ESQUEMA DE DETECÇÃO E RESPOSTA A COLISÕES PARA ANIMAÇÃO FÍSICA SIMPLIFICADA

Yalmar Temistocles Ponce Atencio

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Claudio Esperança, Ph.D.

Prof. Paulo Roma Cavalcanti, D.Sc.

Prof. Ricardo Cordeiro de Farias, Ph.D.

Prof. João Luiz Dihl Comba, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
AGOSTO DE 2005

PONCE ATENCIO, YALMAR TEMISTOCLES

Esquema de detecção e resposta a colisões para animação física simplificada [Rio de Janeiro] 2005

XI, 59 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2005)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1. Animação Física. 2. Detecção de colisão.

3. Resposta a colisão. 4. Corpos rígidos e deformáveis.

I. COPPE/UFRJ II. Título (série)

Aos meus pais, Mariano e Vicentina.
Aos meus irmãos.

Agradecimentos

Aos meus pais, Mariano e Vicentina, por tudo o esforço que tiveram de fazer para que eu conseguisse chegar até aqui, sem contar com as saudades que eles devem ter passado no tempo que estou nesta cidade.

Aos meus irmãos, Sofia, Elsa, Josefa, Fabio, Francisco, Nestor e Nora por todo o apoio e ajuda nos momentos difíceis.

Ao meu extraordinário orientador Claudio Esperança, por seus importantes conhecimentos e idéias transmitidas nas matérias e durante o desemvolvimento da tese, a paciência que teve comigo na hora de escrever e, sobretudo, por seu exemplo.

Aos professores, Paulo e Antonio , pelos coselhos sempre certeiros e, sobretudo, pelas caronas.

Ao professor João Comba, por aceitar o convite para participar da avaliação deste trabalho.

Não podiam faltar os amigos, Guina, Mariela, Margareth, Marisa, Kelly, Felipe, Alvaro, Erick, Karl, Saulo, Ricardo, Vitor e amigos das turmas 2004 e 2005, pela troca de conhecimento e por termos assistido nos diversos eventos (maua, candongueiros, democráticos, etc.).

A todos os colegas de estudos, funcionários e professores. De forma especial, às funcionárias Mercedes, Claudia Prata e Solange.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc)

ESQUEMA DE DETECÇÃO E RESPOSTA A COLISÕES PARA ANIMAÇÃO FÍSICA SIMPLIFICADA

Yalmar Temistocles Ponce Atencio

Agosto/2005

Orientador: Claudio Esperança

Programa: Engenharia de Sistemas e Computação

Neste trabalho descrevemos um sistema para animação física de objetos rígidos e deformáveis. Estes são representados como grupos de partículas ligadas por restrições lineares. Para simular o movimento dos objetos é usado o esquema de integração de Verlet. Diferentemente das abordagens tradicionais, a simulação física é obtida sem que se compute explicitamente matrizes de orientação, torques ou tensores de inércia. A principal contribuição deste trabalho está ligado à forma como as colisões são tratadas pelo sistema, empregando diferentes abordagens para objetos deformáveis e rígidos. Em particular, mostramos um esquema de detecção de colisão usando a metodologia de Gilbert-Johnson-Keerthi [18] e hierarquias de esferas limitantes, os quais são combinados com as técnicas descritas por Jakobsen.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A COLLISION DETECTION AND RESPONSE SCHEME FOR SIMPLIFIED PHYSICALLY BASED ANIMATION

Yalmar Temistocles Ponce Atencio

August/2005

Advisor: Claudio Esperança

Department: Computing and Systems Engineering

In this work we describe a system for physical animation of rigid and deformable objects. These are represented as groups of particles linked by linear constraints, while a Verlet integrator is used for motion computation. Unlike traditional approaches, we accomplish physical simulation without explicitly computing orientation matrices, torques or inertia tensors. The main contribution of our work is related to the way collisions are handled by the system, which employs different approaches for deformable and rigid bodies. In particular, we show how collision detection using the GJK algorithm [18] and bounding sphere hierarchies can be combined with the projection based collision response technique described by Jakobsen [24].

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Modelagem e animação baseada em física vêm sendo pesquisadas há mais de uma década, encontrando aplicação em entretenimento, simulação, desenho assistido por computador e várias outras áreas.

Uma animação realista requer que os objetos em movimento obedeçam a leis físicas. Para tanto, vários aspectos precisam ser considerados. Por exemplo, no mundo real, dois objetos não podem ocupar o mesmo lugar no espaço ao mesmo tempo. Isto significa que objetos podem empurrar outros objetos dependendo de suas massas e velocidades; objetos podem ser empilhados uns sobre os outros; objetos não podem atravessar o chão, e assim por diante. Se queremos simular um ambiente interativo respeitando a impenetrabilidade dos objetos, é preciso incorporar um mecanismo que permita tratar estas situações. Uma tarefa importante para este mecanismo é detectar configurações onde haja interpenetração entre objetos, as quais são chamadas de "colisões". Porém, sendo o processo de detecção de colisões geralmente muito custoso computacionalmente, a simulação de ambientes interativos dificilmente pode ser alcançada em tempo real.

O tratamento de colisões pode ser dividido em duas fases. Na *detecção* de colisões, objetos que se interpenetram ou que estão em vias de o fazer são identificados, enquanto que a *resposta* a colisões envolve a modificação dos diversos parâmetros físicos dos objetos envolvidos – tipicamente posição, orientação e velocidade – de tal forma que uma configuração fisicamente plausível seja obtida.

Em geral, animação baseada em física demanda uma quantidade considerável de recursos computacionais. Isto se deve ao fato de que, a cada instante de tempo da simulação, diversas características físicas têm que ser computadas, tais como velocidades, forças, torques, momentos e outros.

Diversas abordagens [4, 21, 31, 20] para diversos problemas associados à animação

física podem ser encontradas na literatura. A maior parte dessas técnicas se baseia numa abordagem tradicional e requerem, em geral, que se use métodos precisos para computar as diversas equações que regem as leis físicas, particularmente as da dinâmica. Não obstante, em alguns campos de aplicação tais como jogos interativos, por exemplo, a precisão pode ser sacrificada em prol de uma maior velocidade de simulação.

Em 1999-2001, Thomas Jakobsen [24] propôs um esquema simplificado de simulação baseada em física, conseguindo simular ambientes com objetos deformáveis e rígidos. Este esquema é baseado principalmente na implementação de restrições lineares e a combinação de diversas técnicas que se complementam:

- Objetos (rígidos e deformáveis) são representados por sistemas de partículas com restrições lineares.
- A dinâmica das partículas é simulada usando o integrador de Verlet [41].
- É usada uma raiz quadrada aproximada em vez de uma exata para os cálculos de distância.
- As restrições lineares são mantidas constantes usando um processo de relaxamento.
- Um esquema simples de resposta a colisões que consiste em projetar vértices que penetram uma determinada superfície para fora desta.

Este esquema de simulação é recomendado para jogos já que, embora não seja fisicamente correto num sentido estrito, resulta em animações bastante realistas. No artigo de Jakobsen [24], entretanto, vários detalhes importantes foram suprimidos. Em particular, a ligação entre o mecanismo de detecção de colisão e os algoritmos de resposta a colisões é descrita apenas superficialmente e nenhuma estratégia é sugerida para tratar objetos que requeiram um número maior de restrições lineares. Isto é relevante, já que o emprego de muitas restrições lineares pode comprometer o desempenho do sistema.

Neste trabalho abordamos estes assuntos descrevendo o protótipo de um sistema baseado na abordagem de Jakobsen e no qual várias tentativas de solução são apresentadas. Mais especificamente, apresentamos:

- Um esquema para separar objetos em colisão. Isto é feito deslocando os vértices envolvidos na colisão na direção de um ponto que nós chamamos de *ponto de projeção*, o qual é calculado como um subproduto do mecanismo de detecção de colisão.

- Um esquema de relaxamento ordenado de restrições lineares, o qual permite que o laço de relaxamento possa convergir mais rapidamente.

- Uma metodologia para tratar objetos deformáveis baseados em restrições lineares, de forma que eles possam interagir com objetos rígidos.

O mecanismo de detecção de colisão proposto lida com objetos deformáveis e objetos rígidos. Para tanto, é dividido em duas partes. A primeira consiste de uma variante do método original de Gilbert-Johnson-Kerthi [18], o qual foi estendido posteriormente por Van den Bergen [38]. Este é usado para tratar colisão entre objetos rígidos. A segunda parte baseia-se no uso de *Sphere-Trees* [1, 25] para tratar objetos deformáveis.

O restante deste trabalho é dividido nos seguintes capítulos: O Capítulo 2 faz um apanhado geral das diversas técnicas associadas à animação baseada em física. O Capítulo 3 aborda as principais metodologias para detecção e resposta a colisões. No Capítulo 4, o sistema construído é descrito em pormenores. O Capítulo 5 apresenta alguns resultados obtidos com o uso do sistema. Finalmente, o Capítulo 6 apresenta alguns comentários finais e sugestões para trabalhos futuros.

Capítulo 2

Animação baseada em física

Animação baseada em física tem sido intensamente pesquisada nos últimos anos. Diversas abordagens têm sido propostas [3, 4, 31, 21, 38] na literatura, e um esforço substancial tem sido dispendido na construção de algoritmos eficientes e confiáveis.

A animação em geral requer que o parâmetro "tempo" seja acrescentado às representações dos objetos, permitindo assim descrever suas posições, orientações e formas como funções do tempo.

Tipicamente a animação de um objeto é realizada de duas formas:

- Alterações em sua localização: compreendem essencialmente translações e rotações, isto é, tratam o objeto como um corpo rígido. A localização de um corpo rígido no tempo t é representada por $x(t)$ e $R(t)$, respectivamente, a posição e a orientação do objeto no sistema de coordenadas do mundo (assume-se que o objeto é modelado em um sistema próprio de coordenadas).

- Alterações em sua forma: usadas na animação de corpos deformáveis, usualmente objetos representados por malhas poligonais, onde as posições dos vértices dos polígonos são dependentes do tempo. Assim, as deformações podem ser usadas para simular, por exemplo, fluidos, tecidos, ou pele.

Mais especificamente, entende-se por animação física um processo pelo qual representações de objetos "reais" são postos em movimento através da simulação de leis físicas. Um exemplo muito comum consiste em animar objetos geométricos rígidos aos quais se conferem propriedades tais como massa e densidade, e sobre os quais se aplicam forças tais como gravidade e atrito. Através da simulação são computadas as velocidades e posições dos objetos, levando em conta as interações entre eles e as leis físicas que os governam. Na animação física, para representar adequadamente os objetos é necessário

que se considerem várias características adicionais, além da posição e a orientação. Tais características são estudadas na dinâmica dos corpos.

A dinâmica é a parte da física que estuda o movimento dos corpos. Tem como pilares centrais as três leis de Newton (leis do movimento):

Primeira lei de Newton. Na ausência de forças externas, um objeto em repouso permanece em repouso, e se o objeto está em movimento ele permanece em movimento com velocidade constante. Isto é, só forças externas podem mudar o movimento de um objeto.

Segunda lei de Newton. Para um corpo de massa constante m experimentando uma força F, o movimento do corpo sobre o tempo é dado por $F = ma = m\dot{v} = m\ddot{x}$.

Terceira lei de Newton. Se aplicada uma força externa sobre um objeto, há uma força de igual magnitude mas em direção oposta exercida sobre o causador da força, isto é, a toda ação corresponde uma reação.

Assim, para simular o movimento é necessário que as variáveis de estado dos objetos estejam relacionadas. Por exemplo, a posição de um objeto $x(t)$ está relacionada com a velocidade pela equação $v(t) = \dot{x}(t)$, e com a aceleração pela equação $a(t) = \ddot{x}(t)$. Podemos observar que a velocidade $v(t)$ e a aceleração $a(t)$ são dadas por equações diferenciais. Na prática, a simulação do movimento dos objetos requer o uso de métodos numéricos que resolvam estas equações diferenciais. Na seção 2.4 são descritos dois métodos comumente usados em animação física.

2.1 Representação de corpos rígidos

2.1.1 Sólidos simples

Em animação física, convencionalmente são usadas funções paramétricas para representar sólidos simples, tais como cubos, cilindros, cones, esferas, toros, etc. Isto é feito freqüentemente em ambientes interativos complexos, como jogos, onde é preciso animar milhares de objetos simples. As representações paramétricas são vantajosas, pois permitem que propriedades físicas dos objetos sejam computadas facilmente, por exemplo, o centro de massa e o momento de inércia. Além disso, no processo de detecção de colisões, uma colisão entre objetos representados por funções paramétricas pode ser detectada rapidamente. De forma similar, no processo de resposta a colisões, o cálculo do impulso se torna mais simples.

2.1.2 Sistemas de partículas

Um sistema de partículas é um conjunto de partículas, onde cada partícula tem uma posição e velocidade própria. Freqüentemente, o estado de uma partícula no instante t é descrito por sua posição $x(t)$ e sua velocidade $v(t)$, e é representado por um vetor $X(t)$ da forma

$$X(t) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}.$$

Entretanto, o estado de um sistema de partículas pode ser representado por um vetor de componentes $x_i(t)$ e $v_i(t)$, assim, estendendo-se o vetor $X(t)$, o estado de um sistema de partículas pode ser representado por

$$X(t) = \begin{pmatrix} x_1(t) \\ v_1(t) \\ \vdots \\ x_n(t) \\ v_n(t) \end{pmatrix},$$

onde $x_i(t)$ e $v_i(t)$ são a posição e a velocidade da i-ésima partícula [5].

2.2 Dinâmica das partículas

Para simular o movimento de uma partícula, é preciso conhecer a força que age sobre ela no instante t. Seja $F(t)$ a força resultante que age na partícula no tempo t. A função $F(t)$ é a soma de todas as forças que agem na partícula: gravidade, vento, forças de mola, etc. Se a partícula tem massa m, então a variação de $X(t)$ ao longo do tempo é dada por

$$\frac{d}{dt}X(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \frac{F(t)}{m} \end{pmatrix}. \tag{2.1}$$

Dado um valor de $X(t)$, a equação 2.1 descreve como $X(t)$ é instantaneamente alterado no tempo t.

2.3 Dinâmica de corpos rígidos

Corpos rígidos são representados freqüentemente por sistemas de partículas. Porém, corpos rígidos diferem de partículas visto que ocupam um lugar no espaço e podem sofrer rotações, por isso, a dinâmica de corpos rígidos introduz novas considerações.

Para simular o movimento de um corpo rígido é necessário uma equação similar à usada para simular o movimento de uma partícula (eq. 2.1), onde é necessário determinar $\frac{d}{dt}X(t)$.

A seguir, definimos um conjunto de conceitos e relações físicas que ajudam a determinar $\frac{d}{dt}X(t)$ para corpos rígidos.

2.3.1 Posição e orientação

A posição de uma partícula no espaço no instante t pode ser representada pelo vetor $x(t)$, o qual descreve o deslocamento em relação à origem. De modo similar, a posição de um corpo no instante t é descrito por um vetor $x(t)$, que descreve o deslocamento do corpo em relação à origem. Adicionalmente, um corpo rígido pode sofrer rotações. Uma rotação pode ser representada por diferentes técnicas tais como matrizes, ângulos de Euler, ou quatérnios. Se usarmos matrizes, então a rotação de um corpo rígido pode ser representada diretamente por uma matriz $R(t)$. Então, $x(t)$ e $R(t)$ são as variáveis que descrevem o estado do corpo rígido.

Devido ao fato de que um corpo pode sofrer apenas translações e rotações, definimos a forma de um objeto em termos de um espaço fixo e imutável chamado *espaço do corpo*. Dada uma descrição geométrica do corpo nesse espaço, usamos $x(t)$ e $R(t)$ para transformar o espaço do corpo no espaço do mundo (ver a figura 2.1). Para simplificar algumas equações, é necessário que o centro de massa do corpo coincida com O', a origem do espaço do corpo.

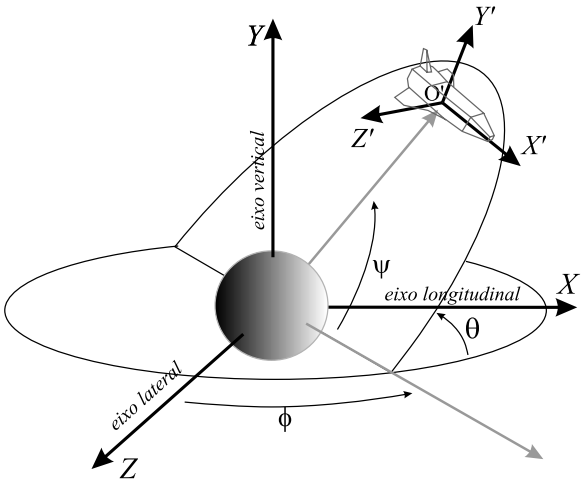

Figura 2.1: Relação do sistema de coordenadas do mundo com o sistema de coordenadas do corpo.

Já que $R(t)$ especifica a rotação do corpo em torno ao centro de massa, então um vetor r fixo no espaço do corpo será rotacionado transformando-se no vetor $R(t) \cdot r$ no espaço do mundo no tempo t. De modo similar, se p_0 é um ponto arbitrário do corpo rígido no espaço do corpo, então a posição de $p(t)$ no espaço do mundo é obtida aplicando a rotação

seguida da translação, ou seja:

$$p(t) = R(t)p_0 + x(t). \tag{2.2}$$

Já que o centro de massa do corpo deve estar localizado na origem do sistema de coordenadas do corpo, a posição do centro de massa no espaço do mundo é dada diretamente por $x(t)$. O que significa que $x(t)$ é a localização do centro de massa no espaço do mundo no tempo t.

