Universidade Federal Fluminense
Instituto de Computação
Graduação em Ciência da Computação

Yan Ramos da Silva

Simulação de tecidos usando o modelo massa-mola e estruturas de dados topológicas

Niterói-RJ
2013

YAN RAMOS DA SILVA

SIMULAÇÃO DE TECIDOS USANDO O MODELO MASSA-MOLA E ESTRUTURAS DE DADOS TOPOLÓGICAS

Trabalho de Conclusão de Curso apresentado ao curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. MARCOS DE OLIVEIRA LAGE FERREIRA

Niterói-RJ

2013

YAN RAMOS DA SILVA

SIMULAÇÃO DE TECIDOS USANDO O MODELO MASSA-MOLA E ESTRUTURAS DE DADOS TOPOLÓGICAS

Trabalho de Conclusão de Curso apresentado ao curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Aprovada em março de 2013.

BANCA EXAMINADORA

Prof. Dr. MARCOS DE OLIVEIRA LAGE FERREIRA - Orientador
IC-UFF

Prof. Dr. ANSELMO ANTUNES MONTENEGRO
IC-UFF

Prof. Dr. LEANDRO AUGUSTO FRATA FERNANDES
IC-UFF

Niterói-RJ
2013

À minha família - inclusive a que pude escolher.

Agradecimentos

Agradeço à minha família - meus pais Odete e Jorge Luiz, minha irmã Tayama e minha prima Talita - por seus constantes incentivos que mantiveram minha cabeça erguida quando isso parecia tão difícil.

Agradeço também aos colegas que estiveram presentes tornando estes quatro anos de estudos os mais agradáveis possível, em especial Douglas e Vinicius, que se tornaram amigos que levarei por toda a vida.

Aos amigos que nunca me deixavam sequer pensar que eu não conseguiria e que me ajudaram nos momentos difíceis: Nícholas, Anna, Ilana, Thaissa, Gabriella e Mattheus.

Aos professores do Colégio Salesianos Santa Rosa, por criaram os alicerces que me permitiram chegar aqui, em especial Adriana Guimarães, Jane e Karla.

Por fim, aos professores do Instituto de Computação, por oferecerem seu conhecimento e apoio nestes oito períodos, sendo a inspiração que me motivou a buscar uma carreira acadêmica. Agradeço especialmente ao meu orientador, prof. Marcos Lage, por sua paciência, compreensão, ajuda e participação na minha formação.

C'est pour cela que je suis née

Joanne D'Arc

Lista de Figuras

Lista de Tabelas

Sumário

Resumo

A simulação de tecidos não é uma tarefa trivial, pois as características físicas particulares a este tipo de *soft body* tornam complexa a sua modelagem. Para realizá-la, foram propostas diversas técnicas baseadas em características geométricas ou físicas dos tecidos. Dentre estas abordagens, o modelo massa-mola-amortecedor destaca-se por permitir uma modelagem suficientemente realista sem exigir cálculos físicos e geométricos de grande complexidade. Esta técnica tem como principal desvantagem a utilização de grande quantidade de memória principal para a representação de malhas com grande número de vértices, necessárias para simulações que exigem alto grau de realismo.

A estrutura de dados topológica chamada CHE (*Compact Half-Edge*) tem como principal característica a escalonabilidade, isto é, é capaz de balancear o uso de memória e processamento para otimizar a representação do modelo 3D e o acesso às suas informações topológicas.

Este trabalho tem como objetivo o desenvolvimento de um simulador de tecidos baseado no modelo massa-mola e na CHE.

Palavras-chave: computação gráfica, tecidos, simulação realista, estruturas de dado topológicas.

Abstract

Cloth simulation isn't a trivial task, as its unique physical characteristics, typical of the soft bodies, make its modelling complex. To accomplish this, many techniques based on the physical or geometrical features of cloths were proposed. Among them, the mass-spring-damper model stands out for allowing a sufficiently realistic modelling without requiring arduous calculations. One of the main drawbacks of this technique is the use of a great amount of main memory to enable the representation of meshes with many vertices, which are needed for simulations that require a high degree of realism.

The topological data structure called CHE (*Compact Half-Edges*) have as its main feature the scalability. In other words, it is capable of balancing memory usage and processing time in other to optimize the representation of a 3D model and the access to its topological information.

This work aims the development of a cloth simulator based on the mass-spring model and on the CHE.

Keywords: computer graphics, cloths, realistic simulation, topological data structures.

Capítulo 1

Introdução

1.1 Descrição do problema

Ao estudar simulação de tecidos em computação gráfica, observa-se que se trata de um problema complexo. Muitos dos detalhes de peças de roupa, bandeiras e outros objetos feitos deste material parecem diferentes a cada vez que são observados, pois são afetados por variáveis do ambiente em que se encontram, como o vento, e por propriedades intrínsecas ao tecido, como sua composição e a organização de suas fibras. Por esse motivo, a representação do seu comportamento através de imagens geradas por computador requer um tratamento especial, não trivial.

Este trabalho descreve um método de simulação tridimensional de tecidos que busca obter o maior realismo funcional possível, para que a cena virtual proporcione ao observador a mesma sensação visual que uma cena real. Assim, a modelagem deve ser realizada de tal forma que seja possível identificar visualmente que o objeto sendo representado na tela é um tecido, mesmo que seu comportamento não seja fisicamente realista.

1.2 Motivação

O emprego no entretenimento digital pode ser citado como uma das principais motivações para os estudos na área da simulação de tecidos. O lançamento do filme *Final Fantasy* (Figura 1.1a) em 2001 definiu um marco nos filmes de animação por ser a primeira obra deste gênero que visou o realismo [22]. Desde então, diversos filmes passaram a buscar a geração de imagens realistas, como foi o caso do filme *Avatar* (Figura 1.1c). Até mesmo em obras com personagens estilizadas e descompromissadas às proporções de humanos reais, como o recente *A Origem dos Guardiões* (Figura 1.1b), a simulação de tecidos está presente, por exemplo, na representação das roupas. Em todos os casos, é dada importância ao realismo funcional, de forma que haja a maior semelhança visual possível entre um tecido real e o representado pela computação gráfica.

(a) *Final Fantasy* (2001).

(b) *A Origem dos Guardiões* (2012).

(c) *Avatar* (2009).

Figura 1.1: Cenas de filmes em CGI (*Computer-Generated Images*) que utilizam a simulação de tecidos.

Os jogos eletrônicos também têm grande importância no incentivo às pesquisas sobre a simulação de tecidos. Com a disponibilidade de plataformas dotadas de CPUs (unidade central de processamento, do inglês *central processing unit*) e GPUs (unidade de processamento gráfico, do inglês *graphics processing unit*) cada vez mais avançadas e grandes quantidades de memória, os estúdios desenvolvedores de videogames passaram a poder utilizar motores gráficos (do inglês *graphic engines*) com capacidade de gerar gráficos extremamente realistas. Muitos jogos são criados com o intuito de criar uma ambientação que se confunda com uma cena real filmada, incluindo personagens extremamente semelhantes a humanos reais. Nestes casos, uma simulação realista de tecidos torna-se muito importante para auxiliar a criar a sensação de verossimilhança desejada. Porém, obter este realismo num jogo, onde a renderização é feita em tempo real, torna-se uma tarefa ainda mais difícil do que na modelagem 3D voltada para o cinema. A evolução da simulação de tecidos em tempo real é visível ao compararmos a demonstração do jogo *Final Fantasy VII* (Figura 1.2a) feita em 2005 pela empresa *Square Enix* para mostrar o poder do console *Playstation 3* e a demo técnica apresentada pela mesma empresa em 2012 (Figura 1.2b).

(a) Demonstração apresentada em 2005 (*Final Fantasy XVII*).

(b) Demonstração apresentada em 2012 (*Agni's Philosophy*).

Figura 1.2: Demonstrações técnicas criadas pela desenvolvedora de jogos *Square Enix*.

A simulação de tecidos também é importante para a indústria da moda, que cada vez mais consegue aliar a informática à atividade artística e artesanal de confecção de roupas. Alguns programas de computador já conseguem simular com bastante fidelidade peças de vestuário a partir do molde e informações sobre o tipo de tecido do qual ela será feita. Com isso, é possível prever como a peça ficará no corpo de uma pessoa antes de ser produzida, permitindo fazer alterações, caso necessárias, sem que haja prejuízo e gastos desnecessários com tecidos e mão-de-obra. Um dos softwares mais utilizados com esta finalidade é o *Marvelous Designer 2*, desenvolvido pela *CLO Virtual Fashion*. Uma simulação de peças de roupa feita com este programa pode ser vista na Figura 1.3 e mostra o quão próximo à realidade ele é capaz de chegar. Este grande realismo é necessário para os profissionais da moda, pois é preciso saber exatamente como a roupa irá ficar quando confeccionada, de forma que as decisões tomadas com base no modelo 3D gerado não sejam equivocadas.

Figura 1.3: Simulação gerada pelo *software Marvelous Designer 2*, da *CLO Virtual Fashion*.

1.3 Organização do trabalho

Este trabalho está organizado em sete capítulos. Neste primeiro capítulo, foi apresentada uma breve introdução ao problema da simulação de tecidos. No segundo capítulo, serão apresentadas informações teóricas sobre as técnicas frequentemente utilizadas para simular tecidos em computação gráfica. Os capítulos 3 e 4 abordarão as escolhas feitas quanto ao modelo físico e a estrutura de dados usados na implementação. No sexto capítulo, serão mostrados os resultados obtidos. Por fim, o capítulo 7 fará uma conclusão apresentará ideias para trabalhos futuros.

Capítulo 2

A simulação de tecidos

2.1 Dificuldades

A simulação de tecidos não é uma tarefa simples. Sua modelagem possui uma série de desafios a serem superados de forma a realizar uma simulação com realismo suficiente para que um observador seja capaz de perceber o tecido como tal.

A complexidade do sistema físico envolvido na modelagem de tecidos é um destes problemas. Tecidos são classificados como corpos macios (do inglês *soft bodies*), nos quais a distância relativa entre um par de pontos em sua superfície pode variar. Devido a este fato, é necessário, para cada *frame* da animação, recalcular a posição de cada ponto usado na modelagem do tecido, que pode ser afetada pelo deslocamento de um grande conjunto de vértices dos triângulos que o compõem no instante anterior. Considerando que, em geral, animações de alta qualidade possuem uma cadência mínima de 24 *frames* por segundo (fps), este recálculo deverá ser feito ao menos 23 vezes por segundo. Em alguns casos em que a taxa de quadros por segundo mínima necessária é maior, como em alguns jogos de ação, por exemplo, este número aumenta ainda mais.

O cálculo das novas posições torna-se ainda mais trabalhoso devido a quantidade de variáveis físicas envolvidas no mesmo, como a elasticidade do tecido, sua densidade e o conjunto de forças que agem sobre cada um de seus pontos, por exemplo. Além disso, algumas peculiaridades do material como a composição das fibras e a forma como elas estão organizadas podem afetar o comportamento do tecido, dificultando ainda mais a sua modelagem. Por este motivo, em aplicações de entretenimento, tecidos costumam ser simulados visando o realismo funcional, priorizando uma simulação que pareça correta aos olhos do observador em detrimento de uma que esteja de acordo com as leis da física, o que envolveria um enorme esforço computacional e, em grande parte das aplicações, é inviável.

Um outro problema existente na simulação de tecidos é, considerando a sua representação através de malhas poligonais, a necessidade de que as utilizadas sejam grandes para representar tecidos com a fidelidade desejada. A figura 2.1 mostra, à esquerda, a foto de um tecido leve e, à direita, uma simulação realizada para tentar reproduzir a configuração de dobras visível na fotografia. Devido à leveza do tecido real, foi necessário utilizar uma malha grande, com dois milhões de triângulos, para representar

as várias dobras e rugas de diferentes tamanhos formadas pelo tecido. Uma malha de menor resolução provavelmente não teria sido capaz de representar a cena fotografada com a riqueza de detalhes desejada. Modelos com grande número de polígonos, porém, além de poderem ocupar bastante espaço em memória, levam muito tempo para serem processados. No caso da simulação da Figura 2.1, cada quadro levou, em média, 6 *minutos* para ser renderizado por um *grid* de 16 processadores [21].

Figura 2.1: Simulação (à direita) visando reproduzir um tecido leve fotografado (à esquerda).

2.2 Técnicas de simulação

Um dos passos mais importantes na simulação de tecidos é a escolha dos modelos físico e numérico utilizados. Para isso, é preciso ter em mente dados sobre o resultado desejado, como o nível de realismo funcional e físico que pretende-se obter, a disponibilidade de recursos computacionais para a renderização e o momento em que este processo será feito (pré-renderização ou em tempo real).

As técnicas de simulação de tecidos podem ser divididas em duas grandes categorias, que são apresentadas nas próximas subseções.

