UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

SIMULAÇÃO FÍSICA DE CORPOS RÍGIDOS EM 3D

VICTOR ARNDT MUELLER

BLUMENAU
2010

2010/2-29

VICTOR ARNDT MUELLER

SIMULAÇÃO FÍSICA DE CORPOS RÍGIDOS EM 3D

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

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

BLUMENAU
2010

2010/2-29

SIMULAÇÃO FÍSICA DE CORPOS RÍGIDOS EM 3D

Por

VICTOR ARNDT MUELLER

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

Presidente:	Prof. Dalton Solano dos Reis, M. Sc. – Orientador, FURB
Membro:	Prof. Antonio Carlos Tavares, Mestre – FURB
Membro:	Prof. Paulo César Rodacki Gomes, Dr. – FURB

Blumenau, 7 de dezembro de 2010

Dedico este trabalho a todos os amigos, pelo incentivo e apoio durante a realização deste trabalho.

AGRADECIMENTOS

À minha família, que mesmo longe, sempre esteve presente.

Aos meus colegas de faculdade André Luís Beling da Rosa, João Ricardo Rodrigues, Maicon Rafael Zatelli e Samuel Deschamps, por vários anos de companheirismo inigualável.

Aos meus colegas de trabalho, pelos conselhos, pelo incentivo e pela união.

À empresa, WK Sistemas, por acreditar no meu potencial, patrocinar meus estudos e ajudar a moldar o profissional que sou hoje.

Regret for the things we did can be tempered by time; it is regret for the things we did not do that is inconsolable.

Sydney Smith

RESUMO

Este trabalho apresenta a implementação de um motor de física para três dimensões. Este motor simula o comportamento físico de corpos rígidos com forma poliédrica, sendo capaz de detectar e tratar colisões entre os corpos. Para demonstrar o funcionamento da simulação física é disponibilizada uma aplicação exemplo, com alguns cenários pré-definidos.

Palavras-chave: Física. Simulação. Corpos rígidos.

ABSTRACT

This paper presents the implementation of a physics engine for three dimensions. This engine simulates the physical behavior of rigid bodies of polyhedral shape. It can detect and respond to collisions between the rigid bodies. To demonstrate the operation of the physics simulation a sample application is available, together with some predefined scenarios.

Key-words: Physics. Simulation. Rigid bodies.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

EPA – *Expanding Polytope Algorithm*

GJK – Gilbert-Johnson-Keerthi

GPU – *Graphics Processing Unit*

M3GE – Mobile 3D Graphics Engine

MBJ – *Mobile Object File*

NSV – Newton-Stormer-Verlet

OpenGL – *Open Graphics Library*

SAT – *Separating Axis Theorem*

SDK – *Software Development Kit*

SIMD – *Single Instruction Multiple Data*

SLERP – *Spherical Linear intERPolation*

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Desde os primeiros jogos de computador, sempre houve grande preocupação em criar ambientes virtuais que simulassem o mundo real de forma verossímil. No início, os algoritmos utilizados eram relativamente simples. Geralmente um objeto era representado com um retângulo ou cubo, chamado de *bounding box*, que aproximava sua forma real. Desta forma era possível testar se objetos colidiam de forma eficiente. Obviamente, os resultados eram apenas aproximações inexatas. Ainda assim, algumas técnicas deste tipo são aplicadas como uma primeira etapa antes de cálculos mais precisos. Por exemplo, um objeto costuma ter uma representação geométrica detalhada que é utilizada para visualização, e outra representação geométrica mais simples utilizada para as demais necessidades da aplicação (SATHE; SHARLET, 2008, p. 148).

O avanço tecnológico do hardware permitiu o uso de técnicas cada vez mais avançadas nestas simulações. Com esta evolução, a simulação física tornou-se um aspecto importante de grande parte dos jogos. Sem física, uma cena pode ser visualmente bem realística, mas o jogo como um todo não será (SATHE; SHARLET, 2008, p. 143).

Nos jogos atuais os objetos geralmente podem ter qualquer forma convexa e objetos mais complexos podem ser simulados usando juntas ou articulações. A simulação de física vai muito além de apenas detectar se houve uma colisão. Agora também é feita a resposta ou tratamento da colisão. É possível empilhar caixas, dirigir veículos, montar armadilhas e cenários, tudo com base no comportamento físico dos objetos. Chegou-se ao ponto onde existem componentes especializados para simulação de física. Pode-se citar o Havok physics (HAVOK, 2010), Newton game dynamics (JEREZ; SUERO, 2007) e PhysX (NVIDIA CORPORATION, 2010). Inclusive, cada um destes componentes pode ter prioridades diferentes. Havok physics e PhysX priorizam desempenho, enquanto Nweton Game Dynamics prioriza robustez e determinismo. O desenvolvedor de um jogo tem a opção de usar estes produtos no seu jogo, dependendo de qual for o mais adequado. Há ganho de tempo, pois o desenvolvedor não precisa criar uma solução própria.

Diante do exposto, este trabalho apresenta o desenvolvimento de um motor de física para corpos rígidos em 3D com a linguagem C++. Um desenvolvedor pode incorporar este motor à sua aplicação e, desta forma, concentrar-se no conteúdo específico do jogo. A plataforma alvo é o *desktop* com o sistema operacional Windows.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um motor de simulação física de corpos rígidos em 3D.

Os objetivos específicos do trabalho são:

a) detectar colisões entre objetos convexos, com as informações sobre o(s) ponto(s) de colisão, normal da colisão e profundidade de penetração;
b) tratar as colisões detectadas para separar os objetos, levando em consideração sua massa e momento de inércia;
c) simular o movimento dos objetos, utilizando as forças calculadas para determinar posição, orientação, velocidade linear e velocidade angular;
d) disponibilizar uma aplicação exemplo para demonstrar a utilização do módulo e visualizar a simulação.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está dividido em quatro capítulos, sendo que o segundo capítulo apresenta a fundamentação teórica com conceitos de física e detecção e tratamento de colisões. Além disso, o capítulo também expõe detalhes de certos trabalhos correlatos.

O terceiro capítulo trata do desenvolvimento do motor de física, iniciando com os requisitos e a especificação da aplicação. Fazem parte desta especificação os diagramas de casos de uso, de classes, de atividades e de sequência. Ainda no terceiro capítulo são comentados os resultados e problemas encontrados durante a implementação do sistema.

Por fim, no quarto capítulo são apresentadas as conclusões finais sobre o trabalho e sugestões para extensões.

2 FUNDAMENTAÇÃO TEÓRICA

Para o entendimento deste trabalho são explanados os princípios físicos da dinâmica de corpos rígidos. Logo em seguida é detalhado o funcionamento da etapa de detecção de colisões. De forma similar, é apresentado o funcionamento da etapa de tratamento de colisões. Na sequência é explicado o assunto de simulação. Também são expostos os conceitos de robustez numérica e orientação em 3D. Por fim, em trabalhos correlatos, são mencionadas ferramentas existentes no mercado e um trabalho com funcionalidades semelhantes às do módulo que foi implementado.

2.1 DINÂMICA DE CORPOS RÍGIDOS

Corpo rígido é um objeto que não é deformado pelas forças que são aplicadas a ele. Sua forma permanece a mesma conforme se move e rotaciona (FEYNMAN; LEIGHTON; SANDS, 1963). A física de corpos rígidos pode ser dividida em cinemática e dinâmica. Na cinemática apenas o movimento dos corpos é estudado. Já na dinâmica as forças que causaram este movimento também são consideradas. Todos os cálculos envolvendo estas forças são baseados nas leis de movimento de Newton, as quais são (NEWTON, 1687, p. 13):

a) um corpo que esteja em movimento ou em repouso, tende a manter seu estado inicial;
b) a resultante das forças que agem num corpo é igual ao produto de sua massa pela aceleração adquirida;
c) para toda força aplicada, existe outra de mesmo módulo, mesma direção e sentido oposto.

Quando um corpo rígido rotaciona, geralmente rotaciona no eixo que passa através do seu centro de massa, a menos que o corpo esteja articulado em outro ponto através do qual é forçado a rotacionar. Portanto, lidar com corpos rígidos envolve dois aspectos distintos (BOURG, 2002, p. 49):

a) rastrear a translação do centro de massa;
b) rastrear a rotação do corpo.

Com relação ao movimento linear, as propriedades relevantes são posição, velocidade,

aceleração, massa e força. A posição representa uma distância a partir de um ponto de referência. Velocidade é a variação da posição em função do tempo. Aceleração por sua vez é a variação da velocidade em função do tempo. Massa, para os propósitos da simulação de corpos rígidos, é a resistência de um corpo à mudança de sua velocidade. Por fim, força é a relação entre massa e aceleração (BOURG, 2002, p. 6).

Com relação ao movimento angular, existem propriedades equivalentes para cada propriedade do movimento linear, as quais são: orientação, velocidade angular, aceleração angular, momento de inércia e torque. Da mesma forma que a posição representa uma distância a partir de um ponto de referência, a orientação representa uma rotação a partir de uma orientação de referência. Momento de inércia é análogo à massa, representando a resistência de um corpo à mudança de sua velocidade angular. Por fim, torque é a força multiplicada pela distância do ponto onde esta foi aplicada até o eixo de rotação (BOURG, 2002, p. 65).

2.2 DETECÇÃO DE COLISÃO

Detecção de colisão é um problema geométrico onde o objetivo é determinar quando e onde dois ou mais objetos colidiram. Além de informar os objetos que estão colidindo, um algoritmo de detecção de colisão deve calcular informações sobre a colisão. Estas informações incluem o(s) ponto(s) de contato, a profundidade de penetração e a normal da colisão. Os resultados obtidos devem ser precisos e confiáveis, pois serão utilizados na próxima etapa, que é o tratamento da colisão (BOURG, 2002, p. 87).

A etapa de detecção pode ser modelada de duas formas, discreta ou contínua. Com a primeira opção a posição dos objetos é atualizada iterativamente com um intervalo de tempo discreto. Feito isso, é verificado se existe interseção entre qualquer par de objetos. Este processo é simples e eficiente, mas apresenta desvantagens. A principal delas é o efeito chamado de *tunneling*. Ao realizar uma iteração da simulação, é possível que um objeto que se move rapidamente atravesse completamente um obstáculo (PAWASKAR, 2007). Esta situação é demonstrada na Figura 1, onde `p` representa a posição inicial de um objeto e `p'` representa sua posição após uma iteração.

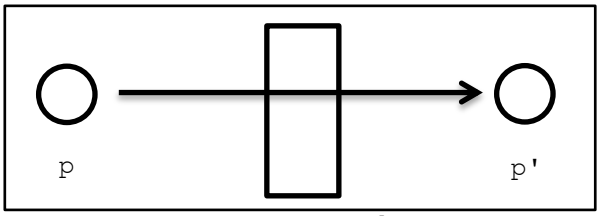


Figura 1 - *Tunneling*

A detecção contínua por outro lado faz uso de um intervalo de tempo variável. Este intervalo é ajustado para cada par de objetos que tem potencial para colidir. Desta forma os corpos nunca intersectam uns aos outros. A desvantagem deste método é que ele tem um custo computacional maior e sua implementação é mais complexa (SATHE; SHARLET, 2008, p. 144).

2.2.1 Algoritmo XenoCollide

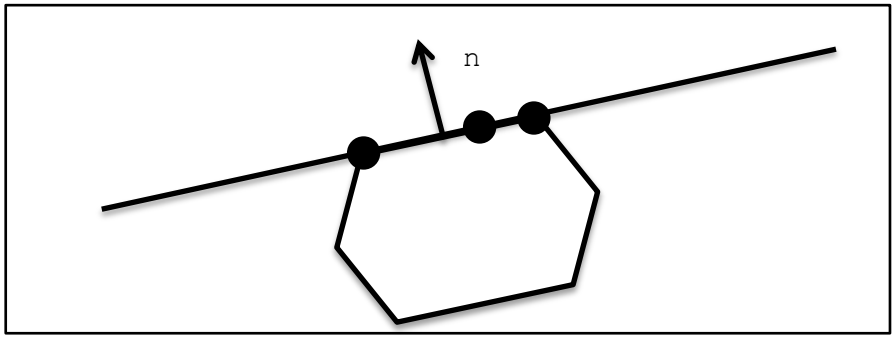
O algoritmo XenoCollide é utilizado para detectar colisões em 3D. Este algoritmo não requer nenhuma informação específica sobre a geometria dos objetos. Ao invés disso, faz uso unicamente de uma função de mapeamento de suporte, que recebe um vetor de direção e retorna o ponto mais distante do objeto nesta direção. Com base nisso o ponto de contato é determinado de forma iterativa, aproximando sua posição a cada iteração. É possível iniciar este processo de aproximação com valores de uma execução anterior, geralmente chegando na resposta correta em poucas iterações. Isso o torna muito popular para simulações de física em tempo real (SNETHEN, 2008, p. 165).

2.2.1.1 Funções de mapeamento de suporte

Algoritmos que operam com um grande número de formas geométricas precisam de uma representação uniforme para estas formas. Uma função de mapeamento de suporte é uma maneira simples e elegante de atender este requerimento, sendo possível representar qualquer forma convexa (SNETHEN, 2008, p. 166).

Uma função de mapeamento de suporte é uma função matemática, geralmente muito simples, que recebe um vetor de direção como entrada e retorna o ponto do objeto convexo que está mais distante na direção informada. Se vários pontos atendem este requisito qualquer um deles pode ser retornado desde que sempre o mesmo ponto seja retornado para a mesma

direção. Esta situação é demonstrada na Figura 2, onde `n` representa o vetor de direção.



Fonte: Snethen (2008, p. 167).

Figura 2 – Vários pontos de suporte válidos

2.2.1.2 Diferença de Minkowski de dois objetos

Toda forma convexa pode ser tratada como um conjunto infinito de pontos. Ao subtrair cada um dos pontos da forma `A` de cada um dos pontos da forma `B` é possível criar uma terceira forma. Esta terceira forma `B-A` é denominada a diferença de Minkowski de `A` e `B`. Geometricamente isto pode ser entendido como mover uma forma ao redor da borda da outra forma. Se `A` e `B` forem convexos, a sua diferença de Minkowski também será convexa. (SNETHEN, 2008, p. 170). A Figura 3 demonstra essa operação. Firth (2010) disponibiliza um *applet* para facilitar a visualização e entendimento desta operação.

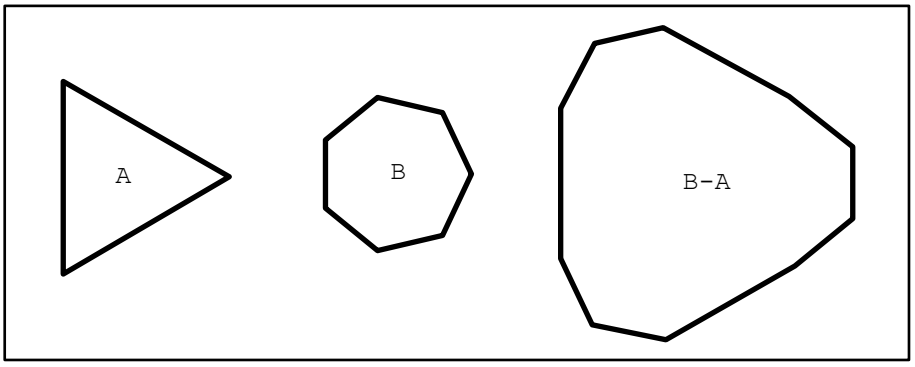


Figura 3 – Diferença de Minkowski

Se duas formas `A` e `B` estão colidindo isto significa que existe pelo menos um ponto do objeto `A` que compartilha a mesma posição de outro ponto do objeto `B`. Quando um destes pontos é subtraído do outro o resultado é o vetor zero, isto é, a origem. De forma similar, se as duas formas não estão colidindo não existem pontos em comum e a subtração de nenhum par de pontos irá gerar a origem. Portanto é possível descobrir se dois objetos estão colidindo se a sua diferença de Minkowski contiver a origem.

Ainda assim, gerar a diferença de Minkowski é uma operação computacionalmente

custosa. Felizmente tudo que é necessário para descobrir se a origem está contida na forma B-A é a função de mapeamento de suporte desta nova forma. Isto é fácil de calcular tendo as funções individuais de A e B (SNETHEN, 2008, p. 170).

