UM SISTEMA PARA ANIMAÇÃO FÍSICA USANDO MALHAS TETRAEDRAIS

Guina Guadalupe Sotomayor Alzamora

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. Antonio Alberto Fernandes de Oliveira, DSc.

Prof. Luiz Henrique de Figueiredo, DSc.

RIO DE JANEIRO, RJ - BRASIL
ABRIL DE 2008

SOTOMAYOR ALZAMORA, GUINA GUADALUPE

Um sistema para animação física usando malhas tetraedrais [Rio de Janeiro] 2008

XIV, 76 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2008)

Dissertacção – Universidade Federal do Rio de Janeiro, COPPE

1. Introdução. 2. Animação baseada em física.
3. Detecção de colisões. 4. Resposta às colisões.
5. Sistema proposto.

I. COPPE/UFRJ II. Título (série)

A Pedro, Domitila,
Percy, Karina, Yemily e
Yalmar
minha pequena grande familia

Agradecimentos

Gostaria de agradecer a todos que contribuíram para a conclusão deste trabalho.

Aos meus pais, Pedro e Domitila, e meus irmãos Percy, Karina e Yemily, pelo seu apoio incondicional, seu carinho, por me fazerem sentir perto de vocês sempre, e por toda a ajuda e importante presença na minha vida.

Aos professores Antônio Oliveira, Paulo Roma, e principalmente ao meu orientador Claudio Esperança pela paciência e o apoio sempre presente.

Ao meu co-orientador não oficial Yalmar, pela grande ajuda no decorrer deste trabalho e pelos conselhos no mestrado e na vida.

A meus amigos(as)) do LCG Saulo, Luis, Ricardo, Alvaro, André, Disney, Vitor, Djeisson, Elisabete, Diego, Flávio, Pilato, Alberto, Felipe e Okamoto. Pelas pequenas ou grandes conversas académicas e pessoais e pelos momentos de lazer.

Aos meus amigos(as) Liliana, Gladys, Raquel, Gary, Katherine, Sholy, Luly, Helard, Alessandra, Wily, Courtney, Rafael, Seimou, Marcelo, Esther, Juan Carlos, e Edgar. Pela força, a amizade e o incentivo no decorrer dos últimos anos.

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

UM SISTEMA PARA ANIMAÇÃO FÍSICA USANDO MALHAS TETRAEDRAIS

Guina Guadalupe Sotomayor Alzamora

Abril/2008

Orientador: Claudio Esperança

Programa: Engenharia de Sistemas e Computação

Apresenta-se uma abordagem simplificada para animação de objetos deformáveis geometricamente complexos, representados como malhas tetraedrais. O sistema detecta e responde a colisões de objetos sujeitos a deformações elásticas de rigidez variável. A abordagem combina várias técnicas, como a detecção de colisões usando *Hashing* espacial, resposta às colisões através do cômputo da superfície de contato, que usa o cálculo da profundidade de penetração por propagação, a estimativa dos vetores de deslocamento dos vértices da região de deformação e busca binária para separar os objetos. A dinâmica está baseada no casamento de formas e na análise modal, na integração do sistema é usado um esquema de Euler explícito-implícito. Resultados preliminares mostram a interação entre objetos constituídos de várias centenas de tetraedros em tempo real.

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

A SYSTEM FOR PHYSICAL ANIMATION USING TETRAHEDRAL MESHES

Guina Guadalupe Sotomayor Alzamora

April/2008

Advisor: Claudio Esperança

Department: Computing and Systems Engineering

We present a simplified approach for animation of geometrically complex deformable objects represented as tetrahedral meshes. Our prototype system detects and responds to collisions of objects subject to elastic deformations of variable stiffness. The proposed approach combines several techniques, namely, collision detection using a Spatial Hashing, collision response through a contact surface that use a consistent penetration depth using propagation, an estimate for displacement vector of the deformation region and binary search to separate objects. The dynamics is based on shape matching and a modal analysis scheme, using an Euler explicit-implicit integrator. Preliminary results show that collisions between objects containing several hundreds tetrahedra can be animated in real-time.

Sumário

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

A animação física tem sido amplamente pesquisada nos últimos anos, encontrando aplicação na indústria de jogos e cinema, na área médica em simulações cirúrgicas e protótipos virtuais, na educação e outras áreas [29]. A animação pode ser criada de diversas maneiras dependendo do propósito da aplicação, equilibrando desempenho e precisão.

Em se tratando de animação física, enquanto a animação de corpos rígidos tem sido investigada intensamente durante as últimas duas décadas, recentemente o foco das pesquisas sobre animação tem recaído sobre objetos deformáveis. Tais objetos têm normalmente representação complexa quando comparados com corpos rígidos, já que podem mudar de forma no tempo pela interação com ele mesmo ou com outros agentes no cenário, o que introduz um fator complicador considerável. Pesquisas recentes nessa área vêm adaptando diversas técnicas para animação de corpos rígidos. Entre elas se destaca a *deformação sem uso de malhas* [27, 26, 31], onde os modelos basicamente são representados por pontos, sem nenhuma informação adicional de conectividade. Este tipo de modelagem é simples e eficiente já que não precisa de estruturas de dados complexas para representar o modelo.

Uma vez escolhida a forma de representação, um sistema para animação física tipicamente emprega algoritmos de detecção de colisões, acompanhados de esquemas de resposta às colisões. Portanto, é comum se recorrer às chamadas *estruturas de dados espaciais*, como as hierarquias de volumes limitantes e as estruturas de subdivisão espacial além de estruturas próprias para representação de campos de distância, *Soma de Minkowski*, entre outras. De todas estas técnicas, uma simples de ser implementada é o *Hashing* espacial, onde o espaço é dividido por uma grade uniforme e suas células são mapeadas numa tabela de dispersão (*hash*).

Outro aspecto importante é o inerente baixo desempenho de sistemas que buscam ob-

ter simulações fisicamente corretas. Por outro lado, aplicações que não requerem correção física podem se beneficiar de técnicas aproximativas capazes de obter animações conduzidas eficientemente com um comportamento físico plausível.

Este trabalho enfoca o uso de uma metodologia simples para animação de objetos deformáveis, que são representados por malhas tetraedrais. O método apresentado combina várias técnicas empregadas para a modelagem de objetos deformáveis, a saber:

- detecção de colisão usando subdivisão espacial [45] e volumes limitantes;
- resposta às colisões através do cômputo da superfície de contato, que utiliza o cálculo da profundidade de penetração por propagação [18], a estimativa dos vetores de deslocamento [18] e a resolução das colisões assimétricas [19] dos vértices da região de deformação, e busca binária para separar os objetos [39];
- a animação é feita usando uma técnica de casamento de formas e um integrador de Euler explícito-implícito [26].

O restante deste trabalho é dividido da seguinte forma: o capítulo dois faz um apanhado geral das abordagens para animação de objetos deformáveis; o três aborda métodos e técnicas para detecção de colisões; o quatro aborda métodos de resposta às colisões; o cinco descreve em detalhe a implementação do protótipo desenvolvido; o seis mostra os resultados obtidos. E, finalmente, o sete apresenta as conclusões e trabalhos futuros.

Capítulo 2

Animação baseada em física

O que se entende pelo termo “animação baseada em física” (*physically based animation*, em inglês) é um processo computacional que visa obter animação de objetos com plausibilidade física. Isto contrasta com o termo “animação física”, empregado usualmente para designar animações que visam replicar processos físicos com alto grau de acurácia, embora por vezes os dois termos sejam usados indistintamente. Este trabalho insere-se mais no contexto do primeiro termo, uma vez que trata de abordagens onde a preocupação com desempenho leva a um tratamento simplificado de determinadas interações entre objetos.

Recentemente, avanços nesta área propõem o uso de abordagens sem malhas, onde os objetos são representados por um conjunto de pontos (chamados partículas, neste contexto), sem conectividade, em vez de polígonos ou malhas volumétricas. Esses métodos garantem simulações estáveis e eficazes, evitando problemas relacionados com a qualidade de elementos da malha, que são desvantagens típicas de abordagens baseadas em malhas. Entretanto, o fato de que as superfícies dos objetos não são definidas explicitamente levam a outros problemas, por exemplo, na verificação de intersecção ou sobreposição de objetos.

Embora os modelos baseados em física não pretendam reproduzir a realidade, eles tentam produzir movimentos com base nos mesmos princípios físicos [30]. A parte da física que estuda o movimento dos corpos é a dinâmica, que é baseada nas *Leis de Movimento de Newton*, a saber:

Primeira Lei de Newton: na ausência de forças externas, um corpo em repouso permanece em repouso e, se este está em movimento, permanece em movimento com velocidade constante. Isto é, só forças externas podem mudar o movimento de um corpo;

Segunda Lei de Newton: para um corpo de massa constante m sofrendo uma força F, o

movimento do corpo sobre o tempo é dado por:

$$F = ma = m\dot{v} = m\ddot{x}, \tag{2.1}$$

onde a é a aceleração do corpo, que pode ser representada como $\dot{v}$ que é a primeira derivada da velocidade do corpo, ou como $\ddot{x}$ que é a segunda derivada da posição do corpo; e

Terceira Lei de Newton: ao aplicar uma força externa sobre um corpo, há uma força de igual magnitude na mesma direção, mas em sentido contrário, exercida sobre o causador da força. Isto é, a toda ação corresponde uma reação.

A animação de objetos rígidos e deformáveis frequentemente se baseia em sistemas de partículas, sendo que a animação de objetos deformáveis deve adotar algum modelo de deformação físico, que permita deformações elásticas ou plásticas. Tais deformações podem ser obtidas usando sistemas massa-mola, elementos finitos ou métodos sem malhas. Na dinâmica de objetos deformáveis podemos distinguir dois métodos clássicos:

Métodos Lagrangeanos: o modelo consiste de um conjunto de partículas, com posições e propriedades variantes, cujas equações derivam da cinemática. Este método é o mais usado na animação de objetos deformáveis. (Na seção 2.4 se verá mais detalhes do referido método);

Métodos Eulerianos: as propriedades do modelo são computadas para um conjunto estático de pontos. Este método é mais usado para simulação de líquidos.

Segundo Nealen et al. [29], na animação de objetos deformáveis baseados em física, os *Métodos Lagrangeanos* podem ser divididos em duas categorias:

- Métodos baseados em malhas.
  - Sistemas massa mola.
  - Método de elementos finitos (FEM).
- Métodos sem malhas.
  - Animação baseada em pontos.
  - Deformação baseada em casamento de formas.

2.1 Sistemas de partículas

Em animação física, partículas são abstrações para objetos sem forma ou dimensões, mas dotados de propriedades físicas como massa, posição, velocidade, podendo ser submetidas a forças externas. Por serem simples, são objetos de fácil manipulação. 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:

$$X(t) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}. \tag{2.2}$$

Por outro lado, um sistema de partículas é um conjunto de partículas que juntas representam algum tipo de **objeto**: rígido, deformável ou líquido [26]. Numa implementação de um sistema de partículas, as partículas interagem principalmente com forças externas e são usadas para calcular a orientação do objeto. Este cálculo envolve a resolução de um sistema de equações diferenciais ordinárias (EDO) [4].

Como no caso de uma partícula, o estado de um sistema de partículas pode ser representado por um vetor de componentes $x_i(t)$ e $v_i(t)$. Assim, podemos estender o vetor $X(t)$ da seguinte forma:

$$X(t) = \begin{pmatrix} x_1(t) \\ v_1(t) \\ \vdots \\ x_n(t) \\ v_n(t) \end{pmatrix}, \tag{2.3}$$

onde $x_i(t)$ e $v_i(t)$ representam a posição e velocidade da i-ésima partícula.

2.2 Dinâmica de partículas

Na dinâmica de partículas, o movimento de uma partícula é determinado pela influência de forças externas (como a gravidade, o vento, as forças de mola, etc.) em um instante de tempo t [26]. Seja $F(t)$ a força resultante que age na partícula no tempo t: se a partícula tem massa m, então a variação de $X(t)$ (equação 2.2) ao longo do tempo é dada por:

$$\frac{d}{dt}X(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \frac{F(t)}{m} \end{pmatrix}. \tag{2.4}$$

2.3 Objetos rígidos

Objetos rígidos ou corpos rígidos podem ser representados por sistemas de partículas. Eles ocupam um lugar no espaço e possuem basicamente uma posição x, uma orientação

R, uma massa m, uma velocidade linear v, uma velocidade angular ω e um tensor de inércia I.

Na dinâmica, o movimento dos corpos rígidos é originado pela reação a forças. A equação de movimento é similar à usada para simular o movimento de uma partícula (equação 2.4), onde é necessário determinar $\frac{d}{dt}X(t)$.

2.3.1 Posição e orientação

A posição de uma partícula no espaço no instante t pode ser representada por um vetor $x(t)$, que descreve o deslocamento em relação à origem. De modo similar, a posição de um corpo no instante t é representado por um vetor $x(t)$, que descreve o deslocamento do corpo em relação à origem.

Adicionalmente, um corpo rígido pode sofrer rotações. Uma rotação é representada, típicamente, por uma matriz 3×3 $R(t)$. Assim, $x(t)$ e $R(t)$ são as variáveis que descrevem o estado do corpo rígido.

Como corpos rígidos só podem sofrer translações e rotações, a forma do objeto é definida em termos de um espaço fixo e imutável chamado *espaço do corpo*. Para transformar este espaço usa-se $x(t)$ e $R(t)$, por exemplo, para encontrar a posição $p(t)$ de um ponto cujas coordenadas no espaço do corpo é p_0 basta aplicar a rotação seguida da translação, seguindo a fórmula:

$$p(t) = R(t)p_0 + X(t), \tag{2.5}$$

onde $R(t)$ especifica a rotação do corpo em torno a seu centro de massa e $x(t)$ é a localização do centro de massa no espaço do mundo no instante t.

2.3.2 Velocidade linear

A velocidade linear descreve a mudança da posição do centro de massa do corpo no tempo t, ou seja, depende da massa do objeto. Dessa forma, se $x(t)$ é a *posição* do centro de massa do corpo no tempo t, então sua velocidade linear $v(t)$ no espaço do mundo é definida como:

$$v(t) = \dot{x}(t) = \frac{d}{dt}x(t). \tag{2.6}$$

2.3.3 Velocidade angular

Um corpo rígido pode girar ao redor de um eixo. Se a posição do centro de massa está fixa no espaço do mundo, 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 $\omega(t)$ e a direção coincide com a direção do eixo de rotação do corpo. (Figura 2.1).

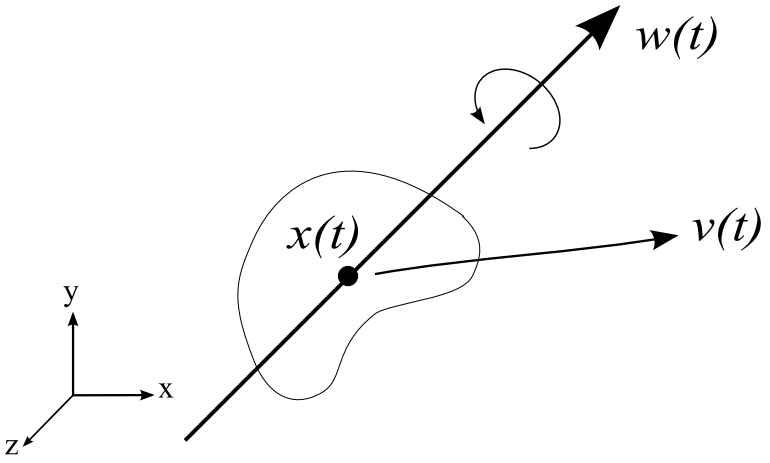


Figura 2.1: velocidade linear $v(t)$ e velocidade angular $\omega(t)$ de um corpo rígido.

A velocidade linear se relaciona com a posição do corpo através da equação 2.6. Analogamente, a rotação $R(t)$ está relacionada com a velocidade angular $\omega(t)$. No entanto, $\dot{R}(t)$ não é $\omega(t)$, já que $R(t)$ é uma matriz e $\omega(t)$ é um vetor, mas este vetor pode ser definido como matriz usando sua matriz anti-simétrica $\omega(t)^*$. Assim, a relação entre $\dot{R}(t)$ e $\omega(t)$ é estabelecida como:

$$\dot{R}(t) = \omega(t)^* \cdot R(t). \tag{2.7}$$

2.3.4 Massa

A massa de um corpo rígido pode ser computada como a soma das massas de suas partículas segundo a fórmula:

$$M = \sum_{i=1}^{N} m_i, \tag{2.8}$$

onde M é a massa total do corpo, N o número total de partículas do corpo e m_i é a massa da i-ésima partícula.

2.3.5 Centro de massa

Freqüentemente, na implementação de sistemas físicos, os corpos podem ser tratados como se sua massa se concentrasse em um simples ponto. Este ponto é conhecido como centro de massa. Por exemplo, suponhamos que um corpo é composto de duas partículas

nas posições x_1 e x_2 e com massas m_1 e m_2, respectivamente. Então, seu centro de massa pode ser encontrado pela relação:

$$m_1 g(x_1 - x_{cm}) + m_2 g(x_2 - x_{cm}) = g[(m_1 x_1 + m_2 x_2) - (m_1 + m_2)x_{cm}] = 0.$$

Reorganizando, obtemos que o centro de massa do objeto é:

$$x_{cm} = \frac{m_1}{m_1 + m_2} x_1 + \frac{m_2}{m_1 + m_2} x_2. \tag{2.9}$$

A Figura 2.2 mostra situações com corpos formados por partículas de igual e diferente tamanho.

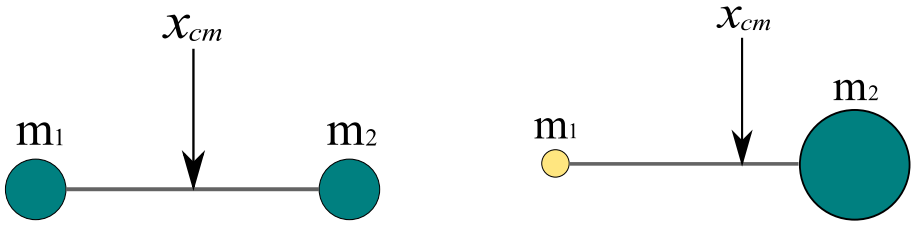


Figura 2.2: centro de massa de um corpo com partículas de massas (a) iguais e (b) diferentes.

Ao estender este raciocínio, conclui-se que o centro de massa de um corpo pode ser computado usando a equação:

$$x_{cm} = \frac{\sum_i m_i x_i}{\sum_i m_i}, \tag{2.10}$$

onde m_i e x_i são a massa e a posição da i-ésima partícula.

2.3.6 Momento linear

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

$$P(t) = \sum m_i \dot{r}_i(t), \tag{2.11}$$

onde $\dot{r}_i(t)$ é a velocidade da i-ésima partícula. O momento linear resultante do corpo é definido por:

$$P(t) = \sum_i m_i \dot{r}_i(t) = Mv(t). \tag{2.12}$$

Se o corpo é uma única partícula de massa M e velocidade $v(t)$, derivando $P(t)$ obtemos a seguinte relação importante:

$$\dot{v}(t) = \frac{\dot{P}(t)}{M}. \tag{2.13}$$

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

2.3.7 Momento angular

Apesar do significado de momento linear ser bastante intuitivo, o conceito análogo de momento angular para um corpo rígido é um pouco menos direto. Para o momento linear, temos $P(t) = Mv(t)$. Analogamente, o momento angular resultante $L(t)$ de um corpo rígido é dado por $L(t) = I(t)\omega(t)$, onde $I(t)$, o chamado *tensor de inércia*, é uma matriz 3×3. Este último 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, enquanto que no momento linear é um escalar. Note 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 $\dot{L}(t) = \tau(t)$, de forma análoga à relação entre $P(t)$ e $F(t)$.

2.3.8 Tensor de inércia

O tensor de inércia $I(t)$ é o fator de escala entre o momento angular $L(t)$ e a velocidade angular $\omega(t)$. Este descreve a distribuição da massa de um corpo e depende de sua forma.

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:

$$I(t) = \sum \begin{pmatrix} m_i(r_{iy}^{'2} + r_{iz}^{'2}) & -m_i r_{ix}^{'} r_{iy}^{'} & -m_i r_{ix}^{'} r_{iz}^{'} \\ -m_i r_{iy}^{'} r_{ix}^{'} & m_i(r_{ix}^{'2} + r_{iz}^{'2}) & -m_i r_{iy}^{'} r_{iz}^{'} \\ -m_i r_{iz}^{'} r_{ix}^{'} & -m_i r_{iz}^{'} r_{iy}^{'} & m_i(r_{iy}^{'2} + r_{iz}^{'2}) \end{pmatrix}. \tag{2.14}$$

A 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:

$$I(t) = \sum m_i r_i^{'T} r_i^{'} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} m_i r_{ix}^{'2} & m_i r_{ix}^{'} r_{iy}^{'} & m_i r_{ix}^{'} r_{iz}^{'} \\ m_i r_{iy}^{'} r_{ix}^{'} & m_i r_{iy}^{'2} & m_i r_{ix}^{'} r_{iz}^{'} \\ m_i r_{iz}^{'} r_{ix}^{'} & m_i r_{iz}^{'} r_{iy}^{'} & m_i r_{iz}^{'2} \end{pmatrix}. \tag{2.15}$$

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

$$I(t) = \sum m_i(r_i'^T r_i')\mathbf{1} - r_i' r_i'^T, \tag{2.16}$$

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:

$$\begin{aligned} I(t) &= \sum m_i(r_i'^T r_i')\mathbf{1} - r_i' r_i'^T \\ &= \sum m_i((R(t)r_{0i})^T(R(t)r_{0i})\mathbf{1} - (R(t)r_{0i})(R(t)r_{0i})^T) \\ &= \sum m_i((r_{0i}^T r_{0i})\mathbf{1} - R(t)r_{0i}r_{0i}^T R(t)^T) \\ &= \sum m_i(R(t)(r_{0i}^T r_{0i})R(t)^T\mathbf{1} - R(t)r_{0i}r_{0i}^T R(t)^T) \\ &= R(t)(\sum m_i((r_{0i}^T r_{0i})\mathbf{1} - r_{0i}r_{0i}^T)R(t)^T. \end{aligned} \tag{2.17}$$

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:

$$I(t) = R(t)I_b R(t)^T, \tag{2.18}$$

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

2.4 Objetos deformáveis

A simulação de objetos deformáveis foi introduzida por Terzopoulos et al. [42], empregando a teoria de elasticidade para construir equações diferenciais que modelam o comportamento de curvas, superfícies e sólidos não rígidos em função do tempo. Assim, os modelos elasticamente deformáveis respondem de forma natural às forças aplicadas, a restrições e a obstáculos impenetráveis. A equação que governa o movimento de modelos deformáveis pode ser escrita na formulação de *Lagrange* como se segue:

$$\frac{\partial}{\partial t}(\mu\frac{\partial r}{\partial t}) + \gamma\frac{\partial r}{\partial t} + \frac{\delta\varepsilon(r)}{\delta r} = f(r,t), \tag{2.19}$$

onde:

- $r(a,t)$ é a posição da partícula em um tempo t;
- $\mu(a)$ é a densidade de massa do objeto em a;
- $\gamma(a)$ é a densidade de amortecimento;
- $\varepsilon(r)$ é um funcional que mede a energia potencial instantânea da deformação elástica do corpo.

