UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ALFREDO MATTOS DE BARCELLOS

Desenvolvimento de um Motor de Física para o Console de Jogos Xbox 360™

Trabalho de Graduação.

Prof. Dr. Manuel Menezes de Oliveira Neto
Orientador

Porto Alegre, novembro de 2008.

AGRADECIMENTOS

Ao professor Manuel, por me manter na linha;

Ao Vitor, por me ajudar na revisão da monografia;

Ao Bruno, por me incentivar na busca da computação gráfica;

Ao Maurício, por me mandar não adiar o trabalho de conclusão;

E à minha família, que sempre me deu todo o suporte e apoio necessários para a realização deste curso.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

O Microsoft® XNA™ é uma plataforma de desenvolvimento de jogos para Windows™ e Xbox 360™. Seu alvo são desenvolvedores casuais que não tiveram a oportunidade de explorar suas idéias por causa de limitações de mercado e/ou de conhecimento. Apesar de ser suficientemente completo em termos de bibliotecas gráficas, o XNA™ não oferece um motor de Física para realizar simulações realistas do movimento de objetos.

Enquanto alguns motores foram desenvolvidos para o XNA™, até então nenhum motor de Física que fosse simples de usar e intuitivo foi demonstrado rodando no Xbox 360™. O objetivo deste trabalho é desenvolver um motor de Física simples capaz de rodar no Xbox 360™ e, assim, contribuir para preencher esse importante vazio. O projeto implementa o movimento básico de corpos e suporta colisões entre esferas e planos. O motor roda em PCs convencionais e no Xbox 360™. Seu código fonte está disponível no Source Forge.

Palavras-Chave: XNA™, motores de Física, jogos, Xbox 360™, simulação de Física.

Development of a physics engine the Xbox 360™

ABSTRACT

The Microsoft® XNA™ is a game development platform for the Windows™ and Xbox 360™ systems. Its targets are casual game developers that so far have not had the opportunity to explore their ideas due to market and/or knowledge limitations. Despite being sufficiently complete in terms of graphic libraries, the XNA™ does not provide a Physics engine to allow realistic simulation of object movements.

While some engines have been developed for the XNA™ platform, so far no easy-to-use and intuitive Physics engine has been demonstrated to run on the Xbox 360™. The goal of this project is to develop a simple Physics engine capable of running on the Xbox 360™ and, therefore, contribute to fill in this important gap. The project implements the basic bodies' movements and provides support for collisions among spheres and planes. The engine runs on conventional PCs and on the Xbox 360™. Its source code is available at Source Forge.

Keywords: XNA™, Physics engine, games, Xbox 360™, Physics simulation.

1 INTRODUÇÃO

O mercado mundial de jogos eletrônicos é um dos ramos da informática que mais cresce no mundo atualmente (GAMESINDUSTRY, 2008). Seu faturamento já é um dos maiores do ramo do entretenimento, superando a milionária renda gerada pela indústria do cinema (GAMESINDUSTRY, 2008). Este mercado é também um dos maiores incentivadores de pesquisa e desenvolvimento novas tecnologias de *hardware* gráfico.

O fato da indústria de jogos estar crescendo com velocidade, fez com o mercado se tornasse muito lucrativo e, conseqüentemente, muito competitivo. Os jogos de hoje em dia são feitos de forma muito mais complexa que antigamente, onde cada uma das áreas que compõe o jogo é profundamente elaborada, requerendo muito investimento por parte das companhias de desenvolvimento e dessa forma, tornando a criação de um jogo um investimento de risco. Como alguns jogos chegam a custar cerca de 30 milhões de dólares, as empresas tendem a investir apenas em projetos cuja chance de sucesso é garantida (GAMASUTRA, 2008). É por esse motivo que temos tantas seqüências de jogos hoje em dia. Se um jogo já fez sucesso, é bem mais provável que vá fazer sucesso novamente com uma nova versão do *software*. Esse fenômeno acaba por inibir a exploração e desenvolvimento de idéias originais para projetos de jogos, pois são considerados de excessivo risco e não valem a aposta de milhões.

No final de 2006, a Microsoft® resolveu incentivar o desenvolvimento das idéias que foram abandonadas por falta de investimento, lançando o Microsoft® XNA™, uma plataforma para desenvolvimento de jogos gratuita e de fácil utilização. Esse tipo de incentivo é um grande motivador para idéias de cunho social, como jogos educativos, que nem sempre são explorados como devem por causa dos custos de desenvolvimento.

Mesmo sendo o Microsoft® XNA™ um interessante recurso, o projeto ainda é jovem e por isso, muitas funcionalidades comumente utilizadas na programação dos jogos ainda não foram desenvolvidas propriamente. A comunidade internacional de XNA™, tem produzido diversas soluções para os tópicos que ainda estão em aberto, seja implementando algoritmos já utilizados em outros ambientes ou desenvolvendo novas formas de solucionar os obstáculos encontrados. Tudo isso é feito esperando que nas novas versões da plataforma, as soluções discutidas e implementadas sejam absorvidas pela ferramenta.

Dentro desse cenário, um dos componentes que expressam grande importância no desenvolvimento de jogos atual é a tentativa de aproximação da realidade. Normalmente, relacionamos muito da realidade com o aspecto visual. Quanto mais se parecer com o que nossa visão nos proporciona, mais "real" está para nós. Por isso, tivemos grandes avanços na parte de visualização com a introdução dos *shaders*[1] que

[1] Técnicas de programação diretamente nas placas gráficas.

proporcionaram inovadoras possibilidades de tratamento da imagem. O resultado disso é a capacidade de prover um maior grau de realismo fotográfico para o jogador.

Outro importante componente para aproximar o virtual do real é a questão do movimento dos corpos dentro de um espaço. A simulação Física é encontrada na grande maioria dos jogos desenvolvidos hoje em dia, exatamente por sua importância dentro da construção de uma cena de um jogo. O grau de realismo proporcionado pela simulação Física se dá pelo fato de que temos as leis que regem nosso mundo representadas dentro do mundo virtual Temos duas opções para o desenvolvedor: implementar um subconjunto das leis Físicas; ou utilizar uma biblioteca de Física previamente desenvolvida. Tais bibliotecas de Física também recebem o nome de Motor de Física (do inglês *Physics Engine).*

Os usuários do Microsoft® XNA™ são, em grande maioria, desenvolvedores iniciantes (CREATORS CLUB, 2008) que estão aprendendo a desenvolver seus jogos e que não tem as noções de Física necessárias para atender seus próprios anseios de retratação da realidade. Não demorou muito para que vários usuários começassem a requisitar, dentro dos veículos de comunicação da comunidade, métodos ou bibliotecas que suprissem suas necessidades. Foram assim que muitos motores de Física começaram a nascer.

A Física quando tratada em duas dimensões (2D) é mais simples de ser implementada do que a tratada em três dimensões (3D). Esse fato ocasionou uma explosão na quantidade de projetos de motores de Física em 2D. Vários obtiveram sucesso, alcançando a maturidade e sendo utilizado pelos usuários; mas muitos outros, nunca chegaram a ser finalizados e caíram em esquecimento.

Uma solução possível para ter um motor de Física no XNA™ seria portar bibliotecas já existentes utilizadas em outros sistemas e linguagens para o XNA™. Essa solução é bem aceita, mas tem uma grave desvantagem: a baixa portabilidade.

A plataforma do XNA™ tem como base a plataforma .NET Compact Framework, modificada para o Xbox 360™. Essa versão implementa um subconjunto do .NET Compact Framework original, e recebeu algumas otimizações para tomar proveito do poder de processamento do Xbox 360™. Assim, o XNA™ Game Studio™[2] permite que jogos para Windows se beneficiem da plataforma .NET original, mas quando o projeto especifica que será para lançamento no Xbox 360™, é utilizado então este subconjunto específico da plataforma.

Quando uma biblioteca é portada de outra linguagem, frequentemente algoritmos pré-compilados que são acessados por um sistema, e ai é que se encontra sua desvantagem. Quando os códigos são compilados sobre outro sistema que não seja a ".NET Compact Framework for Xbox 360™", seu uso dentro do Xbox 360™ fica impossibilitado, visto que a plataforma não executa códigos que não são escritos para seu ambiente.

Outro requisito de escolha de um motor de Física é a sua facilidade de utilização pelos usuários. Em alguns casos, o conhecimento necessário para integrá-lo a um jogo é tão grande, que o desenvolvedor precisa estudar toda a Física envolvida para entender como utilizar a biblioteca. Em contrapartida, temos casos em que uma biblioteca menos

[2] Ferramenta de desenvolvimento utilizada para programação dos jogos.

otimizada mas bem escrita, ou bem comentada – no caso de bibliotecas com código aberto – tem maior aceitação, pois é considerada mais fácil de ser usada.

Uma outra vantagem que motiva a escolha de uma biblioteca é a questão de que será disponibilizado pela Microsoft® a possibilidade de comercializar jogos desenvolvidos para Xbox 360™ utilizando XNA™ dentro de sua rede "Xbox Live™" (CREATORS CLUB, 2008). Fato que atrai muitos desenvolvedores iniciantes, que gostariam de ter frutos do seu trabalho mas sem os grandes investimentos e riscos que exigem o desenvolvimento de jogos atualmente. Podendo ser o início de uma promissora carreira de forma mais segura e acolhedora.

Se tornou grande a necessidade de um motor de Física 3D desenvolvido especificamente para o XNA™, para ser utilizada no Xbox 360™, que tenha abrangência em três dimensões: de código aberto, didático e de fácil utilização.

1.1 Objetivos

Este trabalho teve como objetivo responder à necessidade reportada pelos usuários da comunidade, onde foi desenvolvido um simples motor de Física com objetivos principais em rodar no Xbox 360™ e de ser de fácil uso.

Dentro de suas características, podemos destacar que, nesta fase do projeto, seu sistema de colisões tem realiza detecção apenas usando esferas e planos. Já o sistema de tratamento da colisão é implementado para corpos completamente rígidos com respostas de forças baseadas em impulso.

O resultado final desta etapa foi disponibilizado no Source Forge, batizado com o nome de *Alfred`s World* (ALFREDS WORLD, 2008).

1.2 Organização do Texto

No capítulo 2, será apresentado o histórico e classificação de motores de Física em um contexto geral, desvinculado de plataformas ou linguagens. Os conceitos utilizados em cada motor definem e classificam seu sistema dentro de um conjunto de seus similares, levando em consideração a forma de tratamento de forças, quais tipos de objetos são tratados na simulação e as técnicas de detecção de colisão aplicadas.

No capitulo 3, é apresentada uma revisão da Física que será utilizada para o desenvolvimento da biblioteca, partindo dos conceitos básicos das leis de Newton, e tratamento de colisões até o tratamento de torques e rotação em três dimensões.

No capitulo 4, é descrita a forma como foi implementado o sistema, apresentando conceitos clássicos e detalhes de implementação que fazem relação com a teoria Física apresentada anteriormente.

No capítulo 5, temos a análise dos resultados obtidos, da qualidade e extensão dos objetivos alcançados pelo trabalho, incluindo imagens demonstrativas e estatísticas de desempenho do sistema.

Encerrando, no capítulo 6 são expostas as conclusões do trabalho, relatando como foi a experiência durante o desenvolvimento do trabalho e quais são as possibilidades para realização de trabalhos futuros para melhoramentos do sistema atual.

2 TRABALHOS RELACIONADOS

Criar um motor útil significa balancear a complexidade da programação com a sofisticação dos efeitos que se quer simular. Seguem algumas distinções que ajudam a categorizar as diferentes abordagens (MILLINGTON, 2007).

2.1 Classificação por Tipos de Objetos

A primeira distinção entre tipos de motores de Física é dividida entre aqueles que simulam *corpos rígidos* e os chamados *motores de massas-agregadas* (MILLINGTON, 2007). Os motores de corpos rígidos tratam objetos como um todo e trabalham sua forma de movimento e rotação. Por exemplo, uma caixa é um objeto único e pode ser simulado como um objeto inteiro. Já um motor de massas-agregadas trata objetos como se eles fossem feitos de várias massas menores. Esta mesma caixa podia ser simulada como se fosse feita de oito massas, uma em cada canto, conectadas por varas (Figura 2.1).

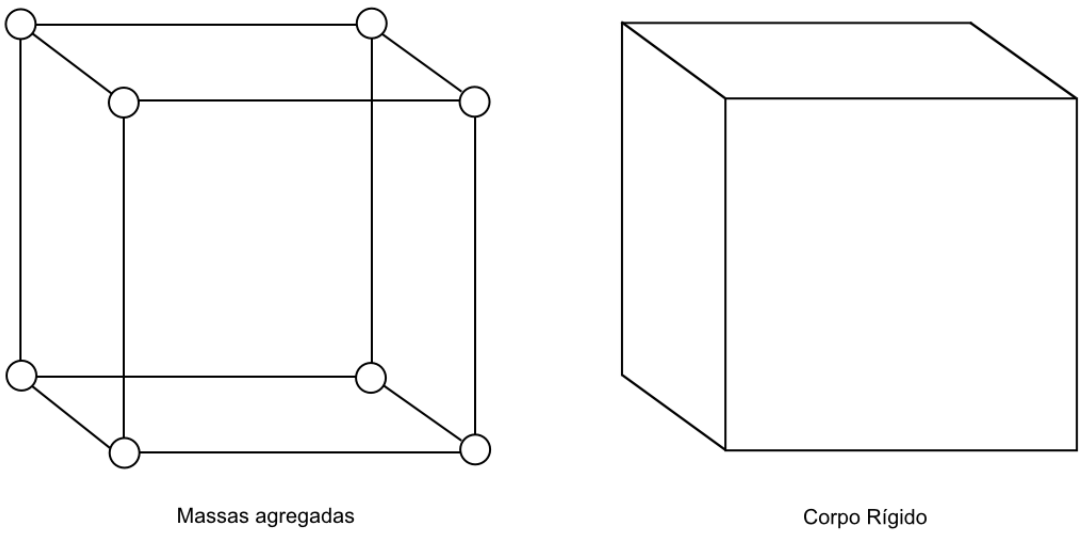

Figura 2.1: Tipos de objetos

Motores de massas-agregadas são mais simples de programar, pois não precisam manipular rotações. Cada massa é posicionada em um ponto específico, e as equações de movimento podem ser expressas puramente em termos de movimento linear. O corpo como um todo irá rotacionar naturalmente como resultado das limitações de movimento linear de cada componente (Figura 2.2).

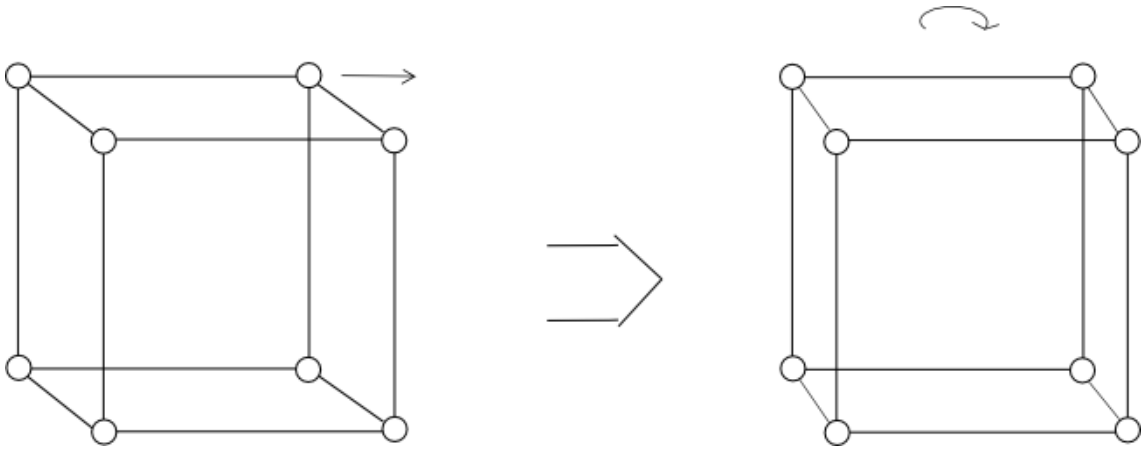

Figura 2.2: Rotação por movimento de uma massa agregada

Como é muito difícil fazer objetos verdadeiramente rígidos em um motor de Física (MILLINGTON, 2007), fica também difícil de fazer objetos firmes em um sistema de massas-agregadas. A caixa do exemplo anterior vai ter certo nível de flexibilidade em si. Para tornar isso invisível para o observador, é necessário implementar mais linhas de código para reconstruir a idéia de caixa rígida a partir desse conjunto de massas. Enquanto o sistema básico de massas-agregadas é simples para programar, esses trechos extras de código tornam-se um ponto negativo para esse paradigma de sistema, onde os custos de implementar esses trechos se tornam maiores que as vantagens oferecidas pela "simples implementação" elogiada anteriormente.