2.3.2 Velocidade linear

A velocidade linear descreve a mudança da posição do centro de massa do corpo no tempo. Assim, define-se $x(t)$ como a *posição* do corpo no tempo t. Então, a equação $\dot{x}(t)$ é a velocidade do centro de massa no espaço do mundo e é definida como:

$$v(t) = \dot{x}(t) = \frac{d}{dt}x(t). \tag{2.3}$$

Imaginando que a orientação do corpo é fixa, então o corpo experimentará apenas uma translação pura. A quantidade vetorial $v(t)$ expressa a velocidade desta translação (ver a figura 2.2).

2.3.3 Velocidade angular

Além da translação, um corpo rígido também pode girar ao redor de um eixo fixo que passa por sua origem. Imaginemos que a posição do centro de massa está fixa no espaço do mundo. Então, qualquer movimento dos pontos do corpo só poderá ocorrer mediante uma rotação em torno de algum eixo que passe pelo centro de massa. Comumente, esta rotação é denotada pelo vetor $w(t)$. A direção de $w(t)$ coincide com a direção do eixo de rotação do corpo (ver a figura 2.2).

A magnitude de $w(t)$, denotada por $|w(t)|$, nos diz quão rápida é a rotação. Então, a quantidade $w(t)$ é chamada *velocidade angular*. A velocidade linear relaciona-se com a posição do corpo através da equação 2.3. Analogamente, a rotação $R(t)$ e a velocidade angular $w(t)$ estão relacionadas. Porém, $\dot{R}(t)$ não é $w(t)$, já que $R(t)$ é uma matriz e $w(t)$ é um vetor. Por outro lado, as colunas de $R(t)$ representam os eixos x, y, e z no tempo t; isto significa que as colunas de $\dot{R}(t)$ descrevem a velocidade com que os eixos x, y, e z são transformados. Então pode-se escrever [43]

$$\dot{R}(t) = \left(w(t) \times \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} w(t) \times \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} w(t) \times \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right). \tag{2.4}$$

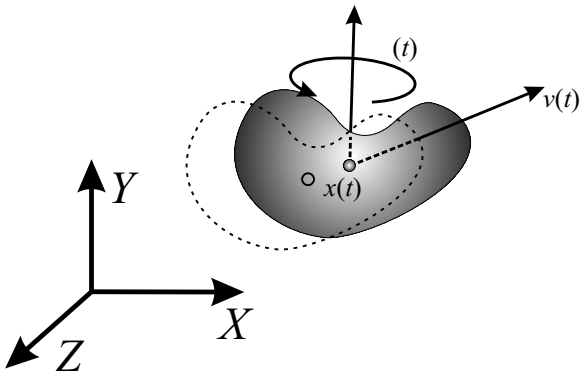

Figura 2.2: velocidade linear $v(t)$ e velocidade angular $\omega(t)$ de um corpo rígido.

Esta expressão é bastante complicada, mas pode ser reescrita de uma maneira mais compreensível. Existe uma característica dos vetores no espaço tridimensional que permite obter um resultado interessante. Dado um vetor v, se define v^* como a matriz anti-simétrica de v dada por

$$\begin{pmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{pmatrix}, \tag{2.5}$$

o que permite definir $u^* \cdot v = u \times v$, e esta igualdade pode ser usada para reescrever a equação (2.4) como

$$\dot{R}(t) = \left(w(t)^* \cdot \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} w(t)^* \cdot \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} w(t)^* \cdot \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right). \tag{2.6}$$

Segundo as regras básicas de multiplicação, podemos fatorar na seguinte expressão

$$\dot{R}(t) = w(t)^* \cdot \left(\begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right), \tag{2.7}$$

que é nada mais do que uma multiplicação de matrizes. Finalmente $\dot{R}(t)$ é expresso como

$$\dot{R}(t) = w(t)^* \cdot R(t). \tag{2.8}$$

Isto, nos dá uma relação entre $\dot{R}(t)$ e $w(t)$. Analogamente, podemos relacionar um vetor $r(t)$ com a velocidade angular $w(t)$ através da equação $\dot{r}(t) = w(t) \times r(t)$.

2.3.4 Massa de um corpo

Para trabalhar com animação baseada em física, é preciso efetuar alguns cálculos de integração sobre o volume dos corpos. Para tornar estes cálculos mais simples, um corpo

freqüentemente é representado por um sistema de partículas. As partículas são indexadas de 1 até N, onde a i-ésima partícula tem massa m_i, e tem a posição constante r_{0i} no espaço do corpo. A localização da i-ésima partícula no espaço do mundo no tempo t é denotada por $r_i(t)$, e está definida como

$$r_i(t) = R(t)r_{0i} + x(t). \tag{2.9}$$

A massa total do corpo, a qual é denotada por M, é a soma das massas das partículas

$$M = \sum_{N}^{i=1} m_i. \tag{2.10}$$

2.3.5 Centro de massa

O centro de massa de um corpo é definido por

$$O_c = \sum_{N}^{i=1} \frac{m_i r_i(t)}{M} = x(t), \tag{2.11}$$

onde M é a massa do corpo. Anteriormente foi definido que $x(t)$ é a localização do centro de massa no espaço do mundo no tempo t. Isto é verdadeiro, já que a i-ésima partícula tem posição $r_i(t) = R(t)r_{0i} + x(t)$ no espaço do mundo no tempo t, então o centro de massa no tempo t é

$$\begin{aligned} \sum_{i=1}^{N} \frac{m_i r_i(t)}{M} &= \sum_{i=1}^{N} \frac{m_i(R(t)r_{0i} + x(t))}{M} \\ &= \frac{\sum_{i=1}^{N} m_i R(t) r_{0i} + \sum_{i=1}^{N} x(t)}{M} \\ &= \frac{x(t)\sum_{i=1}^{N} m_i}{M} \\ &= x(t). \end{aligned} \tag{2.12}$$

2.3.6 Força e torque

Ao representar um corpo rígido como um sistema de partículas, é conveniente imaginar que todas as forças que agem sobre o corpo são aplicadas a cada partícula individualmente.

Seja $F_i(t)$ o somatório das forças externas que agem na i-ésima partícula no tempo t. Então, o *torque* externo $\tau_i(t)$ que age na i-ésima partícula é definido como

$$\tau_i(t) = (r_i(t) - x(t)) \times F_i(t). \tag{2.13}$$

Torques diferem de forças, visto que o torque na partícula depende da localização $r_i(t)$ da partícula, relativa ao centro de massa $x(t)$. De maneira intuitiva, podemos pensar em $\tau_i(t)$ como o eixo de rotação do corpo por influência da força $F_i(t)$, se o centro de massa estivesse firmemente localizado (ver a figura 2.3).

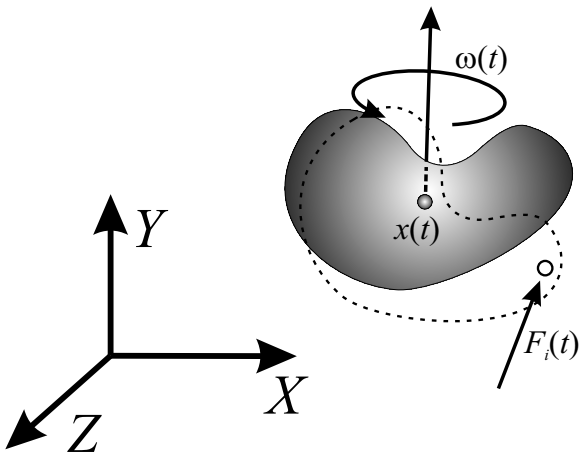

Figura 2.3: O torque $\tau(t)$ devido à força $F_i(t)$ que age na partícula $r_i(t)$ do corpo rígido.

A força externa resultante $F(t)$ que age no corpo é o somatório de $F_i(t)$,

$$F(t) = \sum F_i(t). \tag{2.14}$$

Analogamente, o torque externo resultante é definido por

$$\tau(t) = \sum \tau_i(t) = \sum (r_i(t) - x(t)) \times F_i(t). \tag{2.15}$$

2.3.7 Momento linear

O momento linear ρ de uma partícula de massa m e velocidade v é definido como $\rho = mv$. De maneira análoga, o momento linear $P(t)$ de um corpo rígido é definido como

$$P(t) = \sum m_i \dot{r}_i(t), \tag{2.16}$$

onde a velocidade $\dot{r}_i(t)$ da i-ésima partícula é $\dot{r}_i(t) = v(t) + w(t) \times (r_i(t) - x(t))$. Assim, o momento linear resultante do corpo é definido por

$$\begin{aligned} P(t) &= \sum m_i \dot{r}_i(t) \\ &= \sum (m_i v(t) + m_i w(t) \times (r_i(t) - x(t))) \\ &= \sum m_i v(t) + w(t) \times \sum m_i (r_i(t) - x(t)) \\ &= \sum m_i v(t) = Mv(t). \end{aligned} \tag{2.17}$$

Isto nos dá uma expressão simples para o momento linear resultante de um corpo. Se o corpo é uma única partícula de massa M e velocidade $v(t)$, então obtemos uma relação importante, derivando a equação 2.17 e rearranjando os termos:

$$\dot{v}(t) = \frac{\dot{P}(t)}{M}. \tag{2.18}$$

Finalmente, a força resultante também pode ser obtida derivando-se o momento linear resultante $\dot{P}(t) = F(t)$.

2.3.8 Momento angular

Apesar do significado de momento linear ser bastante intuitivo, o conceito de momento angular para um corpo rígido não é tão simples. Para o momento linear, tem-se que $P(t) = Mv(t)$. Analogamente, o momento angular resultante $L(t)$ de um corpo rígido é dado por $L(t) = I(t)w(t)$. Aqui aparece o novo termo $I(t)$, o qual é chamado *tensor de inércia*, e é uma matriz 3×3. O tensor de inércia $I(t)$ descreve como a massa do corpo está distribuída em relação a seu centro de massa. Note que em ambos os casos (linear e angular), o momento é uma função linear da velocidade – mas no momento angular o fator de escala é uma matriz, e no momento linear é um escalar. Note-se também que $L(t)$ é independente de translações e, de modo similar, $P(t)$ é independente de rotações. A relação entre $L(t)$ e o torque resultante $\tau(t)$ é bastante simples, derivando $\dot{L}(t) = \tau(t)$, o que é análogo à relação entre $P(t)$ e $F(t)$.

2.3.9 Tensor de inércia

O tensor de inércia $I(t)$ é o fator de escala entre o momento angular $L(t)$ e a velocidade angular $w(t)$. Dado um instante t, seja $r_i^{'}(t)$ o deslocamento da i-ésima partícula em relação a $x(t)$, isto é, $r_i^{'}(t) = r_i(t) - x(t)$. O tensor de inércia $I(t)$ é expresso em termos de $r_i^{'}(t)$ como a matriz simétrica

$$I(t) = \sum \begin{pmatrix} m_i(r_{iy}^{'2} + r_{iz}^{'2}) & -m_i r_{ix}^{'} r_{iy}^{'} & -m_i r_{ix}^{'} r_{iz}^{'} \\ -m_i r_{iy}^{'} r_{ix}^{'} & m_i(r_{ix}^{'2} + r_{iz}^{'2}) & -m_i r_{iy}^{'} r_{iz}^{'} \\ -m_i r_{iz}^{'} r_{ix}^{'} & -m_i r_{iz}^{'} r_{iy}^{'} & m_i(r_{iy}^{'2} + r_{iz}^{'2}) \end{pmatrix}. \tag{2.19}$$

À primeira vista, parece necessário computar os somatórios para achar $I(t)$ toda vez que há uma mudança na orientação do corpo. Isto pode ser custoso durante a simulação, a menos que os objetos tenham formas simples (por exemplo, esferas, cubos, etc). Entretanto, é possível computar o tensor de inércia a um baixo custo pré-computando estes somatórios em coordenadas do espaço do corpo. Usando o fato de que $r_i^{'T} r_i^{'} = r_{ix}^{'2} + r_{iy}^{'2} + r_{iz}^{'2}$,

podemos re-escrever $I(t)$ como a diferença

$$I(t) = \sum m_i r_i^{'T} r_i^{'} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} m_i r_{ix}^{'2} & m_i r_{ix}^{'} r_{iy}^{'} & m_i r_{ix}^{'} r_{iz}^{'} \\ m_i r_{iy}^{'} r_{ix}^{'} & m_i r_{iy}^{'2} & m_i r_{ix}^{'} r_{iz}^{'} \\ m_i r_{iz}^{'} r_{ix}^{'} & m_i r_{iz}^{'} r_{iy}^{'} & m_i r_{iz}^{'2} \end{pmatrix}. \tag{2.20}$$

Se denotarmos a matriz identidade 3×3 por $\mathbf{1}$, $I(t)$ pode ser expresso como

$$I(t) = \sum m_i (r_i^{'T} r_i^{'}) \mathbf{1} - r_i^{'} r_i^{'T}, \tag{2.21}$$

como $r_i(t) = R(t) r_{0i} + x(t)$ onde r_{0i} é uma constante, temos $r_i^{'} = R(t) r_{0i}$. Visto que $R(t) R(t)^T = \mathbf{1}$, então podemos re-escrever $I(t)$ como

$$\begin{aligned} I(t) &= \sum m_i (r_i^{'T} r_i^{'}) \mathbf{1} - r_i^{'} r_i^{'T} \\ &= \sum m_i ((R(t) r_{0i})^T (R(t) r_{0i}) \mathbf{1} - (R(t) r_{0i})(R(t) r_{0i})^T) \\ &= \sum m_i ((r_{0i}^T r_{0i}) \mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T) \\ &= \sum m_i (R(t)(r_{0i}^T r_{0i}) R(t)^T \mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T) \\ &= R(t) (\sum m_i ((r_{0i}^T r_{0i}) \mathbf{1} - r_{0i} r_{0i}^T) R(t)^T. \end{aligned} \tag{2.22}$$

Já que $r_{0i} r_{0i}^T$ é um escalar, e usando $I_b = \sum m_i ((r_{0i}^T r_{0i}) \mathbf{1} - r_{0i} r_{0i}^T)$, o tensor de inércia é expresso como

$$I(t) = R(t) I_b R(t)^T, \tag{2.23}$$

onde podemos observar que I_b é especificado no espaço do corpo.

2.3.10 Equações do movimento de corpos rígidos

No começo desta seção foram apresentadas as equações do movimento para uma partícula simples, onde seu estado é representado por um único vetor $X(t)$ que reúne as componentes $x(t)$ e $v(t)$. De modo similar, um corpo rígido pode ser representado por um vetor $X(t)$ expresso por

$$X(t) = \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix}, \tag{2.24}$$

e as demais características são obtidas derivando-se $X(t)$

$$\frac{d}{dt} X(t) = \frac{d}{dt} \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ w(t) * R(t) \\ F(t) \\ \tau(t) \end{pmatrix}. \tag{2.25}$$

Esta formulação é a tradicionalmente usada na maioria das abordagens para animação baseada em física.

2.4 Métodos de integração

Para simular o movimento dos corpos é necessário que se use métodos numéricos para resolver equações diferenciais. Cada método tem características próprias, variando em precisão e estabilidade.

O propósito dos métodos de integração é obter valores instantâneos de determinados parâmetros de cada objeto a partir de suas derivadas. Assim, por exemplo, é possível obter a posição de um objeto integrando sua velocidade durante um intervalo de tempo.

2.4.1 Métodos baseados em expansões de Taylor

Uma ferramenta bastante empregada por métodos de integração é aproximar funções através de expansões de Taylor:

Teorema de Taylor: Dada a função $x(t)$, assume-se que tem n-ésima derivada contínua em $[t_0, t_1]$ e é derivável no intervalo aberto (t_0, t_1). Então, o polinômio de Taylor $P_n(t_1, t_0)$ da função $x(t)$ é definido por

$$\begin{aligned} P_n(t_1,t_0) &= \frac{f^{(n)}(t_0)}{n!}(t_1-t_0)^n + \frac{f^{(n-1)}(t_0)}{(n-1)!}(t_1-t_0)^{n-1} + \cdots + f'(t_0)(t_1-t_0) + f(t_0) \\ &= \sum_{k=0}^{n} \frac{f^{(k)}(t_0)}{k!}(t_1-t_0)^k. \end{aligned} \tag{2.26}$$

A função $x(t_1)$ é definida pela equação $x(t_1) = P_n(t_1, t_0) + R_n(t_1, t_0)$, onde o termo $R_n(t_1, t_0)$, chamado *resto* e, é dado em duas versões:

- forma de Lagrange

$$R_n(t_1,t_0) = \frac{f^{(n+1)}(\bar{t})}{(n+1)!}(t_1-t_0)^{n+1}, \quad \bar{t} \in (t_0,t_1). \tag{2.27}$$

- forma de Cauchy

$$R_n(t_1,t_0) = \frac{f^{(n+1)}(\bar{t})}{n!}(t_1-\bar{t})^n(t_1-t_0), \quad \bar{t} \in (t_0,t_1). \tag{2.28}$$

Note-se que o problema em questão consiste em aproximar uma solução para a equação diferencial

$$\dot{x}(t) \approx f(t, x(t)), \qquad \text{onde} \qquad x(t_0) = x_0. \tag{2.29}$$

Um método clássico, bastante usado devido a sua simplicidade é o método de Euler. Porém, sua maior desvantagem são os erros de precisão, já que o método é pouco estável para muitos exemplos práticos.

O método de Euler usa o Teorema de Taylor, com $n = 2$ no intervalo $[t_i, t_{i+1}]$. Por conseguinte, a expansão de Taylor para o método de Euler é dada por:

$$x(t_{i+1}) = x(t_i) + \dot{x}(t_i)h + \ddot{x}(\bar{t})\frac{h^2}{2}, h = t_{i+1} - t_i \geq 0, \bar{t} \in [t_i, t_{i+1}]. \tag{2.30}$$

Pode-se notar que $x(t)$ é uma solução para a equação 2.29, e podemos trocar $x_i = x(t_i), \forall i$.