2.2.1.3 Refinamento de portais de Minkowski

O algoritmo XenoCollide faz parte de um conjunto de algoritmos baseados na técnica denominada refinamento de portais de Minkowski. O pseudocódigo do algoritmo é apresentado no Quadro 1.

```
// Fase 1: determinar portal inicial
calcular_raio_origem();
calcular_portal_inicial();
while( raio origem não intersecta portal )
{
    calcular_novo_portal_inicial();
}

// Fase 2: refinamento do portal
while( true )
{
    if( origem dentro do portal ) return colisão;
    calcular_suporte_na_direcao_do_portal();
    if( origem além do plano de suporte ) return separado;
    if( plano de suporte próximo do portal ) return separado;
    calcular_novo_portal();
}
```

Fonte: Snethen (2008, p. 171).

Quadro 1 – Pseudocódigo do algoritmo XenoCollide

A primeira fase do algoritmo inicia determinando um ponto qualquer que esteja no interior da diferença B-A. Este ponto pode ser calculado a partir de qualquer ponto no interior de B e subtraindo qualquer ponto no interior de A. O centro de massa do objeto é uma opção conveniente. Este ponto inicial é denominado V0. Se o raio que inicia em V0 e vai até a origem passar pela superfície de B-A antes de passar pela origem significa que a origem está fora de B-A. Reciprocamente, se o raio passar pela origem antes de passar pela superfície de B-A a origem está dentro de B-A (SNETHEN, 2008, p. 171).

Em seguida um portal inicial é calculado usando a função de mapeamento de suporte.

Para montar o portal são necessários três pontos não colineares na superfície de B-A. Estes pontos são denominados V1, V2 e V3. O raio para a origem pode ou não passar por este portal triangular inicial. Para testar isto basta verificar se a origem está dentro dos três planos formados pelas arestas do triângulo do portal e pelo vértice inicial V0. Isto é, os planos formados pelos vértices (V0, V1, V2), (V0, V2, V3) e (V0, V3, V1). Enquanto a origem estiver do lado de fora de um dos planos, é necessário utilizar a normal exterior do plano para buscar um novo ponto de suporte e montar um novo portal (SNETHEN, 2008, p. 172).

Os pontos V0, V1, V2 e V3 formam um tetraedro. Devido à convexidade de B-A este tetraedro está completamente dentro de B-A. Portanto se a origem estiver dentro do tetraedro também estará dentro de B-A. O algoritmo já verificou que a origem está dentro dos três planos das faces do tetraedro. Portanto a segunda fase do algoritmo inicia verificando se a origem está dentro do tetraedro testando o plano da quarta face, isto é, o portal. Se a origem estiver dentro do portal o algoritmo termina indicando que houve colisão (SNETHEN, 2008, p. 174).

Caso contrário a origem está além do portal. Para determinar se a origem está dentro ou fora de B-A é necessário buscar mais informações sobre o que há além do portal. Portanto um novo ponto de suporte, denominado V4, é calculado usando a normal exterior do portal. Se a origem ainda estiver além do plano formado por este novo ponto e pela normal o algoritmo termina indicando que não houve colisão. Caso contrário é necessário refinar o portal descartando um ponto antigo e usando o novo ponto. Para isto é necessário verificar em que lado a origem está dos planos formados pelos vértices (V4, V0, V1), (V4, V0, V2) e (V4, V0, V3) (SNETHEN, 2008, p. 175).

Após algumas iterações do algoritmo o portal irá rapidamente aproximar a superfície de B-A. Porém se esta superfície for curvada a origem pode estar infinitesimalmente próxima da superfície. Neste caso os portais refinados podem precisar de um número arbitrário de iterações até passar da origem. Para terminar o algoritmo nestas condições é utilizada uma tolerância. Quando a distância do portal até a superfície de B-A for menor que esta tolerância o algoritmo termina. É possível terminar indicando que houve ou não colisão. Para simulações físicas é geralmente melhor terminar indicando que não houve colisão. Isto porque uma pequena penetração entre os objetos dificilmente será percebida, enquanto que uma separação não irá aparentar ser natural. Também é interessante notar que o fluxo básico do algoritmo é o mesmo em qualquer dimensão. Em 2D o portal é um segmento de linha ao invés de um triângulo, e em 4D o portal é um tetraedro (SNETHEN, 2008, p. 176).

2.2.1.4 Obtendo informações do contato

Se tudo que é necessário é um resultado booleano indicando se houve colisão ou não o algoritmo pode terminar assim que o portal passar além da origem. Porém se informações do contato forem necessárias para a aplicação o algoritmo pode continuar executando para determinar o ponto de contato, a normal de contato e a profundidade de penetração.

O refinamento de portais de Minkowski oferece várias formas de obter informações do contato. A técnica utilizada no algoritmo XenoCollide é simplesmente continuar projetando o raio da origem na direção da superfície de `B-A`. Isto representa afastar os objetos um do outro através de uma linha conectando seus pontos interiores até que eles estejam apenas encostando. Isto é eficiente e simples, resultando em informações de contato estáveis (SNETHEN, 2008, p. 176).

2.3 TRATAMENTO DE COLISÃO

O tratamento de colisão é um problema físico que determina as novas velocidades linear e angular dos objetos após a colisão, com o objetivo de separá-los. Intuitivamente a solução é aplicar uma força aos corpos, porém isso não os separa instantaneamente. Uma força atua sobre um corpo em função do tempo, ou em outras palavras, uma força leva tempo para alterar a velocidade do objeto. Como a colisão deve ser resolvida imediatamente, outra grandeza é utilizada, o impulso. Pode-se imaginar impulso como sendo uma força muito forte que é aplicada por um período muito curto de tempo (HECKER, 1997, p. 12).

O Quadro 2 demonstra como a velocidade linear é alterada pelo impulso. Neste quadro `j` é o impulso, `M` é a massa do objeto, $\mathbf{n}$ é o vetor normal da colisão, $\mathbf{v_1}$ é a velocidade inicial do objeto, e $\mathbf{v_2}$ é a nova velocidade. É possível perceber que quanto maior for a massa do objeto, menor é o efeito do impulso sobre o objeto (HECKER, 1997, p. 14).

$$\boldsymbol{v_2} = \boldsymbol{v_1} + \frac{j}{M}\boldsymbol{n}$$

Fonte: Hecker (1997, p. 14).

Quadro 2 – Alteração da velocidade pelo impulso

Existem várias formas de calcular o valor do impulso, dependendo da exatidão e realismo desejados. Uma das mais simples é através da lei da restituição instantânea sem

atrito, de Newton. Esta lei assume que o processo de colisão não leva tempo nenhum. Por causa disso, outras forças que não estão envolvidas no impacto, como gravidade, não são consideradas. Esta lei introduz uma nova grandeza, o coeficiente de restituição. Este coeficiente é um valor escalar de 0 a 1 que relaciona a velocidade do ponto de contato antes e depois do impacto. O valor zero indica uma colisão perfeitamente inelástica, e um indica uma colisão perfeitamente elástica. Com as informações obtidas pelo algoritmo de detecção (normal da colisão, ponto de contato, profundidade), as informações do objeto (posição, orientação, velocidade, velocidade angular, massa e momento de inércia) e as informações da colisão (coeficiente de restituição) é possível então calcular o impulso (HECKER, 1997, p. 12).

O Quadro 3 demonstra a equação para calcular o impulso. Neste quadro `j` é o impulso resultante, `e` é o coeficiente de restituição, $\mathbf{v^{AB}}$ é a velocidade relativa no ponto de contato entre os objetos `A` e `B`, $\mathbf{n}$ é o vetor normal da colisão, M_A é a massa de `A`, M_B é a massa de `B`, $\mathbf{I_A^{-1}}$ é o momento de inércia inverso de `A`, $\mathbf{I_B^{-1}}$ é o momento de inércia inverso de `B`, $\mathbf{r_{AP}}$ é o vetor perpendicular ao vetor da origem de `A` até o ponto de contato `P` e $\mathbf{r_{BP}}$ é o vetor perpendicular ao vetor da origem de `B` até o ponto de contato `P` (HECKER, 1997, p. 16).

$$j = \frac{-(1+e)\boldsymbol{v^{AB}} \cdot \boldsymbol{n}}{\boldsymbol{n} \cdot \boldsymbol{n}\left(\frac{1}{M_A} + \frac{1}{M_B}\right) + \left[\left(\boldsymbol{I_A^{-1}}(\boldsymbol{r_{AP}} \times \boldsymbol{n})\right) \times \boldsymbol{r_{AP}} + \left(\boldsymbol{I_B^{-1}}(\boldsymbol{r_{BP}} \times \boldsymbol{n})\right) \times \boldsymbol{r_{BP}}\right] \cdot \boldsymbol{n}}$$

Fonte: adaptado de Hecker (1997, p. 16).

Quadro 3 – Cálculo do impulso

Para calcular a velocidade relativa entre os objetos é necessário primeiro calcular a velocidade de cada objeto no ponto de contato. Como estes objetos podem estar rotacionando é necessário levar em consideração a velocidade angular. O Quadro 4 demonstra o cálculo da velocidade em um ponto `P` do objeto `A`. A velocidade linear do objeto é representada por **v** enquanto que a velocidade angular é representada por $\boldsymbol{\omega}$ (HECKER, 1997, p. 19).

$$\boldsymbol{v^P} = \boldsymbol{v^A} + \boldsymbol{\omega^A} \times \boldsymbol{P}$$

Fonte: adaptado de Hecker (1997, p. 19).

Quadro 4 – Velocidade em um ponto

Com base na velocidade de cada objeto no ponto `P` é possível calcular a velocidade relativa entre os objetos simplesmente através da diferença entre as velocidades de cada objeto. Isto é demonstrado no Quadro 5.

$$\boldsymbol{v^{AB}} = \boldsymbol{v^A} - \boldsymbol{v^B}$$

Fonte: Hecker (1997, p. 19).

Quadro 5 – Velocidade relativa entre dois objetos

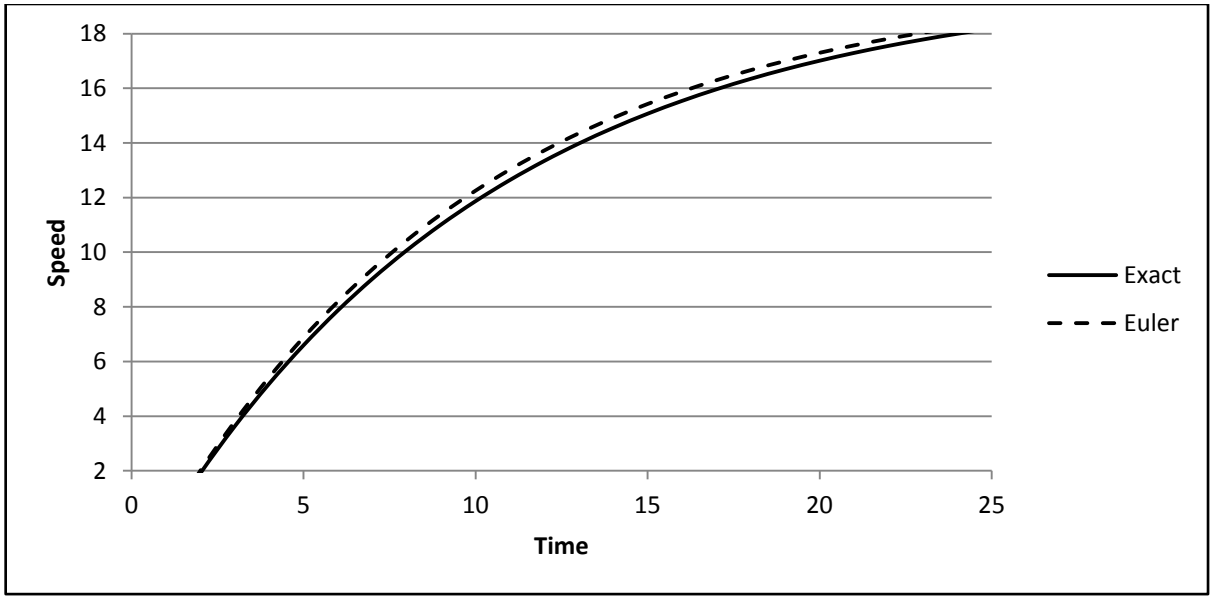
2.4 SIMULAÇÃO

A simulação em tempo real é um problema fundamentalmente matemático. O objetivo é calcular o estado (posição, orientação, velocidade e velocidade angular) de um objeto após um intervalo de tempo, denominado *timestep*. Para fazer isso são utilizadas técnicas de integração numérica. A velocidade, por exemplo, é a derivada da posição em função do tempo. Portanto, para calcular a nova posição de um objeto após uma iteração do simulador é necessário integrar a velocidade (BOURG, 2002, p. 173).

A forma mais simples para realizar esta integração é através do método de Euler, porém também é a menos precisa. A Figura 4 compara os valores obtidos através do método citado com os valores exatos esperados. O método pode ser explicado como uma série de Taylor truncada após o termo que inclui a primeira derivada. É possível melhorar sua precisão diminuindo o *timestep*, porém isso aumenta o custo computacional e, se for usado um *timestep* de valor muito pequeno, pode apresentar problemas devido à representação binária de números de ponto flutuante (BOURG, 2002, p. 174). O Quadro 6 demonstra como a velocidade, denotada por `v`, e a posição, denotada por `p`, são atualizadas em função do tempo, denotado por `t`.

$$\boxed{\begin{aligned} \boldsymbol{p}' &= \boldsymbol{p} + \boldsymbol{v} \cdot t \\ \boldsymbol{v}' &= \boldsymbol{v} + \boldsymbol{a} \cdot t \end{aligned}}$$

Quadro 6 – Método de Euler



Fonte: Bourg (2002, p. 176).

Figura 4 - Comparação do método de Euler

Outra forma de realizar a integração é através do método Newton-Stormer-Verlet

(NSV), também conhecido como método de Euler-Cromer ou Euler semi-implícito. Este método também é um integrador de primeira ordem, muito similar ao método de Euler. A principal diferença está no fato deste integrador ser simplético. Isto significa que a energia do sistema é conservada quase perfeitamente, desde que a aceleração seja independente do tempo (MACDONALD, 2002). O Quadro 7 demonstra como o método NSV é similar ao método de Euler.

$$\boxed{\begin{aligned} \boldsymbol{v}' &= \boldsymbol{v} + \boldsymbol{a} \cdot t \\ \boldsymbol{p}' &= \boldsymbol{p} + \boldsymbol{v}' \cdot t \end{aligned}}$$

Quadro 7 – Método NSV

Em situações onde a aceleração depende do tempo, é possível usar o método Runge-Kutta. Este método também pode ser descrito como uma série de Taylor truncada. A diferença está apenas no ponto onde a série é truncada. Na implementação clássica, ela é truncada após o termo com a quarta derivada. Porém, ao invés de determinar a segunda, terceira e quarta derivada da função sendo integrada, os seus valores são aproximados realizando outra expansão da série de Taylor. Estes valores são então substituídos na expressão original. Quanto mais termos da série forem utilizados, maior é a precisão obtida (BOURG, 2002, p. 180).

2.4.1 *Timestep* fixo e variável

O valor do *timestep* pode ser fixo ou variável. Isso é, a cada iteração para atualizar o estado dos objetos, o *timestep* pode ser sempre igual ou pode variar entre uma iteração e outra. O valor utilizado pode influenciar a simulação de várias formas. O efeito pode ser sutil, como pequenas alterações na forma como a simulação trata colisões. O efeito também pode ser visível, pois um *timestep* muito grande pode causar o efeito *tunneling* descrito anteriormente. Por fim, todos os integradores descritos anteriormente produzem resultados melhores se o *timestep* for constante (FIEDLER, 2006).

Porém ter um valor fixo traz outra dificuldade. A não ser que o valor escolhido seja exatamente igual ao tempo necessário para executar uma iteração, a simulação não irá executar em tempo real. Na prática é impossível escolher um valor que será sempre igual, pois computadores diferentes vão levar tempos diferentes para executar a simulação. Por este motivo, ao utilizar um *timestep* fixo é necessário uma forma de desacoplar a simulação de física das atualizações de tela (FIEDLER, 2006).