Para este trabalho, foi escolhida a abordagem de corpos rígidos, pois, no sistema baseado em corpos completamente rígidos, usufruímos parcialmente da vantagem de fácil implementação citada para o sistema de massas-agregadas, mas desviamos da desvantagem do problema de flexibilidade e rotações ao adicionarmos o tratamento específico de rotação para os corpos.

2.2 Classificação por Resolução de Contatos

A segunda distinção envolve a forma como objetos que se tocam são processados. Grande parte da dificuldade de escrever um motor de Física de corpos rígidos é simular contatos (MILLINGTON, 2007): partes onde dois objetos se tocam ou estão conectados. Neste contexto, se incluem objetos em repouso sobre o chão, objetos conectados entre si, e para colisões analisadas durante o momento de impacto.

Uma abordagem é tratar esses contatos um a um, tendo certeza que cada contato seja detectado e tratado corretamente. Essa é a chamada *abordagem iterativa*, e tem como vantagem a velocidade, pois cada contato é resolvido rapidamente. Porém, a desvantagem é que um contato pode influenciar outro e algumas vezes essas interações podem ser significativas. Esta é a abordagem mais fácil de implementar e é base para outros métodos mais complexos.

Uma forma mais próxima da realidade é calcular a exata interação entre diferentes contatos e calcular um abrangente conjunto de efeitos para serem aplicados a todos os objetos ao mesmo tempo. Essa é chamada *abordagem Jacobiana* (MILLINGTON, 2007), mas tem como desvantagem o fato de ser muito demorada: a matemática envolvida exige muito processamento. Existem casos onde simplesmente não existe resposta válida, e o desenvolvedor precisa resolver esses problemas caso a caso. Vários sistemas de Física desenvolvidos utilizam essa abordagem, e cada uma tem suas próprias técnicas para resolver as equações e tratar as inconsistências.

Uma terceira opção é calcular um novo conjunto de equações baseadas nos contatos e limitações entre objetos (MILLINGTON, 2007). Ao invés de usar as leis de Newton diretamente, com base nelas é criado um conjunto específico de equações exatamente para tratar a configuração dos objetos que estão sendo simulados. Essas equações serão diferentes a cada quadro, e grande parte do esforço do motor de Física estará em desenvolvê-las. Esta é chamada de *abordagem de coordenadas reduzidas* (MILLINGTON, 2007). Alguns sistemas de Física têm sido criados utilizando essa abordagem, e é a mais comum a ser usada em *software*s de engenharia por sua apurada simulação (MILLINGTON, 2007). Infelizmente, esta abordagem é muito lenta e assim, não é útil em aplicações de tempo real, que é o caso de jogos eletrônicos, onde a velocidade e a ilusão de realidade são mais importantes que a precisão.

2.3 Classificação por Tratamento de Forças

A terceira distinção está em como o motor realmente soluciona os contatos. Quando um livro está em repouso sobre uma mesa, a mesa está empurrando o livro com uma força igual a da gravidade que o puxa para baixo. Essa força estará constantemente empurrando o livro pelo tempo que ele continuar em repouso sobre a mesa. A velocidade do livro não mudará em função dessas forças, mesmo que possa ainda ser alterada por outras forças externas.

Alguns motores de Física separam a noção de força e impulso, onde usam forças para contatos em repouso e impulsos para colisões. É uma abordagem rara, pois necessita tratamento separado para repouso e para colisão. Motores de Física mais comuns tratam tudo como uma força: impulsos são então apenas forças atuando em um tempo muito pequeno. Esses são os motores *baseados em força* e funcionam da mesma forma que o mundo real funciona. Infelizmente, a matemática de forças é mais difícil do que a matemática de impulsos. Estes motores *baseados em força* geralmente utilizam a abordagem *Jacobiana* ou a *de coordenadas reduzidas* (MILLINGTON, 2007).

Outros motores usam impulsos para tudo: o livro em repouso sobre a mesa é mantido ali por inúmeras micro-colisões ao invés de ser mantido por uma força constante. Esses são chamados de motores *baseados em impulso*. Em cada quadro do jogo, o livro recebe uma micro-colisão que o mantém sobre a superfície da mesa até o próximo quadro. Se a taxa de quadros ficar extremamente baixa, é possível ver objetos em repouso erroneamente vibrando em função disso. Mesmo assim, esta abordagem não é muito diferente da abordagem *baseada em força*. Só é mais fácil de implementar e tem a vantagem de ser mais flexível e adaptável. É comumente utilizada em motores de Física desenvolvidos por inúmeras empresas comerciais.

3 REVISÃO DE CONCEITOS

Para desenvolver um motor de Física, podemos nos embasar em uma gama de referências (MILLINGTON, 2007; CONGER, 2004; PALMER, 2005; BOURG, 2002) que apresentam diferentes técnicas e modelos de simulação para retratar o movimento dos corpos. Apesar de distintos, um aspecto que a maioria tem em comum é a representação das leis de Newton.

As leis de Newton (PALMER, 2005) expressam a forma como os corpos se movem, através de suas massas e suas distribuições de massa dentro de um corpo, seus movimentos podem ser diferentes, mesmo se tratando de um corpo de mesma forma. Dentro destas leis é estabelecido o conceito de força, que quando aplicada a corpos criam alteração no movimento destes.

Por isso, dividimos as seções a seguir de forma a apresentar os conceitos necessários para as simulações de modo ordenado e crescente, onde serão dispostas as adaptações matemáticas do nosso mundo real para o novo mundo virtual.

3.1 Movimento Linear

O movimento de um corpo pode ser separado em duas componentes, *movimento linear* e *movimento angular* (PALMER, 2005). O *movimento linear* trata exclusivamente da alteração da posição do centro de massa de um corpo, que finalmente retratará a posição do corpo no espaço.

O centro de massa é considerado como o ponto médio de distribuição de todas as massas pontuais que compõe o corpo (PALMER, 2005). O centro de massa não precisa estar especificamente no interior do corpo, pode ter outra posição visto que depende da forma e distribuição da massa do objeto (Figura 3.1).

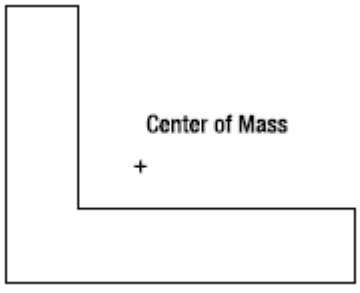

Figura 3.1: Centro de massa (PALMER, 2005). Note que o centro de massa de um objeto pode estar fora do mesmo.

A primeira lei de Newton diz que: “Quando a resultante das forças que atuam sobre um corpo for nula, esse corpo permanecerá em repouso ou em movimento retilíneo uniforme” (PALMER, 2005). Também chamada “princípio da inércia” descreve como o movimento de um corpo se dá quando não existem forças atuando sobre ele. O movimento de um corpo é a alteração da sua posição. Se pudermos mapear a mudança da posição em relação ao tempo que levou para realizar essa mudança, tem-se o conceito de velocidade, que é regido pela equação

$$v = \frac{\Delta x}{\Delta t}$$

onde **Δx** é o deslocamento espacial e **Δt** é o deslocamento temporal. Note que estes deslocamentos discretos representam as derivadas de **x** e **t**. Este mapeamento do tempo é realizado diferenciando-se cada grandeza desejada em relação ao tempo. Então, sabendo que a velocidade determina a mudança da posição de um corpo em relação ao tempo, podemos intuir também o conceito de aceleração. A aceleração é a variação da velocidade em relação ao tempo, que é descrita pela equação

$$a = \frac{\Delta v}{\Delta t}$$

onde **Δv** é a variação de velocidade e **Δt** é o deslocamento temporal. Novamente, estes deslocamentos discretos representam as derivadas de **v** e **t**. Porém, a aceleração tem uma origem especifica. A segunda lei de Newton diz: “Se existe a ação de forças ou a resultante das forças atuantes sobre um corpo não é nula, ele sofrerá a ação de uma aceleração inversamente proporcional à sua massa.” (PALMER, 2005). Com esta lei, introduzimos o conceito de força. Baseando-se no que diz a lei, podemos equacionar a relação entre massa e aceleração para a equação de força

$$F = m * a$$

onde **m** é a massa e **a** é a aceleração. Este conceito pode envolver outros fundamentos como trabalho e energia, que não são necessários para a simulação de movimento simples que esse projeto se propõe. Na Física, a força é descrita como a única grandeza capaz de alterar a posição de um corpo, pois é ela que lhe dá aceleração, que gera velocidade, e que finalmente, altera sua posição.

A massa determina o quão difícil é mover um corpo linearmente. Uma mesma força aplicada a dois corpos com massas diferentes gera movimentos diferentes.

Essas grandezas sempre expressas em escalares, como 20km/h, 9,8m/s², tem suas propriedades direcionais implícitas. Em nossa representação, trocamos estes conceitos escalares pelo tratamento destas métricas usando álgebra de vetores, cujas vantagens intrínsecas se provam ao longo da simulação.

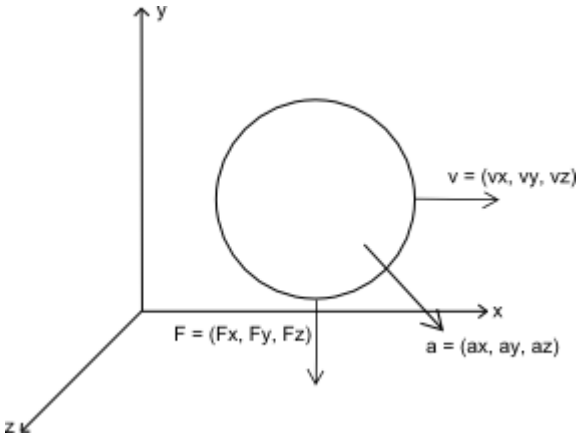

Figura 3.2: Exemplo da representação vetorial. São apresentados 3 vetores que regem a dinâmica de movimento do corpo, onde cada um é construído a partir de 3 componentes para representar o espaço 3D.

A posição de um corpo no espaço 3D pode ser composta de três componentes que expressam sua posição em relação a cada eixo que representa o espaço vetorial As grandezas que variam com o tempo, velocidade e aceleração, são facilmente tratadas como vetores (Figura 3.2), pois carregam junto, além da informação escalar, a informação direcional, onde o módulo do vetor serve para determinar o componente escalar da grandeza. Esta questão de se aproveitar do fator direcional do vetor também é aproveitado pela representação de força, onde ela não varia exatamente com o tempo, mas sendo uma grandeza vetorial, temos seu escalar, sua direção e seu sentido todos embutidos na representação de vetor.

3.2 Movimento Angular

No início da seção anterior, discutimos como o movimento de um corpo pode ser separado entre movimento linear e movimento angular. Sendo o movimento linear responsável pela alteração da velocidade ao longo do tempo e o movimento angular pela orientação do corpo com relação ao tempo.

Primeiramente, é importante apresentar o conceito representativo de orientação no espaço 3D, que é amplamente estudada e existem várias formas de ser representada em sistemas virtuais. A orientação em 2D usa dois eixos relacionados e assim, apenas um ângulo entre eles. Já no caso do espaço 3D, temos três eixos com os quais nos orientar no espaço, o que gera vários ângulos possíveis entre eles, o que torna a orientação um desafio se encarada desta forma “relacional” entre eixos. É por isso que existem estudos sobre outras formas de encarar a orientação em três dimensões que seja intuitiva e facilite o uso.

A abordagem utilizada neste trabalho é chamada de *orientação eixo-ângulo* (CONGER, 2004). Ela trata a orientação como uma rotação sobre um eixo arbitrário por um ângulo qualquer. Dessa forma, é possível ter, primeiramente, a noção de direção provida pelo eixo e também a quantidade necessária de rotação sobre aquele eixo (Figura 3.3).

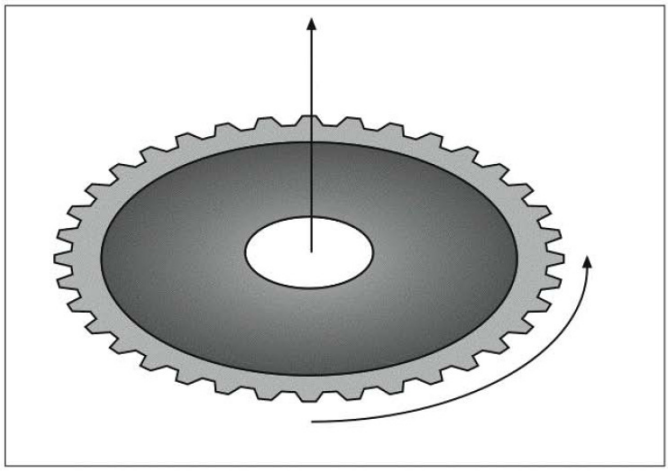

Figura 3.3: Orientação eixo-ângulo (CONGER, 2004). É determinado um eixo, que é representado por um vetor, e um ângulo sobre o qual o objeto é rotacionado em relacao a este eixo.

Se formos considerar que o espaço em que vamos trabalhar é composto de três eixos, utilizando esta orientação eixo-ângulo três vezes (Figura 3.4), uma em cada eixo, teremos uma orientação completa no espaço 3D, que é a chamada representação de ângulos de Euler (CONGER, 2004).

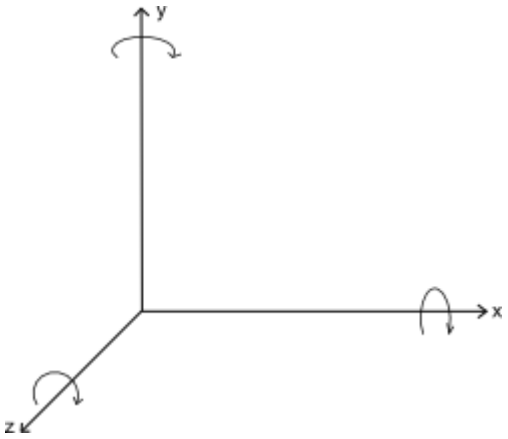

Figura 3.4: Orientação nos três ângulos de Euler. Tendo controle dos 3 graus de liberdade rotacional, podemos orientar o corpo para qualquer direção do espaço 3D.

Como estamos representando a orientação de um corpo baseando-se na rotação de cada eixo do espaço, podemos armazenar essa informação de três componentes dentro de um vetor.

Digamos então que o corpo tem alguns desses ângulos variando em relação ao tempo; apresentamos então a idéia de velocidade angular. A *velocidade angular* é a versão de rotação da velocidade linear, onde invés da posição, temos a orientação do corpo variando com o tempo. Logo, temos como equação da velocidade angular **ω** a equação

$$\omega = \frac{\Delta\theta}{\Delta t}$$

onde **Δθ** é a variação de ângulo e **Δt** é a variação do tempo. Sendo assim, como temos no movimento linear a variação da velocidade com o tempo, também podemos esperar que exista a variação dessa velocidade angular com relação ao tempo. Essa grandeza é, intuitivamente, chamada de aceleração angular **α**:

$$\alpha = \frac{\Delta\omega}{\Delta t}$$

onde **Δω** é a variação da velocidade angular e **Δt** continua sendo a variação do tempo. Seguindo a linha de raciocínio, estabelecemos antes que as forças eram as únicas reais responsáveis por alterar o movimento de um corpo, o mesmo vale para a rotação, onde a força aplicada sobre um ponto é chamada de torque. O torque (Figura 3.5) é a força que é necessária para rotacionar um corpo, em relação ao seu centro de massa.