2.2.1 Técnicas geométricas

A primeira técnica geométrica para a modelagem de tecidos em computação gráfica foi proposta em 1986 por Weil [23]. Nela, o tecido é representado por um grid de pontos 3D e, para cada conjunto de três pontos do grid, é definida uma curva catenária, calculada através da função cosseno hiperbólico. Em casos de conjuntos que se sobrepunham, apenas a curva mais baixa é considerada ao realizar a renderização.

O principal problema da técnica das curvas catenárias é o esforço necessário para realizar o cálculo, já que a função cosseno hiperbólico é dada por

$$coshx = \frac{e^x + e^{-x}}{2}$$

.

Este cálculo é computacionalmente caro, pois a avaliação da função exponencial está diretamente relacionada ao algoritmo escolhido para realizar as multiplicações envolvidas. Por e ser um número de ponto flutuante, não podem ser usados algoritmos otimizados para multiplicações de inteiros, como os de Schönhage–Strassen [20] ou de Fürer [6], por exemplo. Uma das opções mais eficientes é o algoritmo de Karatsuba [9], cuja complexidade $M(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$. Utilizando o algoritmo de exponenciação por redução de Montgomery [15], um método eficiente de cálculo de funções exponenciais, a computação de cada potência tem complexidade $O(x\ M(n))$. Para uma malha com n vértices, são realizadas $2\binom{n}{3}$ exponenciações por *frame*, tornando esta técnica inviável para animações com taxa de cadência média-alta e/ou malhas de alta resolução devido ao elevado custo computacional. Em uma malha com 50 vértices, por exemplo, seria necessário calcular 19.600 cossenos hiperbólicos por *frame*, através de 39.200 funções exponenciais computadas.

Nos anos seguintes, novas técnicas geométricas emergiram, utilizando também informações físicas no modelo. Uma delas utiliza técnicas de geometria computacional para realizar uma estimativa do formato de um tecido pendurado por pontos específicos [19]. Porém, o fato de apenas estimar o formato do tecido e de ser restrita a tecidos pendurados acaba limitando-a demais. Outra técnica utilizando geometria e física foi proposta mais tarde e baseia-se na análise prévia de uma pequena folha de material deformável para identificar características de um tecido se enrugando, que são, então, utilizadas para a geração de um modelo para o tecido [11]. Esta técnica mostra-se eficiente para modelar rugas semelhantes às de tecidos, porém, não apresenta resultados satisfatórios para simulações com dinâmicas complexas.

2.2.2 Técnicas físicas

Na mesma época da proposição da primeira técnica geométrica por Weil [23], também foi apresentada por House e Breen a primeira técnica física de modelagem de tecidos [8]. Ela também utiliza um *grid* de pontos tridimensionais, porém, neste caso, ele é utilizado para gerar um conjunto de equações de energia que representavam informações físicas sobre o tecido, como sua tensão, flexão e a ação da gravidade sobre ele. Esta técnica, derivada da teoria da elasticidade, busca tratar o tecido como um objeto contínuo ao invés de um conjunto de curvas independentemente calculadas.

Em 1992, uma nova técnica foi apresentada, desta vez considerando o tecido como um conjunto de partículas que interagiam entre si e com o ambiente ao redor [3]. Estas interações são descritas por conexões mecânicas, representadas por funções de energia. É também utilizado um algoritmo de gradiente estocástico, para levar as partículas a um estado de repouso estável [2].

Apesar de gerarem simulações fisicamente e, consequentemente, visualmente realistas, as técnicas físicas acima têm custo elevado, pois realizam cálculos complexos nas funções de energia envolvidas na modelagem. Uma modificação da técnica baseada em partículas foi proposta, transformando o modelo em um sistema de equações diferenciais ordinárias [5]. Esta modificação, porém, continuou demandando um esforço computacional grande demais para conseguir realizar uma simulação dinâmica.

Em 1988, Haumann e Parent propuseram um modelo físico, chamado massa-mola ou massa-mola-amortecedor [7], que seria aperfeiçoado por Provot em 1995 [17]. Esta técnica difere das citadas

anteriormente por ser mais simples e capaz de gerar simulações que, além de serem visualmente semelhantes a um tecido real, também possuem um nível considerável de realismo físico. Ela consiste em considerar cada vértice da malha do tecido como uma massa, conectadas entre si por molas. A Figura 2.2 ilustra a configuração tradicional de massas e molas utilizada nesta técnica. As molas estruturais conectam as massas horizontal e verticalmente e tratam da extensão e compressão do tecido. Já as molas de cisalhamento, conectam as massas diagonalmente. Por fim, as molas de dobra do tecido conectam as massas das extremidades com as posicionadas no lado oposto.

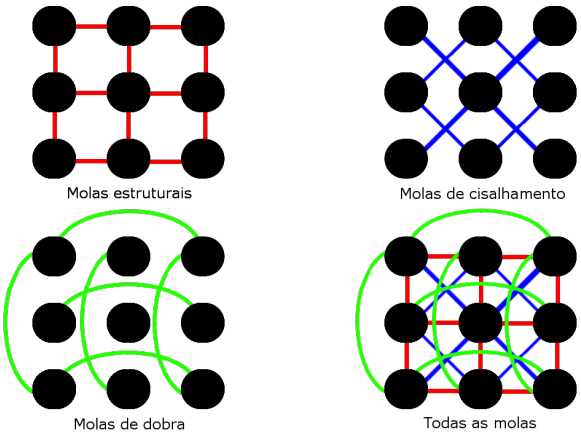


Figura 2.2: Esquema de molas tradicional do modelo massa-mola.

O cálculo das posições das massas baseia-se na Segunda Lei de Newton. A força resultante sobre cada massa é calculada utilizando componentes como a gravidade e a força elástica, além de outras forças externas que possam agir sobre o sistema, como o a resistência do ar ou a ação do vento, por exemplo. Integrando-se numericamente o vetor força resultante, obtêm-se a aceleração da massa. Utilizando esta informação em conjunto com o tempo decorrido desde a realização do último cálculo, é possível encontrar o vetor velocidade da partícula e, com ele, atualizar sua posição.

2.3 Escolha da técnica

Para a realização deste trabalho, optou-se pela modelagem do tecido através do modelo massa-mola. Isto se deve ao fato de que a simulação que pretende-se obter visa a semelhança visual com a realidade, sem priorizar a obtenção de um elevado grau de realismo físico. Apesar deste não ser o objetivo, porém, a técnica utilizada é capaz de propiciar um bom grau de fidelidade, pois é classificada como uma técnica física.

Atualmente, novas técnicas de simulação de tecidos mais sofisticadas foram desenvolvidas, permitindo simulações com maior qualidade e eficiência através de tecnologias como o processamento em GPUs [10] e baseando-se em princípios físicos, matemáticos e computacionais mais complexos, como geometria diferencial [13], mecânica de meios contínuos [1] e refinamento adaptativo anisotrópico de malhas poligonais [16]. Porém, devido a sua complexidade, estas técnicas fogem ao escopo deste trabalho e, por isso, não foram escolhidas.

Capítulo 3

O modelo massa-mola

3.1 Estruturação em massas e molas

A modelagem do tecido usando o modelo massas-mola, ilustrada na Figura 3.1, será baseada na estrutura combinatória de malhas de triângulos. Nela, as *edge springs*, ou molas de aresta, conectam cada vértice aos outros componentes de sua estrela; já as *bending springs*, ou molas de dobra, conectam os vértices opostos por arestas. Na modelagem adotada, as massas são conectadas em uma configuração radial por molas de aresta devido à utilização de triângulos, logo, não é necessário definir o conjunto de molas de cisalhamento do modelo tradicional, baseado em quadrângulos. Já as molas de dobra foram substituídas pelas ilustradas por linhas tracejadas na figura abaixo, que ligam as massas de triângulos adjacentes que não são conectadas por molas de aresta.

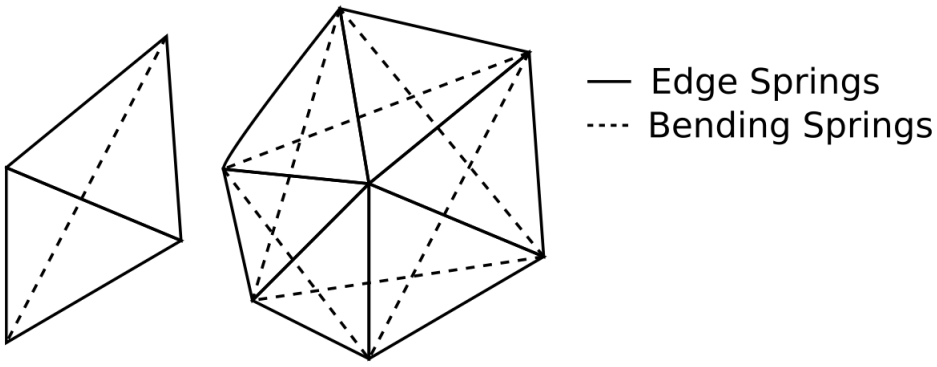


Figura 3.1: Esquema de massas e molas adotado no trabalho.

A opção por triângulos ao invés de quadrângulos se deve ao fato de que, desta forma, é possível realizar a simulação utilizando malhas triangulares arbitrárias, o que torna este método mais flexível.

Nesta configuração, uma malha fechada de n triângulos terá um total da ordem $3n$ molas conectando suas massas. O número de molas ligadas a cada vértice, porém, será variável, dependendo de sua valência.

3.2 Princípios numéricos

Pode-se dizer que a força resultante que age sobre uma massa p_i é resultado da composição de seu peso, da força elástica total exercida pelas molas e_{ij} que possuem nela um de seus extremos e das forças externas, ou seja,

$$\vec{F}_{RES\ i} = \vec{P}_i + \vec{F}_{ELi} + \vec{F}_{EX} = m\vec{g} + \vec{F}_{ELi} + \vec{F}_{EX} \tag{3.1}$$

onde m é o valor da massa e $\vec{g} = (0, -9.789, 0)$.

A força elástica total sobre uma massa p_i, por sua vez, é dada por

$$\vec{F}_{ELi} = \sum_{j=0}^{\#V_j(p_i)} \vec{F}_{ELij} \tag{3.2}$$

onde $\#V_j(p_i)$ denota a vizinhança da massa p_i. Pela Lei de Hooke, temos que a força elástica de uma mola e_{ij} é dada por

$$\vec{F}_{ELij} = -k_s\vec{x}_{ij} \tag{3.3}$$

onde $-k_s$ é a constante elástica e $\vec{x_{ij}}$ é a deformação da mola e_{ij}, que conecta as massas p_i e p_j.

A deformação da mola e_{ij} é calculada por

$$\vec{x}_{ij} = (|\vec{L}_{ij}| - |\vec{L}^0_{ij}|) \cdot \left(\frac{\vec{L}_{ij}}{|\vec{L}_{ij}|}\right) \tag{3.4}$$

onde $\vec{L}_{iJ}$ é o vetor comprimento atual da mola e_{ij} e $\vec{L}^0_{ij}$ é o seu comprimento original.

É necessário também inserir uma força de amortecimento ao cálculo da força elástica para impedir que a mola oscile indefinidamente, alterando a equação da força elástica total para

$$\vec{F}_{ELi} = \sum_{j=0}^{\#V_j(p_i)} \vec{F}_{ELij} + \vec{F}_{AMij} \tag{3.5}$$

Esta força de amortecimento, calculada para cada mola e_{ij} do tecido, é dada por

$$\vec{F}_{AMij} = -k_d(\vec{v}_i - \vec{v}_j) \tag{3.6}$$

onde $-k_d$ é a constante de amortecimento, v_i é a velocidade da massa p_i de uma das extremidades da mola e v_j é a velocidade da massa p_j da outra extremidade da mola.

Finalmente, podemos expressar a equação da força resultante sobre uma massa como

$$\vec{F}_{RESi} = m\vec{g} + \vec{F}_{EX} - \sum_{j=0}^{\#V_j(x_i)} \left[k_s(|\vec{L}_{ij}| - |\vec{L}^0_{ij}|) \cdot \left(\frac{\vec{L}_{ij}}{|\vec{L}_{ij}|}\right) + k_d(\vec{v}_i - \vec{v}_j)\right] \tag{3.7}$$

Nota-se que, no instante inicial da simulação, a velocidade das partículas é nula e o comprimento das molas coincide com o original, fazendo com que, num primeiro momento, apenas a gravidade e as forças externas influenciem no valor da força resultante, fazendo com que as massas que compõem o tecido sejam aceleradas, dando início, assim, ao movimento.