2.5 ROBUSTEZ NUMÉRICA

Robustez é um termo usado para se referir à capacidade de um programa de lidar com computações numéricas ou configurações geométricas que são de alguma forma difíceis de tratar. Um programa robusto não trava nem entra em *loops* infinitos por causa de erros numéricos. Um algoritmo robusto retorna resultados consistentes para entradas equivalentes e trata casos degenerados de forma previsível. Código para detecção de colisão é especialmente sensível a problemas deste tipo devido à sua natureza numérica e geométrica (ERICSON, 2005, p. 427).

Existem duas categorias para os problemas de robustez: aqueles causados por inexatidão e aqueles causados por casos degenerados. Os problemas de inexatidão são causados pelo fato que cálculos feitos em um computador não usam aritmética real e exata, mas sim aritmética de ponto flutuante ou ponto fixo, que possuem precisão limitada. São exemplos: erros de representação, *overflow*, *underflow*, arredondamento e cancelamento de dígitos. Já casos degenerados são situações ou condições especiais para as quais um algoritmo não está preparado. Por exemplo, um algoritmo para determinar o ponto de interseção entre uma reta e um plano pode retornar resultados incorretos se a reta for paralela ao plano (ERICSON, 2005, p. 428).

2.6 ORIENTAÇÃO EM 3D

Diferentemente da posição, que pode ser facilmente representada com um vetor, a representação da orientação em 3D é mais complexa, e não é possível extrapolar o conceito de rotação em 2D. No caso bidimensional há apenas um grau de liberdade. Isso significa que um objeto pode ser rotacionado apenas ao redor de um eixo perpendicular ao plano. Já no caso tridimensional há três graus de liberdade, pois é possível rotacionar ao redor dos três eixos coordenados. As formas mais utilizadas para descrever a orientação de um objeto são ângulos de Euler, matrizes de rotação e *quaternions* (DUNN; PARBERRY, 2002, p. 148).

Ângulos de Euler definem a orientação como sendo uma sequência de três rotações ao redor de três eixos perpendiculares entre si, geralmente os eixos cardinais. Neste caso os ângulos são denominados como direção, inclinação e balanceio. Esta representação é a menor

possível e de fácil entendimento, mas apresenta sérias desvantagens. Uma delas é a dificuldade em realizar interpolações entre duas orientações devido à natureza cíclica de ângulos de rotação. Porém a principal desvantagem é denominada *Gimbal lock*, que ocorre, por exemplo, quando a inclinação é de ±90º. Neste caso os outros dois ângulos irão rotacionar ao redor do mesmo eixo e um dos graus de liberdade é perdido. De forma mais genérica é possível provar matematicamente que qualquer representação de orientação em 3D que utilize apenas três números irá apresentar singularidades como a descrita (DUNN; PARBERRY, 2002, p. 156).

Uma matriz de rotação tem tamanho 3×3 e, quando multiplicada por um vetor, gera como resultado o vetor rotacionado ao redor de um eixo. É possível usar matrizes para converter pontos de um sistema de coordenadas para outro sistema, onde o primeiro está rotacionado em relação ao segundo. Com a matriz inversa é possível transformar o vetor novamente para o sistema original. Toda matriz de rotação é ortogonal, e portanto sua inversa, é igual à sua transposta. Esta representação apresenta desvantagens como consumo maior de memória e dificuldade para ser utilizada por um humano, já que não é intuitiva. Mas a principal desvantagem é que pode representar uma orientação inválida, pois utiliza nove números onde apenas três são necessários. Por exemplo, após várias alterações incrementais numa matriz ela pode deixar de ser ortogonal devido a erros de arredondamento. Se isso ocorrer ela não irá apenas rotacionar objetos, mas também deformá-los. Este fenômeno chama-se *matrix creep* e precisa ser tratado repetidamente com uma ortogonalização da matriz (DUNN; PARBERRY, 2002, p. 150).

Por fim, um *quaternion* representa uma orientação com quatro valores, e assim não apresenta os problemas vistos com ângulos de Euler. Pode-se imaginar um *quaternion* como sendo um eixo $\mathbf{v}$ de comprimento unitário e uma rotação θ ao redor de $\mathbf{v}$. Estas informações não são armazenadas diretamente nos valores, conforme pode ser visto no Quadro 8. Ao invés disso o *quaternion* tem quatro valores, `w`, `x`, `y`, e `z`. O componente `w` está relacionado à θ, mas não são iguais. De forma similar, os componentes `x`, `y`, e `z` estão relacionados à $\mathbf{v}$, mas não são idênticos. A principal vantagem de *quaternions* é uma operação denominada *Spherical Linear intERPolation* (SLERP). Com esta operação é possível interpolar duas orientações de forma suave e sem os problemas que afligem ângulos de Euler. Sua representação também consome razoavelmente pouca memória. Entretanto, de forma similar à matriz de rotação, sua utilização não é intuitiva e é possível representar uma orientação inválida. Este último problema deve ser resolvido com normalização do vetor, para garantir que este sempre terá comprimento unitário (DUNN; PARBERRY, 2002, p. 160).

$$[\cos(\theta/2) \quad (\sin(\theta/2)\ \boldsymbol{v}_x \quad \sin(\theta/2)\,\boldsymbol{v}_y \quad \sin(\theta/2)\,\boldsymbol{v}_z)]$$

Fonte: Dunn; Parberry (2002, p. 162).

Quadro 8 – Os valores de um *quaternion*

2.7 TRABALHOS CORRELATOS

Nesta seção são apresentadas três ferramentas e um trabalho com funcionalidades similares àquelas que serão implementadas no componente de física proposto.

A primeira ferramenta é a Havok Physics (HAVOK, 2010), um motor de física desenvolvido pela empresa Havok. Possui funcionalidades como detecção de colisão contínua e *ragdolls*[1]. Pode ser utilizado em praticamente todas as plataformas no mercado, como PC, Mac, Linux, XBox 360, Play Station 3 e Nintendo Wii. Desde o lançamento do seu *Software Development Kit* (SDK) em 2000, já foi utilizado em mais de 150 jogos, notavelmente Halo 3, Half-Life 2, Starcraft 2 e Star Wars The Force Unleashed. Ao obter sua licença, o desenvolvedor recebe a maior parte do código fonte, permitindo fazer alterações e conversões para outras plataformas.

A segunda ferramenta é a PhysX (NVIDIA CORPORATION, 2010), outro motor de física. Este foi desenvolvido pela empresa AGEIA e depois comprado pela NVIDIA. Apesar de ser uma ferramenta proprietária, ela é dispobilizada gratuitamente para algumas plataformas, incluindo código fonte, podendo ser utilizada em projetos comerciais ou não. Também pode ser utilizada em várias plataformas, especificamente PC, XBox 360, Play Station 3 e Nintendo Wii. Em especial, este *middleware* executa parte da simulação na *Graphics Processing Unit* (GPU) se a placa de vídeo tiver suporte a esta funcionalidade. Isso deixa o processador livre para realizar outras operações. Desde seu lançamento em 2004, vários jogos foram criados utilizando este motor, em especial Batman Arkham Asylum, Mass Effect 2, Mirror's Edge e Unreal Tournament 3.

Uma terceira ferramenta é a Newton Game Dynamics (JEREZ; SUERO, 2007). Diferentemente dos outros motores apresentados, este foca em determinismo e exatidão ao invés de desempenho. Devido à diferença citada, consegue simular interações entre objetos com massa desproporcionais, na ordem de 400 para 1, que causariam problemas nos outros motores. Apesar de ser proprietário e ter código fechado, pode ser utilizado gratuitamente.

[1] *Ragdoll* é um boneco articulado composto por vários corpos rígidos ligados por juntas e com restrições aos seus movimentos (SATHE; SHARLET, 2008, p. 145).

Entre os jogos que fazem uso deste motor pode-se citar Mount & Blade e Penumbra: Overture.

Por fim, há o Mobile 3D Graphics Engine (M3GE) (GOMES; PAMPLONA, 2005). Como indicado pelo nome, é um motor para aparelhos móveis ou com recursos limitados. Este motor de jogos não é focado unicamente em simulação física, pois implementa os principais componentes necessários para um jogo como importação e renderização de ambientes 3D, criação de câmeras, tratamento de eventos e movimentação de personagens no cenário. Neste trabalho foi definido o formato de arquivo *Mobile Object File* (MBJ) para modelos 3D. Também, devido as limitações da plataforma, a física é relativamente simples, sendo apenas em 2D, apesar do universo simulado ser graficamente 3D.

3 DESENVOLVIMENTO

Neste capítulo são abordadas as etapas do desenvolvimento do projeto. São apresentados os principais requisitos, a especificação, a implementação e por fim são listados os resultados e discussão.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O simulador de física deverá:

a) detectar colisões entre poliedros convexos e calcular as informações necessárias para tratar as colisões (Requisito Funcional – RF);
b) tratar as colisões detectadas, separando os objetos e calculando sua nova velocidade linear e velocidade angular, levando em consideração suas massas e seus momentos de inércia (RF);
c) simular o movimento dos corpos (RF);
d) disponibilizar uma aplicação exemplo para demonstrar sua utilização e visualizar a simulação (RF);
e) ser implementado utilizando o ambiente Visual Studio 2010 e a linguagem C++ (Requisito Não-Funcional – RNF);
f) apresentar desempenho de pelo menos 30 quadros por segundo ao simular uma quantidade de objetos que normalmente espera-se encontrar em um jogo (RNF);
g) apresentar robustez numérica (RNF).

3.2 ESPECIFICAÇÃO

A especificação do presente trabalho foi desenvolvida utilizando alguns dos diagramas da *Unified Modeling Language* (UML) em conjunto com a ferramenta Enterprise Architect 8.0.613 para a elaboração dos diagramas de casos de uso, de classes, de atividades e de sequência.

3.2.1 Diagrama de casos de uso

Como este trabalho é um motor de física para ser utilizado numa aplicação, o desenvolvedor da aplicação deverá ser considerado como o usuário. A Figura 5 apresenta o diagrama de casos de uso com as principais interações do usuário com o sistema.

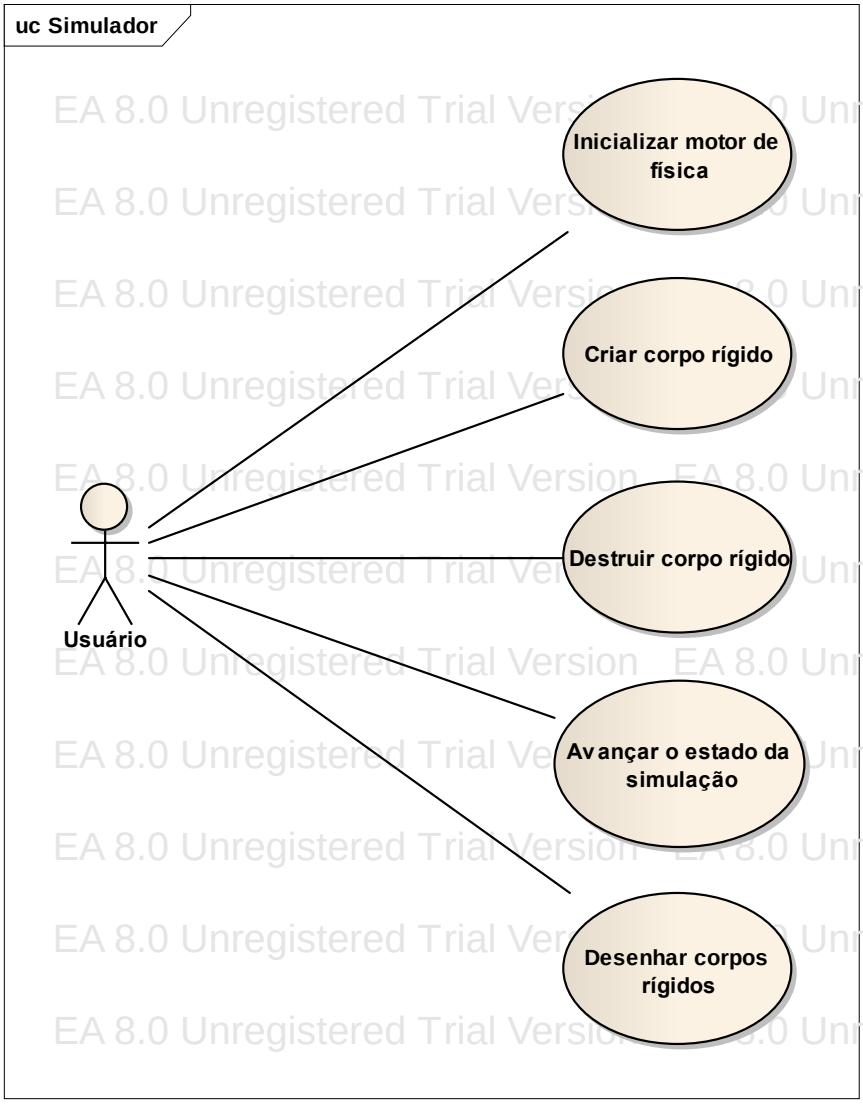


Figura 5 – Diagrama de casos de uso

O caso de uso `Inicializar motor de física` (Quadro 9) descreve como o usuário irá inicializar o sistema de simulação física. Este caso possui apenas um cenário principal.

Inicializar motor de física: possibilita ao usuário inicializar o sistema de simulação física	
Cenário principal	1. O usuário cria uma instância da classe `PhysicsSystem`. 2. O usuário chama o método `Init` para inicializar o simulador.
Pós-condição	A instância criada está pronta para criar e simular corpos rígidos.

Quadro 9 – Caso de uso `Inicializar motor de física`

O segundo caso de uso `Criar corpo rígido` (Quadro 10), demonstra como o usuário pode criar objetos na simulação e especificar suas propriedades como massa, posição inicial e forma. Além do fluxo principal, este caso de uso possui um fluxo alternativo onde o usuário pode criar um objeto que não é movido por colisões com outros objetos. Por exemplo, isto é necessário para criar objetos que sirvam como chão.

Criar corpo rígido: possibilita ao usuário criar objetos na simulação física	
Pré-condição	O usuário deve ter inicializado o simulador de física.
Cenário principal	1. O usuário chama o método `CreateBody` da classe `PhysicsSystem`. 2. O usuário escolhe a forma do objeto retornado através do atributo `shape`. 3. O usuário chama o método `ComputeMassProperties` informando a densidade do objeto. 4. O sistema calcula a massa, centro de massa e momento de inércia do objeto. 5. O usuário informa outros dados relevantes para a simulação, como posição e velocidade inicial.
Fluxo Alternativo 01	No passo 3, caso o usuário deseje que o objeto não possa ser movido por colisões com outros objetos, o método `ComputeMassProperties` não deve ser chamado.
Pós-condição	O objeto criado é adicionado na simulação.

Quadro 10 – Caso de uso `Criar corpo rígido`

O terceiro caso de uso `Destruir corpo rígido` (Quadro 11), demonstra como o usuário pode remover um objeto da simulação. Este caso de uso possui apenas um cenário principal.

Destruir corpo rígido: possibilita ao usuário remover objetos da simulação física	
Pré-condição	O usuário deve ter inicializado o simulador de física.
Cenário principal	1. O usuário chama o método `DestroyBody` da classe `PhysicsSystem`, informando o objeto a ser destruído. 2. O sistema além de destruir o objeto informado também remove todas as informações de contato deste corpo com outros corpos rígidos.
Pós-condição	O objeto e todos seus contatos com outros corpos são destruídos.

Quadro 11 – Caso de uso `Destruir corpo rígido`

O quarto caso de uso `Avançar o estado da simulação` (Quadro 12), demonstra como o usuário pode executar a simulação de física. Além do fluxo principal, este caso de uso possui um fluxo alternativo onde o usuário pode executar a simulação passo-a-passo.

Avançar o estado da simulação: possibilita ao usuário executar a simulação de física	
Pré-condição	O usuário deve ter inicializado o simulador de física.
Cenário principal	1. O usuário chama o método `Update` da classe `PhysicsSystem`, informando quantos segundos devem ser simulados.
Fluxo Alternativo 01	No passo 1, o usuário chama o método `SingleStep` ao invés de `Update` para simular apenas um *timestep* de duração fixa.
Pós-condição	A posição e velocidade dos objetos são atualizadas levando em consideração as colisões detectadas.