- $f(r,t)$ representa as forças externas aplicadas no seguinte intervalo de tempo;

Na equação 2.19, devido ao modelo deformável, as forças externas são equilibradas com as forças internas do objeto deformável. O primeiro termo é a força interna por conta da massa distribuída nos objetos; o segundo é a força de amortecimento devido à dissipação; o terceiro é a força elástica devido à deformação do objeto. Desta forma, a energia potencial de deformação para objetos elásticos pode ser usada como uma medida de deformação. Portanto, aplicando forças externas ao modelo elástico, pode-se conseguir uma dinâmica realista.

Para resolver a equação 2.19, o modelo dinâmico contínuo é discretizado como:

$$M\ddot{p} + D\dot{p} + Kp = f, \tag{2.20}$$

onde p é um vetor posição da amostragem de pontos da malha, M é a matriz de massa, D é a matriz de amortecimento, K é a matriz de rigidez, e f é a soma das possíveis forças externas de todos os pontos da malha, isto é $f = \sum f_{ext}$.

Assim, o modelo deformável pode ser resolvido por abordagens numéricas tais como métodos de elementos finitos ou métodos sem malhas (*meshless*, em inglês).

A seguir são destritos alguns métodos para animação de objetos deformáveis.

2.4.1 Métodos baseados em malhas

2.4.1.1 Elasticidade contínua

Nestos métodos, um objeto deformável é definido por sua geometria na forma não deformada (forma inicial) e por um conjunto de parâmetros materiais, que definem como o objeto se deforma quando forças são aplicadas.

Se aplicarmos uma deformação ao objeto, cada ponto originalmente localizado na posição p será deslocado para uma nova posição $x(p)$. Por outro lado, o campo vetorial de deslocamento de cada ponto $u(p) = x(p) - p$ define o campo de deformação do objeto.

Por último, é computado o tensor de tensão *strain tensor* em inglês $\varepsilon \in R^{3\times3}$, que mede o deslocamento por unidade de área. Este cálculo é feito em função da variação do campo de deslocamento $u(p)$. O método mais usado é conhecido como tensor de *strain* de *Green* [29]:

$$\varepsilon = \frac{1}{2}(\nabla u + [\nabla u]^T + [\nabla u]^T \nabla u), \tag{2.21}$$

onde $\nabla u \in R^{3\times3}$ é a matriz do campo de deslocamento no espaço 3D.

O tensor de estresse (*stress tensor* em inglês) $\sigma \in R^{3\times3}$ fornece a força por unidade de área em relação à tensão. Este tensor pode ser avaliado usando a *Lei Material Linear de Hooke*:

$$\sigma = E \times \varepsilon, \tag{2.22}$$

onde $E \in R^{3\times3}$ depende das características elásticas materiais e determina a rigidez do objeto simulado.

2.4.1.2 Modelo de deformação reduzida

Trata-se de uma formulação descrita por James e Pai [20] onde se assume que a deformação a ser simulada não altera drasticamente a forma do modelo. Dados N pontos não deformados $p = [p_1, ..., p_N]^T$, um modelo de deformação reduzida aproxima pontos deformados $x(p)$, por meio de uma superposição linear de M campos de deslocamento, dadas pelas colunas de U da equação 2.23. A amplitude de cada campo de deslocamento é dada pelas coordenadas reduzidas correspondentes q (Figura 2.3), de forma que:

$$x(p) = p + Uq \quad ou \quad x(p) = p_i + \sum_{j=1}^{M} U_{ij} q_j, \tag{2.23}$$

onde as coordenadas reduzidas q são determinadas por algum processo que pode ser acoplado ao modelo. As colunas de U representam campos de deslocamento da elasticidade contínua, que podem ser obtidos através de processos de análise modal, interpolação ou modelagem multi-resolução.

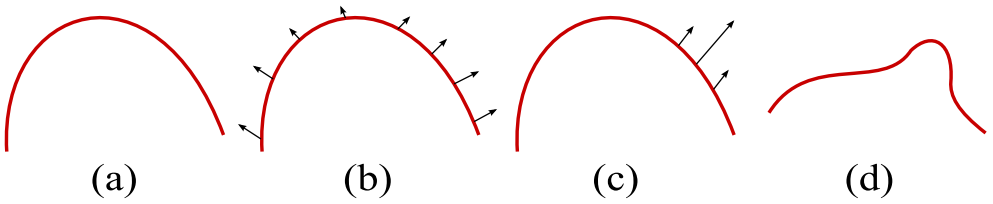


Figura 2.3: modelos de deformação reduzida: (a) forma de referência p, (b) campo de deslocamento U_1, (c) campo de deslocamento U_2 e (d) uma forma de deformação possível $p' = p + U_1 + 0.5U_2$.

2.4.1.3 Análise modal linear

A análise modal desacopla a equação de movimento de um sistema, com o objetivo de obter um sistema de equações independentes, cujas equações iniciais são chamadas de equações modais. Após ter encontrado a solução para cada equação do sistema, elas são acopladas novamente.

Pentland e Williams desenvolveram uma expressão simplificada da dinâmica de objetos deformáveis usando a análise modal [32]. Assim, para resolver este problema generalizado de auto-valor, as matrizes de massa M, amortecimento D e rigidez K da equação 2.20, podem ser desacopladas em equações diferenciais ordinárias independentes (EDOs),

$$M\Phi\Lambda = K\Phi, \tag{2.24}$$

onde $\Phi^T M\Phi = I \quad e \quad \Phi^T K\Phi = \Lambda$ são matrizes diagonais.

As colunas de $\Phi = [\Phi_1, \Phi_2, ...\Phi_N]$ contêm auto-vetores de $M^{-1}K$ e as diagonais de Λ são seus auto-valores. Φ é o termo *matriz modal* ou *matriz modal de deslocamento* e as colunas de Φ formam uma base (a base modal ou *auto*-base) do espaço $3n$-dimensional. Assim, qualquer deslocamento $u(t) = x(t) - x_0$ pode ser re-escrito como uma combinação linear das colunas:

$$u(t) = \Phi q(t), \tag{2.25}$$

onde o vetor $q(t)$ contém a coordenada modal (ou amplitude modal). Substituindo o vetor $u(t)$ da equação anterior em 2.20 e multiplicando-a por Φ^T obtém-se:

$$\Phi^T M\Phi\ddot{q} + \Phi^T D\Phi\dot{q} + \Phi^T K\Phi q = \Phi^T f_{ext}. \tag{2.26}$$

Desde que M e K sejam normalmente simétricas e positivas definidas, é possível diagonalizá-las simultaneamente por Φ:

$$\Phi^T M\Phi = \bar{M} \quad e \quad \Phi^T K\Phi q = \bar{K}, \tag{2.27}$$

onde $\bar{M}$ e $\bar{K}$ são diagonais, D é uma combinação linear delas e também pode ser diagonalizada por Φ:

$$\Phi^T D\Phi = \bar{D}. \tag{2.28}$$

Definindo $\bar{f} = \Phi^T f$, a equação 2.26 pode ser escrita como $3n$ equações independentes:

$$\bar{M}_i\ddot{\bar{q}}_i + \bar{D}_i\dot{\bar{q}}_i + \bar{K}_i\bar{q}_i = \bar{f}_i, \tag{2.29}$$

onde $\bar{M}_i$ é o i-ésimo elemento diagonal de $\bar{M}$ e similarmente para as quatro. Desta forma, o sistema de equações é linearmente independente e cada elemento de $\bar{q}$ é um deslocamento modal correspondente à combinação de deslocamentos em q.

A análise modal é um método eficiente visto que cada uma das equações desacopladas podem ser resolvidas analiticamente [16] e limitações de estabilidade dos métodos de integração numérica são eliminadas. Entretanto, a linearização das equações não-lineares

originais significa que a solução será apenas uma aproximação de primeira ordem da solução verdadeira. Entretanto, quando se quer obter animações em tempo real não se procura uma animação exata, e sim uma animação fisicamente plausível.

Hauser et al. [16] apresentaram um esquema baseado em partículas para calcular forças usando o modo de deformação modal (ver equação 2.23), computando forças no campo de deformação U. Nesta abordagem o cômputo da força envolve a avaliação de uma matriz pseudo-inversa e para resolvê-la usaram decomposição em valores singulares (*SVD - Single Value Decomposition* em inglês) (ver seção 2.4.2.2).

2.4.1.4 Método de elementos finitos (FEM)

O método de elementos finitos é um dos mais usados para resolver equações diferenciais parciais (EDPs) em grades irregulares. A mecânica contínua fornece as *EDPs* que serão convertidas pelo FEM em um conjunto de equações algébricas, as quais podem ser resolvidas numericamente. Neste método o volume do objeto é discretizado em um conjunto de células disjuntas (malha). Em vez de definir e resolver as equações modelando comportamentos elásticos sobre o domínio inteiro, tensores *strain* e *stress* e forças elásticas são derivadas localmente para cada nó (partícula). Conseqüentemente, a integração é executada para cada nó da malha e a função contínua $x(p,t)$ é aproximada usando o valor nodal:

$$\tilde{x}(p,t) = \sum_i x_i(t) b_i(p), \tag{2.30}$$

onde $b_i()$ são funções base nodal (funções de interpolação). (Figura 2.4).

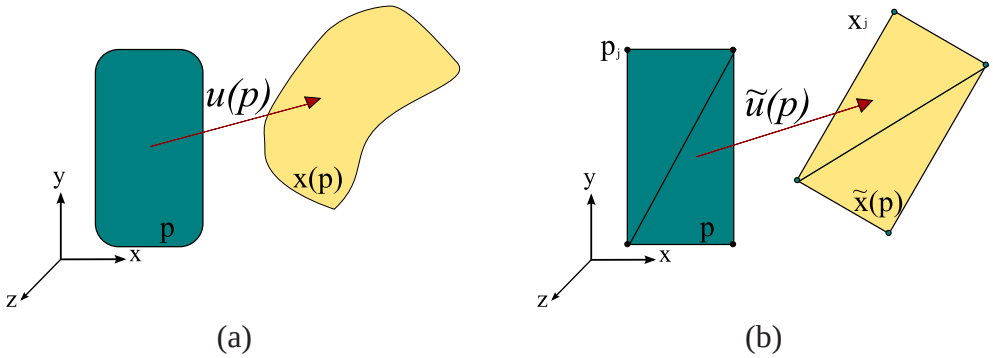


Figura 2.4: no método de elementos finitos, uma deformação contínua (a) é aproximada por uma soma de funções de base linear, definidas em um conjunto de elementos finitos (b).

Freqüentemente, é usada uma forma simples de FEM chamada *FEM explícito*, a qual

pode ser integrada implícita ou explicitamente. Neste método as massas e as forças externas são colocadas juntas nos nós da malha.

Devido a deformações, as forças atuantes nos nós de um elemento são calculadas da seguinte forma: dadas as posições dos vértices de um elemento e as funções bases b_i, o campo da deformação contínua $u(p)$ é obtido usando a equação 2.30. A partir do valor $u(p)$ são computados os tensores de *strain* $\varepsilon(p)$ e *stress* $\sigma(p)$ e a energia de deformação de um elemento é dada por:

$$E = \int_V \varepsilon(x) \cdot \sigma(x) dx, \tag{2.31}$$

onde *ponto* $(\cdot)$ representa o produto escalar componente a componente dos tensores. As forças podem ser computadas como a derivada da energia em relação às posições nodais. Em geral, a relação entre forças e posições nodais é não linear.

Usando forças elásticas linearizadas, a equação de movimento 2.20 de uma malha torna-se:

$$\begin{gathered} u = x - x_0, \\ M\ddot{u} + D\dot{u} + Ku = f_{ext}, \end{gathered} \tag{2.32}$$

onde u representa o campo de deformação formado por x e x_0, que contêm as posições dos nós atuais e em repouso, respectivamente, $M \in R^{n \times n}$ é a matriz da massa, $D \in R^{n \times n}$ é a matriz de amortecimento, $K \in R^{3n \times 3n}$ é a matriz de rigidez e $f_{ext} \in R^n$ são as forças externas aplicadas.

2.4.1.5 Sistemas massa-mola

Nos sistemas massa-mola [7, 4, 44], os modelos consistem de pontos-massa (partículas) conectados entre si por uma rede de molas. O estado do sistema em um tempo t é definido pelas posições x_i e velocidades v_i dos pontos-massa $i = 1..n$. (Figura 2.5).

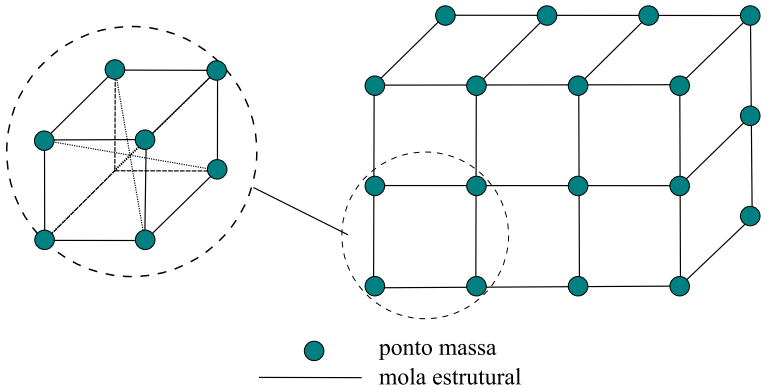


Figura 2.5: um sistema massa-mola.

O alongamento das molas gera forças elásticas em cada massa, devido às conexões das molas com seus vizinhos, além das forças externas como gravidade, fricção e outras. As molas podem ser modeladas como elásticos usando a *Lei de Hooke*:

$$f_i = k_s(|x_{ij}| - l_{ij})\frac{x_{ij}}{|x_{ij}|}, \tag{2.33}$$

que gera forças elásticas lineares, onde k_s é a rigidez (constante elástica da mola) que conecta os dois elementos, x_{ij} é o vetor diferença entre dois pontos-massa e l_{ij} é o comprimento da mola em estado de repouso.

Corpos físicos não são perfeitamente elásticos, eles dissipam energia durante a deformação. Para simular esta característica, um termo de viscosidade pode ser adicionado:

$$f_i = k_d(v_j - v_i), \tag{2.34}$$

onde v_i e v_j são as velocidades das massas e k_d é a constante de amortecimento da mola. O movimento de cada partícula é governado pela *Segunda Lei de Movimento de Newton* (2.1) $f_i = m_i\ddot{x}_i$, para um sistema de partículas inteiro, esta equação pode ser expressa como :

$$M\ddot{x} = f(x, v), \tag{2.35}$$

onde M é uma matriz massa diagonal $3n \times 3n$. Logo, o sistema massa-mola requer a solução de um sistema de equações diferenciais ordinárias, usando um esquema de integração.

Sistemas massa-mola são intuitivos e simples de se implementar. Entretanto, eles não são necessariamente exatos porque não são construídos sobre elasticidade contínua.

Aplicações de sistemas massa-mola são muito usados na simulação de tecidos, onde Baraff e Witkin [4] propuseram um método baseado no sistema massa-mola com um método de integração implícito estável, para um intervalo de tempo maior.

2.4.2 Métodos livres de malhas

As abordagens livres de malhas ou sem malhas se originam no método de elementos finitos e também requerem a solução de equações diferenciais parciais. Diferentemente deste, entretanto, métodos livres de malha, não requerem nenhuma informação de conectividade, uma vez que os objetos são tratados como sistemas de partículas [15]. A seguir são descritas algumas abordagens importantes.

2.4.2.1 Animação baseada em pontos

Recentemente, a combinação de física sem malhas com superfícies amostradas por pontos, conhecida como *Animação baseada em pontos* tornou-se uma abordagem popular. Muller et al. [27] introduziram um trabalho baseado na mecânica contínua para simular elasticidade, plasticidade e derretimento de objetos, onde as forças elásticas do corpo são derivadas via densidade de energia de tensão (*strain energy density*):

$$U = \frac{1}{2}\varepsilon\sigma, \tag{2.36}$$

onde ε é o tensor *strain* e σ o tensor *stress* (seção 2.4.1.1).

A idéia básica é amostrar o volume com partículas e aproximar a função de deslocamento contínuo $u(x)$ usando a aproximação de Taylor de primeira ordem (seção 2.5.1). Isto significa essencialmente combinar o deslocamento de um conjunto discreto de partículas a fim de obter uma função contínua de deslocamento $\tilde{u}(x)$. Para calcular a tensão usando o tensor de *strain* de Green, é preciso de uma aproximação contínua das derivadas de deslocamento (equação 2.21).

Lancaster et al. [23] apresentam uma interpolação de primeira ordem de deslocamento das partículas, usando o método dos mínimos quadrados móveis (MLS). Neste método, a derivada de deslocamento numa partícula ∇u é avaliada usando a soma de pesos do deslocamento dos vizinhos, que são um conjunto de partículas dentro de uma distância máxima. Com esta abordagem é possível simular o comportamento de materiais variados, entre rígidos, elásticos e plásticos.

2.4.2.2 Método sem malhas baseado em casamento de formas

Müller et al. [26] propuseram uma técnica de animação sem malhas, que não se ajusta a nenhuma das categorias precedentes. Os nós da malha são tratados como partículas e animados como um sistema de partículas simples, sem conectividade, onde a cada intervalo de tempo, a configuração original de pontos (estado de repouso da malha) é transformada em uma nova configuração usando casamento de formas. A nuvem de pontos adaptada produz posições alvo para todas partículas. Logo, cada partícula é transladada para sua posição alvo.

O algoritmo tem dois componentes principais: (1) encontrar uma transformação rígida ótima que aproxime uma nova posição e a orientação do objeto (problema de correspondência) e (2) mover as partículas para as posições alvo aplicando o modelo de deformação linear, descrito na seção 2.4.1.3.

Mínimos quadrados: Em [26], o problema de correspondência é resolvido com a técnica de mínimos quadrados. Trata-se de um método clássico usado para estabelecer a melhor correspondência entre duas representações, ou seja, a correspondência com o menor erro quadrático. Neste caso, assume-se que a correspondência é conhecida e o objetivo é encontrar a melhor transformação rígida ou linear entre tais representações.

Dados dois conjuntos de pontos q_i e p_i $i = 1, 2, ...n$ num espaço m-dimensional, deseja-se encontrar parâmetros de transformação A entre estes dois conjuntos de pontos com o mínimo erro quadrático [48]:

$$\sum_i w_i (Aq_i - p_i)^2, \tag{2.37}$$

onde w_i representa o peso da partícula i que, para aplicações de animação física, corresponde naturalmente a m_i, ou seja, à massa da partícula.

Kanatani [21] utilizou o método de mínimos quadrados a fim de encontrar a melhor rotação possível. Ele expandiu a equação 2.37 para encontrar:

$$A = (\sum m_i p_i q_i^T)(\sum m_i q_i q_i^T)^{-1} = A_{pq} A_{qq}, \tag{2.38}$$

onde A_{pq} é uma matriz de correlação e A_{qq} é uma matriz simétrica que pode conter escala mas não rotações. Conclui-se portanto que a estimativa de A requer que se estime o componente de rotação da matriz A_{pq}, o que pode ser realizado através de algum método de decomposição. Em particular, o trabalho de Müller et al. [26] emprega decomposição polar (veja adiante).

Alexa et al. [1] também usam o método de mínimos quadrados num contexto ligeiramente diferente. Uma vez que as correspondências ponto-a-ponto para todos os pontos dos objetos são fixadas, se define uma transformação elástica entre os objetos, que satisfaça as correspondências. Para diminuir a distorção das formas intermediárias, é determinada a parte rígida da transformação, sendo que a parte elástica é interpolada separadamente usando decomposição por valores singulares (SVD).

Métodos de decomposição: As matrizes homogêneas 3×3 ou 4×4 podem ser decompostas em matrizes primitivas de translação, rotação, escala, inclinação e perspectiva. Em animação, a obtenção destas componentes permite separar o comportamento rígido do objeto do comportamento deformável. Em particular, se a matriz foi obtida por algum processo de interpolação direta, apenas a componente de rotação é distorcida. Em outras palavras, se há interesse em realizar interpolação, todas as componentes à exceção da de

rotação podem ser interpoladas diretamente. Observe-se que uma matriz de rotação Q é um tipo de matriz *ortogonal* especial, já que além de $Q^TQ = I$, seu determinante é $det(Q) = +1$.

Existem três decomposições ortogonais principais:

- **decomposição em valores singulares (*SVD*)**. Decompõe uma matriz M em três partes:

  $$M = UKV^T,$$

  onde U e V são ortogonais e K é uma matriz diagonal e positiva. Seu custo computacional é alto e as matrizes ortogonais produzidas não são usadas, porque a matriz que contém a rotação pura pode ser dividida de muitas formas nas duas matrizes da decomposição, o que prejudica a animação;

- **decomposição QR**. Decompõe a matriz M em duas partes:

  $$M = QR,$$

  onde Q é ortogonal e R triangular inferior. Embora este algoritmo seja simples, a decomposição seja única, a matriz ortogonal extraída não tem significado físico uma vez que é dependente do sistema de coordenadas usado.