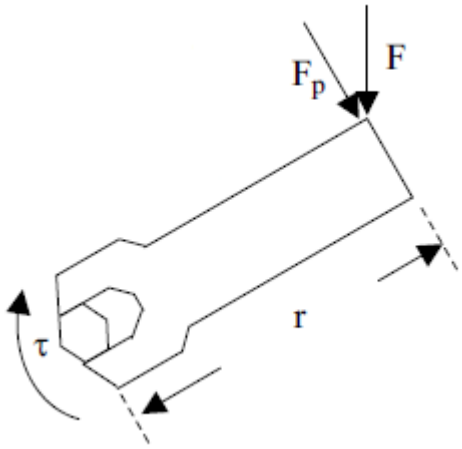

Figura 3.5: Exemplo de aplicação de torque (PALMER, 2005). O torque aplicado em um corpo, exerce uma rotação baseada no cosseno do ângulo entre o vetor tangente ao raio do objeto e a força aplicada.

Porém, da mesma maneira que a massa influencia o quanto uma força gerará de aceleração para um dado corpo, o torque é influenciado pelo *momento de inércia*. O momento de inércia representa o quão difícil é um objeto de ser rotacionado.

A massa é uma propriedade material. A massa de um objeto depende de que material o objeto é feito e quanto deste material existe. O *momento de inércia* é uma propriedade tanto material quanto geométrica. O *momento de inércia* de um objeto depende da massa do objeto e da forma do objeto (CONGER, 2004).

3.3 Detecção de Colisão

Nas seções anteriores, pudemos mapear o movimento dos corpos como um todo, manipulando o movimento linear e o movimento angular, mas grande parte do que faz um motor de Física ser interessante é o tratamento de colisões.

Colisões ocorrem quando dois corpos entram em contato e dependendo da situação, e das propriedades Físicas dos corpos é gerada uma reação. A terceira lei de Newton diz: “Se um corpo A aplicar uma força sobre um corpo B, receberá deste uma força de mesma intensidade, mesma direção e sentido oposto à força que aplicou em B”.

Também conhecida como lei da “ação e reação”, esta lei explica como ocorre a relação de colisão entre os corpos.

Porém, a simulação de *software* tem como premissa, primeiro determinar se ocorreu uma colisão para depois determinar sua reação. Assim, dividimos a simulação de colisões em duas partes: a *detecção de colisão* e o *tratamento de colisão*.

Em uma cena, temos que nossos corpos são compostos de pontos e linhas, sendo a primitiva mais básica gerada por eles, o triângulo. Seria intuitivo imaginar que testar se existe colisão entre dois corpos fosse um exercício de comparação de “toque” entre triângulos, mas dado o grande número de triângulos que os modelos incorporam hoje em dia, este seria um processo lento e ineficiente.

Com essa idéia em mente é que foram criados os volumes limitantes (Figura 3.6), do inglês *bounding volumes* (CONGER, 2004). Estes volumes nada mais são que substitutos invisíveis para as malhas que representam os corpos, e que são aproximações do formato do corpo. A principal vantagem dessa abordagem é que utilizando geometrias básicas ou um conjunto delas, os testes se tornam muito mais baratos em termos de processamento, e o resultado é muito próximo do real.

Figura 3.6: Tigre modelado com uma malha de triângulos e seu possível volume de limitação baseado em caixa (CONGER, 2004). Note que existem muitos triângulos e o algoritmo de colisão teria que testar contra cada um deles. A caixa também não é a melhor opção porque muitos espaços vazios são considerados integrantes do objeto.

Comumente, uma das mais utilizadas geometrias de base são as esferas. O intuitivo seria pensar em uma caixa, porém, por muitas vezes as esferas conseguem representar melhor a forma de um corpo, mesmo que para isso sejam usadas mais de uma esferas para representar o corpo (Figura 3.7).

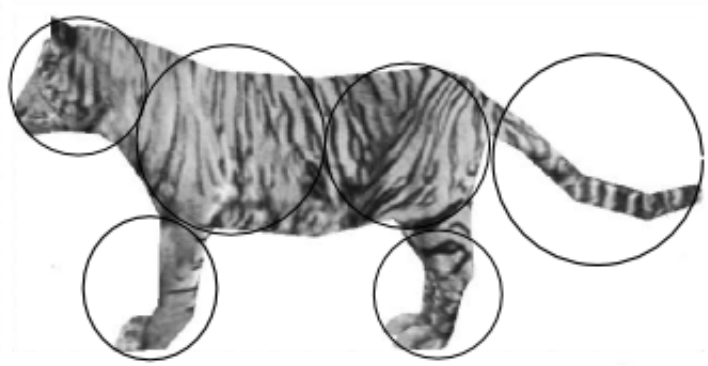

Figura 3.7: Volume limitante usando esferas. Possibilitam apenas 6 testes simples para detectar colisões, ao invés de milhares de testes; e ao mesmo tempo, tem uma composição mais fiel à malha original.

Outra vantagem do teste de colisão de esferas é que é notoriamente simples: podemos saber se uma esfera está em contato com outra se a distancia entre seus centros é menor ou igual que a soma de seus raios (Figura 3.8).

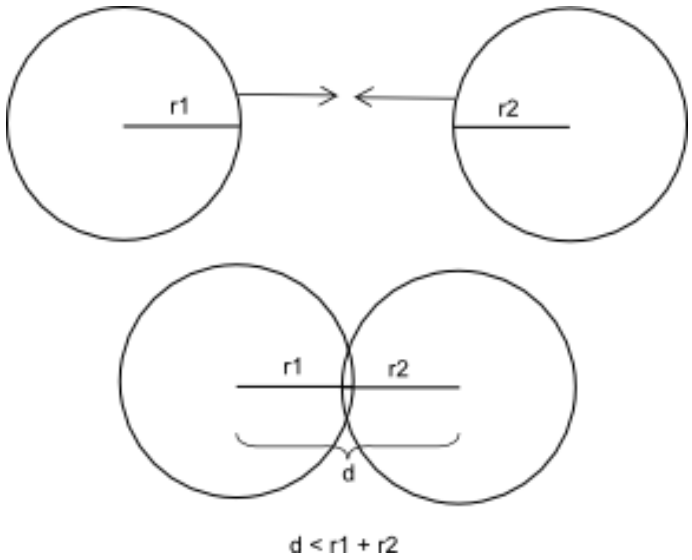

Figura 3.8: Detecção de colisão entre esferas. Se a distância entre seus centros por menor ou igual que a soma dos raios, temos uma colisão.

Ainda assim, existem outras geometrias que não são consideradas como volumes limitantes (*non-bounding volumes*), mas apresentam grande papel no sistema de colisões, cujo exemplo principal é o plano. Um plano tem como função dividir o espaço em dois, e por isso, não pode ser utilizado para interpretar nenhum tipo de corpo. Porém, existem entidades dentro de um jogo que não são necessariamente corpos, como componentes do cenário, por exemplo, as paredes e o chão. Nesses casos que a aplicação de planos no sistema de colisões se justifica, por ser uma solução que visa otimizar o processamento e facilitar a interpretação de cenários com os quais os corpos irão interagir (Figura 3.9).

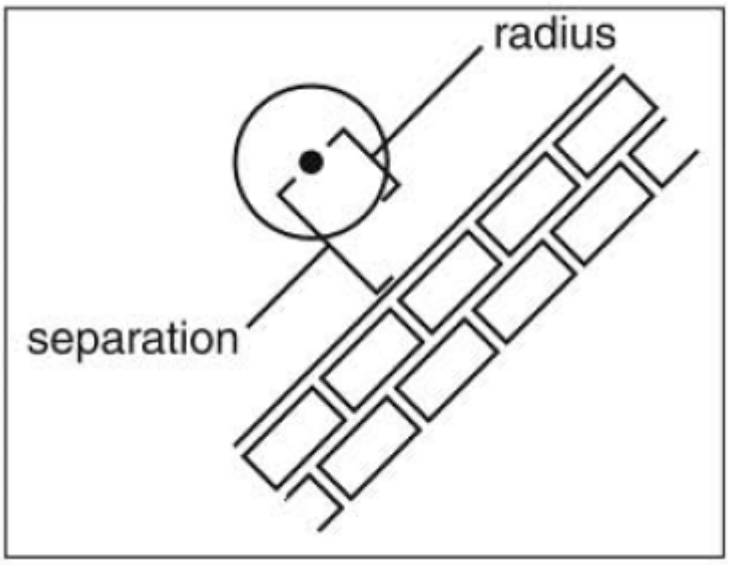

Figura 3.9: Planos no contexto de colisão (CONGER, 2004). O teste de detecção compara o valor do raio com o valor da distância entre o centro da esfera e o plano de colisão. Caso a distância seja menor ou igual que o raio, temos uma colisão.

O teste entre planos e esferas é realizado da seguinte forma: é calculada a distancia entre o plano e o ponto. Se essa distância for menor ou igual que o raio da esfera temos uma colisão. A equação para o cálculo da distância (Bourg, 2002) é

$$D = \hat{n} \cdot x_0 + p$$

onde **D** é a distância, $\hat{n}$ é a normal geradora do plano, $\mathbf{x_0}$ é o ponto em questão e **p** é a distancia do plano em relação à origem.

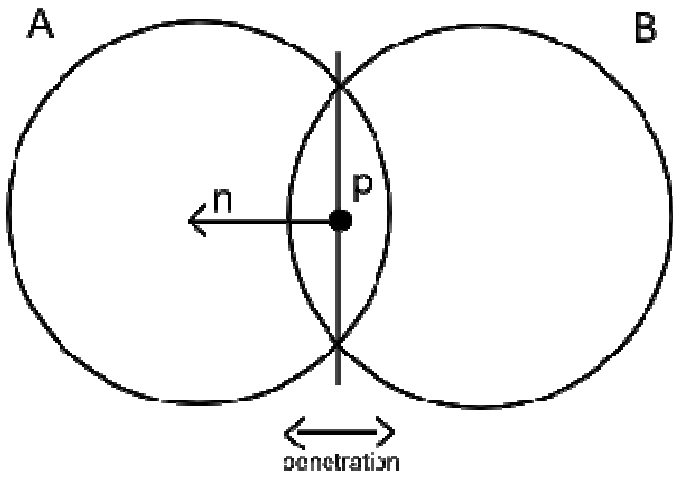

Figura 3.10: Componentes de resposta da detecção de colisão. **A** e **B** são os corpos envolvidos na colisão; **n** é a normal da colisão (que por definição parte de **B** para **A**); **p** é o ponto de colisão e "penetration" é o nível de penetração de um corpo sobre o outro.

Dentro do sistema de detecção de colisão é muito importante retornar algumas informações para o sistema central. Entre elas, estão listados os corpos envolvidos, a normal da colisão (que, por definição, é a normal do segundo corpo em relação ao primeiro), o ponto de colisão e o nível de penetração (Figura 3.10).

3.4 Tratamento de Colisão

Após detectarmos que ocorreu uma colisão, são necessárias medidas que simulem a resposta que os corpos teriam a tal colisão. Com esse último passo estaremos completando a representação da terceira lei de Newton e estabelecendo o principio de ação e reação. A resposta de uma colisão acontece em dois passos: a resolução de interpenetração e o tratamento de impulso.

O problema da interpenetração é o caso que ocorre quando o corpo se movimenta após a detecção da colisão ou, em outro caso, quando a colisão foi detectada, o corpo já estava muito avançado dentro do outro corpo. Essa questão deve ser resolvida antes de serem calculadas as novas velocidades baseadas em impulso, pois se não forem, os corpos que penetraram em uma geometria, podem se considerar em colisão novamente no próximo quadro e assim, infinitamente.

Este problema é solucionado fazendo com que o objeto recue um pouco sua posição em relação à normal de colisão. O valor deste recuo é determinado por uma componente informada pelo sistema de detecção, que é a profundidade de penetração. Estabelecendo este critério antes de iniciar as mudanças de velocidade, temos a garantia que nenhum objeto ficará contido dentro de outro.

Depois de resolvida a interpenetração, é hora de calcular como os corpos irão realmente reagir. A dinâmica nos apresenta o conceito de *impulso* como sendo a força que é gerada na colisão entre corpos. O impulso nada mais é que uma força muito grande exercida sobre um corpo em pouquíssimo tempo, e por isso retrata muito bem as características representadas por uma colisão alterando sua velocidade instantaneamente.

Ainda assim, da mesma forma que o movimento é dividido em movimento linear e movimento angular, a resolução da colisão trata de maneira diferente as respostas lineares e angulares. A resposta linear leva em consideração o vetor velocidade dos corpos, suas massas e seus coeficientes de restituição (também conhecidos como coeficientes de elasticidade). A participação dos coeficientes de restituição representa o quando da energia desenvolvida na colisão realmente se torna impulso. Esta constante varia de 0 a 1, onde 0 seria uma colisão que ocorre entre, por exemplo, duas bolas de lama, onde o impulso é nulo; e o 1 seria uma colisão cuja energia gerada é totalmente convertida em impulso.

A força de impulso $\mathbf{F_I}$ é determinada pela equação

$$F_I = \frac{-v_r(e+1)}{\frac{1}{m_1}+\frac{1}{m_2}+n\bullet\left[\left(\frac{(r_1\times n)}{I_1}\right)\times r_1\right]+n\bullet\left[\left(\frac{(r_2\times n)}{I_2}\right)\times r_2\right]}$$

onde $\mathbf{v_r}$ é a velocidade relativa entre os corpos, ou seja, a diferença entre suas velocidades; **e** é o coeficiente de restituição da colisão, que é calculado como a média

aritmética dos coeficientes de restituição dos corpos envolvidos; $\mathbf{r_1}$ e $\mathbf{r_2}$ são os raios de colisão de cada corpo, que são as distâncias entre o centro do objeto e o ponto de colisão; $\mathbf{m_1}$ e $\mathbf{m_2}$ são as massas dos corpos envolvidos; **n** é a normal da colisão e $\mathbf{I_1}$ e $\mathbf{I_2}$ são os momentos de inércia de cada corpo (MILLINGTON, 2007).

No caso da resposta linear, o impulso terá como função final alterar a velocidade (Figura 3.11), dando-lhe novos módulo, direção e sentido. No caso da resposta angular, o impulso é considerado uma força de torque sobre o corpo, de forma a gerar rotação a partir da colisão caso exista ângulo suficiente para tal.

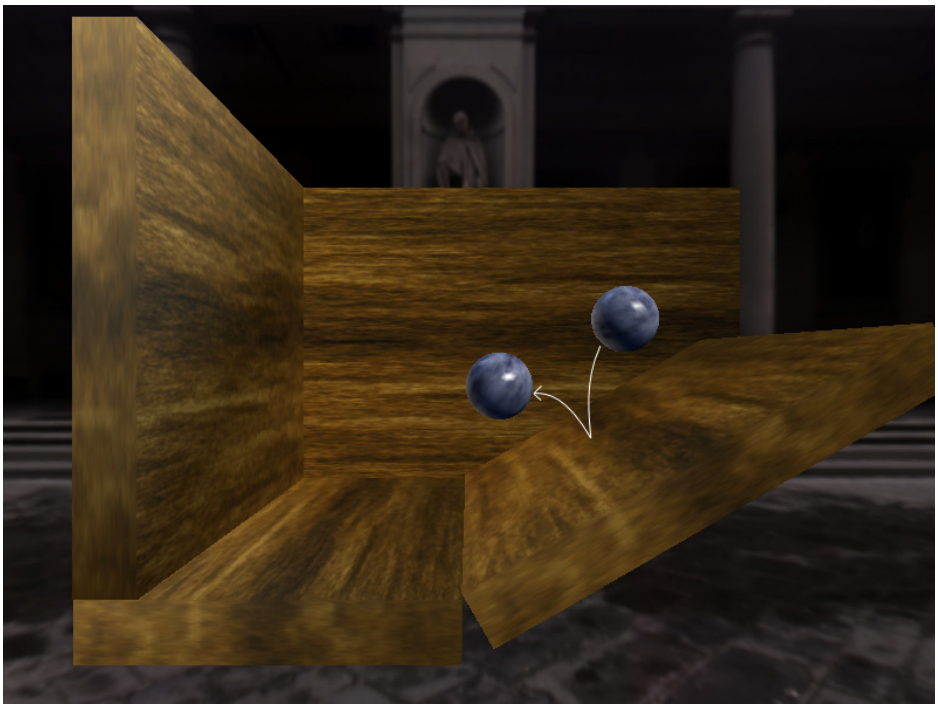

Figura 3.11: Exemplo de impulso no sistema implementado. Note que após a colisão, a força de impulso alterará a direção e intensidade da velocidade.