Através da Segunda Lei de Newton, é possível utilizar o valor da força resultante calculada para obter a aceleração da massa, fazendo

$$\vec{a}_i = \frac{\vec{F}_{RESi}}{m} \tag{3.8}$$

Como a aceleração de um corpo é a derivada em relação ao tempo de sua velocidade, podemos obtê-la através da integração

$$v_i(t) = \int \vec{a}_i \, dt \tag{3.9}$$

Para resolver esta integral, é necessário utilizar um método numérico de integração. Neste trabalho, será utilizado o método de Euler. Tomando como base o problema de valor inicial

$$\frac{d\vec{v}_i}{dt} = \vec{a}_i = f(t, \vec{v}_i) = \frac{\Delta\vec{v}_i}{\Delta t} \, , v_i^x(t_0) = v_i^y(t_0) = v_i^z(t_0) = 0 \tag{3.10}$$

,

onde v_i^x é a componente x de $\vec{v}_i$, v_i^y é a componente y de $\vec{v}_i$ e v_i^z é a componente z de $\vec{v}_i$

Podemos dizer que

$$v_i(t_{n+1}) \cong v_i(t_n) + a_i(t_n) \cdot \Delta t \tag{3.11}$$

É importante ressaltar que a equação (3.11) deverá ser calculada três vezes, uma para cada uma das componentes do vetor velocidade de i (v_i^x, v_i^y e v_i^z).

Desta forma, é possível calcular o valor aproximado da velocidade de cada massa em um dado instante de tempo. Utilizando a posição atualmente armazenada do ponto e o intervalo de tempo Δt, o novo vetor posição pode ser obtido através de uma nova integração numérica:

$$x_i(t_{n+1}) = x_i(t_n) + v_i(t_n) \cdot \Delta t \tag{3.12}$$

3.3 Problemas da técnica

A modelagem pelo método de massa-mola possui alguns problemas que devem ser tratados. Um deles, evidenciado pelas equações na seção 3.2 é a quantidade de cálculos envolvidos. Apesar de se basear na Segunda Lei de Newton, mais simples que os princípios físicos utilizados por outras técnicas físicas ou pelo método geométrico de curvas catenárias, o modelo massa-mola ainda exige uma quantidade considerável de cálculos. Para cada *frame*, a força elástica exercida por cada mola deve ser recalculada e somada às forças que agem em cada massa para obtermos o valor da força resultante.

Além de exigir um grande volume de cálculo para cada *frame* gerado, a simulação de tecidos também necessita que haja uma preocupação com a forma como os dados que representam as massas e as molas são estruturados e armazenados. O modelo massa-mola necessita de um acesso frequente a informações sobre a vizinhança de um vértice para realizar o cálculo das forças elásticas. Obter informações sobre as massas na extremidade de uma mola exige uma busca no caso de malhas arbitrárias e, mesmo que seja usada uma lista ordenada para armazenar os dados sobre as massas, será necessária ao menos uma busca binária, de complexidade $O(log\ n)$, a cada cálculo de força elástica realizado. Além

disso, ao calcular a força resultante sobre uma massa, outra busca pelas arestas com extremidade nela é necessária para obter as componentes da força elástica envolvidas no cálculo. Em ambos os casos, o tempo necessário para realizar a busca é proporcional ao número de vértices da malha. Logo, simulações de tecidos compostos por muitos triângulos, como o exemplo mostrado na Figura 2.1, podem apresentar desempenho muito ruim.

Uma alternativa para solucionar o problema de acesso às informações de vizinhança é a utilização de estruturas de dados topológicas, que armazenam, além das informações geométricas dos vértices da malha, dados sobre sua topologia. Neste trabalho foi utilizada a estrutura de dados topológica CHE, que será apresentada no próximo capítulo.

Outro problema também relacionado a malhas de alta resolução é o consumo de memória. Além das posições dos vértices das malhas, é necessário também guardar informações sobre sua velocidade e as forças elásticas de cada mola, por exemplo. Para garantir uma maior consistência na simulação, também é necessário também gerar uma cópia do conjunto de variáveis físicas da malha a cada interação, de forma que uma, contendo as informações referentes ao tempo t, seja utilizada para calcular as quantidades físicas no tempo t+1. Isto faz com que seja necessário armazenar duas cópias do conjunto de informações geométricas dos vértices da malha.

Capítulo 4

As *Compact Half-Edges*

4.1 Definição de estrutura de dados topológica

De modo geral, uma malha poligonal pode ser representada por uma estrutura de dados que guarde apenas as coordenadas de seus vértices, pois suas informações geométricas são suficientes para que o objeto modelado seja localizado e definido no espaço. É possível, porém, incluir mais um nível de abstração, este topológico, para facilitar o acesso a informações sobre a relação de adjacência ou topológica dos componentes da malha.

A topologia pode ser definida como o estudo das propriedades de um objeto que não são modificadas por sua deformação. As adjacências de vértices, arestas e faces de malhas triangulares representando variedades, nas quais a vizinhança de qualquer ponto é homeomorfa a um disco 2D, são exemplos de informações que mantém-se invariantes a deformações. Assim, uma estrutura de dados é dita topológica quando é capaz de prover acesso a essas informações, respondendo, por exemplo, quais as arestas com extremidade em um dado vértice ou as faces adjacentes a um polígono escolhido.

4.2 Modelagem das CHEs

A CHE (do inglês *Compact Half-Edge*) é uma estrutura de dados topológica aplicável para superfícies, proposta por Lage *et al.* [12] com objetivo de ser uma simplificação da estrutura *Handle-Edge* [14] e uma extensão da estrutura *Corner-Table* [18].

A CHE possui três vantagens importantes. Primeiramente, ela é escalonável, ou seja, permite mudar o uso da memória para diminuir o tempo de processamento da malha, sendo a quantidade de dados armazenada na estrutura adaptável a modelos ou aplicações específicas. Em segundo lugar, ela requer pouca memória por basear-se em contêineres de inteiros e regras aritméticas ao invés de ponteiros, o que também facilita sua implementação. Por fim, a CHE é composta por quatro níveis de estrutura, cada um dos quais complementando o anterior, de forma a diminuir o tempo de execução, utilizando, porém, um pouco mais de memória a cada passo. Tais níveis são implícitos para o programador através da herança de classes.

4.3 O nível 0

A CHE baseia-se no conceito de *half-edge*, ilustrado na Figura 4.1, que permite que um triângulo seja associado a uma das arestas que o delimitam e, analogamente, que esta aresta seja associada a um dos vértices em sua extremidade.

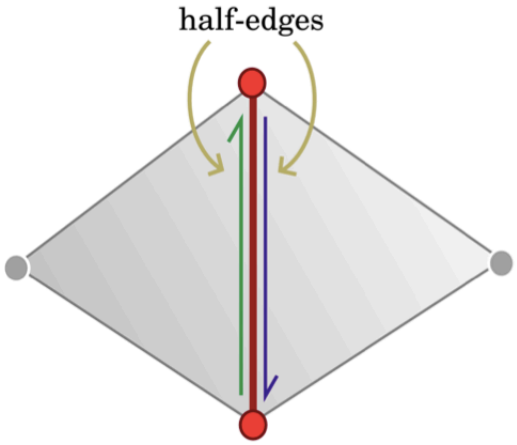


Figura 4.1: Duas *half-edges* associadas a uma mesma aresta

O nível 0 da CHE armazena apenas o conjunto de triângulos que constitui a malha através de um conjunto de *half-edges*. As *half-edges*, os vértices e os triângulos são indexados por inteiros não-negativos. Cada triângulo é representado três *half-edges* consecutivas, o que define sua orientação. As *half-edges* 0, 1 e 2 de uma malha, por exemplo, correspondem ao primeiro triângulo, enquanto as *half-edges* 3, 4 e 5 correspondem ao segundo, e assim sucessivamente.

4.3.1 Geometria dos vértices

A geometria dos vértices é representada através de um contêiner denominado $G[]$, que armazena as informações geométricas como posição e vetor normal de todos os n_0 vértices da malha. Desta forma, para acessar as informações de um vértice identificado por um V_{id} v, é feito um acesso a $G[v]$.

4.3.2 Regras das *half-edges*

Uma *half-edge* indexada por HE_{id} he é associada ao triângulo de índice $\lfloor \frac{he}{3} \rfloor$. Assim, um triângulo representado por T_{id} t possui as *half-edges* $3t, 3t+1$ e $3t+2$. As *half-edges next* e *previous*, mostradas na Figura 4.2, de uma *half-edge* HE_{id} he pertencem ao triângulo $\lfloor \frac{he}{3} \rfloor$ e podem ser obtidas através das seguintes regras:

$$next_{he}\ (he) = 3\lfloor \frac{he}{3} \rfloor + (he+1)\%3 \tag{4.1}$$

$$next_{he}\ (he) = 3\lfloor \frac{he}{3} \rfloor + (he+2)\%3 \tag{4.2}$$

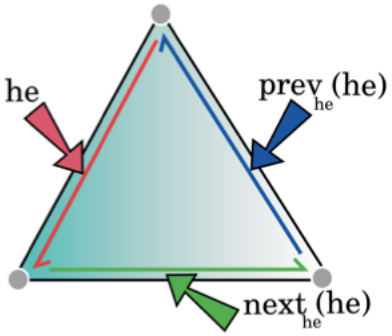


Figura 4.2: As *half-edges next* e *prev* de *he*.

4.3.3 O contêiner de vértices

No nível 0 da CHE, o vértice de partida de cada *half-edge* HE_{id} *he* é armazenado em um contêiner de inteiros, denotado por $V[]$. O inteiro $v = V[he]$ corresponde ao índice do vértice associado à *half-edge he*, conforme ilustrado na Figura 4.3. Este contêiner tem tamanho $3n_2$ e suas entradas variam de 0 a $n_0 - 1$, onde n_2 é o número de triângulos da CHE.

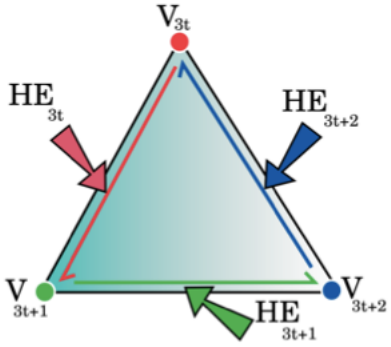


Figura 4.3: Os vértices correspondentes às *half-edges* de um triângulo.

4.4 O nível 1

O nível 1 da CHE adiciona ao nível 0 as informações sobre a adjacência de triângulos. Como assumimos que M é uma 2-*manifold*, cada *half-edge* será sempre incidente a um ou dois triângulos. Para possibilitar o acesso às informações de relação de adjacência de triângulos, é necessário um contêiner adicional, chamado contêiner oposto, denotado por $O[]$. O inteiro $O[he]$ (Figura 4.4) corresponde ao índice da *half-edge* que possui os mesmos vértices que *he*, mas orientação oposta, se existir. Caso *he* esteja na borda da malha, ela não terá uma *half-edge* oposta e $O[he] = -1$. $O[]$ possui tamanho $3n_2$ e varia de -1 a $n_2 - 1$.

4.5 O nível 2

O nível 2 complementa as informações armazenadas pela CHE ao inserir dados sobre as arestas e sua incidência.

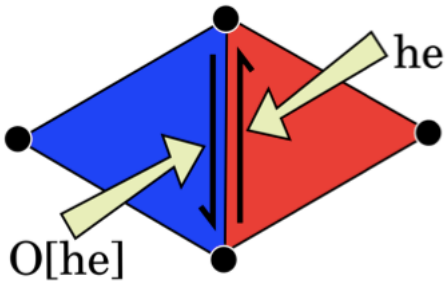


Figura 4.4: A *half-edge* oposta a *he*.

4.5.1 O mapa de arestas

Uma aresta da malha M é representada pela *half-edge* de menor índice que incide sobre ela. Com essa informação, é possível criar um mapa, denotada por EH, para representar explicitamente as arestas, associando cada uma delas ao índices de uma de suas *half-edges* incidentes e eventualmente a suas características, como, por exemplo, seu comprimento original. O mapa EH possui n_1 entradas e pode ser alocado através de um algoritmo de ordem $O(n_1 \log(n_1))$, onde n_1 é o número de arestas da malha M.

4.5.2 O contêiner extra de vértices

Para facilitar uma série de operações geométricas que podem ser realizadas sobre a malha, é interessante que a estrela de vértices seja obtida rapidamente. Para este fim, é criado um novo contêiner de vértices, denotado por $VH[]$, que armazena, para cada vértice v, o índice de uma *half-edge* incidente a ele. Caso v esteja no bordo de M, $VH[v]$ guardará uma *half-edge* do bordo. Este contêiner tem tamanho n_0 e pode ser criado em tempo $O(n_2)$.

4.6 O nível 3

O último nível permite acessar informações sobre a fronteira de 2-*manifolds* M, compostas por conjuntos de 1-*manifolds* combinatórias sem fronteira. A representação eficiente destas informações é importante para algumas aplicações, como a construção e a desconstrução de *manifolds* [14]. Na CHE, esta representação é feita através do armazenamento de uma *half-edge* correspondente a uma das arestas de cada curva de fronteira C. Com base nesta *half-edge*, a curva pode ser percorrida através de operações de estrela da tiangulação.

4.7 As operações da CHE para cada nível

Seja o simplexo σ de dimensão k, ou seja, o fecho convexo de $k+1$ pontos $\{v_0, \ldots, v_k\}$ tais que os vetores $v_1 - v_0, v_2 - v_0, \ldots, v_k - v_0$ são linearmente independentes, denotado por k-simplexo σ. As funções resposta às interrogações topológicas $R_{pq}(\sigma)$ [4] retornam, para um p-simplexo σ, todos os q-simplexos com os quais compartilham faces. No contexto deste trabalho, um 0-simplexo, 1-simplexo e 2-simplexo correspondem, respectivamente, a um vértice, uma aresta e um triângulo da malha M.