- **decomposição polar**. Decompõe a matriz M em duas partes:

  $$M = QS, \tag{2.39}$$

  onde Q é ortogonal e S é simétrico definido positivo. A decomposição é única, independente do sistema de coordenadas, simples e eficiente para se computar. A parte ortogonal Q é a mais próxima possível da matriz ortogonal M. Shoemake e Duff [36] apresentaram uma solução para encontrar a parte rotacional Q, usando decomposição polar. Eles representam a parte simétrica S em função da matriz M:

  $$S^2 = M^TM,$$

  sendo que para avaliar $S^{-1} = (\sqrt{M^TM})^{-1}$ é preciso diagonalizar a matriz simétrica. Logo, a parte rotacional é:

  $$Q = MS^{-1}.$$

Diagonalização de matrizes: O *algoritmo de Jacobi* [41] é um algoritmo simples e estável usado para a diagonalização de matrizes e o cômputo de auto-vetores. O funcionamento do algoritmo se caracteriza pela aplicação de sucessivas operações elementares chamadas de *Rotações de Jacobi*.

Uma matriz de *Rotação de Jacobi* P_{pq} contém 1's ao longo da diagonal, à exceção dos elementos $\cos\phi$ nas fileiras e nas colunas p e q. Além disso, todos os demais elementos fora das diagonais são zero, isto é, exceto os valores $\sin\phi$ e $-\sin\phi$ nas posições q, p e p, q. O ângulo de rotação ϕ de uma matriz inicial A pode ser escolhido como:

$$\cot(2\phi) = \frac{\alpha_{qq} - \alpha_{pp}}{2\alpha_{pq}}. \tag{2.40}$$

Então, a matriz de *Rotação de Jacobi* correspondente, que elimina o elemento P_{pq} fora da diagonal é:

$$P_{pq} = \begin{bmatrix} 1 & & & & & & & & 0 \\ & \ddots & & & \vdots & & & & \\ & & \cos\phi & \cdots & 0 & \cdots & \sin\phi & & \\ & \cdots & 0 & \cdots & 1 & \cdots & 0 & \cdots & \\ & & -\sin\phi & \cdots & 0 & \cdots & \cos\phi & & \\ & & & & \vdots & & & \ddots & \\ 0 & & & & & & & & 1 \end{bmatrix}, \tag{2.41}$$

Geralmente são realizadas de 5 a 10 *rotações de Jacobi* para diagonalizar uma matriz 4×4, como mostra a Figura 2.6.

2.5 Métodos de integração

Para simular a dinâmica de um objeto deformável é preciso computar as coordenadas do mundo dependentes do tempo $x(p,t)$, para todos os pontos p do objeto. Dado $x(p,t)$ podemos mostrar as configurações $x(0), x(\Delta t), x(2\Delta t)...$, uma seguida de outra, resultando numa animação de objetos onde: Δt é um intervalo de tempo fixo da simulação e $x(t)$ representa um campo vetorial no instante t. Este campo é definido implicitamente como a solução de uma equação diferencial, que deriva da equação 2.1 da *Segunda Lei de Movimento de Newton* da forma:

$$\ddot{x} = F(\dot{x}, x, t), \tag{2.42}$$

onde $\ddot{x}$ é o campo aceleração, $\dot{x}$ é o campo velocidade (segunda e primeira derivada de x respectivamente) e F é uma função geral dada pelo modelo físico, que depende de

forças elásticas do objeto deformável. Para encontrar a solução para $x(t)$, esta equação diferencial de segunda ordem é ré-escrita como um conjunto de duas equações de primeira ordem:

$$\dot{x} = v, \tag{2.43}$$

$$\dot{v} = F(v, x, t). \tag{2.44}$$

O conjunto discreto de valores $x(0), x(\Delta t), x(2\Delta t), ...,$ do campo vetorial x, é obtido resolvendo numericamente estes sistemas de equações, integrando-os mediante algum esquema de integração numérica de equações diferenciais ordinárias.

Os esquemas de integração são avaliados por dois critérios principais: sua estabilidade e sua exatidão:

- a **exatidão** é medida pela convergência em relação ao tamanho do intervalo de tempo Δt, ou seja, de primeira ordem $O(\Delta t)$, de segunda ordem $O(\Delta t^2)$, etc;
- em aplicações de simulação interativa, a **estabilidade** pode se considerar mais importante do que a exatidão, desde que o resultado da animação seja fisicamente plausível [26].

Os esquemas de integração podem ser de dois tipos: implícitos ou explícitos.

- *Integração implícita:* garante estabilidade independente do intervalo de tempo selecionado. A resolução das suas fórmulas é complexa, portanto são custosos computacionalmente. Os objetos a serem simulados não podem ser muito complexos geometricamente.
- *Integração explícita:* são métodos menos estáveis que os esquemas de integração implícita, porém são rápidos de se computar.

Dentre os métodos de integração, são de especial interesse o *Método de Taylor*, o *Método de Verlet* e o *Método de Euler*.

2.5.1 Método de Taylor

Segundo o *Teorema de Taylor*, dada uma função $x(t)$ com n-ésima derivada contínua em $[t_0, t_1]$ e derivável no intervalo aberto (t_0, t_1). Então, o polinômio de Taylor $P_n(t_1, t_0)$ é definido por:

$$P_n(t_1, t_0) = \sum_{k=0}^{n} \frac{f^{(k)}(t_0)}{k!}(t_1 - t_0)^k. \tag{2.45}$$

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:

$$R_n(t_1, t_0) = \frac{f^{(n+1)}(\bar{t})}{(n+1)!}(t_1 - t_0)^{n+1}, \quad \bar{t} \in (t_0, t_1). \tag{2.46}$$

- forma de Cauchy:

$$R_n(t_1, t_0) = \frac{f^{(n+1)}(\bar{t})}{n!}(t_1 - \bar{t})^n(t_1 - t_0), \quad \bar{t} \in (t_0, t_1). \tag{2.47}$$

Já que o problema consiste em aproximar uma solução para a equação diferencial, temos:

$$\dot{x}(t) \approx f(t, x(t)), \qquad \text{onde} \qquad x(t_0) = x_0. \tag{2.48}$$

Existem vários métodos que se baseiam no Teorema de Taylor, como por exemplo o Método de Verlet, que projeta o valor da função para os instantes de tempo posterior e anterior a t_i, e o Método de Euler, que usa $n = 2$, no intervalo $[t_i, t_{i+1}]$. Uma discussão mais detalhada destes e outros métodos de integração pode ser encontrada em [10].

2.5.2 Método de Verlet

O método de Verlet utiliza duas expansões:

$$x(t_{i+1}) = x(t_i) + \dot{x}(t_i)\Delta t + \frac{1}{2}\ddot{x}(t_i)\Delta t^2 + \frac{1}{6}x^3(t_i)\Delta t^3 + O(\Delta t^4), \tag{2.49}$$

e

$$x(t_{i-1}) = x(t_i) - \dot{x}(t_i)\Delta t + \frac{1}{2}\ddot{x}(t_i)\Delta t^2 - \frac{1}{6}x^3(t_i)\Delta t^3 + O(\Delta t^4). \tag{2.50}$$

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

$$x(t_{i+1}) = 2x(t_i) - x(t_{i-1})\Delta t + \ddot{x}(t_i)\Delta t^2 + O(\Delta t^4). \tag{2.51}$$

2.5.3 Método de Euler

O esquema de integração de Euler pode ser integrado tanto implícita como explicitamente, a saber:

- esquema de integração explícito, onde a derivada do tempo é substituída por diferenças finitas, as equações são resolvidas, para encontrar os dados do seguinte intervalo de tempo $t + \Delta t$:

$$x(t + \Delta t) = x(t) + \Delta t v(t), \tag{2.52}$$

$$v(t + \Delta t) = v(t) + \Delta t F(v(t), x(t), t). \tag{2.53}$$

  Este esquema de integração é chamado de explícito porque fornece fórmulas explícitas para encontrar a velocidade e a posição no próximo intervalo de tempo. Estes métodos são simples de se implementar mas a estabilidade depende do intervalo de tempo Δt;

- esquema de integração implícito, onde as variáveis do seguinte intervalo de tempo $t + \Delta t$ aparecem em ambos os lados das equações diferenciais.

$$x(t + \Delta t) = x(t) + \Delta t v(t + \Delta t), \tag{2.54}$$

$$v(t + \Delta t) = v(t) + \Delta t F(v(t + \Delta t), x(t + \Delta t), t). \tag{2.55}$$

  Este esquema de integração é chamado de implícito porque as variáveis ainda desconhecidas são dadas implicitamente como parte da solução de um sistema de equações. Este esquema é estável para intervalos de tempo Δt arbitrariamente longos. Porém, deve-se resolver um sistema de equações a cada intervalo de tempo.

2.5.4 Método de Euler Modificado

Este esquema é uma modificação do proposto por Euler também chamado *explícito-implícito*, onde a ordem das equações são invertidas:

$$v(t + \Delta t) = v(t) + \Delta t F(v(t), x(t), t), \tag{2.56}$$

$$x(t + \Delta t) = x(t) + \Delta t v(t + \Delta t). \tag{2.57}$$

Para atualizar a velocidade v se usa o esquema explícito de Euler e para atualizar a posição x se usa o esquema implícito de Euler. Note que o método é ainda explícito, pois $v(t + \Delta t)$ é simplesmente avaliado primeiro. Para sistemas onde as forças são independentes da velocidade, ele reduz a exatidão de segunda ordem do esquema Verlet. O esquema *explícito-implícito* de Euler é mais estável do que esquemas de integração de Euler padrão, sem nenhum custo computacional adicional, como foi mostrado por Müller et al. [26].

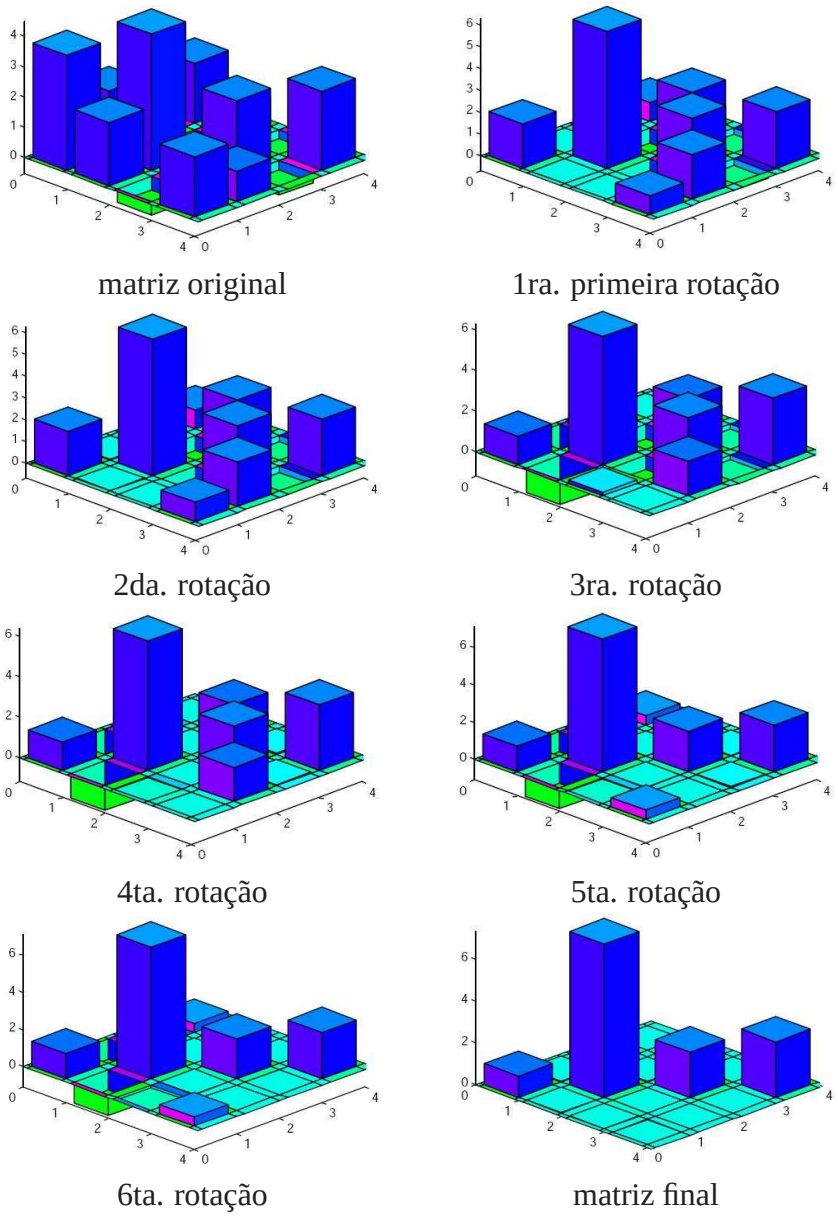


Figura 2.6: um exemplo de Diagonalização de Matrizes: algoritmo de Jacobi. Imagens extraídas da tese de doutorado de Hartono Sumali [41]

Capítulo 3

Detecção de Colisões

A detecção de colisão é um componente fundamental na simulação física e é um problema que vem sendo estudado intensamente nas últimas décadas. O foco, na maioria das pesquisas, é a detecção de colisão de objetos rígidos. Entretanto, a detecção de colisão muda dramaticamente quando os objetos a serem tratados são deformáveis [46]:

Colisão e Auto-colisão: são considerados todos os pontos de contato, inclusive aqueles que provocam auto-colisão.

Pré-processamento: tipicamente são usadas *estruturas de dados espaciais* durante o pré-processamento, mas durante a simulação estas devem ser atualizadas segundo a deformação dos objetos.

Informação de Colisão: para uma resposta real à colisão, precisa-se de informação apropriada e, para tanto, além de detectar a intersecção dos objetos é necessário obter informações sobre a profundidade de penetração, pontos de contato, etc.

Desempenho: a eficiência dos algoritmos é importante, já que a exibição em tempo real é fundamental em ambientes interativos.

Existem diversas abordagens para tratar os problemas mencionados, mas é importante escolher um esquema de detecção de colisões que minimize o número de testes de colisão. Uma abordagem ingênua seria comparar uma primitiva de um objeto com todas as primitivas de todos os elementos da cena, um processo com complexidade $O(M^2)$, onde M é o número total de primitivas. Obviamente, este esquema é apropriado para valores moderados de M. Entre as abordagens para detecção de colisões mais utilizadas temos: *métodos de partição do objeto, subdivisão espacial, campos de distância* e *técnicas no espaço da imagem*.

3.1 Métodos de partição do objeto

São métodos que empregam hierarquias de volumes limitantes (*bounding volume hierarchies – BVH*) para estruturar o processamento geométrico dos objetos envolvidos em colisões. Inicialmente, foram utilizadas para simulações com objetos rígidos, onde a hierarquia era computada numa etapa de pré-processamento. Quando a técnica é aplicada a objetos deformáveis, tais hierarquias devem ser atualizadas para cada iteração no tempo, requerendo portanto uma atualização eficiente e um esforço computacional maior.

3.1.1 Volume limitante

Permite aproximar um objeto complexo por outro de geometria muito mais simples, ajustado ao objeto original da melhor maneira possível. Os volumes limitantes permitem realizar testes de intersecção entre objetos, reduzindo o tempo de computação através de um teste preliminar de intersecção. Desta forma, testes exatos mais custosos são realizados apenas entre pares de objetos cujos volumes limitantes se intersectam. Em alguns esquemas testes exatos podem mesmo ser evitados.

Entre os tipos de volumes limitantes temos: esferas limitantes, caixas limitantes alinhadas aos eixos coordenados (*axis-aligned bounding boxes – AABBs*), caixas limitantes orientadas (*oriented bounding boxes – OBBs*), politopos de orientação discreta (*k-direction oriented polytopes – k-DOPs*), fechos convexos (*convex hulls*), elipsóides limitantes, etc. (Figura 3.1).

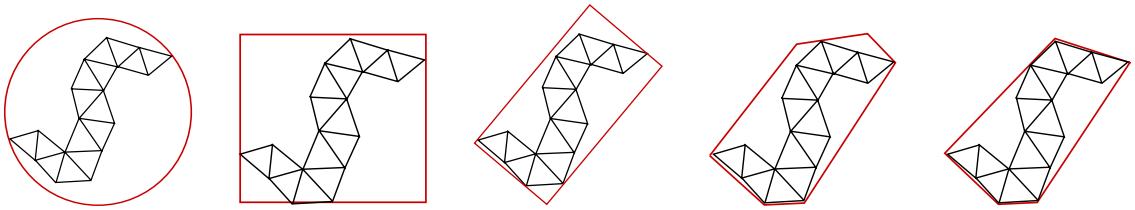


Figura 3.1: alguns tipos de volumes limitantes.

Dentre estes, existem dois que são particularmente interessantes, pela sua simplicidade: as esferas limitantes e as caixas alinhadas aos eixos coordenados.

Esferas limitantes: as esferas são um tipo de volume limitante simples, porque podem ser armazenadas usando apenas *quatro* escalares. Para verificar sobreposição entre duas esferas é necessário apenas *onze* operações aritméticas. Embora a esfera não limite o objeto de forma mais justa, sua simplicidade e o fato de que são invariantes

a rotações fazem com que as esferas sejam um tipo de volume limitante popular em ambientes dinâmicos.

A Figura 3.2 mostra duas esferas de centros c_1 e c_2 e raios r_1 e r_2, e a intersecção entre elas, sendo que duas esferas não se intersectam se:

$$(c_1 - c_2)(c_1 - c_2) > (r_1 + r_2)^2 \tag{3.1}$$

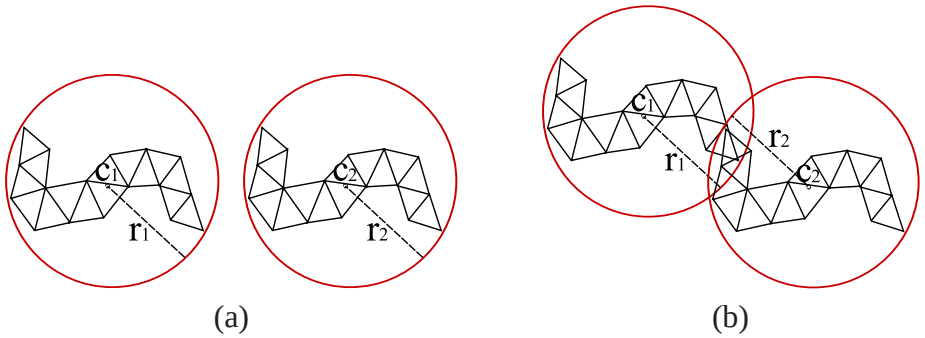


Figura 3.2: (a) duas esferas não intersectadas e (b) duas esferas intersectadas.

Caixas limitantes alinhadas com os eixos: as caixas limitantes alinhadas com os eixos coordenados (*AABBs*) são ainda mais usadas que as esferas limitantes. Embora elas precisem de mais espaço de armazenamento do que as esferas (*seis* escalares), é possível verificar sobreposição com apenas *seis* operações primitivas.

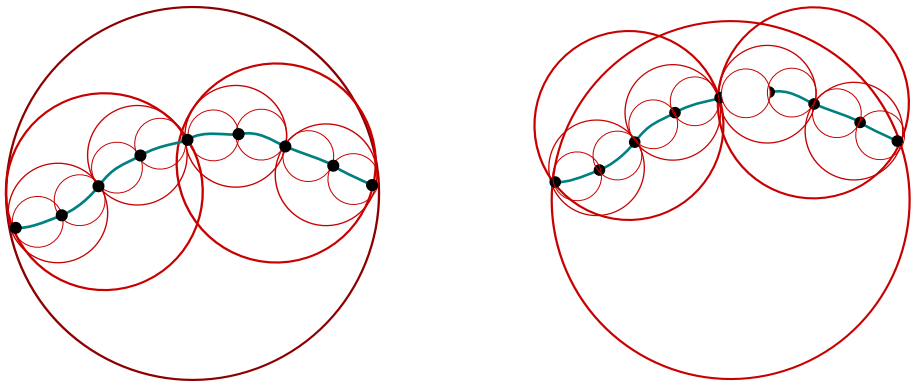
3.1.2 Hierarquia de volumes limitantes

A subdivisão em volumes limitantes forma uma hierarquia (árvore) para cada objeto. A idéia consiste em repartir recursivamente o conjunto de primitivas do objeto até que algum critério de partição seja satisfeito, por exemplo, um número máximo de primitivas em cada nó folha. Cada nó da árvore contém informação sobre o volume limitante respectivo, sendo que as folhas contêm adicionalmente informação sobre as primitivas correspondentes. Uma primitiva é a menor instância que compõe um objeto, geralmente triângulos ou tetraedros. Uma hierarquia de volumes limitantes é uma estrutura de dados eficiente para a detecção de colisão, embora o alto custo de atualização em objetos deformáveis seja uma grande desvantagem [24]. As estratégias para construir uma hierarquia dividem-se em *top-down* e *bottom-up*. (Figura 3.3).

Top-down: a idéia é começar pelo nó raiz (objeto) repartindo o conjunto de primitivas em dois subconjuntos. A heurística para a partição pode visar tanto obter subconjuntos

de cardinalidade igual ou subconjuntos cujo tamanho ou volume dos respectivos volumes limitantes sejam semelhantes. O processo repete-se recursivamente para cada subconjunto até que algum critério de parada seja satisfeito.

Bottom-up: usa a idéia oposta, ou seja, agrupa os nós folhas até obter um nó raiz. Neste caso, deve-se empregar alguma heurística para emparelhar subconjuntos relativamente próximos a cada aglutinação.



top-down bottom-up

Figura 3.3: estratégias de construção da hierarquia de volumes limitantes.

A hierarquia precisa ser atualizada a cada passo de tempo, devido à movimentação e ou deformação do objeto simulado. Os tipos de atualização são: por **reajuste** ou por **reconstrução**. No primeiro, a divisão hierárquica é mantida sendo que apenas os volumes limitantes afetados são recomputados. No segundo, toda a estrutura é recomputada. A técnica de reajuste é normalmente preferível à de reconstrução completa da hierarquia.

A hierarquia de volumes limitantes acelera o processo de detecção de colisão entre dois objetos. Cada par de nós das árvores é testado recursivamente para encontrar sobreposição. Se os volumes limitantes de determinados níveis da hierarquia se sobrepõem, e se nenhum dos nós é folha, então os nós filhos são testados até alcançar os nós folha, para os quais testes exatos entre primitivas são realizados. Se apenas um dos nós é folha, este é testado com cada filho do outro nó, até alcançar os nós folha. (Figura 3.4).