4 DESCRIÇÃO DO SISTEMA

Neste capítulo descrevemos a implementação da biblioteca proposta, cuja abordagem é diferenciada para que seja mais intuitiva e didática. Essa descrição geral se limitará a apresentar as classes e métodos apenas com caráter ilustrativo, ainda que as mais importantes funções sejam abordadas com maior detalhe para maior entendimento de seu funcionamento.

A biblioteca conta com 3 classes principais e algumas subclasses que aproveitam as vantagens da hierarquia de classes para tornar sua implementação mais simples e direta. Sua implementação também se aproveita de estruturas de dados providas pelo C#, linguagem nativa do XNA™, assim como outras oriundas do próprio XNA™.

4.1 Algoritmo Fundamental

Antes de detalhar cada classe, vamos estabelecer qual é o algoritmo utilizado para realizar a simulação Física como um todo. Ele parte do princípio que existem corpos na cena em questão e que suas propriedades Físicas estão previamente determinadas. O algoritmo é um laço composto de 4 passos: Atualização de Forças, Resolução de Movimento, Teste de Colisão e Tratamento de Colisão (Figura 4.1).

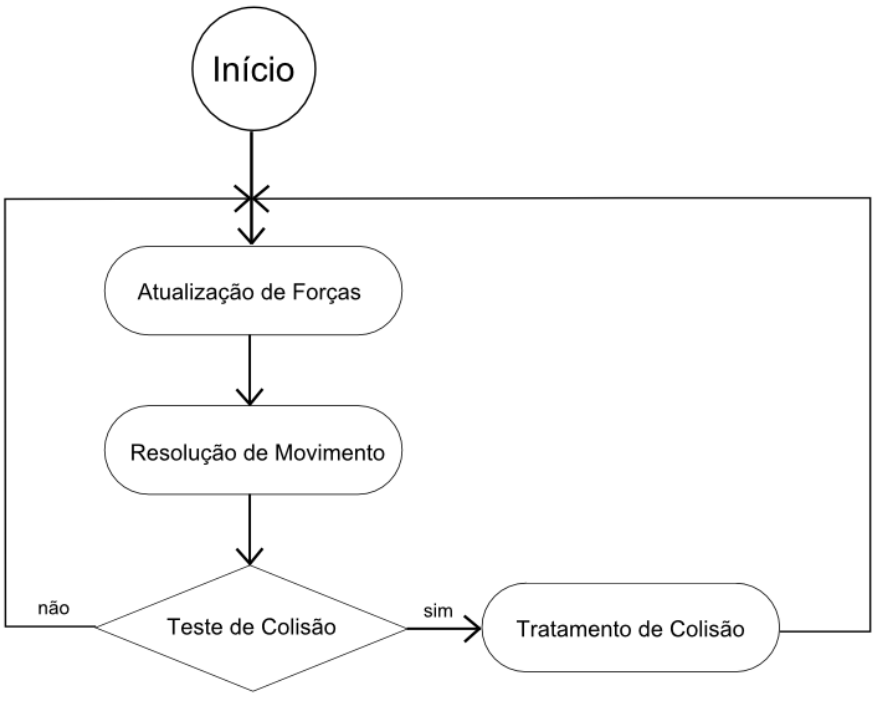

Figura 4.1: Fluxograma em alto nível do algoritmo

A fase de Atualização de Forças corresponde ao passo em que as forças que foram aplicadas sobre cada corpo devem ser transformadas em aceleração de acordo com a massa do corpo afetado.

Na Resolução de Movimento, cada corpo atualiza sua velocidade, posição e orientação no espaço para um dado intervalo de tempo.

O Teste de Colisão checa se dentro dos objetos movidos existe algum que esteja em colisão com outro, realiza este teste para cada corpo, comparando suas geometrias de colisão com a de todos os outros da cena.

O Tratamento de Colisão só é executado caso o Teste de Colisão retorne um valor positivo para o número de colisões naquele instante. Então, serão geradas as novas velocidades baseadas na equação de impulso para que no próximo instante já possam de mover de acordo com o esperado.

Tendo então a idéia do algoritmo, partiremos para as estruturas das classes que compõe o sistema para uma descrição com maior nível de detalhe. Existem quatro classes principais e duas subclasses de implementação de geometrias (Figura 4.2).

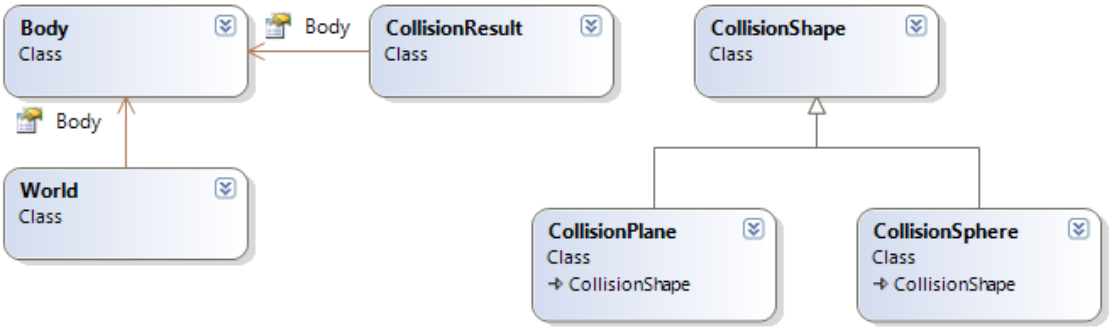

Figura 4.2: Diagrama de Classes

4.2 Classe CollisionShape

A classe CollisionShape (Figura 4.3) trabalha no nível de detecção de colisão e seu objetivo é tratar os volumes limitantes e seus companheiros de detecção de colisão que não se enquadram na definição de volumes limitantes.

Cada geometria que poderá ser utilizada deve herdar a classe CollisionShape. Por isso, esta classe tem métodos virtuais e campos comuns a todas as geometrias, onde cada uma apresentará suas implementações e campos especiais dentro de suas subclasses.

Dentro desta classe é que também ficam os métodos de detecção entre geometrias. Mesmo que não se saiba qual é a geometria, a classe deve estar preparada para realizar o teste corretamente. Abaixo temos uma representação reduzida da classe onde são apresentados somente seus métodos e propriedades.

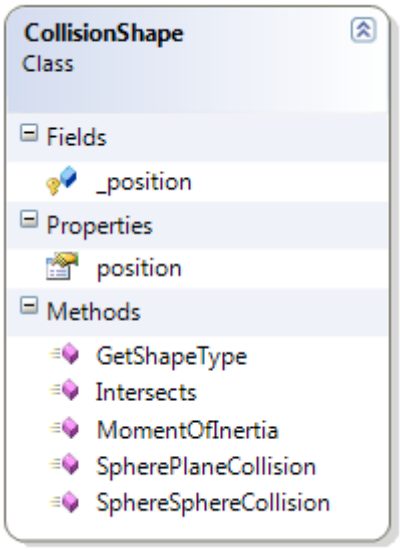

Figura 4.3: Diagrama da classe CollisionShape. É guardada a posição e determinadas as funções comuns entre os volumes de colisão.

Como propriedades, temos apenas a posição, que é a única propriedade básica comum entre as geometrias. Aqui também é que ficam as funções de detecção de colisão por geometria (SpherePlaneCollision e SphereSphereCollision) vistas no capítulo anterior. Existem também, uma função para o cálculo do momento de inércia baseado na geometria, uma função que retorna o tipo da geometria em questão, e por último o método de acesso para o sistema de detecção, que testa a geometria derivada desta classe com outra arbitrária.

Com base nesta classe, podemos criar quantas classes de geometria forem necessárias ao nosso uso (Figura 4.4). Neste trabalho, foram implementadas subclasses para as geometrias de esfera e plano. Seus diagramas seguem abaixo, seguidos de comentários de funcionamento para as diferenças entre elas.

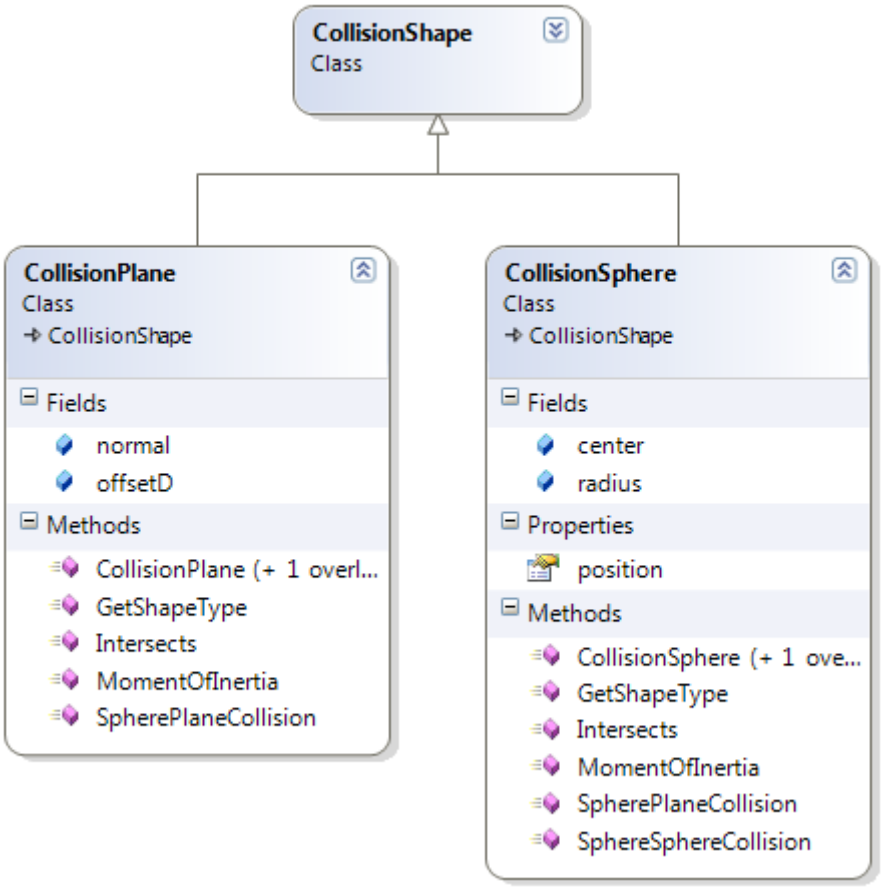

Figura 4.4: Diagrama das subclasses CollisionPlane e CollisionSphere. Herança de classes é utilizada para que todas as classes de colisão possam ser tratados de forma igual quando chamadas pelas funções.

Para a classe CollisionPlane, temos como campos, a normal e o ponto de formação do plano e para a classe CollisionSphere, os campos são o centro e o raio da esfera

Para completar a seção de classes de colisão, temos a classe CollisionResult (Figura 4.5) que tem como função armazenar os resultados de uma dada colisão e serve como ponto de comunicação entre a o sistema central e a resposta do teste de colisão. Seus campos são: os corpos envolvidos, a normal da colisão, o ponto da colisão e o nível de penetração.

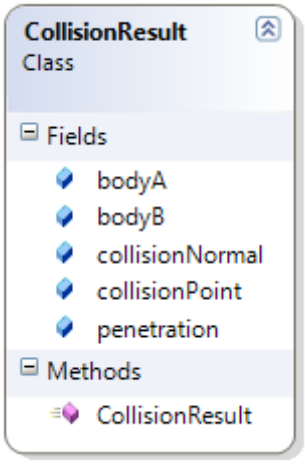

Figura 4.5: Diagrama da classe CollisionResult

4.3 Classe Body

Já estabelecemos, dentro da classe CollisionShape (Figura 4.6), as estruturas necessárias para a detecção de colisão entre geometrias, mas não podemos esquecer que estas geometrias estarão sendo usadas apenas para representar os corpos que estão realmente presentes na nossa cena.

Para tratamento dos corpos como entidade participante do mundo, é introduzida a classe Body, que é basicamente um repositório de características Físicas e dinâmicas do corpo em questão. Também são adicionados à estrutura, elementos não físicos, como uma variável que retém a malha de triângulos que representa o corpo no espaço 3D, que dentro dos conceitos do XNA™, é interpretado como um modelo.

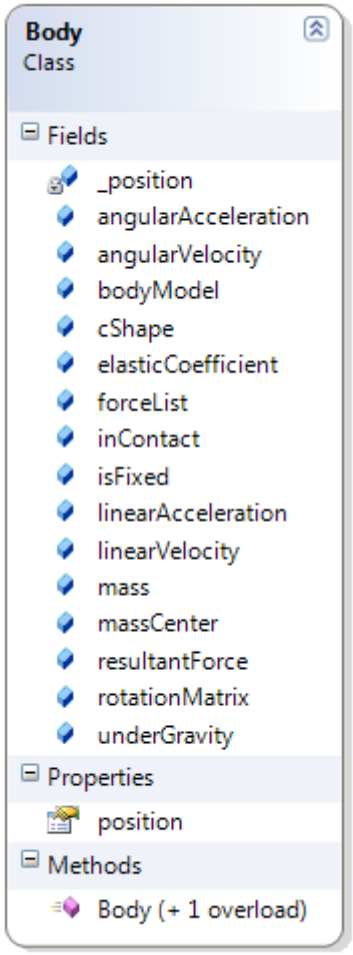

Figura 4.6: Diagrama da classe Body

Temos velocidades e acelerações lineares e angulares; um campo para guardar a malha de triângulos (bodyModel) e outro para guardar a geometria de colisão (cShape); uma variável de coeficiente de restituição; a lista de forças aplicadas no corpo em um dado instante e a sua força resultante; sua massa; seu centro de massa; sua matriz de rotação; e três componentes de estado: inContact, determina se o corpo está em repouso sobre outro; isFixed, determina se um corpo será fixo e não terá alteração na sua posição; e underGravity, para saber se o corpo sofrerá a influencia da força da gravidade, que é tratada diferentemente.

Esta classe será a classe base para o tratamento da movimentação do corpo no espaço, por isso mantém todas as informações de velocidades e estados de movimento do corpo, que irá ser tratada pela próxima classe, a classe World.

4.4 Classe World

Já temos então os objetos e suas representações, e agora podemos trabalhar seus movimentos em conjunto na cena. Esta é a função da classe World (Figura 4.7). Dentro dela, existem alguns parâmetros de caracterização de um mundo, mas principalmente

contém as funções de tratamento de movimento e de resolução de colisão, completando o sistema da simulação.

Nesta seção, aumentaremos o nível de detalhe para apresentar como as funções presentes na classe funcionam e fazem a simulação ocorrer corretamente. Primeiramente, então iremos apresentar os membros e funções da classe e então entrar no detalhe das mais importantes.

