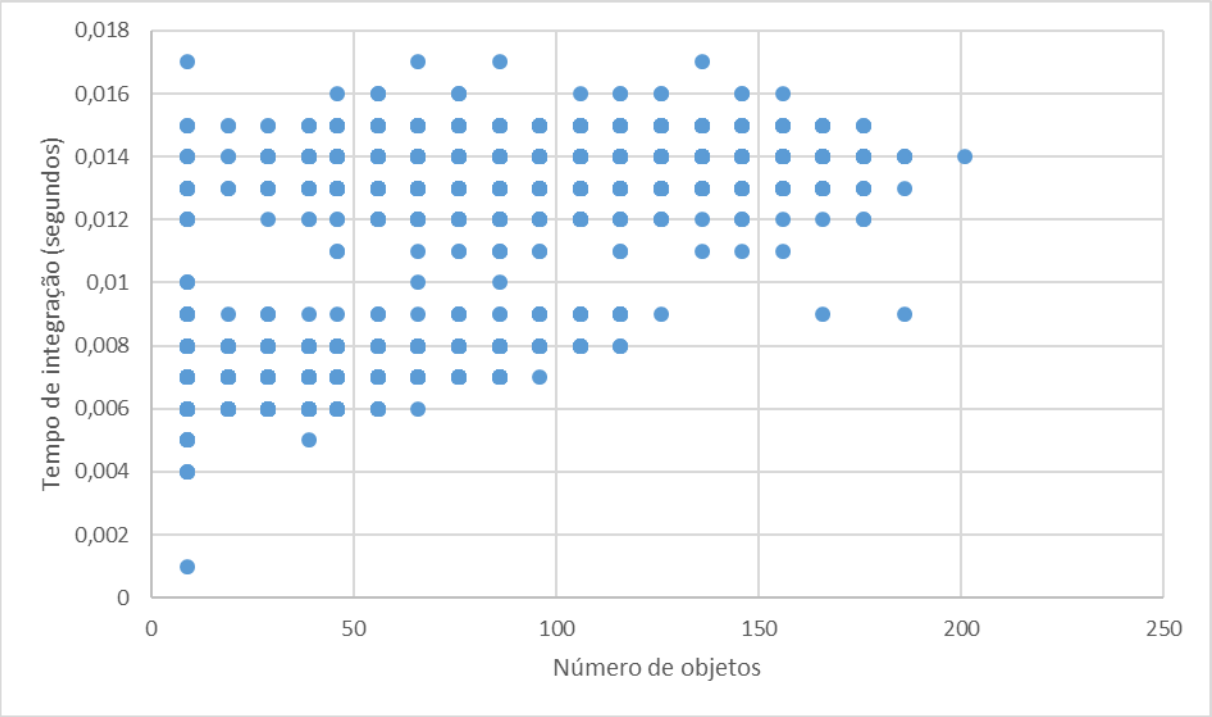


Fonte: elaborado pelo autor.

Para o motor em Javascript também pode se observar aumentos súbitos no tempo de integração dos objetos, mas tais aumentos são de uma magnitude menor, subindo somente até 0,36 segundos. Esse resultado ocorre porque no motor convertido não existe mais a transferência de dados entre o cliente e servidor. Os aumentos periódicos se dão também pelo *bind* dos corpos e colisões, mas o processamento é todo realizado no cliente.

Foi realizada outra validação, utilizando o tempo de integração no motor antigo porém sem considerar o *bind* dos objetos. Para esse teste foram utilizados os resultados expostos no gráfico da Figura 17 e removidos os resultados com tempos de integração maiores do que dez vezes a média do tempo de integração de todos os resultados. Dessa forma somente foram considerados resultados da rotina de integração de objetos, sem a carga adicional de rede que é o *bind* dos corpos no servidor. Esse gráfico adaptado pode ser visto na Figura 19.

Figura 19 - Tempo de integração/número de objetos no motor antigo removendo tempos muito altos



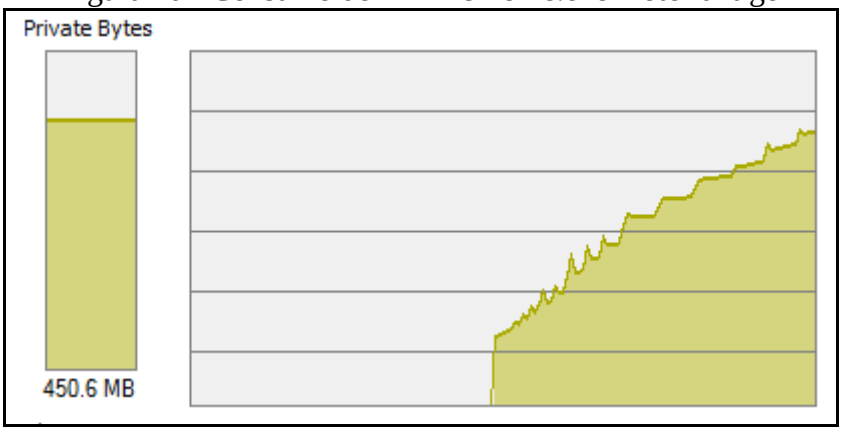
Fonte: elaborado pelo autor.

Pode-se ver que o cenário melhora bastante se for desconsiderado o *bind* dos corpos e colisões no servidor, ficando inclusive comparável com o resultado do motor convertido. Mesmo assim é importante ressaltar que esses testes foram realizados em um cenário que desconsidera latência de rede, já que o servidor e o cliente estavam rodando na mesma máquina. Em um cenário onde o cliente e o servidor estão em máquinas separadas (o cenário mais realista) ainda existe a sobrecarga de rede ao considerar o motor antigo.

Além do teste de tempo de integração também foi realizado um teste medindo o consumo de memória RAM. Esse teste foi realizado para validar que a conversão do motor para Javascript não causaria um consumo excessivo de memória RAM no browser. As Figuras 20 e 21 mostram gráficos de consumo de memória RAM para o motor antigo. Os processos monitorados foram o chrome.exe (browser renderizando a simulação) e o javaw.exe (servidor web).

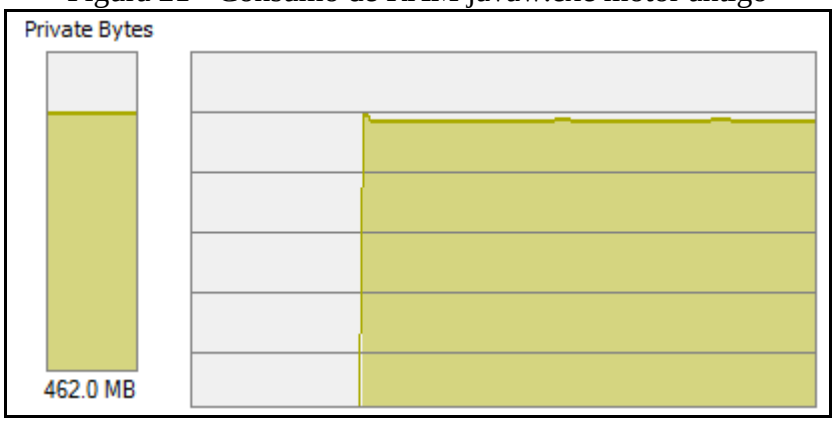
O consumo de RAM do browser começou em aproximadamente 120 MB e foi aumentando gradativamente conforme o número de objetos aumentava na simulação, chegando a um pico de aproximadamente 450 MB. Já o consumo do servidor web ficou constantemente próximo de 460 MB. Considerando os dois, temos um pico de consumo de aproximadamente 910 MB de memória RAM.

Figura 20 - Consumo de RAM chrome.exe motor antigo



Fonte: elaborado pelo autor.

