Universidade Federal do Rio de Janeiro
Escola Politécnica / COPPE

Experimentação em Simulações Físicas Interativas para Jogos

Autor:

João Pedro Schara Francese

Orientador:

Prof. Ricardo Guerra Marroquim, D. Sc.

Examinador:

Prof. Claudio Esperança, Ph. D.

Examinador:

Prof. Antonio Alberto Fernandes de Oliveira, D. Sc.

Poli / COPPE

Fevereiro de 2011

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica – Departamento de Eletrônica e de Computação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária.
Rio de Janeiro – RJ – CEP 21949-900

Francese, João Pedro Schara

Experimentação em Simulações Físicas Interativas para Jogos / João Pedro Schara Francese – Rio de Janeiro: UFRJ/POLI-COPPE, 2011.

XI, 42 p.: il.; 29,7 cm.

Orientador: Ricardo Guerra Marroquim

Projeto (graduação) – UFRJ/POLI/Departamento de Eletrônica e de Computação – COPPE, 2011.

Referências Bibliográficas: p. 40-42.

1. Computação gráfica. 2. Jogos eletrônicos. 3. Simulação física. I. Marroquim, Ricardo Guerra. II. Universidade Federal do Rio de Janeiro, Poli/COPPE. III. Título.

DEDICATÓRIA

Dedico este trabalho às pessoas que buscam, com honestidade e esforço próprio, transformar o mundo em um lugar melhor.

AGRADECIMENTO

Agradeço primeiramente aos meus pais Monica e Horacio, à minha irmã Maria Victoria, ao meu avô Paulo e ao resto de minha família pelo apoio e carinho recebido durante todo o meu período de estudos, antes e durante a faculdade.

Agradeço ao meu orientador, professor Ricardo Marroquim, que, além de ser o idealizador do tema do trabalho, foi fundamental para a elaboração deste projeto, seja de forma direta, com ideias e soluções para os problemas que encontramos, seja de forma indireta, resolvendo questões práticas ante os outros professores ou a secretaria quando eu não pude estar presente.

Agradeço aos meus sócios e amigos, em especial a Leonardo Arnt e Ulysses Vilela, por ajudarem a criar e desenvolver minha empresa Inoa Sistemas, através da qual eu pude aplicar os conhecimentos da graduação e adquirir uma valiosa experiência profissional.

Agradeço aos meus colegas e amigos do curso de Engenharia de Computação e Informação, inclusive os de turmas anteriores ou posteriores, com os quais troquei muito conhecimento e ajuda mútua, que estiveram presentes em momentos de diversão ou de necessidade, e que foram decisivos para moldar minha graduação, com destaque para Daniel Vega, Diogo Menezes, Gustavo Fernandes, Hugo Carvalho, João Luiz Ferreira, Pedro Coutinho, Pedro Pisa, Renan Bernardo, Roosevelt Sardinha, Thiago Xavier e Túlio Ligneul.

Agradeço ao programa Erasmus Mundus, da União Europeia, que me concedeu a chance de realizar um intercâmbio acadêmico na Universidad Politécnica de Valencia, Espanha, uma experiência sem igual em muitos aspectos. Agradeço também aos meus amigos lá conhecidos, em especial a Luciana Rêgo e Ana Paula Albuquerque, excelentes companheiras a milhares de quilômetros de casa.

Agradeço ao professor José Ferreira de Rezende, que assumiu a coordenação do curso de Engenharia de Computação e Informação durante minha graduação, e resolveu diversos problemas que poderiam impactar negativamente na minha formação.

Agradeço aos professores Guilherme Horta Travassos e Tayana Conte, que me orientaram durante o meu período de iniciação científica no Laboratório de Engenharia de Software.

Agradeço a todos os outros professores da UFRJ com quem eu tive aula, sem os quais minha formação não seria completa.

RESUMO

Projeto de Graduação apresentado à Escola Politécnica/COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação e Informação.

EXPERIMENTAÇÃO EM SIMULAÇÕES
FÍSICAS INTERATIVAS PARA JOGOS

João Pedro Schara Francese

Fevereiro/2011

Orientador: Ricardo Guerra Marroquim
Curso: Engenharia de Computação e Informação

Jogos eletrônicos precisam ser envolventes para cumprir satisfatoriamente sua função de divertir o usuário. Dentre as diversas possibilidades de imersão, uma delas consiste em atingir o maior nível de realismo possível na recriação virtual do mundo real. Este realismo também pode ser dividido em várias categorias; dentre elas, a tentativa de representar fielmente a física dos objetos dentro do jogo.

Este trabalho tem como objetivo implementar uma simulação física realista de objetos em um mundo virtual tridimensional, como elemento de um jogo de tiro em primeira pessoa. Através das técnicas utilizadas, é possível que o jogador tenha uma interação real com o cenário, ao invés de tê-los como elementos estáticos. Esta abordagem é usada com sucesso por alguns jogos comerciais modernos.

Para tal, foram utilizados alguns componentes e bibliotecas de código aberto, e diversas técnicas foram experimentadas, visando vencer o desafio típico das simulações em tempo real: qualidade da simulação *versus* tempo de processamento por quadro. Otimizações, sejam elas localizadas ou alterando a estrutura geral do sistema, são elementos importantes que compõem o trabalho, pois sem eles a simulação torna-se inviável para um jogo real.

Palavras-Chave: computação gráfica, jogos, física, simulação interativa.

ABSTRACT

Undergraduate Project presented to POLI/COPPE/UFRJ as a partial fulfillment of requirements for the degree of Computer and Information Engineer.

EXPERIMENTATION IN INTERACTIVE
PHYSICS SIMULATIONS FOR GAMES

João Pedro Schara Francese

February/2011

Advisor: Ricardo Guerra Marroquim
Course: Computer and Information Engineering

Videogames must be involving in order to fulfill satisfactorily their main function of entertaining the user. Among the diverse possibilities of immersion, one of them consists of attaining the highest possible level of realism in the virtual recreation of the real world. This realism can also be divided in various categories; among them, the attempt of representing faithfully the physics of in-game objects.

This project's objective is to implement a realistic physics simulation of objects in a tridimensional virtual world, as an element of a first-person shooter game. By the means of the used techniques, the player is able to engage in a true interaction with the scenery, which is no longer merely a static element. This approach has been used successfully by some modern commercial games.

To that end, some open-source components and libraries were used, and various techniques were experimented, in order to overcome the typical challenge of real-time simulations: simulation quality *versus* processing time per frame. Optimizations, both those localized and those affecting the overall structure of the system, are important elements of the project, as the simulation would be impractical for a real game without them.

Keywords: graphics computing, games, physics, interactive simulation.

SIGLAS

FPS: *first-person shooter* (tiro em primeira pessoa)

GPU: *graphics processing unit* (unidade de processamento gráfico)

IDE: *integrated development environment* (ambiente de desenvolvimento integrado)

ODE: Open Dynamics Engine

QPS: quadros por segundo

UML: Unified Modeling Language

Sumário

Lista de Figuras

Capítulo 1:

Introdução

1.1. Contextualização

Jogos eletrônicos (*video-games*) são um dos meios de entretenimento mais populares da atualidade. Com um faturamento de 57 bilhões de dólares no mundo em 2009 [1], este ramo já ultrapassou o faturamento das tradicionais indústrias da música e do cinema – respectivamente, de 17 bilhões [2] e 30 bilhões [3] no mesmo ano. Considerando apenas as vendas digitais, jogos eletrônicos possuem uma fatia de 35% do mercado, contra 20% do segundo colocado, o de gravadoras [4].

Neste cenário, torna-se evidente o valor que investimentos e pesquisas na área podem trazer. A evolução técnica é um dos motores propulsores deste mercado, e empresas buscam lançar ano a ano produtos que impressionem cada vez mais seus jogadores.

Apesar de um dia a atividade de criação de jogos ter sido um processo individual e artesanal, este quadro deixou, há mais de duas décadas, de representar a realidade da esmagadora maioria dos jogos com fins comerciais. Atualmente, as equipes de desenvolvimento das grandes produtoras são compostas por diversos membros, de *game designers* a programadores, de artistas 3D a compositores da trilha sonora, de gerentes de projeto a pessoas dedicadas exclusivamente a testar os jogos para encontrar defeitos.

O aumento da complexidade do desenvolvimento de jogos e o subsequente aumento das equipes fez com que a especialização se tornasse necessária. Embora seja importante deter um conhecimento horizontal, abrangente e superficial de todas as questões relacionadas à codificação dos jogos, um jogo de última geração requer profissionais com conhecimentos avançados atuando em cada segmento, seja em redes, inteligência artificial ou em interfaces de usuário.

Este projeto reconhece que esta é a realidade do cenário atual, e busca aprofundar em um item específico do processo de criação de jogos: a procura por imersão e realismo através de simulações físicas dos objetos presentes no mundo virtual. Apesar de possuir outros componentes que formam um jogo – interface com o usuário, efeitos sonoros, etc. – dedicou-se uma atenção relativamente muito maior à questão da física dentro do mundo do jogo.

1.2. Motivação

O sucesso da indústria de jogos fez com que a quantidade de lançamentos e de empresas produtoras de jogos crescesse e, por consequência, a competição pela atenção e preferência do consumidor. O padrão no nível de jogos sobe de forma constante, e é preciso encontrar novas formas de atrair o jogador.

Uma das formas de atingir este objetivo é fazer com que o jogo seja imersivo, envolvente. Existem inúmeros fatores que auxiliam na imersão em jogos, fatores estes que podem ser inerentes ao mundo ficcional do jogo – história, etc. – chamados de fatores diegéticos [5], ou externos a este, tais como controles interativos, ambientação sonora e realismo nos gráficos.

Paralelos ao realismo gráfico estão outras formas de realismo. O realismo físico, pertencente a essa categoria, é pivô deste trabalho, e significa que os objetos do mundo virtual reagem à interação com o usuário e com os outros objetos seguindo as leis da física do mundo real – ou, ao menos, aparentam fazê-lo de forma convincente. O vídeo demonstrando a utilização da biblioteca PhysX no jogo Batman: Arhkan Asylum [6] evidencia o impacto que pequenas características de simulação realista têm na experiência do jogador.

Figura 1: Um mapa baseado em física de Crysis [7].

Diversos jogos comerciais modernos fazem uso do realismo físico como fator fundamental em seu apelo comercial. O jogo Portal [8], lançado em 2007, utiliza a física como elemento principal de quebra-cabeça do jogo, chegando a abolir armas

convencionais, apesar de seu formato de tiro em primeira pessoa (FPS). Crysis [9], outro FPS de 2007, recebeu muito destaque devido ao seu motor físico avançado. Esta publicidade deveu-se a dois fatores: a qualidade da simulação, com realismo até então jamais visto (referência na área até os dias atuais), e os pesados requerimentos de *hardware* exigidos pelo jogo – nenhum computador disponível na data do lançamento era capaz de executar o jogo em suas configurações máximas.

Como o próprio exemplo anterior mostra, o realismo da simulação é um fator importante, mas igualmente importante é a velocidade com que tal simulação é feita. Um jogo é, por definição, uma mídia interativa, e isso exige que a simulação seja feita em tempo real. Estes dois quesitos são inversamente proporcionais, e é preciso obter um equilíbrio para que a satisfação do jogador atinja seu máximo; este é o maior desafio no campo das simulações, e torna-se necessário o uso de técnicas especiais para atingir o ponto ótimo dessa curva.

1.3. Objetivo

Este trabalho tem como foco primário implementar uma *demo* (versão com escopo restrito) de um jogo de tiro em primeira pessoa, a fim de demonstrar reações físicas realistas entre os objetos do jogo sem, no entanto, perder a característica da interatividade em tempo real. O jogo deve ser capaz de ser executado satisfatoriamente em *hardware* de médio porte da geração atual.

Através desta implementação, espera-se atingir o objetivo secundário, que é publicar um estudo das técnicas e ferramentas utilizadas. Apesar de comercialmente em voga, a produção acadêmica na área de simulações físicas interativas, e de terrenos destrutíveis em especial, é esparsa.

Espera-se que este trabalho possa servir de base para pesquisadores futuros que queriam estudar o tema, e neles despertar o interesse para aprofundar a pesquisa e gerar novas e mais eficientes técnicas.

1.4. Metodologia

O primeiro passo no desenvolvimento do trabalho foi a seleção das tecnologias a serem usadas. A disponibilidade do código-fonte e a permissão de uso tanto em projetos com fins educativos quanto com fins comerciais foram adotados como critérios eliminatórios. Dentre as opções disponíveis, foram selecionadas aquelas que

apresentavam facilidade de uso, adoção em projetos já existentes e boa documentação *online*.