Descartando o termo de segunda ordem, e trocando o termo y_i por x_i, o método de integração de Euler é dado por

$$y_{i+1} = y_i + hf(t_i, y_i), \qquad i \geq 0. \tag{2.31}$$

Outras variantes baseadas em expansões de Taylor de maior ordem são conhecidas como métodos de Runge-Kutta. Estes permitem maior precisão do que o método de Euler, ainda usando uma função $f(t, x(t))$. Entretanto, precisam da extensão do Teorema de Taylor para funções de duas variáveis. Adicionalmente, dependendo da precisão podem-se usar variantes baseadas em derivadas de segunda, terceira ou quarta ordem.

Além destes métodos baseados no teorema de Taylor, existem outros métodos conhecidos, por exemplo, o *método previsor-corretor* e os *métodos de extrapolação*. Uma explicação detalhada destes métodos é encontrada no texto de Eberly [15].

2.4.2 Método de integração de Verlet

É um método numérico que tem sua origem na dinâmica molecular. Foi descoberto por Verlet [41]. Este método e suas variações se tornaram populares na indústria de jogos através do trabalho de Jakobsen[24], e também é baseado no teorema de Taylor.

$$x(t_{i+1}) = x(t_i) + \dot{x}(t_i)h + \frac{1}{2}\ddot{x}(t_i)h^2 + \frac{1}{6}x^3(t_i)h^3 + O(h^4) \tag{2.32}$$

$$x(t_{i-1}) = x(t_i) - \dot{x}(t_i)h + \frac{1}{2}\ddot{x}(t_i)h^2 - \frac{1}{6}x^3(t_i)h^3 + O(h^4). \tag{2.33}$$

A primeira expansão projeta o valor da função para um instante de tempo posterior a t_i, enquanto que a segunda o faz para um instante de tempo anterior a t_i. Somando estas equações, e mantendo o termo $x(t_{i+1})$ o resultado é

$$x(t_{i+1}) = 2x(t_i) - x(t_{i-1})h + \ddot{x}(t_i)h^2 + O(h^4). \tag{2.34}$$

Os termos de velocidade e o termo de terceira ordem foram cancelados, e também a aproximação da ordem $O(h^4)$. Assim o esquema de integração é dado por

$$x(t_{i+1}) = 2x(t_i) - x(t_{i-1}) + \frac{h^2}{m}F(t_i, x(t_i), \dot{x}(t_i)), \qquad i \geq 1, \tag{2.35}$$

onde $\ddot{x}(t) = \frac{F(t,x(t),\dot{x}(t)}{m}$ é a aceleração.

Uma das principais vantagens deste método é que ele é reversível no tempo. A reversibilidade mostra que a equação $x(t_{i-1}) = 2x(t_i) - x(t_{i+1}) + h^2\frac{F(t_i,x(t_i),\dot{x}(t_i))}{m}$ pode aproximar uma posição anterior no tempo.

2.5 O método simplificado de Jakobsen

A animação baseada em física deve ser visualmente realista, e diversas abordagens foram apresentadas até agora. Na maioria dos casos, é preciso usar métodos sofisticados e exatos para simular a dinâmica. Porém, em aplicações de jogos interativos, a precisão não é realmente o mais importante, mas sim que o resultado final tenha uma aparência realista e que possa ser executado rapidamente.

Jakobsen [24] apresentou um conjunto de métodos e técnicas que, unidas, conseguem atingir em grande parte estes objetivos. Estes métodos são relativamente simples de implementar (comparados com outros esquemas) e têm um bom desempenho. O algoritmo é iterativo e permite aumentar a precisão do método sacrificando parte de sua rapidez: se uma pequena fonte de imprecisão é aceita, o código pode conseguir uma execução mais rápida. Esta margem de erro ainda pode ser ajustada de forma adaptativa em tempo de execução.

Em geral, o sucesso deste método vem da combinação de varias técnicas que se beneficiam umas das outras, principalmente o uso do integrador Verlet, o tratamento de colisões usando projeção, e a resolução de restrições de distância usando relaxamento. A seguir serão descritas as componentes mais importantes da abordagem proposta por Jakobsen.

2.5.1 Uso do integrador de Verlet

O coração da simulação é um sistema de partículas. Em implementações de sistemas de partículas, cada partícula tem duas componentes principais: sua posição $x(t)$ e sua velocidade $v(t)$. Então, no laço de iterações, a nova posição e a nova velocidade freqüentemente são computadas aplicando-se métodos como Euler, Runge-Kutta, e outras variantes. Porém, nesta metodologia é usada uma representação sem velocidade do integrador de Ver-

let: em vez de se armazenar a posição e a velocidade das partículas, é preciso armazenar a posição corrente $x(t_i)$ e a posição anterior $x(t_{i-1})$ da partícula. Mantendo o *passo de tempo fixo*, a regra de atualização é dada por

$$\begin{aligned} x(t_{i+1}) &= 2x(t_i) - x(t_{i-1}) + \frac{h^2}{m} F(t_i, x(t_i), \dot{x}(t_i)) \\ x(t_{i-1}) &= x(t_i). \end{aligned}$$

Como visto na seção 2.4.2, a integração Verlet é bastante estável, já que a velocidade é dada implicitamente. Em conseqüência, velocidade e posição dificilmente saem de sincronismo. Isto se deve ao fato que $2x(t_i) - x(t_{i-1}) = x(t_i) + (x(t_i) - x(t_{i-1}))$, sendo que $x(t_i) - x(t_{i-1})$ é uma aproximação da velocidade corrente. Esta aproximação nem sempre é exata, já que energia pode ser liberada pelo sistema (ou seja, dissipada).

2.5.2 Tratamento de colisões e contatos por projeções

Um esquema bastante usado é *baseado em penalidades*, que trata contatos inserindo molas nos pontos penetrados. Isto é simples de implementar, mas tem grandes desvantagens. Por exemplo, é difícil escolher constantes de mola adequadas tais que, por um lado, objetos não se penetrem e, por outro lado, o sistema resultante não fique instável. A abordagem de Jakobsen emprega outra estratégia: pontos penetrados são simplesmente projetados para fora do obstáculo. Falando de forma genérica, uma projeção consiste em mover os pontos penetrados uma pequena distância até que fiquem livres do obstáculo. Normalmente, isto significa mover os pontos perpendicularmente na direção da superfície do obstáculo (ver a figura 2.4).

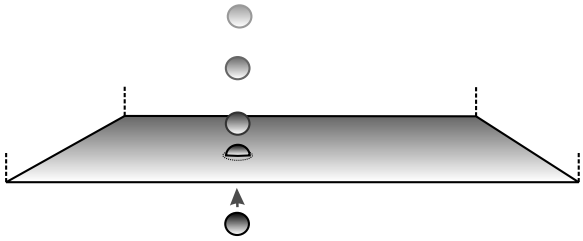

Figura 2.4: Resposta à colisão entre uma partícula e o chão.

2.5.3 Resolvendo restrições concorrentes usando relaxamento

Para entender melhor esta técnica, estudaremos um exemplo: um modelo de um retalho de tecido. Tais objetos podem ser modelados por um sistema simples de molas interconectadas entre partículas. Porém, a simulação destes sistemas freqüentemente requer

a resolução de equações diferenciais, a qual sofre de alguns dos problemas como nos sistemas baseados em penalidades. Porém, é possível conseguir um resultado interessante se a rigidez das molas tendem para o infinito. Neste caso, o sistema repentinamente se resolve de forma estável. Pode-se pensar neste processo como a inserção de uma mola infinitamente dura entre duas partículas, de tal forma que o comprimento de repouso da mola é alcançado instantaneamente. Matematicamente, isto é definido pela equação bilateral $|x_2 - x_1| = d$, chamada de *restrição de distância*, onde d é a distância entre as partículas x_2 e x_1. Na prática, as partículas inicialmente podem estar corretamente posicionadas, mas depois de um passo no tempo, a distância de separação entre elas pode se tornar inválida. Para obter novamente a distância correta, dependendo do caso, pode-se empurrar as partículas para longe uma da outra ou puxá-las mais para perto uma da outra, dependendo do sinal de $|x_2 - x_1| - d$. A figura 2.5 mostra estas configurações.

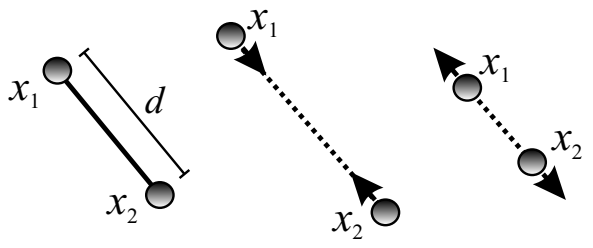

Figura 2.5: O estado estável (esquerda) e possíveis estados instáveis de uma restrição de distância.

Para conseguir um resultado razoável pode-se usar o seguinte algoritmo

1. $\vec{\delta} = x_2 - x_1$;
2. $\epsilon = \frac{|\vec{\delta}| - d}{|\vec{\delta}|}$;
3. $x_1 -= \vec{\delta} \cdot 0.5 \cdot \epsilon$;
4. $x_2 += \vec{\delta} \cdot 0.5 \cdot \epsilon$.

Está técnica pode ser aproveitada para simular corpos deformáveis em diversos tipos de aplicações, e também para simular corpos rígidos.

Como foi descrito, na abordagem tradicional, as equações que governam o movimento de corpos rígidos requerem que se manipulem expressões matemáticas complexas. Em contraste com o conjunto de técnicas apresentadas por Jakobsen, permite simular corpos rígidos de forma simples e, em alguns casos, até com maior rapidez e versatilidade do que a metodologia tradicional.

Capítulo 3

Detecção de colisão

O mecanismo de detecção de colisão é um componente fundamental para um sistema de simulação baseada em física [26, 40, 29, 32, 33]. Este componente permite efetuar consultas rápidas de proximidade geométrica entre objetos durante a simulação. Isso não somente inclui verificações de *como* dois objetos se intersectam, mas também *quando* e *onde* a colisão ocorreu.

Neste capítulo introduzimos conceitos relevantes dentro do contexto deste trabalho e discutimos métodos que geralmente são usados para representar objetos, assim como algoritmos para fazer consultas de proximidade e interseção entre objetos.

3.1 Objetos geométricos

Um objeto geométrico é um conjunto fechado de pontos, limitado e não vazio. É fechado, pois a borda faz parte do objeto e é limitado significa que existe uma esfera de raio finito que limita o objeto. Assim, por exemplo, um plano é fechado mas não limitado.

Tais objetos geométricos podem ser de dois tipos: objetos *convexos* e objetos *côncavos*. Um objeto é *convexo* se contém todos os segmentos de reta que conectam pares de pontos, caso contrário é chamado *côncavo*. Em geral, os objetos convexos permitem executar verificações de interseção mais rapidamente do que os objetos côncavos.

Objetos complexos podem ser compostos de objetos mais simples – tipicamente convexos – chamados de *formas primitivas*, ou *primitivas*. As primitivas mais freqüentemente usadas em modelagem geométrica são esferas, cones, cilindros, caixas [8] e poliedros tais como paralelepípedos, tetraedros, etc.

3.1.1 Objetos poliedrais

Um poliedro é uma forma tridimensional delimitada por um número finito de "faces" poligonais, as quais são partes de planos. As faces são delimitadas por "arestas", as quais são segmentos de reta. As arestas encontram-se em pontos chamados "vértices". Tal representação tem a topologia da esfera o que produz um *grafo planar*. Um grafo é *planar* se este pode ser desenhado sem que suas arestas se intersectem. Para grafos planares o número de diferentes tipos de elementos estão relacionados pela *fórmula de Euler* $v - e + f = 2$, onde v, e, e f denotam, respectivamente, o número de vértices, arestas e faces na fronteira do poliedro.

Uma representação da fronteira de um poliedro é normalmente constituída do conjunto de seus elementos – faces, arestas e vértices – e de uma relação de incidência entre eles.

Uma estrutura de dados adequada para representar a fronteira (superfície) de um poliedro é a estrutura *winged-edge* de Baumgart [6]. Nesta estrutura, a relação de incidência dos elementos é armazenada nos *nós-aresta*, do seguinte modo:

- para cada aresta não orientada do grafo é mantido um nó-aresta;
- um nó-aresta armazena referências para os vértices e faces incidentes e ponteiros para quatro nós-aresta adjacentes;
- para cada vértice incidente são mantidos ponteiros para o sucessor e predecessor da aresta. Estes podem ser usados para percorrer as arestas de uma face incidente no sentido anti-horário. (ver a figura 3.1).

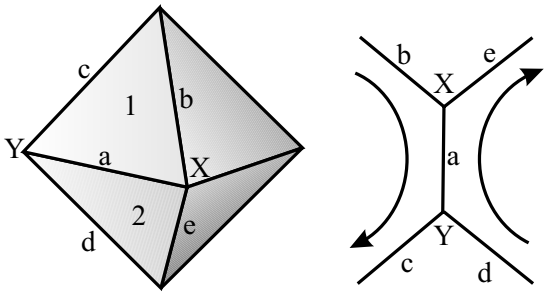

Figura 3.1: A ligação de nós-aresta numa estrutura winged-edge

Nas estruturas winged-edge, cada nó-aresta representa uma aresta não orientada no grafo de adjacência de vértices. Ao percorrer as arestas incidentes num vértice v no sentido anti-horário, é preciso determinar para cada nó-aresta visitado qual de seus dois vértices é v. Isto faz com que o percurso do grafo se torne ineficiente devido a sucessivos

testes. Este problema é resolvido na estrutura conhecida como *halfedge*, na qual cada aresta unidirecional é representada por um par de nós-aresta direcionados [27], isto é, semiarestas (halfedges). Nos grafos de adjacência de vértices, cada halfedge armazena uma referência para o vértice no qual incide, um ponteiro para a halfedge seguinte em relação à face de incidência e um ponteiro para a halfedge de sentido oposto (ver a figura 3.2).

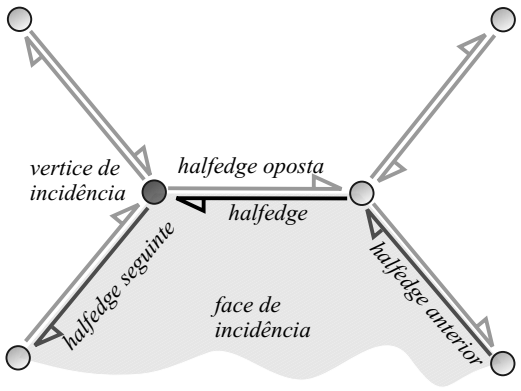

Figura 3.2: A ligação de nós-aresta numa estrutura half-edge

3.1.2 Soma de Minkowski

A *soma de Minkowski* de dois objetos convexos (politopos) A e B é definida por

$$A + B = \{x + y : x \in A, y \in B\}, \tag{3.1}$$

x e y são vetores correspondentes aos pontos de A e B em relação à origem do sistema de coordenadas.

O objeto $A + B$ é o conjunto de pontos obtido por um processo de varredura que translada o centro de massa de B para cada ponto de A, ou seja, faz-se uma cópia do objeto B centrado em cada ponto de A (ver a figura 3.3).

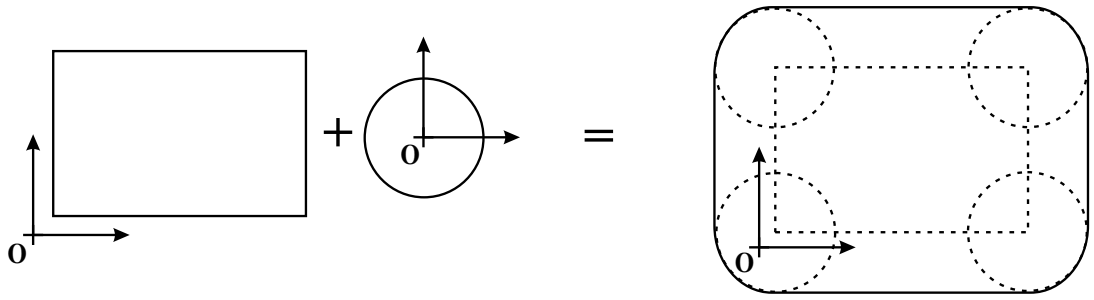

Figura 3.3: A soma de Minkowski de uma caixa e uma esfera.

Uma propriedade muito útil da soma de Minkowski é o fato de que a soma de dois objetos convexos é um objeto convexo. Para obter o politopo da soma de Minkowski [39,

40] pode ser usada uma técnica bastante simples, a qual consiste em computar o fecho convexo do conjunto de todas as combinações $a + b$, onde $a \in A$ e $b \in B$.

3.2 Detecção de colisão em poliedros convexos

Nesta seção apresentaremos alguns algoritmos para executar consultas de proximidade entre pares de poliedros convexos.

Discutiremos com maior ênfase o algoritmo de Gilbert-Johnson-Keerthi (GJK). Trata-se de um método iterativo para computar a distância e a profundidade de penetração entre pares de objetos convexos. Antes de descrevermos o algoritmo GJK, entretanto, vamos apresentar alguns conceitos e algoritmos relacionados.

3.2.1 Consultas de proximidade

Uma consulta de proximidade é uma consulta entre um par de objetos que fornece uma idéia da sua configuração geométrica relativa. Os tipos de consulta para pares de politopos são:

- encontrar um ponto comum;
- encontrar um plano de separação ou um eixo de separação;
- computar a distância e o par de pontos mais próximos;
- computar a profundidade de penetração e o par de pontos que definem essa distância.

Estas consultas de proximidade são tópicos populares em geometria computacional. Para tal propósito existem basicamente dois tipos de algoritmos: algoritmos *single-shot* e algoritmos *iterativos*.

Os primeiros têm interesse predominantemente teórico, já que foram criados na tentativa de estabelecer cotas inferiores e superiores para os diversos tipos de consultas de proximidade. Os segundos, por outro lado, têm maior interesse prático uma vez que podem tomar partido da coerência temporal.