Quadro 12 – Caso de uso `Avançar o estado da simulação`

O quinto e último caso de uso `Desenhar corpos rígidos` (Quadro 13), demonstra como o usuário pode buscar informações dos corpos rígidos após uma etapa de simulação e com estas informações desenhar os objetos. Este caso de uso possui apenas um cenário principal.

Desenhar corpos rígidos: possibilita ao usuário buscar informações dos corpos rígidos após uma etapa de simulação e desenhar os objetos	
Pré-condição	O usuário deve ter inicializado o simulador de física.
Cenário principal	1. O usuário chama os métodos `BodiesBeginIterator` e `BodiesEndIterator` da classe `PhysicsSystem` para obter um iterador de corpos rígidos. 2. Para cada objeto utiliza-se a posição e orientação para desenhar os triângulos da sua representação visual.
Pós-condição	Os objetos são desenhados em tela.

Quadro 13 – Caso de uso `Desenhar corpos rígidos`

3.2.2 Diagrama de classes

A Figura 6 apresenta a especificação das classes do motor de física e da aplicação exemplo. Em seguida é feita uma análise sobre a funcionalidade de cada uma delas. Algumas classes meramente auxiliares foram omitidas para uma melhor visualização do diagrama, não comprometendo o entendimento do mesmo.

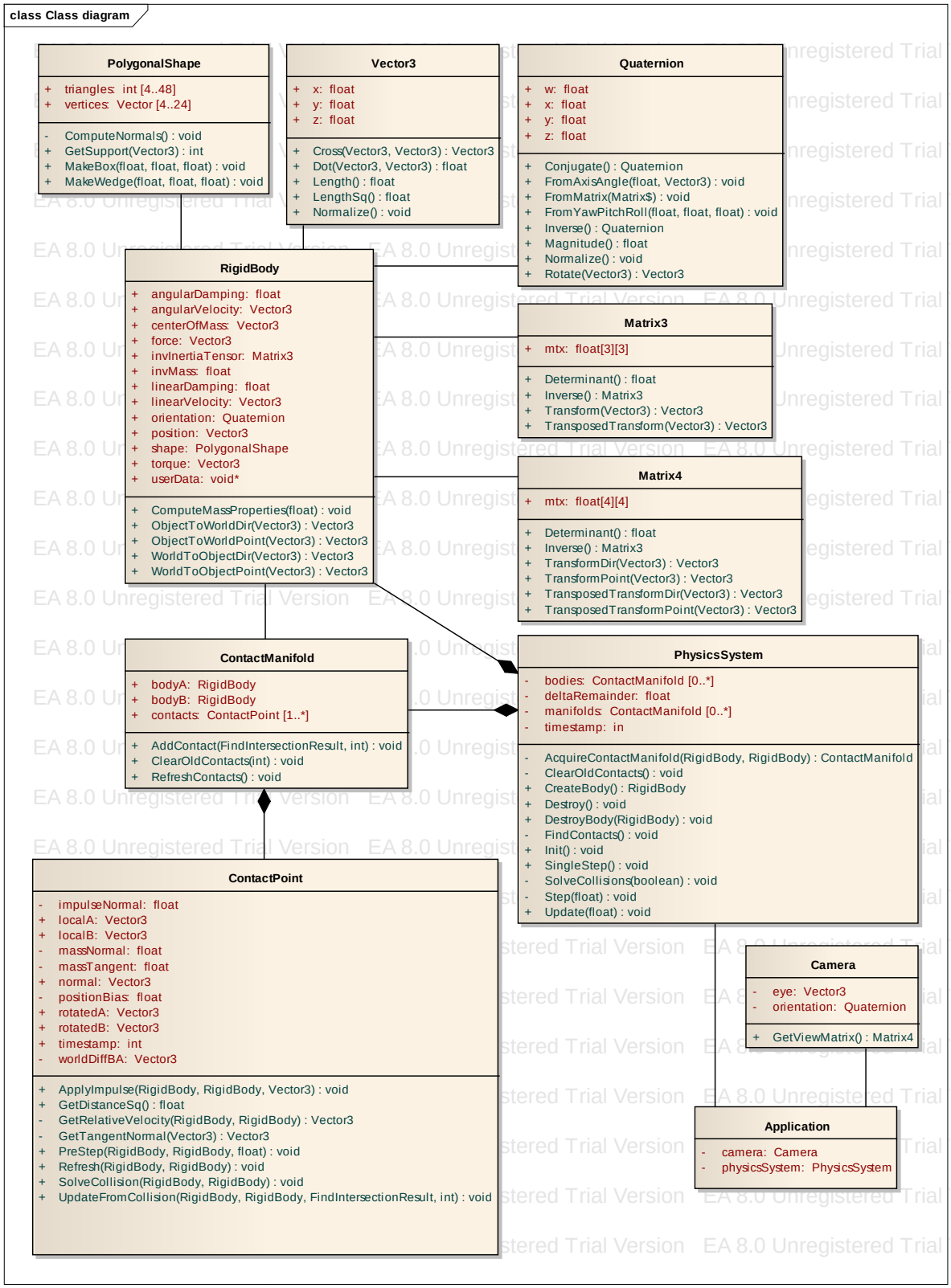


Figura 6 – Diagrama de classes

3.2.2.1 Classe `Vector3`

A classe `Vector3` pode representar um ponto ou direção no espaço tridimensional. Esta classe utiliza sobrecarga de operadores para implementar soma e subtração entre dois vetores e multiplicação e divisão entre um vetor e um escalar. Há métodos que realizam operações comuns sobre vetores como normalização e cálculo da magnitude. Também são disponibilizadas as operações de produto escalar e produto vetorial através dos métodos `Dot` e `Cross`, respectivamente.

3.2.2.2 Classe `Quaternion`

A classe `Quaternion` representa uma orientação no espaço tridimensional como descrito na seção 2.6. Esta classe utiliza sobrecarga de operadores para implementar soma e subtração entre dois quaternions e multiplicação e divisão entre um quaternion e um escalar. Em especial também foi criado um operador de multiplicação entre dois quaternions. Esta operação resulta em um novo quaternion que é a combinação das duas rotações.

Esta classe disponibiliza o método `Slerp` para interpolar duas rotações. Também são disponibilizados métodos para converter entre as demais formas de representar orientações. Isto é, matrizes e ângulos de Euler. Por fim, também é disponibilizado um método para normalizar o quaternion e assim garantir que ele tenha comprimento unitário.

3.2.2.3 Classes `Matrix3` e `Matrix4`

As classes `Matrix3` e `Matrix4` representam matrizes de tamanho 3x3 e 4x4 respectivamente. Ambas disponibilizam operações comuns em matrizes, como cálculo de determinante, cálculo da matriz transposta e cálculo da matriz inversa. Estas classes são utilizadas para transformar vetores de um espaço vetorial para outro espaço vetorial. Portanto ambas também disponibilizam métodos `Transform` e `TransposedTransform`. Com estes métodos é possível transformar um vetor para certo espaço vetorial e de volta para o espaço vetorial original se for necessário.

A diferença entre as classes é que a primeira classe é utilizada para realizar

transformações lineares. Já a segunda classe realiza uma transformação linear seguida de uma translação. Ao transformar um vetor que representa uma direção, a translação não deve ser aplicada, e por este motivo há duas versões de cada método de transformação. Uma versão para transformar pontos e outra versão para transformar direções.

3.2.2.4 Classe `PolygonalShape`

A classe `PolygonalShape` representa uma coleção de vértices e faces. Esta classe é utilizada para representar a forma física de um objeto. Seu método mais importante, `GetSupport`, é utilizado pelo algoritmo de detecção de colisão para buscar um ponto numa direção. Este método é essencialmente a função de mapeamento de suporte discutida na seção 2.2.1.1. A classe também contém métodos auxiliares para criar formas geométricas pré-definidas, como caixas e prismas triangulares, e um método para calcular as normais das faces resultantes.

3.2.2.5 Classe `RigidBody`

A classe `RigidBody` representa um corpo rígido da simulação. Consequentemente, esta classe possui vários atributos necessários para a simulação, como forma, posição, orientação, velocidade linear e angular, massa, momento de inércia, entre outros. Os campos de posição e orientação definem um espaço vetorial da mesma forma que uma matriz de transformação. Portanto a classe disponibiliza métodos auxiliares para transformar vetores de e para este espaço vetorial.

Por fim, esta classe oferece o método `ComputeMassProperties` para calcular automaticamente a massa, o centro de massa e o momento de inércia. Este método calcula estas propriedades utilizando a forma geométrica e também uma densidade que deve ser informada como parâmetro.

3.2.2.6 Classe `PhysicsSystem`

Esta é a principal classe do sistema e tem a responsabilidade de gerenciar toda a simulação de física. Os métodos `Init` e `Destroy` são disponibilizados para iniciar e finalizar a simulação, enquanto os métodos `CreateBody` e `Destroy` permitem criar ou destruir objetos da simulação.

A simulação pode ser executada através do método `SingleStep` ou do método `Update`. O primeiro método pode ser utilizado para visualizar a simulação passo-a-passo. O segundo método recebe um número de ponto flutuante indicando a quantidade de segundos que devem ser simulados.

Esta classe, além de manter uma coleção de corpos sendo simulados, também mantém uma lista dos pontos de contato detectados. Desta forma, informações sobre o impulso aplicado em cada contato podem ser guardadas entre uma iteração da simulação e a próxima. Isto é essencial para o tratamento das colisões.

3.2.2.7 Classe `ContactManifold`

A classe `ContactManifold` representa uma coleção de pontos de contato entre dois corpos rígidos. Esta classe disponibiliza métodos para adicionar novos pontos de contato detectados, atualizar pontos existentes conforme os corpos se movem, ou remover pontos existentes caso os objetos não estejam mais em contato num determinado ponto.

Ao adicionar um ponto através do método `AddContactPoint`, apenas os pontos mais relevantes para a simulação são mantidos. Isto melhora tanto o desempenho quanto a estabilidade da simulação.

3.2.2.8 Classe `ContactPoint`

A classe `ContactPoint` representa um ponto de contato entre dois corpos rígidos e possui vários atributos necessários para o tratamento da colisão. Por exemplo, os pontos de colisão em cada objeto, o vetor normal da colisão, o impulso aplicado na iteração anterior, um inteiro representando a idade do contato, entre outros.

Esta classe é responsável por calcular o impulso e o atrito necessários para tratar uma colisão. O tratamento da colisão é feito através dos métodos `PreStep` e `SolveCollision`. O primeiro método é chamado uma vez no início da etapa de tratamento de colisões de cada iteração da simulação. O segundo método é chamado várias vezes dentro de uma mesma iteração da simulação, e efetivamente aplica os valores calculados.

Conforme os corpos rígidos se movem é necessário atualizar os pontos de contato. Isto é feito através dos métodos `UpdateFromCollision` e `Refresh`. O primeiro método é utilizado quando um novo ponto de contato é detectado mas está muito próximo de um ponto já existente. O segundo método é chamado a cada iteração da simulação para atualizar todas as informações do contato, independentemente de um novo ponto ter sido detectado ou não.

3.2.2.9 Classes `Application` e `Camera`

As classes `Application` e `Camera` não fazem parte do motor de física em si, e sim da aplicação exemplo. Elas são apresentadas aqui apenas para demonstrar como uma aplicação interage com o motor de física.

3.2.3 Diagrama de atividades

Na Figura 7 é possível visualizar como o motor de física é integrado no laço principal da aplicação exemplo. É comum encontrar em jogos um laço principal que segue este padrão.

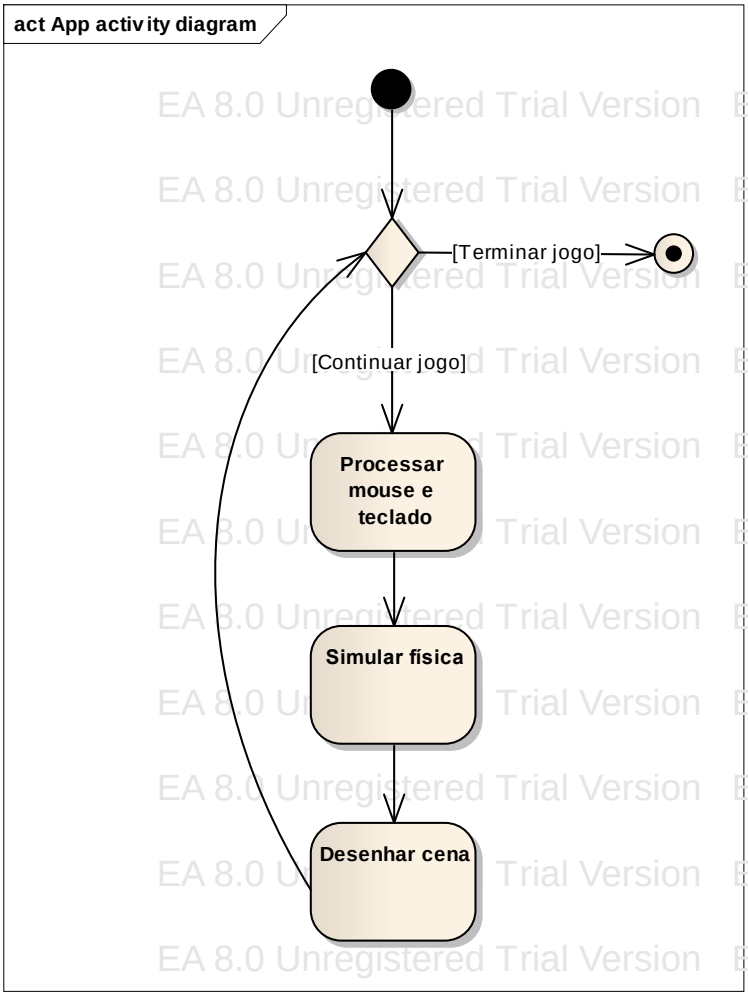


Figura 7 – Diagrama de atividades da aplicação exemplo

A Figura 8 detalha uma iteração da simulação de física. As etapas apresentadas no diagrama são detalhadas na seção 3.3.1.

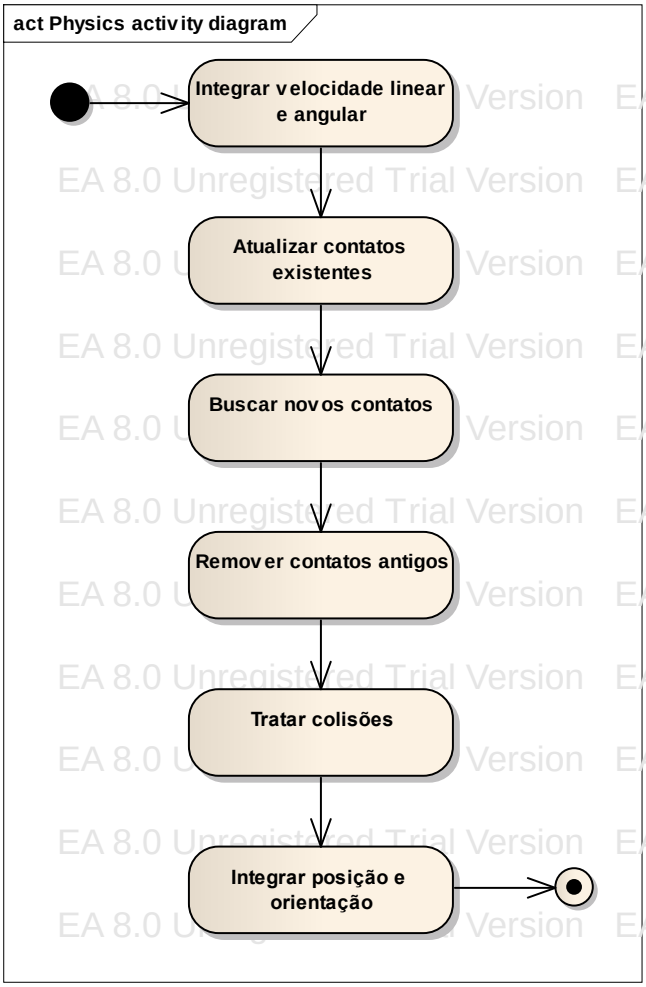


Figura 8 – Diagrama de atividades do motor de física

3.2.4 Diagrama de sequência

A Figura 9 também detalha uma iteração da simulação de física, mas demonstrando a iteração entre os componentes do sistema. Os detalhes da interação entre estes componentes são apresentados na seção 3.3.1.