Em seguida, foi definida a funcionalidade-alvo. A partir da proposta em sua forma ampla (simulação física realista e interativa), um escopo restrito foi escolhido para aprofundamento: a destruição dinâmica do cenário, cuja expressão prática é a criação de uma parede que pode ser afetada pelos tiros da arma do jogador.

A etapa seguinte correspondeu à seleção dos critérios de medição, e suas metas associadas. O quesito qualidade foi definido de forma subjetiva, como o realismo físico percebido nos objetos durante a simulação. O quesito velocidade foi definido através da métrica objetiva "quadros por segundo" (QPS), ou seja, o número de imagens que o sistema composto de simulação física e renderização gráfica era capaz de gerar a cada segundo. Estipulou-se um mínimo de 30 QPS para a situação de repouso (inicial), e um mínimo de 15 QPS após a destruição de um bloco unitário do cenário (tijolo). Considerou-se ainda desejável, mas não obrigatório para a aprovação do sistema, manter a taxa de ao menos 15 QPS ao longo de toda a simulação.

Por fim, seguiu-se a etapa de desenvolvimento, no qual diversas técnicas foram testadas a fim de se atingir as metas propostas. A cada iteração, o jogo era verificado novamente para ver se a avaliação nos quesitos havia melhorado ou piorado, e a mudança era mantida ou não dependendo do resultado.

1.5. Estrutura do documento

No próximo capítulo, são apresentadas as bibliotecas de *software* utilizadas no projeto – Panda3D e ODE – incluindo a razão pela qual foram escolhidas, a forma com que foram usadas, e informações sobre seu funcionamento.

O capítulo 3 detalha a modelagem física utilizada internamente e externamente pelo motor físico do sistema, de forma que o leitor seja, ao seu término, capaz de compreender a interface, as capacidades e as limitações da simulação física do jogo.

A implementação realizada no projeto é apresentada em detalhes no capítulo 4, com destaque para as aproximações, suposições e otimizações feitas para que fosse possível atingir um equilíbrio entre qualidade e velocidade. Este capítulo fala, ainda, dos aspectos não físicos do jogo, tais como as formas de interação com o usuário.

O quinto capítulo traz as conclusões obtidas a partir da realização do trabalho.

Por fim, o capítulo 6 discute possibilidades de evoluções e trabalhos futuros que podem ser feitos com base no projeto desenvolvido.

Capítulo 2:

Bibliotecas Utilizadas

2.1. Aspectos gerais

Bibliotecas são componentes que permitem o compartilhamento e o reuso de código através de diferentes projetos. Em uma situação normal, é uma boa prática utilizar uma biblioteca já existente que atenda suas necessidades em vez de escrever seu próprio código. Existem situações na qual é aceitável não seguir essa regra, mas são ocasiões raras, tais como incompatibilidade de licenças ou para fins educativos.

O motor de um jogo (*game engine*) é um sistema de software que funciona como a base para a criação de um novo jogo, oferecendo componentes de alto nível que implementam funções necessárias, ocultando detalhes de baixo nível [10]. Um motor possui uma API (*application programming interface*), ou seja, uma interface de código bem definida para acesso às suas bibliotecas, e um SDK (*software development kit*), composto por coleções de bibliotecas, APIs e ferramentas auxiliares.

Existem incontáveis motores de jogos, com licenças variadas (aberta, gratuita, paga), para plataformas distintas (ou multiplataforma), em diferentes linguagens de programação, com funções de mais alto ou baixo nível, especializando-se em algum aspecto da criação de jogos ou com funcionalidades gerais. Este trabalho faz uso de dois motores: um para os aspectos gerais do jogo, incluindo a parte gráfica, e outro voltado somente para a simulação física.

O primeiro passo no desenvolvimento do trabalho foi a seleção das tecnologias a serem usadas. A disponibilidade do código-fonte e a permissão de uso tanto em projetos com fins educativos quanto com fins comerciais foram adotados como critérios eliminatórios. Dentre as opções disponíveis, foram selecionadas aquelas que apresentavam facilidade de uso, adoção em projetos já existentes e boa documentação *online*.

O principal critério para seleção das bibliotecas para este trabalho foi a disponibilidade de código-fonte online. Este critério está alinhado com o objetivo do trabalho de prover um estudo sobre simulações físicas em jogos, pois com este pré-requisito é possível ler o código das bibliotecas para entender seu funcionamento. Adicionalmente, as bibliotecas deveriam ter uma licença compatível com a definição de

"software livre" [11], garantindo que seria possível, legalmente, alterar o código para corrigir erros, e que este estaria sempre disponível para consultas futuras, mesmo no caso do mantenedor perder o interesse pela biblioteca – risco real em caso de sistemas proprietários.

O segundo critério fundamental é a possibilidade de executar os jogos criados em múltiplas plataformas. Não era aceitável, para um trabalho acadêmico, a exigência de plataformas proprietárias tais como Microsoft Windows ou OS X (Mac). Por outro lado, era desejável que o aplicativo final pudesse ser executado em tais plataformas, que representam a fatia dominante do mercado.

A existência de documentação *online* foi outro ponto importante considerado. Guias introdutórios, referências de API e comentários de uso são fundamentais para o aprendizado de uma nova biblioteca. Também relacionado a este ponto foi a exigência de uma comunidade ativa de usuários, capaz de engajar em discussões sobre técnicas de implementação e responder dúvidas de problemas encontrados quando necessário.

Por fim, buscou-se usar motores já consagrados, com uso em projetos (para fins comerciais ou não) já completos, com o objetivo de evitar usar uma biblioteca obscura, incompleta ou excessivamente instável e cheia de defeitos não descobertos.

Mais detalhes sobre a seleção de cada biblioteca podem ser encontrados nas respectivas seções.

2.2. Panda3D

O Panda3D [12] é uma biblioteca que engloba um motor de jogos tridimensionais e um arcabouço para desenvolvimento de jogos. Seu foco é no aspecto gráfico, permitindo que o programador utilize técnicas modernas da computação gráfica em seus jogos, tais como *shaders* e outras técnicas avançadas de renderização. Porém, possui uma curva de aprendizado tranquila, e programadores iniciantes são capazes de criar jogos simples sem grande esforço. Por esta razão, é utilizado em diversos cursos de introdução ao desenvolvimento de jogos, tal como o da própria Universidade Federal do Rio de Janeiro [13].

Além de classes e funções para manipular os gráficos do jogo, diversas funcionalidades auxiliares estão disponíveis para facilitar o trabalho do programador: gerenciamento de tarefas e de temporização (*timing*), máquinas de estado, física e detecção de colisão, inteligência artificial, redistribuição, áudio e rede, entre outros. Não há, evidentemente, grande ênfase nem profundidade em vários desses campos, mas o

arcabouço provê o básico necessário para a criação de um jogo completo, e sempre permite o uso de bibliotecas externas especializadas quando necessário.

A biblioteca utiliza o conceito de tarefas (*tasks*), funções que são chamadas a cada iteração do laço principal do aplicativo, para a temporização do jogo. Tarefas podem ser criadas arbitrariamente, e nelas devem ser colocadas pelo programador o código que atualiza o estado do jogo, a verificação de comandos efetuados pelo jogador e quaisquer outras rotinas que devem ser executadas continuamente.

O Panda3D foi desenvolvido inicialmente pela Disney para uso em atrações de realidade virtual dos seus parques. Teve então seu código aberto em 2002, e a evolução da biblioteca passou a receber a colaboração do Entertainment Technology Center da universidade Carnegie Mellon. Atualmente é disponibilizada através da licença BSD [14], que permite o uso tanto em projetos de software livre quanto para fins comerciais.

2.2.1. **Linguagem**

Internamente, o motor Panda3D foi desenvolvido em C++, resultando em uma implementação eficiente e portável através de plataformas. A linguagem principal usada no desenvolvimento de jogos, porem, é Python [15], o que se traduz em diversas vantagens e algumas desvantagens. Todos os elementos da API (classes, funções, etc.) são disponibilizados completamente em Python através de um gerador automático de *wrappers*.

Todos os lados positivos e negativos de uma linguagem interpretada – facilidade de testar novos trechos de código, redução no desempenho – se fazem presentes devido ao uso de Python. Porém, no geral, a parte computacionalmente mais custosa não está no código do desenvolvedor do jogo, e sim no código do motor (responsável pela renderização, por exemplo) que, por ser escrito em C++, não está no sujeito a essas penalidades. A velocidade na criação de novas iterações do código e a sintaxe clara e com pouca poluição visual do Python, porém, são bem-aproveitadas por atuar precisamente sobre o código do usuário da API.

É importante notar que a troca de contexto do Python para o C++, ainda que relativamente pouco custosa perto do processamento efetuado internamente no motor, não é totalmente livre de custos. O uso excessivo de *callbacks*, em especial nas funções de tratamento de colisões, pode tornar o jogo perceptivelmente mais lento, pois a troca de contexto passa a ocorrer milhares de vezes a cada quadro. Deve-se evitar o uso de *callbacks* quando não são necessários, reorganizando a lógica do código com esse fim.

2.2.2. **Seleção**

Panda3D foi escolhido como o motor 3D deste trabalho por atender aos critérios estabelecidos: é publicado pela licença BSD e está disponível em diversas plataformas. Além disso, ainda está sob desenvolvimento ativo, possui uma documentação extensa e uma comunidade atuante. Exemplos de jogos comerciais que o utilizam incluem Disney's Pirates of the Caribbean Online e Aladdin Pinball [16].

O fator decisivo na escolha da biblioteca foi a possibilidade de usar a linguagem Python, facilitando o desenvolvimento iterativo esperado para as experimentações deste trabalho, sem, no entanto, perder significativamente em desempenho devido à sua implementação em C++. Dentre os motores 3D para Python, Panda3D é um dos que apresentam maior maturidade e um conjunto de funcionalidades mais completo.

2.2.3. **Grafo de cena**

O Panda3D é um motor baseado em um grafo de cena (*scene graph*), ou seja, o mundo virtual é formado por um grafo composto de uma ou mais árvores de nós que contém os objetos [17]. Estes podem ser inseridos pelo programador em qualquer ponto da árvore, reposicionados como filhos de outro nó, ou removidos da cena.

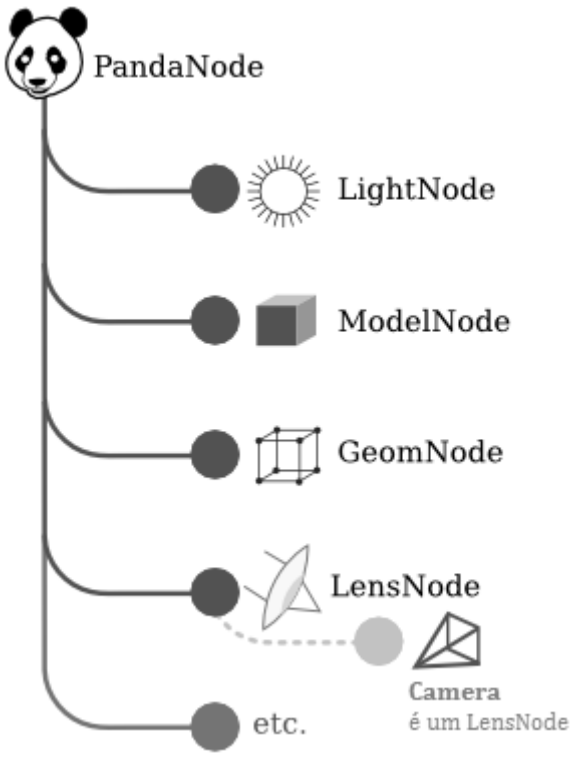

Figura 2: Tipos de PandaNodes. Adaptado de [18].

Um nó é a uma instância da classe *PandaNode* ou suas derivadas; pode corresponder a um modelo tridimensional carregado, a uma luz, a uma figura

geométrica, ou mesmo um elemento vazio. É uma entidade abstrata, cuja finalidade real depende da subclasse à qual pertence.

O programador não trabalha diretamente com nós do tipo *PandaNode* (ou suas subclasses); ele manipula instâncias da classe *NodePath*, que contém um ponteiro para um único *PandaNode* e dados que determinam a posição do nó no grafo da cena – um *PandaNode* em si não sabe em que ponto da árvore ele está. No entanto, o uso da classe *NodePath* é transparente, e a maioria dos métodos pode ser utilizada diretamente sobre esse objeto.