4.7.1 Estrela de vértices

A operação R_{0*} permite obter a estrela de vértices, sendo realizada, no nível 0, em tempo $O(n_2)$, pois é preciso percorrer todo o contêiner de vértices $V[]$ em busca dos vértices que compartilham uma aresta com um vértice v_i. Já no nível 1, $V[]$ é percorrido até que uma *half-edge* incidente ao vértice de entrada seja encontrada para que, então, sua estrela possa ser obtida em tempo $O(deg(v))$ através do contêiner $O[]$ e das regras descritas nas equações 4.1 e 4.2. No pior caso, R_{0*} possui complexidade $O(n_2)$ no nível 1, mas, em média, este processo é $deg(v)$ vezes mais rápido. Por fim, nos níveis 2 e 3, a complexidade desta operação é reduzida a $O(deg(v))$, pois $VH[v]$ já armazena a primeira *half-edge* incidente a v, necessária para que seja percorrida a sua estrela.

4.7.2 Estrela de arestas

A obtenção da estrela de arestas de um vértice é mais simples que a operação R_{0*}, pois, como cada aresta é representada por uma *half-edge he* incidente a ela, seus vértices podem ser obtidos por $v_0 = next_{he}\ (he)$ e $v_1 = prev_{he}\ (he)$, respondendo R_{10} em tempo constante em qualquer nível. As relações R_{11} e R_{12} têm complexidade $O(n_2)$ no nível 0, pois é necessário percorrer todo o contêiner $V[]$ para encontrar a *half-edge* oposta, se existir. A partir do nível 1, porém, sua complexidade torna-se $O(1)$, pois a aresta é identificada por uma das *half-edges* a ela incidentes e o contêiner $O[]$ permite acessar a outra diretamente.

4.7.3 Incidência e adjacência de triângulos

No nível zero, a função para a obtenção da estrela de faces, R_{22}, é obtida em tempo $O(n_2)$, pois é preciso encontrar todas as *half-edges* que começam ou terminam em um dado vértice, mas, a partir do nível 1, a complexidade do processamento torna-se constante, pois, usando o contêiner $O[]$, é possível partir de uma *half-edge he* armazenada em $V[]$ e, utilizando as operações $next(he)$ e $prev(he)$ e o contêiner $O[]$, obter as outras *half-edges* incidentes, para, usando as regras aritméticas descritas em 4.3.2, obter os índices dos seus respectivos triângulos.

4.8 Vantagens do uso na simulação de tecidos

O cálculo da força resultante que age sobre cada massa em um tecido no modelo massa-mola requer a obtenção frequente de informações sobre a adjacência de vértices. Neste caso, a CHE mostra-se uma boa opção para o armazenamento dos dados da malha de triângulos que modela o tecido, pois permite o acesso eficiente à estrela de um vértice. A estrela de vértices é necessária para que seja feito o cálculo da força resultante que age sobre o mesmo, utilizado para recalcular sua posição, conforme descrito na seção 3.2.

Além disso, o baixo consumo de memória da CHE é vantajoso para o problema de simulação de tecidos, principalmente nos casos em que a malha simulada possui grande número de vértices e arestas.

Capítulo 5

Implementação

5.1 Informações gerais

O simulador de tecidos apresentado neste trabalho foi desenvolvido na linguagem de programação *Java*. O programa foi criado como um aplicativo para o sistema de processamento de malas poligonais desenvolvido e disponibilizado pelo Prof. Dr. Marcos de Oliveira Lage Ferreira. Este sistema encapsula o código OpenGL e fornece um conjunto de estruturas e algoritmos auxiliares, possibilitando que o programador preocupe-se apenas com a implementação do código diretamente relacionada ao trabalho.

5.2 Estrutura de classes

Um diagrama de classes do aplicativo criado para este trabalho é apresentado no Apêndice A. Ele mostra as classes criadas especialmente para este trabalho, não contemplando todas as classes já existentes no sistema de processamento de malhas poligonais.

A tabela 5.1 apresenta as classes criadas para este trabalho, assim como uma breve descrição de sua função no programa.

Tabela 5.1: Classes da aplicação de simulação de tecidos.

Classe	Descrição
ClothAppSettings	Responsável pela configuração da classe de aplicação e pelo armazenamento de seus parâmetros.
Cloth	Objeto que representa o tecido, estendendo a classe *GLMesh*, que modela a malha poligonal e tem acesso à CHE que a define.
ClothSettings	Responsável pela configuração da classe *Cloth* e pelo armazenamento de seus parâmetros, que incluem as variáveis físicas do tecido e da simulação.
MassInfo	Classe que modela o objeto que armazena as informações sobre uma massa, como o módulo de sua velocidade e seu vetor normalizado.
SpringInfo	Classe que modela o objeto que armazena as informações sobre uma mola, como seu comprimento inicial e a mola de dobra transversal a ela, se houver (vide Figura 3.1).

5.3 Arquivos de configuração

A definição dos parâmetros do programa é feita a partir de arquivos de texto com extensão *.settings*. Cada um deles é tratado por uma classe de configuração. O arquivo é composto por linhas contendo a atribuição de um parâmetro, seguindo o formato

$$chave = valor.$$

Comentários podem no arquivo são precedidos do caracter `#`. O endereço em disco do arquivo deve ser passado como parâmetro para o construtor da classe de configuração correspondente. Após este processo, o valor do parâmetro poderá ser obtido através da chamada de função

$$valor = getProperty(chave),$$

que retorna um *String*.

A tabela apresenta o nome dos arquivos de configuração utilizados pelo programa, os parâmetros por eles definidos e uma breve descrição dos mesmos.

Tabela 5.2: Arquivos de configuração da aplicação de simulação de tecidos.

Arquivo	Parâmetro	Descrição
ClothAppSettings	cmap	Definição do mapa de cores utilizado. Suas opções de valor são citadas nos comentários do arquivo.
	cmapMin	Valor mínimo do mapa de cores.
	cmapMax	Valor máximo do mapa de cores.
	rmode	Modo de renderização utilizado. Aceita os valores *Smooth*, *Points* e *Wireframe*.
	mfile	Endereço do arquivo *.off* com as informações da malha.
	clothScale	Fator de escala aplicado sobre o modelo 3D do tecido.
	clothCenter	Posição do centro do modelo 3D do tecido, representado por um vetor no formato (x, y, z)
	collision	Endereço do arquivo *.off* com as informações do objeto com o qual haverá colisão, ou a palavra *NONE* quando não houver colisão.
	collisionScale	Fator de escala aplicado sobre o modelo 3D do objeto de colisão, se houver.
	collisionCenter	Posição do centro do modelo 3D do objeto de colisão, se houver, representado por um vetor no formato (x, y, z)
ClothSettings	k	Constante elástica, expressa N/m.
	m	Valor da massa dos vértices, expresso em kg.
	g	Aceleração da gravidade, expressa em m/s^2.
	d	Fator de amortecimento das molas, expresso em Ns/m.
	anchors	Lista de vértices-âncora, ou seja, que permanecem imóveis na simulação. É representada por uma sequência de inteiros, cada um dos quais indicando o V_{id} de um vértice, separados por vírgula, ou a palavra *NONE* quando não houver nenhum vértice-âncora.
	forces	Lista de vetores de forças externas. É representada por uma sequência de vetores no formato (x, y, z), cada um dos quais representando as coordenadas vetor, ou a palavra *NONE*, quando não houver forças externas na simulação.

5.4 Funcionamento

Nesta seção serão apresentados os principais algoritmos desenvolvidos neste trabalho.

5.4.1 Inicialização de massas e molas

Quando o simulador é iniciado, os arquivos *.off* com as informações do tecido e do objeto com o qual ele colide, se houver, são lidos e utilizados para gerar as CHEs das malhas. Elas então são escalonadas e transladadas de acordo com os parâmetros *clothCenter*, *clothScale*, *collisionCenter* e *collisionScale* de *ClothAppSettings.settings*. Além disso, as informações das massas e molas do tecido são inicializadas e armazenadas nos vetores $G[]$ e $EH[]$, respectivamente. Cada posição $G[i]$ armazenará o objeto *MassInfo* referente à massa v_i, enquanto cada posição $EH[he]$ armazenará dois objetos *SpringInfo*: um guardará informações da mola estrutural associada à aresta identificada pela *half-edge he*, enquanto a outra conterá informações da mola de dobra transversal a esta mesma aresta, se houver, como ilustra a Figura 3.1. O algoritmo desta inicialização é descrito abaixo.

Algoritmo 1: Inicialização dos objetos *MassInfo* e *SpringInfo* do tecido.

```
for v_i ⟵ G[i] do
    arestas ⟵ estrelaDeArestas(v_i);
    massInfo.vizinhosDiretos ⟵ arestas;
    for e_j ⟵ arestas[j] do
        he ⟵ O[next(e_j)];
        if he > 0 then
            massInfo.adicionaVizinhoTransversal(he);
        end if
        if EH[he] não foi calculado then
            distância ⟵ calculaDistância(v_i, V[next(he)]);
            springInfoDireta.distância ⟵ distância;
            if O[he] ≠ −1 then
                distânciaTransversal = calculaDistância(V[prev(he)], V[prev(O[he])]);
                springInfoTransversal.distância ⟵ distânciaTransversal;
            end if
            EH[he].direta ⟵ springInfoDireta;
            EH[he].transversal ⟵ springInfoTransversal;
        end if
        G[i].info ⟵ massInfo;
    end for
end for
```

Após a execução do algoritmo 2, as informações das massas e molas do modelo estarão armaze-

nadas e poderão ser obtidas diretamente para a realização dos cálculos descritos na próxima seção.

5.4.2 Recálculo de posições

Quando a simulação é iniciada, é criado um contador de passos, com valor inicial 0. Antes de cada renderização da cena, é verificado se este contador armazena um valor inferior ao número máximo de passos, definido como 1000. Caso seja menor, o algoritmo 3 é executado antes da execução do método de renderização. Sua função é, com base no estado atual do tecido e das forças que agem sobre ele, calcular as novas posições dos vértices que o compõem.

Algoritmo 2: Atualização das posições dos vértices da malha.

```
novoG[] ⟵ G[];
for v_i ⟵ G[i] do
    if v_i não é âncora then
        força ⟵ calculaForçaResultante(v_i);
        aceleração ⟵ calculaAceleração(força);
        novaVelocidade ⟵ calculaVelocidade(v_i.info.velocidade, aceleração, Δt);
        novaPosição ⟵ calculaPosição(v_i.posição, Δt);
        if houve colisão then
            novaPosição ⟵ trataColissão(novaPosição);
        end if
        novoG[i].info.velocidade ⟵ novaVelocidade;
        novoG[i].posição ⟵ novaPosição;
    end if
end for
G[] ⟵ novoG[];
```

É importante notar que os valores da velocidade e da posição de um vértice usados no cálculo são obtidos do vetor $G[]$, porém, os novos valores calculados são salvos em $novoG[]$, uma cópia de $G[]$ criada antes da estrutura de repetição, sendo atribuído ao original através do comando $G[] \longleftarrow novoG[]$ apenas após o percorrimento de todos os vértices não-âncora. Isto é feito para que o cálculo do estado do tecido no tempo t_{n+1} leve em consideração apenas informações do instante anterior t_n. Caso os valores calculados fossem salvos imediatamente em $G[]$, no cálculo da força resultante, descrito no algoritmo 4, a busca pela vizinhança de um vértice responderia com dados tanto de t_{n+1}, para vizinhos já atualizados, quanto de t_n.

Algoritmo 3: Cálculo da força resultante sobre um vértice v_i

$externa \longleftarrow calculaForçaExterna();$
for $v_j \longleftarrow G[\ v_i.info.vizinhosDiretos[j]\ ]$ **do**
  $elástica \longleftarrow elástica + calculaForçaElástica(v_i, v_j);$
  $amortecedora \longleftarrow amortecedora + calculaForçaAmortecedora(v_i, v_j);$
end for
for $v_k \longleftarrow G[\ v_i.info.vizinhosTransversais[k]\ ]$ **do**
  $elástica \longleftarrow elástica + calculaForçaElástica(v_i, v_k);$
  $amortecedora \longleftarrow amortecedora + calculaForçaAmortecedora(v_i, v_k);$
end for
return $(externa + elástica + amortecedora);$

As fórmulas utilizada para calcular as forças elástica e amortecedora podem ser vistas nas equações 3.3 e 3.6, respectivamente. Para fins de consulta, o código completo está transcrito no Apêndice B deste trabalho.

Capítulo 6

Resultados

Esta seção apresentará resultados de alguns testes realizados sobre o aplicativo criado para este trabalho. Alguns parâmetros de configuração, listados na tabela 6.1, foram mantidos constantes para todos casos testados.