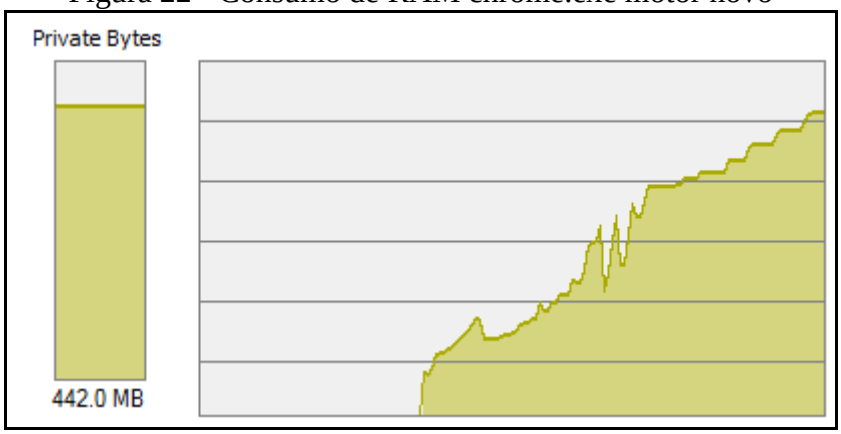
Figura 21 - Consumo de RAM javaw.exe motor antigo



Fonte: elaborado pelo autor.

No motor convertido não temos o servidor web. Por isso o único processo monitorado foi o chrome.exe que estava executando a simulação. O gráfico expondo esses resultados pode ser visto na Figura 22.

Figura 22 - Consumo de RAM chrome.exe motor novo



Fonte: elaborado pelo autor.

Pode-se observar o mesmo aumento no consumo de RAM observado no motor antigo, chegando a um pico um pouco menor de 442 MB. Com esse resultado pode-se concluir que a conversão do motor para Javascript não causou impacto no consumo de memória RAM,

mantendo os mesmos valores no cliente mas economizando 460 MB utilizados no antigo servidor.

3.4.3 Comparação com correlatos

No Quadro 8 é apresentada uma comparação entre o sistema desenvolvido e os trabalhos correlatos.

Quadro 8 - Comparação com correlatos

Característica	Brummit (2014)	Ferreira et al. (2011)	Silva (2012)	Hefesto Javascript (2017)
Customização de cenas 3D	-	-	X	X
Motor de física próprio	X	?	X	X*
Executa em dispositivos móveis	X**	-	X	X**
Executa na Web	X	-	-	X
Possui aplicação própria	-	X	X	X
Usa realidade aumentada	-	X	-	-

Fonte: elaborado pelo autor.

No Quadro 8 o item marcado com um asterisco (*) mostra que a aplicação desenvolvida usa parte do CannonJS e parte do Hefesto, então o motor de física final é uma junção dos dois motores anteriores adaptados para as necessidades desse trabalho. Os itens marcados com dois asteriscos (**) indicam que o trabalho pode executar em dispositivos móveis por ter sido desenvolvido na web.

Pode-se observar pelo Quadro 8 que o trabalho desenvolvido aqui tem as principais características dos correlatos selecionados. A customização de cenas 3D é uma característica inerente ao trabalho por se tratar de um motor de física. O trabalho de Brummit (2014) tem a capacidade de realizar a customização de cenas, mas não em três dimensões. O motor de física MatterJS realiza a simulação em duas dimensões. O trabalho de Ferreira et al. (2011) não se trata de um motor de física, mas sim de uma aplicação educacional mais direcionada.

Por se tratar de uma combinação de dois outros motores de física com algumas adaptações em ambos, foi considerado que a aplicação desenvolvida tem um motor de física próprio, característica presente em todos os outros trabalhos exceto o trabalho de Ferreira et al. (2011). No trabalho de Ferreira et al. (2011) não ficou claro no texto se a aplicação foi desenvolvida usando um motor de física próprio, de terceiro, ou se não foi usado um motor de física e somente foram animados os cenários em realidade aumentada.

As características “executa em dispositivos móveis” e “executa na web” são muito próximas uma da outra, uma aplicação web desenvolvida de forma responsiva e comportando eventos de toque também irá executar em dispositivos móveis. A biblioteca desenvolvida não desenha nada na tela do browser, pois essa parte do trabalho fica a cargo da aplicação cliente. Por isso uma aplicação cliente que for desenvolvida de forma que suporte dispositivos móveis irá permitir que o motor novo também execute nesses dispositivos. No entanto nenhuma das duas aplicações usadas para validar esse motor (Ballistic e editor Hefesto) foram planejadas para executar em dispositivos móveis.

A característica de “aplicação própria” indica um trabalho que tem uma aplicação utilizável e não disponibiliza apenas uma biblioteca. O Hefesto Javascript tem como aplicação própria o editor Hefesto, que pode ser usado para gerar simulações com todas as características disponibilizadas pelo motor. Dos correlatos somente Brummit (2014) não tem uma aplicação própria. Esse trabalho se trata de um motor de física sem nenhuma aplicação disponível. Ferreira et al. (2011) disponibilizou uma aplicação educacional usando realidade aumentada e Silva (2012) disponibilizou um editor de simulações semelhante ao editor Hefesto mas para dispositivos móveis.

4 CONCLUSÕES

Nesse trabalho foi possível remover a dependência de um servidor na execução de simulações de física utilizando o motor Hefesto. Foi possível realizar essa melhoria utilizando as rotinas de integração do motor de física CannonJS em conjunto com as rotinas de detecção e tratamento de colisões do Hefesto (convertidas para Javascript). Essa alteração trouxe um grande ganho de performance por remover a comunicação e a latência de rede entre o cliente Javascript e o servidor que executava as simulações de física na linguagem Java.

O principal objetivo desse trabalho, remover essa dependência, foi cumprido. Mas não foi possível cumprir essa meta totalmente da forma como ela foi proposta. Inicialmente a ideia era utilizar todas as rotinas do CannonJS, um motor de física com um tempo de desenvolvimento maior que o Hefesto e também utilizado em mais aplicações. Não foi possível cumprir essa meta pois percebe-se que o tratamento de colisões dos dois motores são diferentes. Sendo assim as aplicações Ballistic (ZANLUCA, 2015) e o editor Hefesto (TEIXEIRA, 2015) não se comportariam da mesma forma com o motor novo. Mesmo depois da junção das rotinas dos dois motores o resultado ainda foi positivo em questão de tempo de processamento e consumo de memória RAM.

A remoção da dependência de um servidor deixa as aplicações que utilizam o Hefesto mais consistentes, corrigindo problemas refentes a arquitetura escolhida por Teixeira (2015) que foram apontados por Reis (2016) durante a proposta do trabalho. Entre esses problemas um muito eminente era a inconsistência no acesso a internet de escolas, causando problemas quando se tentava executar alguma simulação usando o Hefesto. Outro grande problema era a latência na comunicação entre o cliente desenhando a simulação e o servidor processando-a. Como esses dois problemas são decorrentes da estratégia de executar a simulação de física em um servidor web, eles são resolvidos por esse trabalho.

O motor de física novo ainda tem limitações. Atualmente ele só consegue utilizar um núcleo do processador, causando um gargalo de processamento em simulações complexas. Além disso como esse trabalho teve o intuito de corrigir um problema de arquitetura ele não adicionou nenhuma funcionalidade. Sendo assim, as limitações funcionais apontadas por Teixeira (2015) ainda se aplicam.

4.1 EXTENSÕES

As extensões sugeridas por Teixeira (2015, p. 66) ainda se aplicam:

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;
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 discriminação de colisões;
g) criar mecanismos para obter valores associados a corpos rígidos específicos 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.