Alguns nós são inseridos na cena automaticamente pelo motor no início do jogo. O nó padrão *render* é a raiz da árvore que contém os objetos a serem desenhados na cena tridimensional; um modelo que não seja colocado em um ramo dessa árvore não será visível. Objetos que devem ser desenhados de forma fixa, bidimensional, na tela (e.g. elementos indicativos do HUD – *heads up display* – como a energia do personagem) devem ser colocados em uma árvore separada, cuja raiz é o nó padrão *render2d*. Outro nó padrão importante é o *camera*, cuja posição e rotação tomam parte na definição de qual parte da cena será vista pelo jogador. Maiores informações sobre estes e outros nós padrão podem ser encontradas na documentação do Panda3D [17].

A principal característica do grafo de cena é a propagação de transformações e atributos através da árvore. Um nó recebe todas as transformações e atributos definidos explicitamente sobre ele e sobre seus pais, até a raiz da árvore; transformações (e.g. translações) são combinadas, enquanto que atributos são substituídos. Isto permite que alterações sejam feitas rapidamente em um grande grupo de objetos, bastando defini-las em um nó que seja pai de todos eles – para esta finalidade, é comum criar nós que são objetos vazios, cujo objetivo é apenas agrupar nós definindo-os como seus filhos.

Outra vantagem é que deslocamentos podem ser quebrados em transformações menores, individuais. Um nó que representa uma mão pode estar atrelado ao nó que representa o braço de um indivíduo; a mão pode ser rotacionada individualmente, mas, ao mover o braço, a mão será movida automaticamente conforme esperado (e sem perder a rotação definida anteriormente).

2.3. Open Dynamics Engine (ODE)

O ODE [19] é uma biblioteca que expõe um motor de simulação física, acoplado a um detector de colisões entre objetos. É apropriada para modelar de forma realista ambientes de realidade virtual, tais como veículos (v. Figura 3), criaturas articuladas

(e.g. robôs) e objetos móveis. É um software livre, e seu código é disponibilizado através das licenças BSD [14] e GNU Lesser General Public License [20].

O ODE realiza apenas a computação, sendo independente do motor gráfico utilizado para visualizar os objetos do mundo virtual. As estruturas de dados que representam o mundo e os objetos em termos físicos não possuem relação com as estruturas que representam os polígonos que são desenhados na tela, ficando a cargo do programador criar novas estruturas que façam a ponte entre esses dois aspectos.

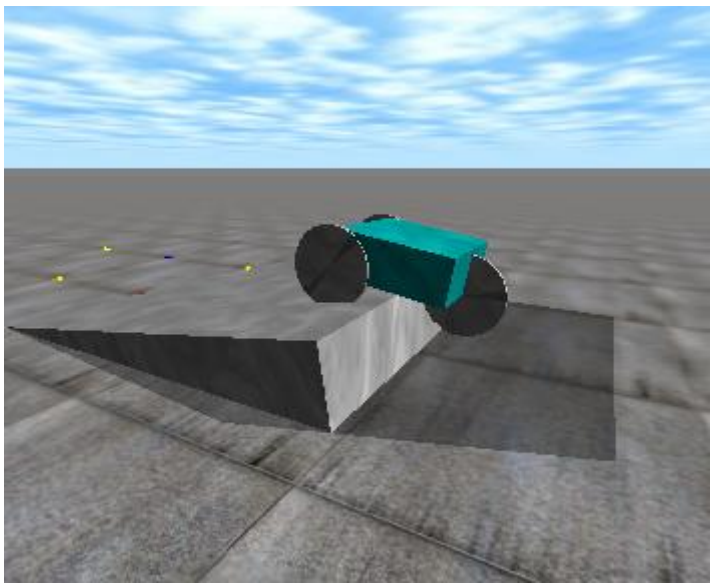

Figura 3: Demo "buggy" distribuído com o ODE [21].

O motor permite ainda que o desenvolvedor utilize uma biblioteca alternativa para a detecção de colisões (ou desenvolva seus próprios métodos para isso). O único requisito é que esta biblioteca forneça ao ODE as informações sobre quais pontos de cada objeto estão em contato com quais pontos de outros objetos, em um formato específico.

O ODE propicia uma simulação interativa, ou seja, rápida o bastante para que uma pessoa consiga controlá-la. Porém, ele não provê uma simulação em tempo real, cuja definição estrita é "uma que garante que o sistema seja atualizado um número fixo de vezes por segundo" [22]; o tempo de cada passo da simulação depende das características do sistema sendo simulado (número de objetos, quantidade de colisões ocorrendo, etc.).

A modelagem física adotada pelo ODE é apresentada em detalhes no próximo capítulo.

2.3.1. **Linguagem e integração**

O ODE é feito em C++, e possui *bindings* (interface para acesso em outra linguagem de programação) para Python na forma da biblioteca independente PyODE [23]. Porém, como o Panda3D possui integração nativa ao ODE, não foi necessário o uso dessa biblioteca. No entanto, a seção relativa ao ODE na documentação do Panda3D é pobre, e por essa razão a documentação do PyODE foi usada como referência, conforme sugestão da própria documentação do Panda3D [24].

É importante notar que o Panda3D embute uma versão antiga da biblioteca ODE em sua distribuição, e nem todas as funções existentes no ODE estão disponíveis através da interface Python do Panda3D (e.g. obter o vetor de forças acumuladas sobre um corpo). Este foi um empecilho encontrado algumas vezes durante a elaboração do trabalho, e requereu a elaboração de soluções criativas e caminhos alternativos para realizar tarefas que deveriam ser supostamente simples e diretas.

2.3.2. **Seleção**

O ODE foi escolhido como o motor físico do projeto por atender aos requisitos definidos (código aberto, uso real, disponibilidade de documentação, multiplataforma) e por estar presente na distribuição padrão do Panda3D, evitando assim que problemas de integração causassem distrações ao objetivo principal do trabalho.

Dentre os produtos já publicados que usam este motor, é possível citar os jogos comerciais Call of Juarez e World of Goo, além de alguns simuladores robóticos. Uma lista com outros aplicativos está disponível em [25].

Capítulo 3:

Modelagem Física

3.1. Visão Geral

Computadores têm poder de cálculo finito e, por isso, são incapazes de reproduzir com total fidelidade o mundo real, por mais rápido que sejam as máquinas escolhidas para realizar uma tarefa. Portanto, um motor de simulação física não conseguirá gerar resultados iguais à física verdadeira; é necessário realizar uma série de aproximações e suposições para que os cálculos possam ser feitos em tempo factível. Tais aproximações da modelagem física são o que determinam o poder de expressão do motor e os casos em que ele possui boa aplicabilidade.

É possível dividir a simulação da física em cinco passos [22]:

- Entender o sistema físico que se quer simular; este pode ser um sistema climático, mecânico, de comunicação ou de manobras militares.
- Modelar o sistema através de equações.
- Escrever um algoritmo de simulação, ou seja, um método para resolver as equações e determinar as mudanças que o sistema sofre ao longo do tempo.
- Escrever um programa de computador para implementar o modelo e o algoritmo do sistema.
- Executar a simulação.

Dentro desse arcabouço, o ODE é um motor pensado para simular sistemas mecânicos, modelando a dinâmica (movimento e interações) de corpos rígidos articulados [26]. Os principais componentes de tais sistemas, explicados em detalhes nas seções deste capítulo, são corpos rígidos (objetos sólidos), junções (dobradiças, encaixes, etc.) e dispositivos (molas, etc. – não disponíveis no ODE); outros itens abstratos que afetam a simulação são a fricção e a colisão entre objetos. Por se tratar de um sistema dinâmico, a base das alterações na simulação física são as forças, as quais geram velocidades lineares e angulares que, por sua vez, modificam a posição e a rotação dos objetos.

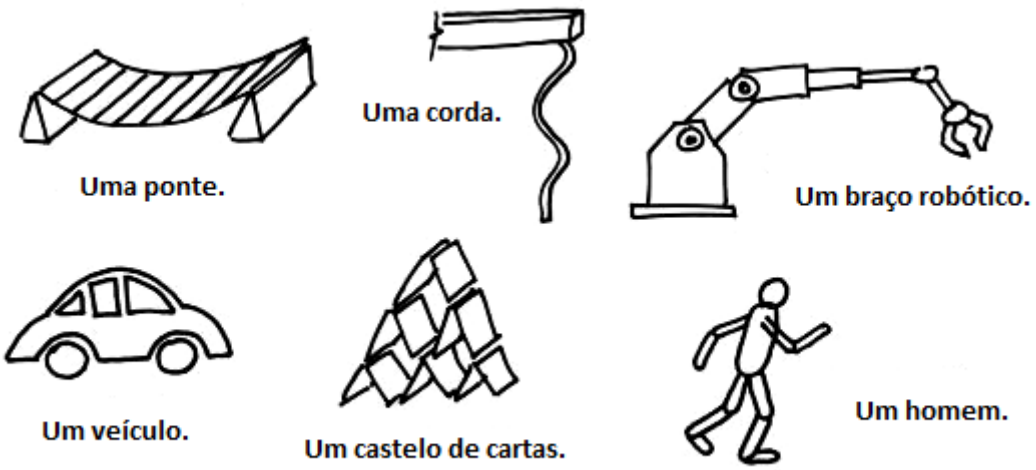

Figura 4: Exemplos de sistemas apropriados para simulação pelo ODE. Adaptado de [22].

A estrutura do código mínima em um aplicativo que use ODE se encontra abaixo (adaptado de [26]):

- Criar um mundo dinâmico (*dynamics world*) do ODE.
- Criar corpos (objetos) no mundo dinâmico e definir seus estados, inclusive as posições e velocidades.
- Criar junções no mundo dinâmico, ligá-las aos corpos e definir seus parâmetros.
- Criar um mundo de colisão (*collision world*) e geometrias de colisão para os objetos.
- Criar um grupo de junções para armazenar as junções de contato (*contact joints*) que detêm as informações de colisão.
- Rodar em laço, até o fim do jogo, as etapas abaixo, integrando com os laços dos outros motores do jogo conforme apropriado:
    - Aplicar forças aos objetos.
    - Ajustar os parâmetros de junções conforme necessário.
    - Chamar as rotinas de detecção de colisão do ODE (ou externas).
    - Criar uma junção de contato para cada ponto de colisão indicado e colocá-la no grupo de junções criado para esse fim.
    - Rodar um passo de simulação do ODE. Isto cria novas forças, com o objetivo de satisfazer as restrições das junções, e a partir das forças resultantes, determinar novas velocidades, posições e rotações dos objetos.
    - Remover todas as junções do grupo de junções de contato

É interessante notar que o Panda3D abstrai algumas dessas etapas, tal como a verificação de colisões e criação das junções de contato, reduzindo o esforço exigido do programador. Para que a integração ocorra corretamente, o controle do laço principal do jogo deve ficar com o Panda3D, e as etapas do laço acima devem ser chamado de dentro de uma tarefa do Panda3D.

3.2. Corpos rígidos

Um corpo rígido (*rigid body*, referido na documentação do ODE apenas por *body*) é uma aproximação de um corpo físico real, com tamanho finito, sobre o qual se supõe que não ocorre deformação, ou seja, selecionando-se dois pontos quaisquer do corpo, a distância entre eles não muda ao longo da simulação. É uma aproximação suficientemente boa para muitos tipos de corpos; quando usada em objetos deformáveis, ou *soft bodies* (e.g. uma bandeira), o resultado é uma aparência e uma movimentação pouco realista. Esse problema pode ser mitigado em parte através da decomposição em corpos menores, que podem sofrer movimentos e transformações diferentes.

Na modelagem adotada pelo ODE, as propriedades do corpo rígido são somente aquelas que afetam seu comportamento dinâmico; os cálculos são realizados de forma independente para cada corpo, e as rotinas de detecção de colisão são isoladas. Por este motivo, algumas características que dizem respeito apenas ao tratamento de colisões (tal como o formato do corpo) não pertencem à estrutura de dados *body*.

Um corpo rígido possui algumas propriedades constantes ao longo do tempo, que definem as características do objeto:

- Massa do corpo.
- Posição do centro de massa em relação à posição do centro de referência do corpo; na implementação atual, devem coincidir obrigatoriamente.
- Matriz de inércia, que descreve a distribuição da massa ao redor do centro de massa.

É possível definir essas propriedades do corpo através de funções simplificadas que determinam automaticamente a massa e a matriz de inércia a partir de uma densidade fornecia pelo programador e do formato aproximado do corpo; opções disponíveis incluem esferas, caixas, cápsulas e cilindros. É importante notar que os valores de densidade (da ordem de 10^4) sugeridos no manual do Panda3D [27] são excessivamente altos e causam comportamentos instáveis ou pouco realistas no sistema, sendo recomendado o uso de densidades entre as ordens de 10^2 e 10^3.