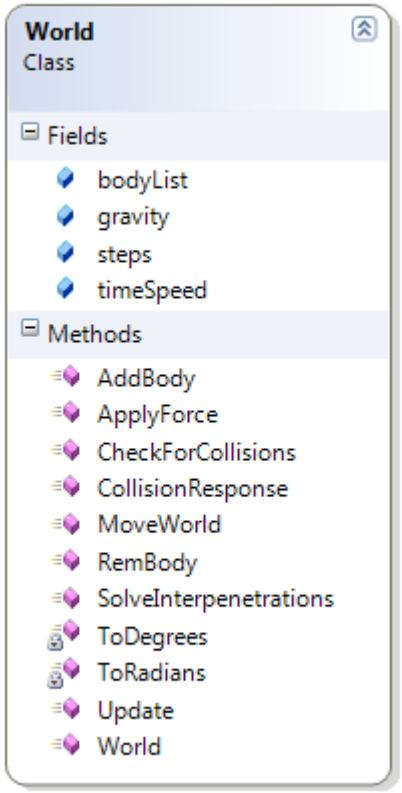

Figura 4.7: Diagrama da classe World

Temos como membros da classe, uma lista dos corpos presentes no espaço, o valor da gravidade aplicado para a simulação, o número máximo de passos para a procura do tempo exato de colisão (steps) e a velocidade do tempo, pois é desejado haver a possibilidade de diminuir a velocidade do tempo com o objetivo de criar um efeito de câmera lenta. Existem também funções mais simples como as de adição e remoção de corpos do sistema, uma função para aplicação de forças a corpos, e funções de conversão de vetores de radianos para ângulos e vice-versa.

Atualizando o que conhecemos do sistema até então, podemos entrar em maior nível de detalhe (Figura 4.8) para o algoritmo apresentado no inicio do capitulo, para tanto, utilizaremos pseudo-código para simplificar o contexto das instruções.

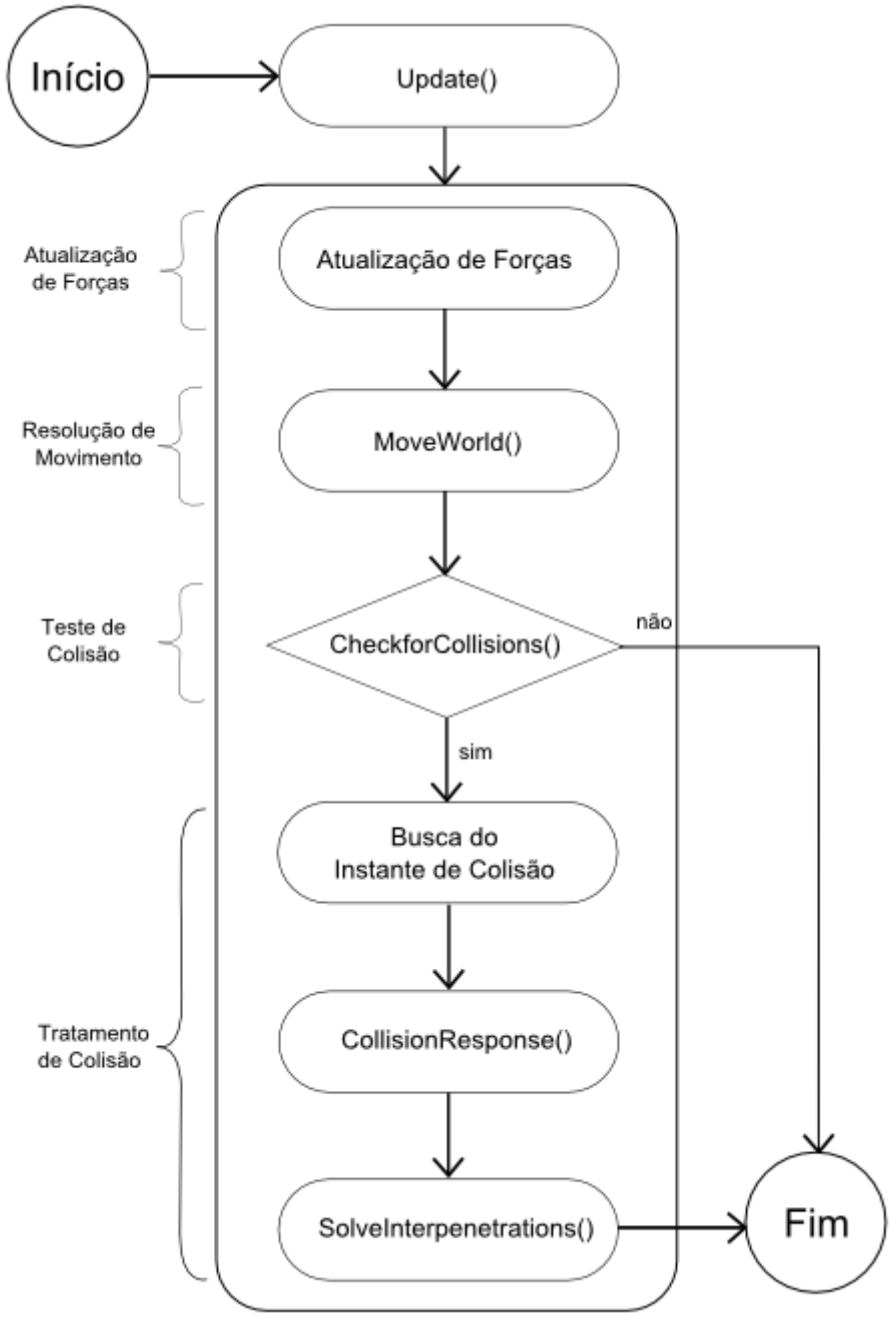

Figura 4.8: Fluxograma em detalhe. A chamada principal, Update(), realiza os passos para a simulação Física, passando pelas etapas na ordem: atualização de forças, resolução de movimento, detecção de colisão e tratamento de colisão.

Para começar, analizemos a função de entrada utilizada pelo usuário, que é a chamada Update(), que deverá ser chamada dentro do laço de atualização do próprio jogo. Sua implementação dispara cada uma das chamadas necessárias para a simulação alterar as posições e orientações dos objetos do jogo.

```
Update(time)
{
      // Atualização de forças
      para cada corpo presente no sistema,
      {
            se (o corpo nao é fixo)
            {
                  CalculaForçaResultante();
                  aceleracaoLinear = forçaResultante / massa;

                  se (o corpo está sobre influencia da gravidade)
                        aceleracaoLinear += aceleracaoDaGravidade;
            }
            // configuração inicial de contato:
            emContato = falso;
      }
```

Figura 4.9: Função Update() parte 1

Neste trecho de código (Figura 4.9), temos a resolução de forças diretamente aplicadas. Primeiramente, iteramos cada corpo existente neste mundo e, caso ele não seja um corpo fixo, calculamos a força resultante de todas as aplicadas. A partir disso, extraímos da teoria Física a fórmula que gera aceleração a partir da relação entre força e massa. Um parâmetro extra é adicionado para o caso de o corpo em questao estar sob influência da força da gravidade deste mundo.

A gravidade é a força de atração entre os corpos, porém essa força varia de acordo com a massa dos corpos envolvidos. Corpos diferentes sobre forças de gravidade diferentes. Existe uma constante de atração dos corpos que remete que a aceleração dessa força é sempre a mesma, e por isso na simulação a gravidade é tratada diretamente como uma aceleração. Para encerrar, um pequenino trecho de código garante que os corpos são considerados livres de contato em um primeiro momento.

A próxima fase é a de Resolução de Movimento, controlada pela função MoveWorld().

```
      // Resolução de movimento
      MoveWorld(time);

      // Teste de Colisão
      CheckForCollisions();
```

Figura 4.10: Função Update() parte 2

Aqui (Figura 4.10) se encontra o núcleo de toda a simulação. É invocada a função MoveWorld(), onde os corpos são movimentados de acordo com suas velocidades e o tempo predisposto. Somente após a movimentação dos corpos é que temos a primeira detecção de colisões chamada pela função CheckForCollisions(), que se retornar um valor positivo, novos testes serão executados para determinar o momento exato da colisão.

```
      // Tratamento de Colisões
      se (existem colisões)
      {
            tempoDoPasso = tempoTotal / máximoDePassos;

           // volta para o início do quadro
            MoveWorld(-tempoTotal);

            // Busca do instante de colisão
            tempoAtual = tempoDoPasso;
            enquanto (tempoAtual <= tempoTotal)
            {
                  // avança no tempo
                  MoveWorld(tempoDoPasso);
                  // procura colisão
                  CheckForCollisions();

                  se (existem colisões)
                  {
                        // volta no tempo
                        MoveWorld(-tempoDoPasso);
                        // aplica a colisão aos corpos
                        CollisionResponse(collisions);
                        // avança no tempo novamente
                        MoveWorld(tempoDoPasso);
                  }
                  tempoAtual += tempoDoPasso;
            }
      }
      // resolve interpenetrações que tenham ficado pendentes
      SolveInterpenetrations();
}
```

Figura 4.11: Função Update() parte 3

Caso não ocorram colisões, o movimento dos corpos é encarado como correto e o procedimento acaba por este quadro. Porém, se ocorreram colisões durante a movimentação deste quadro, é necessário voltar atrás no tempo, e avançar novamente até o momento de colisão; resolver as colisões existentes, e prosseguir até o final do quadro (Figura 4.11). Ao final da função é chamada a função de resolução de interpenetrações que tratará de corpos cujas velocidades geradas por impulso não foram suficientes para separar os objetos.

Tendo conhecimento de como funciona o veio central de simulação, agora podemos analisar com detalhes as funções que foram descritas apenas por sua funcionalidade. Primeiramente temos a função MoveWorld().

```
MoveWorld(tempo)
{
      para cada corpo no sistema
      {
            se (o corpo não é fixo)
            {
                  se (o tempo é positivo)
                  {
                        velocidadeLinear = aceleraçãoLinear * tempo;
                        posicao += velocidadeLinear * tempo;

                        velocidadeAngular = aceleraçãoAngular * tempo;
                        matrizDeRotação *= CriarMatrizDeRotação(velocidadeAngular
* tempo);

                  }
                  else
                  {
                        matrizDeRotação *=
CriarMatrizDeRotação(velocidadeAngular * tempo);
                        velocidadeAngular = aceleraçãoAngular * tempo;

                        posicao += velocidadeLinear * tempo;
                        velocidadeLinear = aceleraçãoLinear * tempo;
                  }
            }
      }
}
```

Figura 4.12: Função MoveWorld()

A função MoveWorld() (Figura 4.12) funciona sobre a premissa de que dado o valor de tempo recebido, deve atualizar as acelerações, velocidades e posições (ou rotações) de todos os corpos móveis, tanto para um avanço no tempo, quanto para uma volta no tempo. É por isso que existe o teste de sinal sobre a variável de tempo, para que a ordem das atualizações esteja correta, dependendo da "direção" do tempo.'

A próxima função chamada pela Update() é a CheckForCollisions().

```
CheckForCollisions() {
      para cada corpo no sistema {
            se (o corpo tem uma geometria de colisão) {
                  para cada outro corpo no sistema {
                        se (o outroCorpo tem uma geometria de colisão) {
                              resultado =
geometriaCorpo.Intersects(geometriaDoOutroCorpo);
                              se (resultado != null) {
                                    resultado.bodyA = corpoAtual;
                                    resultado.bodyB = outroCorpo;
                                    listaDeResultados = resultado;
                              }
                        }
                  }
            }
      }
      return listaDeResultados;
}
```

Figura 4.13: Função CheckForCollisions()

Esta função (Figura 4.13) deve varrer todos os objetos pertencentes ao mundo e testar suas geometrias para eventos de colisão utilizando as funções implementadas pela classe CollisionShape. Em adição ao resultado da existência de colisão, são adicionados os corpos participantes. Ao final da função, é retornada uma lista de colisões que estão ocorrendo no dado instante.

Dentro do laço da função Update(), temos o controle para, caso ocorram colisões, ser descoberto o momento mais próximo da colisão e então serem tratadas as colisões daquele momento. É neste passo que entra a função CollisionResponse(). Como esta função é grande e de maior importância, vamos novamente quebrá-la em pedaços para maior compreensão de suas partes.

```
CollisionResponse() {
     para cada colisao    {
          // coeficiente de restituição da colisão
          // é a média aritmética dos coeficientes dos corpos
          e = (corpoA.e + corpoB.e) / 2.0f;

          se (um corpo é fixo)
               corpo.massa = infinito;    // ou um valor muito grande

          // distancia entre os centros dos corpos e o ponto de colisão
          raioAtéCorpoA = pontoDeColisão - corpoA.centroDeMassa;
          raioAtéCorpoB = pontoDeColisão - corpoB.centroDeMassa;
```

Figura 4.14: Função CollisionResponse() parte 1

No primeiro momento da função, estabelecemos alguns parâmetros iniciais como os corpos envolvidos na colisão (Figura 4.14); o coeficiente de restituição da colisão em si; as massas de corpos fixos que devem ser marcadas como infinitamente positivas, para que a formula do impulso esteja correta contra objetos fixos; e os raios da colisão, que são basicamente as linhas que vão do centro de massa até o ponto de colisão.

```
          momentoDeInerciaA = geometriaA.MomentOfInertia();
          momentoDeInerciaB = geometriaB.MomentOfInertia();

          velocidadeRelativa = corpoA.velocidadeLinear -
corpoB.velocidadeLinear;

          // aplicação da fórmula do impulso
          impulso = - velocidadeRelativa * (e + 1) /
 ((1 / corpoA.massa) + (1 / corpoA.massa) +

DotProduct(normalDaColisao,
     (CrossProduct(
          CrossProduct(raioAtéCorpoA, normalDaColisão) /
               momentoDeInerciaA,
          raioAtéCorpoA))) +

DotProduct(normalDaColisão,
     (CrossProduct(
          CrossProduct(raioAtéCorpoB, normalDaColisão) /
               momentoDeInerciaB,
          raioAtéCorpoB)));
```

Figura 4.15: Função CollisionResponse() parte 2

Separamos também os momentos de inércia de cada corpo (Figura 4.15), que é calculado pelas subclasses de CollisionShapes; e a velocidade relativa entre os corpos envolvidos. Tendo tais variáveis separadas, calculamos a força de impulso baseada na fórmula que apresentamos no capitulo anterior.

Finalmente, temos a aplicação da força de impulso diretamente nos objetos utilizando as fórmulas apresentadas, pois como foi discutido anteriormente, o impulso é uma força que é aplicada em um mínimo instante de tempo, e por isso o que vemos é apenas uma mudança na velocidade, pois foi uma grande aceleração que ocorreu mas não existe mais. Assim, ao invés de ser considerado mais uma força para o sistema de forças tratado no inicio da função Update(), o impulso é aplicado aqui diretamente para já retornar os objetos com suas novas velocidades (Figura 4.16).

```
            // acelerações instantâneas
            acelA = impulso / corpoA.massa;
            acelB = -impulso / corpoB.massa;

            corpoA.velocidadeLinear +=
                  DotProduct(acelA, normalDaColisão) * normalDaColisão;
            corpoB.velocidadeLinear +=
                  DotProduct(acelB, normalDaColisão) * normalDaColisão;

            corpoA.velocidadeAngular =
                  CrossProduct(raioAtéCorpoA, impulso) / momentoDeInerciaA
            corpoB.velocidadeAngular =
                  CrossProduct(raioAtéCorpoB, impulso) / momentoDeInerciaB
```

Figura 4.16: Função CollisionResponse() parte 3

Para encerrar, temos ao final do método, a determinação do estado de contato dos corpos, onde um caso especial é criado para separar a situação de contato da situação de colisão propriamente dita (Figura 4.17).

```
            // contact feedback
            limiarDeRolagem = 3.0f;

            estadoDeRolagem = DotProduct(normalDaColisao,
corpoA.velocidadeLinear);
            if (estadoDeRolagem < limiarDeRolagem)
            {
                  corpoA.emContato = true;
            }

            estadoDeRolagem = DotProduct(normalDaColisao,
corpoB.velocidadeLinear);

            if (estadoDeRolagem < limiarDeRolagem)
            {
                  corpoB.emContato = true;
            }
      }
}
```

Figura 4.17: Função CollisionResponse() parte 4

Assim, temos toda a anatomia do sistema apresentada em vários níveis de detalhe, onde o usuário só necessita utilizar algumas intuitivas funções e passar alguns parâmetros para que a simulação já possa ser lançada com resultados muito agradáveis.