Em todos os testes, foram realizadas 1000 iterações do método de atualização de posições da malha.

Tabela 6.1: Parâmetros de configuração constantes em todos os testes.

Arquivo	Parâmetro	Valor
ClothAppSettings	cmap	MAP_SPRING
	cmapMin	-10.0
	cmapMax	10.0
	rmode	*Smooth*
	clothScale	1.0
ClothSettings	k	5
	m	0.001
	g	9.789
	d	5
	forces	$NONE$

6.1 Testes sem colisão

Nestes testes, foi testada a queda do tecido sob o efeito da força da gravidade, preso por alguns pontos específicos, definidos pelo parâmetro *anchors* de *ClothSettings.settings*. Em todos os testes sem colisão, foram usados os parâmetros descritos abaixo, na tabela 6.2.

Tabela 6.2: Parâmetros de configuração constantes nos testes sem colisão.

Arquivo	Parâmetro	Valor
ClothAppSettings	mfile	$./meshes/test.off$
	collision	$NONE$
	collisionCenter	$(0.0, 0.0, 0.0)$
	collisionScale	1.0

É importante assinalar que, apesar de serem definidos parâmetros referentes à malha do objeto com o qual há colisão, eles serão desconsiderados, já que o valor $NONE$ para o parâmetro *collision* indica que não há modelo 3D a ser renderizado para esta finalidade.

A Figura 6.1 mostra o estado inicial da simulação nos testes sem colisão.

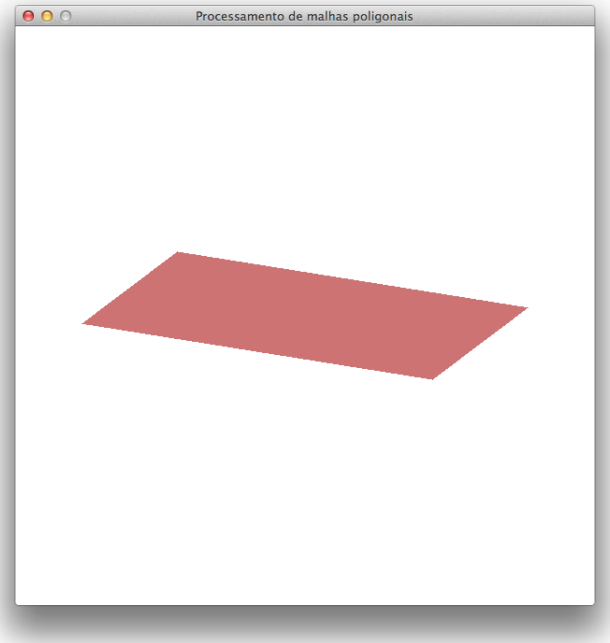


Figura 6.1: Estado inicial dos testes sem colisão.

6.1.1 Teste 1: tecido preso pelos quatro cantos

Neste teste, os quatro cantos do tecido foram definidos como âncoras, deixando ele cair sob o seu próprio peso. Os parâmetros deste teste estão descritos na tabela 6.3.

Tabela 6.3: Parâmetros de configuração do teste 1.

Arquivo	Parâmetro	Valor
ClothAppSettings	clothCenter	$(0.0, 0.0, 0.0)$
ClothSettings	anchors	$0, 20, 630, 650.$

O estado final do teste está ilustrado na Figura 6.2.

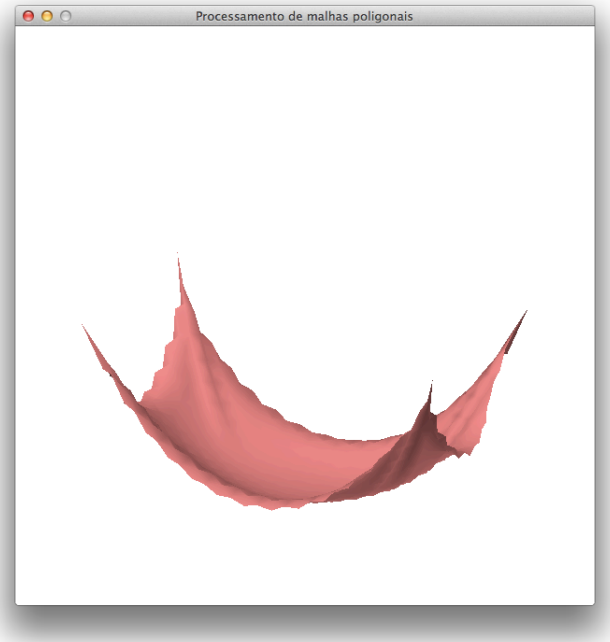


Figura 6.2: Resultado do teste 1.

6.1.2 Teste 2: tecido preso por quatro pontos internos

Para este teste, foram definidos como âncoras quatro pontos em seu interior, formando um quadrado. Novamente, a animação representou a ação da força da gravidade sobre o tecido. Os parâmetros modificados para este teste estão representados tabela 6.4.

Tabela 6.4: Parâmetros de configuração do teste 2.

Arquivo	Parâmetro	Valor
ClothAppSettings	clothCenter	$(0.0, 0.0, 0.0)$
ClothSettings	anchors	$215, 226, 446, 457.$

Após os 1000 *frames*, o estado final da simulação foi o que se segue, ilustrado na Figura 6.3.

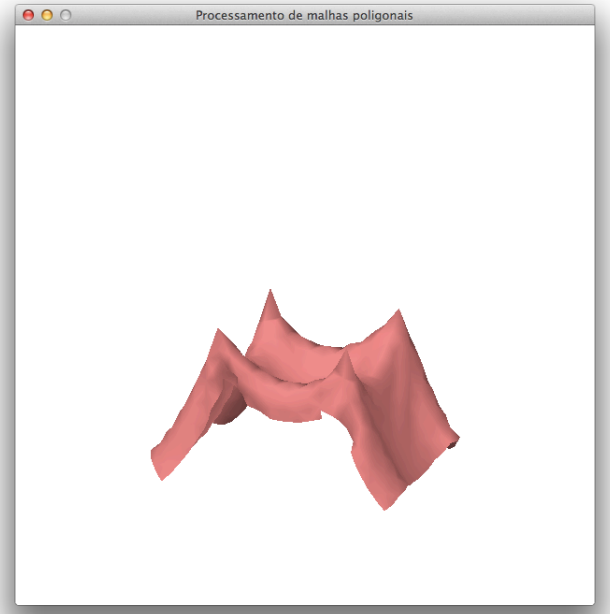


Figura 6.3: Resultado do teste 2.

6.1.3 Teste 3: tecido preso por um de seus lados

Neste teste, os vértices ao longo de um dos lados do tecido foram definidos como âncoras, para que este caísse como uma bandeira. Seus parâmetros específicos estão listados na tabela 6.5 e seu resultado é visto na Figura 6.4.

Tabela 6.5: Parâmetros de configuração do teste 3.

Arquivo	Parâmetro	Valor
ClothAppSettings	clothCenter	$(0.5, 0.0, 0.0)$
ClothSettings	anchors	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20.$

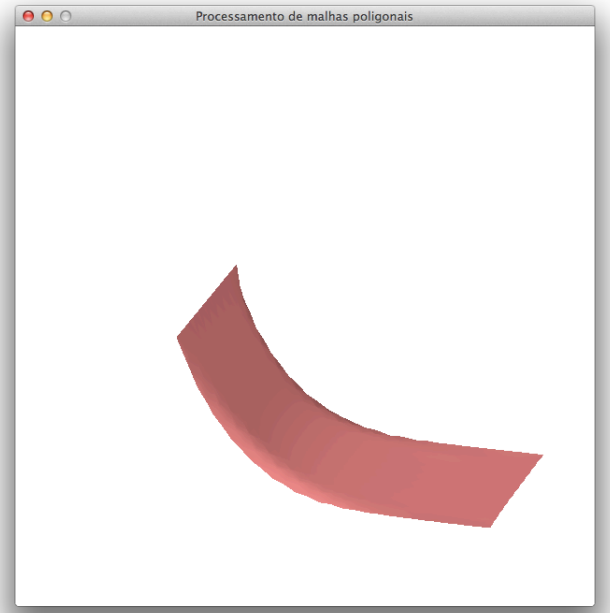


Figura 6.4: Resultado do teste 3.

6.2 Testes com colisão

Nestes testes, foi verificada a interação do tecido com outros objetos na cena. Como a colisão tecido-obstáculo foge ao escopo deste trabalho, o algoritmo de tratamento de colisão utilizado a fim de testes foi simples, sem tratamento de resposta.

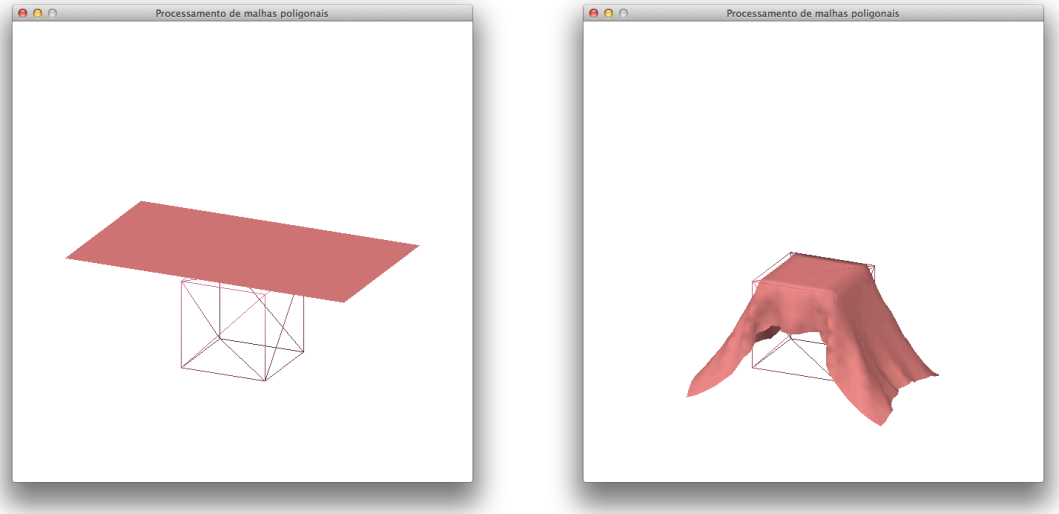
Em todos os testes com colisão, o valor do parâmetro *collision* do arquivo *ClothAppSettings.settings* foi mantido como $./meshes/cubo.off$, endereço do arquivo *.off* referente ao modelo 3D de um cubo.

6.2.1 Teste 4: tecido caindo sobre um cubo

Neste teste, o tecido foi deixado livre, sem âncoras, para cair sobre um cubo. Seus parâmetros estão definidos abaixo, na tabela 6.6, seu estado inicial na Figura 6.5a.

Tabela 6.6: Parâmetros de configuração do teste 4.

Arquivo	Parâmetro	Valor
ClothAppSettings	mfile	$./meshes/test.off$
	clothCenter	$(0.0, 0.0, 0.0)$
	collisionScale	0.3
	collisionCenter	$(0.0, -1.5, 0.0)$
ClothSettings	anchors	$NONE$.



(a) Estado inicial.

(b) Estado final.

Figura 6.5: Estados do teste 4.

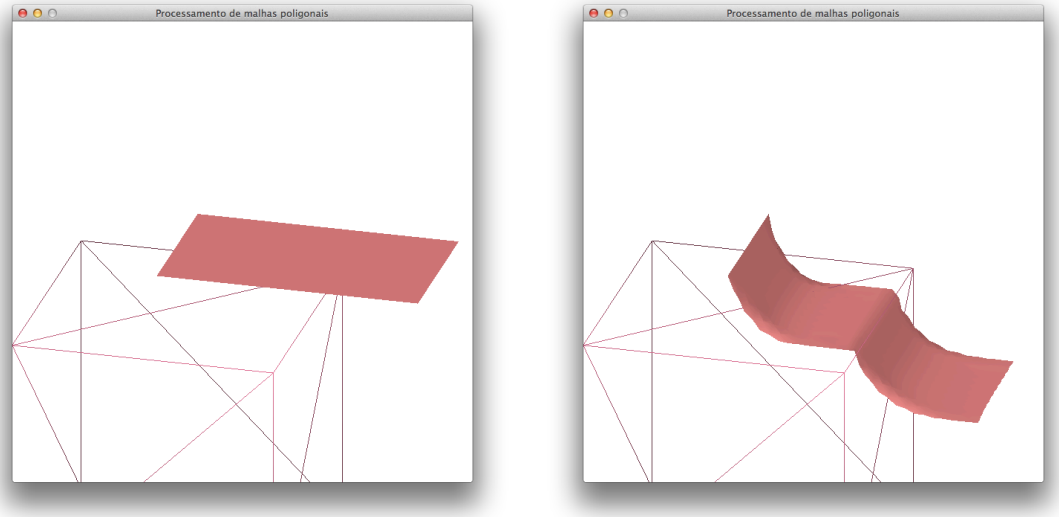
6.2.2 Teste 5: tecido preso por um de seus lados caindo sobre um cubo

Este teste assemelha-se bastante com o teste 3, porém, neste caso há a colisão com o cubo. Os valores de seus parâmetros estão abaixo, na tabela 6.7.

