MARCELLO KERA

Ambiente Virtual Interativo com Colisão e Deformação de Objetos para Treinamento Médico Utilizando a API Java

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Hélio Pedrini
Co-orientadora: Prof. Dra. Fátima L. S. N. Marques

CURITIBA

2008

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AABB	*Axis-Aligned Bounding Boxes*	Caixas Envoltórias Alinhadas
ARV		Ambiente de Realidade Virtual
AV		Ambiente Virtual
BSP	*Binary Space Partitioning*	Particionamento do Espaço Binário
CPU	*Central Processing Unit*	Unidade de Processamento Central
FEM	*Finite Element Method*	Método de Elementos Finitos
FFD	*Free Form Deformation*	Deformação de Forma Livre
FPS	*Frames Per Second*	Quadros Por Segundo
GPU	*Graphics Processing Unit*	Unidade de Processamento Gráfico
HMD	*Head Mounted Display*	Óculos de Realidade Virtual
JOGL	*Java OpenGL*	
J3D	*Java 3D*	
LWJGL	*Lightweight Java Game Library*	Biblioteca para Jogos em Java
MS	*Mass Spring*	Massa-Mola
OBB	*Oriented Bounding Boxes*	Caixas Envoltórias Orientadas
OpenGL	*Open Graphics Library*	Biblioteca OpenGL
RV		Realidade Virtual
VRML	*Virtual Reality Modeling Language*	Linguagem de Modelagem para Realidade Virtual

LISTA DE FIGURAS

ABSTRACT

Medical procedure training can be benefited from the use of interactive virtual environments that realistically simulates the actions of the user. The enviroment should produce quick answers to the user such as object collision, deformation, motion limitation, or forces and vibrations when objects collide. This paper describes the development of a virtual environment prototype for medical training. Classes and methods are designed and implemented in the environment using Java programming language. Object collision and deformation methods are used to incorporate realism to the scene. The modeled objects are represented by polygonal meshes. The collision detection between objects are based on spatial hierarchical subdivision with octrees. Mass-spring deformation is used to simulate shape changes in the objects that collide. Experiments are conducted to demonstrate the effectiveness of the prototype.

Keywords: Virtual Environments; Medical Training; Collision Detection; Object Deformation.

RESUMO

O treinamento de procedimentos médicos pode ser beneficiado com o uso de ambientes virtuais interativos que simulam com realismo as ações do usuário. A simulação deve emitir respostas rápidas ao usuário relativas ao encontro de objetos, deformação, restrição de movimento ou mesmo produzir forças e vibrações. Este trabalho descreve a criação de um protótipo de ambiente virtual para treinamento médico. Classes e métodos são projetados e implementados no ambiente por meio da linguagem de programação Java. Métodos de colisão e de deformação de objetos são utilizados para incorporar realismo à cena, sendo itens complexos e dependentes das informações de interação monitoradas no ambiente virtual. Os objetos modelados são representados por malhas poligonais. A deteção de colisão entre objetos é baseada na subdivisão hierárquica do espaço com *octrees*. Deformação massa-mola é utilizada para simular a alteração na forma dos objetos que se colidem. Experimentos são realizados para demonstrar as funcionalidades do protótipo.

Palavras-chaves: Ambientes Virtuais; Treinamento Médico; Colisão de Objetos; Deformação de Objetos.

CAPÍTULO 1

INTRODUÇÃO

Este capítulo visa descrever o contexto no qual o trabalho de pesquisa está inserido, destacando-se alguns aspectos sobre ambientes virtuais interativos, áreas de aplicação, objetivos, justificativa e contribuições do trabalho, bem como a organização do texto.

1.1 Realidade Virtual

O termo Realidade Virtual (RV) possui diversas definições, tanto na área acadêmica quanto na área industrial. De uma forma direta, pode ser definida como uma interface natural e poderosa de interação homem-máquina, por permitir ao usuário interação, navegação e imersão em um ambiente tridimensional sintético, gerado por computador, através de canais multissensoriais, tais como visão, audição e tato [34]. De acordo com Hancock [25], a RV é a forma mais avançada de interface do usuário com o computador até agora disponível.

A RV fornece mecanismos mais adequados para construção de aplicações que exigem interação avançada com o usuário, sendo uma forma das pessoas visualizarem, manipularem e interagirem com computadores e dados extremamente complexos [3]. A RV é um paradigma no qual se usa um computador para a interação com algo irreal, mas que pode ser considerado real no momento em que está sendo utilizado [60].