3.2.2 Plano de separação

Um *plano de separação* (PS) é um plano que está localizado entre dois objetos convexos, separando-os. A rigor, um plano de separação não garante a disjunção de dois objetos,

mas garante a disjunção dos seus interiores. Eventualmente, A está situado no lado positivo e B no lado negativo ou vice-versa. Matematicamente, um PS é definido por $H(v, \delta)$, onde v é chamado de *eixo de separação* (ES).

É fácil ver que, se v é um eixo de separação, então existe um escalar δ tal que o plano de separação possa ser definido. Portanto, encontrar um eixo de separação é suficiente para provar que dois objetos não se intersectam.

Uma abordagem original para encontrar um eixo de separação foi apresentado por Chung e Wang [10]. Um eixo de separação para os politopos A e B é definido matematicamente da seguinte maneira: seja $v \neq 0$ um vetor tal que $v \cdot a \geq v \cdot b$ para todo $a \in vert(A)$ e $b \in vert(B)$.

O algoritmo de Chung e Wang é um método iterativo e usa *mapeamento de suporte* dos politopos para computar aproximações do PS. Um mapeamento de suporte descreve completamente a geometria de um objeto convexo e pode ser visto como uma representação implícita do objeto. O mapeamento de suporte de um objeto convexo A é definido como uma função S_A que, para um vetor v, seleciona um ponto de A de acordo com:

$$S_A(v) = \max\{v \cdot (x - x_c) : x \in A\}, \tag{3.2}$$

onde x_c é o centro de massa do objeto A.

O resultado de um mapeamento de suporte é chamado de *ponto de suporte*.

Este algoritmo permite detectar rapidamente colisões entre entre objetos convexos, usando o fato que dois poliedros não colidem se existe um eixo de separação entre eles. Em cada iteração, o algoritmo encontra um eixo candidato, e em tempo constante verifica se este eixo é um eixo de separação. Neste caso, os objetos não colidem, e o eixo é armazenado para ser usado como candidato inicial numa consulta de colisão futura (possivelmente no quadro seguinte). Caso contrário, o algoritmo continua a procura. Se o algoritmo determina que não existe eixo de separação, então reporta que há colisão.

3.2.2.1 O algoritmo de Chung e Wang

Seja A um poliedro, $a = S_A(-v)$ o ponto de suporte de A na direção de $-v$ e $b = S_B(v)$ o ponto de suporte de B na direção de v. Se $v \cdot (a - b) > 0$, então A e B não colidem.

Segundo a definição acima, o algoritmo trabalha da seguinte maneira: sejam A e B dois poliedros, v_0 um vetor unitário inicial escolhido, a_0 um ponto de suporte de A na direção $-v_0$ e b_0 um ponto de suporte de B na direção v_0. Então, para algum i, A e B não

colidem se

$$v_i \cdot (a_i - b_i) \geq 0. \tag{3.3}$$

Em geral, se o teste para o vetor v_i fracassa, os poliedros A e B ainda podem não colidir. Neste caso, escolhemos um novo vetor v_{i+1} a partir de v_i dado pela seguinte equação:

$$v_{i+1} = v_i - 2(r_i \cdot v_i)r_i, i = \{0, 1, ...\}, \tag{3.4}$$

onde $r_i = \frac{w_i}{\|w_i\|}$ e $w_i = a_i - b_i$. Isto corresponde a refletir o vetor v_i em torno do vetor n_i ($n_i = (w_i \times v_i) \times w_i$) que é o vetor normal a w_i no plano que contém v_i (ver a figura 3.4).

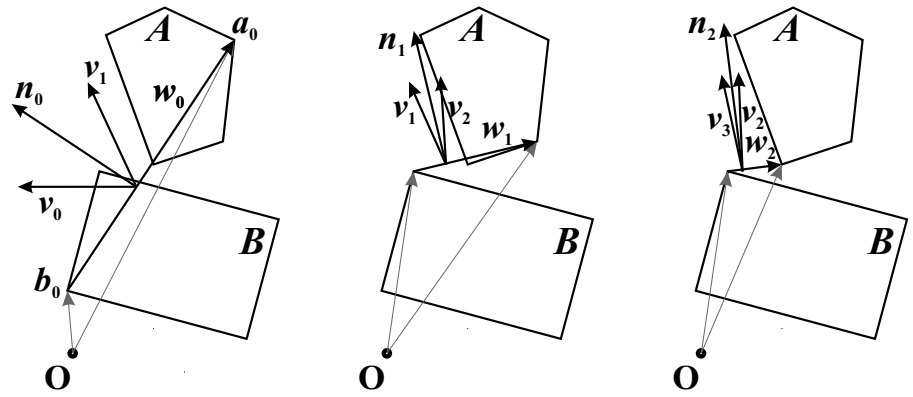

Figura 3.4: Três iterações do algoritmo de Chung e Wang.

Este algoritmo termina se existe um v_i para $i = \{0, 1, ...\}$ que é um eixo de separação (ou seja, $v_i \cdot w_i \geq 0$), ou quando temos evidência de que os interiores dos objetos se intersectam [10].

Porém, o algoritmo pode não convergir rapidamente devido à seguinte característica: examinando várias iterações do algoritmo, se observa que quando v_i está perto de ser um eixo de separação (isto é, $v_i \cdot w_i \approx 0$), então $v_{i+1} \cdot w_{i+1} \approx 0 \approx v_i \cdot w_i$, o que mostra que a convergência é lenta. Este problema ocorre freqüentemente quando os objetos estão quase se tocando. Para minimizar este problema, na prática, é preciso computar um fator de convergência ρ usando a fórmula

$$\rho = \frac{r_i \cdot v_i}{r_{i-1} \cdot v_{i-1}}. \tag{3.5}$$

Note-se que $r_i \cdot v_i = \frac{\|v_{i+1} - v_i\|}{2}$. Para os casos onde os objetos apenas se tocam, freqüentemente $\rho \approx 1$, o que mostra que o algoritmo pode convergir lentamente. Em geral, o número de iterações pode crescer arbitrariamente. Porém, Chung e Wang contornam o problema usando uma propriedade que permite terminar o algoritmo num número finito de iterações (ver pág. 114 em [40]).

3.2.3 Cálculo da distância e profundidade de penetração

Dobkin e Kirkpatrick [13] apresentaram uma das primeiras soluções para o cálculo de distância entre dois politopos. Eles desenvolveram um algoritmo para computar a distância entre dois poliedros convexos num tempo que é linear no número de vértices. Também mostraram que a distância entre um par de politopos com m e n vértices pode ser encontrado em tempo $O(\log m \log n)$ usando uma representação hierárquica [14].

Em contraste, Van den Bergen [40] usa o conceito de *configuração do espaço de obstáculos* (configuration space obstacle - CSO) de dois politopos A e B para computar a distância entre eles.

Configuração do espaço de obstáculos. Para um par de objetos convexos A e B sua CSO é dada por $A - B$, ou seja, a soma de Minkowski de A e $-B$. Este conjunto é particularmente útil em detecção de colisão, visto que A e B se intersectam se e somente se sua CSO contém a origem:

$$A \cap B \neq \emptyset \equiv 0 \in A - B. \tag{3.6}$$

Mais ainda, sua distância é dada por

$$d(A, B) = \min\{\|x\| \colon x \in A - B\}. \tag{3.7}$$

De modo similar, a profundidade de penetração pode ser expressa em termos da CSO como

$$p(A, B) = \min\{\|x\| \colon x \in A - B\}. \tag{3.8}$$

Para um par de objetos que se intersectam, a profundidade de penetração é a distância do ponto na casca de $A - B$ que está mais perto da origem (ver a figura 3.5).

Então, a distância entre A e B é computada pelo ponto na sua CSO que está mais próximo da origem. Se os objetos se intersectam, então a profundidade de penetração é computada como a distância entre esse ponto e a origem.

3.2.4 O algoritmo de Gilbert-Johnson-Keerthi (GJK)

O algoritmo de GJK é um método iterativo para computar a distância entre objetos convexos, mas pode ser usado para resolver vários tipos de consulta de proximidade. Este método foi usado neste trabalho para fazer consultas de proximidade entre objetos convexos e foi escolhido pelas seguintes razões:

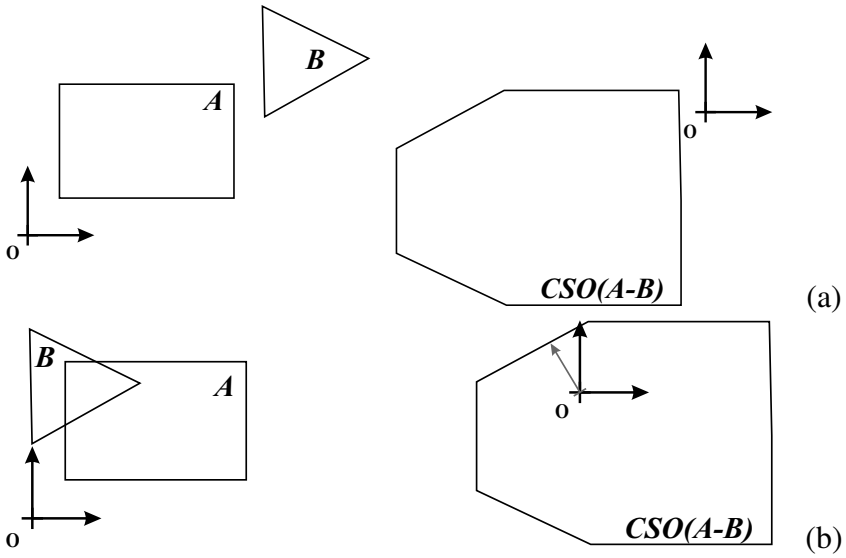

Figura 3.5: Criação do CSO: (a) CSO de A e B quando estes não se intersectam. (b) CSO de A e B quando estes se intersectam

- GJK é extremamente versátil. Pode ser aplicado a objetos convexos em geral, usando a soma de Minkowski de objetos convexos.
- GJK é um dos métodos mais rápidos para executar consultas de proximidade. Versões incrementais do GJK exploram coerência de quadros alcançando um desempenho que é competitivo com outros métodos.
- Apesar de GJK ser complexo, sua implementação é relativamente simples. O algoritmo só precisa levar em conta alguns casos especiais.

Por ser um método iterativo, o algoritmo é bastante suscetível a erros numéricos, que podem causar um comportamento ruim. Por conseguinte, devem-se tomar alguns cuidados na implementação dos detalhes numéricos do algoritmo.

O método usa mapeamento de suporte e a definição do CSO como recursos para executar consultas rápidas entre politopos. Essencialmente, é um método que busca aproximar iterativamente um ponto próximo da origem de $A - B$ (a CSO de A e B), onde A e B são objetos convexos. Denotamos este ponto como $v(A - B)$, onde $v(C) \in C$ e $\|v(C)\| = min\{\|x\|: x \in C\}$. Por conseguinte, a distância entre A e B pode ser expressa como $d(A, B) = \|v(A - B)\|$.

GJK aproxima o ponto $v(A - B)$ da seguinte forma: em cada iteração é construído um simplexo, contido em $A - B$ e que é mais próximo da origem do que o simplexo construído na iteração anterior (ver a figura 3.6). Um simplexo é o fecho convexo de

um conjunto de vértices independentemente afins. Por conseguinte, um simplexo em 3D pode ter de um até quatro vértices, ou seja, pode ser um ponto, um segmento de reta, um triângulo ou um tetraedro.

Define-se W_k como o conjunto de vértices do simplexo construído na k-ésima iteração ($k \geq 1$), e v_k como $v(conv(W_k))$, o ponto do simplexo mais próximo da origem. Inicialmente, tomamos $W_0 = \emptyset$, e v_0, um ponto arbitrário em $A - B$. Já que $A - B$ é convexo e $W_k \subseteq A - B$, observe que $v_k \in A - B$, e assim $\|v_k\| \geq \|v(A - B)\|$ para todo $k \geq 0$. Então o comprimento de v_k é um limite superior para a distância entre A e B.

GJK constrói cada simplexo com pontos de suporte que são obtidos do mapeamento de suporte de $A-B$. Em cada iteração se insere um novo ponto de suporte $w_k = S_{A-B}(-v_k)$ como um vértice no simplexo corrente W_k. O ponto $v_{k+1} = v(conv(W_k \cup w_k))$, é o ponto mais próximo da origem do simplexo corrente. E W_{k+1} é o menor conjunto $X \subseteq W_k \cup w_k$, tal que v_{k+1} é contido em $conv(X)$, onde X pode ser visto como um conjunto independentemente afim.

Portanto, no decorrer das iterações são inseridos vértices no simplexo e vértices inseridos anteriormente são descartados. No entanto, este algoritmo, assim como o algoritmo para encontrar um eixo de separação, pode não convergir rapidamente. Uma condição de término é quando o vértice v_k e o vértice v_{k+1} são o mesmo, mas quando objetos estão muito próximos, a convergência pode ser muito lenta, o que obriga o uso de um critério de parada. Para objetos convexos em geral, pode-se escolher $\|v_k\|^2 - v_k \cdot w_k \leq \delta^2, \delta > 0$.

3.2.4.1 Verificação de interseção

Para determinar se dois objetos convexos se intersectam, não é preciso conhecer a distância entre eles, mas simplesmente saber se a distância é igual a zero ou não. Como um limite inferior para a distância temos a distância com sinal desde a origem ao plano de suporte $H(-v_k, v_k \cdot w_k)$. Por conseguinte, se $v_k \cdot w_k > 0$, a origem está do lado positivo do plano de suporte e o CSO de A e B está no lado negativo do plano de suporte, o que nos diz que o vetor v_k é um eixo de separação de A e B.

Em geral, GJK precisa de poucas iterações para encontrar um eixo de separação entre objetos que não se intersectam. Por exemplo, a figura 3.6 mostra que um eixo de separação é encontrado em $k = 2$. Para objetos que se intersectam, o algoritmo termina com $v_k = 0$.

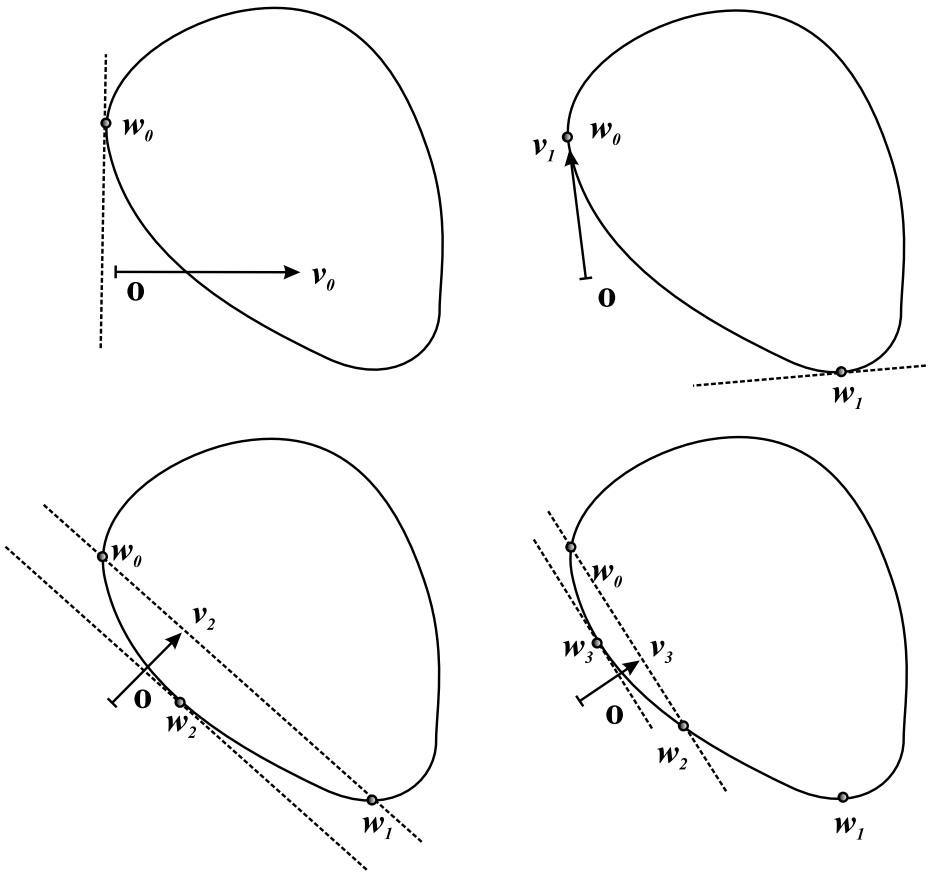

Figura 3.6: Quatro iterações do algoritmo de GJK.

3.2.4.2 Profundidade de penetração

Van den Bergen [40] apresentou um método iterativo para computar a profundidade de penetração de um par de objetos intersectados. Este método é parecido com o método original de GJK. Como no algoritmo GJK, este método usa o mapeamento de suporte para ler a geometria dos objetos. No entanto, este método requer como pré-requisito um simplexo que contenha a origem e que esteja contido no CSO de A e B.

O algoritmo GJK termina quando é gerado um simplexo que contém a origem, freqüentemente este simplexo é um tetraedro. Então, um tetraedro gerado pelo algoritmo GJK é um politopo inicial apropriado para ser usado neste método.