Tabela 6.7: Parâmetros de configuração do teste 5.

Arquivo	Parâmetro	Valor
ClothAppSettings	mfile	$./meshes/test.off$
	clothCenter	$(0.5, 0.0, 0.0)$
	collisionScale	1.0
	collisionCenter	$(0.0, -0.5, 0.0)$
ClothSettings	anchors	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20.$

O estado inicial e o resultado da simulação do teste 5 estão abaixo, nas Figuras 6.6a e 6.6b.



(a) Estado inicial.

(b) Estado final.

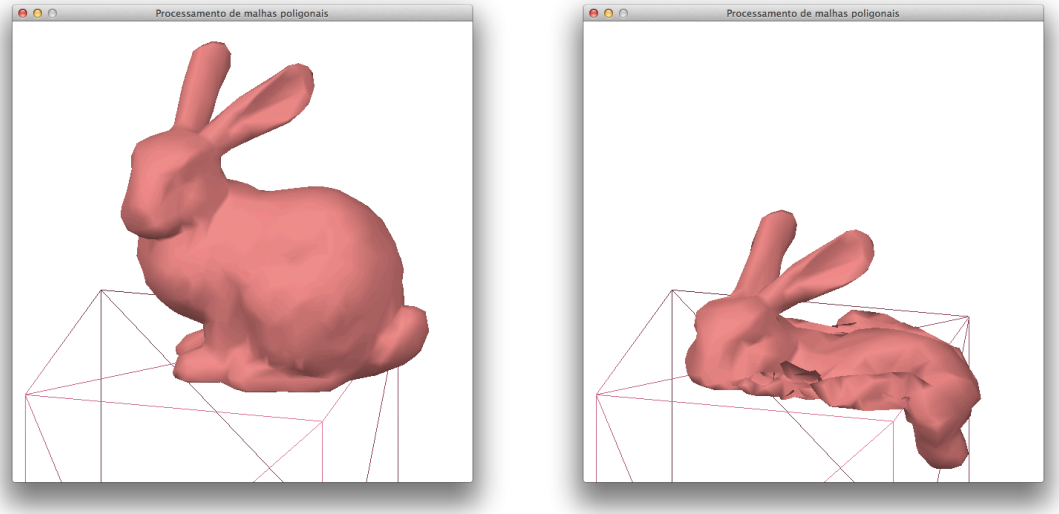
Figura 6.6: Estados do teste 5.

6.2.3 Teste 6: coelho de tecido caindo sobre um cubo

Este teste foi realizado para mostrar que qualquer malha de triângulos pode ser utilizada pela aplicação como o tecido a ser simulado. Para este teste, cujos parâmetros estão descritos na tabela 6.8 e os estados ilustrados na Figura 6.7, foi utilizado o modelo de um coelho.

Tabela 6.8: Parâmetros de configuração do teste 6.

Arquivo	Parâmetro	Valor
ClothAppSettings	mfile	$./meshes/bunny_tri.off$
	clothCenter	$(0.5, 0.0, 0.0)$
	collisionScale	1.0
	collisionCenter	$(0.0, -2.0, 0.0)$
ClothSettings	anchors	$NONE$.



(a) Estado inicial.

(b) Estado final.

Figura 6.7: Estados do teste 6.

É importante notar que, na Figura 6.7b, o estado final da simulação não apresenta um resultado ideal devido a falta de tratamento da auto-colisão dos vértices do tecido, fazendo com que a parte superior do tecido atravesse a inferior após sua colisão com o cubo.

6.3 Tempos de execução dos testes

Para cada um dos testes descritos nas seções 6.1 e 6.2, foi medido o tempo médio de recálculo das posições dos vértices e o tempo total e simulação. Estes estão listados abaixo, na tabela 6.9.

Tabela 6.9: Tempos de execução médio por *frame* e total dos testes 1 a 6.

Teste	Tempo médio de cada *frame*	Tempo total da simulação
1	$0.0050s$	$7.9966s$
2	$0.0059s$	$8.6257s$
3	$0.0092s$	$12.0535s$
4	$0.0090s$	$12.3075s$
5	$0.0100s$	$13.1631s$
6	$0.0144s$	$17.7502s$

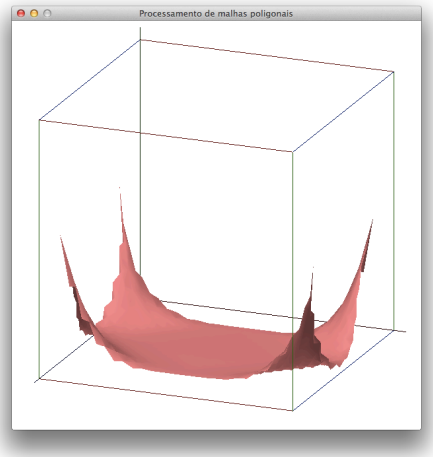
É possível verificar que os testes 4, 5 e 6 tiveram os maiores tempos totais de execução, o que se deve ao tratamento da colisão que foi necessário nos mesmos. O teste 6 obteve os maiores tempos de execução devido ao número maior de vértices no modelo 3D do coelho.

6.4 Análise de valores das constantes físicas

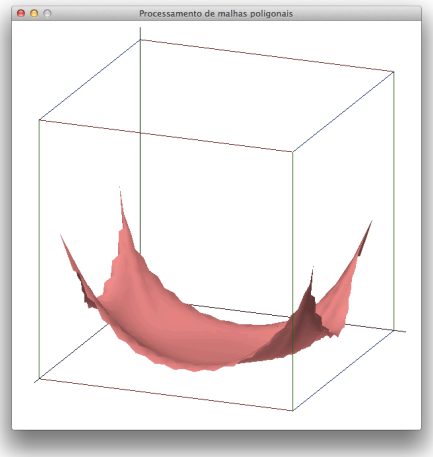
Através da manipulação dos valores da constante elástica k e da massa m, é possível modelar diferentes tipos de tecido. Para verificar esta diferença, foi realizado novamente o teste 1, cujos parâmetros estão listados nas tabelas 6.1 e 6.2. Foram feitas variações, porém, nos parâmetros k e m do arquivo *ClothSettings.settings* para analisar seu efeito sobre a simulação. Estes testes estão descritos nas seções abaixo.

6.4.1 Constante elástica

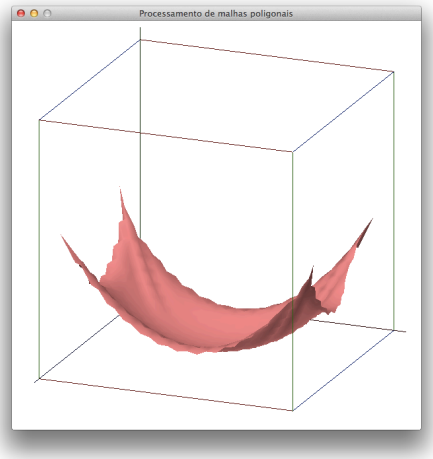
Para este teste, foi variado a valor da constante elástica k, responsável por definir a flexibilidade do tecido, mantendo o valor da massa em 0.001. Quanto menor for seu valor, maior será o comprimento que as molas serão capazes de atingir. A Figura 6.8 apresenta os valores adotados nos testes e seus respectivos resultados.



(a) $k = 1$



(b) $k = 5$.

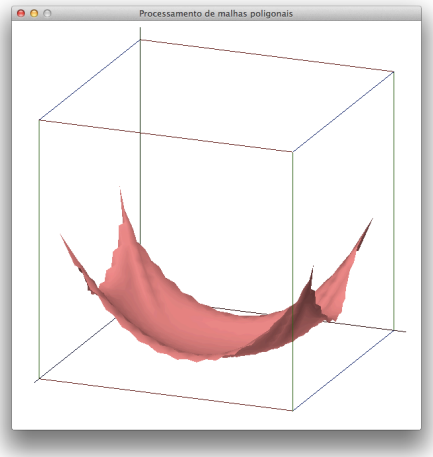


(c) $k = 15$.

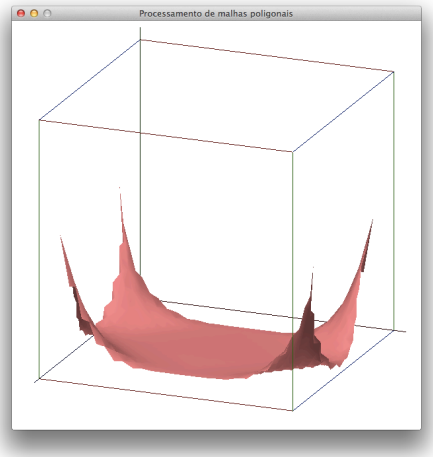
Figura 6.8: Resultados da variação do valor da constante elástica.

6.4.2 Massa

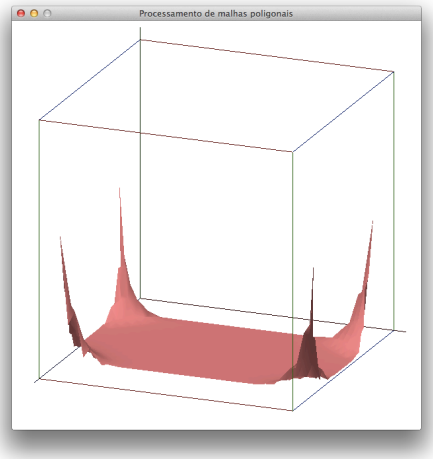
Foram realizados novamente três testes, desta vez fixando o valor da constante elástica em 5. Os valores atribuídos às massas em cada teste e seus resultados são apresentados na Figura 6.9.



(a) $m = 0.0005$



(b) $m = 0.005$.



(c) $m = 0.05$.

Figura 6.9: Resultados da variação do valor da massa.

É possível verificar que o aumento do valor da massa fez com que o tecido se esticasse mais nas extremidades, como se houvesse um peso sobre seu centro. Na Figura 6.9c, o centro do tecido está plano pois, devido à alta massa, após 1000 *frames* da animação a aceleração gerada pela força elástica ainda não havia sido suficiente para balancear a ação da gravidade além das extremidades do tecido, fazendo com que sua porção central continuasse em queda livre.

Capítulo 7

Conclusão e trabalhos futuros

Este trabalho possibilitou um estudo sobre a utilização de uma estrutura de dados que facilita o acesso a dados topológicos de malhas para tornar mais eficiente a simulação de tecidos através do modelo massa-mola, notável por sua simplicidade. O uso da CHE tornou menos complexa a tarefa de busca pela vizinhança de um vértice, que é realizada com frequência para realizar o recálculo das posições dos pontos. Esta alternativa acarretou em um gasto maior de memória para o armazenamento das estruturas dos diferentes níveis das CHEs, mas, considerando as maiores quantidades de memória principal disponíveis nos computadores atuais e a simplicidade das malhas utilizadas no trabalho, este acréscimo no consumo de RAM não foi problemático para a aplicação.

Conforme dito na seção 2.3, novas técnicas de simulação de tecidos mais sofisticadas foram propostas recentemente, se utilizando novas tecnologias, princípios físicos e matemáticos avançados e algoritmos mais complexos que permitem simulações melhores, com maior grau de realismo e/ou menor tempo de renderização de *frames*. Devido a esta defasagem do modelo massa-mola devido a sua simplicidade, continuar a desenvolvê-lo em trabalhos futuros pode não ser uma boa abordagem. Em próximos trabalhos na área de simulação de tecidos, será buscada e pesquisada uma técnica alternativa mais robusta, capaz de gerar animações de maior qualidade gráfica.

Referências Bibliográficas

[1] J. Bender and C. Deul. Efficient cloth simulation using an adaptive finite element method. In *Workshop on Virtual Reality Interaction and Physical Simulation*, pages 21–30, 2012.

[2] D. Breen and D. House. A physically based particle method of woven cloth. In *The Visual Computer*, 1992.

[3] D. Breen, D. House, and P. Getto. On the dynamic simulation of physically-based particle system models. In *Third Eurographics Workshop on Animation and Simulation Proceedings*, 1992.

[4] L. de Floriani and A. Hui. A scalable data structure for three-dimensinal non-manifold objects. In *Symposium on Geometry Processing*, pages 72–82, 2003.

[5] B. Eberhardt, A. Weber, and W. Strasser. A fast, flexible particle system model for cloth draping. In *IEEE Computer Graphics and Applications*, 1996.

[6] M. Fürer. Faster integer multiplication. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, San Diego, California, USA, June 2007.

[7] D. Haumann and R. Parent. The behavioural test bed. obtaining complex behaviour from simple rules. In *The Visual Computer*, 1998.

[8] D. House and D. Breen. Cloth modelling and animation, 2000.

[9] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. In *Proceedings of the USSR Academy of Sciences*, number 145, pages 293–294, 1962.