A grande vantagem desse tipo de interface é que o conhecimento intuitivo do usuário, a respeito do mundo físico, pode ser utilizado para manipular o mundo virtual. Dispositivos não convencionais como capacetes, luvas de dados, dispositivos hápticos (figura 1.1) podem ser utilizados para a visualização desse mundo e para que o usuário tenha uma sensação de estar em um mundo real [60].

Latta e Oberg [37] afirmam que a RV envolve a criação e experimentação de ambientes. O objetivo é colocar o usuário em um ambiente em que não pode ser encontrado trivialmente no mundo real e estabelecer uma ligação entre ambos. Valerio Netto et al. [60] lembram que a

Figura 1.1: Interação e motivação do usuários em um Sistema de Realidade Virtual. Fonte: [10].

partir do momento que o usuário pode interagir com o mundo virtual, por meio de dispositivos de entrada e saída, esse mundo virtual passa a ser um Ambiente Virtual (AV) ou um Ambiente de RV (ARV). Além da interação, três aspectos têm que ser considerados para esse ambiente se tornar um AV ou ARV: imersão, interação e envolvimento.

A imersão deve proporcionar ao usuário a sensação de presença dentro do mundo virtual. A RV pode ser considerada imersiva ou não imersiva. Na RV imersiva, capacetes ou cavernas (salas em que paredes, teto e chão são telas de projeção) são utilizados, enquanto a não imersiva utiliza apenas monitores de vídeo. Há também diferentes graus de imersão, levando em consideração dispositivos baseados em outros sentidos, como a audição e o tato.

A interação está ligada à influência das ações do usuário no comportamento dos objetos, ou seja, o AV é modificado de acordo com os comandos do usuário.

O envolvimento, por sua vez, está associado ao grau de motivação que o mundo virtual proporciona a este usuário, podendo ser passivo a esse AV como, por exemplo, apenas a leitura de algo no AV, ou ativo, como participar de um jogo.

Na RV, a renderização deve ser feita em tempo real, isto é, imagens do AV têm que ser atualizadas sempre que ocorrer uma modificação na cena e a inclusão da descrição funcional dos objetos [60]. Na prática, a RV faz com que o usuário navegue e observe um mundo

tridimensional, em tempo real e com seis graus de liberdade, referentes aos seis tipos de movimento: para frente/para trás, acima/abaixo, esquerda/direita, inclinação para cima/para baixo, angulação à esquerda/à direita e rotação à esquerda/à direita. Em sua essência, a RV é uma cópia da realidade física, na qual o indivíduo tem a capacidade de interagir com o mundo ao seu redor. Os equipamentos de RV (dispositivos não convencionais) simulam essas condições, em que o usuário pode até mesmo "tocar" os objetos de um mundo virtual e fazer com que eles respondam ou mudem, de acordo com suas ações [63].

Morie [46] define um AV como um ambiente tridimensional ou espaço imaginário gerado por computador. Em geral, os ambientes virtuais visam reproduzir situações do mundo real ou próximas daquelas percebidas no mundo real. Eles têm a finalidade de permitir simulação, treinamento, visualização ou outro tipo de atividade por meio de técnicas de RV.

Normalmente, os ambientes virtuais construídos para aplicações de simulação e treinamento são compostos por dois ou mais objetos que podem se movimentar a partir da interação do usuário. Para propiciar a execução dessas atividades, os AVs normalmente devem prever características inerentes ao mundo real como movimentos, colisões e deformações, a fim de imprimir maior realismo às aplicações.

Vários domínios de conhecimento podem ser beneficiados pela Realidade Virtual, tais como os listados a seguir [34]:

a) *Medicina*

b) *Educação*

c) *Entretenimento*

d) *Treinamento*

e) *Visualização de Informação*

f) *Auditórios Virtuais ou Teatros de Realidade Virtual*

g) *Artes*

h) *Telepresença e Telerrobótica*

i) *Anúncio Experiencial*

j) *Sistemas de Manutenção usando Realidade Aumentada*

1.2 Objetivos do Trabalho

O objetivo deste trabalho consiste na criação de um protótipo de ambiente virtual para treinamento médico. Duas classes principais são projetadas e implementadas no ambiente com a linguagem de programação Java. Métodos de detecção de colisão entre objetos são estudados e avaliados, buscando-se uma solução com alto nível de precisão e bom desempenho, de forma a permitir uma simulação com realismo. Além disso, uma vez que os objetos que colidem podem sofrer alterações em suas formas, métodos de deformação são investigados e implementados, em particular, aqueles que utilizam malhas poligonais para representação dos objetos.