3.2 Subdivisão espacial

Os métodos de subdivisão espacial repartem o espaço de forma implícita ou explícita. Uma estrutura de dados espacial tem que ser flexível e eficiente em relação ao uso de tempo e memória. Diferentemente dos métodos de partição, usados para organizar objetos

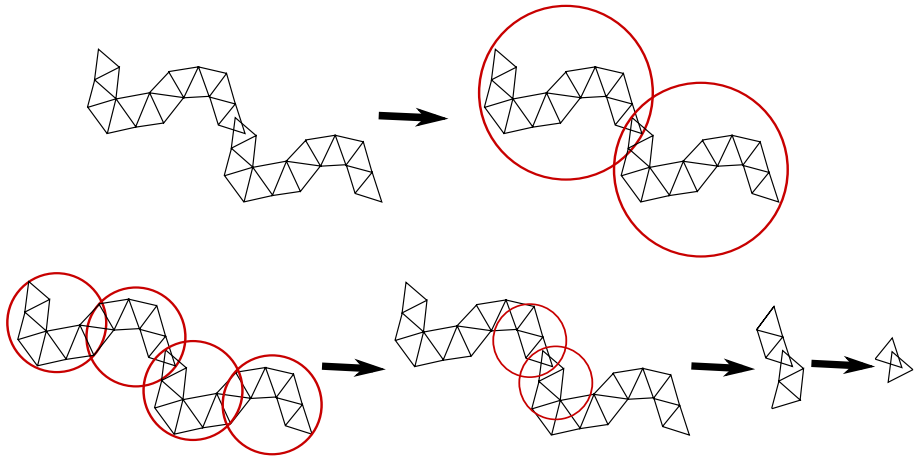


Figura 3.4: teste de intersecção entre BVHs.

hierarquicamente, a subdivisão espacial é mais frequentemente empregada para organizar o espaço onde os objetos interagem. Neste caso a idéia geral consiste em detectar regiões ocupadas por mais de um objeto.

Entre as estruturas hierárquicas mais usadas podemos citar as *octrees*, *k-d-trees* e *BSP-trees*. Outra estrutura bastante simples, não hierárquica, é a grade uniforme que costuma ser eficiente em ambientes de simulação onde as primitivas dos objetos mudam dinamicamente. (Figura 3.5).

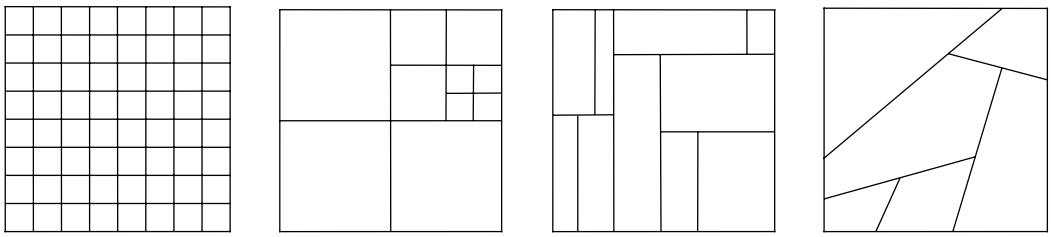
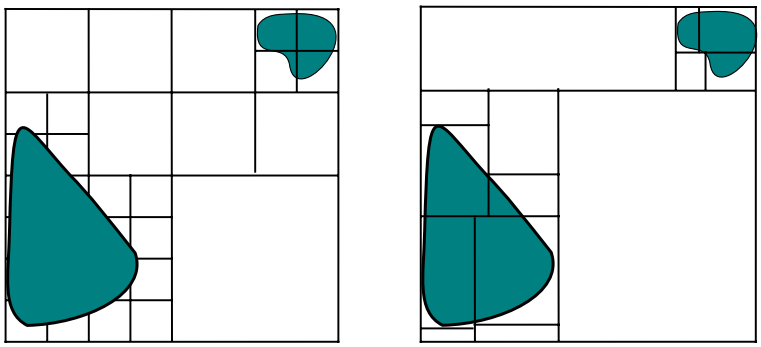


Figura 3.5: estruturas de partição do espaço (versões bidimensionais).

3.2.1 *Octrees* e *k-d-trees*

Octrees e *k-d-trees* são estruturas de dados hierárquicas que subdividem o espaço 3D em células regulares alinhadas com os eixos coordenados. O nó raiz representa o espaço total onde os objetos interagem. Tais estruturas são mais úteis em sistemas com um grande número de objetos estáticos. A partição recursiva do espaço se dá de forma a obter nos nós folha uma quantidade limitada de primitivas dos objetos estáticos. Durante a simulação, são buscadas na estrutura nós folha correspondentes às regiões ocupadas por objetos em movimento. A Figura 3.6 mostra uma representação visual destas estruturas.

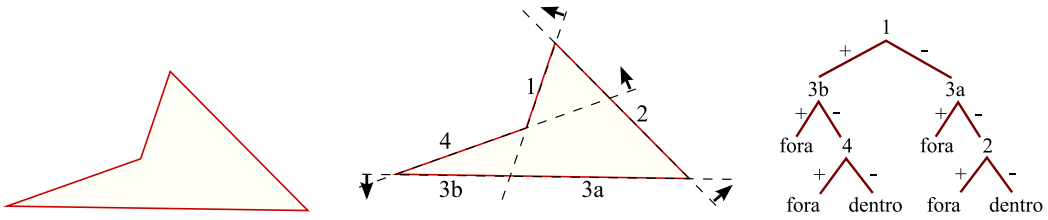


Octree k-d-tree

Figura 3.6: exemplos de partição do espaço do modelo 2D.

3.2.2 Árvores de partição binária do espaço (*BSP-trees*)

Uma *BSP-tree* é uma estrutura hierárquica que subdivide o espaço em planos orientados arbitrariamente. Pode-se dizer que é uma *k-d-tree* generalizada (Figura 3.7). As células resultantes do processo de subdivisão são politopos convexos.



forma original partição da cena àrvore

Figura 3.7: BSP-tree

A maior vantagem dessas estruturas é a sua capacidade de se adequar de forma mais justa às formas dos objetos, embora o custo de cada teste de intersecção seja mais elevado. A busca de um ponto em uma *BSP-tree* se dá da raiz para as folhas e em cada nó um teste classifica o ponto contra o semi-espaço plano que define a divisão. A busca então prossegue através da sub-árvore que contém o ponto. (Figura 3.8).

3.2.3 Grades Uniformes

Uma grade é uma subdivisão uniforme do espaço $\mathcal{R}^3$ em células retangulares chamadas voxels. Cada voxel é uma caixa alinhada com os eixos coordenados e contém elementos dos objetos que a intersectam (Figura 3.9).

Na detecção de colisões, as grades são úteis para rejeitar rapidamente pares de objetos que não se intersectam [47, 13, 50], sendo recomendadas para verificar intersecções em ambientes complexos [45], contendo centenas de objetos.

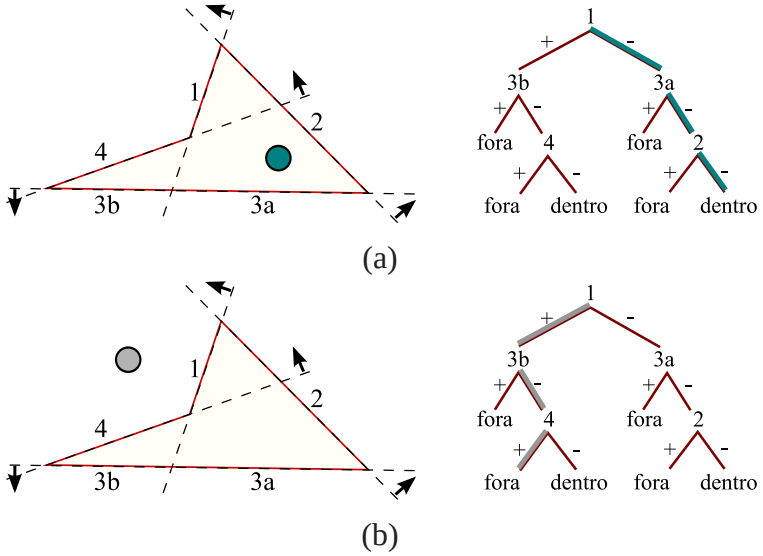


Figura 3.8: consulta em BSP-tree: (a) ponto dentro do objeto e (b) ponto fora dele.

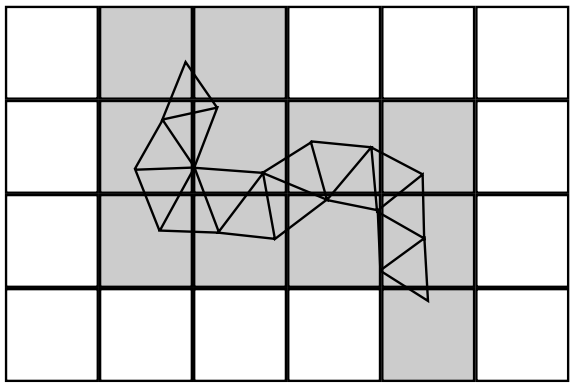


Figura 3.9: uma Grade Uniforme contendo um objeto.

Um algoritmo que usa grades uniformes procura encontrar todos os pares de células (em grades alinhadas) não vazias que se intersectam e verificar intersecção entre as primitivas contidas nestas células. O principal benefício do uso de grades é o acesso em tempo constante às partições do espaço (células). Por outro lado, o processo é bastante sensível ao tamanho estipulado da grade. Grades excessivamente finas levam a uma grande quantidade de células ocupadas por cada primitiva além de um desperdício de memória na representação de células vazias. Por outro lado, grades muito grosseiras têm pouca serventia como filtro espacial já que levam a células ocupadas por muitas primitivas.

O método *Hashing* espacial de Teschner et al. [45] se baseia no uso de uma grade uniforme implementada através de uma tabela de dispersão (*hash*). Cada célula tem três coordenadas inteiras i, j e k que a associam a uma posição na tabela de dispersão usando uma função $h = hash(i, j, k)$. Este esquema atenua o desperdício de memória mantendo

acesso rápido às células. Cada célula contém uma lista das primitivas de objetos que a intersectam, mas listas somente são alocadas para células que contêm algum elemento. O esquema assume que objetos são representados por malhas tetraedrais. A cada quadro da animação, os tetraedros de cada objeto são inseridos nas células que os intersectam. Numa segunda etapa, os vértices das malhas são buscados na estrutura permitindo assim encontrar os *vértices colididos*, que são os vértices que penetram tetraedros de outros objetos ou do mesmo objeto (auto-colisão). A Figura 3.10 ilustra o processo.

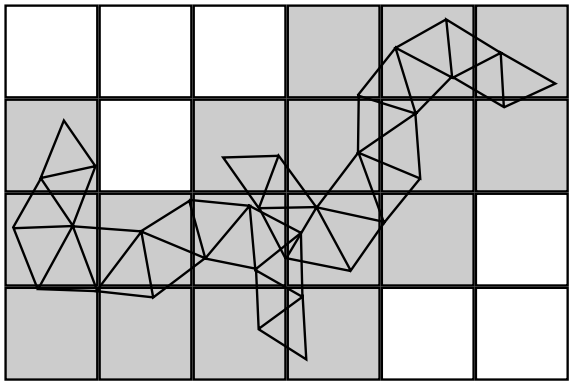


Figura 3.10: intersecção de dois objetos numa tabela *Hash*.

3.3 Campos de distância

É um campo escalar que especifica a distância mínima de um objeto de superfície fechada a todos os pontos num campo, permitindo determinar se estes pontos estão dentro ou fora do objeto. As abordagens de campos de distância não têm restrições quanto à topologia do objeto, sendo capazes de equilibrar desempenho e precisão em relação à resolução do campo de distância. Campos de distância são gerados num passo de pré-processamento, geralmente de alguns segundos de duração. As estruturas de dados mais comumente usadas para representar campos de distância são:

- grades uniformes: contêm valores de distância para o centro de cada voxel da grade, sendo que pontos intermediários são estimados por interpolação linear. Possuem, portanto, resolução limitada na representação de formas com características afiadas;
- *BSP-trees*: utilizam aproximação linear em regiões do campo de distância, mas sua construção é custosa e a descontinuidade entre células é mais difícil de resolver do que em *ADFs* (Figura 3.11);

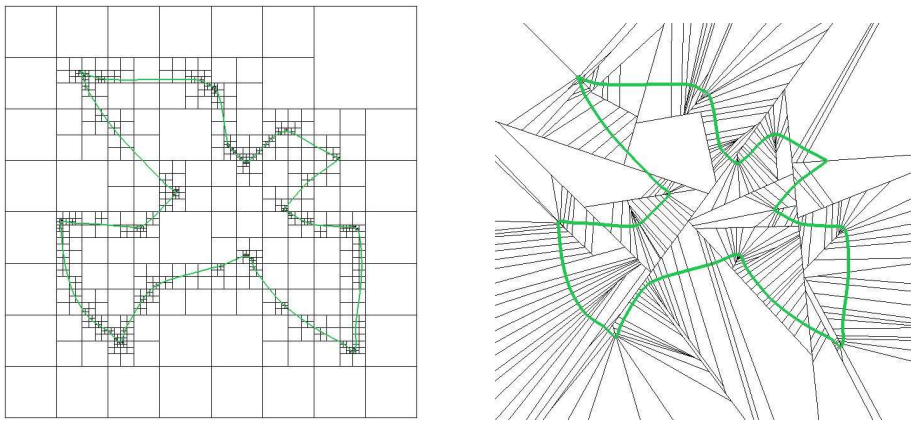
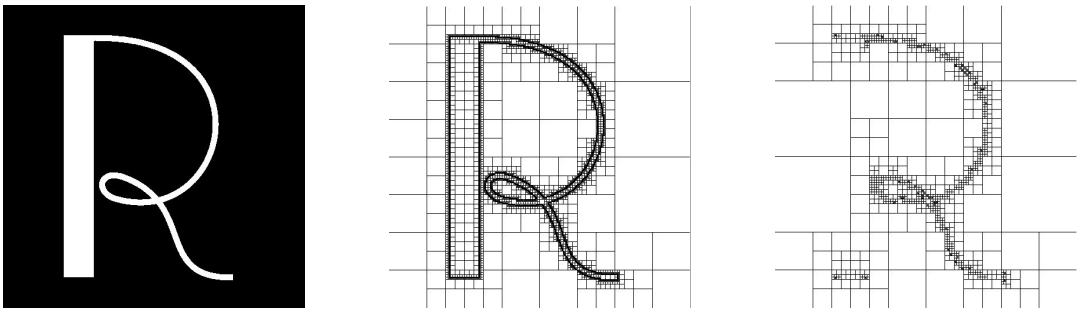


Figura 3.11: mostra-se uma ADF com 895 células e uma BSP-tree de 254 células. Imagem extraida do trabalho de Wu et al. [49]

- campos de distância amostrados adaptativamente (*adaptive distance fields –ADFs*): armazenam os dados numa *octree*, permitindo incrementar a taxa de amostragem em regiões com maior detalhe. Na construção de uma *ADFs*: cada célula é subdividida até que o resultado da interpolação tri-linear aproxime o campo de distância original dentro de uma margem de erro estipulada ou até que um limite máximo de subdivisão seja alcançado. Esta regra de subdivisão difere de *octree* padrão onde cada célula que não esteja completamente dentro ou fora do objeto é dividida (Figura 3.12).



(a) (b) (c)

Figura 3.12: campos de distância adaptativos: (a) forma original, (b) quadtree 3-color, 23573 células e (c) ADF, 1713 células. Imagens extraídas do trabalho de Frisken et al. [12].

A construção de um campo de distância normalmente pressupõe que os objetos de interesse são representados por malhas triangulares, principalmente em aplicações de detecção de colisão. Existem essencialmente duas abordagens para montar esta estrutura de dados:

- os *métodos de propagação* avaliam uma faixa estreita de pontos próximas à superfície e através da busca de vizinhos, a informação é propagada para regiões mais distantes do volume [34];

- nos métodos baseados em *diagramas de Voronoi*, estas estruturas são construídas para faces e vértices da malha. Cada região de Voronoi é então encapsulada num poliedro limitante e estes são fatiados ao longo do volume, resultando em polígonos que são rasterizados gerando os voxels do campo de distância [8] (Figura 3.13). Recentemente, esta abordagem foi melhorada usando programação de GPUs [37].

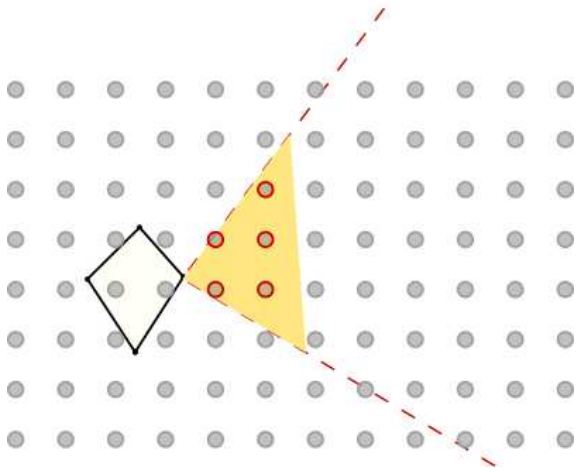


Figura 3.13: construção de campos de distância usando diagramas de Voronoi.

Campos de distância podem ser usados para detecção de colisão entre objetos A e B avaliando a distância entre A e vértices de B ou vice-versa. Em particular, se A é um objeto deformável e B é um objeto rígido, apenas os vértices de A precisam ser testados. Assume-se que um vértice v colide se a distância entre v e o outro objeto é menor que zero. Para contornar problemas de amostragem do campo de distância, entretanto, é comum empregar uma margem de erro ϵ e considerar que v colide apenas se a distância é menor que ϵ (Figura 3.14).

É importante observar que, além de obter profundidade de penetração, este método também fornece as normais no ponto de contato, que podem ser usadas na resposta à colisão [11].

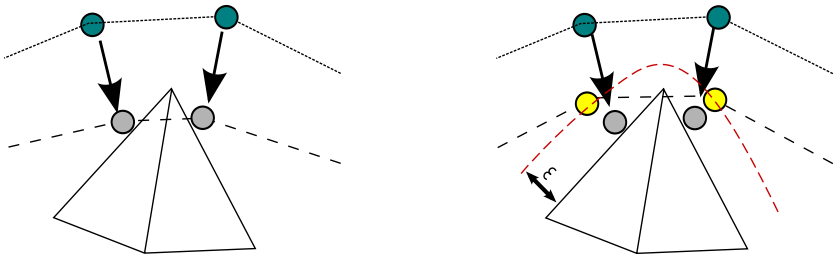


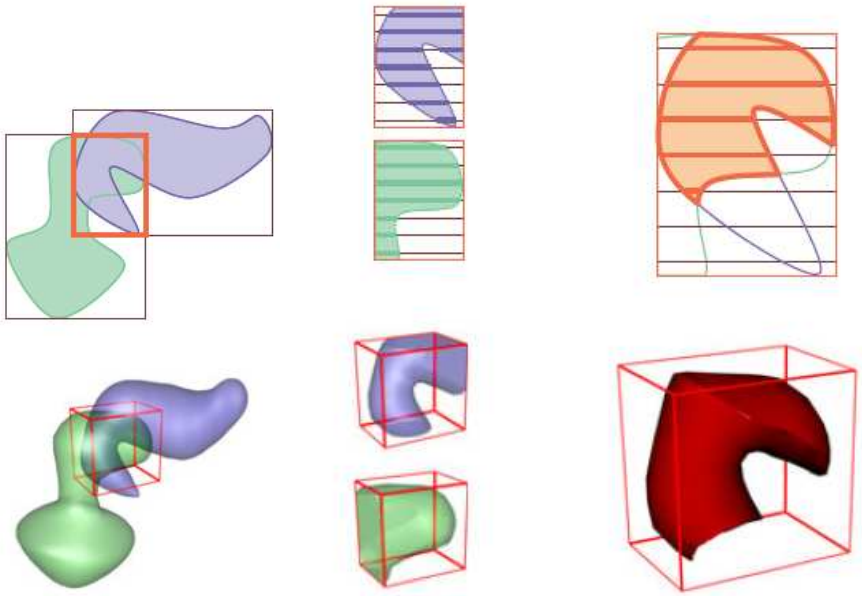
Figura 3.14: (a) sem margens de erro artefatos de interpenetração de vértices podem ocorrer durante a detecção de colisões e (b) introduzindo um ϵ-offset resolve o problema.

3.4 Técnicas no espaço da imagem

As técnicas no espaço da imagem trabalham geralmente com superfícies trianguladas e representações discretizadas dos objetos. Não são influenciadas por mudanças de topologia nem fornecem informação de colisão exata, embora esta possa ser estimada através de um pós-processamento. Além disso, técnicas no espaço da imagem costumam se beneficiar quando implementadas em hardware gráfico (GPU) [43].

A idéia principal é empregar um ou mais buffers de profundidade (*z-buffer*) para detectar interferência entre projeções de objetos. Em geral, esta abordagem está restrita a objetos convexos [3, 35], embora seja possível estendê-la para objetos côncavos de complexidade limitada usando um esquema de pré-ordenação [28].

O uso desta técnica na colisão de objetos deformáveis é problemático, embora algum sucesso tenha sido reportado. Por exemplo, a abordagem denominada *Layered Depth Image – LDI* [17] é capaz de lidar com objetos deformáveis de forma arbitrária através de um processo de três estágios. No primeiro, uma intersecção entre as AABBs dos dois objetos é computada. Caso a intersecção seja não-nula, um buffer de profundidade em camadas (LDI) é computada para a AABB delimitando a região de intersecção. Finalmente, duas consultas usando as LDIs são computadas: uma consulta para detectar a intersecção entre os volumes dos dois objetos e outra consulta para detectar os vértices ou outras primitivas de um objeto que penetram o outro objeto. A Figura 3.15 ilustra esses três estágios. É importante notar que esta técnica não lida com auto-interseções.



(a) (b) (c)

Figura 3.15: intersecção no espaço da imagem usando LDIs (2D e 3D): (a) intersecção *AABB*, (b) geração do LDI com o volume e (c) cômputo da intersecção do volume. Imagens extraídas do trabalho de Heidelberger et al. [17].