[10] B. Kevelham and N. Magnenat-Thalmann. Fast and accurate gpu-based simulation of virtual garments. In *VRCAI '12 Proceedings of the 11th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry*, pages 223–226, 2012.

[11] T. Kunii and H. Gotoda. Singularity, theoretical modelling and animation of garment wrinkle formation process. In *The Visual Computer*, 1990.

[12] M. Lage, T. Lewiner, H. Lopes, and L. Velho. *CHE: A scalable topological data structure for triangular meshes*. PhD thesis, Pontifícia Universidade Católica Rio de Janeiro,, May 2005.

[13] L. Liu. A cloth simulation method based on differential geometric energy. In *Fourth International Conference on Digital Home (ICDH)*, pages 1–6, November 2012.

[14] H. Lopes and G. Tavares. Structural operators for modeling 3-manifolds. ACM, 1997.

[15] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[16] R. Narain, A. Samii, and J. O'Brien. Adaptive anisotropic remeshing for cloth simulation. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2012*, 31(152), November 2012.

[17] X. Provot. Deformation constraints in a mass spring model to describe cloth behavior. In *Proceedings Graphics Interface*.

[18] J. Rossignac, A. Safonova, and A. Szymczak. 3d compression made simple: Edgebreaker on a corner-table. IEEE, 2001.

[19] I. Rudomin. *Simulating cloth using a mixed Geometric Physical method*. PhD thesis, University of Pennsylvania.

[20] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. In *Computing*, number 7, pages 281–292, 1971.

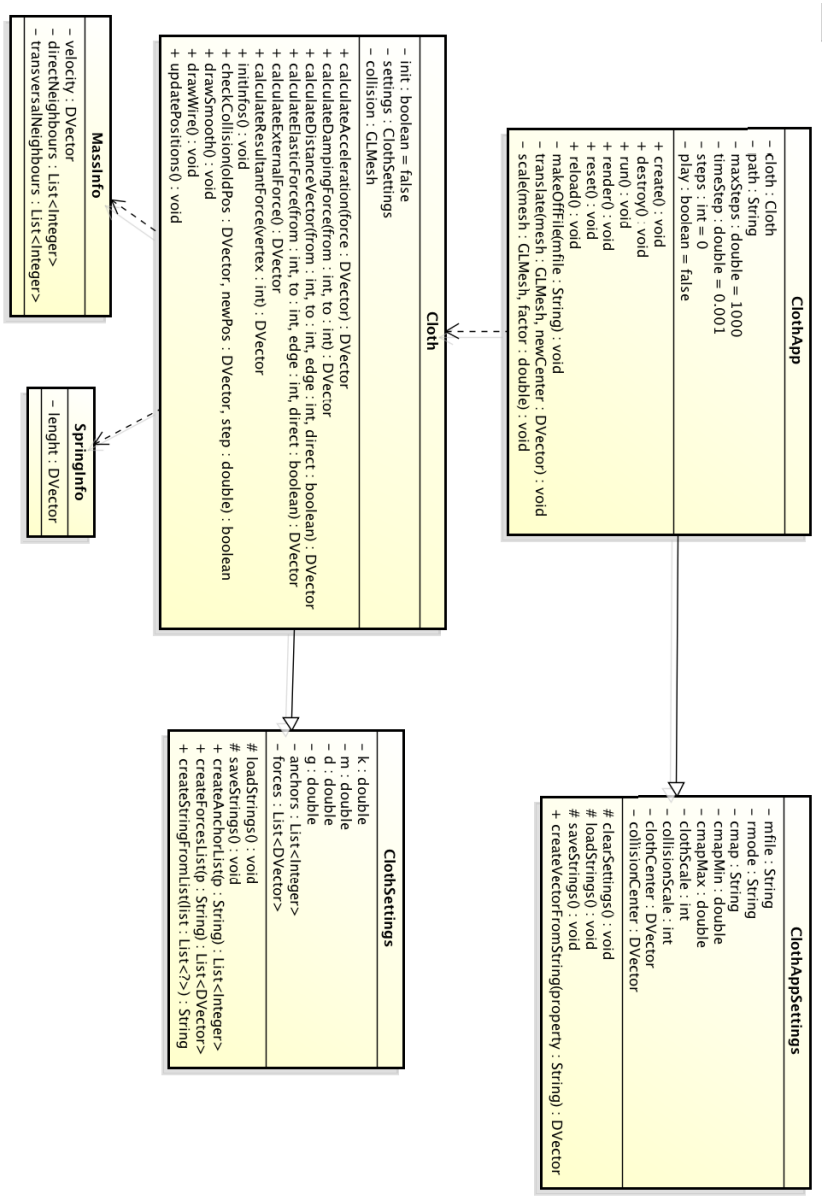
[21] A. Selle, J. Su, G. Irving, and R. Fedkiw. Robust high-resoution cloth using parallelism, history-based collisions and accurate friction. In *IEEE Transactions. on Visualization and Computer. Graphics*, 2007.

[22] C. Taylor. Cinema: A painstaking fantasy. Time Magazine, July 2000.

[23] J. Weil. The synthesis of cloth objects. In *Computer Graphics Proceedings (SIGGRAPH)*, 1986.

Apêndice A

Diagrama de classes



Apêndice B

Código

B.1 Classe *ClothAppSettings*

```
package ic.apps.cloth;

import ic.mlibs.linalg.DVector;
import ic.mlibs.util.Settings;

class ClothAppSettings extends Settings{

    final static int X = 0;
    final static int Y = 1;
    final static int Z = 2;

    private String mfile;
    private String rmode;
    private String  cmap;
    private String  collision;

    private double cmapMin;
    private double cmapMax;
    private double clothScale;
    private double collisionScale;

    private DVector clothCenter;
    private DVector collisionCenter;
```

```
public ClothAppSettings(String mfile, String rmode, String cmap, double cmapMin,
double cmapMax, double clothScale, double collisionScale, String arq,
String collision, DVector clothCenter, DVector collisionCenter) {
    super(arq);
    this.mfile = mfile;
    this.rmode = rmode;
    this.cmap = cmap;
    this.collision = collision;
    this.cmapMin = cmapMin;
    this.cmapMax = cmapMax;
    this.clothScale = clothScale;
    this.collisionScale = collisionScale;
    this.clothCenter = clothCenter;
    this.collisionCenter = collisionCenter;
}

public ClothAppSettings(String fileName){
    super(fileName);
}

public String getMfile() {
    return mfile;
}

public String getRmode() {
    return rmode;
}

public String getCmap()  {
    return cmap;
}

public double getCmapMin() {
    return cmapMin;
}

public double getCmapMax() {
    return cmapMax;
}
```

```
public String getCollision() {
    return collision;
}

public double getClothScale() {
    return clothScale;
}

public void setClothScale(double clothScale) {
    this.clothScale = clothScale;
}

public double getCollisionScale() {
    return collisionScale;
}

public void setCollisionScale(double collisionScale) {
    this.collisionScale = collisionScale;
}

public DVector getClothCenter() {
    return clothCenter;
}

public void setClothCenter(DVector clothCenter) {
    this.clothCenter = clothCenter;
}

public DVector getCollisionCenter() {
    return collisionCenter;
}

public void setCollisionCenter(DVector collisionCenter) {
    this.collisionCenter = collisionCenter;
}

 public void setCmapMax(double cmapMax) {
    this.cmapMax = cmapMax;
```

```
}

public void setCmapMin(double cmapMin) {
    this.cmapMin = cmapMin;
}

@Override
protected void clearSettings() {
    mfile = null;
    collision = null;
    rmode   = null;
    cmap    = null;
    cmapMin = 0.0f;
    cmapMax = 0.0f;
    super.clearSettings();
}

@Override
protected void loadStrings() {
    mfile = settings.getProperty("mfile");
    collision = settings.getProperty("collision");
    rmode = settings.getProperty("rmode");
    cmap = settings.getProperty("cmap");
    cmapMin = Double.parseDouble(settings.getProperty("cmapMin"));
    cmapMax = Double.parseDouble(settings.getProperty("cmapMax"));
    clothScale = Double.parseDouble(settings.getProperty("clothScale"));
    collisionScale = Double.parseDouble(settings.getProperty("collisionScale"));
    clothCenter = createVectorFromString(settings.getProperty("clothCenter"));
    collisionCenter =
    createVectorFromString(settings.getProperty("collisionCenter"));
}

@Override
protected void saveStrings() {
    settings.setProperty("mfile" , mfile);
    settings.setProperty("collision", collision);
    settings.setProperty("rmode", rmode);
    settings.setProperty("cmap", cmap);
    settings.setProperty("cmapMin", Double.toString(cmapMin));
```

```
        settings.setProperty("cmapMax", Double.toString(cmapMax));
        settings.setProperty("clothScale", Double.toString(clothScale));
        settings.setProperty("collisionScale", Double.toString(collisionScale));
        settings.setProperty("clothCenter", clothCenter.toString());
        settings.setProperty("collisionCenter", collisionCenter.toString());
    }

    public DVector createVectorFromString (String property){
        DVector result = new DVector(0.0f, 0.0f, 0.0f);
        property = property.replace(" ", "");
        property = property.replace("(", "");
        property = property.replace(")", "");
        String[] forcesArray = property.split(",");
        if (forcesArray.length % 3 != 0) {
            throw new IllegalArgumentException
            ("One of the vectors is missing components");
        } else {
            for (int i = 0; i < forcesArray.length; i++) {
                switch (i % 3) {
                    case X:
                        result.setVecData(Double.parseDouble(forcesArray[i]), X);
                        break;
                    case Y:
                        result.setVecData(Double.parseDouble(forcesArray[i]), Y);
                        break;
                    case Z:
                        result.setVecData(Double.parseDouble(forcesArray[i]), Z);
                        break;
                }
            }
        }
        return result;
    }
}
```