Experimentos são realizados para demonstrar as funcionalidades do protótipo. As classes e os métodos desenvolvidos serão posteriormente integrados no *framework* ViMeT [47]. Como este *framework* está em sua primeira versão, uma tentativa de melhorar suas funcionalidades também será parte do objetivo deste trabalho.

1.3 Justificativa e Contribuições do Trabalho

O ambiente desenvolvido neste trabalho visa à simulação do procedimento de punção mamária em exames de biópsia, o qual consiste na extração de pequenas partes de tecidos do órgão em questão para auxiliar a elaboração do diagnóstico médico. A partir dessa experiência, pretende-se tornar possível a utilização do ambiente em novas aplicações para treinamento médico.

O trabalho inclui o estudo, proposição, implementação e avaliação de métodos para colisão e deformação de objetos representados com malhas poligonais. O desenvolvimento do protótipo adota ferramentas computacionais de baixo custo, com tecnologia aberta, orientada a objetos, livre e multiplataforma, o que traz benefícios para a comunidade de usuários

por facilitar a utilização e expansão de suas funcionalidades e por permitir a integração das aplicações existentes no *framework*.

1.4 Organização do Texto

O restante do trabalho é dividido como segue. O capítulo 2 apresenta uma revisão dos principais fundamentos e abordagens relacionados à análise de detecção de colisão e deformação de objetos, além de um estudo de ferramentas para desenvolvimento do ambiente virtual interativo. A metodologia proposta neste trabalho é descrita no capitulo 3, onde são ilustradas as etapas para a construção do protótipo. O capítulo 4 apresenta os resultados obtidos com a aplicação da metodologia. O capítulo 5 apresenta a conclusão do trabalho e considerações finais, bem como a citação de trabalhos futuros. O apêndice A descreve formato dos arquivos utilizados na representação das malhas poligonais. O apêndice B apresenta as principais classes e métodos desenvolvidos no trabalho.

CAPÍTULO 2

TÓPICOS RELACIONADOS

Este capítulo aborda tópicos inerentes ao trabalho desenvolvido, incluindo uma explanação sobre detecção de colisão (definição e métodos), deformação (definição e métodos) e ferramentas que podem ser utilizadas para o desenvolvimento de Ambientes Virtuais (AVs).

2.1 Detecção de Colisão

Detectar uma colisão é verificar a aproximação entre objetos de um ambiente virtual. A aproximação desses objetos deve ser suficientemente pequena a ponto de possibilitar a ocorrência de uma sobreposição entre objetos. Essa proximidade fornece assim um maior realismo às aplicações de Realidade Virtual (RV). A sua percepção exige a presença de, no mínimo, dois objetos em um ambiente virtual, sendo que pelo menos um deles deve estar em movimento.

Trata-se de um problema complexo, visto que objetos virtuais podem ter formas, tamanhos e movimentos variados, dependendo de suas naturezas e da finalidade da aplicação. Além disso, os objetos em cena podem ser rígidos ou deformáveis.

A detecção de colisão é um dos itens mais complexos e dependentes das informações de interação monitoradas em um AV. Permitindo responder às interações entre objetos no mundo virtual, fator importante para obtenção do realismo. Esse requisito exige programas específicos para gerenciar as informações de interação, envolvendo rotinas de controle e computação gráfica [41].

O problema da detecção de colisão entre objetos é fundamental em qualquer simulação do mundo físico, sendo estudado por diversas comunidades, incluindo a robótica, a computação gráfica e a geometria computacional. Vários algoritmos utilizam primitivas simples para delimitação do espaço de colisão do objeto, tais como esferas, caixas envoltórias alinhadas (AABB, do inglês *Axis-Aligned Bounding Boxes*) e caixas envoltórias orientadas (OBB,

do inglês *Oriented Bounding Boxes*), ilustradas na figura 2.1.