Além disso também são adicionadas as seguintes sugestões:

a) estudar a viabilidade de utilizar as estruturas dispostas pelo CannonJS para extender o motor;
b) possibilitar o processamento *multithread* utilizando o recurso de *web workers* do HTML5;
c) possibilitar a adição de um chão customizado no mundo da simulação de física.

REFERÊNCIAS

BRASIL. Ministério da Educação (MEC), Secretaria de Educação Básica. **Parâmetros Curriculares Nacionais do Ensino Médio(PCNEM)**. Ciências da Natureza, Matemática e suas Tecnologias, 1999. Disponível em: <http://portal.mec.gov.br/seb/arquivos/pdf/ciencian.pdf>. Acesso em: 04 set. 2016.

BRUMMIT, Liam. **matter.js.** 2014a. Disponível em: <http://brm.io/matter-js/>. Acesso em: 06 nov. 2016.

_____. **Matter.js Demo.** 2014b. Disponível em: <http://brm.io/matter-js/demo/>. Acesso em: 06 nov. 2016

CARNEIRO, Neyla L. **A prática docente nas escolas públicas, considerando o uso do laboratório didático de física**. 2007. 91 f. Monografia (Licenciatura Plena de Física) – Centro de Ciência e Tecnologia da Universidade Estadual do Ceará, Fortaleza.

CROCKFORD, Douglas. **JavaScript: The Good Parts**. Sebastopol: O'Reilly Media Inc., 2008.

FERREIRA, Douglas S. et al. SimulAR: desenvolvimento de um software utilizando técnicas de realidade aumentada para simular fenômenos físicos. In: SBGAMES, 10., 2011, Salvador. **Procceedings.** Salvador: Sbc, 2011. p. 1 - 9. Disponível em: <http://www.sbgames.org/sbgames2011/proceedings/sbgames/papers/cult/full/92044_1.pdf>. Acesso em: 09 mar. 2015.

GAMMA, Eric et al. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Artmed, 2000. 364 p.

HALLIDAY, David; RESNICK, Robert; WALKER, Jearl. **Fundamentos de física**: Mecânica. 9. ed. Tradução Ronaldo Sérgio de Biasi. Rio de Janeiro: -Livros Técnicos e Cientíicos Editora Ltda., 2012.

HEDMAN, Stefan. **Github**: wiki CannonJS. 2013a. Disponível em: <https://github.com/schteppe/cannon.js/wiki>. Acesso em: 02 set. 2016.

_____. **Github**. Cannon.js Hello Cannon.js! 2013c. Disponível em: <https://github.com/schteppe/cannon.js/wiki/Hello-Cannon.js!>. Acesso em: 06 nov. 2016.

HENICK, Ben. **HTML & CSS: The Good Parts**. Sebastopol: O'Reilly Media Inc., 2010.

MILLINGTON, Ian. **Game physics engine development**: How to build a robust commercial grade physics engine for your game. 2. ed. Burlington: Morgan Kaufmann, 2010.

REIS, Dalton Solano dos. **Entrevista sobre problemas da biblioteca Hefesto atual**. Santa Catarina, FURB, 11 out. 2016. Entrevista verbal durante apresentação do pré-projeto de TCC.

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.

TEIXEIRA, Thiago H. **HEFESTO – Framework para simulações utilizando um motor de física em 3D**. 2015. 81 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

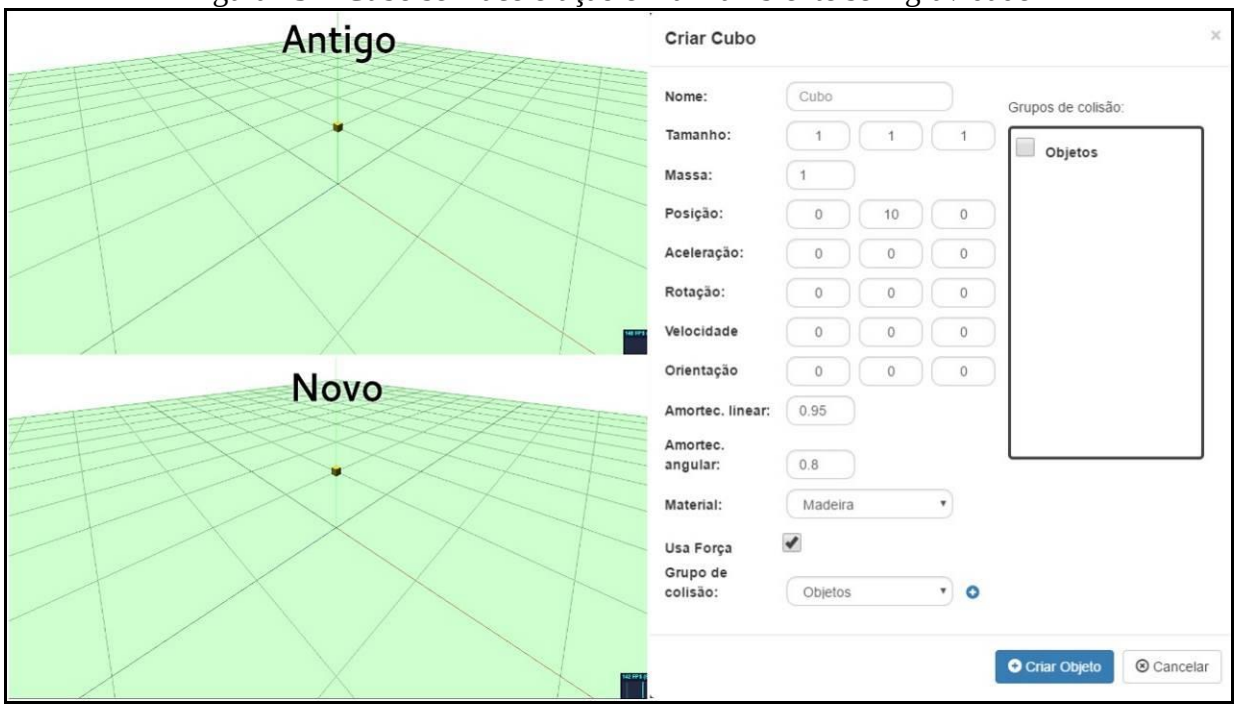
YASUSHI, Ando. **Box2DJs**. 2008. Disponível em <http://box2d-js.sourceforge.net/>. Acesso em: 03 abr. 2017.

ZANLUCA, Gabriel. **Ballistic**. 2015. Disponível em: <http://www.inf.furb.br/gcg/visedu/fisica/ballistic>. Acesso em: 27 ago. 2016.