B.2 Classe *Cloth*

```
package ic.apps.cloth;
```

```
import com.sun.tools.javac.util.Pair;
import ic.mlibs.gl.GLMesh;
import ic.mlibs.linalg.DVector;
import ic.mlibs.structures.Point3D;
import ic.mlibs.util.Helper;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import static org.lwjgl.opengl.GL11.*;

public class Cloth extends GLMesh {

    final static int X = 0;
    final static int Y = 1;
    final static int Z = 2;
    private boolean init = false;
    private ClothSettings settings;
    private GLMesh collision;

    public void setCollision(GLMesh collision) {
        this.collision = collision;
    }

    public Cloth(String path) {

        String newPath = path;
        int slashIndex = newPath.lastIndexOf("/");
        newPath = newPath.substring(0, slashIndex + 1);
        newPath = newPath.concat("ClothSettings.settings");

        this.settings = new ClothSettings(newPath);

        settings.getForces().add(new DVector(0.0f,
        (-1.0f) * settings.getG() * settings.getM(), 0.0f));

        this.collision = null;
    }

    public GLMesh getCollision() {
```

```
    return collision;
}

public boolean isInit() {
    return init;
}

public DVector calculateExternalForce() {

    DVector result = new DVector(0.0f, 0.0f, 0.0f);

    for (int i = 0; i < settings.getForces().size(); i++) {
        result = result.opAdd(settings.getForces().get(i));
    }
    return result;
}

public DVector calculateDistanceVector(int from, int to, int edge, boolean direct) {
    Point3D a = getG(to);
    Point3D b = getG(from);

    double initialLenght = 0;
    Pair<SpringInfo, SpringInfo> pair = (Pair<SpringInfo, SpringInfo>) getEH(edge);

    if (direct){
        initialLenght = pair.fst.getLenght();
    } else if(pair.snd != null){
        initialLenght = pair.snd.getLenght();
    }

    DVector currentLenght = a.getPos().opSub(b.getPos());
    double currLenNorm = currentLenght.norm();

    currentLenght = currentLenght.opScale((float) currentLenght.normalize());
    currentLenght = currentLenght.opScale( (float)(currLenNorm - initialLenght) );

    return currentLenght;
}
```

```
public DVector calculateElasticForce(int from, int to, int edge, boolean direct) {

    DVector distanceVector = calculateDistanceVector(to, from, edge, direct);
    return distanceVector.opScale( (float)(-1.0f*settings.getK() ) );
}

public DVector calculateDampingForce(int from, int to) {

    MassInfo vertexMassInfo = (MassInfo)getG(from).getObj();
    MassInfo  otherMassInfo = (MassInfo)getG(  to).getObj();

    DVector vertexVelocity = vertexMassInfo.getVelocity();
    DVector  otherVelocity =  otherMassInfo.getVelocity();

    DVector result = otherVelocity.opSub(vertexVelocity);
    result = result.opScale((float)(-1.0f*settings.getD() ) );

    return result;
}

public DVector calculateResultantForce(int vertex) {

    DVector result = calculateExternalForce();
    MassInfo massInfo = (MassInfo) getG(vertex).getObj();
    DVector totalElastic = new DVector(0.0f, 0.0f, 0.0f);
    DVector totalDamping = new DVector(0.0f, 0.0f, 0.0f);

    for (int i = 0; i < massInfo.getDirectNeighbours().size(); i++) {

        int neighbour = massInfo.getDirectNeighbours().get(i);
        int otherVertex = getV( next(neighbour) );

        DVector dampingForce = calculateDampingForce(vertex, otherVertex);
        DVector elasticForce =
        calculateElasticForce(vertex, otherVertex, neighbour, true);

        DVector partial = elasticForce.opAdd(dampingForce);
        result = result.opAdd(partial);
    }
```

```
    for (int i = 0; i < massInfo.getTransversalNeighbours().size(); i++) {

        int neighbour = massInfo.getTransversalNeighbours().get(i);
        int otherVertex = getV( prev(neighbour) );

        DVector dampingForce = calculateDampingForce(vertex,otherVertex);
        DVector elasticForce =
        calculateElasticForce(vertex,otherVertex, neighbour, false);

        DVector partial = elasticForce.opAdd(dampingForce);
        result = result.opAdd(partial);
    }

    return result;
}

public DVector calculateAcceleration(DVector force) {

    float factor = (float) (1.0f / settings.getM());
    return force.opScale(factor);

}

public void initInfos() {
    for (int i = 0; i < getNvert(); i++) {
        MassInfo massInfo = new MassInfo();
        List<Integer> edges = R_01(i);
        massInfo.getDirectNeighbours().addAll(edges);
        getG(i).setObj(massInfo);
        DVector posi = getG(i).getPos();
        for (int k = 0; k < edges.size(); k++) {
            SpringInfo edge = null;
            SpringInfo tran = null;
            int heID = edges.get(k);
            int trId = getO(next(heID));
            if( trId > 0){
                massInfo.getTransversalNeighbours().add( trId );
            }
```

```
            if( getEH( heID ).getClass() == SpringInfo.class){
                continue;
            }
            DVector posk = getG( getV(next(heID)) ).getPos();
            edge = new SpringInfo(posk.opSub(posi).norm());
            int oppHeID = getO(heID);
            if( oppHeID != -1 ) { //Não é aresta de bordo.
                DVector posl = getG(getV(prev(   heID))).getPos();
                DVector poso = getG(getV(prev(oppHeID))).getPos();
                tran = new SpringInfo(posl.opSub(poso).norm());
            }
            setEH( heID, new Pair<SpringInfo, SpringInfo>(edge, tran) ) ;
        }
    }
    init = true;
}

public boolean checkCollision(DVector oldPos, DVector newPos, double step){
    for (int i = 0; i < collision.getNface(); i++){
        int he = collision.base(i);
        Point3D v1 = collision.getG(collision.getV(   he  ));
        Point3D v2 = collision.getG(collision.getV(next(he)));
        Point3D v3 = collision.getG(collision.getV(prev(he)));
        boolean res = Helper.triInterEdge(new Point3D(oldPos.getVecData()),
        new Point3D(newPos.getVecData()), v1, v2, v3);
        if( res ){
            return true;
        }
    }
    return false;
}

public void updatePositions(double currentTime, double stepSize) {
    ArrayList<Point3D> newPositions = new ArrayList<Point3D>();
    for (int i = 0; i < getNvert(); i++) {
        newPositions.add(i, getG(i));
        if (!settings.getAnchors().contains(i)) {
            MassInfo massInfo = (MassInfo) getG(i).getObj();
            DVector resultant = calculateResultantForce(i);
```

```
            DVector acceleration = calculateAcceleration(resultant);
            DVector velocity = new DVector( massInfo.getVelocity().getVecData() );
            DVector position = new DVector( getG(i).getPos().getVecData() );
            velocity = velocity.opAdd( acceleration.opScale( (float)stepSize) );
            position = position.opAdd( velocity.opScale( (float)stepSize) );
            if (collision != null){
                boolean colide = checkCollision(getG(i).getPos(), position, stepSize);
                if( colide ){
                    position = position.opAdd(getG(i).getPos());
                    position = position.opScale(0.5f);
                    settings.getAnchors().add(i);
                }
            }
            newPositions.get(i).setPos(position);
            newPositions.get(i).setObj(massInfo);
        }
    }
    G = newPositions;
}

@Override
public void drawWire() {
    glLineWidth(1.0f);
    glBegin(GL_LINES);
    {
        Iterator<Integer> i = EH.keySet().iterator();
        while(i.hasNext())
        {
          int he = i.next();
          Point3D v1 = getG( getV(    he    ));
          Point3D v2 = getG( getV( next(he) ));
          MassInfo info = (MassInfo) v1.getObj();
          map.setGLColor(info.getVelocity().norm(), (byte) 255, true );
          glNormal3d(v1.getNrmX(), v1.getNrmY(), v1.getNrmZ());
          glVertex3d(v1.getPosX(), v1.getPosY(), v1.getPosZ() );
          info = (MassInfo) v2.getObj();
          map.setGLColor(info.getVelocity().norm(), (byte) 255, true );
          glNormal3d(v2.getNrmX(), v2.getNrmY(), v2.getNrmZ());
          glVertex3d(v2.getPosX(), v2.getPosY(), v2.getPosZ() );
```

```
        }
    }
    glEnd();
}


@Override
public void drawSmooth() {
    glBegin(GL_TRIANGLES);
    {
        for (int i = 0; i < getNtrig(); i++) {
            int he = base(i);
            Point3D v0 = getG(getV(he++));
            MassInfo info = (MassInfo) v0.getObj();
            map.setGLColor(info.getVelocity().norm(), (byte) 255, true);
            glNormal3d(v0.getNrmX(), v0.getNrmY(), v0.getNrmZ());
            glVertex3d(v0.getPosX(), v0.getPosY(), v0.getPosZ());
            Point3D v1 = getG(getV(he++));
            info = (MassInfo) v1.getObj();
            map.setGLColor(info.getVelocity().norm(), (byte) 255, true);
            glNormal3d(v1.getNrmX(), v1.getNrmY(), v1.getNrmZ());
            glVertex3d(v1.getPosX(), v1.getPosY(), v1.getPosZ());
            Point3D v2 = getG(getV(he++));
            info = (MassInfo) v2.getObj();
            map.setGLColor(info.getVelocity().norm(), (byte) 255, true);
            glNormal3d(v2.getNrmX(), v2.getNrmY(), v2.getNrmZ());
            glVertex3d(v2.getPosX(), v2.getPosY(), v2.getPosZ());
        }
    }
    glEnd();

    glBegin(GL_QUADS);
    {
        for (int i = getNtrig(); i < getNface(); i++) {
            int he = base(i);
            Point3D v0 = getG(getV(he++));
            MassInfo info = (MassInfo) v0.getObj();
            map.setGLColor(info.getVelocity().norm(), (byte) 255, true);
            glNormal3d(v0.getNrmX(), v0.getNrmY(), v0.getNrmZ());
```

```
                glVertex3d(v0.getPosX(), v0.getPosY(), v0.getPosZ());
                Point3D v1 = getG(getV(he++));
                info = (MassInfo) v1.getObj();
                map.setGLColor(info.getVelocity().norm(), (byte) 255, true);
                glNormal3d(v1.getNrmX(), v1.getNrmY(), v1.getNrmZ());
                glVertex3d(v1.getPosX(), v1.getPosY(), v1.getPosZ());
                Point3D v2 = getG(getV(he++));
                info = (MassInfo) v2.getObj();
                map.setGLColor(info.getVelocity().norm(), (byte) 255, true);
                glNormal3d(v2.getNrmX(), v2.getNrmY(), v2.getNrmZ());
                glVertex3d(v2.getPosX(), v2.getPosY(), v2.getPosZ());
                Point3D v3 = getG(getV(he++));
                info = (MassInfo) v3.getObj();
                map.setGLColor(info.getVelocity().norm(), (byte) 255, true);
                glNormal3d(v3.getNrmX(), v3.getNrmY(), v3.getNrmZ());
                glVertex3d(v3.getPosX(), v3.getPosY(), v3.getPosZ());
            }
        }
        glEnd();
    }
}
```

B.3 Classe *ClothSettings*

```
package ic.apps.cloth;

import ic.mlibs.linalg.DVector;
import ic.mlibs.util.Settings;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author yanramos
 */
public class ClothSettings extends Settings{
    final static int X = 0;
    final static int Y = 1;
    final static int Z = 2;
```

```
private double k;
private double m;
private double d;
private double g;

private List<Integer> anchors;  // Índices dos vértices imóveis
private List<DVector> forces;  // Índices das forças externas

public ClothSettings(String arq){
    super(arq);
}

public double getK() {
    return k;
}

public void setK(double k) {
    this.k = k;
}

public double getM() {
    return m;
}

public void setM(double m) {
    this.m = m;
}

public double getD() {
    return d;
}

public void setD(double d) {
    this.d = d;
}

public double getG() {
    return g;
```

```
}

public void setG(double g) {
    this.g = g;
}

public List<Integer> getAnchors() {
    return anchors;
}

public void setAnchors(List<Integer> anchors) {
    this.anchors = anchors;
}

public List<DVector> getForces() {
    return forces;
}

public void setForces(List<DVector> forces) {
    this.forces = forces;
}

public List<Integer> createAnchorsList(String p){
    String property = p;
    List<Integer> result = new ArrayList<Integer>();
    if (!property.equalsIgnoreCase("NONE")){
        property = property.replace(" ", "");
        String[] anchorsArray = property.split(",");
        for (int i = 0; i < anchorsArray.length; i++){
            result.add(Integer.parseInt(anchorsArray[i].trim()));
        }
    }
    return result;
}

public List<DVector> createForcesList(String p){
    String property = p;
    List<DVector> result = new ArrayList<DVector>();
    if (!property.equalsIgnoreCase("NONE")){
```

```
        DVector force = new DVector(0.0, 0.0, 0.0);
        property = property.replace(" ", "");
        property = property.replace("(", "");
        property = property.replace(")", "");
        String[] forcesArray = property.split(",");
        if (forcesArray.length % 3 != 0){
            throw new IllegalArgumentException
            ("One of the vectors is missing components");
        } else {
            for (int i = 0; i < forcesArray.length; i++){
                switch (i%3){
                    case X:
                        force.setVecData(Double.parseDouble(forcesArray[i]), X);
                        break;
                    case Y:
                        force.setVecData(Double.parseDouble(forcesArray[i]), Y);
                        break;
                    case Z:
                        force.setVecData(Double.parseDouble(forcesArray[i]), Z);
                        result.add(force);
                        break;
                }
            }
        }
    }
    return result;
}

public String createStringFromList (List<?> list){
    String result = "";
    if (list.isEmpty()){
        result = "NONE";
    } else {
        for (int i = 0; i < list.size(); i++){
            result = result.concat(list.get(i).toString());
            if (i != list.size() - 1){
                result = result.concat(", ");
            }
        }
```

```
        }
        return result;
    }

    @Override
    protected void loadStrings() {
        k = Double.parseDouble(settings.getProperty("k"));
        m = Double.parseDouble(settings.getProperty("m"));
        d = Double.parseDouble(settings.getProperty("d"));
        g = Double.parseDouble(settings.getProperty("g"));
        anchors = createAnchorsList(settings.getProperty("anchors"));
        forces = createForcesList(settings.getProperty("forces"));
    }

    @Override
    protected void saveStrings() {
        settings.setProperty("k", Double.toString(k));
        settings.setProperty("m", Double.toString(m));
        settings.setProperty("d", Double.toString(d));
        settings.setProperty("g", Double.toString(g));
        settings.setProperty("anchors", createStringFromList(anchors));
        settings.setProperty("forces", createStringFromList(forces));
    }
}
```

B.4 Classe *MassInfo*

```
package ic.apps.cloth;

import ic.mlibs.linalg.DVector;
import java.util.ArrayList;
import java.util.List;

public class MassInfo {

    private DVector velocity;
    private List<Integer> directNeighbours;
    private List<Integer> transversalNeighbours;
```

```
    public MassInfo(){
        this.velocity = new DVector(0.0f, 0.0f, 0.0f);
        this.directNeighbours = new ArrayList<Integer>();
        this.transversalNeighbours = new ArrayList<Integer>();
    }

    public List<Integer> getTransversalNeighbours() {
        return transversalNeighbours;
    }

    public void setTransversalNeighbours(List<Integer> transversalNeighbours) {
        this.transversalNeighbours = transversalNeighbours;
    }

    public List<Integer> getDirectNeighbours() {
        return directNeighbours;
    }

    public void setDirectNeighbours(List<Integer> neighbours) {
        this.directNeighbours = neighbours;
    }

    public DVector getVelocity(){
        return this.velocity;
    }

    public void setVelocity(DVector v){
        this.velocity = v;
    }
}
```

B.5 Classe *SpringInfo*

```
package ic.apps.cloth;

public class SpringInfo {

    private double lenght;
```

```
    public SpringInfo(double lenght) {
        this.lenght = lenght;
    }

    public double getLenght() {
        return lenght;
    }

    public void setLenght(double lenght) {
        this.lenght = lenght;
    }
}
```