5 AVALIAÇÃO DOS RESULTADOS

Existem vários aspectos que são utilizados para avaliar a qualidade de um motor de Física. Os principais são: a relação custo-benefício entre precisão e tempo de processamento, mas também a facilidade de integração da biblioteca com outras aplicações, visto que o objetivo é que esta seja utilizada por outros usuários.

Existe um motor de Física que poderia ser utilizado pelos usuários da comunidade XNA™ para o Xbox 360™ chamada BulletX. Porém, esta biblioteca se trata de uma re-escrita em C# de uma distribuição do mesmo motor que existe para ambientes Windows™. Seu foco estava voltado diretamente para desempenho, partindo do princípio que usuários que fossem utilizá-la deveriam se habituar com a linguagem de funcionamento da biblioteca.

Ao desenvolver este motor de Física, o maior foco de abordagem é a usabilidade. A principal motivação de sua construção foi a falta de um motor simples e amigável para os usuários que estão começando a desenvolver jogos, e alguns apenas iniciando a programar. Assim, poderemos ver nos testes realizados que o motor em seu estado atual ainda sofre por não ter métodos de otimização de sistemas de Física, que normalmente aceleraram mais o processamento.

5.1 Usabilidade do Sistema

Como foi apresentado no capitulo anterior, a sistemática de implementação do sistema se aproxima mais da teoria Física e da simplicidade de algoritmos iterativos do que do conceito de construção de *software*, o que se espera que seja um facilitador para os usuários-alvo da biblioteca.

Como exemplo, iremos apresentar em linguagem C#, como funciona a integração do sistema de Física a um demo que foi desenvolvido com o intuito de testar a biblioteca em sua fase de desenvolvimento. Este demo também serve como exemplo de como qualquer *software* poderia agregar a simulação de Física com facilidade para o seu sistema.

```
using PhysicsLibrary.Entities;
…
World testWorld;
Body sphereBody;
Body floorBody;
…
protected override void LoadContent()
{
      testWorld = new World();

      // entrada de uma esfera no sistema
      sphereBody = new Body();
      sphereBody.cShape =
            new CollisionSphere(Vector3.Zero, 2.0f * 1.5f);
      sphereBody.position = new Vector3(5.0f, -6.0f, 0.0f);
      testWorld.AddBody(sphereBody);

      // entrada de um plano no sistema
      floorBody = new Body();
      floorBody.cShape =
            new CollisionPlane(Vector3.Up, new Vector3(0.0f, -3.0f, 0.0f));
      floorBody.position = new Vector3(0.0f, -5.0f, 0.0f);
      testWorld.AddBody(floorBody);
}

public void Update(GameTime gameTime)
{
      testWorld.Update(gameTime);
}
```

Figura 5.1: Código de exemplo de implementação

O demo parcialmente codificado na Figura 5.1 estabelece uma cena com um plano, sendo representado por um chão de madeira, e uma esfera, sendo representada por uma bola de vidro. O resultado pode ser visualizado na Figura 5.2.

Figura 5.2: Demo em execução: bola de vidro em repouso sobre o plano de madeira.

Essa clareza pode ser visualizada nas linhas de código dentro do método LoadContent(), onde as propriedades Físicas podem ser determinadas com facilidade

estabelecendo as características do corpo a ser simulado. E depois, no método Update(), temos a execução do motor em si, com apenas umas simples chamada de atualização.

5.2 Desempenho

O motor de Física sem otimizações tem complexidade quadrática no número de elementos presentes na cena, pois compara todos os corpos entre si no momento de verificar a ocorrência de colisões. Existem algumas técnicas para otimizar esta verificação, como por exemplo, a divisão do espaço, ou a limitação dos possíveis corpos que entrariam em contato com um dado corpo. Mas como em um primeiro momento, a maior preocupação foi com a abordagem didática para o publico alvo, o desempenho ficou comprometido.

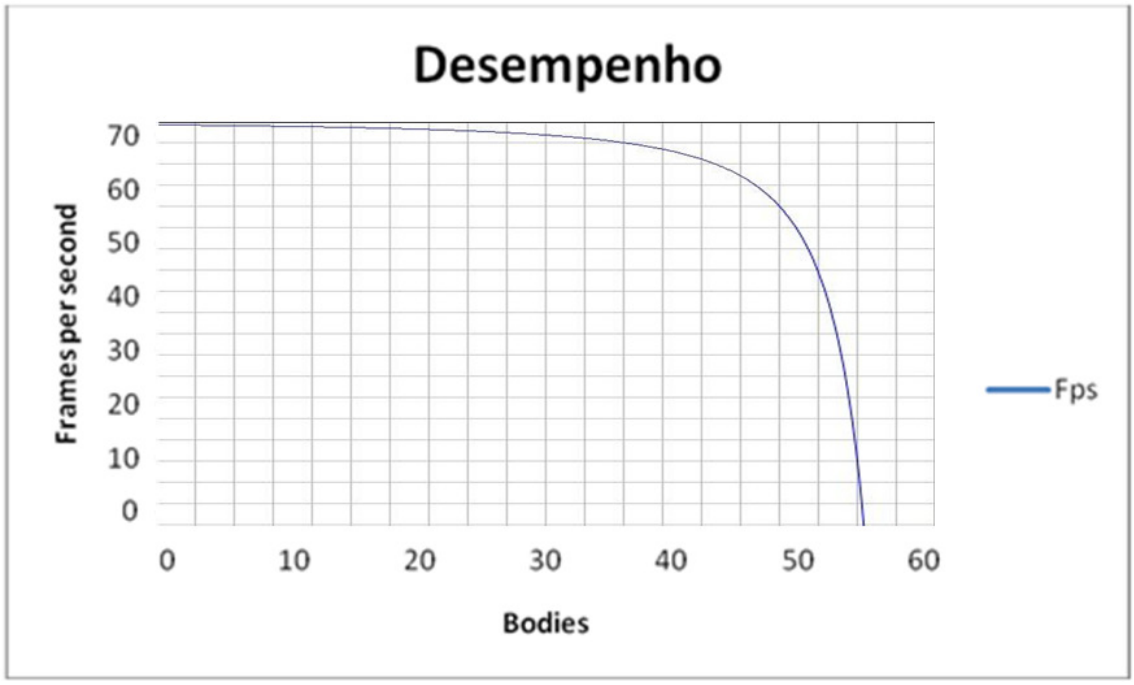

Figura 5.3: Gráfico de desempenho

Ainda assim, temos resultados muito satisfatórios na simulação. Utilizamos um demo desenvolvido paralelamente à biblioteca para os testes, expandindo sua capacidade para poderem ser adicionados corpos ao sistema em tempo real, assim como, ampliando o cenário, utilizando mais planos para montar a estrutura fixa de madeira.

Podemos ver pelas imagens na Figura 5.4, a relação entre o número de quadros por segundo (valor amarelo no canto superior direito) e o número de corpos na simulação (valor branco no canto superior direito). Os passos retratam os pontos mais expressivos de decaimento do desempenho, ilustrado no gráfico da Figura 5.3.

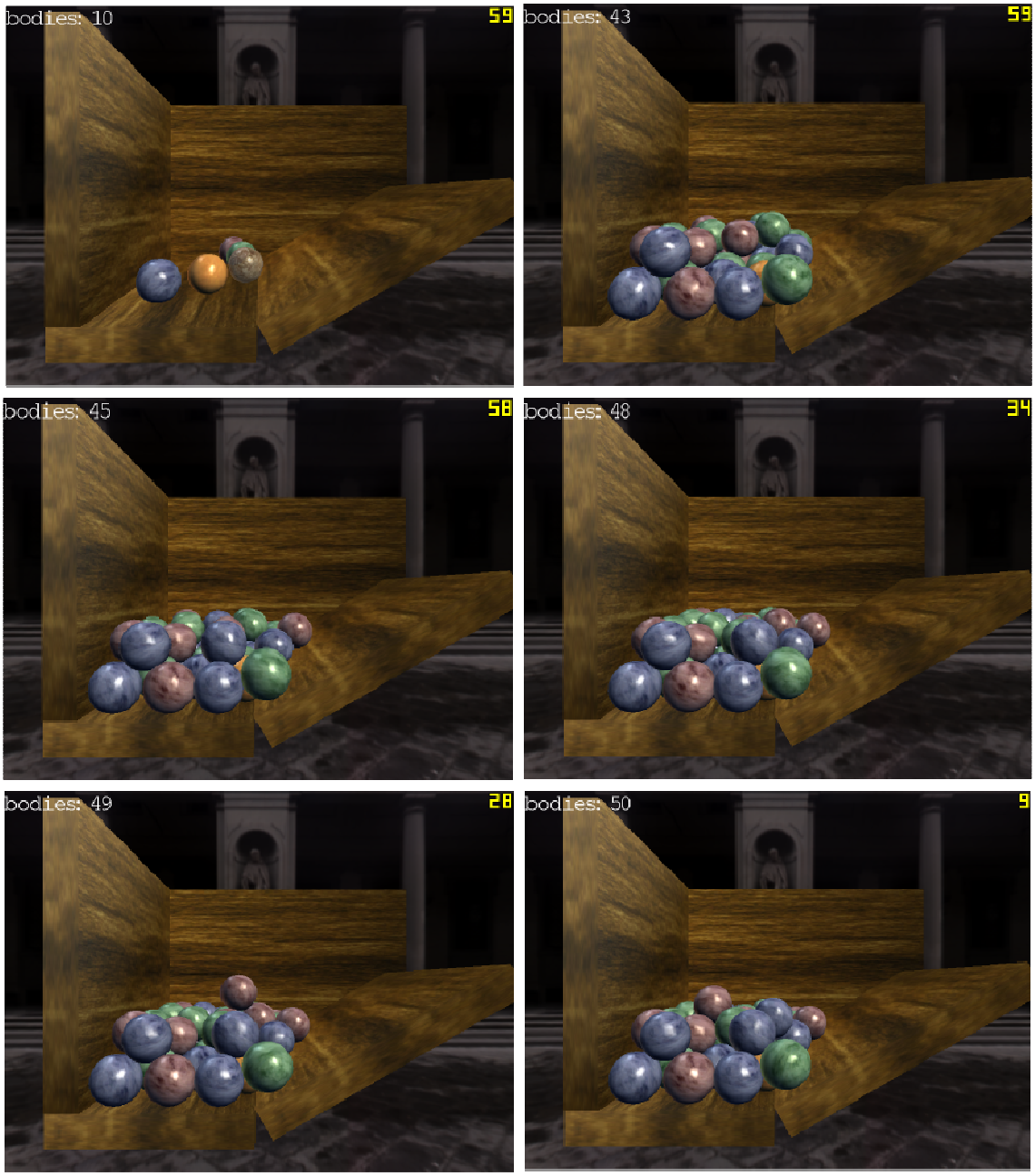

Figura 5.4: Análise de desempenho (corpos x quadros por segundo)

5.3 Objetivos Alcançados

Ao desenvolver a pesquisa e o projeto de implementação é que se teve real conhecimento de como é complexa a aplicação da teoria para o *software*, onde vários detalhes podem passar despercebidos e são abordados de formas diferentes em bibliografias diferentes.

O projeto original previa que nesse primeiro estágio, outras funcionalidades extras fossem agregadas ao sistema, mas à medida que se foi tomando conhecimento de como estava se dando o ritmo de implementação, foi necessário diminuir o alcance inicial do projeto para se limitar apenas a tratar a dinâmica de corpos.

Porém, o conceito de usabilidade aparenta ter sido alcançado, a medida que, como foi apresentado no exemplo, bastam apenas alguns parâmetros e poucas chamadas para que o sistema se encarregue de realizar as simulações sem que o usuário necessite maiores controles.

Já tivemos nosso primeiro usuário real para a biblioteca desenvolvida: um aluno do mestrado da UFRGS que necessitava de uma simulação de colisão baseada em Física.

6 CONCLUSÃO

Neste trabalho apresentamos um motor de Física para XNA™ com foco na usabilidade e em sua utilização no Xbox 360™. Conseguimos bons resultados e um usuário que adotou a biblioteca desenvolvida. Assim é possível afirmar que o trabalho desenvolvido é de interesse para um mercado ativo da comunidade de desenvolvimento de jogos. Cada vez maior, a comunidade de desenvolvedores do Microsoft® XNA™ cresce em número e em nível de conhecimento.

Outro grande aspecto que deve ser destacado foi a experiência em desenvolver o sistema de Física. Primeiramente, a idéia que se tinha de como era estruturado um motor de Física era completamente ingênua e cheia de ilusão. Apenas após "colocar a mão na massa", foi possível reconhecer o nível de detalhe e de realismo representado nos sistemas de simulação de Física. Foi possível dar ainda mais valor aos sistemas comerciais, onde inúmeras técnicas são aplicadas visando otimização e maior realismo.

Finalizando, é possível afirmar que ainda existe trabalho pela frente para que a biblioteca fique mais completa, e seja ainda mais robusta e mais poderosa. Não serão medidos esforços na finalização deste trabalho, cujo fruto é possível de ser a semente para uma carreira profissional no mercado de desenvolvimento de jogos eletrônicos.

6.1 Trabalhos Futuros

Primeiramente, podemos adicionar várias técnicas para otimizar o poder de processamento do motor. Dentre elas pode-se citar a subdivisão espacial, que se caracteriza por separar o universo em sub-espaços distintos para que corpos que não estejam no mesmo sub-espaço não sejam testados desnecessariamente uns contra os outros para detecção de colisões. Outra técnica de otimização de teste de detecção de colisões é a de cálculo de vizinhos possíveis, onde são limitados os corpos que podem entrar em contato com um dado corpo.

Entre outras opções que podem ser adicionadas no futuro, temos o tratamento de corpos articulados, que são basicamente uniões de corpos caracterizados por limitações em seus movimentos dadas exatamente pelas ligações entre eles. Com esta estrutura geométrica é possível construir volumes limitantes complexos compostos de vários volumes primitivos mais simples.

Falando ainda dos volumes limitantes, é possível a adição de novas geometrias para o sistema, complementando o suporte fornecido, hoje em dia, apenas para esferas e planos. Exemplo de tais geometrias seriam cubos, cilindros, etc. Sendo o motor

construído para tratar genericamente corpos arbitrários, novas geometrias se encaixariam dentro do sistema com facilidade, onde os maiores desafios de implementação seriam o cálculo do teste de colisão com as outras geometrias existentes e o cálculo do momento de inércia das novas geometrias.

Outra idéia a ser explorada é o desenvolvimento de um algoritmo para segmentar uma malha 3D em um conjunto de esferas. Desta maneira, poderiam ser representados objetos complexos na visualização, enquanto objetos mais simples seriam utilizados para realização de simulações físicas.

REFERÊNCIAS

MILLINGTON, I. **Game Physics Engine Development**. San Francisco: Morgan Kaufmann, 2007.

CONGER, D. **Physics Modeling for Game Programmers**. Boston: Course Tecnology, 2004.

PALMER, G. **Physics for Game Programmers**. New York: APress, 2005.

BOURG, D. M. **Physics for Game Developers**. Sebastopol: O'Reilly & Associates, Inc., 2002.

LEMOS, R. R. **Animação e tratamento de colisões de corpos rígidos utilizando análise dinâmica**. 1993. 105 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

HECKER, C. Physics, The Next Frontier. **Game Developer Magazine**, Out. 1996. Disponível em: <http://chrishecker.com/Rigid_Body_Dynamics>. Acesso em: nov. 2008.

HECKER, C. Physics, Part 2: Angular Effects. **Game Developer Magazine**, Dez. 1996. Disponível em: <http://chrishecker.com/Rigid_Body_Dynamics>. Acesso em: nov. 2008.

HECKER, C. Physics, Part 3: Collision Response. **Game Developer Magazine**, Fev. 1997. Disponível em: <http://chrishecker.com/Rigid_Body_Dynamics>. Acesso em: nov. 2008.

HECKER, C. Physics, Part 4: The Third Dimension. **Game Developer Magazine**, Jun. 1997. Disponível em: <http://chrishecker.com/Rigid_Body_Dynamics>. Acesso em: nov. 2008.