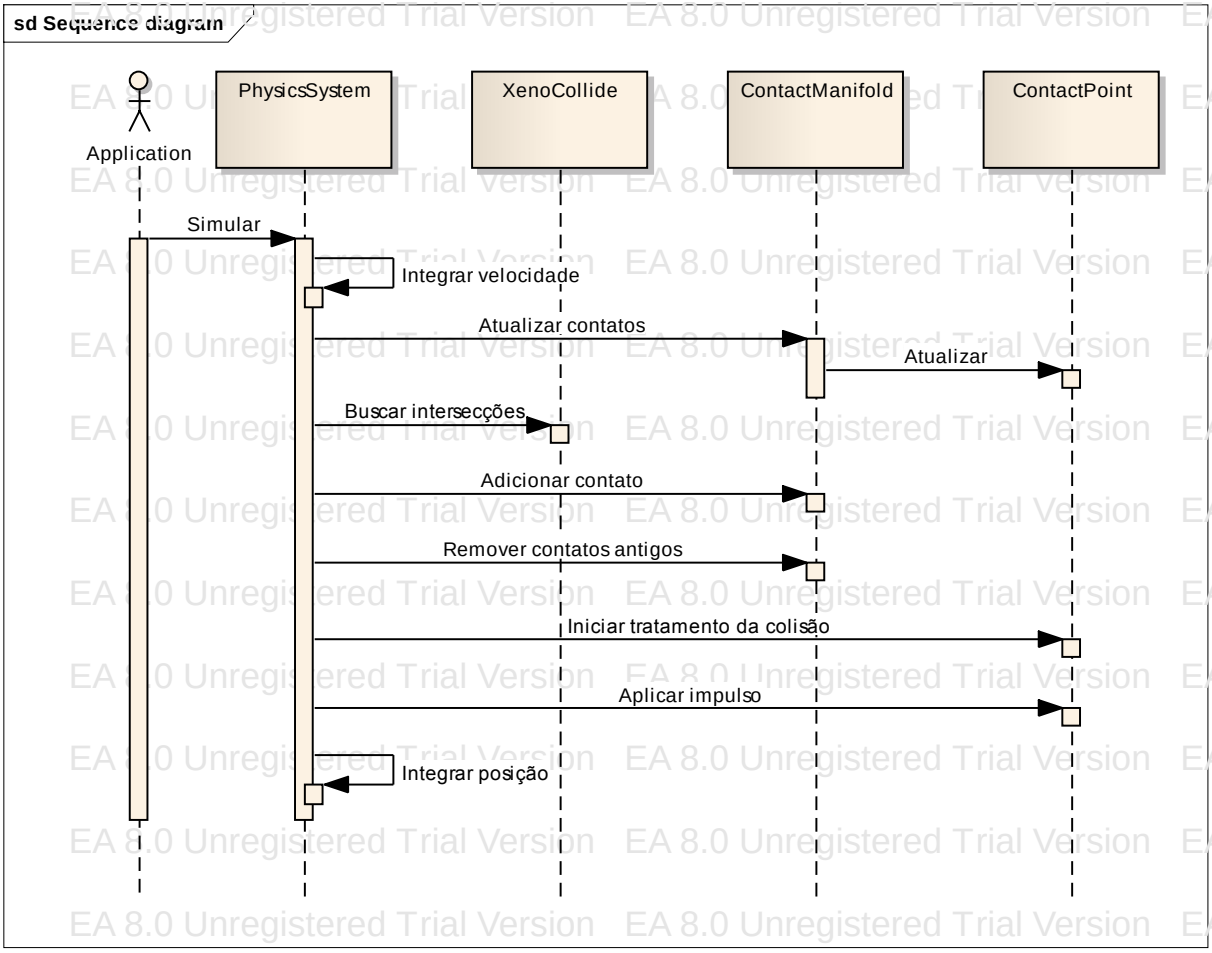


Figura 9 – Diagrama de sequência

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da aplicação exemplo.

3.3.1 Técnicas e ferramentas utilizadas

Para a implementação da ferramenta foi utilizada a linguagem de programação C++ e a biblioteca gráfica OpenGL 1.1. O ambiente de desenvolvimento escolhido foi o Visual Studio 2010.

3.3.1.1 Detecção de colisão

A detecção de colisão é realizada pela função `Geometry_FindIntersection`. Esta função implementa o algoritmo XenoCollide descrito na seção 2.2.1, e seu código fonte é apresentado no Apêndice A.

A função de mapeamento de suporte, que é essencial para o algoritmo de detecção, é implementada no método `GetSupport` da classe `PolygonalShape`. Seu código fonte é apresentado no Quadro 14. Este método utiliza o produto escalar para procurar o vértice mais distante na direção informada.

```
int PolygonalShape::GetSupport( const Vector3& dir ) const
{
    int bestIndex = 0;
    float bestValue = Vector3::Dot( this->vertices[0], dir );

    for( int i = 1; i < this->vertexCount; ++i )
    {
        float value = Vector3::Dot( this->vertices[i], dir );

        if( value > bestValue )
        {
            bestIndex = i;
            bestValue = value;
        }
    }

    return bestIndex;
}
```

Quadro 14 – Código fonte do método `GetSupport`

3.3.1.2 Pontos de contato

Ao detectar uma colisão as informações da colisão são adicionadas numa coleção de pontos de contato entre dois objetos. Desta forma é possível persistir informações da colisão entre uma iteração da simulação e a próxima. Isto é essencial para o tratamento da colisão.

Porém não é necessário guardar todos os pontos de contato detectados. Parte disto é uma questão de desempenho. Quanto mais pontos de contato, mais tempo será gasto na etapa de tratamento da colisão. Outro motivo é que uma grande quantidade de pontos não vai necessariamente contribuir para o realismo da simulação, e dependendo da forma como o tratamento é implementado, pode até gerar resultados piores.

Por estes motivos, a classe `ContactManifold` implementa um mecanismo para manter

no máximo quatro pontos. Este mecanismo foi originalmente desenvolvido por Coumans (2010). Primeiramente, procura-se o ponto existente mais próximo do novo ponto. Se a distância entre eles for menor do que um valor mínimo, as informações do ponto existente serão mescladas com o novo ponto. Caso nenhum ponto existente esteja próximo o suficiente e o limite de pontos já foi atingido, um dos pontos existentes ou o novo ponto será descartado. Como primeira regra, o ponto de penetração mais profunda sempre é mantido. Os demais pontos serão escolhidos de forma a maximizar a área de contato. O ponto que não contribui para maximizar a área é descartado. Este mecanismo é implementado, em parte, no método `AddContact`, como pode ser visto no Quadro 15.

```
void ContactManifold::AddContact( const FindIntersectionResult&
collisionInfo, UINT timestamp )
{
    ContactPoint newContact;
    newContact.UpdateFromCollision( this->pBodyA, this->pBodyB,
collisionInfo, timestamp );

    int iInsertIndex = this->FindContactForUpdate( newContact );

    if( iInsertIndex >= 0 )
    {
        this->contacts[iInsertIndex].UpdateFromCollision( this->pBodyA,
this->pBodyB, collisionInfo, timestamp );
        return;
    }

    if( this->nContacts == MAX_MANIFOLD_CONTACTS )
    {
        iInsertIndex = this->SortCachedPoints( newContact );
    }
    else
    {
        iInsertIndex = this->nContacts++;
    }

    this->contacts[iInsertIndex] = newContact;
}
```

Quadro 15 – Código fonte do método `GetSupport`

Pontos são criados quando uma colisão é detectada, e precisam ser destruídos quando os corpos não estão mais em contato. Os contatos não são destruídos imediatamente, mas apenas quando os pontos de contato originais de cada objeto afastarem-se o suficiente. Isto é necessário porque ao tratar a colisão de um objeto empilhado sobre outro, é possível que os objetos se afastem levemente e não estejam estritamente em contato. Sem esta tolerância os contatos seriam destruídos e criados novamente a cada quadro. O efeito visual seria um objeto constantemente quicando ao invés de ficar apoiado de forma estável sobre outro. O código para destruir contatos antigos é implementado no método `ClearOldContact`, e é apresentado no Quadro 16.

```
void ContactManifold::ClearOldContacts( UINT currentTimestamp )
{
    for( int i = 0; i < this->nContacts; )
    {
        ContactPoint& contact = this->contacts[i];

        if( contact.timestamp != currentTimestamp &&
contact.GetDistanceSq() > CONTACT_BREAKING_THRESHOLD_SQ )
            this->RemoveContact( i );
        else
            ++i;
    }
}
```

Quadro 16 – Código fonte do método `ClearOldContact`

3.3.1.3 Tratamento da colisão

O tratamento da colisão é realizado para todos os pontos de contato de todos os pares de objetos em colisão. Isto significa que em situações onde um objeto está em colisão com outros dois objetos, ou há mais de um contato entre os objetos, as colisões são consideradas apenas individualmente, e não de forma global. Para chegar numa solução global que resolva as colisões para todos os objetos e pontos de contato, deve-se realizar várias iterações do tratamento individual. Isto é necessário pois ao tratar o segundo par de objeto em colisão, os resultados calculados anteriormente para o primeiro par são alterados. Este processo é feito no método `SolveCollisions`, como pode ser visto no Quadro 17. O tratamento é dividido em duas partes. A primeira parte calcula valores que não serão alterados entre as várias iterações. A segunda parte é executada várias vezes para aproximar a solução global para todos os contatos, utilizando os valores calculados anteriormente.

```
void PhysicsSystem::SolveCollisions( float step )
{
    for each( ContactManifold* pManifold in m_Manifolds )
    {
        for( int i = 0; i < pManifold->nContacts; ++i )
        {
            pManifold->contacts[i].PreStep( pManifold->pBodyA, pManifold-
>pBodyB, step );
        }
    }

    for( int iter = 0; iter < CONTACT_SOLVER_ITERATIONS; ++iter )
    {
        for each( ContactManifold* pManifold in m_Manifolds )
        {
            for( int i = 0; i < pManifold->nContacts; ++i )
            {
                pManifold->contacts[i].SolveCollision( pManifold->pBodyA,
pManifold->pBodyB );
            }
        }
    }
}
```

Quadro 17 – Código fonte do método `SolveCollisions`

O tratamento consiste em modificar a velocidade dos corpos envolvidos. Para tanto, deve-se calcular dois valores, o impulso normal e o impulso tangencial. O primeiro é o impulso que é aplicado na direção da normal da colisão. Isto impede que os objetos permaneçam intersectando. O segundo impulso é aplicado numa direção perpendicular à normal e simula atrito entre os objetos. A primeira parte mencionada anteriormente e que calcula alguns valores necessários para determinar ambos os impulsos tem seu código fonte no Quadro 18.

```
void ContactPoint::PreStep( RigidBody* pBodyA, RigidBody* pBodyB, float
step )
{
    // Compute the positional constraint error (scaled by the Baumgarte
coefficient 'beta')
    float flBeta = 0.50f / step;
    m_flPositionBias = Vector3::Dot( flBeta * m_vecWorldDiffBA, this-
>normal );

    // Add a boundary layer to the position error -- this will ensure the
objects remain in contact
    m_flPositionBias -= 1.0f;

    // This represents how much angular velocity we get for every unit of
momentum transferred along the constraint direction
    Vector3 vecAngularComponentA = pBodyA->invInertiaTensor.Transform(
Vector3::Cross( this->rotatedA, this->normal ) );
    Vector3 vecAngularComponentB = pBodyB->invInertiaTensor.Transform(
Vector3::Cross( this->rotatedB, this->normal ) );

    Vector3 relativeVelocity = GetRelativeVelocity( pBodyA, pBodyB );
    Vector3 tangentNormal = GetTangentNormal( relativeVelocity );

    m_flMassNormal = 1.0f / (pBodyA->invMass + pBodyB->invMass +
Vector3::Dot( this->normal, Vector3::Cross( vecAngularComponentA, this-
>rotatedA ) + Vector3::Cross( vecAngularComponentB, this->rotatedB ) ));
    m_flMassTangent = 1.0f / (pBodyA->invMass + pBodyB->invMass +
Vector3::Dot( tangentNormal, Vector3::Cross( vecAngularComponentA, this-
>rotatedA ) + Vector3::Cross( vecAngularComponentB, this->rotatedB ) ));

    this->ApplyImpulse( pBodyA, pBodyB, m_flImpulseNormal * this->normal
);
}
```

Quadro 18 – Código fonte do método `PreStep`

O método `GetTangentNormal` é responsável por calcular a direção do impulso tangencial, como pode ser visto no Quadro 19. Calcular esta direção não é inteiramente simples. Em três dimensões existem infinitas direções perpendiculares a um vetor, efetivamente formando um plano. Para escolher a direção correta levamos em consideração a velocidade relativa dos objetos no ponto de contato. Caso os objetos estejam parados ou esta velocidade seja muito pequena, uma direção qualquer é escolhida.

```
Vector3 ContactPoint::GetTangentNormal( const Vector3& relativeVelocity )
const
{
    Vector3 vecTangentNormal = relativeVelocity - Vector3::Dot(
relativeVelocity, this->normal ) * this->normal;
    float flMagnitudeSq = vecTangentNormal.LengthSq();

    if( flMagnitudeSq <= FLT_EPSILON )
    {
        Vector3 temp;
        Math_PlaneSpace( this->normal, vecTangentNormal, temp );
    }
    else
    {
        vecTangentNormal /= sqrt( flMagnitudeSq );
    }

    return vecTangentNormal;
}
```

Quadro 19 – Código fonte do método `GetTangentNormal`

Por fim o trecho de código que calcula e aplica o impulso normal e o impulso tangencial pode ser visto no Quadro 20. Inicialmente calcula-se a velocidade relativa no ponto de contato. Para calcular o impulso normal, o produto escalar é utilizado para obter a parte dessa velocidade que está na direção da normal da colisão. O processo para calcular o impulso tangencial é similar, basta calcular a parte da velocidade que está na direção da tangente.

```
void ContactPoint::SolveCollision( RigidBody* pBodyA, RigidBody* pBodyB )
{
    Vector3 relativeVelocity = GetRelativeVelocity( pBodyA, pBodyB );

    // Project the relative velocity onto the constraint direction
    float flNormalVelocity = Vector3::Dot( relativeVelocity, this->normal
);
    float flNormalImpulse = m_flMassNormal * (-flNormalVelocity -
m_flPositionBias);

    // Clamp normal impulse
    flNormalImpulse = min( flNormalImpulse, -m_flImpulseNormal );
    m_flImpulseNormal += flNormalImpulse;

    // Apply normal impulse
    this->ApplyImpulse( pBodyA, pBodyB, flNormalImpulse * this->normal );

    if( m_flMassTangent <= 0 )
        return;

    relativeVelocity = GetRelativeVelocity( pBodyA, pBodyB );
    Vector3 tangentNormal = GetTangentNormal( relativeVelocity );

    // Compute friction impulse
    float flTangentVelocity = Vector3::Dot( relativeVelocity,
tangentNormal );
    float flTangentImpulse = m_flMassTangent * (-flTangentVelocity);

    const float STD_FRICTION = 0.05f;

    // Clamp tangential impulse
    flTangentImpulse = max( flTangentImpulse, STD_FRICTION *
m_flImpulseNormal );

    // Apply tangential impulse
    this->ApplyImpulse( pBodyA, pBodyB, flTangentImpulse * tangentNormal
);
}
```