Outras propriedades se modificam durante a simulação:

- Vetor de posição do centro de referência do corpo (x, y, z).
- Vetor da velocidade linear do ponto de referência do corpo (vx, vy, vz).
- Orientação do corpo, representado por um quatérnio (qs, qx, qy, qz) ou uma matriz de rotação 3x3.
- Vetor de velocidade angular (wx, wy, wz).

A partir dessas propriedades, é possível definir um eixo de coordenadas para o corpo, que se movimenta e gira com ele (Figura 5).

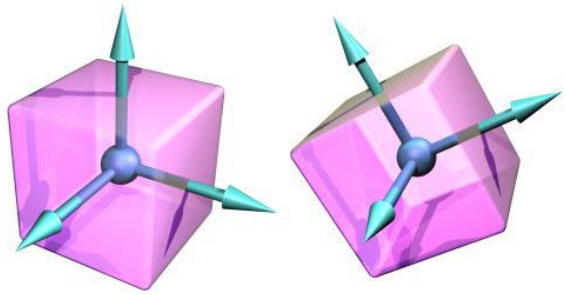

Figura 5: O eixo de coordenadas de um corpo rígido [26].

É possível desativar um corpo, de forma que suas propriedades dinâmicas deixem de ser computadas e atualizadas a cada passo, reduzindo assim o tempo gasto com a simulação física.

3.3. Junções

Corpos são conectados através de junções (*joints*), que determinam as restrições da conexão. A cada etapa da simulação, forças são introduzidas automaticamente para aproximar os corpos de uma situação em que as restrições sejam atendidas, mantendo assim a posição e a orientação relativa entre eles dentro de certos requerimentos.

O ODE trabalha com o conceito de "ilhas de corpos", ou seja, o conjunto mínimo de corpos que está conectado um ao outro por junções, diretamente ou através de outros corpos da ilha. O tempo gasto para processar uma etapa da simulação física é linear com respeito ao número de ilhas presentes no mundo.

É importante notar que um corpo inativo que esteja em uma ilha com ao menos um corpo ativo será ativado automaticamente no próximo passo. Como a detecção de colisões não é anulada quando um corpo está inativo – apenas os cálculos dinâmicos são – se um corpo inativo colidir com um corpo ativo, uma junção será criada automaticamente, e o corpo (e sua ilha) será ativado.

O tipo de junção escolhido indica qual a relação entre os corpos que ela conecta. O ODE possui suporte a diversos tipos de junção, enumerados mais abaixo, cada um apropriado para uma situação diferente.

É possível agrupar junções em grupos de junção (*joint groups*) no momento em que elas são criadas. A remoção de um grupo exclui todas as junções nele contidas, sendo mais eficiente em termos de tamanho de código e de velocidade de execução. Esta técnica é usada com as junções de contato, que são criadas e removidas em lote a cada etapa da simulação.

3.3.1. **Fixo**

A junção fixa (*fixed*) é a mais básica provida pelo ODE. Ela mantém uma posição relativa e uma rotação relativa fixas entre os dois objetos. Em alguns casos, o uso desta junção pode ser substituído por um único corpo representando os dois objetos.

3.3.2. **Bola e encaixe**

Uma junção de bola e encaixe (*ball and socket*) obriga que a bola do segundo corpo esteja sempre na mesma posição que o encaixe do primeiro corpo (v. Figura 6). Caso eles tomem distância entre si, esta será corrigida na próxima iteração. Este tipo de junção não impõe restrições quanto à rotação de seus objetos.

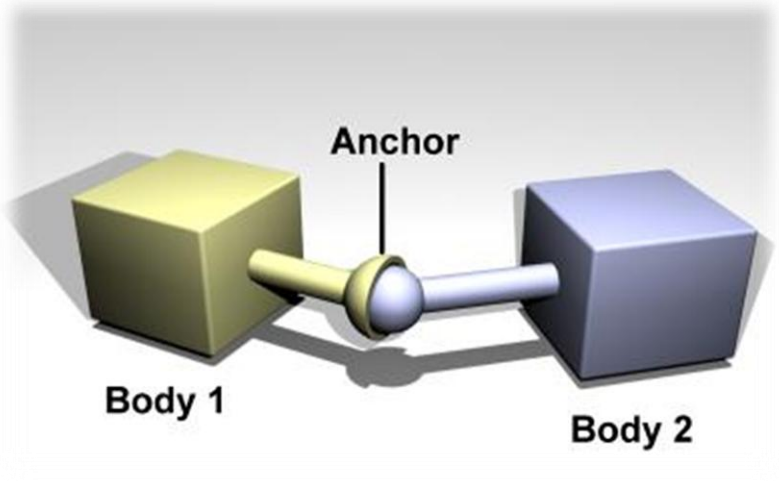

Figura 6: Uma junção do tipo bola e encaixe [26].

A junção de bola e encaixe tem aplicações diversas; o quadril humano é um exemplo deste tipo de junção, e ele também pode ser usado para modelar conexões de cabos e correntes com outros objetos.

3.3.3. **Dobradiça**

Uma junção de dobradiça (*hinge*) acrescenta, em relação à junção de bola e encaixe, uma restrição quanto à orientação de seus objetos: eles são obrigados a estarem alinhados em um dos eixos, definido pelo programador (v. Figura 7).

Tem utilidade para modelar dobradiças do mundo real (por exemplo, entre uma porta e a parede) e outros objetos assemelhados, tais como a conexão entre braço e antebraço em um ser humano.

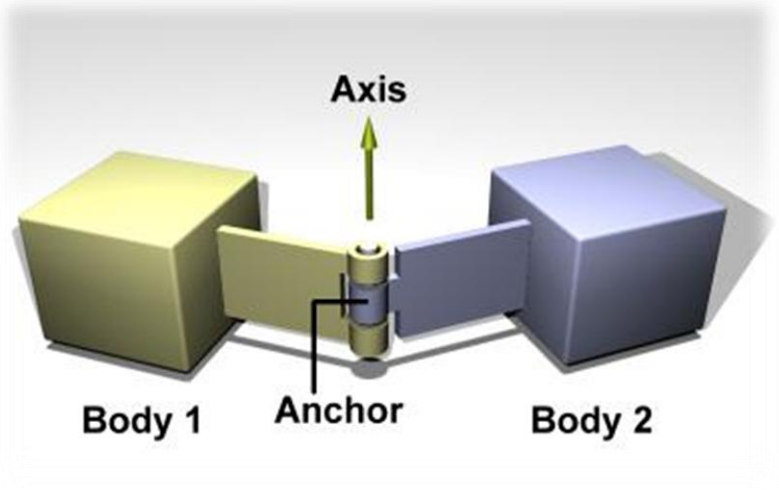

Figura 7: Uma junção do tipo dobradiça [26].

3.3.4. **Contato**

Uma junção de contato (*contact*) representa o contato entre dois corpos físicos, e evita que ocorra uma penetração entre eles, restringindo suas velocidades de forma que ocorram no sentido inverso do centro de referência à posição de contato. Ela pode ainda, dependendo dos parâmetros escolhidos, impor uma fricção aos objetos na direção perpendicular à penetração (v. Figura 8).

No uso regular do ODE, cabe ao programador criar as junções de contato a partir das informações de colisão fornecidas pela biblioteca, antes do passo de simulação, e excluir todas as junções após o passo. O Panda3D, no entanto, realiza a etapa de criação automaticamente para o desenvolvedor, através de um método que chama a rotina de detecção de colisão do ODE e já cria as junções de contato necessárias.

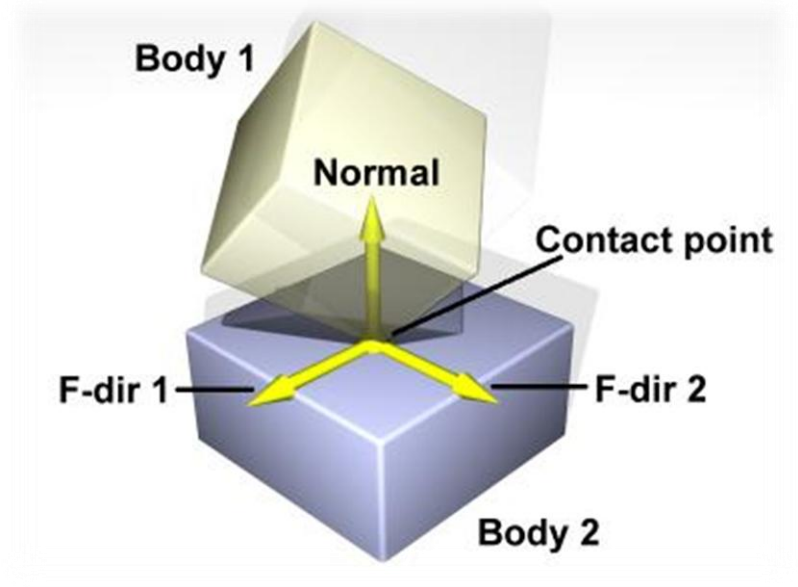

Figura 8: Uma junção de contato [26].

3.3.5. **Outros tipos**

Existem ainda outros tipos de junção, que permitem interações complexas entre objetos. São utilizados para simular motores, transmissões de carros e engrenagens robóticas; sua aplicação é pouco comum em jogos, sendo mais usual em sistemas de simulação industrial, de engenharia ou científica.

- Universal: é similar à bola e encaixe, mas remove um dos graus de liberdade de rotação.
- Dobradiça dupla (*hinge-2*): equivale ao uso de duas dobradiças em série, com eixos de alinhamento diferentes.
- Deslizante (*slider*): força o alinhamento entre dois objetos, impedindo que haja movimento relativo exceto no eixo determinado.
- Pistão (*piston*): similar ao deslizante, mas permite que os objetos girem em torno do eixo de deslizamento.
- Prismática e rotóide (PR): combina uma junção deslizante (*prismatic*) com uma junção de dobradiça (*rotoide*), sem a necessidade de criar um corpo intermediário entre as duas.
- Prismática e universal: combina uma junção deslizante com uma universal, sem a necessidade de criar um corpo intermediário entre as duas.
- Motor linear e motor angular: permite, respectivamente, controlar a velocidade linear ou angular relativa entre dois corpos.

A lista completa dos tipos de junção suportados pelo ODE, e os parâmetros associados a cada tipo, pode ser encontrada no manual da biblioteca [26].

3.4. Forças e integração

O processo de recálculo dos estados (velocidades e posições) dos corpos recebe o nome de integração; isso é feito a cada passo da simulação, avançando o tempo atual da simulação física em um intervalo definido pelo programador e ajustando as propriedades do corpo de acordo.

Um integrador possui duas qualidades principais: precisão (similaridade entre o comportamento do sistema simulado e o comportamento que ocorreria no mundo real) e estabilidade (grau no qual erros de cálculo podem introduzir falhas e comportamentos não físicos no sistema simulado). O integrador do ODE é estável no geral – desde que o intervalo de tempo usado em cada passo seja sempre o mesmo – e preciso, quando o intervalo usado é pequeno.

Cada corpo rígido no ODE possui um acumulador de forças. O programa pode aplicar novas forças aos corpos a cada passo, antes de rodar uma iteração da simulação, e estas serão adicionadas ao acumulador. Quando a iteração da simulação é executada e o integrador é chamado, as forças acumuladas, junto às forças geradas pelas junções para atender suas restrições, são aplicadas para definir o novo estado do corpo.

É importante notar que o ODE disponibiliza uma função para que o programador obtenha o vetor correspondente à força acumulada sobre um corpo, mas tal função não é acessível através da interface do Panda3D. Isso impede que o programador tome ações condicionadas à força resultante – por exemplo, ativar a destruição de um objeto caso a intensidade da força que atua sobre o corpo esteja acima de um limiar.

Forças são sempre definidas através de um vetor de três elementos. É possível definir o torque atuante sobre o corpo separadamente das forças regulares.

3.5. Tratamento de colisões

O tratamento de colisões (*collision handling*) é dividido em duas etapas: detecção de colisão e reação às colisões.

O ODE possui rotinas para detectar se colisões estão ocorrendo entre dois ou mais corpos ou com o ambiente estático; essas funções retornam, para cada colisão, o ponto em que ocorre o contato entre os corpos, o vetor que determina a normal da superfície entre eles no ponto da colisão, e a profundidade de penetração entre os

corpos. A partir dessas informações, cabe ao usuário (ou ao Panda3D) criar as junções de contato.

A etapa de reação ocorre durante o passo da simulação física, no qual o ODE define forças para os corpos que possuem junções de contato associados. Após o passo da simulação, o programador tem a obrigação de remover todas as junções de contato; isso não é feito automaticamente pela biblioteca.