A profundidade de penetração é definida pelo ponto sobre a superfície do CSO de A e B que está mais próximo à origem. Este método usa uma estratégia iterativa, que consiste em selecionar a face do politopo mais próxima da origem, e subdividir o politopo usando pontos de suporte como vértices adicionais (ver a figura 3.7). O algoritmo termina quando é computado um vértice $v_{k+1} = v_k$. Entretanto, isto também sofre de problemas de convergência, requerendo que se avalie $\frac{v_k \cdot w_k}{\|v_k\|} - \|v_k\| \leq \delta, \delta > 0$, como condição de

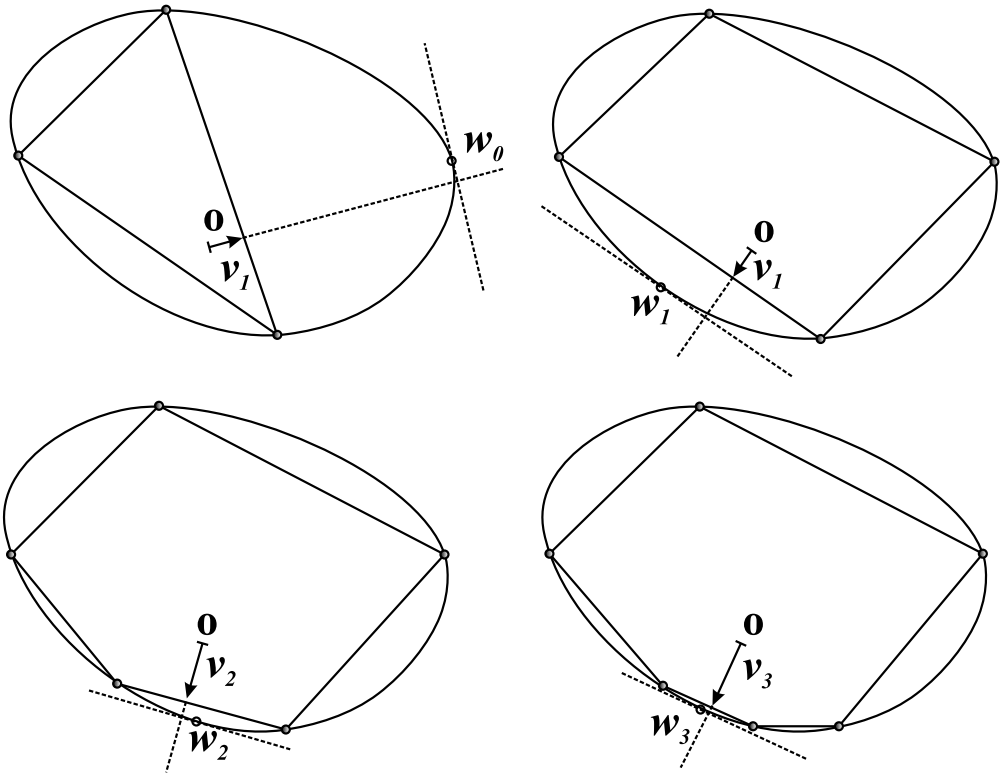

Figura 3.7: Uma seqüência de iterações do algoritmo para computar a profundidade de penetração. A seta v_k denota um ponto na superfície do politopo mais próximo da origem, w_k é um vértice de suporte, e as linhas pontilhadas representam o plano de separação $H(v_k, -v_k \cdot w_k)$.

terminação.

3.3 Estruturas de dados espaciais

Em ambientes complexos compostos por milhares de objetos, onde cada objeto pode ser composto por milhares de primitivas (elementos), o número de pares de primitivas que precisam ser verificadas no processo de detecção de colisão pode ser muito grande. Por tal razão, é preciso que se use estruturas de dados que ajudem a minimizar o número de verificações de interseção entre objetos.

Nesta seção discutiremos algumas estruturas de dados hierárquicas comumente usadas para acelerar o processo de detecção de colisão. Estas estruturas de dados têm dois propósitos essenciais:

1) Reduzir o número de verificações de interseção entre objetos estáticos e em movimento num ambiente. Como para n objetos há potencialmente $\binom{n}{2} = \frac{1}{2}n(n-1)$ pares de objetos a serem testados, estruturas de busca podem ser usadas para rejeitar

rapidamente a maioria dos pares que não se intersectam.

2) Reduzir o número de verificações entre pares de primitivas que podem se intersectar, na verificação de interseção de dois modelos complexos ou de um modelo complexo com uma primitiva (elemento) de outro modelo.

Uma consideração importante neste contexto é a *coerência geométrica*, já que ela permite rejeitar pares de objetos baseando-se na região do espaço que os objetos ocupam. Estruturas de dados espaciais, tais como *grades de voxels*, *hierarquias de particionamento do espaço* e *hierarquias de volumes limitantes*, podem ser usados para "explorar" coerência geométrica. Estas estruturas de dados tipicamente se dividem em duas categorias: *particionamento do espaço* e *particionamento do modelo*.

3.3.1 Partição do espaço

Uma partição do espaço é o processo pelo qual o espaço é subdividido em regiões convexas, chamadas *células*. Cada célula na partição mantém uma lista de referências a objetos que nela estão (parcialmente) contidos.

Uma estrutura de dados que representa um particionamento do espaço permite que pares de objetos sejam rejeitados rapidamente na verificação de interseção, já que é preciso verificar apenas pares de objetos que compartilham uma mesma célula.

3.3.1.1 Grades de voxels

Uma grade de voxels ou simplesmente grade, é uma partição uniforme do espaço em células retangulares. Uma grade é representada por uma caixa alinhada com os eixos coordenados, fechando todos os objetos do modelo. Internamente, é um arranjo tridimensional de células de tamanho $N_1 \times N_2 \times N_3$, onde N_1, N_2, e N_3 são o número de subdivisões ao longo dos eixos coordenados.

Cada célula mantém uma lista dos objetos que a intersectam. Portanto, inserir, mover ou remover um objeto na grade se resume em atualizar a lista de células que cobre o objeto.

As grades de voxels são úteis para rejeitar rapidamente pares de objetos que não se intersectam. Porém, sofrem de um grande problema: objetos que ocupam muitas células prejudicam o desempenho da estrutura.

Grades de voxels têm sido recomendadas para verificação de interseções em ambientes complexos [16] contendo milhares de objetos rígidos, onde as primitivas de cada objeto

são mantidas por grades que estão alinhadas com o sistema de coordenadas do objeto.

Um algoritmo que usa grades deverá achar todos os pares de células (em grades alinhadas relativamente) não vazias que se intersectam, e verificar interseção entre as primitivas contidas nestas células. Os benefícios de se usar uma grade de voxels são: pouco uso de memória no armazenamento e acesso rápido às células.

A maior desvantagem das grades de voxels é que o desempenho é altamente dependente da densidade dos objetos no espaço, isto é, depende de como os objetos estão distribuídos no espaço. Por exemplo, se há muitos objetos agrupados em poucas regiões do espaço, então poucas células conterão a maioria dos objetos e a maior parte das células ficará vazia. Em tais casos, uma grade não é muito útil para a rejeição de pares de objetos na verificação de interseção.

Outro problema é escolher uma resolução adequada para a grade. Se usarmos uma grade com poucas células, então as células ficarão muito cheias e muitas verificações de interseção serão executadas para pares de objetos distintos da célula. Se, por outro lado, a grade tiver uma resolução fina, então a grade precisará de muito espaço de armazenamento, e muitas células manterão referências a objetos idênticos. Além disso, mover um objeto grande no espaço da grade pode ser bastante custoso, já que o objeto pode estar coberto por muitas células. Grades têm bom desempenho quando um modelo é composto por um conjunto de primitivas geometricamente coerentes com densidade e tamanho uniformes. Porém, na prática, poucos modelos satisfazem estas condições. Por conseguinte, esquemas de particionamento adaptativo, usando particionamento recursivo do espaço costumam ter melhor desempenho.

3.3.1.2 Octrees e k-d trees

Octrees e k-d trees são estruturas de dados hierárquicas para particionar o espaço 3D em células retangulares. Cada nó na árvore corresponde a uma região retangular do espaço, isto é, um paralelepípedo. O nó raiz corresponde ao espaço do modelo, e é representado por uma caixa retangular que envolve completamente o modelo. Cada nó interno na árvore representa uma subdivisão da sua região em regiões menores, às quais correspondem os nós filhos. As regiões dos nós folha são células nas quais os objetos ou primitivas são mantidos.

Octrees e k-d trees diferem na forma como as regiões são subdivididas. Numa octree cada nó interno divide sua região ao longo dos três eixos obtendo oito sub-regiões (octantes) de mesmas dimensões. Numa k-d Tree, o espaço é divido em duas partes (não

necessariamente iguais) por um plano ortogonal a um dos três eixos coordenados. A figura 3.8 mostra uma representação visual de ambas as estruturas.

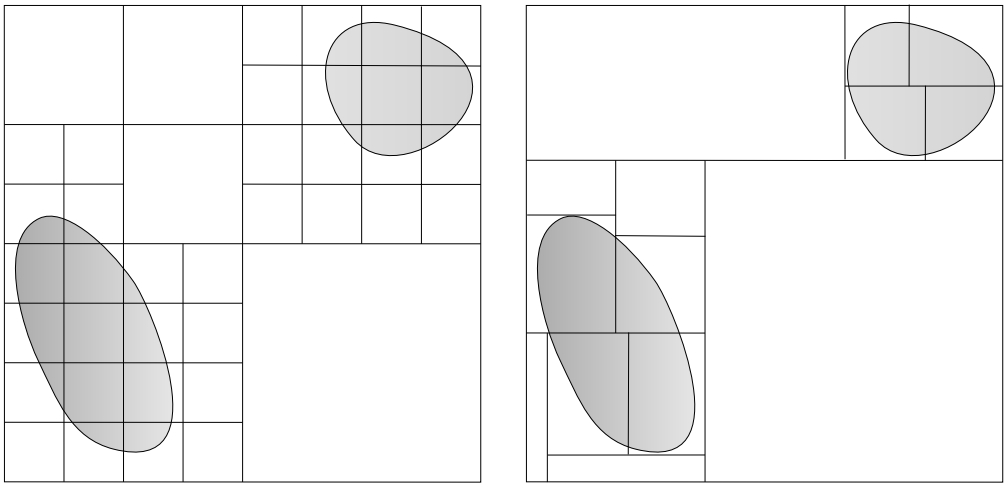

Figura 3.8: Particionamento de um modelo 2D usando uma quadtree (esquerda) e uma k-d tree (direita).

k-d trees são mais flexíveis que octrees já que podem se adaptar melhor a objetos com relações de aspecto variadas. Também têm a vantagem de ser extensíveis a espaços de maior número de dimensões. Por outro lado, a subdivisão regular das octrees levam a diversos algoritmos eficientes de busca e de construção [35].

3.3.1.3 Árvores de Partição binária do espaço (BSP-Trees)

Uma BSP-Tree é uma estrutura hierárquica para particionar o espaço recursivamente em células convexas. Cada nó interno na árvore divide sua região convexa associada em duas regiões, usando um plano de orientação livre. Uma variante bastante popular é a k-d tree, onde os planos são restritos a orientações perpendiculares aos eixos coordenados.

BSP-Trees podem ser usadas para representar estruturas de células pelos quais objetos se movimentam. Porém, uma aplicação de maior interesse para as BSP-Trees é servir como representação para poliedros côncavos. De fato, um poliedro côncavo pode ser representado como a união de um subconjunto de células de uma BSP-Tree. Para este propósito, as folhas da BSP-Tree são rotuladas como *dentro* ou *fora*, conforme a célula correspondente esteja dentro ou fora do poliedro [37] (ver a figura 3.9).

Uma variante da BSP-Tree, é a *BSP-Tree de orientação discreta* (DOBSP-Tree), na qual as orientações dos planos de particionamento são escolhidos de um conjunto finito de orientações. As DOBSP-Trees requerem menos armazenamento do que BSP-Trees comuns, pelo que são adequadas para muitas aplicações.

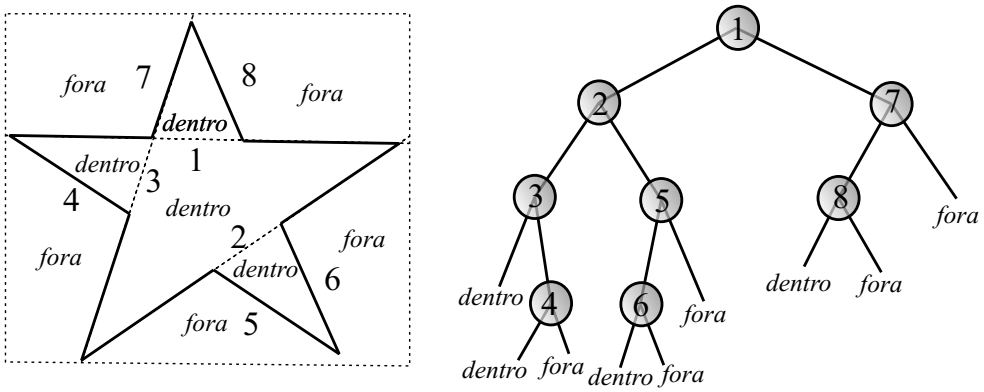

Figura 3.9: Um polígono e sua representação BSP-tree correspondente.

3.4 Métodos de partição

Para detecção de colisão, freqüentemente é melhor usar uma partição do modelo do que uma partição do espaço, já que as estruturas de partição do modelo não sofrem os mesmos problemas que as estruturas de partição do espaço, por exemplo, ter múltiplas referências ao mesmo objeto.

A estratégia é subdividir o modelo em subconjuntos de objetos geometricamente coerentes e computar um volume limitante tão ajustado quanto possível a cada subconjunto de objetos, tal que em verificações de interseção, subconjuntos de objetos possam ser rejeitados rapidamente se seus volumes limitantes não se intersectam.

3.4.1 Volumes Limitantes

Um volume limitante de um modelo é uma forma primitiva que limita o modelo e deve ter as seguintes propriedades:

- O volume limitante deve envolver o modelo de forma tão justa quanto possível para que numa consulta de interseção tenha pouca probabilidade de intersectar o volume mas não o modelo.

- Verificações de interseção entre volumes limitantes devem ser computacionalmente simples. Em particular, elas devem ser muito menos complexas do que as verificações de interseção entre os modelos.

- O volume limitante deve ocupar pouco espaço de armazenamento.

- O custo de computar um volume limitante para um modelo deve ser baixo. Esta

propriedade é ainda mais importante se o volume tiver que ser recomputado constantemente.

3.4.1.1 Esferas Limitantes

As esferas são o tipo de volume limitante mais simples que existe, sendo por isso bastante populares. Uma esfera pode ser armazenada usando apenas *quatro* escalares. Para verificar uma interseção entre duas esferas são necessárias apenas *onze* operações aritméticas.

Para muitos modelos, as esferas não são o tipo de volume que limita o objeto de forma mais justa. Porém, sua simplicidade e o fato de que elas são invariantes às rotações fazem com que sejam um tipo de volume limitante popular em ambientes dinâmicos.

3.4.1.2 Caixas limitantes alinhadas aos eixos coordenados

As caixas limitantes alinhadas aos eixos coordenados (Axis Aligned Bounding Boxes - AABBs) são usadas ainda mais extensivamente do que as esferas limitantes. Embora elas precisem de mais espaço de armazenamento do que as esferas (na verdade, seis escalares), com as AABBs pode-se verificar uma interseção usando apenas *seis* operações primitivas. Porém, na média, elas também não limitam adequadamente os modelos [38].

3.4.1.3 Politopos de orientação discreta (DOPs)

Este tipo de volumes limitantes são uma extensão das AABBs, e podem limitar um modelo de maneira mais justa do que as AABBs usando eixos adicionais – k eixos no total – daí o nome popular k-DOPs. Uma AABB, portanto, é equivalente a uma 6-DOP [28]. Diferentemente das esferas, entretanto, as AABBs e k-DOPs não são invariantes a rotações. Elas devem ser recomputadas cada vez que o modelo limitado sofre uma rotação. Não obstante, isto pode ser feito de forma razoavelmente rápida, computando simplesmente os intervalos da projeção dos objetos sobre os eixos. Os intervalos de projeção podem ser computados diretamente usando o "mapeamento de suporte". Isto é, o intervalo de projeção para um eixo v de A é dado por $[v \cdot S_A(-v), v \cdot S_A(v)]$.

3.4.1.4 Caixas limitantes orientadas

As caixas limitantes orientadas (Oriented Bounding Boxes - OBBs) não são o tipo de volume limitante mais econômico em termos de custo de armazenamento e custo computacional para verificação de interseção. Porém, este tipo de volume limitante, em contraste com os outros antes mencionados, aproxima de maneira bastante justa a superfície

dos modelos, justificando portanto o alto custo computacional e de armazenamento. A orientação da OBB é normalmente representada por uma matriz 3×3, a qual define uma orientação com respeito ao sistema de coordenadas do modelo limitado. Ainda que uma representação por ângulos de Euler ou por quatérnios possam ser usadas, é preferível usar uma matriz, já que esta é mais eficiente em verificações de interseção [19].

3.4.2 Hierarquias de volumes limitantes

Uma hierarquia de volumes limitantes é uma estrutura hierárquica (árvore), na qual primitivas são armazenadas nas folhas. Cada nó mantém um volume que limita um subconjunto de primitivas, as quais são representadas pelo nó. A maior vantagem de se usar hierarquias de volumes limitantes é que pode-se executar verificações rápidas de interseção, e o requerimento de espaço é linear no número de objetos no modelo. Sua maior desvantagem reside no alto custo de construção e de atualização.

Freqüentemente, as hierarquias de volumes limitantes são usadas para explorar a coerência geométrica dos modelos. Na literatura, diversas abordagens foram propostas, sendo que as de maior importância são: AABB-Trees, OBB-Trees, k-DOP Trees e Sphere-Trees.

3.4.2.1 AABB-Trees

AABB-Trees são hierarquias de caixas limitantes alinhadas com os eixos do sistema de coordenadas do modelo, ou seja, todas as caixas na hierarquia têm a mesma orientação [11]. Van den Bergen [38] mostrou que as AABB-Trees podem ser tão eficientes quanto outros tipos de hierarquias, e podem ser atualizados rapidamente. Assim, as AABB-Trees podem ser usadas para modelar objetos rígidos e deformáveis. Freqüentemente as AABB-Trees são construídas de "cima para baixo", usando uma subdivisão recursiva do modelo. Em cada passo da recursão, é computada a AABB do conjunto de primitivas, e esta é dividida em dois subconjuntos por um plano cuidadosamente escolhido. Este processo continua até que cada subconjunto contenha apenas um elemento. Os planos divisores devem ser escolhidos de forma a obter uma partição aproximadamente eqüitativa e, desta forma, gerar uma árvore balanceada. Freqüentemente é usado um plano de particionamento ortogonal ao eixo de maior comprimento de uma AABB, para tentar garantir células aproximadamente cúbicas.