APÊNDICE A – Testes funcionais realizados sobre o editor Hefesto

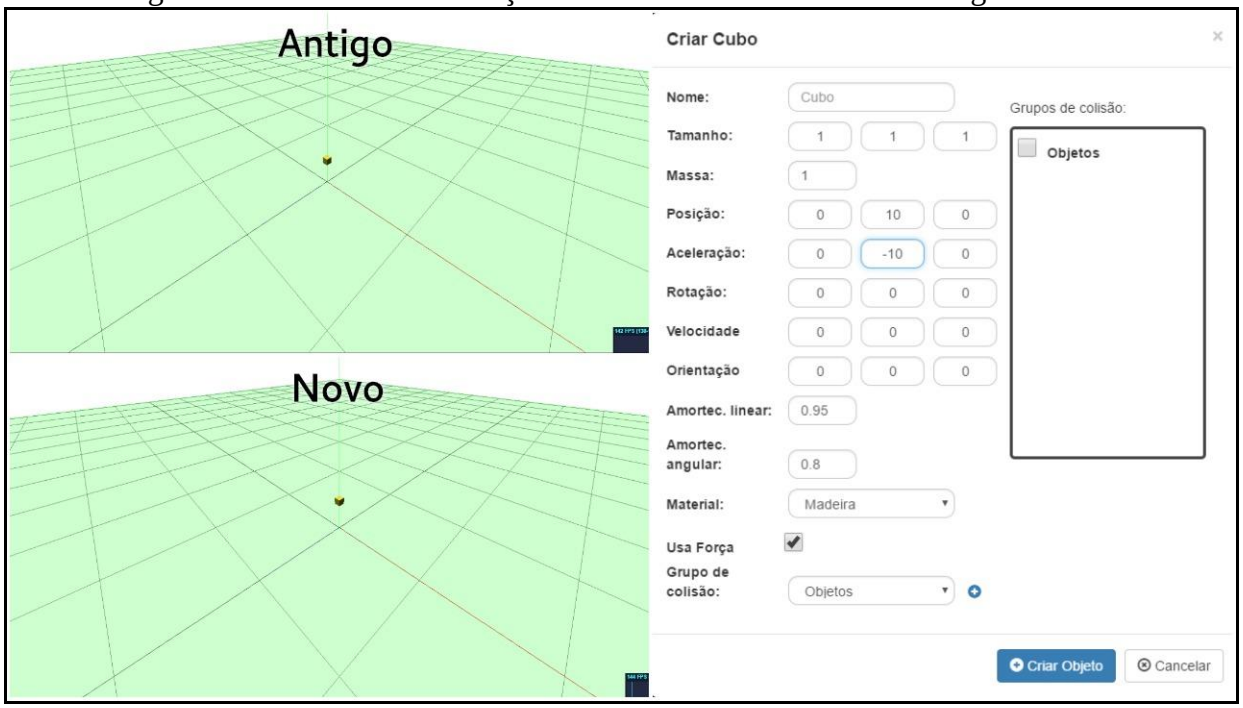
A seguir, nas Figuras 23 até 32, são apresentados os testes funcionais realizados sobre o editor Hefesto.

Figura 23 - Cubo sem aceleração em um ambiente sem gravidade



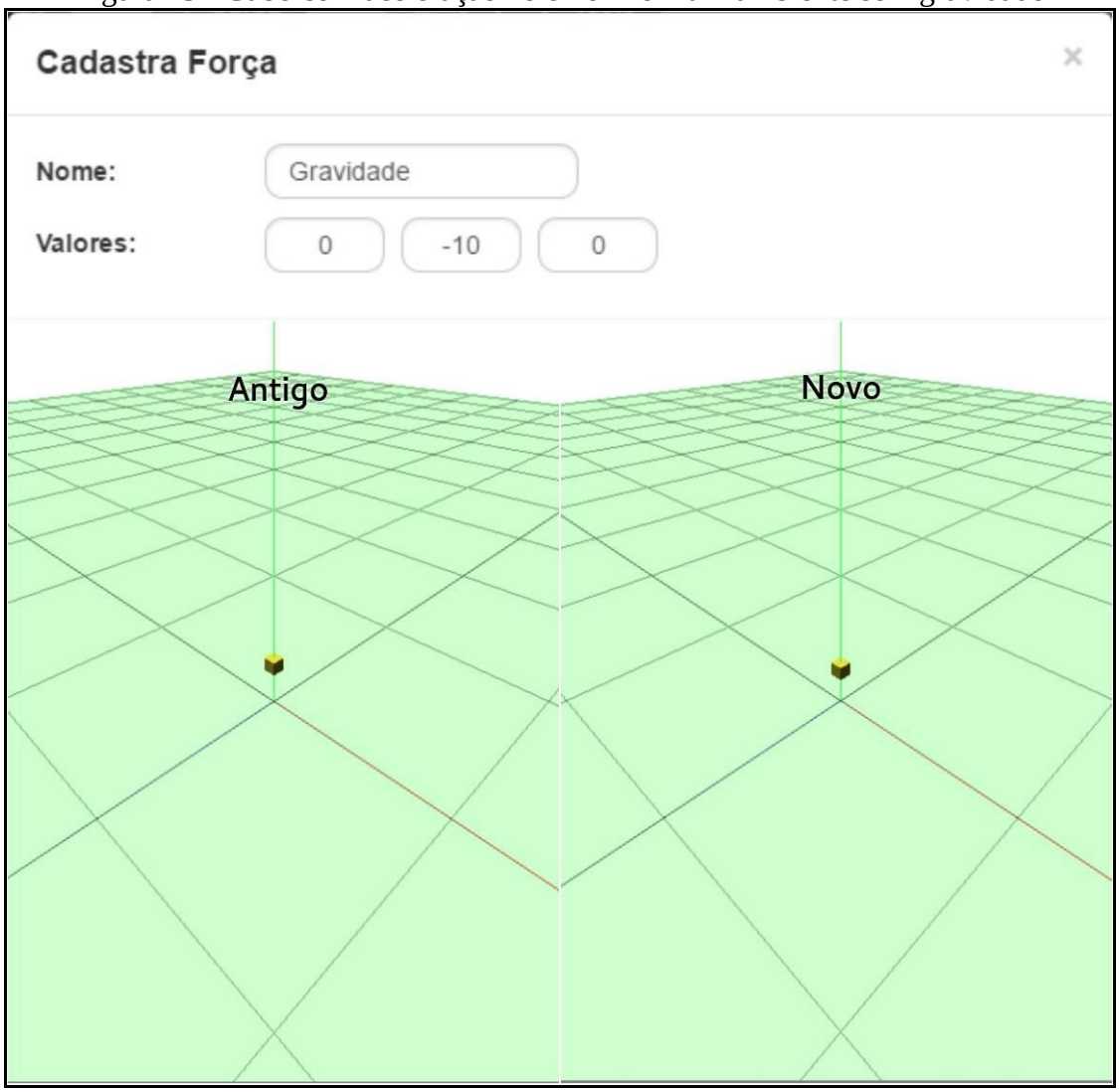
Fonte: elaborado pelo autor.

Figura 24 - Cubo com aceleração no eixo Y em um ambiente sem gravidade



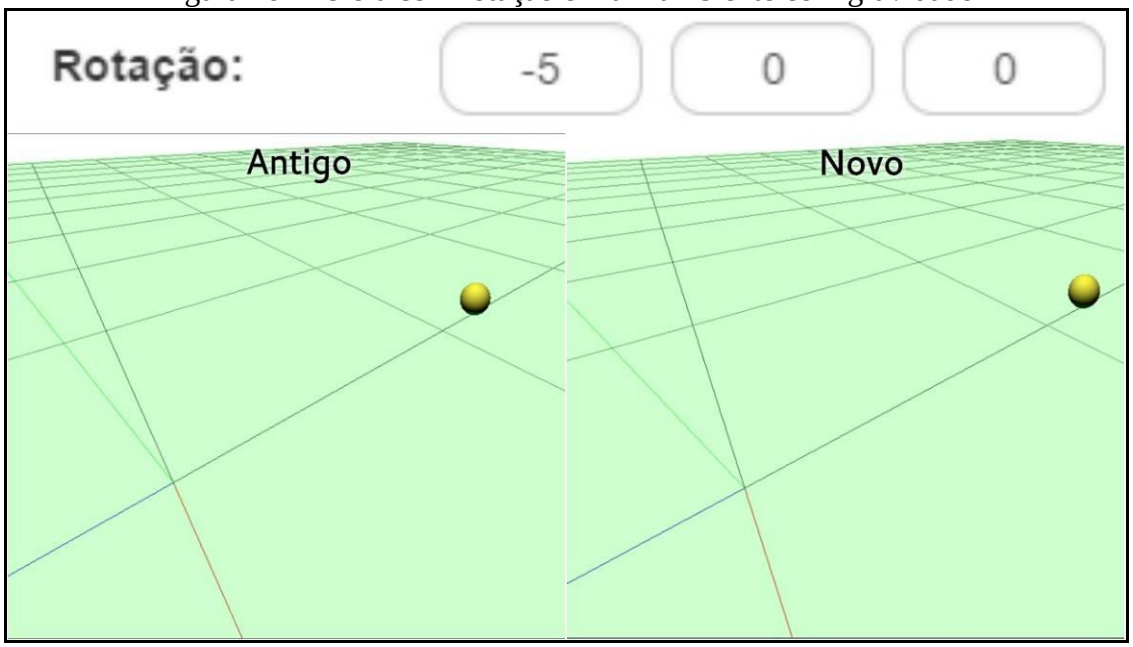
Fonte: elaborado pelo autor.