3.5.1. **Detecção de colisões**

O ODE permite que o usuário indique, através de agrupamento em espaços (não disponível através do Panda3D) e de máscaras de bits, quais objetos podem colidir com quais outros. Pares de objetos que não estão no mesmo espaço de colisão ou que não possuem bits em comum nas suas máscaras não são testados, melhorando a performance da etapa de detecção de colisões.

O agrupamento em espaços embute outras otimizações em sua implementação, pois armazena os corpos de forma que objetos distantes sequer chegam a ser testados para colisão, novamente acelerando o processo.

3.5.2. **Reação às colisões**

As junções de contato do ODE representam uma técnica chamada *hard contacts*, que adiciona uma restrição às fórmulas de integração para evitar a penetração entre corpos. Alguns outros simuladores adotam uma abordagem de inserir molas temporárias nos pontos de colisão; essa opção foi descartada no ODE por trazer instabilidade e ter uma implementação complexa.

O usuário do ODE deve definir, para cada junção de contato, uma série de parâmetros que determinam o comportamento da colisão. Estes são:

- mu: coeficiente de fricção de Coloumb. Determina a fricção que será aplicada aos corpos, no sentido perpendicular à penetração entre eles.
- bounce: determina o coeficiente de restituição elástica entre os corpos ou, em outras palavras, a perda de energia causada pela colisão.
- bounce_vel: velocidade mínima necessária para que os corpos saltem ao colidir. Impede que os corpos quiquem indefinidamente com amplitudes mínimas.
- slip: determina o grau em que os corpos em colisão podem deslizar entre si.
- dampen: simula artificialmente um efeito de amortecimento em comportamentos oscilatórios.

- soft_erp e soft_cfm: parâmetros de correção de erro, que reduzem a instabilidade do sistema ao custo de uma pequena redução na precisão (ex.: permite que haja uma pequena penetração entre os corpos após o passo da simulação).

Tais parâmetros devem ser definidos cuidadosamente para que o comportamento de colisões se assemelhe ao desejado. Esta configuração é uma etapa demorada, normalmente feita por tentativa e erro a partir de um conjunto de parâmetros estimado, de forma iterativa até que o comportamento apresentado esteja satisfatório. Valores padrão para os parâmetros estão sugeridos na seção correspondente do manual do ODE [26].

Usuários do ODE através do Panda3D definem tais parâmetros de forma ligeiramente diferente, através da noção de superfícies: cada objeto criado deve possuir um tipo de superfície, e o programador define em uma tabela, na inicialização do programa, os parâmetros a serem aplicados para cada par de superfícies. Assim, no momento da colisão, os parâmetros da junção de contato são preenchidos automaticamente pelo Panda3D.

3.6. Representações geométricas

Para que possam tomar parte nas rotinas de tratamento de colisão do ODE, os corpos rígidos precisam ter geometrias (*geoms*) associadas. Estas são criadas separadamente da criação do corpo e determinam uma forma geométrica rígida ou um conjunto delas.

É possível também criar geometrias não associadas a objetos (ou, visto por outro ângulo, associadas ao ambiente como um todo), representando colisões com o cenário. Tais colisões afetam apenas o objeto que possui um corpo associado.

Os tipos de representação geométrica disponíveis no ODE atendem à maioria das necessidades dos desenvolvedores: caixa (*box*), cilindro (*cylinder*), cápsula (*capped cylinder*), plano (*plane*), ray (*ray*) e esfera (*sphere*). Para criar uma representação geométrica, é necessário fornecer a posição do centro de referência do objeto, e o seu tamanho – os parâmetros exatos variam de acordo com o tipo de geometria (por exemplo, o raio para uma esfera, mas o tamanho dos lados para uma caixa).

Há ainda uma representação arbitrária, definida a partir de uma malha de triângulos fornecida pelo programador (*trimesh*), que pode ser usada quando as outras

aproximações são insuficientes; porém, há restrições para o seu uso, e testes com malhas arbitrárias são significativamente mais lentos.

Os *geoms* são representações geométricas, mas não dinâmicas; por isso, eles possuem uma posição e uma rotação, assim como os corpos, mas ao contrário destes, não possuem velocidade linear ou angular, nem um acumulador de forças.

Capítulo 4:

Implementação

4.1. Visão geral

Este capítulo descreve a implementação criada em Python para aplicar os conceitos do trabalho, à qual foi dado o nome "Heavy Destruction".

Definiu-se primeiramente que o projeto seria uma *demo*, ou seja, ele não apresenta todas as características usuais de um jogo completo, como tela inicial, objetivo, história, progressão, etc. O jogador é lançado diretamente no meio da cena de ação, e pode andar, pular e atirar livremente pelo cenário, devendo fechar o programa quando sentir-se satisfeito com a sessão – não há limite de tempo.

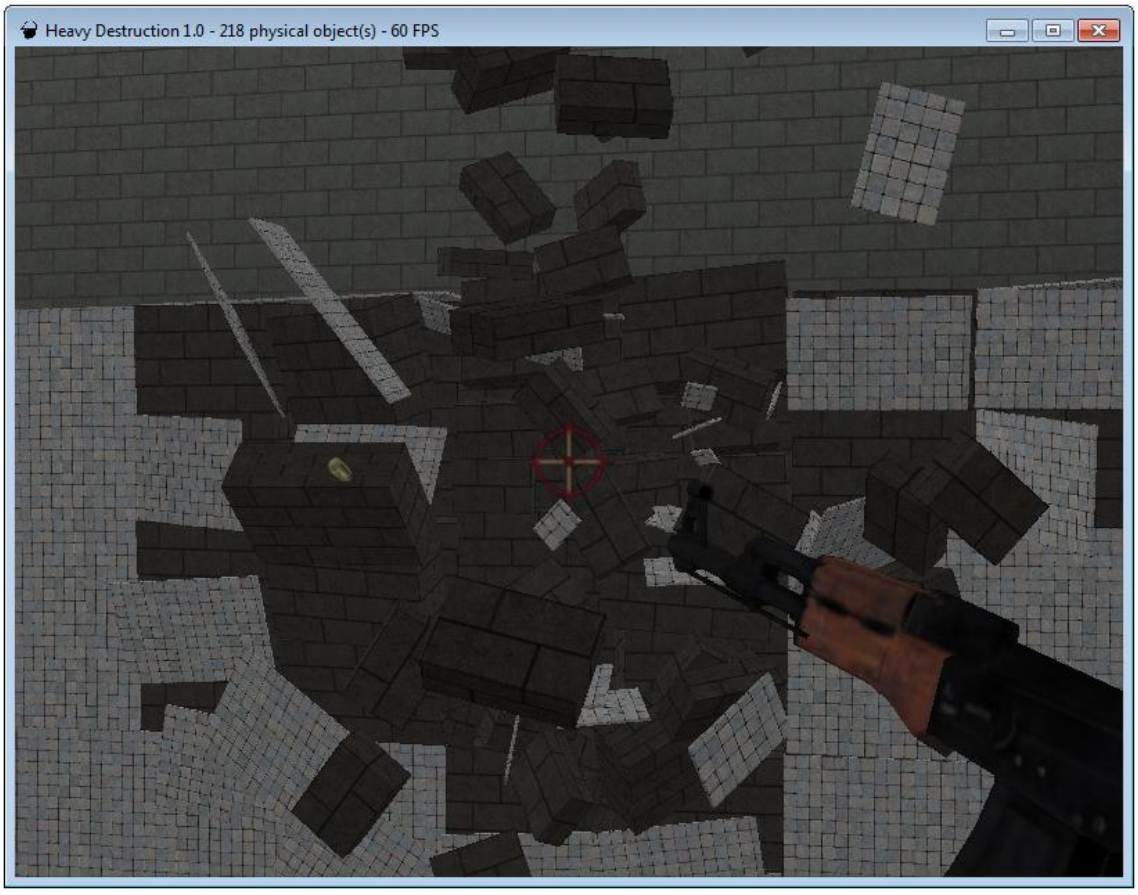

Figura 9: Demo em execução.

A cena padrão do jogo é composta por paredes indestrutíveis que compõem os limites do cenário, e um muro de tijolos destrutível revestido por azulejos. Inicialmente, os azulejos estão grudados aos tijolos, e os tijolos também estão colados entre si. Ao

serem atingidos por disparos com velocidade suficiente, os azulejos podem se descolar da parede ou estilhaçar; os tijolos também podem ser quebrados em pedaços menores.

O jogador pode andar pelo cenário com as teclas W, A, S e D, pular com a barra de espaço, girar a câmera com o *mouse* e atirar com o botão esquerdo. A velocidade do tiro pode ser controlada segurando o botão esquerdo do *mouse*. O modo de tela cheia pode ser ativado ou desativado com a combinação Alt+Enter, ou passando o parâmetro *full* ao chamar o script main.py.

O jogador pode ainda interromper a simulação física (dinâmica dos objetos e tratamento de colisões) e com a tecla ESC. Nesse modo, é possível caminhar livremente pelo cenário. A simulação pode ser retomada pressionando novamente a tecla ESC.

É possível ativar uma cena alternativa (Figura 10), criada a fim de testar a biblioteca no início do desenvolvimento, passando o parâmetro *balls* para o script main.py. Esta cena apresenta algumas bolas que quicam pela tela e paralelepípedos giratórios, fixos em um eixo através da junção de dobradiça.

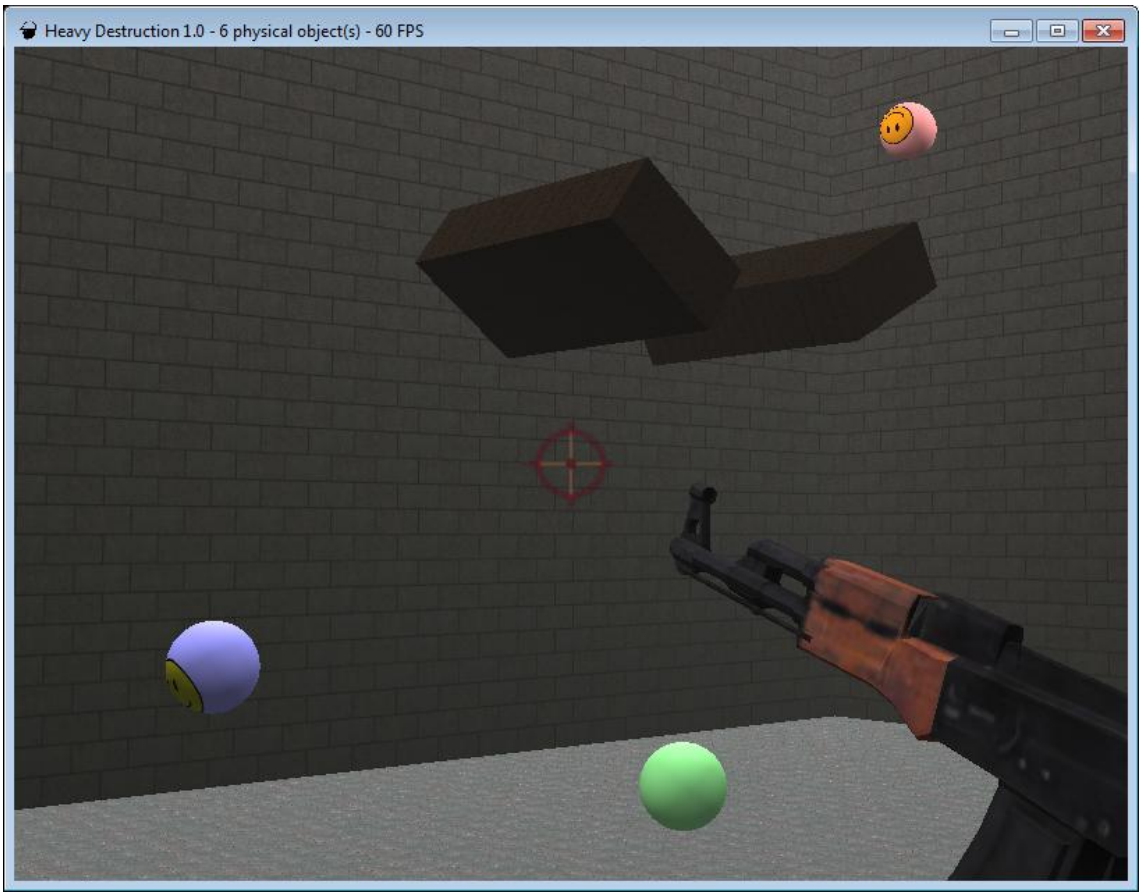

Figura 10: Cena alternativa da demo.

Algumas alterações para depuração podem ser ativadas passando o parâmetro *debug* para o script main.py. Neste modo, objetos físicos cuja simulação foi desativada são coloridos em verde, e azulejos cuja cola foi removida são coloridos em azul.