GAMESINDUSTRY. Disponível em: < http://www.gamesindustry.biz/>. Acesso em: nov. 2008.

CREATOR`S CLUB. Disponível em: < http://creators.xna.com/>. Acesso em: nov. 2008.

GAMASUTRA. Disponível em: < http://www.gamasutra.com/>. Acesso em: nov. 2008.

ALFRED'S WORLD. Disponível em: < https://sourceforge.net/projects/alfredsworld/>. Acesso em: nov. 2008.

APÊNDICE A CÓDIGO FONTE DA BIBLIOTECA

CollisionShape.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;

namespace PhysicsLibrary.Entities
{
    public class CollisionShape
    {
        protected Vector3 _position;

        public virtual Vector3 position
        {
            set
            {
                _position = value;
            }
            get
            {
                return _position;
            }
        }

        public CollisionResult SphereSphereCollision(CollisionSphere
cs1, CollisionSphere cs2)
        {
            float dist = Vector3.Distance(cs1.center, cs2.center);
            if (dist <= cs1.radius + cs2.radius)
            {
                Vector3 normal = cs1.center - cs2.center;
                normal.Normalize();
                Vector3 point = cs2.center + normal * cs2.radius;

                return new CollisionResult(point, normal, cs1.radius +
cs2.radius - dist);
            }
            else
                return null;
        }

        public CollisionResult SpherePlaneCollision(CollisionSphere
cs, CollisionPlane plane)
        {

float d = - Vector3.Dot(plane.normal, plane.offsetD);
            float dist = Vector3.Dot(plane.normal, cs.center) + d;
            // dist = ax + by + cz + d
            if (dist <= cs.radius)
            {
```

```
                Vector3 point = new Vector3();
                point = cs.center - plane.normal * dist;

                return new CollisionResult(point, plane.normal,
cs.radius - dist);
            }
            else
                return null;
        }

        public virtual CollisionResult Intersects(CollisionShape cs)
        {
            return null;
        }

        public virtual float MomentOfInertia(float mass, Vector3 rad)
        {
            return 1.0f;
        }

        public virtual string GetShapeType()
        {
            return "";
        }
    }

    public class CollisionSphere : CollisionShape
    {
        public Vector3 center;
        public float radius;

        public override Vector3 position
        {
            set
            {
                _position = value;
                center = value;
            }
            get
            {
                return _position;
            }
        }

        public CollisionSphere()
        {
        }

        public CollisionSphere(Vector3 sCenter, float sRadius)
        {
            center = sCenter;
            radius = sRadius;
        }

        public CollisionResult SphereSphereCollision(CollisionSphere
cs)
        {
            return SphereSphereCollision(this, cs);
        }
```

```
        public CollisionResult SpherePlaneCollision(CollisionPlane cp)
        {
            return SpherePlaneCollision(this, cp);
        }

        public override CollisionResult Intersects(CollisionShape cs)
        {
            switch (cs.GetType().Name)
            {
                case "CollisionSphere":
                    return SphereSphereCollision((CollisionSphere)cs);

                case "CollisionPlane":
                    return SpherePlaneCollision((CollisionPlane)cs);

                default:
                    return null;
            }
        }

        public override float MomentOfInertia(float mass, Vector3 rad)
        {
            return (2.0f / 5.0f) * mass * rad.LengthSquared();
        }

        public override string GetShapeType()
        {
            return "sphere";
        }
    }

    public class CollisionPlane : CollisionShape
    {
        public Vector3 normal;
        public Vector3 offsetD;

        public CollisionPlane()
        {
        }

        public CollisionPlane(Vector3 pNormal, Vector3 pOffsetD)
        {
            pNormal.Normalize();
            normal = pNormal;
            offsetD = pOffsetD;
        }

        public CollisionResult SpherePlaneCollision(CollisionSphere
cs)
        {
            return SpherePlaneCollision(cs, this);
        }

        public override CollisionResult Intersects(CollisionShape cs)
        {
            switch (cs.GetType().Name)
            {
                case "CollisionSphere":
                    return SpherePlaneCollision((CollisionSphere)cs);
```

```
                default:
                    return null;
            }
        }

        public override float MomentOfInertia(float mass, Vector3 rad)
        {
            return (2.0f / 5.0f) * mass * rad.LengthSquared();
        }

        public override string GetShapeType()
        {
            return "plane";
        }
    }

    public class CollisionResult
    {
        public Vector3 collisionPoint;
        public Vector3 collisionNormal;

        public Body bodyA;
        public Body bodyB;

        public float penetration;

        public CollisionResult(Vector3 colPoint, Vector3 colNormal,
float pen)
        {
            collisionPoint = colPoint;
            collisionNormal = colNormal;
            penetration = pen;
        }
    }
}
```

Body.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace PhysicsLibrary.Entities
{
    public class Body
    {
        private Vector3 _position;

        public Vector3 position
        {
            set
            {
                _position = value;
                massCenter = value;
                if (cShape != null) cShape.position = value;
            }
            get
```

```
            {
                return _position;
            }
        }

        public Matrix rotationMatrix;

        public Vector3 linearVelocity;
        public Vector3 angularVelocity;

        public Vector3 linearAcceleration;
        public Vector3 angularAcceleration;

        public float mass;
        public Vector3 massCenter;

        public List<Vector3> forceList;
        public Vector3 resultantForce;

        public CollisionShape cShape;
        public Model bodyModel;

        public float elasticCoefficient;
        public bool isFixed;
        public bool inContact;
        public bool underGravity;

        public Body()
        {
            forceList = new List<Vector3>();
            mass = 1.0f;
            elasticCoefficient = 1.0f;
            isFixed = false;
            inContact = false;
            underGravity = true;

            linearVelocity = Vector3.Zero;
            linearAcceleration = Vector3.Zero;
            angularVelocity = Vector3.Zero;
            angularAcceleration = Vector3.Zero;

            position = Vector3.Zero;
            rotationAngles = Vector3.Zero;
            resultantForce = Vector3.Zero;

            rotationMatrix = Matrix.Identity;
        }

        public Body(Model model, CollisionShape cs)
        {
            forceList = new List<Vector3>();
            mass = 1.0f;
            elasticCoefficient = 1.0f;
            isFixed = false;
            inContact = false;
            underGravity = true;

            linearVelocity = Vector3.Zero;
            linearAcceleration = Vector3.Zero;
            angularVelocity = Vector3.Zero;
```

```
            angularAcceleration = Vector3.Zero;

            position = Vector3.Zero;
            rotationAngles = Vector3.Zero;
            resultantForce = Vector3.Zero;

            bodyModel = model;
            cShape = cs;

            rotationMatrix = Matrix.Identity;
        }
    }
}
```

World.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;

namespace PhysicsLibrary.Entities
{
    public class World
    {
        public List<Body> bodyList;
        float timeSpeed;
        int steps;

        Vector3 gravity;

        public World()
        {
            bodyList = new List<Body>();
            timeSpeed = 1.0f;
            steps = 16;

            gravity = new Vector3(0.0f, -49.0f, 0.0f);
        }

        public void AddBody(Body nBody)
        {
            bodyList.Add(nBody);
        }

        public void RemBody(Body nBody)
        {
            bodyList.Remove(nBody);
        }

        public void ApplyForce(Body nBody, Vector3 force)
        {
            nBody.forceList.Add(force);
        }

        private Vector3 ToDegrees(Vector3 vec)
        {
            return new Vector3(MathHelper.ToDegrees(vec.X),
MathHelper.ToDegrees(vec.Y), MathHelper.ToDegrees(vec.Z));
```

```
        }

        private Vector3 ToRadians(Vector3 vec)
        {
            return new Vector3(MathHelper.ToRadians(vec.X),
MathHelper.ToRadians(vec.Y), MathHelper.ToRadians(vec.Z));
        }

        public void MoveWorld(List<Body> nBodyList, float time)
        {
            foreach (Body nBody in nBodyList)
            {
                if (!nBody.isFixed)
                {
                    if (time > 0.0f)
                    {
                        nBody.linearVelocity +=
nBody.linearAcceleration * time;
                        nBody.position += nBody.linearVelocity * time;

                        nBody.angularVelocity +=
nBody.angularAcceleration * time;
                        nBody.rotationMatrix *=
                            Matrix.CreateFromYawPitchRoll(
MathHelper.ToRadians(nBody.angularVelocity.Y * time),

MathHelper.ToRadians(nBody.angularVelocity.X * time),

MathHelper.ToRadians(nBody.angularVelocity.Z * time));
                    }
                    else
                    {
                        nBody.rotationMatrix *=
                             Matrix.CreateFromYawPitchRoll(
MathHelper.ToRadians(nBody.angularVelocity.Y * time),

MathHelper.ToRadians(nBody.angularVelocity.X * time),

MathHelper.ToRadians(nBody.angularVelocity.Z * time));
                        nBody.angularVelocity +=
nBody.angularAcceleration * time;

                        nBody.position += nBody.linearVelocity * time;
                        nBody.linearVelocity +=
nBody.linearAcceleration * time;
                    }
                }
            }
        }

        public List<CollisionResult> CheckForCollisions(List<Body>
nBodyList)
        {
            List<Body> chkBodies = new List<Body>();
            List<CollisionResult> results = new
List<CollisionResult>();
            CollisionResult result = null;

            foreach (Body nBody in nBodyList)
            {
```

```
                if (nBody.cShape != null)
                {
                    foreach (Body other in nBodyList)
                    {
                        if (other != nBody &&
!chkBodies.Contains(other) && other.cShape != null)
                        {
                            result =
nBody.cShape.Intersects(other.cShape);
                            if (result != null)
                            {
                                result.bodyA = nBody;
                                result.bodyB = other;
                                results.Add(result);
                            }
                        }
                    }
                }
                chkBodies.Add(nBody);
            }
            return results;
        }

        public void SolveInterpenetrations(List<Body> nBodyList)
        {
            List<CollisionResult> collisions =
CheckForCollisions(bodyList);
            if (collisions.Count > 0)
            {
                foreach (CollisionResult result in collisions)
                {
                    if (!result.bodyA.isFixed &&
!result.bodyB.isFixed)
                    {
                        result.bodyA.position +=
result.collisionNormal * result.penetration / 2;
                        result.bodyB.position += -
result.collisionNormal * result.penetration / 2;
                    }
                    else if (result.bodyA.isFixed &&
!result.bodyB.isFixed)
                    {
                        result.bodyB.position +=
result.collisionNormal * result.penetration;
                    }
                    else if (!result.bodyA.isFixed &&
result.bodyB.isFixed)
                    {
                        result.bodyA.position +=
result.collisionNormal * result.penetration;
                    }
                }
            }
        }

        public void Update(GameTime gameTime)
        {
            float timeLimit =
(float)gameTime.ElapsedGameTime.TotalSeconds * timeSpeed;
```

```
            foreach (Body nBody in bodyList)
            {
                // forces
                if (!nBody.isFixed)
                {
                    nBody.resultantForce = Vector3.Zero;
                    foreach (Vector3 force in nBody.forceList)
                        nBody.resultantForce += force;
                    nBody.forceList.Clear();

                    nBody.linearAcceleration = nBody.resultantForce /
nBody.mass;
                    if (nBody.underGravity)
                        nBody.linearAcceleration += gravity;
                }

                // starting contact configuration:
                nBody.inContact = false;
            }

            // update world
            MoveWorld(bodyList, timeLimit);

            List<CollisionResult> collisions =
CheckForCollisions(bodyList);

            if (collisions.Count > 0)
            {
                // if there was a collision, the exact collision time
must be found.
                float stepTime = timeLimit / steps;

                // go back in time
                MoveWorld(bodyList, -timeLimit);

                // find exact collision time
                float thisTime = stepTime;
                while (thisTime <= timeLimit && timeLimit != 0.0f)
                {
                    // clear collisions list
                    collisions.Clear();
                    // update world one stepTime
                    MoveWorld(bodyList, stepTime);
                    // test for collisions
                    collisions = CheckForCollisions(bodyList);

                    if (collisions.Count > 0)
                    {
                        // go back one stepTime and do
collisionResponse
                        MoveWorld(bodyList, -stepTime);
                        // collisionResponse (change directions)
                        CollisionResponse(collisions);
                        // advance one stepTime again
                        MoveWorld(bodyList, stepTime);
                    }
                    thisTime += stepTime;
                }
                SolveInterpenetrations(bodyList);
            }
```

```
        }

        public void CollisionResponse(List<CollisionResult> results)
        {
            foreach (CollisionResult result in results)
            {
                Body bodyA = result.bodyA;
                Body bodyB = result.bodyB;
                Vector3 normal =
Vector3.Normalize(result.collisionNormal);
                float e = (bodyA.elasticCoefficient +
bodyB.elasticCoefficient) / 2.0f;

                if (bodyA.isFixed)
                    bodyA.mass = float.PositiveInfinity;
                if (bodyB.isFixed)
                    bodyB.mass = float.PositiveInfinity;

                Vector3 radiusA = result.collisionPoint -
bodyA.massCenter;
                Vector3 radiusB = result.collisionPoint -
bodyB.massCenter;

                float momentOfInertiaA =
bodyA.cShape.MomentOfInertia(bodyA.mass, radiusA);
                float momentOfInertiaB =
bodyB.cShape.MomentOfInertia(bodyB.mass, radiusB);

                Vector3 relativeVelocity = bodyA.linearVelocity -
bodyB.linearVelocity;

                Vector3 impulseForce = -relativeVelocity * (e + 1) /
                    ((1 / bodyA.mass) + (1 / bodyB.mass) +
                    Vector3.Dot(normal,
(Vector3.Cross(Vector3.Cross(radiusA, normal) / momentOfInertiaA,
radiusA))) +
                    Vector3.Dot(normal,
(Vector3.Cross(Vector3.Cross(radiusB, normal) / momentOfInertiaB,
radiusB))));

                Vector3 acelA = impulseForce / bodyA.mass;
                Vector3 acelB = -impulseForce / bodyB.mass;

                bodyA.linearVelocity += Vector3.Dot(acelA, normal) *
normal;
                bodyB.linearVelocity += Vector3.Dot(acelB, normal) *
normal;

                bodyA.angularVelocity = Vector3.Cross(radiusA,
impulseForce) / momentOfInertiaA;
                bodyB.angularVelocity = Vector3.Cross(radiusB, -
impulseForce) / momentOfInertiaB;

                // contact feedback
                float rollingParameter = 3.0f;

                float rollingStatus = Vector3.Dot(normal,
bodyA.linearVelocity);
                if (rollingStatus < rollingParameter)
                {
```