Capítulo 4

Resposta às colisões

Após a identificação das partes dos objetos que se interpenetram ou que estão em vias de o fazer, a resposta às colisões envolve a modificação de diversos parâmetros físicos dos objetos envolvidos, tipicamente posição, orientação e velocidade, de tal forma que uma configuração fisicamente plausível seja obtida [33]. Existem dois esquemas para obter a resposta à colisões de objetos deformáveis: os métodos baseados em restrições e os baseados em penalidades.

Os métodos baseados em restrições evitam a interpenetração entre objetos. Algumas abordagens deste tipo se baseiam na solução de problemas de complementaridade linear (*Linear Complementarity Problem – LCP*) [22], onde as forças de contato são obtidas da solução do LCP. Estes métodos são particularmente interessantes na dinâmica dos corpos rígidos, já que eles fornecem menos graus de liberdade [22]. Por outro lado, Pauly et al. [31] propuseram um método para encontrar forças de contato entre objetos semi-rígidos onde a superfície de contato é determinada através de restrições. Esta abordagem, entretanto, depende de um modelo de deformação que preserve o volume.

Já os métodos baseados em penalidades computam uma força de resposta para cada ponto colidido cujo valor está relacionado a uma medida de interpenetração. Conseqüentemente, o esforço numérico cresce com a intensidade da penetração [25], o que torna necessária uma computação robusta da profundidade de penetração dos pontos colididos.

Numa comparação entre os métodos citados, pode-se dizer que os métodos baseados em penalidades são mais rápidos, enquanto que os métodos baseados em restrições são mais robustos permitindo intervalos de tempo maiores. Entretanto, os sistemas de restrições são mais custosos e inadequados para resolver colisões em tempo real. Observando as vantagens de ambos métodos, Spillman et al. [38] apresentaram uma solução híbrida procurando conciliar a eficiência dos métodos baseados em penalidades com a

estabilidade numérica dos métodos baseados em restrições.

A implementação de um esquema de resposta às colisões normalmente envolve a determinação de dois parâmetros geométricos: a profundidade de penetração e, para objetos deformáveis, a região de deformação. Descrevemos abaixo algumas das abordagens mais relevantes para o cálculo desses parâmetros.

4.1 Profundidade de penetração

O cálculo da profundidade de penetração é um dos passos fundamentais para o processo de separação ou resposta à colisão, principalmente nos métodos baseados em penalidades. A profundidade de penetração entre dois objetos é a translação mínima necessária para separá-los (Figura 4.1).

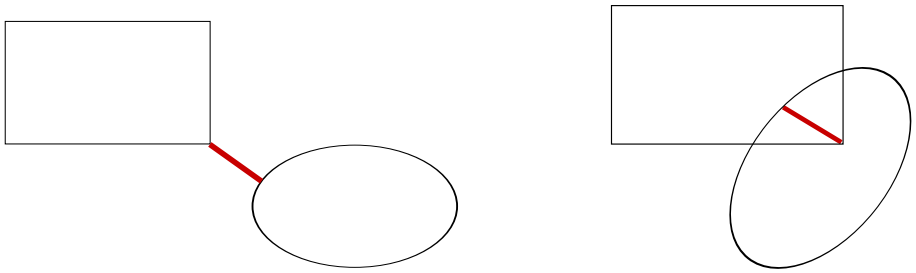


Figura 4.1: (a) objetos separados por uma distância d e (b) profundidade de penetração p entre dois objetos em colisão.

Existem diversos algoritmos para determinar a profundidade de penetração. Entre os mais utilizados estão o método de Gilbert, Johnson e Keerthi (GJK) [14] e o método de expansão de politopos (do inglês *Expanding Polytope algorithm* - *EPA*) de Gino Van den Bergen [6]. Estes algoritmos usam a *Soma de Minkowski* e o conceito de Configuração do Espaço de Obstáculo (do inglês *Configuration Space Obstacle*-CSO).

Soma de *Minkowski*: a *Soma de Minkowski* de dois objetos convexos (politopos) A e B é definida por:

$$A + B = \{x + y : x \in A, y \in B\}, \tag{4.1}$$

onde 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 dos 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 (Figura 4.2).

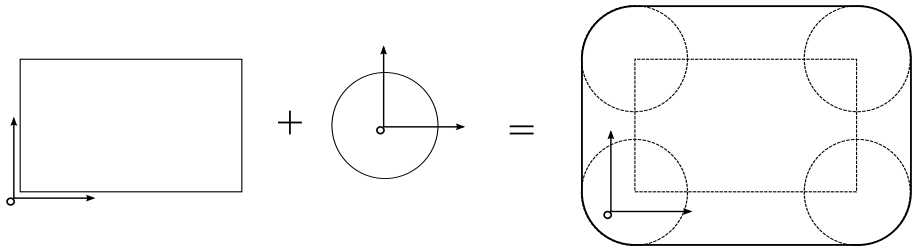


Figura 4.2: a *Soma de Minkowski* de uma caixa e uma esfera.

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* [5, 6] 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 $b \in B$ [33].

Configuração do Espaço de Obstáculo: Van den Bergen [6] usa o conceito de Configuração do Espaço de Obstáculo de dois politopos A e B para computar a distância entre eles.

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 na detecção de colisão visto que A e B se intersectam se, e somente se, sua CSO contém a origem:

$$A \cap B \neq \emptyset \equiv 0 \in A - B. \tag{4.2}$$

Mais ainda, sua distância é dada por:

$$d(A, B) = \min\{\|x\|\colon x \in A - B\}. \tag{4.3}$$

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

$$p(A, B) = \min\{\|x\|\colon x \in A - B\}. \tag{4.4}$$

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 (Figura 4.3).

O algoritmo *Gilbert-Johnson-Keerthi (GJK)*: trata-se de um método iterativo para computar a distância e a profundidade de penetração entre pares de objetos convexos. Por ser um método iterativo, o algoritmo é suscetível a erros numéricos, que podem

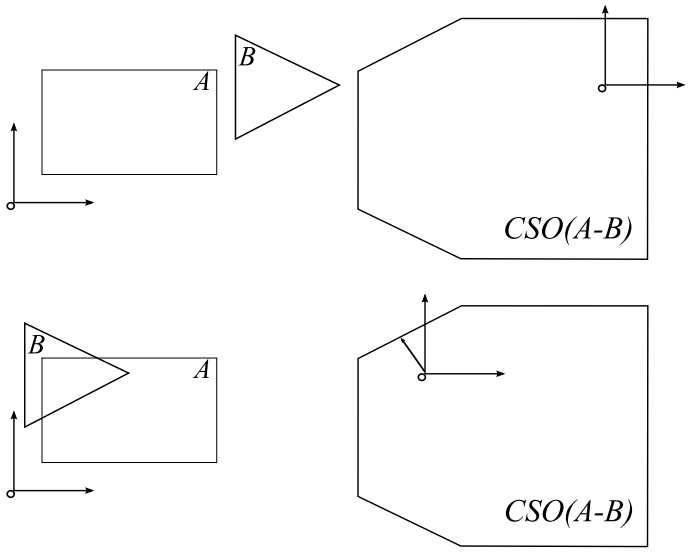


Figura 4.3: criação do CSO: (a) se A e B não se intersectam e (b) se A e B se intersectam.

causar um comportamento ruim. Por consseqüência, devem-se tomar alguns cuidados na implementação dos detalhes numéricos deste algoritmo [14] (Figura 4.4).

O algoritmo de expansão de politopo: Van den Bergen [6] apresentou um método GJK estendido para computar a profundidade de penetração de um par de objetos intersectados usando 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 da 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 (Figura 4.5).

Um *plano de separação* (PS) é um plano localizado entre dois objetos convexos, separando-os. Matematicamente, um PS é definido por H(v,δ), onde v é chamado de *eixo de separação* (ES) tal que $v\dot{a} \geq v\dot{b}$ para todo $a \in vert(A)$ e $b \in vert(B)$ como descrito por Chung e Wang [9].

Estes métodos tipicamente são usados em simulação de objetos rígidos, sendo ina-

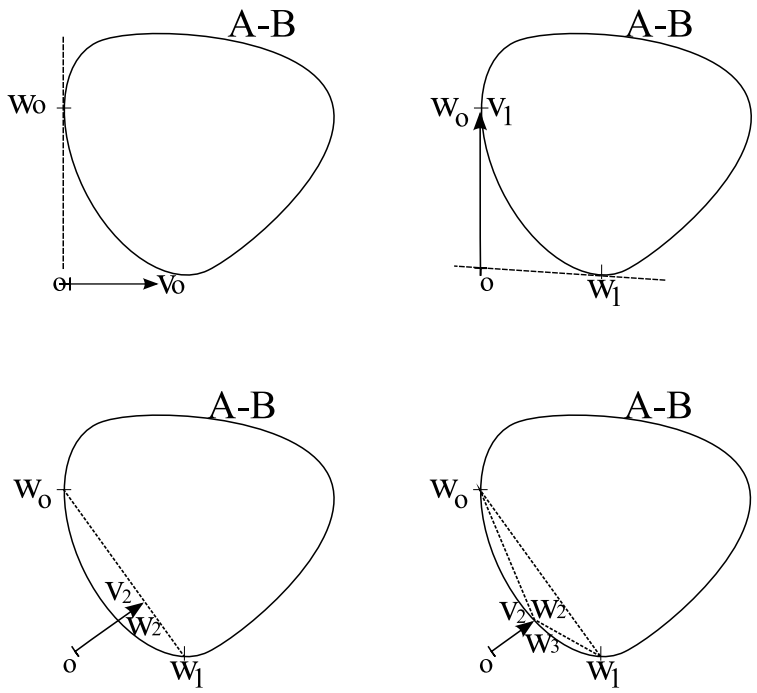


Figura 4.4: quatro iterações do algoritmo de GJK.

dequado sua utilização em sistemas de objetos deformáveis.

Por outro lado, Heidelberger et al. [18] propuseram um método para objetos deformáveis baseado em malhas tetraedrais, que permite encontrar a profundidade para todos os pontos interpenetrados (Figura 4.6).

Algoritmo baseado em malhas tetraedrais: este método computa a profundidade e a direção de penetração para cada vértice colidido fazendo uso das malhas tetraedrais. Tal profundidade pode ser usada para computar forças de penalidade que forneçam respostas realistas às colisões.

A idéia é computar uma profundidade de penetração consistente, onde os vetores de profundidade de penetração para os pontos próximos da superfície de penetração mudam suavemente, enquanto que para os pontos com penetrações profundas eles mudam de forma mais significativa (Figura 4.6). Para tanto, primeiro é computada a profundidade de penetração para os vértices próximos da superfície, após, esta informação é propagada para os vértices mais internos.

Finalmente, um triângulo de contato é calculado para cada vértice colidido. Tal triângulo de contato é uma face na superfície do objeto penetrado que intersecta com o vetor de direção de profundidade de penetração.

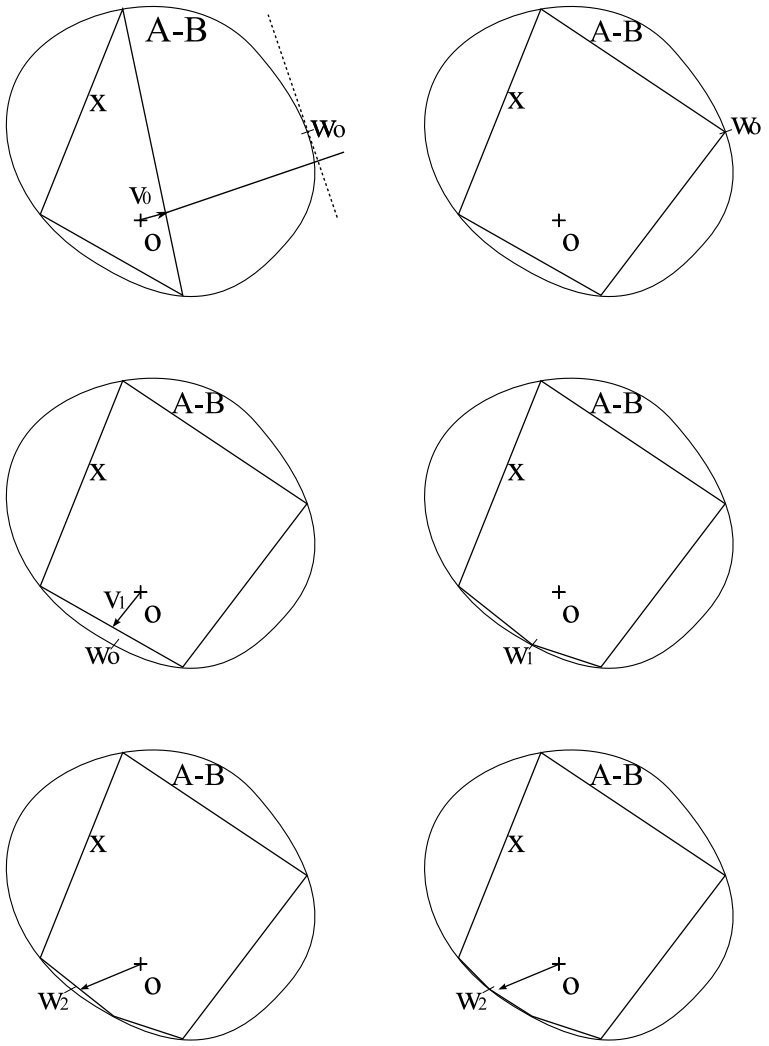
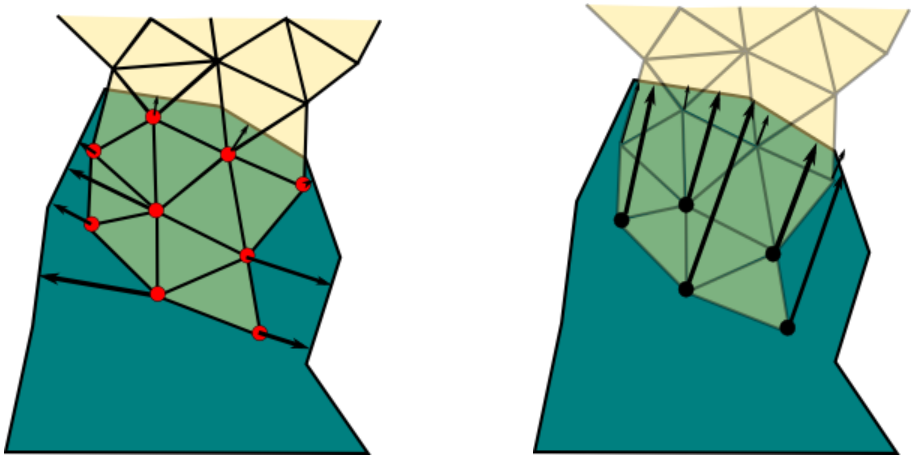


Figura 4.5: uma seqüência de iterações do algoritmo para computar a profundidade de penetração. A seta v_k denota um ponto na superfície do politopo mais próximo da origem, w_k é um vértice de suporte e as linhas pontilhadas representam o plano de separação $H(v_k, -v_k \cdot w_k)$.

4.2 Região de Deformação

Após o cálculo da profundidade de penetração, os objetos devem ser separados, a fim de fornecer um estado fisicamente correto. Para tal são usados os vértices colididos, embora vértices não colididos devam intervir no processo de separação (Figura 4.7(a)). Devido à necessidade de se encontrar um equilíbrio de forças entre os objetos interpenetrados, a força interna de uma área da superfície do objeto A deve ser igual em magnitude à força interna da área da superfície do objeto B.

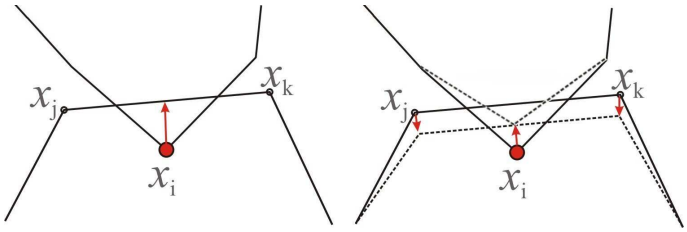
Assim, para alcançar um equilíbrio de forças é escolhida uma *região de deformação* (Figura 4.7(b)), que considera a união dos vértices colididos e os vértices dos triângulos



(a) (b)

Figura 4.6: em lugar de computar estritamente distâncias mínimas (a), computa-se distâncias consistentes (b) de profundidade de penetração.

de contato, que não necessariamente colidem (x_i, x_j e x_k na Figura 4.7 (b), colisões com essa configuração são chamadas de *colisões assimétricas*).



(a) (b)

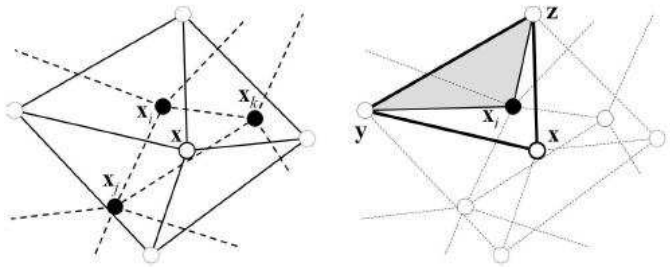
Figura 4.7: (a) se somente o vértice em colisão x for considerado no cômputo da superfície de contato, então o objeto B não é afetado e o equilíbrio da força não pode ser alcançado. (b) A região de deformação, consistente de x, x_i, x_j e x_k, permite uma reação simétrica à colisão, o equilíbrio de força pode ser alcançado.

Spillman et al. [39] apresentaram um método para computar um vetor de deslocamento s para cada vértice da região de deformação, usando a equação:

$$s = \frac{\sum_i w_i d_i + d}{\sum_i w_i + 1},$$

onde d_i é a profundidade de penetração dos vértices que têm seus triângulos de contato no conjunto de faces incidentes ao vértice de deslocamento e d é a profundidade de penetração do vértice de deslocamento. Se o vértice de deslocamento não é colidido, d é 0 e w_i representa o peso baricêntrico em relação aos vértices de deslocamento (Figura 4.8).

Este método apresenta um inconveniente devido a que os vértices não colididos não



(a) (b)

Figura 4.8: (a) o vetor de deslocamento de um vértice x é a soma dos pesos da profundidade de penetração dos vértices x_i, x_j e x_k, que por sua vez têm seus triângulos de contato incidentes em x. (b) O peso baricêntrico w_i do vértice x_i em relação a x é $w_i = \frac{A(x_i,y,z)}{A(x,y,z)}$. A (x,y,z) é a área do triângulo de contato de x_i, Spillman et al. [39].

possuem um triângulo (face) de contato, o que dificulta o cálculo do vetor de deslocamento.

Por outro lado, Jakobsen [19] apresentou um método para tratar colisões por projeção, isto é, os vértices colididos são projetados para fora do obstáculo, movendo-os até que fiquem livres de intersecção. Este método é adequado para tratar as colisões assimétricas descritas anteriormente. Por exemplo, na Figura 4.7, o vetor de deslocamento de x_i é sua profundidade de penetração, e os vetores de deslocamento dos vértices não colididos são calculados da seguinte forma: seja x_i o vértice colidido, e x_i' sua projeção na aresta de contato (x_j e x_k), aquele pode ser expresso como uma combinação linear $x_i = \alpha_1 x_j + \alpha_2 x_k$, tal que $\alpha_1 + \alpha_2 = 1$. Os valores α representam a proporção de deslocamento dos vértices da aresta de contato, assim, os vetores de deslocamento x_j e x_k são:

$$s_j = \frac{\alpha_1}{\alpha_1^2 + \alpha_2^2}(x_i' - x_i),$$
$$s_k = \frac{\alpha_2}{\alpha_1^2 + \alpha_2^2}(x_i' - x_i),$$

onde s_j e s_k são os vetores de deslocamento de x_j e x_k, respectivamente.

4.2.1 Busca Binária

Após ter computado os vetores de deslocamento, é estimada uma superfície de contato implícita que permite a separação dos objetos. Este deslocamento deve ser algo em torno da metade do comprimento dos vetores de deslocamento, para tal pode ser usado um esquema de busca binária [39]:

$$x^{i+1} \leftarrow x^i \pm \frac{1}{2^{(i+2)}} s, \tag{4.5}$$

onde s é o vetor de deslocamento, x é a posição do vértice de deslocamento original. Em outras palavras, para cada iteração, o intervalo de deslocamento é dividido por dois.

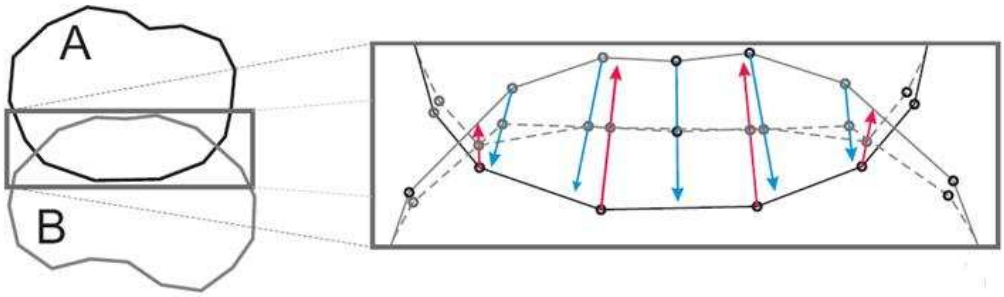


Figura 4.9: a primeira iteração da busca binária: os vértices na região de deformação são deslocados a metade do comprimento de seu vetor de deslocamento (entre a antiga posição e a superfície do outro objeto). Portanto, uma superfície de contato resulta exatamente no meio da intersecção. Isto corresponde à superfície de contato de dois objetos de elasticidade igual. Note-se que os vértices não colididos, adjacentes a vértices colididos, também são deslocados.

A busca binária converge rapidamente. Experimentos indicam que quatro iterações fornecem deslocamento suficiente para que o equilíbrio da força seja alcançado.

Capítulo 5

Sistema proposto

Neste capítulo descrevemos um sistema para animação de objetos deformáveis. O sistema manipula a detecção e a resposta às colisões usando técnicas descritas nos capítulos 3 e 4. Em particular, é usada uma abordagem de volumes limitantes para diminuir o número de colisões potenciais e uma abordagem baseada em uma *Hashing* espacial encontra as colisões reais. A resposta às colisões começa com o cômputo da profundidade de penetração dos vértices colididos, seguido pelo cálculo de uma região de deformação formada pelos vértices envolvidos na colisão. Finalmente, a separação dos objetos é efetuada usando Busca Binária.