3.4.2.2 OBB-Trees

Gottschalk et al. [19] propuseram uma estrutura de dados hierárquica de caixas limitantes orientadas (OBBs) chamada *OBB-Tree*. Sua maior contribuição reside nos algoritmos de construção e atualização da árvore. Apesar de serem bastante adaptáveis às geometrias dos objetos encapsulados, e de suportarem algoritmos de complexidades assintóticas comparáveis aos de outras estruturas, as OBBs são menos populares que AABBs e esferas.

3.4.2.3 Sphere-Trees

Hierarquias de esferas limitantes têm sido extensivamente usadas em detecção de colisão [34, 7]. As esferas têm a útil propriedade de serem invariantes a rotações e, conseqüentemente, estruturas hierárquicas baseadas em esferas podem ser atualizadas de maneira mais eficiente do que outras hierarquias de volumes limitantes (OBB-Trees, k-DOP-Trees (k>6)), especialmente quando os objetos encapsulados são deformáveis, isto é, não rígidos.

São conhecidas duas variantes deste tipo de estruturas: hierarquias *wrapped*, onde as esferas envolvem a geometria do modelo de maneira justa; e hierarquias *layered*, no qual as esferas dos nós pai simplesmente envolvem as esferas dos nós filhos (ver a figura 3.10). Por conseguinte, a construção de uma hierarquia *layered* toma tempo $O(n)$, e a construção de uma hierarquia *wrapped* requer tempo $O(n \log n)$, já que para construir a esfera para um nó interno, todas as folhas descendentes precissam ser visitadas.

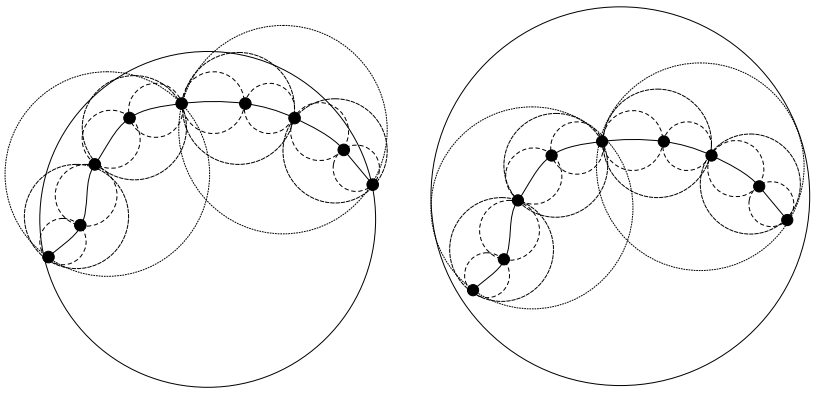

Figura 3.10: Dois tipos de hierarquias baseadas em esferas: Wrapped Sphere-Tree (esquerda) e Layered Sphere-Tree (direita)

Agarwal et al. [1] apresentaram algoritmos para atualizar hierarquias *wrapped*, onde o custo de atualização esperado é linear. O método é aplicável a objetos deformáveis

semelhantes a colares, tratando colisões e auto-interferência.

James e Pai [25] propuseram um esquema de detecção de colisão para objetos deformáveis baseado numa variante de Sphere-Tree chamada *Bounded Deformation Tree* (BD-Tree). Esta variante é usada para detecção de colisão entre objetos sujeitos a deformações com amplitude reduzida. A principal vantagem desta estrutura é que suporta detecção de colisão em tempo dependente apenas da complexidade da colisão. Este método, porém, está restrito a uma classe de deformações de relativamente pequenas dimensões, não podendo ser usada para objetos deformáveis em geral. As esferas da estrutura podem ser atualizadas após uma deformação em qualquer ordem (por exemplo, da raiz para as folhas, ou vice-versa), e a um custo que é independente da complexidade da geometria do modelo.

Capítulo 4

Descrição do sistema proposto

O sistema desenvolvido lida com objetos rígidos, objetos parcialmente deformáveis (por exemplo, esteiras e faixas) e objetos completamente deformáveis (por exemplo, cordas e tecidos). Para este propósito, distinguimos dois tipos de representação para cada objeto:

1) Uma representação de sua geometria;

2) Uma representação de suas propriedades físicas.

A primeira é usada para exibir os objetos e executar o processo de detecção de colisão entre objetos, enquanto que a segunda é responsável por estimar como os objetos devem reagir a estímulos físicos, tais como colisões ou forças gravitacionais.

Todos os objetos exceto cordas (que são representadas por linhas poligonais) são representadas geometricamente por superfícies trianguladas. Dessa forma, por exemplo, um cubo é representado por 8 vértices, 12 triângulos (faces) e 18 arestas, e retalhos de tecido são representados por superfícies trianguladas regularmente.

Para a representação das propriedades físicas, todos os objetos são modelados como sistemas de partículas. Em particular, todas as partículas coincidem com os vértices da sua representação geométrica e têm massa idêntica. Adicionalmente, cada tipo de objeto é modelado usando um conjunto de restrições lineares e/ou angulares. Dessa forma, por exemplo, as restrições lineares de um cubo rígido coincidem com as arestas da representação geométrica e adicionalmente são usadas duas restrições coincidentes com diagonais principais da representação geométrica para garantir a rigidez. No entanto, um retalho de tecido contém só restrições lineares coincidindo com as arestas da sua representação geométrica. De forma semelhante, uma corda tem restrições lineares para cada segmento de reta da sua representação geométrica.

Em nossa implementação, a geometria de cada objeto é representada por duas estruturas de dados. A primeira é uma estrutura de dados *halfedge* [30] a qual suporta operações

como: percorrer as faces do objeto, percorrer arestas, percorrer vértices, e determinar relações de incidência e vizinhança entre estes elementos.

A segunda é uma estrutura de dados hierárquica *Sphere-Tree* [1] a qual é usada no mecanismo de detecção de colisão.

As propriedades físicas do objeto são representadas principalmente pelas restrições lineares. Por conveniência, a massa de cada partícula é armazenada como atributo do vértice correspondente na estrutura de dados halfedge.

4.1 Detecção de colisão

Em resumo, o problema de detecção de colisão consiste em estabelecer para os objetos na cena todos os pares de objetos da forma (A, B), $A \neq B$ tal que $A \cap B \neq \emptyset$. Em nosso sistema isto é efetuado enumerando todos pares de objetos para verificar intersecção entre eles. A rigor, isto não é muito eficiente visto que obriga a verificação de $O(n^2)$ pares, onde n é o número de objetos na cena. No entanto, estamos interessados na simulação física de um número relativamente pequeno de objetos. Dessa forma, na prática, o desempenho da simulação não é comprometida, desde que a verificação de intersecção entre dois objetos seja executada eficientemente.

Nosso sistema emprega dois esquemas para a detecção de colisão entre objetos.

- O primeiro é uma adaptação do algoritmo de GJK [18, 38] e é empregado quando precisamos detectar uma colisão entre dois objetos rígidos, quer dizer, não deformáveis.

- O segundo é um algoritmo baseado numa estrutura hierárquica *Sphere-Tree* [22, 23, 1, 25, 7] e é usada para detectar colisões quando pelo menos um dos objetos é deformável. Objetos deformáveis estão sujeitos a auto-intersecção, isto é, colisões de um objeto consigo mesmo. Tais colisões também são tratadas com o algoritmo baseado em Sphere-Trees.

4.1.1 O algoritmo de GJK adaptado

O algoritmo de Gilbert-Johnson-Keerthi [18] provê um método bastante eficiente para detectar colisões entre objetos convexos (ver a seção 3.2.4).

A idéia geral do algoritmo GJK é examinar a CSO de dois objetos determinados (A e B) procurando um simplexo que contenha a origem. Se esta busca produz uma resposta

negativa – quer dizer, a origem está fora da CSO – então os objetos não se intersectam. Neste caso, o ponto da CSO que é mais próximo da origem representa um eixo separador de A e B, e esta informação pode ser usada como ponto de partida para o processo de detecção de colisão nos quadros subseqüentes.

Por outro lado, se a busca teve sucesso, então os objetos se intersectam, e para efetuar a resposta à colisão, diversos detalhes com respeito à colisão devem ser computados. Por exemplo, esquemas típicos tentam determinar a *profundidade de penetração*, a qual requer que se descubra o ponto na superfície da CSO que está mais próxima da origem. Para este propósito, Van den Bergen [38] sugere usar um *algoritmo de expansão de politopos*. Isto é feito usando como politopo inicial o simplexo que contém a origem. Este é expandido inserindo novos vértices que ficam na superfície da CSO. Os vértices inseridos são obtidos usando o mapeamento de suporte de $A - B$ (CSO). A estratégia básica deste algoritmo é selecionar de maneira iterativa a face do politopo de expansão mais próxima da origem e subdividí-la inserindo novos vértices de suporte. Este processo termina quando o algoritmo encontra um novo vértice de suporte que é igual ao vértice de suporte anterior. A figura 3.7 mostra o comportamento deste algoritmo.

Nosso sistema, no entanto, computa uma outra informação. Trata-se da face (*face de colisão*) da CSO mais próxima da origem. Analisando os vértices desta face é possível determinar que elementos dos objetos estão envolvidos na colisão. Nós distinguimos dois casos principais: colisão aresta-aresta e colisão vértice-face*.

Para compreender como estes elementos são identificados, observe que cada vértice do CSO corresponde a um par de vértices (a_i, b_j), $a_i \in A$ e $b_j \in B$. Por exemplo, um vértice de A colidindo com uma face de B pode ser caracterizado tendo três vértices da face de colisão que correspondem ao mesmo vértice de A e três vértices diferentes de B (ver as figuras 4.1 e 4.2).

4.1.2 O uso da Sphere-Tree

A detecção de colisão entre objetos deformáveis é usualmente suportado por algum esquema hierárquico de volumes limitantes. Em geral, é preferível usar uma hierarquia *wrapped* em vez de uma hierarquia *layered*, já que uma hierarquia *wrapped* se ajusta melhor à superfície do objeto do que uma hierarquia *layered*. Durante a simulação, um aspecto importante é o que diz a respeito das estratégias de atualização das *Sphere-Trees*

*Embora outras configurações sejam possíveis – por exemplo, aresta-face ou vértice-aresta – estas podem ser reduzidas aos dois casos mencionados.

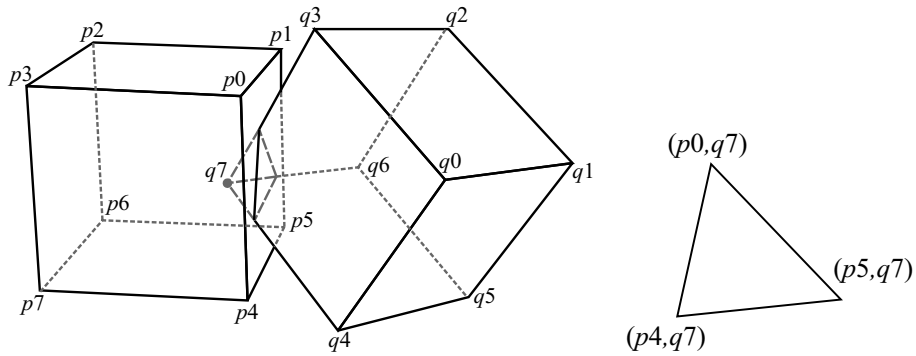

Figura 4.1: Uma colisão vértice-face e a correspondente face de colisão no CSO.

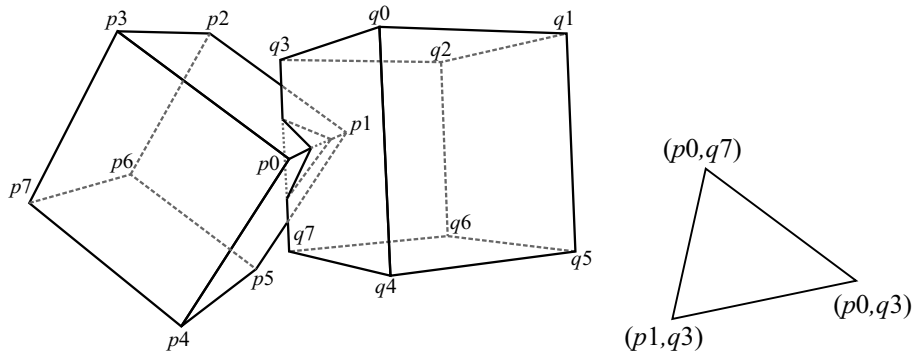

Figura 4.2: Uma colisão aresta-aresta e a correspondente face de colisão no CSO.

dos objetos, já que após um passo de tempo os objetos podem se mexer ou se deformar. Em particular, se o objeto envolvido é não-deformável, ambos os tipos de hierarquia suportam um processo de atualização que pode ser executado de *cima para baixo* numa maneira "preguiçosa", quer dizer, os filhos de um nó precisam ser atualizados apenas se foi determinado que o nó intersecta outro nó. No pior caso, a atualização requer tempo $O(n)$ se todos os nós-folhas participam na colisão, mas freqüentemente, quando apenas um ou um número pequeno de folhas estão envolvidos, a atualização pode tomar tempo $O(\log n)$. Hierarquias *wrapped*, por outro lado, podem ser atualizadas em tempo esperado $O(n)$, para alguns tipos de objetos como colares [†], cordas, proteínas [1], entre outros. Mas para suportar objetos deformáveis em geral é preciso gastar um tempo $O(n \log n)$ no pior caso. Para estes objetos, uma Sphere-Tree *layered* a qual pode ser atualizada num tempo $O(n)$ é freqüentemente a melhor escolha.

Nosso sistema mantém uma Sphere-Tree para cada objeto. Em particular, hierarquias *wrapped* são usadas para objetos rígidos e hierarquias *layered* são usadas para objetos deformáveis.

[†]James e Pai [25] reportaram um algoritmo de atualização num tempo sub-linear, o qual é restrito a objetos com deformações reduzidas (em oposição a deformações em geral).

Algoritmo de detecção de colisão Para detecção de colisão entre dois objetos deformáveis é empregado um algoritmo tradicional de detecção de colisão [1] (ver o algoritmo 1).

Algoritmo 1 Detecção de colisão entre duas Sphere-Trees.

```
Q = {(S_A, S_B)}
while Q ≠ ∅ do
  (u, v) = dequeue(Q)
  if C_u ∩ C_v ≠ ∅ then
    if u ∉ {v, adj(v)} then
      return colisão
    else
      if u =Split(u, v) then
        enqueue((left(u), v), Q)
        enqueue((right(u), v), Q)
      else
        enqueue((u, left(v)), Q)
        enqueue((u, right(v)), Q)
      end if
    end if
  end if
end while
```

Este algoritmo é baseado num processo recursivo descendente de duas hierarquias. Em cada passo, uma verificação de interseção é executada para um par de nós-esfera (S_A, S_B), onde S_A (ou S_B) é a esfera limitante da Sphere-Tree correspondente ao objeto A (respectivamente B). Se um nó folha é visitado durante o processo, o elemento do objeto representado no nó folha é submetido a uma verificação de intersecção exata.

Construção das Sphere-Trees. As Sphere-Trees são construídas numa maneira similar à relatada no trabalho de Quinlan [34]. Diversas metodologias podem ser empregadas para este propósito. Por exemplo, para o caso de um objeto, seja S o conjunto de todos os centróides das faces da superfície do objeto, e seja A a caixa mínima que limita S e é alinhada com os eixos coordenados. Então podemos usar diversas abordagens, como as relatadas a seguir:

1) Dividimos A pelo eixo de maior comprimento em duas caixas de igual tamanho, separando S em dois subconjuntos (os quais serão nós internos). Este processo é feito recursivamente até que as caixas contenham só um elemento (estes são nós folhas).

2) Ordenamos S com respeito ao eixo de maior comprimento de A e subdividimos S em dois subconjuntos. Este processo também é feito recursivamente até que os

subconjuntos S' contenham um só elemento.

3) De modo semelhante ao anterior, ordenamos S com respeito ao eixo de maior comprimento de A é usamos a mediana para subdividir S [‡].

Um nó-folha é construído para cada elemento de S, usando a referência que este tem à face f_i de onde provém. A *esfera mínima* é então construída usando os vértices de f_i, e são armazenados os *pontos de suporte* do nó-esfera folha. Note que uma *esfera mínima* para um conjunto de pontos pode ser construída como a esfera que contém um subconjunto de no máximo quatro pontos, os quais são nomeados de *pontos de suporte* [42, 17].

Porém, um nó interno é construído de duas maneiras, dependendo da hierarquia usada:

- Se a hierarquia for *wrapped*, se computa a esfera mínima que limita os pontos dos elementos que estão limitados pelos nós-filho, e são armazenados os pontos de suporte da esfera mínima.
- Se a hierarquia for *layered*, é computada a esfera mínima que limita as esferas dos nós-filho.

Atualização. Durante a simulação, Sphere-Trees associadas a objetos deformáveis são atualizadas de *baixo para cima*. Os nós folhas são atualizados construindo-se novas esferas limitantes, e os nós internos são atualizados construindo-se uma nova esfera mínima que envolve as esferas de seus nós filhos. Para Sphere-Trees associadas a objetos rígidos, o processo de atualização é feito de *cima para baixo*, já que pode se tirar vantagem do fato que as esferas dos nós podem ser construídas a partir de no máximo quatro pontos de suporte. Como mencionado acima, durante a construção de uma Sphere-Tree, os pontos de suporte são referências a partículas da geometria do objeto. Deste modo, quando as partículas se mexem, as esferas envolventes são re-computadas em tempo $O(1)$.