Quadro 20 – Código fonte do método `SolveCollision`

3.3.1.4 Atualização velocidade e posição

O método de integração implementado neste trabalho é o NSV, como descrito na seção 2.4. A sua implementação pode ser vista no Quadro 21. O primeiro passo consiste em integrar a força e o torque aplicados no objeto, desta forma alterando a velocidade linear e velocidade angular,

respectivamente. O segundo passo, que é realizado após o tratamento das colisões, consiste em integrar a velocidade linear e a velocidade angular, desta forma alterando a posição e a orientação.

```
void PhysicsSystem::Step( float step )
{
    ++m_iTimestamp;

    // Integrate velocity and apply damping
    for each( RigidBody* pBody in m_Bodies )
    {
        if( pBody->invMass == 0 )
            continue;

        pBody->linearVelocity += pBody->invMass * pBody->force * step;
        pBody->angularVelocity += pBody-
>invInertiaTensor.TransposedTransform( pBody->torque ) * step;

        ...
    }

    ...

    for each( RigidBody* pBody in m_Bodies )
    {
        pBody->position += pBody->linearVelocity * step;

        pBody->orientation += (pBody->orientation * Quaternion( 0, pBody-
>angularVelocity )) * step * 0.5f;
        pBody->orientation.Normalize();
    }
}
```

Quadro 21 – Código fonte do método `Step`

3.3.1.5 *Timestep*

Neste trabalho foi utilizado um *timestep* fixo. Especificamente, a simulação sempre é avançada em intervalos de 0,033 segundos. Como descrito na seção 2.4.1, foi necessário implementar um mecanismo para desacoplar a simulação de física da atualização de tela. Caso a aplicação esteja iterando mais rapidamente do que a simulação de física necessita, o tempo de cada iteração vai para um acumulador. A simulação de física apenas é executada quando este acumulador é maior ou igual ao tempo do *timestep*. No caso oposto, onde a aplicação está iterando muito lentamente, serão feitas várias iterações de física e a tela será desenhada apenas uma vez. No pior caso, quando a simulação de física gasta mais de um segundo real para simular um segundo no mundo virtual, a aplicação simplesmente não irá executar em tempo real. O código deste mecanismo está disponível no Quadro 22.

```
void PhysicsSystem::Update( float delta )
{
    m_flDeltaRemainder += delta;

    int nSteps = static_cast<int>( m_flDeltaRemainder / FIXED_TIME_STEP );

    for( int i = 0; i < nSteps; ++i )
        Step( FIXED_TIME_STEP );

    m_flDeltaRemainder -= nSteps * FIXED_TIME_STEP;
}
```

Quadro 22 – Código fonte do método `Update`

3.3.1.6 Propriedades da massa

A massa, centro de massa e momento de inércia são essenciais para o tratamento das colisões. Todas essas propriedades podem ser calculadas a partir da forma física do corpo rígido. Portanto, para diminuir o trabalho do desenvolvedor que utilizar o motor de física, foi disponibilizado o método `ComputeMassProperties`. Este método deve ser chamado depois que a forma geométrica do corpo rígido estiver definida, e deve receber a densidade do objeto como parâmetro. O código deste método pode ser visto no Apêndice B.

Para calcular o momento de inércia é feita uma integral sobre as posições dos vértices. Para calcular a massa e centro de massa, a forma geométrica é tratada como vários tetraedros. Cada tetraedro é formado por uma face da forma geométrica e pela origem. Calcula-se o volume do tetraedro, e com a densidade informada é possível calcular sua massa. A massa do corpo rígido é igual à soma das massas dos tetraedros. E a média ponderada dos centros de massa de cada tetraedro resulta no centro de massa do corpo rígido.

O Quadro 23 demonstra como calcular várias propriedades para um tetraedro que tem como um de seus vértices a origem. Em primeiro lugar é calculado o volume a partir dos vértices. Em segundo lugar a massa é calculada com base no volume e na densidade `d` do objeto. E por fim o centro de massa do tetraedro é calculado fazendo a média dos seus vértices. Já o Quadro 26 demonstra como as propriedades do objeto original são calculadas a partir das propriedades dos tetraedros. A massa é representada por `M` e o centro de massa é representado por `C`.

$$\mathrm{V} = \frac{|\boldsymbol{a} \cdot \boldsymbol{b} \times \boldsymbol{c}|}{6}$$
$$M = V \cdot d$$
$$\boldsymbol{C} = \frac{\boldsymbol{a} + \boldsymbol{b} + \boldsymbol{c}}{4}$$

Quadro 23 – Volume, massa e centro de massa de um tetraedro

$$M = \sum M_i$$
$$\boldsymbol{C} = \frac{\sum \boldsymbol{C_i} \cdot M_i}{\sum M_i}$$

Quadro 24 – Centro de massa de um objeto

3.3.2 Operacionalidade da implementação

Esta seção tem por objetivo mostrar a operacionalidade da implementação em nível de usuário e para tanto aborda as principais funcionalidades do aplicativo exemplo.

Inicialmente a aplicação inicia em um mundo vazio, sem nenhum objeto sendo simulado. O usuário deve então escolher um dos cenários pré-definidos de exemplo. Para isto basta pressionar uma das teclas de 1 a 8 no teclado. Cada número corresponde a um cenário diferente.

Os cenários 1 até 6 servem para demonstrar que colisões de todos os tipos possíveis. Cada cenário testa, respectivamente, colisão vértice-vértice, vértice-aresta, vértice-face, aresta-aresta, aresta-face e face-face. O primeiro, com colisão vértice-vértice, cenário pode ser observado na Figura 10.

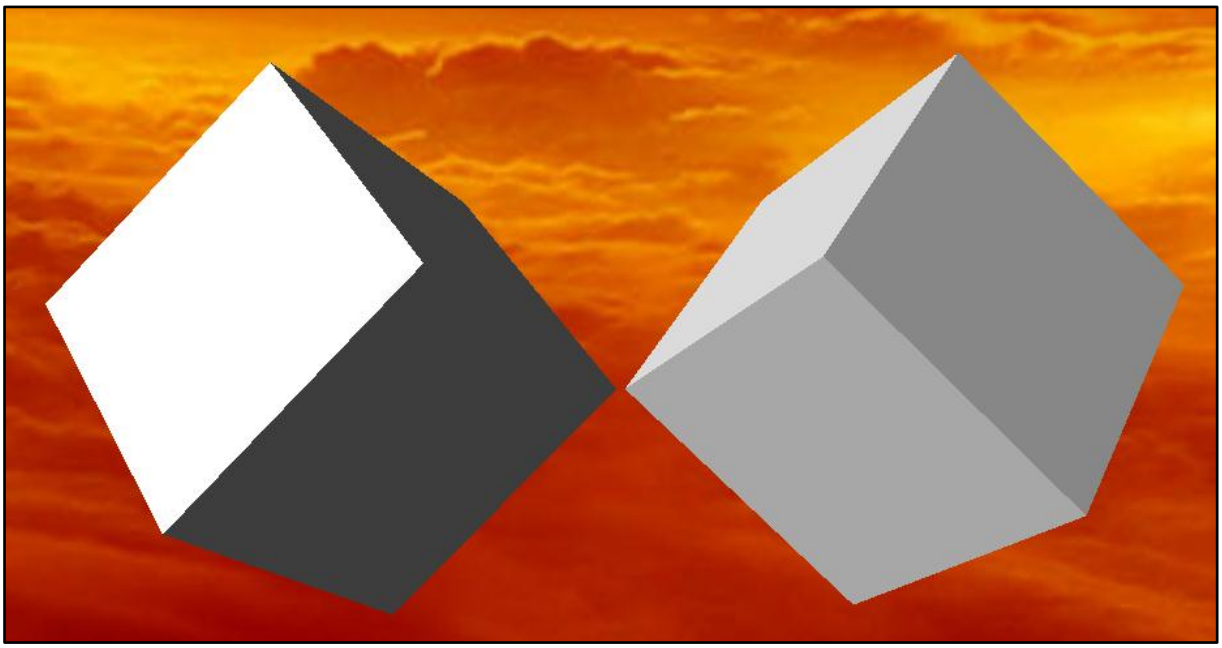


Figura 10 – Colisão vértice-vértice do cenário 1

O cenário 7 demonstra um bloco em cima de uma tábua, que por sua vez está apoiada sobre um prisma retangular, e este finalmente está em contato com o chão. Este cenário demonstra o tratamento de colisão entre vários objetos, e pode ser observado na Figura 11.

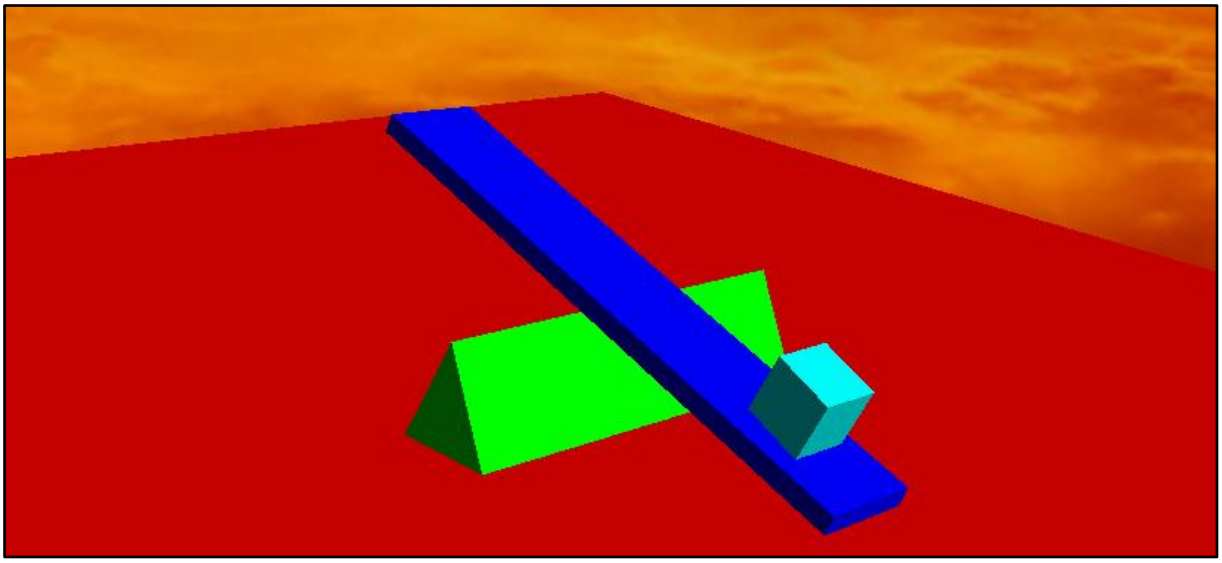


Figura 11 – Cenário 7

Por fim o cenário 8 demonstra o tratamento de colisão entre um número ainda maior de objetos. Neste caso várias peças de dominó estão enfileiradas. Após derrubar a primeira peça, as outras vão caindo em sequência. Este cenário pode ser observado na Figura 12. No Apêndice C é possível ver uma sequência de imagens que demonstra a simulação do cenário 7 e 8.

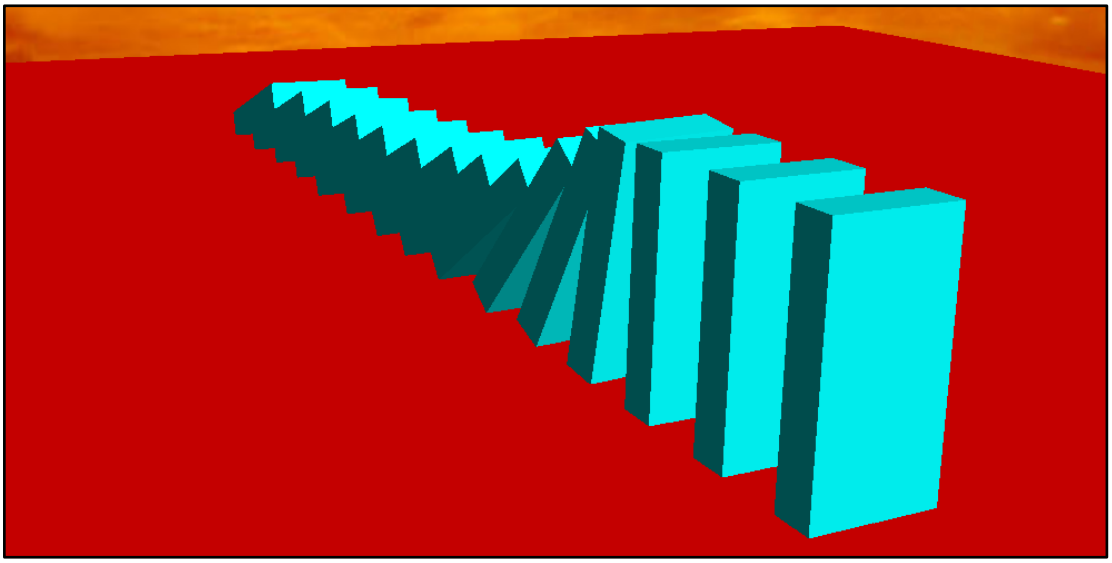


Figura 12 – Cenário 8

O usuário também pode mover a câmera pelo mundo simulado para poder visualizar a simulação de qualquer posição e ângulo. As teclas `W`, `A`, `S` e `D` movem a câmera para frente, para a esquerda, para trás e para a direita, respectivamente. As teclas `Z` e `X` movem a câmera para cima e para baixo no eixo Y. As teclas `Q` e `E` rotacionam a câmera ao redor do eixo de visão. Por fim, o mouse pode ser usado para mover a direção para onde a câmera aponta.

Em todos os cenários, é possível executar a simulação em modo passo-a-passo. Para

isto basta pressionar a barra de espaço. A simulação será avançada um passo a cada vez que o usuário pressionar a barra de espaço novamente. Mesmo neste modo passo-a-passo, ainda é possível mover a câmera normalmente. Isto permite visualizar uma colisão em detalhes e de qualquer ângulo. Para retornar ao modo normal de execução basta pressionar a tecla `F5`. A qualquer momento é possível sair da aplicação pressionando a tecla `ESC`.

Por fim, a todo instante é atualizado no canto superior esquerdo a quantidade de quadros por segundo. Isto pode ser visto na Figura 13.

Figura 13 – Quadros por segundo

3.4 RESULTADOS E DISCUSSÃO

Durante a implementação houve situações onde foi necessário escolher entre uma técnica ou outra. Uma dessas situações foi a escolha do método de integração. O método implementado foi o NSV devido a vários motivos. Primeiramente, este método é apropriado para situações onde a aceleração e o torque são constantes. Na aplicação exemplo a única força presente é a gravidade, que pode ser considerada constante para fins práticos. Em segundo lugar, sua implementação é extremamente simples, tão simples quanto o método de Euler. E por fim, o método NSV é muito eficiente, o que o torna apropriado para jogos.

Outra situação que apresentou uma escolha ocorreu durante a implementação da detecção de colisões. O algoritmo XenoCollide foi escolhido pois Snethen (2008) disponibiliza uma implementação deste algoritmo que já trata os principais casos degenerados. Entre os outros algoritmos que foram considerados mas não foram escolhidos estão *Separating Axis Theorem* (SAT) e Gilbert-Johnson-Keerthi (GJK). O algoritmo SAT, apesar de ser simples de implementar, não apresenta bom desempenho. O algoritmo GJK é similar ao XenoCollide, ambos procuram a origem dentro da diferença de Minkowski. Apenas

a forma de procurar a origem é diferente. Porém não é possível obter informações sobre a colisão diretamente com o GJK. É necessário usar um segundo algoritmo, *Expanding Polytope Algorithm* (EPA). Este segundo algoritmo é complexo e difícil de implementar.

Durante os testes do motor de física, atentou-se para três características. São elas: robustez, uso de memória e desempenho. Para testar robustez na detecção de colisão foram criados cenários para todos os tipos possíveis de colisão entre objetos. Estes tipos são: vértice-vértice, vértice-aresta, vértice-face, aresta-aresta, aresta-face e face-face. Em todos estes casos as colisões foram detectadas e tratadas corretamente. Porém a detecção é feita de forma discreta, não contínua, como descrito na seção 2.2. Para evitar o efeito *tunneling* foi colocado um limite máximo nas velocidades linear e angular dos objetos simulados.

Apesar da detecção da colisão ser robusta, o tratamento da colisão não é. A simulação é muito sensível a pequenas variações. Por exemplo, corpos rígidos iguais, mas colocados em posições diferentes sobre um plano, se comportam de forma diferente. Pequenos erros de precisão também geram grandes diferenças na simulação dos objetos. O processo de tratamento de colisão deve ser revisto em relação à robustez.