Figura 25 - Cubo com aceleração no eixo Y em um ambiente sem gravidade



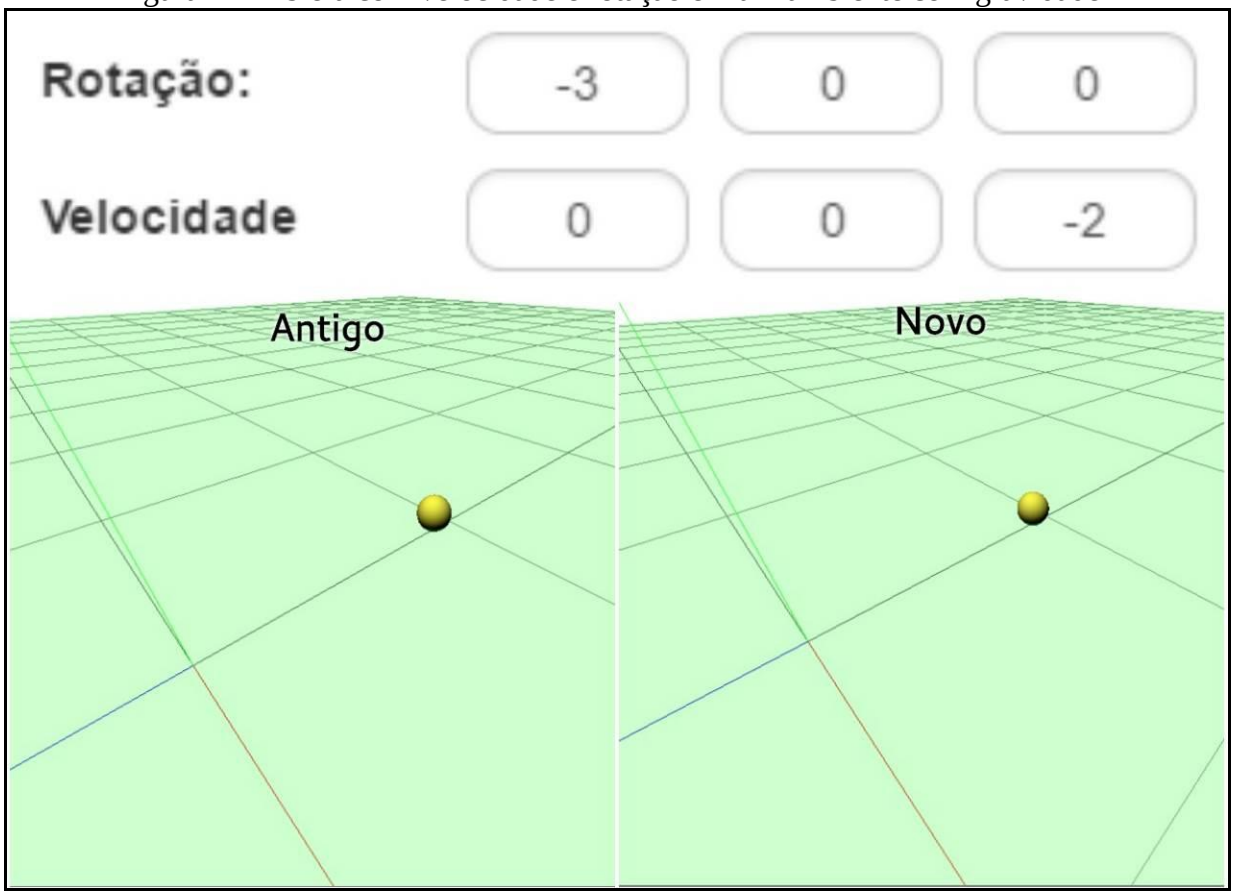
Fonte: elaborado pelo autor.

Figura 26 - Esfera com rotação em um ambiente com gravidade



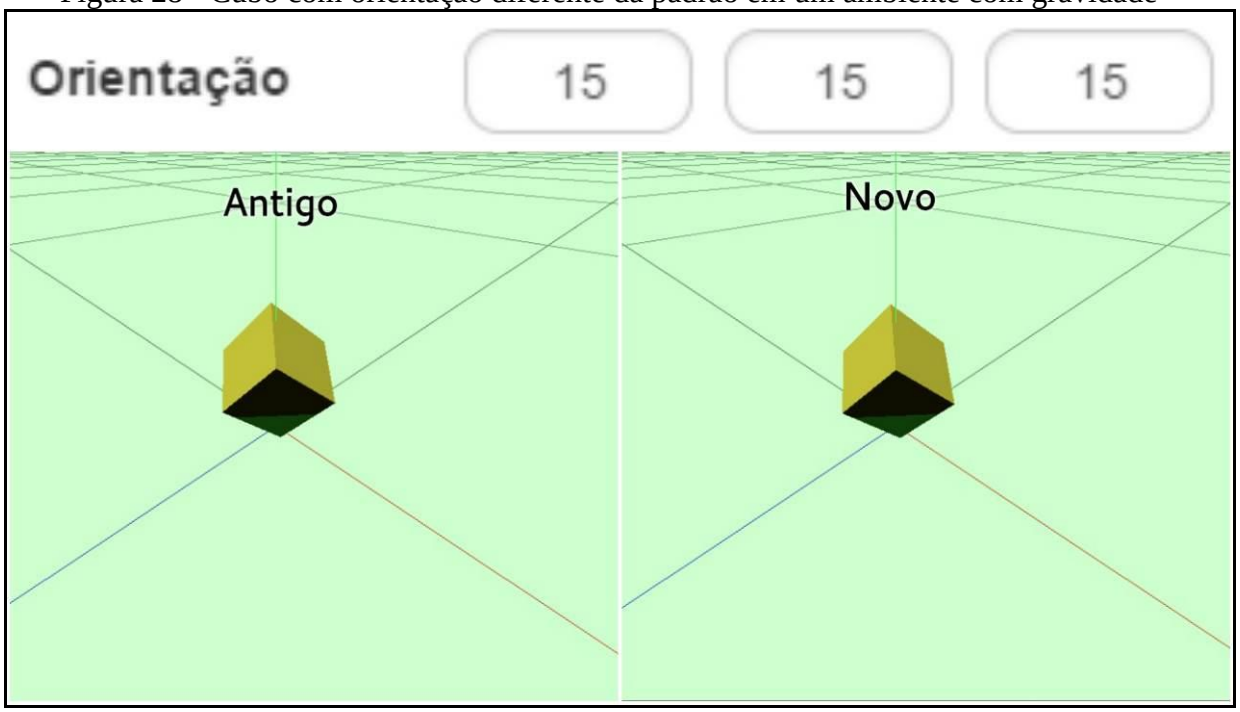
Fonte: elaborado pelo autor.