4.2 Resposta às colisões

A resposta a uma colisão é composta de dois passos. O primeiro passo consiste em separar os elementos dos objetos (vértice, aresta ou face) que estão se intersectando. Este passo é puramente geométrico, já que é executado movendo as partículas na geometria

[‡]nenhuma de estas abordagens é eficiente para todos os tipos de objetos em geral, já que depende da distribuição das partículas no objeto.

da colisão. O segundo passo é um processo de relaxamento iterativo no qual os elementos dos objetos que estão envolvidos na colisão encontram as suas posições apropriadas usando as restrições como suporte.

4.2.1 Processo de separação

Se dois objetos não-deformáveis A e B colidem, o processo de separação requer três parâmetros, os quais são computados pelo mecanismo de detecção de colisão. O primeiro é p_A, o ponto de contato em A, o segundo é p_B, o ponto de contato em B e o terceiro é o que nós chamamos de *ponto de projeção*, ou seja, um ponto q no espaço onde a simulação assume que os dois objetos estão em contato. Jakobsen descreve de maneira correta um esquema simples para resposta a colisões [24], movendo os vértices (partículas) penetrantes por projeção, no entanto, não são mostrados detalhes sobre os parâmetros requeridos para computar este processo.

Analisamos aqui como o sistema computa a resposta à colisão para um objeto A. Uma vez que p_A e q são identificados, p_A é movido para a posição de q. Se o elemento é um vértice, então isto é trivial. Se o elemento é uma aresta ou uma face, então seus vértices devem ser movidos de uma forma apropriada (ver a figura 4.3)

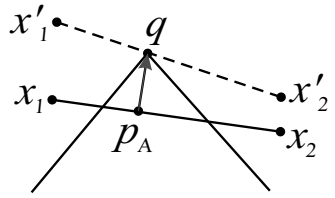

Figura 4.3: Resposta à colisão de uma aresta.

Já que todas as partículas do objeto A têm massa idêntica, a nova posição do elemento depende só da sua geometria. Por exemplo, na Figura 4.3, se x_1 e x_2 são os pontos finais de uma aresta e $p_A = (1-\alpha)x_1 + \alpha x_2$, então Jakobsen computa as novas posições da aresta como

$$x_1' = x_1 + \frac{1-\alpha}{(1-\alpha)^2+\alpha^2}(q - p_A) \quad (4.1)$$

$$x_2' = x_2 + \frac{\alpha}{(1-\alpha)^2+\alpha^2}(q - p_A). \quad (4.2)$$

Para o caso 3D, o processo de separação é feito de forma similar. Em geral, duas construções são possíveis dependendo do tipo de colisão – vértice-face ou aresta-aresta – .

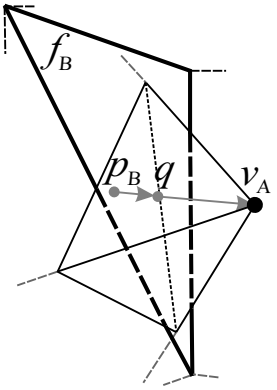

Figura 4.4: Colisão vértice-face.

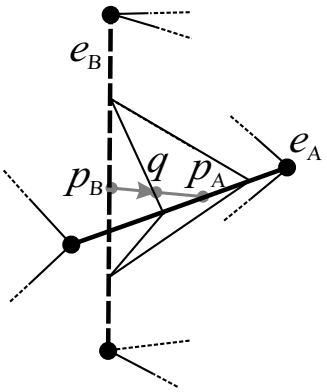

Figura 4.5: Colisão aresta-aresta.

Se uma colisão ocorre entre um vértice de $v_A \in A$ e uma face $f_B \in B$ (ver a figura 4.4), então $p_A = v_A$ e p_B é um ponto sobre f_B o qual é mais próximo a p_A. O ponto q esta situado sobre o segmento de reta $p_A p_B$, e a posição exata de q depende das massas dos corpos A e B:

$$q = \frac{m_B}{m_A + m_B} p_A + \frac{m_A}{m_A + m_B} p_B. \tag{4.3}$$

Quando ocorre uma colisão entre as arestas e_A e e_B, a construção é similar (ver a figura 4.5). p_A é o ponto de e_A mais próximo de e_B e p_B é o ponto de e_B mais próximo de e_A, enquanto q é computado pela Eq. 4.3.

Quando uma colisão ocorre entre dois objetos deformáveis, então o processo de separação é feita de maneira menos elaborada. Em particular, a detecção de colisão baseada em Sphere-Trees se resume a detectar a intersecção entre duas esferas limitantes correspondentes a nós-folha da Sphere-Tree, isto é, o sistema não faz um teste de colisão exata nos elementos envolvidos pelas esferas. Este esquema é razoável para simulação de cordas e retalhos de tecido, já que a sua geometria consiste de uma amostra relativamente fina, e portanto a esfera de um nó-folha provê uma boa aproximação para um elemento

do objeto. Consideremos dois elementos $f_A \in A$ e $f_B \in B$ cujas esferas limitantes s_A e s_B se intersectam (ver a figura 4.6).

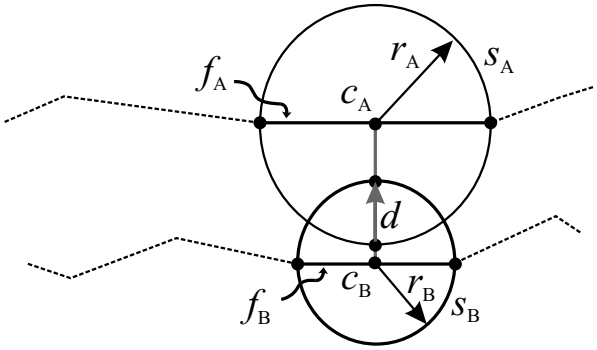

Figura 4.6: Separação de dois elementos limitados por esferas limitantes.

Sejam c_A (c_B) o centro e r_A (r_B) o raio de s_A (s_B). Então todas as partículas de f_A são deslocadas de $-\vec{d}/2$ e todas as partículas de f_B são deslocadas de $\vec{d}/2$, onde $\vec{d}$ é o vetor dado por

$$\vec{d} = \left(\frac{(r_A + r_B)}{||c_B - c_A||} - 1 \right) (c_B - c_A). \tag{4.4}$$

Finalmente, quando ocorre uma colisão entre um objeto deformável A e outro não-deformável B, o processo para separar os elementos que colidem é uma combinação dos casos que foram descritos antes. O sistema considera a intersecção entre uma esfera limitante S_A e um elemento f_B como uma colisão só se C_A, o centro de S_A, está dentro de B (ver a figura 4.7).

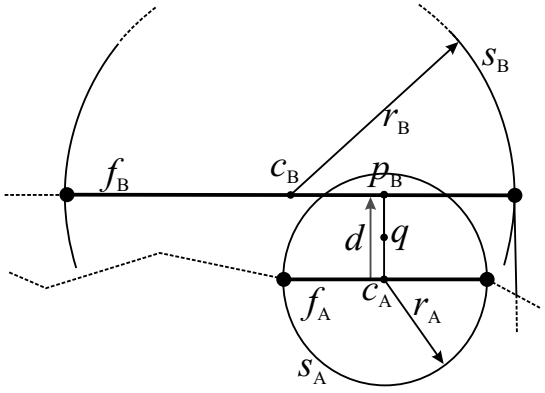

Figura 4.7: Separação de dois elementos pertencentes, um a um objeto não-deformável, e o outro a um objeto deformável.

O elemento f_B é movido como no primeiro caso (objeto não-deformável), enquanto que o elemento f_A é movido como no segundo caso (objeto deformável). O ponto de projeção q é computado como o ponto médio do segmento de reta $c_A p_B$, onde o ponto

de contato p_B é o ponto de f_B mais próximo de c_A. Finalmente, o vetor $\vec{d}$ é computado simplesmente por $p_B - c_A$.

4.2.2 Relaxamento iterativo

O processo de relaxamento é usado para restabelecer as restrições do sistema após o movimento das partículas devido a forças que agiram sobre elas ou como resultado do processo de separação de elementos. Deve-se assegurar a validade de uma restrição linear após o movimento de uma ou ambas as partículas.

Jakobsen [24] sugere que todas as restrições sejam processadas seqüencialmente um número pequeno de vezes (por volta de dez) sem nenhuma ordem em particular. Infelizmente, se as restrições lineares são violadas, o relaxamento pode necessitar um número maior de iterações. Este, por exemplo, é o caso quando um objeto rígido que contém um grande número de restrições bate com alta velocidade numa parede firme. É razoável supor que, neste caso, as partículas que foram movidas como conseqüência do processo de projeção tenham suas restrições relaxadas antes das demais. Além disso, estas restrições provavelmente deverão ser processadas mais vezes do que as outras. Esta observação nos sugere que se use um esquema heurístico no qual as restrições dos objetos rígidos sejam relaxados de um modo diferente. Especificamente, o sistema emprega uma ordem de relaxamento para cada partícula do objeto, a qual é obtida a partir do *grafo de restrições* do objeto (ver a figura 4.8).

Se define o *grafo de restrições* do objeto A como G_A, onde para cada vértice (partícula) $v \in A$, é construída uma lista de restrições lineares L_v de maneira que todas as restrições lineares de A apareçam numa ordem crescente de distância – no sentido do caminho mais longo – de v no grafo de restrições [§]. Desta forma, por exemplo, para a partícula v na figura 4.8, o relaxamento deve ser feito segundo a lista ordenada de restrições L_v a qual contém $< l_1, l_5, l_2, l_6, l_7, l_4, l_3 >$. Além disso, as restrições lineares que correspondem às arestas vizinhas de v (por exemplo, l_1 e l_5) são repetidas n vezes na lista, onde n é a distância à restrição mais distante no grafo de restrições. De forma similar, aquelas arestas que são as vizinhas das vizinhas de v (l_2, l_6, l_7 e l_4 no exemplo da figura 4.8) são repetidas $n/2$ vezes e assim por diante, com arestas (restrições de distância) aparecendo pelo menos uma vez na lista L_v. Uma observação importante é que as arestas internas não estão ligadas com outras arestas, isto é, as arestas diagonais apenas mantêm informação

[§]Os vértices v_i são referências para partículas do objeto rígido e as arestas representam restrições lineares entre duas partículas vizinhas.

sobre seus dois vértices extremos.

O algoritmo 2 mostra como é construída uma lista de restrições para um vértice.

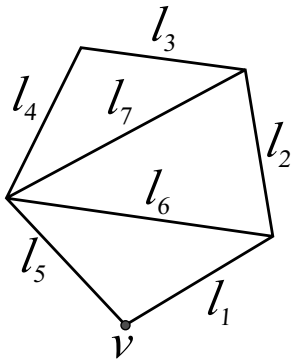

Figura 4.8: Exemplo de um grafo de restrições.

As listas de restrições são pré-computadas no início da simulação. Durante a simulação, ao se detectar uma colisão, são selecionadas as partículas envolvidas, e suas listas de restrições são usadas no processo de relaxamento. Mas, se o objeto não colide com nenhum outro, então suas restrições são relaxadas numa ordem simples, como feito no esquema do Jakobsen.

Algoritmo 2 Construção do grafo de restrições para um vértice.

$L =< \forall l \in A >$
$L_{v_i} =<>$
$Q =< v_i >$
while $Q \neq <>$ **do**
  $v \leftarrow$ dequeue(Q)
  while $\exists\, l \in L : l =< v, v' >$ **and** $v' \in adj(v)$ **do**
    enqueue(l, L_{v_i})
    $L \leftarrow L - \{l\}$
    enqueue(v', Q)
  end while
end while

Capítulo 5

Resultados

Um protótipo do sistema descrito foi implementado na linguagem C++ usando OpenGL [12] para desenhar as cenas e a biblioteca CGAL [9] (*Computational Geometry Algorithms Library*) a qual provê diversas estruturas de dados geométricas.

O sistema foi usado para realizar diversos experimentos num computador equipado com 512Mb de memória principal, um processador Pentium-IV 1.8 GHz, e uma placa gráfica ATI Radeon 9800 Pro.

O primeiro experimento consiste em uma cena simples com icosaedros rígidos (50), os quais são jogados dentro de um recipiente cúbico (ver a figura 5.1). A simulação roda em tempo real (alcançando uma execução de 30 fps aproximadamente). A figura 5.2 mostra um gráfico no qual a freqüência com que os quadros são exibidos é relacionada com o número de objetos na cena.

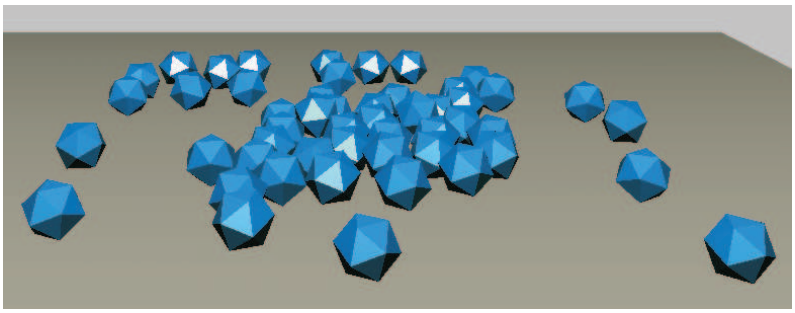

Figura 5.1: Uma cena simples com 50 icosaedros.

Em um experimento semelhante procurou se avaliar a efetividade do método de relaxamento heurístico proposto quando comparado com o esquema de relaxamento não ordenado do Jakobsen. Neste experimento, relaxamos as restrições dos objetos apenas se as restrições têm mais que o 1% do seu comprimento prescrito. O gráfico na figura 5.3 mostra o número total de relaxamentos feitos para cada esquema de relaxamento como

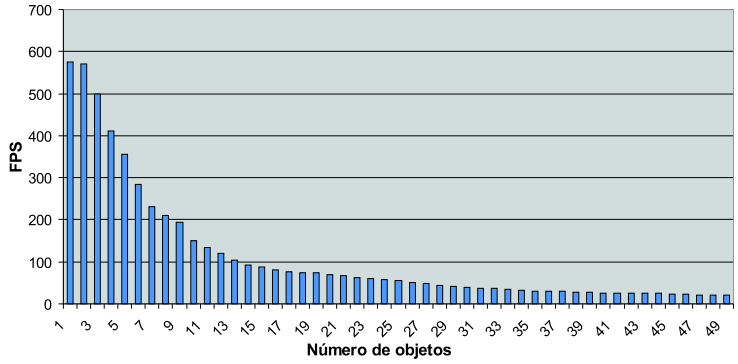

Figura 5.2: Taxa de quadros por segundo para a cena com icosaedros rígidos.

uma função do número de objetos na simulação. O resultado mostra que o esquema proposto requer na média aproximadamente a metade dos relaxamentos do processo de relaxamento não ordenado.

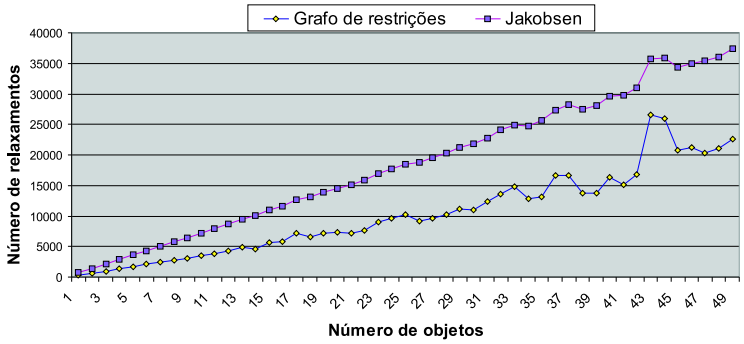

Figura 5.3: Contraste entre o método de relaxamento original e o método apresentado.

Outros dois experimentos que envolvem objetos deformáveis foram testados. No primeiro, deixa-se cair um retalho de tecido sobre um cubo e, no segundo experimento, o retalho de tecido é solto sobre uma corda que está pendurada por seus pontos terminais, como mostrado nas figuras 5.4 e 5.5. Nestes experimentos, a resolução do retalho de tecido foi variada desde uma malha grosseira de 20×20 até uma malha mais densa de 40×40. A corda pendurada tem 50 segmentos em todos os testes. O número de quadros por segundo é plotado como uma função da resolução do tecido na figura 5.6. Estes resultados mostraram que um retalho de tecido com uma resolução fina exige alto custo computacional, principalmente no mecanismo de detecção de colisão. Mesmo com um tecido modelado através de uma malha grosseira, o desempenho foi apenas próximo de tempo real (14 fps aproximadamente). Isto pode ser atribuído ao custo de execução dos testes de autointerseção usando a hierarquia de esferas limitantes, a qual não é muito

ajustada à geometria do objeto.

Um detalhe importante a ser mencionado é que a animação de objetos com alta resolução devem ser conduzidos com passos de tempo pequenos para não se correr o risco de que um objeto “atravesse” outro.

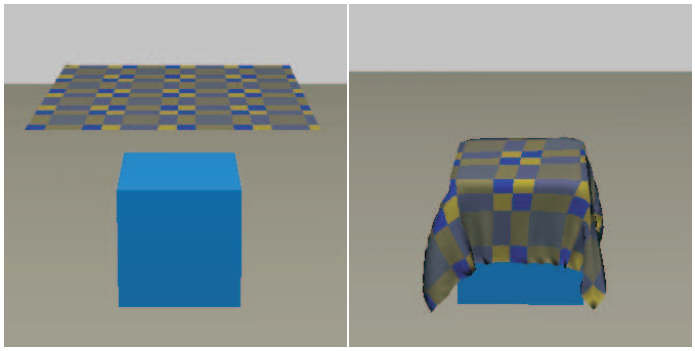

Figura 5.4: Um retalho de tecido caindo sobre um cubo rígido.