Já o uso de memória é influenciado por dois fatores distintos, o número de objetos sendo simulados e o número de objetos em contato. A quantidade de contatos entre os objetos em si não influencia, pois esta quantidade está sempre limitada a quatro. Ao detectar colisão entre um par de objetos, já é reservada memória para quatro contatos. A memória ocupada por um corpo rígido é aproximadamente 1.274 bytes. A maior parte desta memória, 1.064 bytes, é destinada para a representação geométrica do objeto. Isto é, vértices, faces e outras informações necessárias para aplicar iluminação ao objeto. Esta informação não é diretamente necessária para a simulação de física, apenas para desenhar o objeto. Por exemplo, corpos que podem ser definidos parametricamente como esferas, cápsulas ou cones poderiam ocupar bem menos memória. Já o uso de memória para cada par de corpos em contato é aproximadamente 380 bytes. Apesar de ser consideravelmente menos memória, a quantidade de corpos colidindo ao mesmo tempo ainda tem certa influência na utilização de memória. O Quadro 25 e a Figura 14, que está em escala logarítmica, demonstram a utilização de memória. Como esperado, esta utilização de memória cresce linearmente em função do número de objetos sendo simulados.

Corpos rígidos	Memória utilizada (KB)
10	12
100	136
1.000	1.320
10.000	13.156
100.000	131.512

Quadro 25 – Utilização de memória

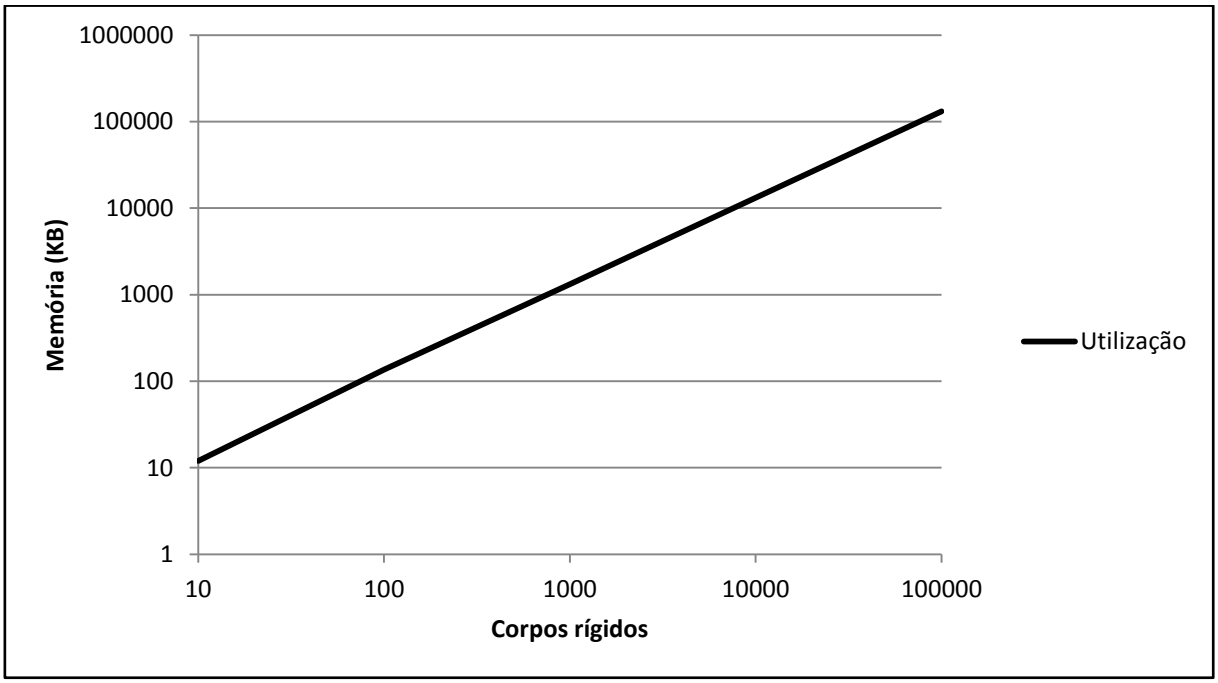


Figura 14 – Utilização de memória

Já o desempenho da aplicação também depende do número de objetos e do número de contatos entre estes objetos. O algoritmo de detecção de colisão tem complexidade assimptótica $O(N^2)$, pois precisa testar todos os pares de objetos. O Quadro 26 e a Figura 15 demonstram o tempo necessário para realizar uma iteração da simulação, e é possível perceber claramente a natureza quadrática do algoritmo.

Corpos rígidos	Tempo (segundos)
500	1,72
1000	6,55
1500	14,79
2000	26,28
2500	41,10
3000	59,31
3500	80,53

Quadro 26 – Tempo para detectar colisões

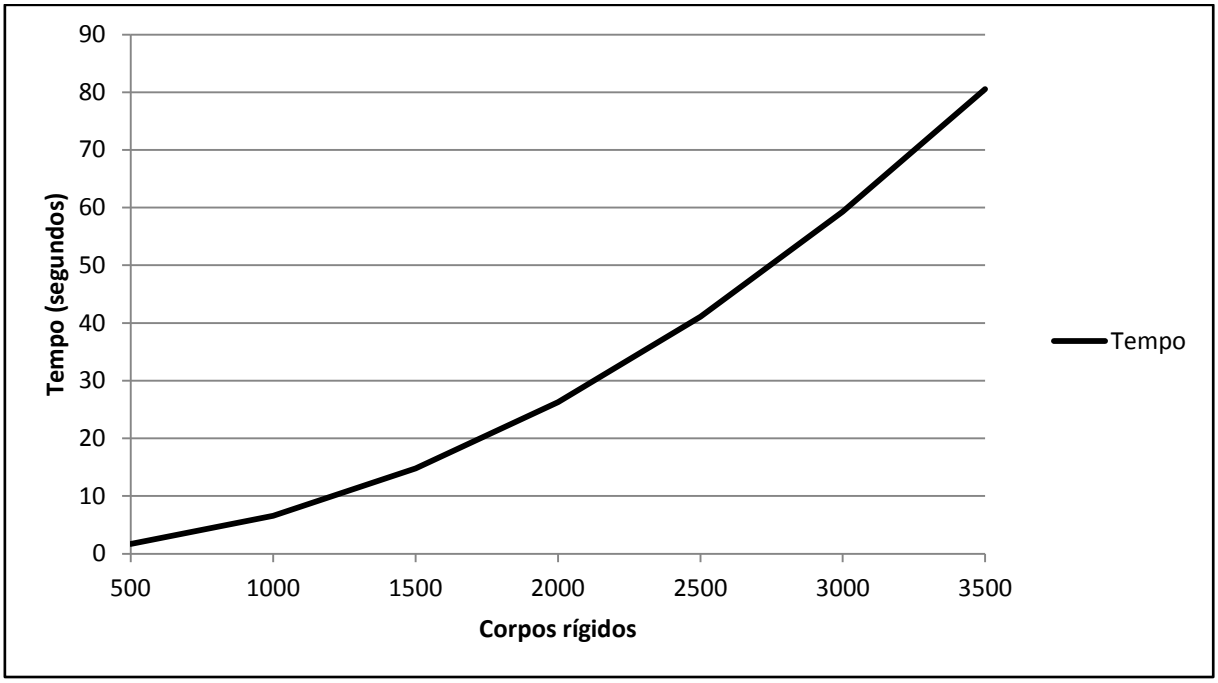


Figura 15 – Tempo para detectar colisões

Há duas formas de melhorar o desempenho da etapa de detecção. Uma forma é diminuir o custo de um teste, e a outra é diminuir o número de testes em si. Para tornar o teste mais rápido, é possível dividir a detecção de colisão em duas etapas, *broad-phase* e *narrow-phase*. A primeira etapa deve realizar testes rápidos e grosseiros, como por exemplo, criar esferas que envolvem os objetos e verificar se estas esferas estão intersectando. Apenas se as esferas estiverem intersectando, é realizado o teste completo de colisão na segunda etapa, utilizando o algoritmo XenoCollide. Outra forma é tornar o algoritmo XenoCollide mais rápido, otimizando a função de mapeamento de suporte. Ao invés de percorrer todos os vértices de um objeto, é possível utilizar uma técnica denominada *hill-climbing* para percorrer apenas alguns vértices. Com esta técnica, a função de mapeamento teria complexidade assimptótica `O(log N)`, onde `N` é o número de vértices.

Já para diminuir o número de testes em si, é possível utilizar técnicas como *sweep and prune*. Outra opção é marcar objetos que não se moveram recentemente como estando num estado adormecido. Objetos que estão dormentes não precisam ser simulados, nem é necessário testar colisões entre objetos neste estado.

4 CONCLUSÕES

O presente trabalho possibilitou desenvolver um motor de física capaz de simular corpos rígidos, desde que sua forma seja um poliedro convexo. O algoritmo XenoCollide foi fundamental para a etapa de detecção de colisões. Com este algoritmo é possível buscar as informações da colisão como vetor normal e ponto de contato. Não apenas isto, mas a implementação do algoritmo também é robusta, sendo capaz de detectar colisões mesmo em condições incomuns como colisão vértice-vértice ou vértice-aresta.

A simulação também segue as leis físicas do mundo real, levando em consideração a massa, o centro de massa e o momento de inércia dos corpos rígidos. O método de tratamento de colisão funciona corretamente para colisões simples e funciona em parte para situações onde os objetos devem permanecer em contato. Para este segundo caso, as colisões são tratadas e os objetos realmente ficam apoiados um sobre o outro, porém o impulso aplicado nem sempre produz um resultado estável ou visualmente realista.

Independentemente disto, o método de integração NSV também foi de grande importância. Este método foi muito apropriado para a simulação, pois é simples a ponto de ser intuitivo e também é muito eficiente.

A aplicação exemplo foi reaproveitada de um trabalho anterior desenvolvido pelo próprio autor deste trabalho. Isto poupou muito esforço, pois as classes de matemática e de câmera, que são razoavelmente complexas, já estavam prontas para serem utilizadas. A aplicação disponibilizada serviu muito bem seu propósito, pois é possível facilmente visualizar a simulação física de vários ângulos. Além disso, a opção de simular passo-a-passo também foi útil durante o desenvolvimento e depuração.

A escolha da linguagem, C++, também se mostrou adequada. A linguagem C++ permite que o desenvolvedor tenha controle absoluto sobre a memória e as operações realizadas pelo programa. Isto tem grande importância numa aplicação que necessita de ótimo desempenho, como este motor de física. Uma desvantagem com relação a linguagens mais modernas é a dificuldade em usar um *profiler*[2] para detectar gargalos de desempenho.

[2] *Profiler* é uma ferramenta para análise dinâmica da execução de um programa. O propósito desta análise é determinar quanto tempo é consumido por cada parte do código.

4.1 EXTENSÕES

Sugerem-se as seguintes extensões para a continuidade do trabalho:

a) reescrever a etapa de tratamento de colisão, substituindo a implementação atual que não é estável. Seria ideal que este novo método de tratamento de colisão conseguisse simular vários objetos empilhados um sobre o outro;
b) realizar mais testes de robustez, envolvendo colisões de objetos muito grandes com objetos muito pequenos e objetos muito pesados com objetos muito leves;
c) melhorar o desempenho do motor de física utilizando técnicas como *hill-climbing*, objetos adormecidos e detecção de colisão separadas em etapas *broad-phase* e *narrow-phase*. Também é possível reescrever as classes de matemática para fazer uso de instruções *Single Instruction Multiple Data* (SIMD);
d) permitir que um desenvolvedor possa escolher o coeficiente de restituição e o coeficiente de atrito entre corpos rígidos;
e) realizar detecção de colisão contínua para objetos que se movem muito rápido;
f) realizar parte da simulação de física na GPU;
g) implementar juntas e restrições, permitindo montar objetos complexos a partir de formas convexas simples;
h) permitir que o motor de física seja mais facilmente utilizado em jogos, disponibilizando mais opções para a aplicação. Por exemplo, adicionar *callbacks* para que a aplicação possa decidir se um par de objetos pode colidir, e adicionar *callbacks* para quando um contato é detectado ou destruído;
i) criar funções de mapeamento para algumas formas geométricas convexas que não são poliedros, como por exemplo esferas, cilindros, ou cones.

REFERÊNCIAS BIBLIOGRÁFICAS

BOURG, David M. **Physics for game developers**. Sebastopol: O'Relly, 2002.

DUNN, Fletcher; PARBERRY, Ian. **3D math primer for graphics and game development**. Plano: Wordware Publishing, 2002.

COUMANS, Erwin. **Bulltet**. [S.l.], 2010. Disponível em: <http://bulletphysics.org/>. Acesso em: 13 dez. 2010.

ERICSON, Christer. **Real-time collision detection**. San Francisco: Morgan Kaufmann, 2005.

FEYNMAN, Richard P.; LEIGHTON, Robert B.; SANDS, Matthew. **The Feynman lectures on physics**. 6th ed. Redwood: Addison-Wesley, 1963. Não paginado.

FIEDLER, Glenn. **Fix your timestep**. [S.l.], 2006. Disponível em: <http://gafferongames.com/game-physics/fix-your-timestep/>. Acesso em: 21 mar. 2010.

FIRTH, Paul. **Minkowski difference**. [S.l.], 2010. Disponível em: <http://www.pfirth.co.uk/minkowski.html>. Acesso em: 10 nov. 2010.

HAVOK. **Havok physics**: battle tested. [Dublin], 2010. Disponível em: <http://www.havok.com/uploads/Havok_Physics_Brief_Mar%209.pdf>. Acesso em: 18 mar. 2010.

HECKER, Chris. Physics, part 3: collision response. **Game Developer Magazine**, Skokie, p. 11-18, Mar. 1997. Disponível em: <http://chrishecker.com/images/e/e7/Gdmphys3.pdf>. Acesso em: 23 mar. 2010.

JEREZ, Julio; SUERO, Alain. **Newton game dynamics**. [S.l.], 2007. Disponível em: <http://newtondynamics.com/forum/newton.php>. Acesso em: 18 mar. 2010.

MACDONALD, James. **The Euler-Cromer method**. [S.l.], 2002. Disponível em: <http://www.physics.udel.edu/~jim/Ordinary%20Differential%20Equations/Euler-Cromer%20Method.htm>. Acesso em: 21 mar. 2010.

NEWTON, Isaac. **Philosophiae naturalis principia mathematica**. London, 1687. Disponível em: <http://astro.if.ufrgs.br/newton/principia.pdf>. Acesso em: 21 mar. 2010.

NVIDIA CORPORATION. **PhysX**. [S.l.], 2010. Disponível em: <http://developer.nvidia.com/object/physx.html>. Acesso em: 18 mar. 2010.

GOMES, Paulo C. R.; PAMPLONA, Vitor F. M3GE: um motor de jogos 3D para dispositivos móveis com suporte a Mobile 3D Graphics API. In: **Proceedings of IV Brazilian Symposium on Computer Games and Digital Entertainment**, 2005, São Paulo. Anais do WJogos. Porto Alegre: SBC, 2005. p. 55-65.

PAWASKAR, Chandan. **Continuous collision detection for translating ellipsoid**. [S.l.], 2007. Disponível em: <http://www.gamedev.net/reference/programming/features/ellipsoid-ccd/>. Acesso em: 21 mar. 2010.

SNETHEN, Gary. XenoCollide: complex collision made simple. In: JACOBS, Scott. **Game programming gems 7**. Boston: Charles River Media, 2008. p. 165-178.

SATHE, Raul; SHARLET, Dillon. Fast rigid-body collision detection using farthest feature Maps. In: JACOBS, Scott. **Game programming gems 7**. Boston: Charles River Media, 2008. p. 143-151.