```
                    bodyA.inContact = true;
                }

                rollingStatus = Vector3.Dot(normal,
bodyB.linearVelocity);
                if (rollingStatus < rollingParameter)
                {
                    bodyB.inContact = true;
                }
            }
        }
    }
}
```

APÊNDICE B CÓDIGO FONTE DO *SOFTWARE* DEMO

Game1.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
using PhysicsLibrary.Entities;

namespace PhysicsDemo
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        Model sphere;
        Model sphereWood;
        Model sphereGlass;
        Model sphereMetal;
        Model sphereMarpleR;
        Model sphereMarpleG;
        Model sphereMarpleB;
        Model floor;

        World testWorld;
        Body sphereBody;
        Body sphereBody2;
        Body sphereBody3;
        Body sphereBody4;

        Body sphereBody41;
        Body sphereBody42;

        List<Body> sphereList;

        Body floorBody;
        Body floorBody2;
        Body floorBody3;
        Body floorBody4;
        Body floorBody5;
```

```
        SpriteFont Font1;
        Vector2 FontPos;

        Vector3 camPosition;
        Matrix projectionMatrix;
        Matrix viewMatrix;

        Model SkySphere;
        Effect SkySphereEffect;

        KeyboardState oldState;
        int camTarget;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        protected override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);

            camPosition = new Vector3(0.0f, 0.0f, 30.0f);
            float aspectRatio =
(float)graphics.GraphicsDevice.Viewport.Width /
(float)graphics.GraphicsDevice.Viewport.Height;
            projectionMatrix =
Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(60.0f),
aspectRatio, 1.0f, 10000.0f);
            viewMatrix = Matrix.CreateLookAt(camPosition,
Vector3.Zero, Vector3.Up);
            camTarget = 2;

            SkySphereEffect = Content.Load<Effect>("SkySphere");
            TextureCube SkyboxTexture =
Content.Load<TextureCube>("uffizi_cross");
            SkySphere = Content.Load<Model>("SphereHighPoly");


SkySphereEffect.Parameters["ViewMatrix"].SetValue(viewMatrix);

SkySphereEffect.Parameters["ProjectionMatrix"].SetValue(projectionMatr
ix);

SkySphereEffect.Parameters["SkyboxTexture"].SetValue(SkyboxTexture);
            // Set the Skysphere Effect to each part of the Skysphere
model
            foreach (ModelMesh mesh in SkySphere.Meshes)
            {
                foreach (ModelMeshPart part in mesh.MeshParts)
                {
                    part.Effect = SkySphereEffect;
                }
            }
```

```
            Font1 = Content.Load<SpriteFont>("Courier New");
            FontPos = Vector2.Zero;

            testWorld = new World();

            sphere = Content.Load<Model>("sphereMarple");
            sphereWood = Content.Load<Model>("sphereXNA");
            sphereGlass = Content.Load<Model>("sphereGlass");
            sphereMetal = Content.Load<Model>("sphereMetal");
            sphereMarpleR = Content.Load<Model>("sphereMarpleR");
            sphereMarpleG = Content.Load<Model>("sphereMarpleG");
            sphereMarpleB = Content.Load<Model>("sphereMarpleB");
            floor = Content.Load<Model>("woodbox2");
            Texture2D texture = Content.Load<Texture2D>("Wood");

            sphereList = new List<Body>();

            sphereBody = new Body();
            sphereBody.bodyModel = sphereMarpleB;
            sphereBody.cShape = new CollisionSphere(Vector3.Zero, 2.0f
* 1.5f);
            sphereBody.mass = 2000.0f;
            sphereBody.position = new Vector3(5.0f, -6.0f, 0.0f);
            sphereBody.elasticCoefficient = 0.5f;
            testWorld.AddBody(sphereBody);

            sphereBody2 = new Body();
            sphereBody2.bodyModel = sphereMarpleR;
            sphereBody2.cShape = new CollisionSphere(Vector3.Zero,
2.0f * 1.5f);
            sphereBody2.mass = 2000.0f;
            sphereBody2.position = new Vector3(-4.0f, 6.0f, 0.0f);
            sphereBody2.elasticCoefficient = 0.5f;
            testWorld.AddBody(sphereBody2);

            sphereBody3 = new Body();
            sphereBody3.bodyModel = sphereGlass;
            sphereBody3.cShape = new CollisionSphere(new
Vector3(11.0f, 10.0f, 0.0f), 2.0f * 1.5f);
            sphereBody3.mass = 1000.0f;
            sphereBody3.position = new Vector3(11.0f, 10.0f, 0.0f);
            sphereBody3.elasticCoefficient = 0.8f;
            testWorld.AddBody(sphereBody3);

            sphereBody4 = new Body();
            sphereBody4.bodyModel = sphereMarpleG;
            sphereBody4.cShape = new CollisionSphere(new Vector3(9.0f,
5.0f, 0.0f), 2.0f * 1.5f);
            sphereBody4.mass = 2000.0f;
            sphereBody4.position = new Vector3(9.0f, 5.0f, 0.0f);
            sphereBody4.elasticCoefficient = 0.5f;
            testWorld.AddBody(sphereBody4);

            sphereBody41 = new Body();
            sphereBody41.bodyModel = sphereMarpleB;
            sphereBody41.cShape = new CollisionSphere(new
Vector3(9.0f, 5.0f, 0.0f), 4.0f);
            sphereBody41.mass = 500.0f;
            sphereBody41.position = new Vector3(9.0f, 8.0f, -2.0f);
```

```
            sphereBody41.elasticCoefficient = 0.8f;
            testWorld.AddBody(sphereBody41);

            sphereBody42 = new Body();
            sphereBody42.bodyModel = sphereWood;
            sphereBody42.cShape = new CollisionSphere(new
Vector3(9.0f, 5.0f, 0.0f), 1.875f * 1.5f);
            sphereBody42.mass = 1000.0f;
            sphereBody42.position = new Vector3(7.0f, 12.0f, 1.0f);
            sphereBody42.elasticCoefficient = 0.7f;
            testWorld.AddBody(sphereBody42);

            floor = Content.Load<Model>("woodbox2");

            floorBody = new Body();
            floorBody.bodyModel = floor;
            floorBody.cShape = new CollisionPlane(new Vector3(-0.5f,
(float)(Math.Sqrt(3)/2), 0.0f), new Vector3(7.0f, -9.0f, 0.0f));
            floorBody.isFixed = true;
            floorBody.elasticCoefficient = 0.8f;
            testWorld.AddBody(floorBody);
            floorBody.position = new Vector3(7.6f, -11.0f, 0.0f);

            floorBody2 = new Body();
            floorBody2.bodyModel = floor;
            floorBody2.cShape = new CollisionPlane(new Vector3(1.0f,
0.0f, 0.0f), new Vector3(-28.0f, 0.0f, 0.0f));
            floorBody2.isFixed = true;
            floorBody2.elasticCoefficient = 0.8f;
            testWorld.AddBody(floorBody2);
            floorBody2.position = new Vector3(-30.0f, 0.0f, 0.0f);

            floorBody3 = new Body();
            floorBody3.bodyModel = floor;
            floorBody3.cShape = new CollisionPlane(new Vector3(0.0f,
1.0f, 0.0f), new Vector3(-20.0f, -18.0f, 0.0f));
            floorBody3.isFixed = true;
            floorBody3.elasticCoefficient = 0.8f;
            testWorld.AddBody(floorBody3);
            floorBody3.position = new Vector3(-19.95f, -20.0f, 0.0f);

            floorBody4 = new Body();
            floorBody4.bodyModel = floor;
            floorBody4.cShape = new CollisionPlane(new Vector3(0.0f,
0.0f, 1.0f), new Vector3(-10.0f, 0.0f, -19.0f));
            floorBody4.isFixed = true;
            floorBody4.elasticCoefficient = 0.8f;
            testWorld.AddBody(floorBody4);
            floorBody4.position = new Vector3(-10.0f, 0.0f, -20.0f);

            floorBody5 = new Body();
            floorBody5.bodyModel = floor;
            floorBody5.cShape = new CollisionPlane(new Vector3(0.0f,
0.0f, -1.0f), new Vector3(-10.0f, 0.0f, 19.95f));
            floorBody5.isFixed = true;
            floorBody5.elasticCoefficient = 0.8f;
            testWorld.AddBody(floorBody5);
            floorBody5.position = new Vector3(-19.95f, -20.0f, 0.0f);

        }
```

```
        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed ||
                Keyboard.GetState().IsKeyDown(Keys.Escape))
                this.Exit();

            float linearStep = 0.5f;

            if (Keyboard.GetState().IsKeyDown(Keys.Space))
sphereBody.linearVelocity.Y += 1.0f;

            if (Keyboard.GetState().IsKeyDown(Keys.R))
sphereBody.linearVelocity.Y -= linearStep;
            if (Keyboard.GetState().IsKeyDown(Keys.F))
sphereBody.linearVelocity.Y += linearStep;
            if (Keyboard.GetState().IsKeyDown(Keys.W))
sphereBody.linearVelocity.Z -= linearStep;
            if (Keyboard.GetState().IsKeyDown(Keys.S))
sphereBody.linearVelocity.Z += linearStep;
            if (Keyboard.GetState().IsKeyDown(Keys.D))
sphereBody.linearVelocity.X += linearStep;
            if (Keyboard.GetState().IsKeyDown(Keys.A))
sphereBody.linearVelocity.X -= linearStep;

            if (Keyboard.GetState().IsKeyDown(Keys.F1)) camTarget = 1;
            if (Keyboard.GetState().IsKeyDown(Keys.F2)) camTarget = 2;
            if (Keyboard.GetState().IsKeyDown(Keys.F3)) camTarget = 3;
            if (Keyboard.GetState().IsKeyDown(Keys.F4)) camTarget = 4;

            switch (camTarget)
            {
                case 1:
                    viewMatrix = Matrix.CreateLookAt(camPosition,
sphereBody.position, Vector3.Up);
                    break;
                case 2:
                    viewMatrix = Matrix.CreateLookAt(new Vector3(-
7.5f, -3.0f, 57.5f), new Vector3(-7.5f, -3.0f, 0.0f), Vector3.Up);
                    break;
                case 3:
                    viewMatrix = Matrix.CreateLookAt(new Vector3(7.5f,
15.0f, 50.0f), new Vector3(-7.5f, -5.0f, 0.0f), Vector3.Up);
                    break;
                case 4:
                    viewMatrix = Matrix.CreateLookAt(new Vector3(-
15.0f, -15.0f, 20.0f), new Vector3(-15.0f, -15.0f, -100.0f),
Vector3.Up);
                    break;
            }

            // sphere generator
            if (Keyboard.GetState().IsKeyDown(Keys.Add) &&
!oldState.IsKeyDown(Keys.Add))
```

```
            {
                Body newSphere = new Body();
                newSphere.bodyModel = sphereWood;
                newSphere.cShape = new CollisionSphere(new
Vector3(9.0f, 5.0f, 0.0f), 1.875f);
                newSphere.mass = 500.0f;
                newSphere.position = new Vector3(9.0f, 5.0f, 0.0f);
                newSphere.elasticCoefficient = 0.8f;
                sphereList.Add(newSphere);
                testWorld.AddBody(sphereList[sphereList.Count - 1]);
            }

            testWorld.Update(gameTime);

            oldState = Keyboard.GetState();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

            graphics.PreferMultiSampling = true;

            // Set the View and Projection matrix for the effect

SkySphereEffect.Parameters["ViewMatrix"].SetValue(viewMatrix);

SkySphereEffect.Parameters["ProjectionMatrix"].SetValue(projectionMatr
ix);
            // Draw the sphere model that the effect projects onto
            foreach (ModelMesh mesh in SkySphere.Meshes)
            {
                mesh.Draw();
            }

            // Undo the renderstate settings from the shader
            graphics.GraphicsDevice.RenderState.CullMode =
CullMode.CullCounterClockwiseFace;
            graphics.GraphicsDevice.RenderState.DepthBufferWriteEnable
= true;

            graphics.GraphicsDevice.RenderState.DepthBufferEnable =
true;

            Vector3 spScale = new Vector3(0.17f, 0.17f, 0.17f) * 1.5f;
            DrawBody(floorBody2, new Vector3(9.0f, 1.0f, 10.0f) /
4.0f, new Vector3(0.0f, 0.0f, 90.0f), "wood");
            DrawBody(floorBody, new Vector3(8.4f, 1.0f, 9.9f) / 4.0f,
new Vector3(0.0f, 0.0f, 30.0f), "wood");
            DrawBody(floorBody3, new Vector3(6.0f, 1.0f, 10.0f) /
4.0f, new Vector3(0.0f, 0.0f, 0.0f), "wood");
            DrawBody(floorBody4, new Vector3(9.0f, 1.0f, 18.0f) /
4.0f, new Vector3(90.0f, 90.0f, 0.0f), "wood");


            DrawBody(sphereBody, spScale, Vector3.Zero, "marble");
            DrawBody(sphereBody2, spScale, Vector3.Zero, "marble");
            DrawBody(sphereBody3, spScale, Vector3.Zero, "glass");
```

```
            DrawBody(sphereBody4, spScale, Vector3.Zero, "");

            DrawBody(sphereBody42, spScale, Vector3.Zero, "");

            foreach (Body newSphere in sphereList)
            {
                DrawBody(newSphere, spScale, Vector3.Zero, "");
            }

            spriteBatch.Begin();

            string output = "ElapsedGameTime = " +
gameTime.ElapsedGameTime.TotalSeconds + "\n";
            output += "ElapsedRealTime = " +
gameTime.ElapsedRealTime.TotalSeconds + "\n";
            output += "sphereBody1 position = " + sphereBody.position
+ "\n";
            output += "sphereBody1 linearAcceleration = " +
sphereBody.linearAcceleration + "\n";
            output += "sphereBody1 linearVelocity = " +
sphereBody.linearVelocity + "\n";
            output += "sphereBody1 linearVelocity.Length = " +
sphereBody.linearVelocity.Length() + "\n";
            output += "sphereBody1 angularVelocity = " +
sphereBody.angularVelocity + "\n";
            output += "sphereBody1 angularAcceleration = " +
sphereBody.angularAcceleration + "\n";
            output += "sphereBody1 rotationAngles = " +
sphereBody.rotationAngles + "\n";

spriteBatch.DrawString(Font1, output, FontPos, Color.White, 0,
Vector2.Zero, 0.7f, SpriteEffects.None, 0.5f);
            spriteBatch.End();

            base.Draw(gameTime);
        }

        private void DrawBody(Body b, Vector3 scale, Vector3 rotate,
string material)
        {
            Model m = new Model();
            m = b.bodyModel;
            Matrix[] transforms = new Matrix[m.Bones.Count];
            m.CopyAbsoluteBoneTransformsTo(transforms);

            foreach (ModelMesh mesh in m.Meshes)
            {
                foreach (BasicEffect effect in mesh.Effects)
                {
                    switch(material)
                    {
                        case "marble":
                            {
                                effect.EnableDefaultLighting();

                                break;
                            }
                        case "metal":
                            {
                                effect.EnableDefaultLighting();
```

```
                                    effect.DiffuseColor = new
Vector3(0.3f, 0.3f, 0.3f);
                                    effect.AmbientLightColor = new
Vector3(0.5f, 0.5f, 0.5f);
                                    effect.SpecularColor = new
Vector3(1.0f, 1.0f, 1.0f);
                                    effect.SpecularPower = 20.0f;
                                    effect.PreferPerPixelLighting = true;

                                    break;
                                }
                            case "glass":
                                {
                                    effect.EnableDefaultLighting();
                                    effect.Alpha = 0.8f;
                                    effect.DiffuseColor = new
Vector3(0.7f, 0.7f, 0.7f);
                                    effect.AmbientLightColor = new
Vector3(0.5f, 0.5f, 0.5f);
                                    effect.SpecularColor = new
Vector3(1.0f, 1.0f, 1.0f);
                                    effect.PreferPerPixelLighting = true;

                                    break;
                                }
                            case "wood":
                                {
                                    effect.EnableDefaultLighting();
                                    effect.DiffuseColor = new
Vector3(0.5f, 0.5f, 0.5f);
                                    effect.AmbientLightColor = new
Vector3(0.8f, 0.8f, 0.8f);
                                    effect.SpecularColor = new
Vector3(0.3f, 0.3f, 0.3f);
                                    effect.SpecularPower = 5.0f;
                                    effect.DirectionalLight0.Enabled =
true;
                                    effect.DirectionalLight0.DiffuseColor
= Vector3.One;
                                    effect.DirectionalLight0.Direction =
Vector3.Normalize(new Vector3(1.0f, -1.0f, -1.0f));
                                    effect.DirectionalLight0.SpecularColor
= Vector3.One;

                                    break;
                                }
                            default:
                                {
                                    effect.EnableDefaultLighting();
                                    break;
                                }
                        }

                        effect.View = viewMatrix;
                        effect.Projection = projectionMatrix;
                        effect.World = transforms[mesh.ParentBone.Index] *
                            Matrix.CreateScale(scale) *
                            b.rotationMatrix *

Matrix.CreateRotationZ(MathHelper.ToRadians(rotate.Z)) *
```

```
Matrix.CreateRotationY(MathHelper.ToRadians(rotate.Y)) *

Matrix.CreateRotationX(MathHelper.ToRadians(rotate.X)) *
                        Matrix.CreateTranslation(b.position);
                }
                mesh.Draw();
            }
        }
    }
}
```