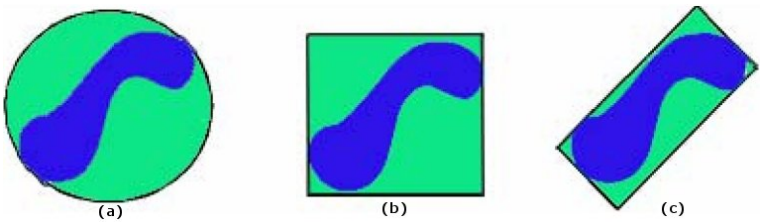

Figura 2.1: Representação em 2D da delimitação de volume por primitivas. (a) esfera; (b) caixa alinhada (AABB); (c) caixa orientada (OBB). Fonte: [24].

Esses métodos normalmente são eficientes apenas com objetos não muito complexos ou para aplicações em que a detecção de colisão não necessite refletir o objeto como seu espaço delimitador de colisão. Para objetos com complexidade um pouco mais elevada, tem-se uma imprecisão na detecção de colisão. Diante disso, diversos métodos e algoritmos têm sido propostos para verificar a detecção de colisão entre objetos.

2.2 Métodos de Detecção de Colisão

Devido ao fato de que a detecção de colisão depende extremamente das informações provenientes das interações do usuário com o AV, algoritmos eficientes devem ser desenvolvidos para alcançar precisão dentro de tempos de respostas aceitáveis. Este requisito pode demandar programas específicos para gerenciar as informações de interação, envolvendo rotinas de controle e computação gráfica [42].

Outro ponto importante a se destacar é o número de testes de colisão possíveis feitos em um AV e, dependendo do volume dos testes, os recursos da máquina podem ficar comprometidos. Por exemplo, se há n objetos em uma cena, então o primeiro objeto pode se colidir com $n-1$ objetos (desde que a intra-colisão não seja considerada), o segundo objeto com $n-2$ objetos adicionais, havendo assim muitas possibilidades para colisão, que podem ser expressas como:

$$(n-1)*(n-2)*(n-3)...1 \tag{2.1}$$

que é equivalente a

$$(n-1)! \tag{2.2}$$

As seções a seguir descrevem sucintamente um conjunto de abordagens de algoritmos de detecção de colisão existentes na literatura e agrupados por tipo de algoritmos.

2.2.1 Volumes Limitantes

A detecção de colisão por volumes limitantes são algoritmos que utilizam primitivas simples para delimitação do espaço de colisão do objeto, tais como esferas, caixas alinhadas aos eixos (*Axis-Aligned Bounding Boxes - AABB*) e caixa orientada (*Oriented Bounding Box - OBB*). A principal idéia desse tipo de abordagem é verificar os pontos mais afastados do objeto e inserir a primitiva nesse espaço.

2.2.1.1 Caixas Envoltórias Alinhadas aos Eixos (AABB)

O algoritmo de caixas envoltórias alinhadas aos eixos (*Axis Align Bounding Boxes - AABB*) verifica os dois pontos mais afastados do objeto e envolve esse objeto com uma caixa, alinhando-a com os eixos de coordenada do sistema [35, 61]. O algoritmo 1 demonstra em um pseudocódigo a construção de uma AABB.

```
início
    para todo vértice faça
        determinar menor valor x;
        determinar menor valor y;
        determinar menor valor z;
        determinar maior valor x;
        determinar maior valor y;
        determinar maior valor z;
    fim
    atribuir valores mínimos (x,y,z) como vértice da caixa;
    atribuir valores máximos (x,y,z) como vértice da caixa;
fim
```

Algoritmo 1: Pseudocódigo de criação de uma AABB.

A figura 2.2 mostra um exemplo do método de caixas envoltórias alinhadas (AABB).

O teste de sobreposição de caixas é bem simples. Sabendo-se que todas as caixas são orientadas

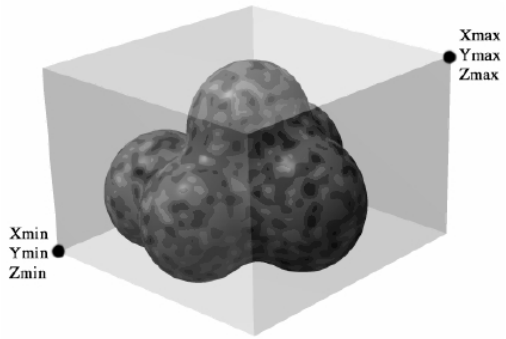

Figura 2.2: Exemplo do método AABB. Fonte: [24].