APÊNDICE A – Implementação do algoritmo XenoCollide

O código fonte da implementação do algoritmo XenoCollide é apresentada no Quadro 27. São passados sete parâmetros, sendo os três primeiro a forma, posição e orientação do primeiro corpo rígido. Os próximos três parâmetros contém a mesma informação, mas se referem ao segundo corpo rígido. Por fim, o último parâmetro armazena as informações da colisão, caso ela seja detectada.

```
bool Geometry_FindIntersection( const PolygonalShape& p1, const
Quaternion& q1, const Vector3& t1, const PolygonalShape& p2, const
Quaternion& q2, const Vector3& t2, FindIntersectionResult& result )
{
    static float COLLIDE_EPSILON = 1e-3f;

    // v0 = center of Minkowski sum
    Vector3 v01 = t1;
    Vector3 v02 = t2;
    Vector3 v0 = v02 - v01;

    // Avoid case where centers overlap -- any direction is fine in this
case
    if( v0.IsZero() )
        v0.SetValue( 0.00001f, 0, 0 );

    // v1 = support in direction of origin
    Vector3 n = -v0;
    Vector3 v11 = Geometry_GetSupport( p1, q1, t1, -n );
    Vector3 v12 = Geometry_GetSupport( p2, q2, t2, n );
    Vector3 v1 = v12 - v11;

    if( Vector3::Dot( v1, n ) <= 0 )
    {
        result.normal = n;
        return false;
    }

    // v2 - support perpendicular to v1,v0
    n = Vector3::Cross( v1, v0 );

    if( n.IsZero() )
    {
        result.normal = v1 - v0;
        result.normal.Normalize();
        result.pointA = v11;
        result.pointB = v12;
        return true;
    }

    Vector3 v21 = Geometry_GetSupport( p1, q1, t1, -n );
    Vector3 v22 = Geometry_GetSupport( p2, q2, t2, n );
    Vector3 v2 = v22 - v21;

    if( Vector3::Dot( v2, n ) <= 0 )
    {
        result.normal = n;
```

```
            return false;
        }

        // Determine whether origin is on + or - side of plane (v1,v0,v2)
        n = Vector3::Cross( v1 - v0, v2 - v0 );
        float dist = Vector3::Dot( n, v0 );

        ASSERT( !n.IsZero() );

        // If the origin is on the - side of the plane, reverse the direction
of the plane
        if( dist > 0 )
        {
            std::swap( v1, v2 );
            std::swap( v11, v21 );
            std::swap( v12, v22 );
            n = -n;
        }

        // Phase One: Identify a portal
        for( ;; )
        {
            // Obtain the support point in a direction perpendicular to the
existing plane
            // Note: This point is guaranteed to lie off the plane
            Vector3 v31 = Geometry_GetSupport( p1, q1, t1, -n );
            Vector3 v32 = Geometry_GetSupport( p2, q2, t2, n );
            Vector3 v3 = v32 - v31;

            if( Vector3::Dot( v3, n ) <= 0 )
            {
                result.normal = n;
                return false;
            }

            // If origin is outside (v1,v0,v3), then eliminate v2 and loop
            if( Vector3::Dot( Vector3::Cross( v1, v3 ), v0 ) < 0 )
            {
                v2 = v3;
                v21 = v31;
                v22 = v32;
                n = Vector3::Cross( v1 - v0, v3 - v0 );
                continue;
            }

            // If origin is outside (v3,v0,v2), then eliminate v1 and loop
            if( Vector3::Dot( Vector3::Cross( v3, v2 ), v0 ) < 0 )
            {
                v1 = v3;
                v11 = v31;
                v12 = v32;
                n = Vector3::Cross( v3 - v0, v2 - v0 );
                continue;
            }

            bool hit = false;

            // Phase Two: Refine the portal
            int phase2 = 0;

            // We are now inside of a wedge...
```

```
        for( ;; )
        {
            phase2++;

            // Compute normal of the wedge face
            n = Vector3::Cross( v2 - v1, v3 - v1 );

            // Can this happen???  Can it be handled more cleanly?
            if( n.IsZero() )
            {
                ASSERT( FALSE );
                return true;
            }

            n.Normalize();

            // Compute distance from origin to wedge face
            float d = Vector3::Dot( n, v1 );

            // If the origin is inside the wedge, we have a hit
            if( d >= 0 && !hit )
            {
                result.normal = n;

                // Compute the barycentric coordinates of the origin
                float b0 = Vector3::Dot( Vector3::Cross( v1, v2 ), v3 );
                float b1 = Vector3::Dot( Vector3::Cross( v3, v2 ), v0 );
                float b2 = Vector3::Dot( Vector3::Cross( v0, v1 ), v3 );
                float b3 = Vector3::Dot( Vector3::Cross( v2, v1 ), v0 );

                float sum = b0 + b1 + b2 + b3;

                if( sum <= 0 )
                {
                    b0 = 0;
                    b1 = Vector3::Dot( Vector3::Cross( v2, v3 ), n );
                    b2 = Vector3::Dot( Vector3::Cross( v3, v1 ), n );
                    b3 = Vector3::Dot( Vector3::Cross( v1, v2 ), n );

                    sum = b1 + b2 + b3;
                }

                float inv = 1.0f / sum;

                result.pointA = (b0 * v01 + b1 * v11 + b2 * v21 + b3 *
v31) * inv;
                result.pointB = (b0 * v02 + b1 * v12 + b2 * v22 + b3 *
v32) * inv;

                // HIT!!!
                hit = true;
            }

            // Find the support point in the direction of the wedge face
            Vector3 v41 = Geometry_GetSupport( p1, q1, t1, -n );
            Vector3 v42 = Geometry_GetSupport( p2, q2, t2, n );
            Vector3 v4 = v42 - v41;

            float delta = Vector3::Dot( v4 - v3, n );
            float separation = -Vector3::Dot( v4, n );
```

```
            // If the boundary is thin enough or the origin is outside the
support plane for the newly discovered vertex, then we can terminate
            if( delta <= COLLIDE_EPSILON || separation >= 0 /*|| phase2 >
300*/ )
            {
                result.normal = n;
                return hit;
            }

            // Compute the tetrahedron dividing face (v4,v0,v1)
            float d1 = Vector3::Dot( Vector3::Cross( v4, v1 ), v0 );

            // Compute the tetrahedron dividing face (v4,v0,v2)
            float d2 = Vector3::Dot( Vector3::Cross( v4, v2 ), v0 );

            // Compute the tetrahedron dividing face (v4,v0,v3)
            float d3 = Vector3::Dot( Vector3::Cross( v4, v3 ), v0 );

            if( d1 < 0 )
            {
                if( d2 < 0 )
                {
                    // Inside d1 & inside d2 ==> eliminate v1
                    v1 = v4;
                    v11 = v41;
                    v12 = v42;
                }
                else
                {
                    // Inside d1 & outside d2 ==> eliminate v3
                    v3 = v4;
                    v31 = v41;
                    v32 = v42;
                }
            }
            else
            {
                if( d3 < 0 )
                {
                    // Outside d1 & inside d3 ==> eliminate v2
                    v2 = v4;
                    v21 = v41;
                    v22 = v42;
                }
                else
                {
                    // Outside d1 & outside d3 ==> eliminate v1
                    v1 = v4;
                    v11 = v41;
                    v12 = v42;
                }
            }
        }
    }
}
```

Quadro 27 – Código fonte da função `Geometry_FindIntersection`

APÊNDICE B – Cálculo das propriedades da massa

O código fonte do método `ComputeMassProperties` é disponibilizado no Quadro 28. Este método calcula a massa, centro de massa e momento de inércia de um corpo rígido.

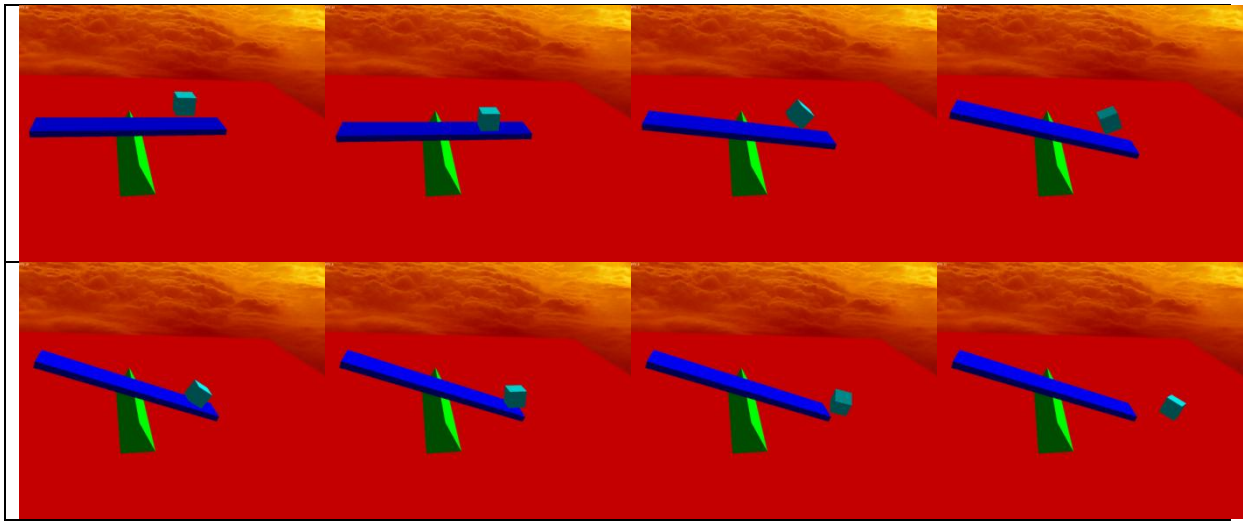
```
void RigidBody::ComputeMassProperties( float density )
{
    Vector3 diag = VEC_ZERO;
    Vector3 offDiag = VEC_ZERO;
    Vector3 weightedCenterOfMass = VEC_ZERO;
    float flMass = 0;

    for( int tri = 0; tri < this->shape.triangleCount; ++tri )
    {
        // Get vertices of triangle i.
        Vector3 v0 = this->shape.vertices[this-
>shape.triangles[tri].idx[0]];
        Vector3 v1 = this->shape.vertices[this-
>shape.triangles[tri].idx[1]];
        Vector3 v2 = this->shape.vertices[this-
>shape.triangles[tri].idx[2]];

        float det =
            v0.x * (v1.y * v2.z - v1.z * v2.y) +
            v0.y * (v1.z * v2.x - v1.x * v2.z) +
            v0.z * (v1.x * v2.y - v1.y * v2.x);

        // Volume
        float tetVolume = det / 6.0f;

        // Mass
        float tetMass = tetVolume * density;
        flMass += tetMass;

        // Center of Mass
        Vector3 tetCenterOfMass = (v0 + v1 + v2) / 4.0f; // Note: includes
origin (0, 0, 0) as fourth vertex
        weightedCenterOfMass += tetMass * tetCenterOfMass;

        // Inertia Tensor
        for( int i = 0; i < 3; ++i )
        {
            int j = (i + 1) % 3;
            int k = (i + 2) % 3;

            diag[i] += det * (v0[i] * v1[i] + v1[i] * v2[i] + v2[i] *
v0[i] + v0[i] * v0[i] + v1[i] * v1[i] + v2[i] * v2[i]) / 60.0f;

            offDiag[i] += det * (
                v0[j] * v1[k] + v1[j] * v2[k] + v2[j] * v0[k] +
                v0[j] * v2[k] + v1[j] * v0[k] + v2[j] * v1[k] +
                2 * v0[j] * v0[k] + 2 * v1[j] * v1[k] + 2 * v2[j] * v2[k])
/ 120.0f;
        }
    }

    Vector3 centerOfMass = weightedCenterOfMass / flMass;
```

```
    diag *= density;
    offDiag *= density;

    Matrix3 intertiaTensor;
    intertiaTensor[0][0] = diag[1] + diag[2];
    intertiaTensor[1][1] = diag[2] + diag[0];
    intertiaTensor[2][2] = diag[0] + diag[1];
    intertiaTensor[1][2] = intertiaTensor[2][1] = -offDiag[0];
    intertiaTensor[0][2] = intertiaTensor[2][0] = -offDiag[1];
    intertiaTensor[0][1] = intertiaTensor[1][0] = -offDiag[2];

    intertiaTensor[0][0] -= flMass * ( centerOfMass.y * centerOfMass.y +
centerOfMass.z * centerOfMass.z);
    intertiaTensor[0][1] -= flMass * (-centerOfMass.x * centerOfMass.y);
    intertiaTensor[0][2] -= flMass * (-centerOfMass.x * centerOfMass.z);
    intertiaTensor[1][1] -= flMass * ( centerOfMass.x * centerOfMass.x +
centerOfMass.z * centerOfMass.z);
    intertiaTensor[1][2] -= flMass * (-centerOfMass.y * centerOfMass.z);
    intertiaTensor[2][2] -= flMass * ( centerOfMass.x * centerOfMass.x +
centerOfMass.y * centerOfMass.y);

    // Symmetry
    intertiaTensor[1][0] = intertiaTensor[0][1];
    intertiaTensor[2][0] = intertiaTensor[0][2];
    intertiaTensor[2][1] = intertiaTensor[1][2];

    this->invMass = 1.0f / flMass;
    this->invInertiaTensor = intertiaTensor.Inverse();
    this->centerOfMass = centerOfMass;
}
```

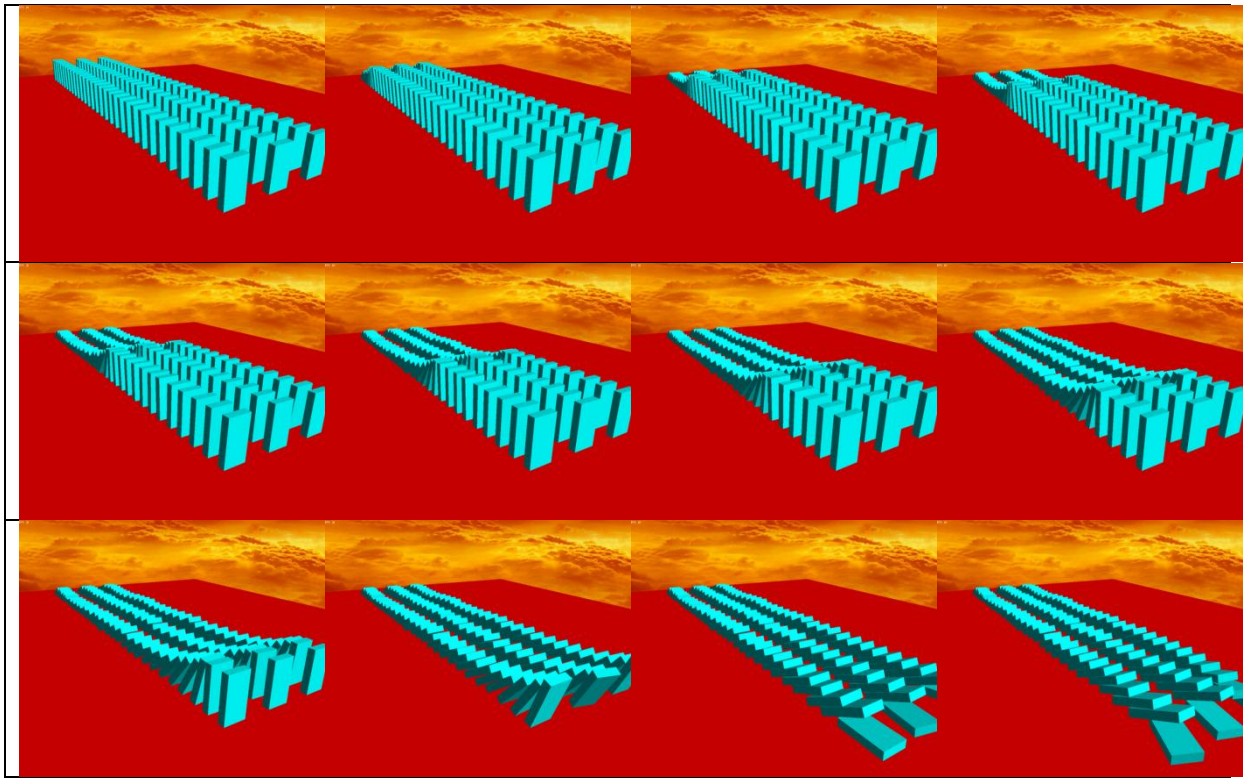
Quadro 28 – Código fonte do método `ComputeMassProperties`

APÊNDICE C – Animações dos cenários 7 e 8

No Quadro 29 e no Quadro 30 é possível ver sequências de imagens da simulação dos cenários 7 e 8, respectivamente.



Quadro 29 – Animações da simulação do cenário 7



Quadro 30 – Animações da simulação do cenário 8