O código fonte completo do projeto está disponível no repositório público Google Code [28], acrescido de instruções de como executar e a lista de dependências necessárias.

4.2. Ambiente de desenvolvimento

Foi utilizada a IDE (*integrated development environment* – ambiente de desenvolvimento integrado) NetBeans [29] com o plugin para desenvolvimento em Python, ambos softwares livres. Para integração correta com o Panda3D, foi necessário configurar a IDE para usar o interpretador Python alternativo fornecido com a biblioteca.

Não foi possível usar o depurador do NetBeans, pois este parava de funcionar tão logo o laço principal do Panda3D era iniciado. Foi feita uma pergunta sobre o assunto no fórum da comunidade do Panda3D, mas não houve respostas.

O projeto foi desenvolvido e testado em um laptop com hardware equivalente a um computador de mesa de nível médio-avançado atual. Suas configurações incluem processador Intel Core i5 de 2.53GHz, 6GB de memória RAM e placa de vídeo discreta NVIDIA GeForce GT 330MB com 1GB de memória. O sistema operacional utilizado foi o Microsoft Windows 7 64-bits.

Toda a mídia utilizada dentro do jogo – texturas, imagens e sons – foi obtidas do site OpenGameArt.org [30], no qual são disponibilizadas com uma licença compatível com software livre. O ícone do projeto foi obtido no site Open Icon Library [31], também com licença de software livre. O endereço de cada mídia individual encontra-se no arquivo "Media/source.txt" do repositório.

4.3. Estrutura de código

O projeto foi criado usando o paradigma de orientação a objetos. Utilizando a notação UML (*Unified Modeling Language*), a relação entre as classes é definida conforme a Figura 11.

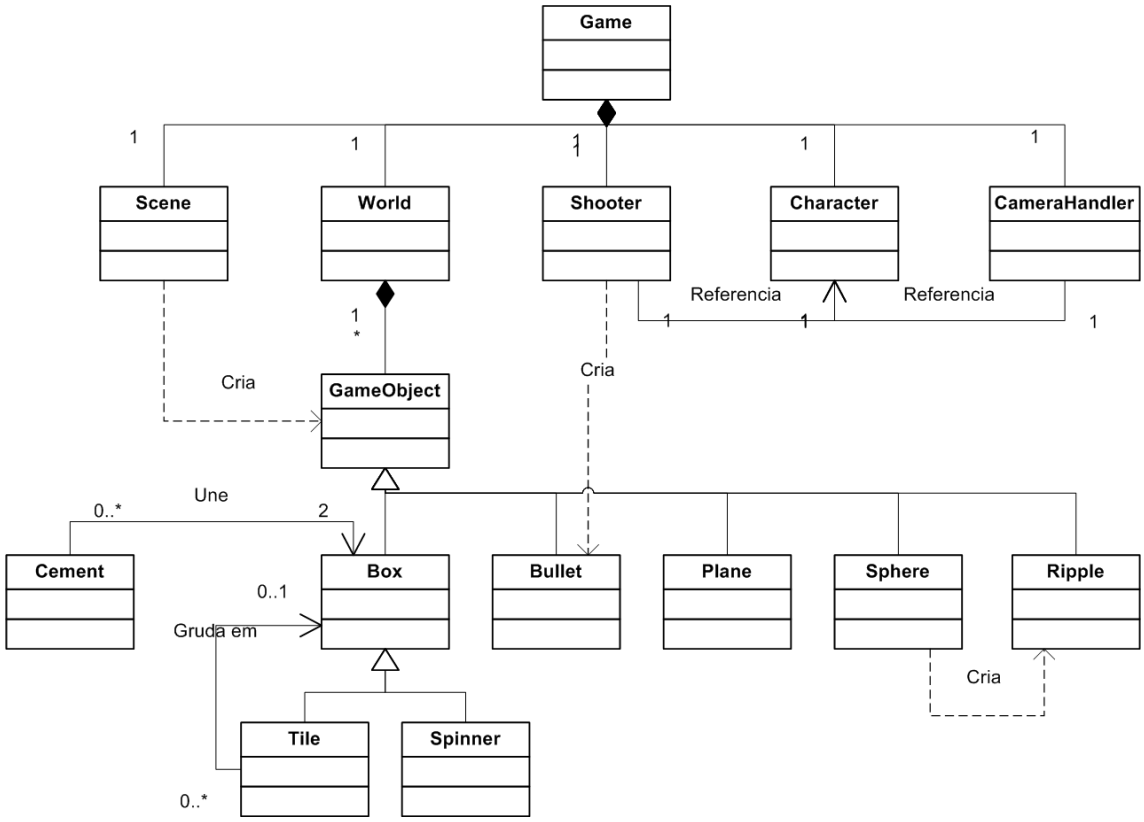

Figura 11: Diagrama de classes do projeto.

Existem ainda dois arquivos, com funções de apoio, que não definem classes próprias: vecUtils e pandaUtils. O primeiro possui funções de manipulação de vetores do Panda3D e listas de Python, incluindo funções para conversão entre os dois tipos e uma função que implementa o produto de Hadamard (multiplicação item-a-item entre dois vetores, produzindo um terceiro vetor).

O arquivo pandaUtils expõe funções que afetam o ambiente do Panda3D, tais como ativação/desativação do modo tela cheia, mudança do título da janela e ocultação do ponteiro do *mouse*. É neste arquivo que está definida também a classe SoundWrapper (omitida do diagrama), que permite que um arquivo de som seja pausado e retomado sem que a posição atual dele seja perdida – funcionalidade ausente na classe provida pelo Panda3D.

As classes do jogo que compõem o pacote-raiz são apresentadas a seguir nesta seção, após a explicação do fluxo de execução normal do programa. As classes que representam os objetos que afetam a mecânica do jogo pertencem ao pacote *objects* e encontram-se na próxima seção.

4.3.1. **Fluxo de execução**

O fluxo de execução do programa tem início no arquivo main.py, que verifica os argumentos de linha de comando e cria uma instância única (*singleton*) da classe Game. A instância de Game, então, assume o controle. Ela inicializa o motor gráfico do Panda3D, cria uma instância de cada uma entre as classes World, Character, CameraHandler e Shooter, cria uma instância de cena apropriada de acordo com a seleção do usuário e passa o controle ao laço principal do Panda3D.

As classes mencionadas acima criam diversas tarefas no Panda3D, as quais são executadas a cada passo do jogo, que normalmente executa a uma velocidade entre 30 e 60 QPS. A principal tarefa é o método World.processPhysics, que chama o método de detecção de colisões e a simulação física do ODE.

É importante observar que o ODE exige passos de mesmo tamanho para fornecer uma simulação estável e, por isso, em vez de usar a duração entre chamadas à tarefa processPhysics como passo da simulação física, utiliza-se um passo fixo (1/90 de um segundo) e é feito uso de um acumulador, que determina quantos passos fixos de simulação física serão executados no passo do Panda3D em questão.

4.3.2. **Classe Game**

Esta classe representa o jogo como um todo; deve existir apenas uma instância durante a execução do programa. Além de atuar como um container para as outras classes, também é responsável por gerenciar a janela do programa – ícone, texto da barra de título, tamanho – e por calcular a taxa de quadros do jogo a cada instante.

4.3.3. **Classe World**

Esta classe atua como um container de todos os objetos do jogo (instâncias de classes derivadas de GameObject), gerenciando seu ciclo de vida, e é responsável por gerenciar também a simulação física. Os parâmetros do mundo físico estão definidos no construtor, exceto pelos parâmetros de superfícies, que estão definidos no método setSurfaceTables.

Objetos criados ou removidos devem chamar, respectivamente, os métodos addObject e removeObject para que eles possam entrar ou sair da simulação física. Esta ocorre no método processPhysics, que realiza os passos descritos na seção 3.1. Colisões detectadas pelo ODE dentro desse método ativam o *callback* onCollision, que por sua vez repassa o controle para o método de mesmo nome do(s) objeto(s) que colidiram, permitindo assim que estes reajam da forma adequada à sua classe.

Em adição ao sistema de tarefas do Panda3D, foi criado um sistema de tarefas adicional, que chama as funções registradas após o passo da simulação física, em vez de após o passo do motor principal. A implementação deste é feita no método processPhysics, através da chamada ao método performPostStepTasks. O registro de uma função é feito através do método performAfterStep.

4.3.4. **Classe GameObject**

Esta classe é a base para os objetos que possuem uma representação visual e uma representação física, e não deve ser instanciada diretamente. Possui alguns métodos de apoio, tais como getMomentum, que retorna o momento linear do corpo, e dissipate, que simula a perda de energia do corpo, reduzindo suas velocidades linear e angular.

A classe possui também um construtor e um destrutor (método destroy) que realizam tarefas básicas de manutenção e devem ser chamadas sempre pelas classes derivadas.

4.3.5. **Classe Scene**

Esta classe abstrata povoa o mundo do jogo, inicialmente vazio (apenas com o jogador) com todos os objetos necessários para a cena, incluindo os limites do cenário. É aqui também que são definidas a música ambiente e a iluminação, pois estas podem variar de acordo com o cenário selecionado.

As duas cenas existentes (FallingBalls e BasicWall) são implementadas como subclasses desta classe. Para ativar uma cena, basta instanciar a classe correspondente.

4.3.6. **Classe Character**

Esta classe controla a movimentação do jogador, incluindo aspectos como sua posição e velocidade. Ela é uma classe derivada de GameObject, pois o jogador possui uma representação física. Isso permite que ele interaja com os outros objetos da cena, impedindo, por exemplo, que ele atravesse paredes. Também possibilita que seus saltos ocorram naturalmente, obedecendo a gravidade definida para a simulação, e que o personagem suba em outros objetos.

4.3.7. **Classe CameraHandler**

Esta classe controla a câmera, reagindo à movimentação do *mouse* na forma de alterações na rotação da cena visível. A posição da câmera está atrelada à posição do jogador.

Esta classe disponibiliza métodos para que outras classes possam ativar efeitos especiais, tais como *shake* (tremor rápido da tela) e *flash* (lampejo momentâneo em uma cor selecionável).

4.3.8. **Classe Shooter**

Esta classe é responsável por todas as tarefas relacionadas ao disparo da arma do jogador, incluindo o tratamento do botão esquerdo do *mouse*, a exibição do indicador de velocidade do disparo enquanto o botão é pressionado, a execução do efeito sonoro, e a exibição do modelo tridimensional da arma na tela, com prioridade sobre todos os outros modelos.

Esta classe é uma máquina de estados finitos e, portanto, herda da classe FSM (*finite state machine*) do Panda3D. Seus estados são *Shooting* (atirando) e *Waiting* (aguardando). A transição dos estados ocorre no momento em que o jogador pressiona ou libera o botão esquerdo do *mouse*.

A criação de uma bala é feita no método exitShooting. A bala é posicionada inicialmente no mesmo local que o personagem do jogador, com a mesma orientação que a câmera. A bala recebe uma velocidade linear que varia entre 20 m/s e 60 m/s, de acordo com o tempo que o botão permaneceu pressionado. Ela recebe ainda uma velocidade angular de 80º/s a 400º/s em torno do próprio eixo, simulando o efeito de rotação que ocorre no mundo real.

4.4. Objetos físicos

Os arquivos de código dos objetos que compõem a mecânica do jogo podem ser encontrados dentro do pacote *objects*.

4.4.1. **Classe Box**

Esta classe representa objetos no formato de um paralelepípedo. Seu tamanho, orientação, cor e textura podem ser definidos no momento de sua instanciação. Ela também serve como classe-base para as classes Tile e Spinner, que são objetos com formato similar, mas que apresentam alguns comportamentos específicos. Os tijolos que compõem a parede destrutível do jogo são instâncias diretas desta classe.

Tijolos são quebrados quando algum outro objeto colide com eles com um módulo de velocidade linear igual ou maior que o limiar definido na instanciação do objeto do tijolo (método onCollision). Quando isso ocorre, o tijolo original é removido da cena e são introduzidos em seu lugar 8 tijolos, cada um com 1/8 de seu tamanho,

preenchendo o mesmo volume (método shatter). Em seguida, esses novos objetos são impelidos para fora, simulando uma explosão, com velocidade angular e linear proporcional à velocidade do objeto que colidiu com o tijolo (método spawnTask). Cada novo tijolo tem sua velocidade na direção dada pelo vetor que se origina no centro do tijolo original e termina no centro do novo tijolo (Figura 12). A velocidade dos novos tijolos sofre, por fim, um pequeno acréscimo na direção da velocidade linear do tijolo original e na velocidade do objeto com o qual este colidiu, garantindo certa conservação dos momentos lineares.