na mesma direção, então é apenas necessária a comparação dos mínimos e máximos das coordenadas x, y e z. Se uma caixa 'A' é definida por AxMin, AxMax, AyMin, AyMax, AzMin, AzMax e uma caixa 'B' por BxMin, BxMax, ByMin, ByMax, BzMin, BzMax, então essas caixas se sobrepõem quando ocorrem as seguintes condições:

$$AxMin > BxMax \text{ e } AxMax > BxMin$$
$$AyMin > ByMax \text{ e } AyMax > ByMin$$
$$AzMin > BzMax \text{ e } AzMax > BzMin$$

No entanto, isso apenas se aplica se as caixas envoltórias forem alinhadas aos eixos (AABB). Se uma caixa envoltória for definida nas coordenadas locais e uma das caixas estiver sob uma transformada como rotação, outras soluções devem ser adotadas. Pode se utilizar um algoritmo que detecta a intersecção entre caixas orientadas (OBB), em coordenadas absolutas, o que é mais complexo, ou então recalcular a AABB a cada *frame* nas coordenadas absolutas, sendo que efetuar cálculos a cada *frame* pode sobrecarregar o processamento do computador.

2.2.1.2 Caixas Orientadas (OBB)

O algoritmo de caixas orientadas ou OBB (*Oriented Bounding Box*) corresponde a uma caixa que possui sua própria orientação. A OBB é definida pelos seguintes atributos: centro, 3 vetores unitários que representam a direção da caixa e a extensão em cada direção. A OBB verifica os pontos mais afastados do objeto para definir seu volume limite e faz com que esse volume se oriente pela forma do objeto e se encaixe de tal forma que sua área fique a mais próxima possível da forma do objeto [22].

Em relação ao ajuste, a OBB apresenta resultados significativamente melhores do que uma AABB, porem, o cálculo de interseção é muito mais complexo e lento, requerendo assim muito mais processamento comparado com as esferas e com a AABB.

A figura 2.3 mostra a diferença entre o envolvimento de uma AABB com relação a uma OBB, bem como a diferença na orientação dos eixos de coordenadas. A AABB é alinhada aos eixos de coordenadas espaciais e a OBB com seus eixos alinhados às coordenadas locais do objeto.

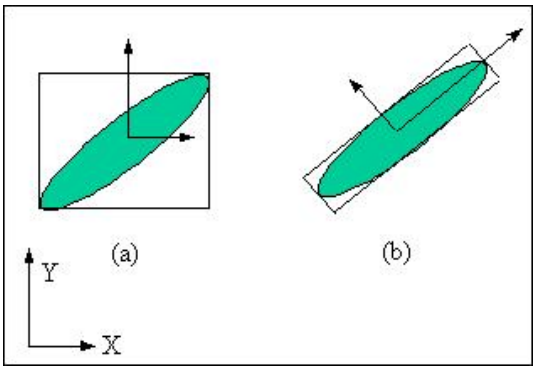

Figura 2.3: Objeto envolvido com (a) uma AABB e (b) uma OBB [6].

A criação da OBB também é mais complexa e é baseada na distribuição Gaussiana dos vértices do modelo. A distribuição Gaussiana é a distribuição de probabilidade com a forma $A\exp((X - M)C^{-1}(X - M))$, onde A é um fator de escala apropriado, M é a média da distribuição e C é a matriz de covariância da distribuição.

Eberly [18] demonstra um método sofisticado para construir uma caixa orientada que normalmente ajusta os pontos de maneira melhor do que as caixas alinhadas aos eixos.

O centro da caixa é a média dos pontos V, ou seja

$$C = \frac{1}{n}\sum_{j=0}^{n} V_j \tag{2.3}$$

Os eixos da caixa são determinados como sendo os auto-vetores da matriz de covariância

$$M = \frac{1}{n}\sum_{j=0}^{n}(V_j - C)(V_j - C)^T \tag{2.4}$$

Se U_j são os auto-vetores unitários, as extensões ao longo desses eixos são os extremos das projeções dos pontos sobre esses eixos

$$e_i = \max_j |U_i.(V_j - C)| \tag{2.5}$$

A técnica de Eberly [18] é descrita no algoritmo 2.