Figura 27 - Esfera com velocidade e rotação em um ambiente com gravidade



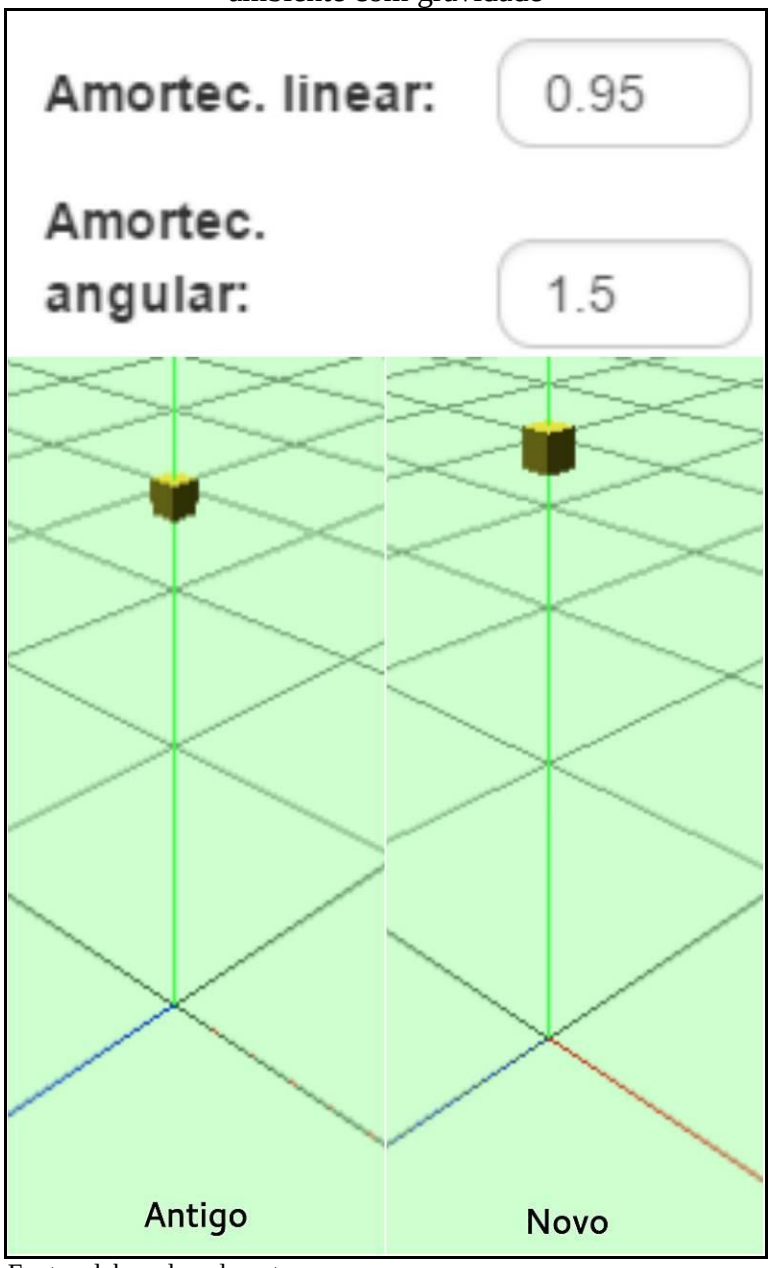
Fonte: elaborado pelo autor.

Figura 28 - Cubo com orientação diferente da padrão em um ambiente com gravidade



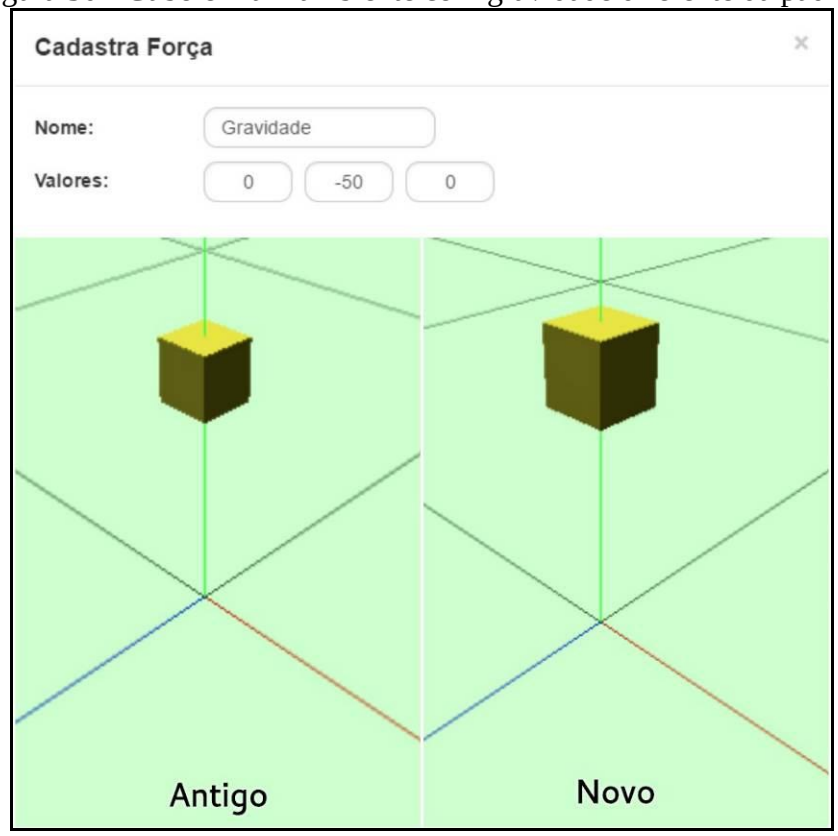
Fonte: elaborado pelo autor.

Figura 29 - Cubo com um amortecimento angular e linear diferente do padrão em um ambiente com gravidade



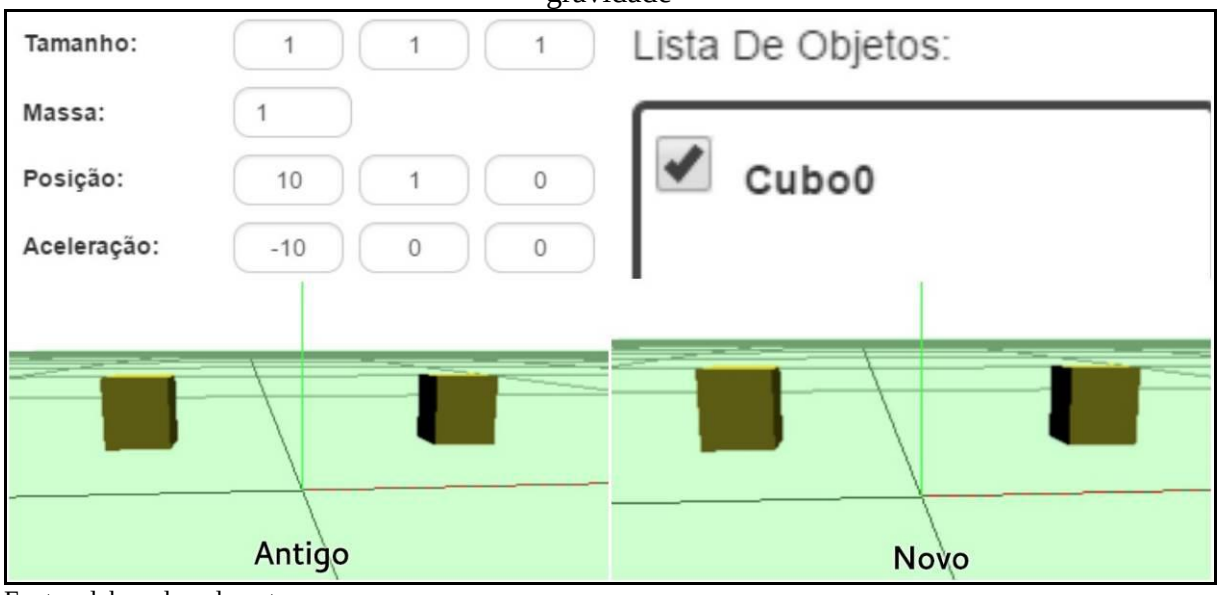
Fonte: elaborado pelo autor.