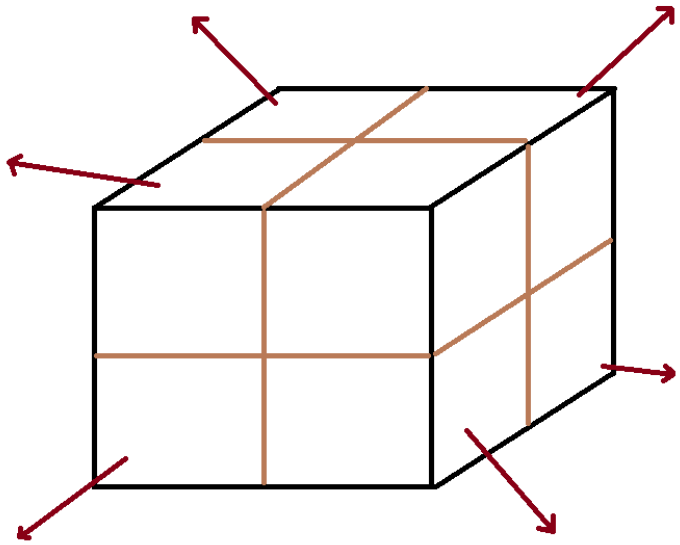

Figura 12: Quebra e explosão de um tijolo.

Um tijolo oriundo de uma quebra pode ser novamente destruído e transformado em pedaços menores. Esse limite é definido em um dos parâmetros do construtor da classe Box. Cabe ao programador definir a quantidade de quebras possíveis ao criar os blocos iniciais; a cada quebra subsequente, os novos tijolos são criados com um limite equivalente ao seu tijolo de origem menos um. Tijolos com limite zero não podem ser subdivididos.

Inicialmente, os tijolos estão colados entre si com o uso da classe Cement. Isto reduz a instabilidade da parede que, caso contrário, se desmancha frente ao impacto de poucos disparos. A camada inferior de tijolos é também colada ao chão (ao ambiente). Quando um bloco é quebrado, os cimentos a ele atrelados são destruídos automaticamente.

4.4.2. **Classe Tile**

Esta classe, que herda da classe Box, representa um azulejo que pode estar ou não grudado a um tijolo. Esta opção, implícita no construtor da classe, cria uma junção do tipo fixa para unir o azulejo ao tijolo. A criação de um azulejo sem tijolos é permitida apenas para uso interno da classe, através do método make2, e só é usada para criar azulejos menores oriundos da quebra de um azulejo regular.

Um azulejo também pode ser quebrado, da mesma forma que um tijolo. Porém, os azulejos menores criados para substituí-lo não recebem velocidade inicial, limitando-se a cair no chão através da força da gravidade. Por terem uma de suas dimensões (profundidade) desprezível, o azulejo é dividido apenas nas outras dimensões, gerando portanto quatro novos objetos ao ser quebrado.

Além da quebra, a colisão com objetos pode causar outro efeito sobre o azulejo: o descolamento. Este possui um limiar diferente do limiar de quebra, normalmente inferior, possibilitando que uma bala disparada com pouca velocidade retire a cola do azulejo, fazendo-o cair, mas não o estilhace.

4.4.3. **Classe Spinner**

Esta classe, que também herda da classe Box, representa um tijolo que está fixo ao cenário em um dos eixos, através de uma junção de tipo dobradiça. Foi usada nas etapas iniciais de desenvolvimento para realizar experimentos com junções. Sua inspiração advém dos pisos falsos giratórios comuns em jogos de plataforma.

4.4.4. **Classe Cement**

Esta classe modela um elemento que une dois tijolos, ou um tijolo e o chão, e por isso recebeu o nome de cimento. Corresponde apenas a uma junção do tipo fixa e o código apropriado para criação e destruição.

4.4.5. **Classe Bullet**

Esta classe representa a bala disparada da arma do jogador. Sua representação gráfica é um modelo tridimensional, e a representação física, um cilindro.

Inicialmente as balas têm a gravidade desativada ao serem criadas, pois a simulação padrão, com gravidade e sem resistência do ar, faz com que as balas caiam em demasia ao serem disparadas, resultando num comportamento não natural. Após a primeira colisão com qualquer objeto a gravidade sobre a bala é ativada.

Para evitar que ocorram ricochetes ilimitados e pouco naturais, uma dissipação de energia artificial é introduzida sobre a bala a cada colisão, proporcional ao grau de

penetração ocorrido na colisão (método dissipate da classe GameObject e método onCollision). Isto faz também com que a bala perca gradualmente sua velocidade de rotação ao girar sobre o chão; é um fator necessário, tendo em vista que a simulação do ODE não possui o chamado *rolling friction* (fricção de rolamento).

Como a bala é criada na mesma posição que o jogador, ela inicialmente aparece com um tamanho desproporcionalmente grande em relação ao resto da cena. Para evitar esse efeito, a bala é definida como invisível ao ser criada, tornando-se visível apenas após 10ms.

4.4.6. **Classe Plane**

Esta classe representa um plano que limita o cenário. O corpo físico associado é um semi-espaço tridimensional, definido pelo plano; um corpo que esteja no lado "atrás" do plano é considerado como estando em colisão com este. Foi criada uma função que retorna os coeficientes da equação do plano (requeridos pelo método do ODE que cria a geometria) a partir de um ponto e um vetor normal, que são os parâmetros recebidos no construtor.

A representação gráfica é um retângulo sem espessura. Apesar da representação física se estender indefinidamente ao longo do plano, a representação visual tem altura e largura finitas. É possível definir uma textura para o plano através do construtor.

4.4.7. **Classe Sphere**

Esta classe representa uma esfera simples, com um corpo físico associado. Seu raio pode ser selecionado em seu construtor. É usada exclusivamente na cena FallingBalls.

4.4.8. **Classe Ripple**

Esta classe representa a mancha que as esferas deixam ao colidir com alguma parede. Foi criada nas etapas iniciais de desenvolvimento para testes com detecção de colisões e as informações da colisão que podem ser obtidas, como posição e direção da superfície de contato. Não possui representação física e, portanto, não herda da classe GameObject. Instâncias desta classe são removidas automaticamente após um período de tempo.

4.5. Otimizações

Diversas otimizações foram feitas ao longo do projeto para que fosse possível atingir simultaneamente o objetivo de qualidade (tijolos e azulejos quebrando-se de

forma realista) e de velocidade (manutenção da taxa mínima de 15QPS após a destruição de um tijolo).

4.5.1. **Remoção de objetos inativos**

A quantidade de objetos na simulação física é a principal causa de lentidão. Na versão final do projeto, são criados inicialmente pela classe Scene 24 tijolos (uma parede 4x2x3) e 44 azulejos, além do personagem do jogador, totalizando 69 objetos. Na máquina de testes, essa condição, chamada de repouso, manteve a velocidade máxima possível de 60 QPS.

Outros objetos, porém, são introduzidos no mundo físico durante o jogo. A primeira situação advém do disparo de balas; cada disparo acrescenta um objeto físico. Este aumento ocorre lentamente, mas pode ser infinito, bastando que o jogador não pare de atirar. Para resolver este problema, as balas são removidas do jogo após um certo número de colisões com o chão. Para minimizar o impacto visual, um efeito de transparência gradativa é aplicado sobre a bala para indicar que ela está prestes a desaparecer.

A segunda situação, mais problemática, ocorre na quebra de um tijolo ou azulejo. Quatro ou oito novos objetos são introduzidos na cena e apenas um é retirado; quando ocorrem reações em cadeia, a quantidade de objetos em cena sobe dramaticamente de forma quase instantânea. Além disso, um tijolo pode ser quebrado múltiplas vezes, e o número de objetos na cena pode crescer de forma exponencial. Este problema foi resolvido parcialmente inativando tijolos e azulejos quando eles estão em repouso – ou seja, com uma velocidade suficientemente próxima a zero – e sem colidir com outros objetos, exceção feita apenas ao chão. Ambas as condições devem ser atendidas por uma quantidade mínima de passos consecutivos para que a inativação ocorra, evitando situações estranhas nas quais objetos são inativados em pleno ar ou quando estão encostados em outros objetos. Com isto, a simulação ainda sofre uma redução na velocidade após a quebra de muitos blocos, mas esta é temporária e a velocidade volta, após alguns segundos, a níveis próximos dos anteriores à quebra.

4.5.2. **Detecção de colisão condicional**

A segunda causa de lentidão advém do número de objetos na cena que podem colidir entre si. Quando deixado da forma padrão, o tempo de processamento sobre de forma quadrática com o número de objetos na cena.

Durante o desenvolvimento, logo após a introdução de azulejos no jogo, instantaneamente a velocidade na situação de repouso ficou abaixo dos 30 QPS. A solução encontrada foi separar os objetos em categorias, usando a funcionalidade de *bitmasks* provida pelo ODE, de forma que a colisão entre algumas categorias fosse desabilitadas. A lista de categorias está definida na classe GameObject, e a seleção da categoria para cada tipo de objeto ocorre no construtor de cada subclasse.

Ao serem criados, azulejos são colocados na categoria *TileGlued*, e a colisão entre objetos dessa categoria com tijolos ou quaisquer outros azulejos é removida. Os testes de colisão são reativados para um azulejo após seu descolamento do tijolo correspondente.

Além disso, foi removida a colisão entre as balas e o personagem do jogador, de forma que este não interfira na trajetória das balas nos instantes iniciais após o disparo.

Com estas modificações, a velocidade da simulação na situação em repouso, com os azulejos inclusos na cena, passou ao máximo de 60 QPS.

4.5.3. **Limitação no número de passos da simulação**

Como os passos da simulação física ocorrem na mesma *thread* que o passo do laço principal do programa, quando a simulação física demora muito tempo para ser processada, o passo principal também fica mais lento. Isso implica que na próxima iteração do passo principal, o simulador tentará realizar mais passos físicos para compensar esse atraso, ocasionando uma demora ainda maior do passo principal em um ciclo vicioso. Por esse motivo, foi introduzida uma restrição no método World.processPhysics de forma que não mais do que nove passos físicos ocorram a cada iteração do laço principal do programa.

4.6. Além da física

Apesar da ausência de elementos comuns de jogos, tais como tela de título, inimigos e objetivo, alguns aspectos não físicos receberam atenção para que a simulação pudesse ser envolvente e satisfatória. Uma música ambiente é executada durante todo o jogo com esse fim; efeitos sonoros também podem ser ouvidos durante o caminhar e o salto do personagem.

A interface inclui um alvo na tela que indica a direção que a bala tomará ao ser disparada, facilitando assim a interação do usuário. Foi incluído também um modelo de uma arma; apesar de não influenciar de qualquer forma a simulação, por não se tratar de

um objeto físico, é algo que o jogador espera encontrar em um jogo de tiro durante o modo de jogo principal.

Foram implementados alguns efeitos especiais para deixar a experiência mais realista. Ao atirar, tanto o jogador quanto a arma são impelidos para trás, simulando a reação à impulsão da bala, conhecida como "coice". O jogador recebe uma velocidade no sentido oposto ao tiro (somente no eixo horizontal, evitando que tiros para o chão resultem em um salto involuntário), reagindo de forma física e parando apenas devido ao atrito com o chão, enquanto que a arma tem um comportamento pré-definido e não-físico de voltar após alguns instantes à posição de repouso em relação ao jogador. Além do recuo, o disparo também ativa um efeito sonoro e um lampejo temporário, simulando a explosão que ocorre ao atirar.

A quebra de um tijolo também ativa um efeito sonoro associado, além de causar uma vibração temporária na tela. Esta técnica é comum em jogos de tiro, e auxilia na transmissão da sensação de potência do impacto da quebra do bloco.

Capítulo 5:

Conclusões

A implementação de uma simulação física é um procedimento complexo, e a troca (*trade-off*) entre qualidade e velocidade é inevitável. A qualidade pode ser ainda dividida em dois aspectos: quantidade de objetos que a simulação é capaz de suportar e realismo do comportamento apresentado.

Durante a elaboração do trabalho, diversas ideias de otimizações foram pensadas, e a maioria delas foi implementada; ao serem testadas, nem todas apresentaram melhorias significativas. Algumas foram totalmente ineficientes e, por somente acrescentar complexidade ao código, foram removidas.

Após a criação do arcabouço inicial, o procedimento adotado foi o de desenvolvimento de novas ideias de otimização, implantação, testes e aprovação ou reprovação, de forma iterativa. Este procedimento mostrou-se eficiente e apropriado para o tipo de trabalho, e é recomendado para futuras evoluções.