Primeiramente, o algoritmo determina o centro da OBB e efetua o cálculo da matriz de covariância dos pontos. Os autovetores, extraídos dessa matriz, serão os eixos da caixa orientada. Tendo definidos esses eixos, faz-se o cálculo da extensão em cada eixo (projeção do vetor), então um novo centro para o OBB é definido com os valores máximo e mínimo da projeção dos vetores.

```
início
    // TCaixa é composta por centro, eixo[3] e extensão[3]
    TCaixa caixa;
    // calcula média dos pontos
    Tponto3D somatório = v[0];
    para (i = 1; i<n; i++) faça
        somatório += v[i];
    fim
    caixa.centro = somatório/n;
    // Calcula as covariâncias dos pontos
    TMatrix3x3 mat = 0;
    para (i = 0; i < n; i++) faça
        Tponto3D delta = v[i] - caixa.centro;
        mat += Tensor(delta,delta);
    fim
    TMatrix3x3 covariância = mat/n;
    // Os auto-vetores da matriz de covariância são os eixos da caixa
    ExtrairAutoVetores(covariância,caixa.eixos[3]);
    // Calcula as extensões como sendo os valores extremos das projeções dos pontos sobre os eixos
       da caixa
    caixa.extensão = 0;
    para (i = 0; i < n; i++) faça
        Tponto3D delta = v[i] - caixa.centro;
        para (j = 0; j < 3; j++) faça
            real escalar = |ProdutoEscalar(caixa.eixo[j],delta)|;
            se (escalar > caixa. extensão[j]) então
                caixa. extensão[j] = escalar;
            fim
        fim
    fim
fim
```

Algoritmo 2: Exemplo de algoritmo de criação de uma OBB [18].

No algoritmo, $Tensor(W, W)$ corresponde à matriz $W.W^T$ e supõe-se a existência de uma função para obtenção dos auto-vetores de uma matriz 3×3. Os auto-vetores podem ser calculados utilizando-se uma solução aproximada em vez de um esquema iterativo.

2.2.1.3 Esferas

Similar ao algoritmo AABB, o método de esferas (*spheres*) testa os pontos mais afastados do objeto e envolve o mesmo com uma esfera. Esferas envolventes são representadas por um centro e um raio. Uma esfera é definida pelo conjunto de todos os pontos x eqüidistantes de um ponto central com uma distância $r > 0$. A equação quadrática que define esse conjunto é $|X - C|^2 = r^2$. Para um objeto geométrico que consiste de uma coleção de pontos $(V_i)_{i=0}^{n}$, uma esfera envolvente pode ser calculada de várias formas.

Ritter [54] desenvolveu um algoritmo 3 que garante a construção de uma esfera quase ótima e garante que a esfera resultante desse algoritmo é apenas 5% maior que a esfera ótima. O algoritmo requer duas passagens pelo conjunto de vértices e sua ordem é linear.

início
- **para todo** *vértice* **faça**
  - encontrar valor mínimo x;
  - encontrar valor mínimo y;
  - encontrar valor mínimo z;
  - encontrar valor máximo x;
  - encontrar valor máximo y;
  - encontrar valor máximo z;
  - **para** *mínimo (x,y,z) e máximo (x,y,z)* **faça**
    - encontrar maior distância entre os vértices;
  - **fim**
  - criar esfera com o par de maior distância com centro sendo o ponto médio do segmento e o raio a metade da distância entre eles;
  - **para todo** *vértice* **faça**
    - **se** *vértice estiver fora da esfera criada* **então**
      - calcular novo raio

$$r = \frac{(d + r)}{2}$$

`// em que` d `é a distância do ponto ao centro da esfera`

novo centro da esfera será

$$k = Vertice - r$$

$$Centro = \frac{(r * velhoCentro + k * Vertice)}{d}$$

    - **fim**
  - **fim**
- **fim**

fim

Algoritmo 3: Pseudocódigo da construção da esfera ótima [54].

A sobreposição das esferas ocorre quando

$$(ax - bx) * 2 + (ay - by) * 2 + (az - bz) * 2 < (ar - br) * 2 \tag{2.6}$$

A vantagem desse método é que ele é independente da orientação, em contraste com o método de caixas que requer o alinhamento dos eixos.

A desvantagem é que esse método pode não envolver adequadamente um objeto longo e fino, o que pode resultar em falsa detecção de colisão. Nesses casos, um segundo teste poderia ser realizado.

2.2.2 Subdivisão Hierárquica do Espaço

Neste método, o ambiente é subdividido em um espaço hierárquico. Objetos no ambiente são agrupados de acordo com a região em que se encontram.