A animação de objetos deformáveis usa a técnica de casamento de formas descrita no capítulo 2, na qual não é necessário o uso de malhas, embora estas sejam usadas no processamento de colisões.

O protótipo destinado a servir como prova de conceito para o sistema foi desenvolvido em uma estação de trabalho com ambiente Fedora Linux utilizando a linguagem C++, OpenGL e a biblioteca Glut. Uma observação importante é que o uso do Vertex Buffer Object (VBO), extensão do OpenGL, foi crucial para a obtenção de uma boa taxa de renderização dos objetos.

O protótipo trabalha com objetos representados por malhas tetraedrais. Portanto, foi necessário criar versões tetraedrais de malhas poliedrais e estruturas de dados adequadas para o uso no protótipo desenvolvido.

O objetivo é lidar com objetos deformáveis em tempo real, obtendo uma simulação dinâmica estável, com comportamento físico plausível. Ressalve-se, entretanto, que as propriedades do material não foram modeladas neste protótipo.

5.1 Estruturas de dados

Cada objeto que compõe o sistema possui um volume limitante (esfera), tetraedros, faces na superfície e vértices. Cada vértice é considerado como uma partícula, possuindo velocidade, massa, coordenadas no espaço e listas de vértices, arestas, faces (apenas para os vértices da superfície) e tetraedros nele incidentes.

Os vértices são usados em todos os estágios da animação, portanto, eles são armazenados num vetor de elementos consecutivos, permitindo acesso aleatório em tempo constante. As arestas possuem apenas os índices dos dois vértices incidentes; as faces, de forma similar, possuem os índices dos três vértices incidentes e os tetraedros os índices dos quatro vértices que os compõem. Os volumes limitantes das faces e os tetraedros não são armazenados, embora estes sejam sempre usados, isto porque trabalhamos com objetos deformáveis, assim, eles são computados sempre que necessário.

Por outro lado, a animação dos objetos precisa das posição iniciais dos vértices, portanto, é necessário guardar estas posições.

Assim temos as seguintes classes abstratas:

```
struct Vertex{
  Point pos;            // posição
  Point p0;             // posição original
  Vector vel;           // velocidade
  Scalar mass;          // massa
  int timestamp;        // flag
  list incidentVertex; //lista de vértices incidentes
  list incidentCells   //lista de tetraedros incidentes
};

struct SrfVertex : public Vertex {
  Vector normal;        // normal à superfície
  list incidentFaces;  // lista de faces incidentes
};

struct Face {
  int vertices[3];      // índices dos vértices da face
};

struct Tetrahedron {
  int vertices[4];      // índices dos vértices do tetraedro
};

struct Body {
  Vertices vtx;         // vértices do objeto
```

```
  Point     cm;          // centro de massa do objeto

  Body();
  ComputeQ();
  ComputeP();
  ComputeGoal();
  ApplyForces();
  Integrate(double dt);
};

struct World {
  Vertices vtx;          // todos os vértices (de todos os objetos)
  Faces    faces;        // triângulos das superfícies dos objetos
  Tetras   tetras;       // tetraedros dos objetos

  World();
  InitializeScene();     // inicializa os objetos da cena
  InitializeHash();      // inicializa a tabela hash
  BroadCollision();      // detecção de colisão grosseira

  /* atualiza a tabela hash nas regiões de colisão */
  UpdateHashOnCollisions();

  NarrowCollision();     // detecção de colisões exata

  /* calcula a profundidade de penetração dos vértices colididos */
  ComputePenetrationDepth();

  /* separa as regiões em colisão (resposta às colisões) */
  SeparateDeformRegions();

  /* usa casamento de formas para restaurar a forma do objeto */
  ShapeMatching();

  AtualizaVBO();         // atualiza o VBO para renderizar os objetos

};
```

Outra estrutura de dados importante é a tabela *hash*, que armazena os dados avaliados na colisão, para evitar a atualização de toda a tabela durante a simulação, se usa um *flag time-stamp* para cada vértice, face, tetraedro, e célula.

5.2 Arquitetura geral

Em linhas gerais, é usado um algoritmo clássico (Algoritmo 1) para animação física.

Algoritmo 1 Algoritmo Geral

Requer: todos os objetos

1: $precomputarParametros()$
2: **repete**
3: $aplicarForcas()$
4: /*Detecção de Colisões*/
5: $deteccaoDeColisoes()$
6: /*Resposta às Colisões*/
7: $computarProfundidadePenetracao()$
8: $computarRegiaoDeformacao()$
9: $buscaBinaria()$
10: /*Animação*/
11: $casamentoDeFormas()$
12: $integracao()$
13: $t \leftarrow t + \Delta t$
14: **até** $parada$

As estruturas de dados são iniciadas e são calculados os parâmetros iniciais da tabela *hash* (veja Figura 5.1):

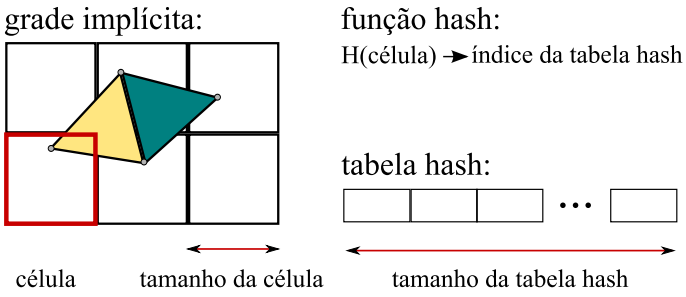


Figura 5.1: parâmetros da tabela *hash*.

tamanho da tabela: tipicamente é um número primo grande. Influi no desempenho do algoritmo uma vez que tabelas maiores reduzem o risco de inserir posições diferentes no mesmo índice, reduzindo o número de colisões. Em compensação, implica no aumento do uso de memória. O tamanho ótimo tem relação direta com o número de primitivas a serem avaliadas;

tamanho da célula: influi diretamente no número de primitivas que intersectam a célula da grade, pois células maiores contêm maior número de primitivas, incrementando a possibilidade de intersecção. Por outro lado, células menores contêm primitivas em maior número de índices, incrementando o número de células a serem testadas para intersecção. Heidelberger et al. [18] e Steinemann [40] sugerem o uso da média do

comprimento das arestas dos tetraedros ou a média do comprimento dos volumes limitantes dos tetraedros, a primeira apresenta melhores resultados e é usada neste sistema;

função *"hash"*: serve para encontrar um índice que distribua as células arbitrariamente na tabela *hash*:

$$h = hash(i, j, k) = H(i, j, k) ::= (x \cdot \alpha \oplus y \cdot \beta \oplus z \cdot \gamma) \,\%\, n, \tag{5.1}$$

onde i, j, k são coordenadas do vértice nas coordenadas da grade, α, β e γ são números primos grandes e n é o tamanho da tabela *hash*.

A primeira etapa do laço da animação consiste em aplicar forças externas (gravidade, vento, etc.) em cada partícula do objeto. O algoritmo usado na aplicação de forças é o Algoritmo 2.

Algoritmo 2 AplicarForcas

Requer: vértices dos objetos, gravidade, outras forças externas
Saida: vértices com posições e velocidades atualizadas

1: $Objs \leftarrow <$ todos os objetos $>$
2: **para todo** $O \in Objs$ **fazer**
3: **para todo** $v \in O$ **fazer**
4: $v_f \leftarrow \frac{(gravidade + fext) \cdot \Delta t}{v_m}$
5: **fim para**
6: **fim para**

As etapas seguintes compreendem o tratamento de colisões, a animação e a integração do sistema, conforme descrito nas seções a seguir.

5.3 Detecção de colisões

O objetivo da detecção de colisões não é apenas saber quais objetos colidem, mas também os pontos exatos de colisão, de forma a permitir computar uma resposta realista à colisão.

À diferença do método *Hashing* espacial descrito na seção 3.2.3, o método proposto neste trabalho não mapeia todo o espaço do universo de simulação na tabela *hash*, mas apenas as regiões onde existem colisões potenciais. Para evitar atualizar desnecessariamente a tabela, é usada uma fase de filtragem grosseira (em inglês, *broad phase*), que trata colisões entre os volumes limitantes de objetos, retornando regiões em colisão potencial. Subsequentemente, a fase de filtragem exata (em inglês, *narrow phase*) encontra as colisões reais nessas regiões usando *Hashing* espacial.

5.3.1 Filtragem grosseira: esferas limitantes

O objetivo dessa fase é descartar regiões onde não há colisões. Para tanto, é efetuado um teste simples de colisão entre cada par de objetos, considerando apenas seus volumes limitantes. O volume limitante usado é a esfera, por permitir testes bastante rápidos como discutido na subseção 3.1.1.

Algoritmo 3 Filtragem grosseira

Requer: lista dos objetos

1: $Objs \leftarrow <$ todos os objetos $>$
2: **para todo** (O_i, O_j) tais que $O_i, O_j \in Objs$ e $i \neq j$ **fazer**
3: $c_i \leftarrow O_i.center$
4: $c_j \leftarrow O_j.center$
5: $r_i \leftarrow O_i.radius$
6: $r_j \leftarrow O_j.radius$
7: $\vec{d} \leftarrow c_i - c_j$
8: $d2 \leftarrow d \cdot d$
9: $r\_soma \leftarrow r_i + r_j$
10: **se** $d2 < r\_soma \cdot r\_soma$ **então**
11: AtualizaHash(O_i, c_j, r_j)
12: AtualizaHash(O_j, c_i, r_i)
13: **fim se**
14: **fim para**

O Algoritmo 3 funciona da seguinte forma: um par de objetos A e B colide apenas se a distância entre os centros das esferas é menor que a soma de seus raios. Assim, para cada par de objetos, são identificados os vértices envolvidos na colisão potencial ($v \subset A \cap B$), usando apenas consultas para verificar se um vértice v do objeto A está dentro da esfera do objeto B e vice-versa.

5.3.2 Filtragem exata: *Hashing* espacial

Após ter encontrado as regiões de colisão potencial, encontramos as colisões reais, isto é, encontrarmos os vértices que realmente colidem. Para tal, usamos *Hashing* espacial, que subdivide implicitamente o espaço do mundo numa grade uniforme de células regulares, armazenando-as num endereço da tabela de espalhamento obtido usando a função "*hash*". Entretanto, a tabela é atualizada somente nas células que contêm vértices das regiões com colisão potencial. Também são mapeados os tetraedros e as faces incidentes nesses vértices e inseridos na tabela. Para evitar inserções duplicadas, é armazenado em cada célula um inteiro que identifica em qual quadro da animação esta foi alterada. Este

identificador é conhecido como *timestamp*. Assim, cada célula tem uma lista de tetraedros e faces que a intersectam e vice-versa. As células são acessadas por um índice inteiro como mostra a Figura 5.2.

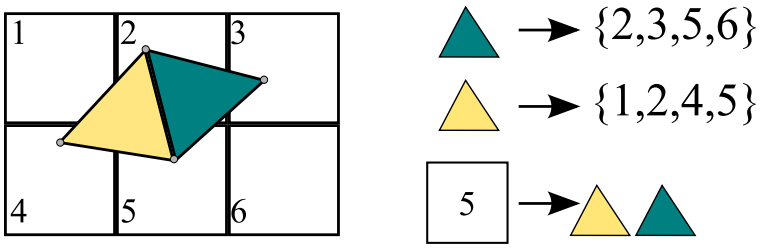


Figura 5.2: a célula 5 contém as faces verde e amarela, do mesmo modo, estas faces contêm as células $2, 3, 5, 6$ e $1, 2, 4, 5,$ respectivamente.

Logo, para cada entrada da tabela é executado um teste de intersecção entre vértices e tetraedros contidos nessa célula, encontrando assim os vértices colididos.

Algoritmo 4 Filtragem exata

Requer: vértices em colisão potencial
Saida: vértices colididos

```
1:  Vp ←< vértices em potencial colisão>
2:  Vc ←<> /*vértices colididos*/
3:  para todo v em Vp fazer
4:      celula ← tabelaHash[v.celula_id]
5:      para todo t em celula fazer
6:          /*t é um tetraedro cruzando a célula da grade*/
7:          para todo v_v em celula fazer
8:              /*v_v é um vértice da célula*/
9:              se t.contem(v_v) então
10:                 Vc.insere(v_v)
11:             fim se
12:         fim para
13:     fim para
14: fim para
```

O algoritmo de filtragem exata (Algoritmo 4) processa todos os vértices em regiões de colisão potencial, sendo que cada vértice guarda o id da célula a que pertence. Para cada vértice v é realizado um teste de intersecção com cada tetraedro t da célula. Para acelerar o processo, avalia-se primeiro se o vértice está inserido no *AABB* do tetraedro ($v \subset t.AABB()$). Caso haja intersecção, o vértice é marcado como vértice colidido. Este processo também detecta auto-intersecção, como mostra a Figura 5.3. O teste pode ser feito usando coordenadas baricêntricas.

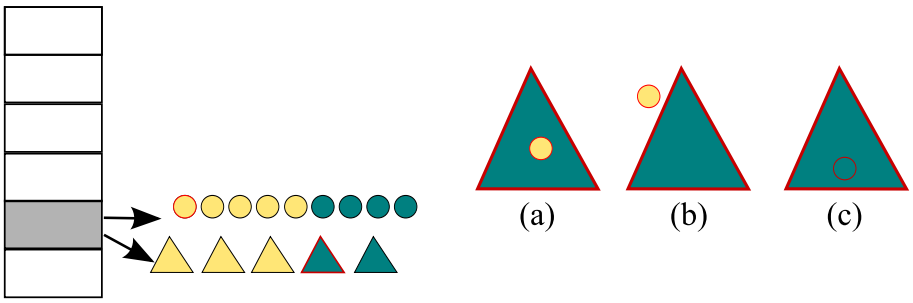


Figura 5.3: para cada entrada não vazia da tabela *hash*, testa-se intersecção entre vértices e tetraedros, para verificar se existe: (a) colisão, (b) não colisão ou (c) auto-colisão.

Neste teste de coordenadas baricêntricas: v é expresso $\beta = (\beta_1, \beta_2, \beta_3)^T$ com relação a um sistema de coordenadas cuja origem coincide com x_0, um dos vértices do tetraedro, e cujos eixos coincidem com as arestas de t adjacentes a x_0 : $v = x_0 + A\beta$, onde $A = [x_1 - x_0, x_2 - x_0, x_3 - x_0,]$ é uma matriz de dimensão 3×3. As coordenadas β de v nesta nova coordenada *frame* são: $\beta = A^{-1}(p - x_0)$. Agora, o vértice v está localizado dentro do tetraedro t, se $\beta_1 >= 0, \beta_2 >= 0, \beta_3 >= 0$ e $\beta_1 + \beta_2 + \beta_3 <= 1$ (Figura 5.4).

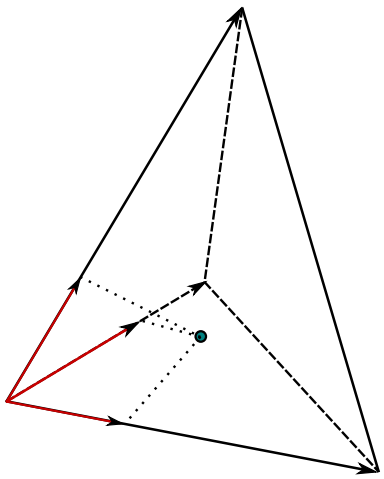


Figura 5.4: teste de colisão vértice-tetraedro baseado em coordenadas baricêntricas.

A seguir são apresentados detalhes sobre como o mapeamento dos objetos na tabela *hash* é realizado e como se procede à atualização desta.

5.3.2.1 Mapeamento dos objetos

Todos os elementos da simulação são mapeados numa grade uniforme 3D composta por células e as células são inseridas numa tabela *hash*, sendo que a atualização da tabela é realizada apenas nas primitivas das regiões de colisão potencial. Por exemplo, o algoritmo 5 atualiza a região do objeto A (B) que esta dentro da esfera envolvente do objeto B (A).

Algoritmo 5 AtualizaHash

Requer: Objeto A, centro(c) e raio(r) da esfera envolvento do objeto B, tamanho da célula da grade l

Saida: tabela hash atualizada nas regiões de colisão potencial

1: $r^2 \leftarrow r * r$
2: **para todo** v em O_j **fazer**
3: $\vec{d} \leftarrow v - c$
4: **se** $v.timestamp \neq timestamp$ **e** $\vec{d} \cdot \vec{d} < r^2$ **então**
5: $v.timestamp \leftarrow timestamp$
6: $(i, j, k) \leftarrow (\lfloor v.x/l \rfloor, \lfloor v.y/l \rfloor, \lfloor v.z/l \rfloor)$
7: $h \leftarrow hash(i, j, k)$
8: **se** $tabelaHash[h].timestamp \neq timestamp$ **então**
9: $tabelaHash[h].apagaconteudo()$
10: $tabelaHash[h].timestamp \leftarrow timestamp$
11: **fim se**
12: $v.timestamp \leftarrow timestamp$
13: $tabelaHash[h].insere(v)$
14: $V_p.insere(v)$
15: $atualizaVizinhanca(v)$
16: **fim se**
17: **fim para**

O algoritmo de mapeamento (Algoritmo 5) mostra a inserção das primitivas da região de colisão potencial nas células da grade correspondentes. O algoritmo trata os seguintes elementos:

vértices: cada vértice é armazenado na célula correspondente da forma:

$$(i, j, k) : i = \lfloor x/l \rfloor, \lfloor y/l \rfloor, \lfloor z/l \rfloor.$$

Logo, uma função *hash* insere a célula (i, j, k) num índice $h = hash(i, j, k)$ da tabela *hash* como mostra a Figura 5.5;

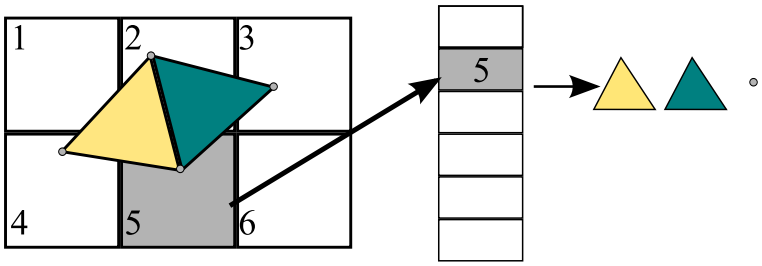


Figura 5.5: exemplo de mapeamento de primitivas de objetos. As faces amarela e verde são armazenadas na célula 5. Por outra parte, a célula 5 é mapeada num índice arbitrário da tabela *hash*.

Algoritmo 6 AtualizaVizinhanca

Requer: vértices em potencial colisão
Requer: tamanho da célula l
Saida: tabela hash da região em potencial colisão atualizada

```
1:  S ←<Vértices na superfície>
2:  para todo v ∈ V_p fazer
3:      /*mapea tetraedros incidentes em v*/
4:      T_inc ← v.tetraedrosIncidentes()
5:      para todo t ∈ T_inc fazer
6:          se t.timestamp ≠ timestamp então
7:              t.timestamp = timestamp
8:              bb ← t.caixaLimitante()
9:              min ← ⌊bb.min/l⌋
10:             max ← ⌊bb.max/l⌋
11:             para z ← min.z até max.z fazer
12:                 para y ← min.y até max.y fazer
13:                     para x ← min.x até max.x fazer
14:                         h ← hash(x, y, z)
15:                         tabelaHash[h].insere(t)
16:                     fim para
17:                 fim para
18:             fim para
19:         fim se
20:     fim para
21:     /*mapea faces incidentes em v*/
22:     se v ∈ S então
23:         F_inc ← v.facesIncidentes()
24:         para todo f ∈ F_inc fazer
25:             se f.timestamp ≠ timestamp então
26:                 f.timestamp = timestamp
27:                 bb ← f.caixaLimitante()
28:                 min ← ⌊bb.min⌋
29:                 max ← ⌊bb.max⌋
30:                 para z ← min.z até max.z fazer
31:                     para y ← min.y até max.y fazer
32:                         para x ← min.x até max.x fazer
33:                             h ← hash(x, y, z)
34:                             tabelaHash[h].insere(f)
35:                         fim para
36:                     fim para
37:                 fim para
38:             fim se
39:         fim para
40:     fim se
41: fim para
```

tetraedros: para simplificar o armazenamento do tetraedro nas suas células correspondentes é utilizado o *AABB* do tetraedro. Assim, os vértices mínimo e máximo que descrevem o *AABB* representam o intervalo das células que intersectam o *AABB*. Finalmente, todos os índices h encontrados são inseridos na tabela *hash*, como mostra a Figura 5.5;

faces: para mapear as faces, segue-se o mesmo esquema usado para inserir tetraedros, usando as *AABB* das faces;

arestas: para armazenar somente as células cruzadas por arestas de intersecção, foi adaptada uma técnica descrita por Amanatides e Woo [2]. O algoritmo básico é atravesar, com um raio, as células de uma partição do espaço 3D. Ir de uma célula a sua vizinha exige somente duas comparações de ponto flutuante. Também são eliminadas intersecções múltiplas do raio com objetos que estáo em mais de uma célula.

A atualização da tabela *hash* consiste em associar um rótulo global (*time-stamp*) a cada intervalo de tempo t. Assim, se uma primitiva é inserida numa célula da tabela *hash*, e o time-stamp da célula estiver desatualizado, o conteúdo da célula é descartado e a primitiva transforma-se no único elemento. Pode-se dizer que o conteúdo da célula num outro intervalo de tempo é descartado. Sua vantagem é sua simplicidade e seu uso econômico de memória.

5.4 Resposta às colisões

O processo de detecção de colisão, descrito na seção anterior, fornece uma lista de vértices colididos. Estes são usados na resposta às colisões. O processo começa com o cálculo da profundidade de penetração dos vértices colididos, seguido do cômputo da região de deformação, a qual contém todos os vértices envolvidos na colisão. Finalmente, os objetos são separados usando uma técnica de Busca Binária.