Figura 30 - Cubo em um ambiente com gravidade diferente da padrão



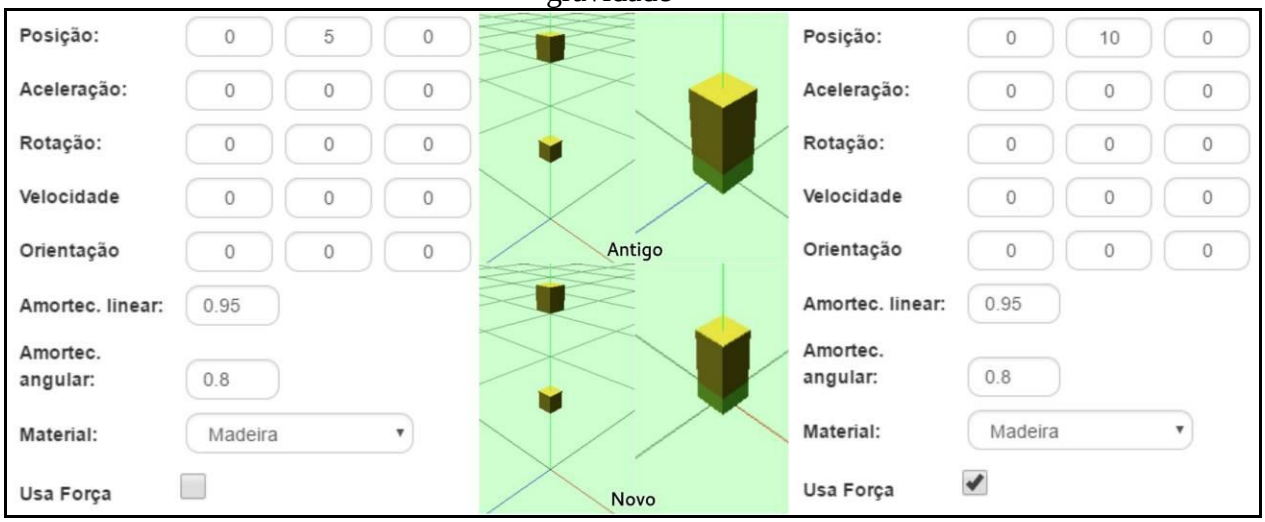
Fonte: elaborado pelo autor.

Figura 31 - Colisão entre dois objetos que colidem um com outro em um ambiente com gravidade



Fonte: elaborado pelo autor.

Figura 32 - Um objeto que não "Usa Força" colidindo com outro objeto em um ambiente com gravidade

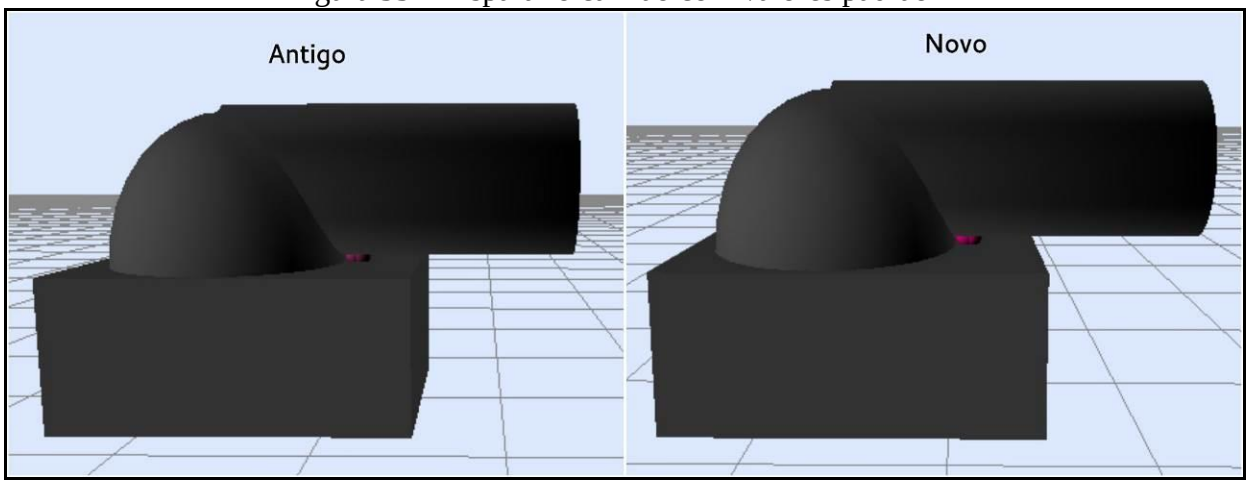


Fonte: elaborado pelo autor.

APÊNDICE B – Testes funcionais realizados sobre o Ballistic

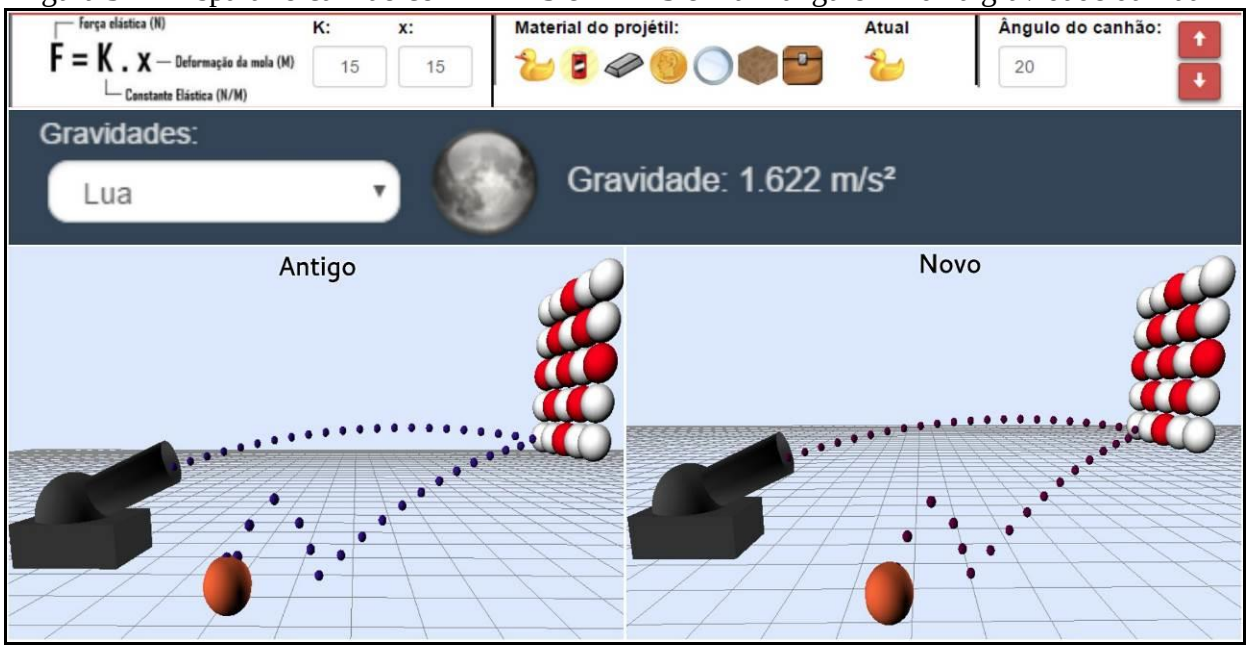
A seguir, nas Figuras 33 até 37 são apresentados os testes funcionais realizados sobre o Ballistic.