Quando um objeto no ambiente se movimenta e troca de posição, movendo-se para uma outra região do espaço, apenas os objetos do novo espaço precisam ser verificados se colidem ou não com o objeto em movimento (figura 2.4).

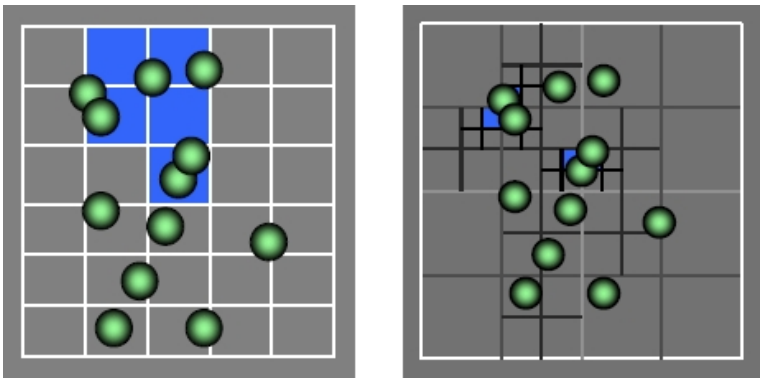

Figura 2.4: Método de subdivisão hierárquica do espaço. Fonte: [24].

O ambiente virtual pode ser subdividido utilizando-se alguns métodos conhecidos tais como *octrees*, *K-d trees* e *BSP trees*, descritos a seguir.

2.2.2.1 Octrees

Segundo Hearn e Baker [26], *octrees* são estruturas de árvores hierárquicas em que cada nó interno tem até oito filhos e é organizada para que cada nó corresponda a uma região do espaço

tridimensional. O esquema de codificação divide regiões de espaços tridimensionais, normalmente cubos, em octantes e armazena elementos em cada nó da árvore.

As *octrees* são mais comumente usadas para particionar um espaço tridimensional, subdividindo-o recursivamente em oito octantes. Octrees são uma analogia tridimensional das *quadtrees*.

Cada nó da *octree* armazena um ponto tridimensional, que é o centro do octante que foi dividido. O ponto define um dos cantos para cada um dos oito filhos. O ponto de subdivisão é implicitamente o ponto central do espaço que o nó representa.

A figura 2.5 ilustra um espaço virtual particionado por uma *octree*.

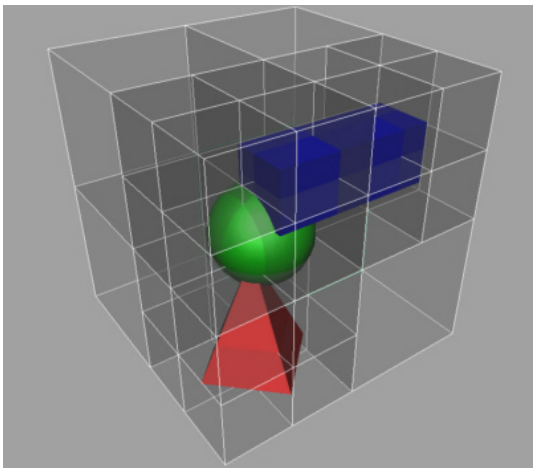

Figura 2.5: Representação de um espaço particionado por uma octree. Fonte: [21].

2.2.2.2 Árvores k-dimensionais (K-d trees)

Árvores k-dimensionais (*K-d trees*), em que k é o número de dimensões, são estruturas de árvores multi-dimensionais balanceadas que particionam o espaço para organização de pontos de um espaço k-dimensional. Elas são estruturas de dados muito úteis em várias aplicações, tais como as buscas que envolvem chaves multi-dimensionais (busca em escala e busca do vizinho mais próximo) [7, 17].

Árvores k-dimensionais são casos especiais de árvores BSP, descritas a seguir. Uma *k-d tree* usa apenas planos de divisão que são perpendiculares a um dos eixos de coordenadas do sistema, em contraste com uma árvore BSP, que pode se utilizar de planos de divisão arbitrários. Como conseqüência, cada plano de divisão deve atravessar um dos outros pontos da árvore [8, 17].

Adicionalmente, cada nó de uma *k-d tree*, da raiz até as folhas, guarda um único ponto, outro ponto contrastante com relação às árvores BSP, no qual folhas são tipicamente os únicos nós que

contêm pontos ou outra primitiva [52].