Uma avaliação subjetiva com diversas pessoas sem relação com o desenvolvimento do trabalho mostrou-se positiva no que diz respeito ao impacto que simulações físicas realistas podem ter em jogos eletrônicos. Ao serem apresentadas à demo do projeto, todos mostraram-se entusiasmados, em diferentes graus, com a possibilidade de destruir o cenário, mas tiveram o entusiasmo reduzido quando confrontados com a queda da velocidade causada pela existência de muitos objetos na tela. Ainda que sem validez científica devido ao não uso de procedimentos padronizados, este pequeno experimento apenas confirma as expectativas que nortearam e motivaram a elaboração do trabalho.

Capítulo 6:

Trabalhos Futuros

O fator mais importante para poder atingir uma simulação mais rápida e realística encontra-se no uso de técnicas mais eficientes de detecção de colisão e simulação física condicional. Este é um ponto que inevitavelmente deve sofrer melhorias em versões futuras do projeto.

Porém, durante a elaboração do trabalho, foi possível perceber a limitação imposta pela combinação de motores escolhidos com respeito à performance. A presença de mais do que cem objetos simultâneos na cena causava reduções drásticas na taxa de quadros por segundo, fazendo com que a simulação perdesse a característica de interatividade em tempo real após a introdução de duzentos a trezentos objetos simultâneos. Por este motivo, sugere-se que outros motores físicos, mais modernos, sejam avaliados.

Outra ramificação possível corresponde à implementação de outras formas de interação física realista entre jogador e objetos. O trabalho explorou em profundidade apenas a destruição de cenários, mas esta é apenas uma das formas de simulação do mundo real que podem prover maior imersão ao usuário.

6.1. Ampliação da demo

Através da descoberta e implementação de novas otimizações, a quantidade de objetos físicos em cena pode vir a ser aumentada mantendo a interatividade da simulação. Com isso, será possível revestir todo o limite do cenário (paredes, chão e teto) com uma camada destrutível, aumentando o potencial de imersão do jogador.

Um exemplo de otimização sugerido para atingir esse fim é a ativação/inativação dinâmica de regiões da camada destrutível com base na posição do jogador, permitindo que o cenário seja arbitrariamente grande sem impacto no custo da simulação.

É sugerido também o agrupamento dos objetos através de estruturas dados espaciais, tais como grades regulares e *octrees*. A detecção de colisões do motor físico pode então ser ativada somente para objetos dentro de cada célula, sem prejuízo para a qualidade da simulação, pois é garantido que objetos em células diferentes não estão em

colisão. Um problema a ser tratado nessa abordagem é o de objetos em alta velocidade, que podem pular células de um quadro para o outro.

6.2. Bullet Physics

Bullet Physics [32] é um motor de física e detecção de colisão de código aberto, assim como o ODE. Apesar de ser mais recente e menos madura, esta biblioteca apresenta algumas vantagens sobre o ODE. Uma delas é a detecção de colisão contínua (necessária para evitar que corpos em alta velocidade atravessem objetos) [33], que possui aplicação direta no escopo deste trabalho. Ao contrário do ODE, possui suporte à dinâmica de corpos deformáveis (*soft bodies*) que, apesar de fora do escopo corrente do trabalho, possui aplicações interessantes para o realismo de simulações físicas.

Dentre os jogos comerciais que utilizam esta biblioteca, alguns são de amplo destaque, tais como Grand Theft Auto IV e Red Dead Redemption, produzido pela Rockstar Games [34]. Foi usada ainda na elaboração de efeitos especiais em filmes; Sherlock Holmes, lançado em 2009 pela Warner Bros. Pictures, é um exemplo [35].

6.3. NVIDIA PhysX

O PhysX [36], desenvolvido originalmente pela empresa americana Ageia e posteriormente adquirido pelo fabricante de placas gráficas NVIDIA, é outro motor de simulações físicas em tempo real. Ao contrário das outras bibliotecas discutidas no trabalho, é proprietária, apesar de ser gratuita para desenvolvedores.

O grande fator de destaque desta biblioteca é sua integração com placas de vídeo recentes da NVIDIA, possibilitando que cálculos físicos sejam feitos diretamente na GPU. A aptidão do processador gráfico para realizar cálculos de ponto flutuante, associada à natureza paralelizável do problema de simulação física, aumenta significativamente a performance da simulação.

Este mesmo ponto é uma grande desvantagem do PhysX: atualmente, ele é incompatível com placas de vídeo da AMD, que possui aproximadamente metade do mercado. O desenvolvedor que deseja que seu jogo funcione em qualquer computador torna-se obrigado a manter uma versão extra de seu código em paralelo, utilizando outra biblioteca (ODE, por exemplo) que será ativada quando o computador do usuário não tiver suporte ao PhysX.

6.4. Simulações alternativas

O impacto visual e o potencial de imersão trazidos pelo uso de simulações físicas realistas nos jogos não estão, obviamente, restritos à destruição de cenários. O vídeo demonstrativo de Batman: Arkhan Asylum [6] exibe diversos outros efeitos de interesse do jogador: folhas de papel com interação dinâmica (corpos flexíveis); simulação realística de roupas, bandeiras e outros tecidos; vapor, fumaça e outras simulações de fluidos; centelhas e outras simulações de partículas.

Atualmente, grande parte das técnicas, algoritmos e detalhes de implementação que compõem o estado da arte em sistemas físicos interativos, usadas de fato em jogos comerciais, é proprietária e restrita às empresas nas quais foram desenvolvidos, e seria benéfica a publicação de mais estudos acessíveis à comunidade científica.

Referências Bibliográficas

1. ANDROVICH, M. **GamesIndustry.biz**, 30 jun. 2008. Disponivel em: <http://www.gamesindustry.biz/articles/industry-revenue-57-billion-in-2009-says-dfc>. Acesso em: 20 fev. 2011.
2. IFPI. Recording Industry in Numbers 2010. **International Federation of the Phonographic Industry**, 2010. Disponivel em: <http://www.ifpi.org/content/section_news/20100428.html>. Acesso em: 20 fev. 2011.
3. DOBUZINSKIS, A. Global movie box office nears $30 billion in 2009. **Reuters**, 2010. Disponivel em: <http://www.reuters.com/article/2010/03/10/us-boxoffice-idUSTRE62955520100310>. Acesso em: 20 fev. 2011.
4. IFPI. **Digital Music Report 2009**. International Federation of the Phonographic Industry. Londres, p. 4. 2009.
5. JUNIOR, P. H. N. D. Jogos e Imersão. **Baixo Frente Soco**, 2010. Disponivel em: <http://www.baixofrentesoco.com/eu-gosto-de-jogos-8-%E2%80%93-jogos-e-imersao/>. Acesso em: 20 fev. 2011.
6. GPU PhysX in Batman Arkham Asylum Demo, 2009. Disponivel em: <http://www.youtube.com/watch?v=vINH6Z9kqgI>. Acesso em: 20 fev. 2011.
7. SNEZKHO, I. Crysis Physics Maps. **Taringa!**, 2009. Disponivel em: <http://www.taringa.net/posts/downloads/2284850/Crysis-Physics-Maps.html>. Acesso em: 23 fev. 2011.
8. VALVE CORPORATION. Portal. **The Orange Box**, 2007. Disponivel em: <http://orange.half-life2.com/portal.html>. Acesso em: 20 fev. 2011.
9. EA GAMES. Crysis. **Online PC Computer Games**, 2007. Disponivel em: <http://www.ea.com/games/crysis>. Acesso em: 20 fev. 2011.
10. WARD, J. What is a Game Engine? **Game Career Guide**, 2008. Disponivel em: <http://www.gamecareerguide.com/features/529/what_is_a_game_.php>. Acesso em: 20 fev. 2011.
11. O que é Software Livre? **Software Livre Brasil**. Disponivel em: <http://softwarelivre.org/portal/o-que-e>. Acesso em: 20 fev. 2011.
12. **Panda3D**. Disponivel em: <http://www.panda3d.org/>. Acesso em: 20 fev. 2011.
13. DISCIPLINA COS600 - Animação e Jogos. **Laboratório de Computação Gráfica da UFRJ**. Disponivel em: <http://www.lcg.ufrj.br/Cursos/jogos/cos600-jogos>. Acesso em: 20 fev. 2011.
14. BSD License Definition. **The Linux Information Project**, 2005. Disponivel em: <http://www.linfo.org/bsdlicense.html>. Acesso em: 20 fev. 2011.
15. **Python Programming Language**. Disponivel em: <http://www.python.org/>. Acesso em: 20 fev. 2011.
16. GALLERY of Screenshots. **Panda3D**. Disponivel em:

<http://www.panda3d.org/screens.php>. Acesso em: 23 fev. 2011.

17. THE Scene Graph. **Panda3D Manual**. Disponivel em: <http://www.panda3d.org/manual/index.php/The_Scene_Graph>. Acesso em: 20 fev. 2011.

18. INGLE, D. Panda3D diagrams. **Panda3DProjects**, 2009. Disponivel em: <http://www.p3dp.com/doku.php?id=dotherwise:start>. Acesso em: 23 fev. 2011.

19. **Open Dynamics Engine**. Disponivel em: <http://ode.org/>. Acesso em: 20 fev. 2011.

20. THE GNU Lesser General Public License, version 2.1. **Open Source Initiative**, 1999. Disponivel em: <http://www.opensource.org/licenses/lgpl-2.1.php>. Acesso em: 20 fev. 2011.

21. SCREENSHOT of an official demo of ODE. **Wikimedia Commons**, 2007. Disponivel em: <http://en.wikipedia.org/wiki/File:ODE_buggy.png>. Acesso em: 23 fev. 2011.

22. SMITH, R. What is a physics SDK? **Open Dynamics Engine**, 2001. Disponivel em: <http://www.ode.org/slides/slides.html>. Acesso em: 23 fev. 2011.

23. **PyODE**. Disponivel em: <http://pyode.sourceforge.net/>. Acesso em: 20 fev. 2011.

24. USING ODE with Panda3D. **Panda3D Manual**. Disponivel em: <http://www.panda3d.org/manual/index.php/Using_ODE_with_Panda3D>. Acesso em: 20 fev. 2011.

25. PRODUCTS that use ODE. **Open Dynamics Engine Wiki**. Disponivel em: <http://opende.sourceforge.net/wiki/index.php/Products_that_use_ODE>. Acesso em: 06 fev. 2011.

26. SMITH, R. v0.5 User Guide. **Open Dynamics Engine**, 2006. Disponivel em: <http://www.ode.org/ode-latest-userguide.html>. Acesso em: 20 fev. 2011.

27. WORLDS, Bodies and Masses (ODE). **Panda3D Manual**. Disponivel em: <http://www.panda3d.org/manual/index.php/Worlds%2C_Bodies_and_Masses>. Acesso em: 23 fev. 2011.

28. FRANCESE, J. P. S. heavy-destruction project hosting. **Google Code**, 2011. Disponivel em: <http://code.google.com/p/heavy-destruction/>. Acesso em: 24 fev. 2011.

29. WELCOME to NetBeans. **NetBeans**, 2011. Disponivel em: <http://netbeans.org/>. Acesso em: 23 fev. 2011.

30. FREE, legal art for open source game projects. **OpenGameArt.org**. Disponivel em: <http://opengameart.org/>. Acesso em: 23 fev. 2011.

31. **Open Icon Library**. Disponivel em: <http://openiconlibrary.sourceforge.net/>. Acesso em: 24 fev. 2011.

32. GAME Physics Simulation. **Bullet Physics**, 2011. Disponivel em: <http://bulletphysics.org>. Acesso em: 20 fev. 2011.

33. ODE vs Bullet. **GameDev.net**, 2008. Disponivel em: <http://archive.gamedev.net/community/forums/topic.asp?topic_id=519399>.

Acesso em: 23 fev. 2011.

34. AAA titles using Bullet. **Bullet Physics Forum**, 2009. Disponivel em: <http://www.bulletphysics.org/Bullet/phpBB3/viewtopic.php?p=11971>. Acesso em: 20 fev. 2011.

35. BULLET in SIGGRAPH collision detection course, Toy Story 3 game, Sherlock Holmes and A-Team movie. **Bullet Physics**, 2010. Disponivel em: <http://bulletphysics.org/wordpress/?p=187>. Acesso em: 20 fev. 2011.

36. PHYSX. **NVIDIA**. Disponivel em: <http://www.nvidia.com/object/physx_new.html>. Acesso em: 20 fev. 2011.

37. RABIN, S. (Ed.). **Introduction to Game Development**. 2ª. ed. Boston: Charles River Media, 2010.

38. SMITH, R. Manual (Introduction). **Open Dynamics Engine Wiki**, 2006. Disponivel em: <http://opende.sourceforge.net/wiki/index.php/Manual_%28Introduction%29>. Acesso em: 06 fev. 2011.