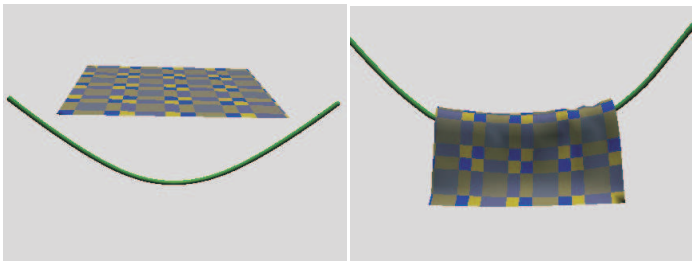

Figura 5.5: Um retalho de tecido sendo pendurado numa corda.

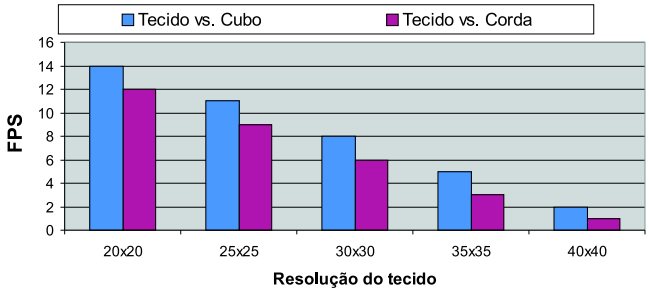

Figura 5.6: Taxa de quadros por segundo para a simulação de um retalho de tecido (objetos deformáveis).

Por outro lado, foi avaliada o desempenho do sistema proposto versus o desempenho da biblioteca ODE (*Open Dynamics Engine*), para tanto, foi testado um cenário onde se deixam cair 50 cilindros simultaneamente. Embora os cenários não sejam exatamente iguais, devido a que cada sistema usa os objetos geométricos de diferente forma. Para

avaliar de forma eqüitativa ambos os sistemas foram configurados para usar: cilindros com 48 faces e 26 vértices, no caso do sistema proposto os cilindros têm 60 restrições lineares; e um espaço de colisão simples, ou seja, são testados todos os objetos contra todos os objetos. As figuras 5.7 e 5.8 mostram os cenários criados.

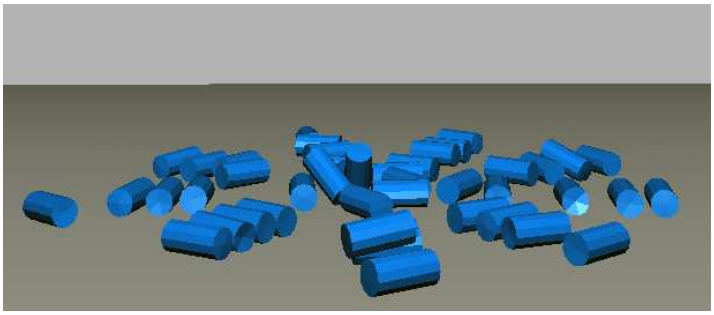

Figura 5.7: Cenário com 50 cilindros usando o sistema proposto.

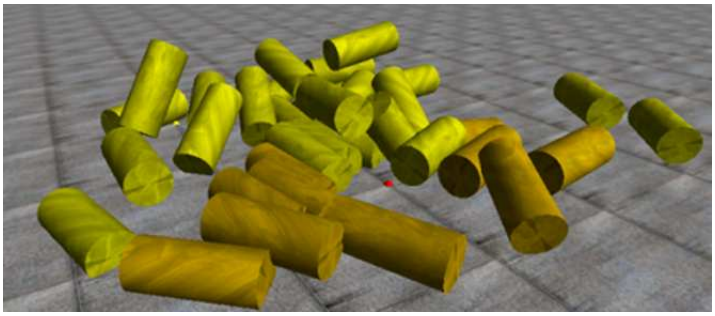

Figura 5.8: Cenário com 50 cilindros usando a biblioteca ODE.

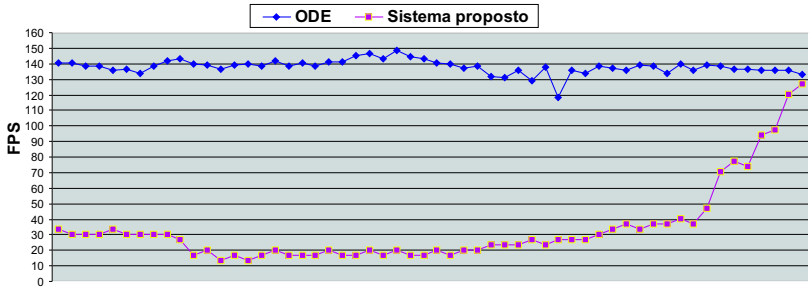

Figura 5.9: Taxa de quadros por segundo para a cena com cilindros rígidos.

O gráfico 5.9 mostra o desempenho de ambos os sistemas em quadros por segundo (FPS). Pode-se observar que a biblioteca ODE usando objetos primitivos alcança uma simulação bastante rápida. Entretanto, o sistema proposto tem um desempenho inferior ao primeiro quando há multiples contatos. Isto se deve principalmente a que o sistema proposto, por sua simplicidade, dispensa de importantes informações físicas que podem ser

úteis para melhorar o desempenho do sistema e pelo uso excessivo de restrições lineares. Porém, a biblioteca ODE [36] usa diversas melhorias para alcançar esses resultados, por exemplo, o uso de um número máximo de pontos de contato.

Adicionalmente, outros resultados obtidos são mostrados nas figuras 5.10, 5.11 e 5.12.

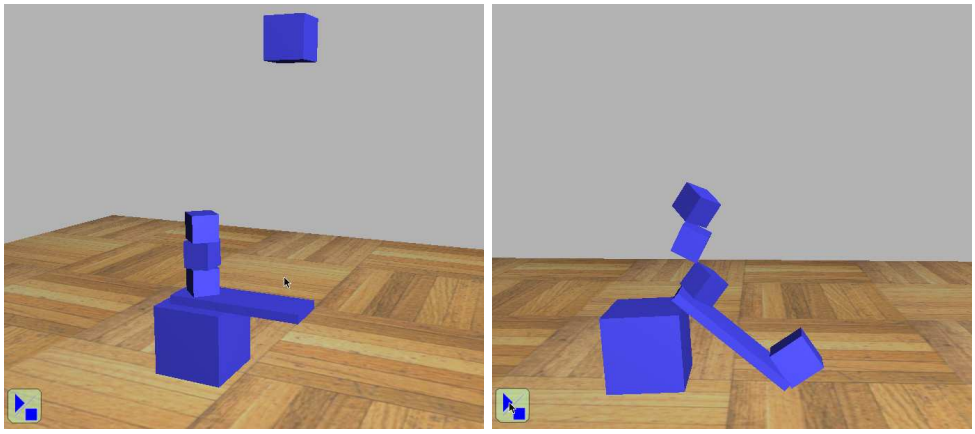

Figura 5.10: Cubos de diferentes tamanhos estão empilhados.

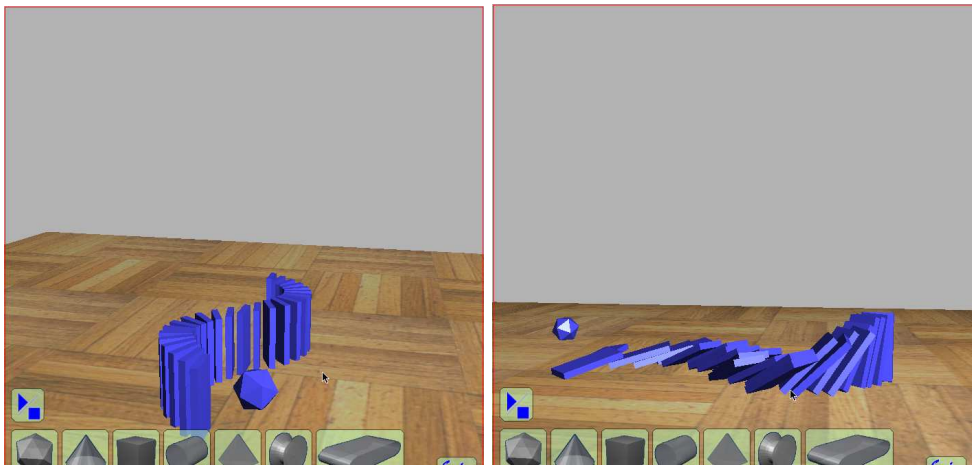

Figura 5.11: Múltiplos objetos colidindo ao mesmo tempo.

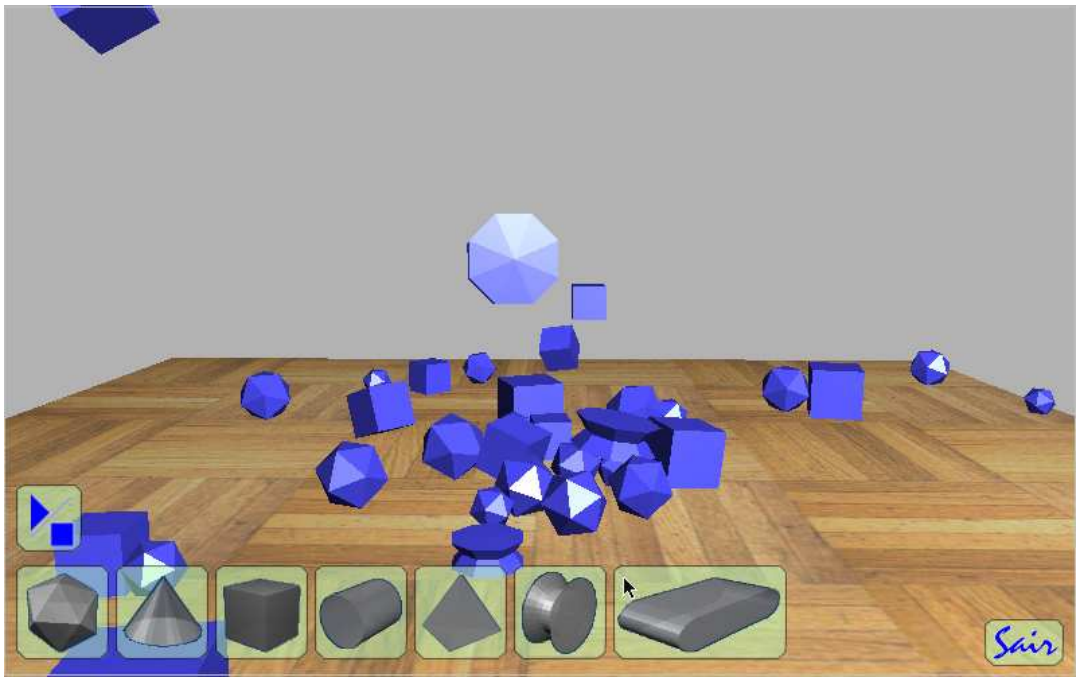

Figura 5.12: Cena com objetos de diferente geometria e tamanho.

Capítulo 6

Conclusões e trabalhos futuros

Foi apresentado um protótipo de sistema de animação baseada em física, o qual usa a estrutura de animação física simplificada originalmente proposta por Jakobsen [24]. A estrutura original foi estendida usando-se um mecanismo de detecção de colisões real e um esquema de resposta a colisões baseado em projeções.

O mecanismo de detecção de colisões e o esquema de resposta a colisões permitem identificar e resolver colisões entre corpos rígidos de geometria convexa e tambem objetos deformáveis.

O método de integração de Verlet pode ser substituído por uma versão que inclui fricção [15, 2], o qual é usado no protótipo de sistema descrito para alcançar uma animação semelhante à obtida com as metodologias tradicionais.

O sistema, apesar de não ser matematicamente exato, produz respostas a colisões de maneira convincente e próxima às metodologias tradicionais, conseguindo tratar casos complexos como objetos empilhados onde há multiplos contactos.

Para trabalhos futuros, planejamos diversas extensões ao protótipo de sistema apresentado. Como um primeiro passo, a física rotacional deverá ser incorporada usando um esquema similar ao de Baltman [2]. Depois, as hierarquias de esferas deverão ser empregadas de maneira mais eficiente, provavelmente adotando algumas técnicas reportadas recentemente [25]. Finalmente o código precisa ser otimizado para alcançar um melhor desempenho.

Referências Bibliográficas

[1] Pankaj Agarwal, Leonidas Guibas, An Nguyen, Daniel Russel, and Li Zhang. Collision detection for deforming necklaces. In *Comput. Geom. Theory Appl.*, volume 28, pages 137–163, Amsterdam, The Netherlands, The Netherlands, 2004. Elsevier Science Publishers B. V.

[2] R. Baltman. Verlet integration and constraints in a six degree of freedom rigid body physics simulation. *Game Developers Conference*, 2004.

[3] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 23–34, New York, NY, USA, 1994. ACM Press.

[4] David Baraff. Interactive simulation of solid rigid bodies. volume 15, pages 63–75, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[5] David Baraff. Linear-time dynamics using Lagrange multipliers. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 137–146, New York, NY, USA, 1996. ACM Press.

[6] B. G. Baumgart. A polyhedron representation for computer vision. In *Proceedings of AFIPS National Computer Conference*, volume 4, pages 589–596, 1975.

[7] Gareth Bradshaw and Carol O'Sullivan. Adaptive medial-axis approximation for sphere-tree construction. *ACM Trans. Graph.*, 23(1):1–26, 2004.

[8] Rikk Carey and Gavin Bell. *The annotated VRML 2.0 reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1997.

[9] CGAL. The computational geometry algorithms library. http://www.cgal.org.

[10] K. Chung and W. Wang. Quick collision detection of polytopes in virtual environments. In *ACM Symposium on Virtual Reality Software and Technology*, July 1996.

[11] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. I-COLLIDE: an interactive and exact collision detection system for large-scale environments. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff., New York, NY, USA, 1995. ACM Press.

[12] Corporate OpenGL Architecture ReviewBoard. *OpenGL reference manual: the official reference document for OpenGL, release 1*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[13] David P. Dobkin and David G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. Technical report, Vancouver, BC, Canada, Canada, 1983.

[14] David P. Dobkin and David G. Kirkpatrick. Determining the separation of preprocessed polyhedra: a unified approach. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 400–413, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[15] D. Eberly. *Game Physics*. MK-Morgan Kaufman, San Francisco, CA 94111, 2004.

[16] Alejandro Garcia-Alonso, Nicolás Serrano, and Juan Flaquer. Solving the collision detection problem. *IEEE Comput. Graph. Appl.*, 14(3):36–43, 1994.

[17] Bernd Gartner. Fast and robust smallest enclosing balls. In *Proceedings of the 7th Annual European Symposium on Algorithms*, pages 325–338, 199.

[18] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three dimensional spaces. *IEEE Journal of Robotics and Automation*, April 1988.

[19] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180, New York, NY, USA, 1996. ACM Press.

[20] James K. Hahn. Realistic animation of rigid bodies. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 299–308, New York, NY, USA, 1988. ACM Press.

[21] C. Hecker. Rigid body dynamics, 1998. http://www.d6.com/users/checker/dynamics.htm.

[22] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[23] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph.*, 15(3):179–210, 1996.

[24] T. Jakobsen. Advanced character physics. In *Proceedings of GDCONF'2001 Game Developer's Conference 2001*, 2001.

[25] Doug L. James and Dinesh K. Pai. Bd-tree: output-sensitive collision detection for reduced deformable models. In *ACM Trans. Graph.*, volume 23, pages 393–398, New York, NY, USA, 2004. ACM Press.

[26] P. Jimenez, F. Thomas, and C. Torras. 3d collision detection: A survey. *Computer and Graphics*, 25(2):269–285, April 2001.

[27] Lutz Kettner. Designing a data structure for polyhedral surfaces. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, pages 146–154, New York, NY, USA, 1998. ACM Press.

[28] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[29] Ming C. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *IMA Conference on Mathematics of Surfaces*, pages 37–56, 1998.

[30] M. Mantyla. *Introduction to Solid Modeling*. W. H. Freeman & Co., New York, NY, USA, 1988.

[31] Brian Vincent Mirtich. *Impulse-based dynamic simulation of rigid body systems*. PhD thesis, University of California at Berkeley, 1996.

[32] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 289–298, New York, NY, USA, 1988. ACM Press.

[33] Madhav Ponamgi, Dinesh Manocha, and Ming C. Lin. Incremental algorithms for collision detection between solid models. In *SMA '95: Proceedings of the third ACM symposium on Solid modeling and applications*, pages 293–304, New York, NY, USA, 1995. ACM Press.

[34] S. Quinlan. Efficient distance computation between non-convex objects. *IEEE Intern. Conf. on Robotics and Automation*, pages 3324–3329, 1994.

[35] Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[36] Russell Smith. Open dynamics engine, 2004. http://ode.org/.

[37] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 153–162, New York, NY, USA, 1987. ACM Press.

[38] Gino van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *J. Graph. Tools*, 2(4):1–13, 1997.

[39] Gino van den Bergen. A fast and robust GJK implementation for collision detection of convex objects. *Journal of Graphics Tools*, 4(2):7–25, 1999.

[40] Gino van den Bergen. *Collision Detection in Interactive 3D environments*. MK-Morgan Kaufman, San Francisco, CA 94111, 2004.

[41] A. Verlet. Computer experiments on classical fluids: I. thermodymamic properties of leonard-jones molecules. *Phys. Review*, 159:98–103, 1967.

[42] E. Welzl. Smallest enclosing disks(ball and ellipsoids). In *New Results and New Trends in Computer Science, Volume 555 of Lecture Notes Comput. Sci.*, pages 359–370. Springer-Verlag, 1991.

[43] Andrew Witkin and David Baraff. Physically based modeling: Principles and practice, 1997. http://www.cs.cmu.edu/ baraff/sigcourse/.