Figura 33 - Disparar o canhão com valores padrão



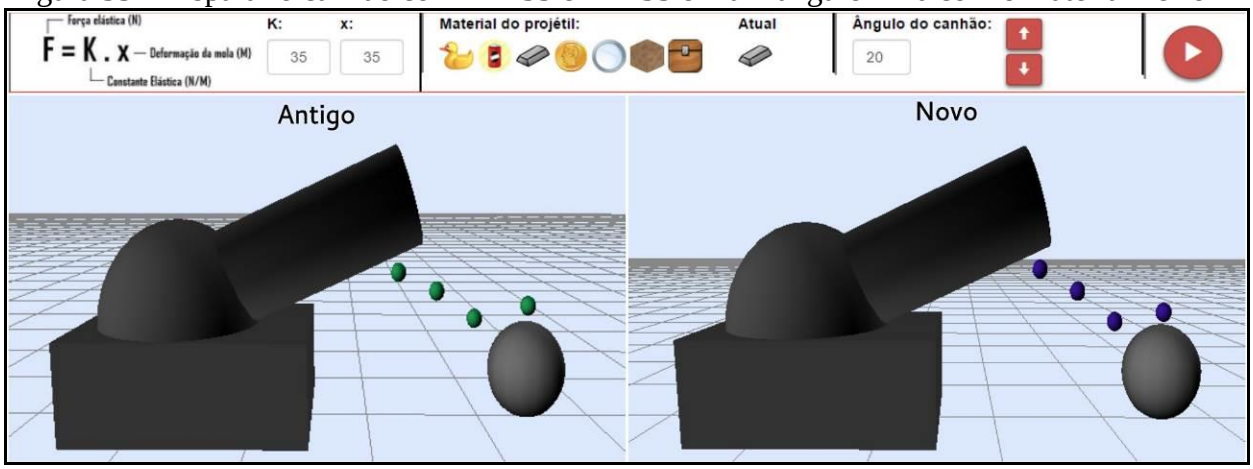
Fonte: elaborado pelo autor.

Figura 34 - Disparar o canhão com K = 15 e X = 15 em um ângulo = 20 na gravidade da Lua



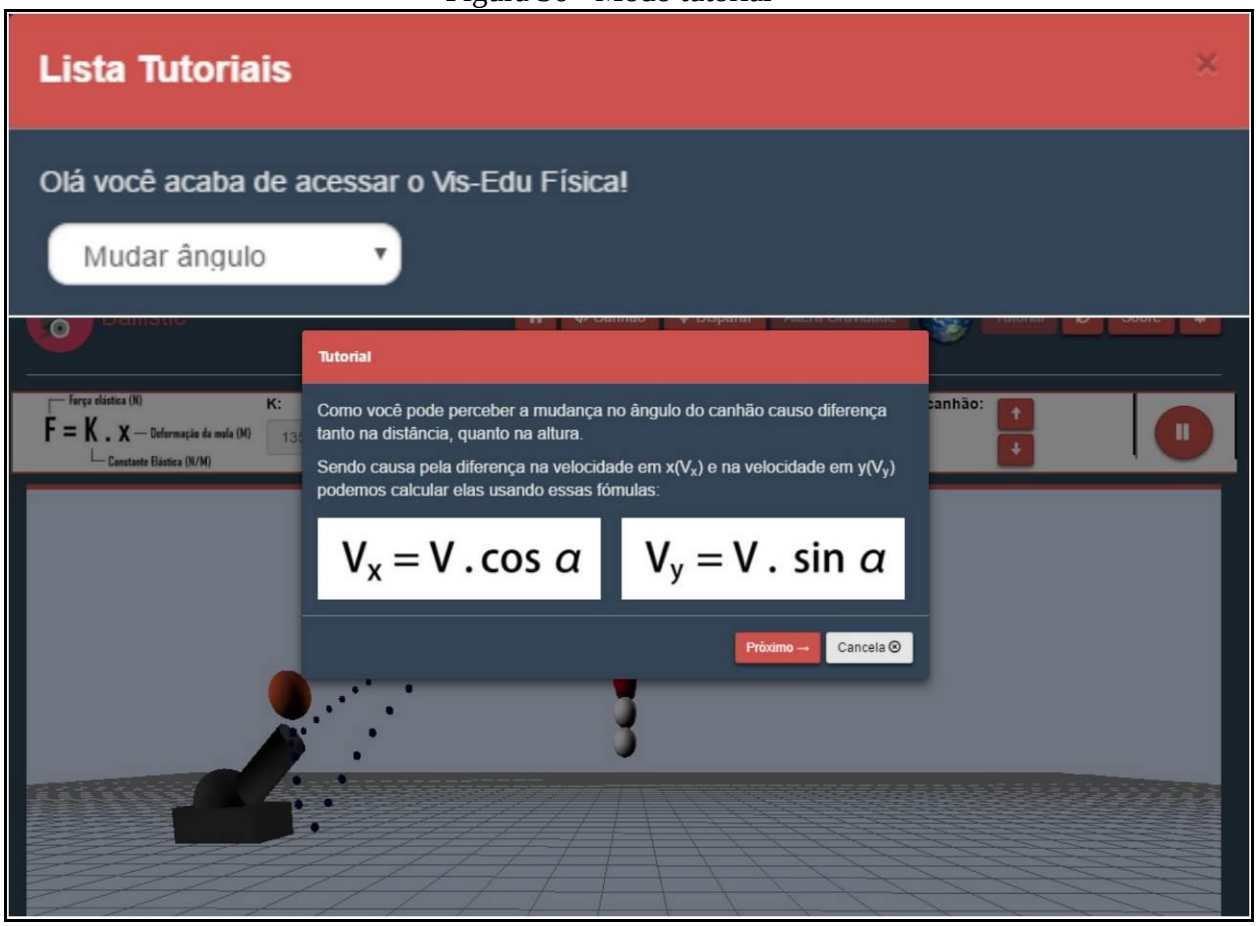
Fonte: elaborado pelo autor.

Figura 35 - Disparar o canhão com K = 35 e X = 35 em um ângulo = 20 com o material Ferro



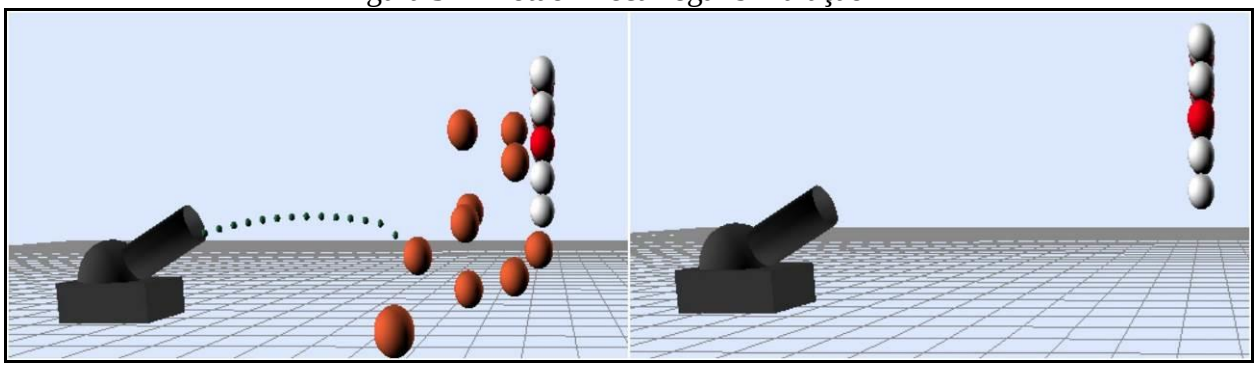
Fonte: elaborado pelo autor.

Figura 36 - Modo tutorial



Fonte: elaborado pelo autor.

Figura 37 - Botão "Recarregar simulação"



Fonte: elaborado pelo autor.