5.4.1 Profundidade de penetração

O método para encontrar a profundidade de penetração dos objetos colididos é similar ao proposto por Heidelberger et al. [18], visto na seção 4.1. A idéia é classificar os vértices colididos em relação a sua profundidade de penetração. Assim, primeiro são avaliados os vértices mais próximos à superfície. Essa informação é então propagada aos vértices que possuem maior penetração, até que todos os vértices colididos sejam processados.

No final, cada vértice colidido possui uma profundidade de penetração d e um vetor de direção de penetração $\vec{r}$.

Classificação: se um vértice colidido tem um ou mais vértices incidentes não colididos este é um *vértice da borda*, caso contrário é um *vértice interno* (Figura 5.6).

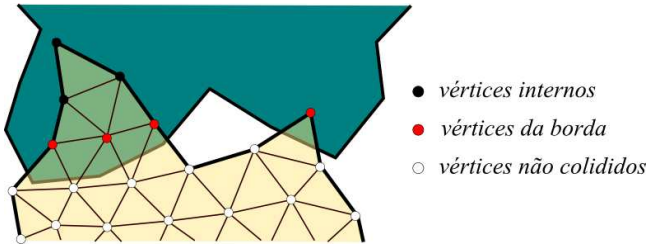


Figura 5.6: vértices colididos: da borda ou internos.

Profundidade de penetração dos vértices da borda: computada pelo algoritmo 7. Este funciona da seguinte forma: primeiro são identificadas as *arestas de intersecção*, ou seja, arestas que possuem um vértice da borda e um vértice não colidido. A seguir, procura-se a face da superfície mais próxima que intersecta a aresta. O teste pode ser feito usando coordenadas baricêntricas, já que permitem computar a normal da face intersectada n_{srf} e obter o ponto exato de intersecção p_{int}. Estes dados são armazenados na aresta e servem para encontrar a profundidade de penetração dos vértices da borda (Figura 5.7). Para cada vértice da borda v, é computada sua profundidade de penetração $d(v)$ e seu

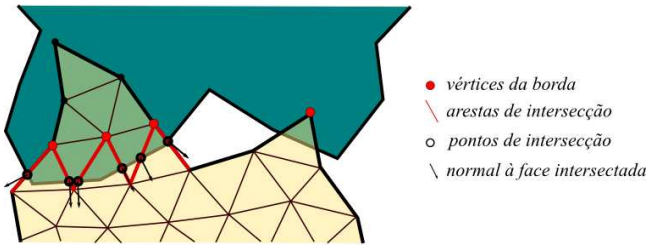


Figura 5.7: vértices da borda, arestas de intersecção, pontos exatos de intersecção e normais às faces intersectadas.

vetor de direção de penetração $\vec{r}(v)$ da seguinte forma: para cada aresta de intersecção incidente em v, computa-se um peso ponderado w entre o ponto de intersecção da aresta p_{int} e v:

$$w(p_{int}, v) = \frac{1}{\|p_{int} - v\|^2}, \tag{5.2}$$

Algoritmo 7 Profundidade de Penetração

Requer: vértices colididos
Saida: profundidade de penetração dos vértices da borda, vértices processados

1: $Vc \leftarrow <$ Vértices colididos$>$
2: $vtx\_processados \leftarrow <>$
3: **para todo** $v \in Vc$ **fazer**
4: $d_{sum} \leftarrow 0$
5: $w_{sum} \leftarrow 0$
6: $\vec{r}_{sum} \leftarrow \vec{0}$
7: /*classifica v: v_b se é de borda, v_i se é interno e v_n se não colide*/
8: **para todo** $v_{inc} \in v.incident()$ **fazer**
9: **se** $!v_{inc}.colide()$ **então**
10: $v.processado(true)$
11: $e \leftarrow Aresta(v_{inc}, v)$
12: $p_{int}, n_{srf} \leftarrow intersectaSuperficieAresta(e)$
13: $\vec{v}_{tmp} \leftarrow p_{int} - v$
14: $d^2 \leftarrow \vec{v}_{tmp} \cdot \vec{v}_{tmp}$
15: $w \leftarrow 1/d^2$
16: $w_{sum} \leftarrow w_{sum} + w$
17: $d_{sum} \leftarrow d_{sum} + w \cdot v_{tmp} \cdot n_{srf}$
18: $\vec{r}_{sum} \leftarrow \vec{r}_{sum} + n_{srf} \cdot w$
19: **fim se**
20: **fim para**
21: **se** $v.processado()$ **então**
22: $vtx\_processados.insere(v)$
23: **fim se**
24: $v.profundidadePenetracao \leftarrow d_{sum}/w_{sum}$
25: $v.direcaoPenetracao \leftarrow \vec{r}_{sum}.normalizado()$
26: **fim para**

após o que são computados $d(v)$ e $\vec{r}(v)$ através das fórmulas:

$$d(v) = \frac{\sum_{i=1}^{k} (w(p_i, v) \cdot (p_i - v) \cdot n_i)}{\sum_{i=1}^{k} (p_i, v)}, \tag{5.3}$$

$$\vec{r}(v) = \frac{\sum_{i=1}^{k} (w(p_i, v) \cdot n_i)}{\sum_{i=1}^{k} (p_i, v)}, \tag{5.4}$$

onde k é o número de arestas de intersecção incidentes em v, p_i e n_i representam o i-ésimo p_{int} e n_{srf} a serem avaliados. A Figura 5.8 ilustra a profundidade de penetração para os vértices da borda v. No final, todos os vértices da borda são inseridos numa lista de vértices processados.

Profundidade de penetração dos vértices internos: após o processamento de todos os vértices da borda, a informação da profundidade de penetração obtida é propagada para

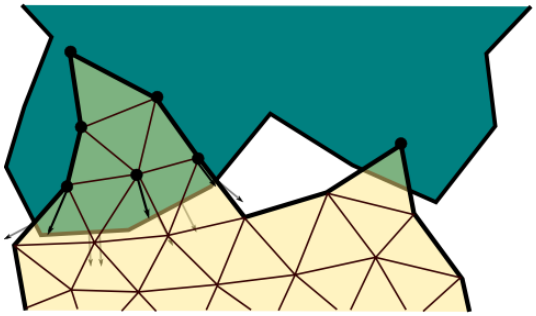


Figura 5.8: profundidade de penetração de vértices da borda.

os vértices internos v_i (Figura 5.9).

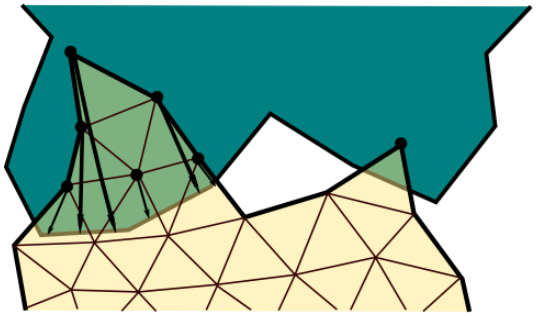


Figura 5.9: profundidade de penetração de vértices internos usando propagação.

Algoritmo 8 Propaga vértices

Requer: vértices processados
Saida: vértices a processar

```
1: para todo v in vtx_processados fazer
2:     para todo v_inc ∈ v.incident() fazer
3:         se v_inc.colidido() e !v_inc.processado() então
4:             vtx_a_processar.insere(v_inc)
5:         fim se
6:     fim para
7: fim para
```

O Algoritmo 8 fornece uma lista de vértices a processar, isto é, dada uma lista de vértices processados, é obtida uma lista de vértices a processar. Desta forma, vê-se que o cômputo da profundidade de penetração é feito por níveis, até que todos os vértices internos sejam computados.

No Algoritmo 9 computa-se a profundidade de penetração $d(v)$ e a direção de penetração $\vec{r}(v)$ dos vértices internos v, enquanto existam vértices internos a serem processados. Primeiro, para cada vértice processado v_{inc} incidente no vértice v, computa-se

Algoritmo 9 Propaga Profundidade de Penetração

Requer: vértices a processar
Saida: profundidade de penetração dos vértices internos

1: /*vértices internos adjacentes a vtx_processados*/
2: $vtx\_a\_processar \leftarrow propagaVertice(vtx\_processados)$
3: **enquanto** $!vtx\_a\_processar.vazio()$ **fazer**
4: /*enquanto há vértices a processar*/
5: **para todo** $v \in vtx\_a\_processar$ **fazer**
6: $d_{sum} \leftarrow 0$
7: $w_{sum} \leftarrow 0$
8: $\vec{r}_{sum} \leftarrow \vec{0}$
9: **para todo** $v_{inc} \in v.incident()$ **fazer**
10: **se** $v_{inc}.processado()$ **então**
11: $\vec{v}_{tmp} \leftarrow v_{inc} - v$
12: $d^2 \leftarrow \vec{v}_{tmp} \cdot \vec{v}_{tmp}$
13: $w \leftarrow 1/d^2$
14: $\vec{r} \leftarrow v_{inc}.direcaoPenetracao$
15: $d \leftarrow v_{inc}.profundidadePenetracao$
16: $d_{sum} \leftarrow d_{sum} + (\vec{v}_{tmp} \cdot \vec{r}) \cdot w + d$
17: $w_{sum} \leftarrow w_{sum} + w$
18: $\vec{r}_{sum} \leftarrow \vec{r}_{sum} + \vec{r} \cdot w$
19: **fim se**
20: $v.profundidadePenetracao \leftarrow d_{sum}/w_{sum}$
21: $v.direcaoPenetracao(\vec{r}_{sum}.normalizado())$
22: $v_{inc}.processado(true)$
23: **fim para**
24: **fim para**
25: $vtx\_processados \leftarrow vtx\_a\_processar$
26: $vtx\_a\_processar \leftarrow propagaVertice(vtx\_processados)$
27: **fim enquanto**

um peso ponderado μ:

$$\mu(v_{inc}, v) = \frac{1}{\|v_{inc} - v\|^2},$$

depois, computa-se $d(v)$ e $\vec{r}(v)$ segundo as fórmulas:

$$d(v) = \frac{\sum_{j=1}^{k} (\mu(v_j, v) \cdot ((v_j - v) \cdot r(v_j) + d(v_j)))}{\sum_{j=1}^{k} \mu(v_j, v)}, \tag{5.5}$$

$$\vec{r}(v_i) = \frac{\sum_{j=1}^{k} \mu_j r(v_j)}{\sum_{j=1}^{k} \mu_j}, \tag{5.6}$$

onde k é o número de vértices incidentes no vértice v, e v_j e o i-ésimo v_{inc} avaliado.

5.4.2 Separação

O processo de separação é precedido da determinação da região de deformação, computada como discutido na seção 4.2. A região de deformação contém todos os vértices envolvidos na colisão, isto é, os vértices colididos e seus triângulos de contato. Para todos os vértices da região de deformação é computado um vetor de deslocamento, que será usado no processo de separação. Nesta etapa também são resolvidas as colisões assimétricas, usando o método de projeção de Jakobsen [19](Figura 5.10). Finalmente, os objetos são separados usando uma técnica de Busca Binária (seção 4.2.1)(Figura 5.11), deslocando os vértices aproximadamente à metade do comprimento de seu vetor de deslocamento (Figura 5.12).

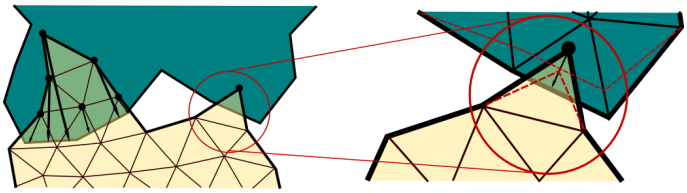


Figura 5.10: solução de colisões assimétricas.

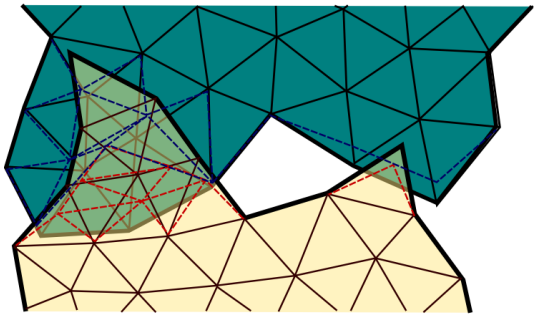


Figura 5.11: solução de busca binária.

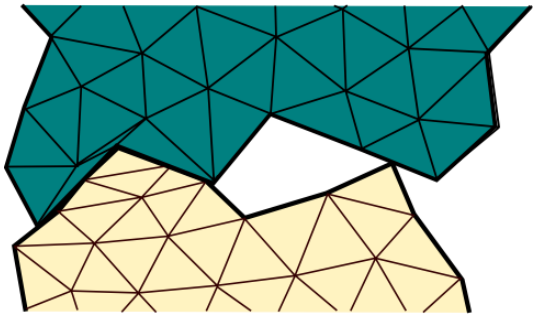


Figura 5.12: superfície de contato.

5.5 Animação

Na animação dos objetos é aplicado o método de casamento de formas usando mínimos quadrados, descrito na seção 2.4.2.2. Esta técnica permite encontrar uma correspondência entre os vértices no estado inicial x_i^0 e estado atual x_i.

Algoritmo 10 Cômputo de Q

Requer: vértices de controle do objeto
Saida: centro de massa inicial, posição relativa q e matriz A_{qq}

1: $m \leftarrow 0$
2: $x_{cm}^0 \leftarrow 0$
3: $S \leftarrow <$ vértices de controle do objeto$>$
4: **para todo** $v \in S$ **fazer**
5: $\quad m \leftarrow m + v.mass$
6: $\quad x^0 \leftarrow v.x^0$ /*posição original*/
7: $\quad x_{cm}^0 \leftarrow x_{cm}^0 + x^0$
8: **fim para**
9: $x_{cm}^0 \leftarrow x_{cm}^0/m$ /*centro de massa de x^0*/
10: **para todo** $v \in S$ **fazer**
11: $\quad x^0 \leftarrow v.x^0$ /*posicao original */
12: $\quad q \leftarrow x^0 - x_{cm}^0$
13: $\quad A_{qq} \leftarrow A_{qq} + q * q^T$
14: **fim para**

Algoritmo 11 Cômputo de P

Requer: vértices de controle do objeto
Saida: centro de massa, posição relativa p e matriz A_{pq}

1: $x_{cm} \leftarrow 0$
2: $S \leftarrow <$ vértices de controle do objeto$>$
3: **para todo** $v \in S$ **fazer**
4: $\quad x \leftarrow v.x$ /*posição atual*/
5: $\quad x_{cm} \leftarrow x_{cm} + x$
6: **fim para**
7: $x_{cm} \leftarrow x_{cm}/m_i$ /*posicao original de x*/
8: **para todo** $v \in S$ **fazer**
9: $\quad x \leftarrow v.x$ /*posicao atual */
10: $\quad p \leftarrow x - x_{cm}$
11: $\quad A_{pq} += p * q^T$
12: **fim para**
13: $R \leftarrow decomposicaoPolar(A_{pq})$

O objetivo é encontrar a melhor matriz de transformação afim, que permita efetuar deformações plausíveis e que permita que o objeto volte à sua forma original (Figura 5.13).

Algoritmo 12 Cômputo de G

Requer: todos os vértices do objeto, centro de massa inicial, centro de massa atual
Saida: posições alvo

1: $S \leftarrow <$ vértices do objeto $>$
2: **para todo** $v \in S$ **fazer**
3: $\quad v_g \leftarrow R * (v_{x^0} - x^0_{cm}) + x_{cm}$
4: **fim para**

Algoritmo 13 Integração

Requer: todos os vértices do objeto
Saida: novas posições

1: $S \leftarrow <$ vértices do objeto $>$
2: **para todo** $v \in S$ **fazer**
3: $\quad v_{vel} \leftarrow v_{vel} + \alpha \frac{v_g - v_x}{\Delta t} + v_f$
4: $\quad v_x \leftarrow v_x + v_{vel} \cdot \Delta t$
5: **fim para**

Considerando os pesos das partículas, uma transformação linear composta de uma translação t e uma rotação R sobre o ponto t_0 pode ser encontrada minimizando:

$$\sum_i w_i (R(x_i^0 - t_0) + t - x_i)^2.$$

Podemos estabelecer para o problema $w_i = m_i$, isto é, que a ponderação das partículas é representada por suas massas, e que os vetores de translação ótimos são o *centro de massa* da forma inicial e o centro de massa da forma atual. Assim, temos

$$t_0 = x^0_{cm} = \frac{\sum_i m_i x_i^0}{\sum_i m_i}, \quad e \quad t = x_{cm} = \frac{\sum_i m_i x_i}{\sum_i m_i}.$$

Para encontrar um R ótimo é necessário encontrar uma matriz de transformação linear A. Observe que A não necessariamente é orto-normal. Para encontrar A define-se as posições relativas

$$q_i = x_i^0 - x^0_{cm} \quad e \quad p_i = x_i - x_{cm}.$$

Estas posições definem o campo de deformação das partículas, tratando deformações elásticas de forma implícita.

Logo, a matriz de transformação linear ótima A é encontrada minimizando o termo:

$$\sum_i m_i (Aq_i - p_i)^2.$$

Para resolver esta equação, se usa a técnica de mínimos quadrados, decomposição polar de matrizes e diagonalização de matrizes Jacobi, descritas na seção 2.4.2.2. Portanto,

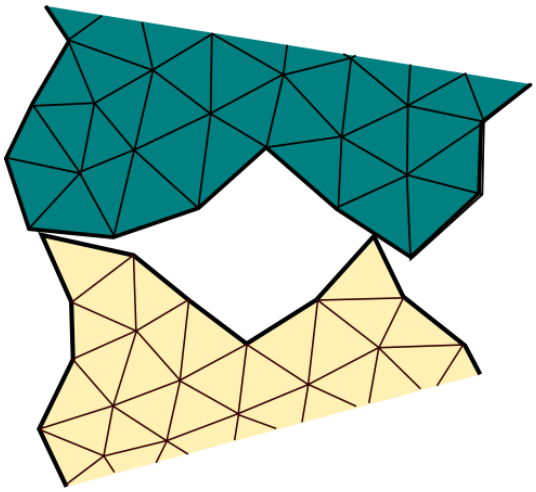


Figura 5.13: solução da deformação mostrada na figura 5.12.

a matriz de rotação ótima é:

$$R = A_{pq}(\sqrt{A_{pq}^T A_{pq}})^{-1}.$$

Finalmente, havendo encontrado a matriz de rotação é realizada a seguinte transformação linear para rotação e translação do objeto, a fim de encontrar a melhor posição objetivo de cada partícula:

$$g_i = R(x_i^0 - x_{cm}^0) + x_{cm}.$$

Os pontos são movidos para as posições g_i, exatamente a cada intervalo de tempo.

5.6 Método de integração

Nosso esquema deriva de um método de integração de Euler (2.5.4), que inclui uma parte explícita (a atualização da velocidade) e uma parte implícita (a atualização da posição).

Para computar as posições dos objetos, as acelerações e velocidades são integradas numericamente para cada intervalo de tempo utilizando as seguintes equações:

$$v_i(t + \Delta t) = v_i(t) + \alpha \frac{g_i(t) - x_i(t)}{\Delta t} + \frac{\Delta t}{m_i} f_{ext}(t), \tag{5.7}$$

$$x_i(t + \Delta t) = x_i(t) + \Delta t v_i(t + \Delta t), \tag{5.8}$$

onde $v_i(t + \Delta t)$ é a velocidade no próximo intervalo de tempo, $v_i(t)$ é a velocidade no intervalo de tempo atual, α é o parâmetro que simula rigidez, $g_i(t)$ é a melhor posição do vértice $x_i(t)$ (para resolver a deformação), Δt é a magnitude do intervalo de tempo, m_i é a massa da partícula e $f_{ext}(t)$ são as forças externas.

Capítulo 6

Resultados

Para avaliar o protótipo desenvolvido foram conduzidos experimentos envolvendo objetos deformáveis de formas variadas. Todos os experimentos foram executados numa estação de trabalho com sistema operacional Linux (Fedora 8) com processador Intel Core 2 Duo a 2.4Ghz e 1 GB de memória.

Os experimentos visam avaliar o desempenho do sistema em função do número de objetos envolvidos na simulação e da complexidade desses objetos. A Tabela 6.1 mostra as resoluções em vértices, faces e tetraedros dos objetos usados.

Tabela 6.1: objetos com resoluções diferentes.

Objeto	Resolução	Vértices na superfície	vértices	faces	tetraedros
coelho		436	510	868	1750
pato		424	519	846	1819
tubo		220	340	436	1270
esfera	A	98	125	192	320
	B	218	343	432	1080
	C	386	729	768	2560
cubo	A	132	168	260	592
	B	199	279	394	1065
	C	309	459	614	1764

Todas as simulações usaram coeficiente de rigidez $\alpha = 0.8$ nos objetos (quase rígidos).

Foram coletados alguns indicadores em termos de quadros por segundo, quantidade de primitivas em colisão (vértices em colisão potencial, faces, tetraedros e vértices em colisão real) processadas a cada intervalo de tempo e a porcentagem gasta por cada sub-processo (detecção de colisões potenciais, detecção exata de colisões, profundidade de penetração, casamento de formas) em milisegundos.

Um primero experimento consiste na simulação física de uma cena com diferentes tipos de objetos (veja a Figura 6.1): 3 patos, 2 coelhos e 3 esferas de resolução C (Tabela 6.1). Este resultado mostra que o protótipo lida com objetos de geometria arbitrária desde que estejam triangulados adequadamente.

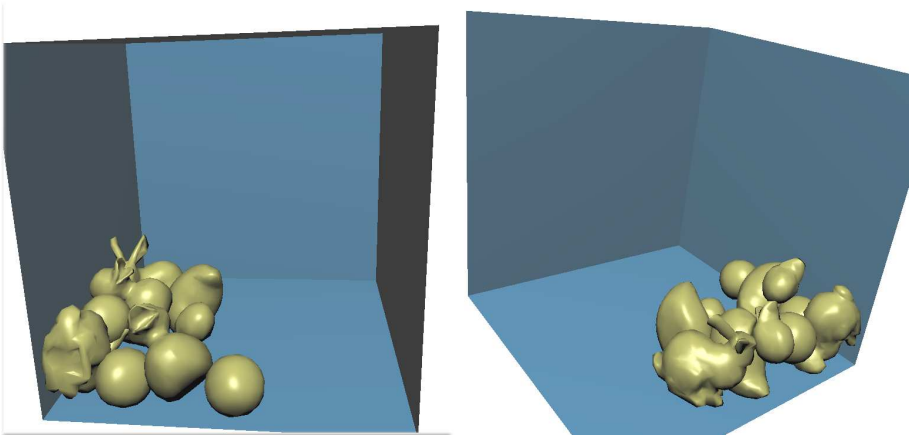


Figura 6.1: oito objetos em contato: 3 patos, 2 coelhos e 3 esferas. A cena contém 2952 vértices e 9917 tetraedros animados a 32 fps.

Do experimento podemos obter algumas informações, por exemplo, a Figura 6.2 mostra o tempo gasto em milisegundos nos sub-processos mais importantes: detecção de colisões potenciais, detecção exata de colisões, computação da profundidade de penetração e casamento de formas a cada intervalo de tempo. Pode-se ver que a detecção de colisões, nas duas fases, toma a maior parte do tempo, sendo que o cálculo da profundidade de penetração se mantém quase constante e o casamento de formas usando todos os vértices da superfícies dos objetos toma um tempo insignificante em relação aos outros sub-processos.

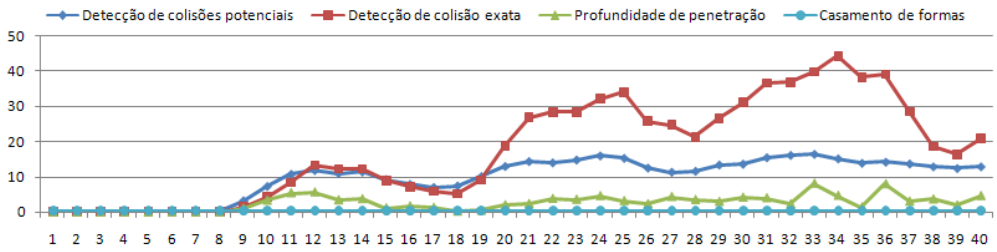


Figura 6.2: tempo gasto em milisegundos para cada sub-processo a cada intervalo de tempo.

A Figura 6.3 mostra, para o mesmo experimento, a quantidade de primitivas em colisão (vértices em colisão potencial, faces, tetraedros e vértices em colisão) por intervalo

de tempo. Note que a filtragem grosseira detecta uma quantidade de vértices em colisão potencial significativamente inferior ao total de vértices na cena. Em média, a simulação registra 487 vértices em colisão potencial, sendo que as colisões reais envolvem 37 faces, 82 tetraedros e 28 vértices em média.

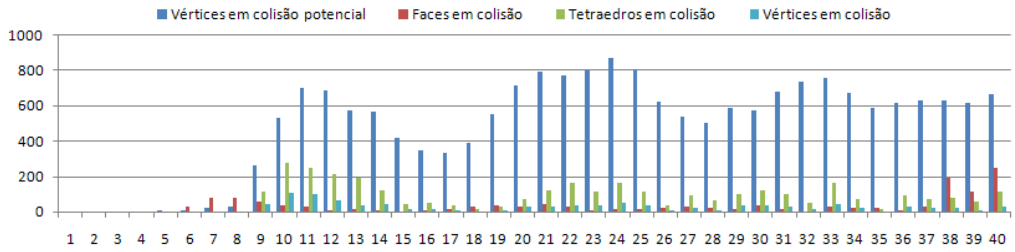
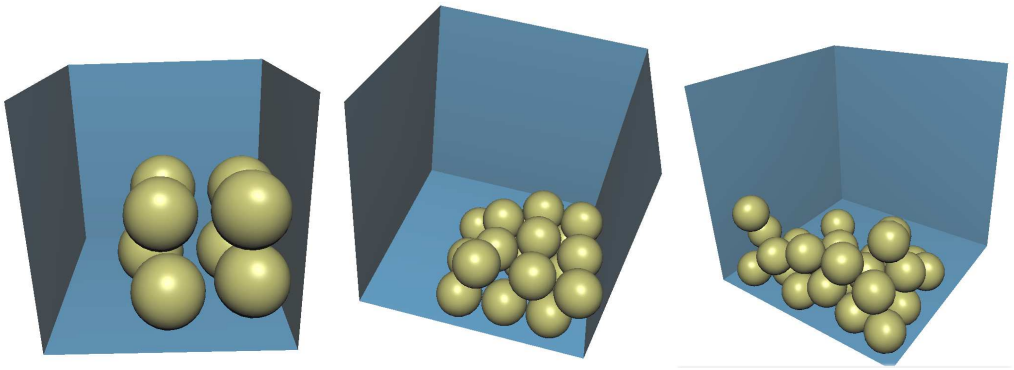


Figura 6.3: número de primitivas em colisão, a cada intervalo de tempo.

Foram também conduzidos outros experimentos usando quantidades variáveis de esferas de forma a avaliar a escalabilidade do método. A Figura 6.4, mostra a simulação de cenas com 8,18 e 27 esferas de resolução C (Tabela 6.1). As cenas contêm 5832 vértices e 20480 tetraedros (8 esferas), 13122 vértices e 46080 tetraedros (18 esferas) e 19683 vértices e 69120 tetraedros (27 esferas). O desempenho do método para esses experimentos foi de 62, 41 e 22 quadros por segundo em média, respectivamente. Um gráfico com a taxa de quadros por segundo a cada intervalo de tempo é mostrado na Figura 6.5.



(a) (b) (c)

Figura 6.4: experimentos com 8 (a), 18 (b) e 27 (c) esferas de resolução C.

Nas Figuras 6.6 e 6.7 são mostrados gráficos para os tempos gastos em cada subprocesso e o número de primitivas em colisão por intervalo de tempo. Os valores médios para essas estatísticas são relacionados na Tabela 6.2.

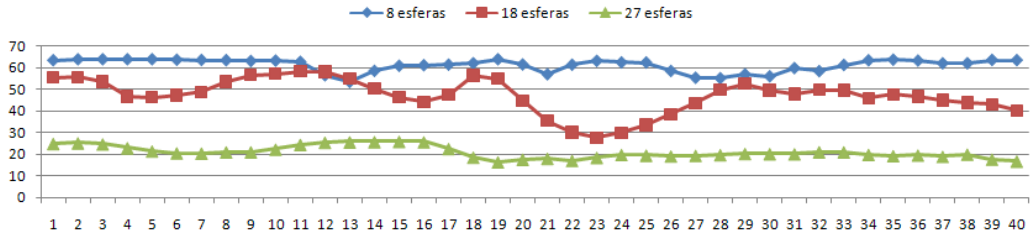
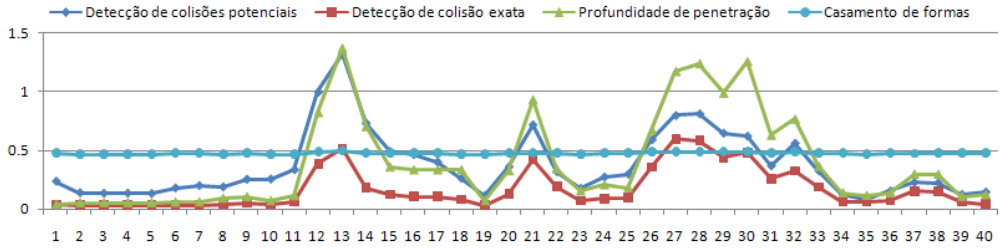


Figura 6.5: taxa de quadros por segundo por intervalo de tempo de esferas com 8, 18 e 27 objetos.

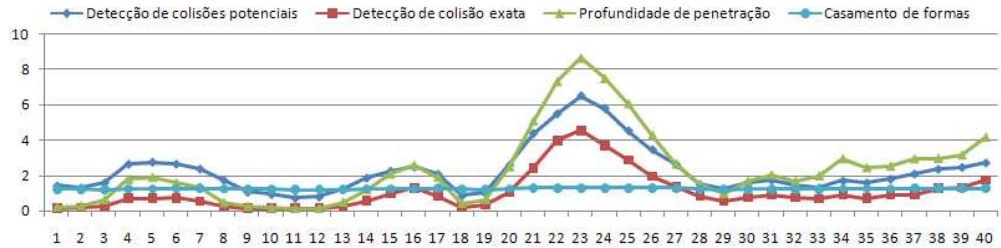
Tabela 6.2: valores médios para primitivas em colisão e tempos de sub-processos para experimentos com número variável de esferas.

No. Esferas	Primitivas em Colisão				Tempo de sub-processos			
	Vért. pot.	Faces	Tet.	Vért. col.	Col. pot.	Col. exata	Prof. penet.	Cas. formas
8	23	12	35	18	0,37	0,17	0,38	0,45
18	108	65	179	90	2,2	1	2	1,2
27	210	123	367	177	4,4	2,3	4,7	1,9

(a)



(b)



(c)

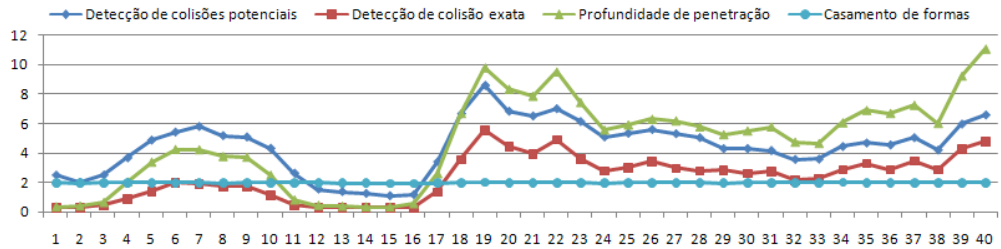


Figura 6.6: tempo em milisegundos gasto para cada sub-processo a cada intervalo de tempo para o experimento envolvendo 8(a), 18(b) e 27(c) esferas.

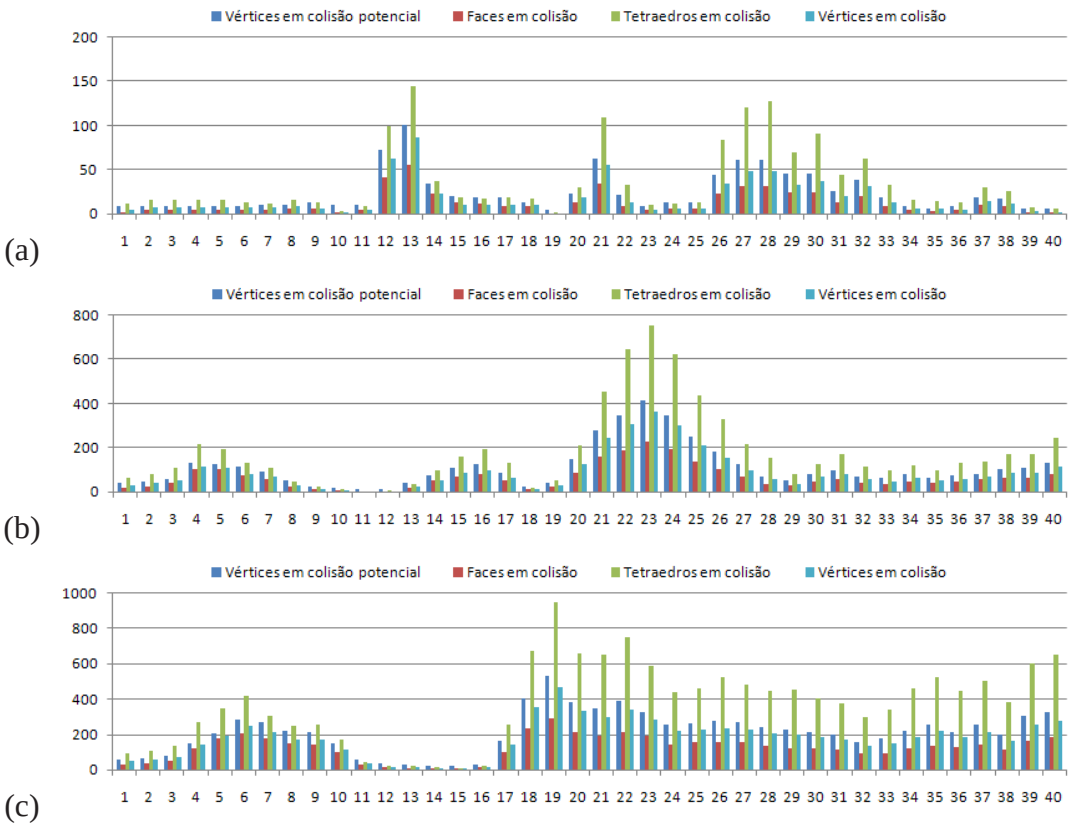


Figura 6.7: número de primitivas em colisão por intervalo de tempo para o experimento envolvendo 8(a), 18(b) e 27(c) esferas.

Capítulo 7

Conclusões e trabalhos futuros

Foi apresentado um protótipo de sistema de animação baseada em física, que usa várias técnicas, como a detecção de colisões usando uma *Hashing* espacial [43] e volumes limitantes, a resposta às colisões através do cômputo da superfície de contato, que utiliza o cálculo da profundidade de penetração por propagação [18], a estimativa dos vetores de deslocamento [18] e a resolução das colisões assimétricas [19] dos vértices da região de deformação, e Busca Binária para separar os objetos [39], a animação é feita usando uma técnica de casamento de formas e um integrador de Euler explícito-implícito [26].

A estrutura original de detecção de colisões foi estendida usando-se um mecanismo de detecção de regiões de colisão potencial, para minimizar as regiões atualizadas pela tabela *hash*, onde se efetuam as colisões reais. Também, no esquema de resposta a colisões, é usado o método de projeção de Jakobsen [19] para resolver colisões assimétricas.

A detecção e resposta às colisões permitem identificar e resolver colisões dos objetos baseados em malhas tetraedrais, em objetos rígidos e deformáveis.

O sistema oferece simulações plausíveis, produzindo respostas às colisões de forma convincente e próxima às metodologias tradicionais, conseguindo manipular objetos complexos e objetos empilhados.

Para trabalhos futuros, planeja-se extender o protótipo do sistema apresentado, na animação, da deformação (agora linear) à quadrática, e plástica; na detecção de colisões o uso de uma hierarquia de esferas limitantes, e finalmente, para aperfeiçoar o código e alcançar um melhor desempenho, desenvolver o sistema usando *GPU*.

Referências Bibliográficas

[1] Marc Alexa, Daniel Cohen-Or, and David Levin. As-rigid-as-possible shape interpolation. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 157–164, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[2] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, 1987.

[3] George Baciu, Wingo Wong, and H. Sun. Recode: An image-based collision detection algorithm. *The Journal of visualization and Computer Animation*, 10(4):181 – 192, 1998.

[4] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 43–54, New York, NY, USA, 1998. ACM.

[5] Gino Van Den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *Journal Graph. Tools*, 4(2):7–25, 1999.

[6] Gino Van Den Bergen and Gino Johannes Bergen. *Collision Detection*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[7] David Breen, Donald House, and Michael Wozny. Predicting the drape of woven cloth using interacting particles. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 365–372, New York, NY, USA, 1994. ACM.

[8] David Breen, Sean Mauch, Ross Whitaker, and Jia Mao. 3D metamorphosis between different types of geometric models. In *Eurographics 2001 Proceedings*, volume 20(3), pages 36–48. 2001.

[9] Kelvin Chung and Wenping Wang. Quick collision detection of polytopes in virtual environments. In *ACM Symposium on Virtual Reality Software and Technology*, pages 1–4, University of Hong Kong, Hong Kong, 1996. ACM.

[10] Dave Eberly. *Game Physics*. Elsevier Science Inc., New York, NY, USA, 2003.

[11] Susan Fisher and Ming C. Lin. Deformed distance fields for simulation of non-penetrating flexible bodies. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 99–111, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[12] Sarah Frisken, Ronald Perry, Alyn Rockwood, and Thouis Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[13] Fabio Ganovelli, John Dingliana, and Carol O'Sullivan. Buckettree: Improving collision detection between deformable objects. In *Spring Conference in Computer Graphics (SCCG)*, 2000.

[14] Elmer Gilbert, Daniel Johnson, and Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *Journal of Robotics and Automation*, 4(2):193 – 203, 1988.

[15] Xiaohu Guo. *Point-Based Modeling, Animation, and Simulation System for Computer Graphics*. PhD thesis, Stony Brook University, New York, USA, 2006.

[16] Kris Hauser, Chen Shen, and James O'Brien. Interactive deformation using modal analysis with constraints. In *Graphics Interface*, 2003.

[17] Bruno Heidelberger, Matthias Teschner, and Markus Gross. Real-time volumetric intersections of deforming objects. In *Proceeding of Vision, Modeling, Visualization VMV*, pages 461–468, 2003.

[18] Bruno Heidelberger, Matthias Teschner, Richard Keiser, and Matthias Müller. Consistent penetration depth estimation for deformable collision response. In *Proceedings of Vision, Modeling, Visualization VMV'04*, pages 157–164, Stanford, USA, 2004.

[19] Thomas Jakobsen. Advanced character physics. In *Proceedings, Game Developer's Conference 2001*, SJ, USA, 2001. GDC Press.

[20] Doug James and Dinesh Pai. BD-Tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics (SIGGRAPH 2004)*, 23(3), aug 2004.

[21] Kenichi Kanatani. Analysis of 3-d rotation fitting. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(5):543–549, 1994.

[22] Danny Kaufman, Timothy Edmunds, and Dinesh Pai. Fast frictional dynamics for rigid bodies. *ACM Trans. Graph.*, 24(3):946–956, 2005.

[23] P. Lancaster and K. Salkauskas. Surfaces generated by moving leas squares methods. In *Mathematics of Computation*, pages 141–158, 1981.

[24] Tomas Larsson and Tomas Akenine-Möller. Collision detection for continuously deforming bodies. *In Eurographics*, pages 325 –333, 2001.

[25] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animationr3. *SIGGRAPH Comput. Graph.*, 22(4):289–298, 1988.

[26] Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. In *Proceedings of SIGGRAPH'05*, pages 471–478, New York, NY, USA, 2005.

[27] Matthias Müller, Richard Keiser, Andrew Nealen, Mark Pauly, Markus Gross, and Marc Alexa. Point based animation of elastic, plastic and melting objects. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 141–151, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[28] Karol Myszkowski, Oleg G. Okunev, and Tosiyasu L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. In *The Visual Computer*, volume 11, pages 497–512, New York, NY, USA, 1995. Springer Berlin-Heidelberg.

[29] Andrew Nealen, Matthias Müller, Richard Keiser, Eddy Boxerman, and Mark Carlson. Physically based deformable models in computer graphics. In *Computer Graphics Forum*, pages 809–836. Blackwell Publishing Ltd., 2006.

[30] Luciana Porcher Nedel. Simulation of deformable objects based on dynamic analysis. Master's thesis, Curso de Pós-Graduação em Ciência da Computação UFRGS, Porto Alegre, RS, Brasil, 1993.

[31] Mark Pauly, Dinesh Pai, and Leonidas Guibas. Quasi-rigid objects in contact. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 109–119, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[32] A. Pentland and J. Williams. Good vibrations: modal dynamics for graphics and animation. *SIGGRAPH Comput. Graph.*, 23(3):207–214, 1989.

[33] Yalmar Ponce. Esquema de detecção e resposta a colições para animação física simplificada. Master's thesis, Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, Brasil, 2005.

[34] J.A. Sethian. A fast marching level set method for monotonically advancing fronts. In *National Academy of Science*, pages 1591–1595, 93(4).

[35] Mikio Shinya and Marie-Claire Forgue. Interference detection through rasterization. In *The Journal of Visualization and Computer Animation*, volume 2, pages 132–134, 1991.

[36] Ken Shoemake and Tom Duff. Matrix animation and polar decomposition. In *Proceedings of the conference on Graphics interface '92*, pages 258–264, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[37] Christian Sigg, Ronald Peikert, and Markus Gross. Signed distance transform using graphics hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 12, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Jonas Spillmann, M. Becker, and Matthias Teschner. Non-iterative computation of contact forces for deformable objects. *Journal of WSCG*, 15(1-3):33–40, 2007.

[39] Jonas Spillmann and Matthias Teschner. Contact surface computation for coarsely sampled deformable objects. In *Proceedings of Vision, Modeling, Visualization VMV'05*, pages 16–18, Stanford, USA, 2005.

[40] Denis Steinemann. Generation and animation of shells. Technical report, Computer Graphics Laboratory - Department of Computer Science, ETH Zürich, 2004.

[41] Hartono Sumali. *A New Adaptive Array of Vibration Sensors*. PhD thesis, Mechanical Engineering Virginia Polytechnic Institute and State University, Virginia, USA, 1992.

[42] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 205–214, New York, NY, USA, 1987. ACM.

[43] Matthias Teschner, Bruno Heidelberger, Dinesh Manocha, Naga Govindaraju, Gabriel Zachmann, Stefan Kimmerle, Johannes Mezger, and Arnulph Fuhrmann. Collision handling in dynamic simulation environments. In *Eurographics Tutorial # 2*, pages 1–4, Dublin, Ireland, 29 August 2005. Eurographics Association.

[44] Matthias Teschner, Bruno Heidelberger, Matthias Müller, and Markus Gross. A versatile and robust model for geometrically complex deformable solids. In *CGI '04: Proceedings of the Computer Graphics International (CGI'04)*, pages 312–319, Washington, DC, USA, 2004. IEEE Computer Society.

[45] Matthias Teschner, Bruno Heidelberger, Matthias Müller, and Danat Pomeranets. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling, Visualization VMV'03Proceedings of SPM 2005*, pages 47–54, 2003.

[46] Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, Marie-Paule Cani, François Faure, N. Magnetat-Thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. In *Computer Graphics Forum*, pages 61–81. Eurographics Association, Eurographics Association and Blackwell Publishing, 2005.

[47] Greg Turk. Interactive collision detection for molecular graphics. Master's thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1990.

[48] Shinji Umeyama. Least-squares estimation of transformation parameters between two point patterns. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(4):376–380, 1991.

[49] Jianhua Wu and Leif Kobbelt. Piecewise linear approximation of signed distance fields. In *Proceedings of Vision, Modeling and Visualization 03*, pages 513 – 520. RWTH, 2003.

[50] Dongliang Zhang and Matthew M. F. Yuen. Collision detection for clothed human animation. In *PG '00: Proceedings of the 8th Pacific Conference on Computer Graphics and Applications*, page 328, Washington, DC, USA, 2000. IEEE Computer Society.