A figura 2.6 mostra um exemplo de particionamento de espaço por árvores *k-d*.

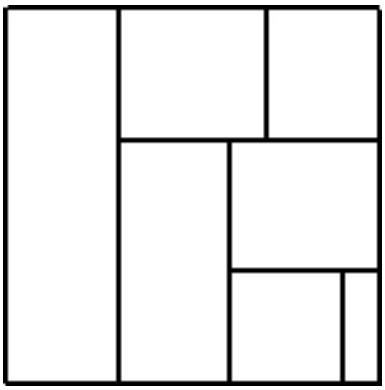

Figura 2.6: Representação de um espaço 2D particionado por uma *k-d tree* [24].

2.2.2.3 Árvores de Particionamento do Espaço Binário (BSP trees)

A *BSP tree* (árvore de particionamento do espaço binário, do inglês *binary space partitioning tree*) representa uma divisão recursiva hierárquica ou a subdivisão de um espaço n-dimensional em subespaços convexos. A construção da *BSP tree* é um processo que considera um subespaço e o particiona por qualquer hiperplano que intersecta o interior do subespaço. O resultado disso são dois novos espaços que podem ser particionados novamente por um método recursivo.

Um hiperplano em um espaço n-dimensional é um objeto de dimensão $n-1$ que pode ser usado para dividir o espaço em dois subespaços. Por exemplo, em um espaço tridimensional, o hiperplano é um plano e, em um espaço bidimensional, uma linha.

As *BSP trees* são extremamente versáteis, porque são poderosas na classificação e classificadores de estruturas. Podem ser usadas desde a remoção de superfícies escondidas e hierarquias de *ray tracing* até a modelagem de sólidos e o planejamento de um movimento de um robô [59].

A figura 2.7 mostra o exemplo de uma *BSP tree* no espaço bidimensional.

2.2.3 Subdivisão Hierárquica do Objeto

Na subdivisão hierárquica do objeto, este é subdividido em regiões, sendo que a detecção de colisão é feita quando uma dessas regiões é intersectada por um outro objeto.

A hierarquia de volumes envolventes é baseada em dois tipos de nós. Nós internos e nós folhas.

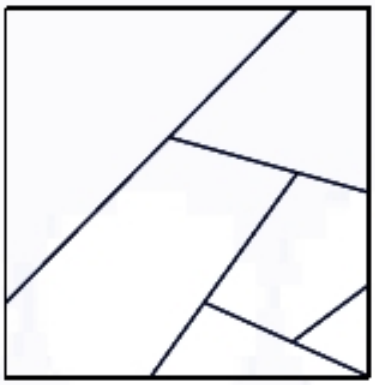

Figura 2.7: Espaço particionado por uma *BSP tree*. Fonte: [24].

Cada nó, independentemente do tipo, possui um volume envolvente associado. Nós folhas possuem volumes envolventes que englobam somente a geometria (ou seja a malha de triângulos) enquanto nós internos possuem volumes envolventes que englobam todos os volumes envolventes de seus filhos. Por isso, pode-se descartar todos os filhos se o volume envolvente do nó pai não colidir com outro nó.

Devido a esse fato, para cada tipo de volume envolvente é implementado um método responsável por "somar" dois volumes envolventes do mesmo tipo, obtendo-se

$$BV_{pai} = \sum_{i=0}^{n} BVi \tag{2.7}$$

em que n é o número de filhos do nó pai em questão.

Outro aspecto muito importante da hierarquia é a noção de espaço. Como todos os nós e seus volumes envolventes são definidos no espaco local do objeto, surge a necessidade de associar a cada nó uma matriz de transformação responsável por mapear este nó para o espaço do nó pai. A detecção de colisão é efetuada no espaço global.

Uma forma de se implementar a subdivisão hierárquica do objeto é utilizar o método de árvores de volume limite (AVL) [35], onde cada nó n corresponde a um subconjunto $C.n$ pertencente a C, com o nó raiz sendo associado a um volume limite com o total do conjunto C. Cada nó interno tem dois ou mais filhos, sendo que o número máximo de filhos para cada nó interno é chamado de grau da árvore. A união de todos os subconjuntos associados com o filho do conjunto n é igual a $C.n$. Cada nó é associado também a um volume limite $v(C)$, que é uma aproximação do espaço ocupado por $C.n$, usando formas mínimas de representação de formas como caixas e esferas.

Os métodos mais conhecidos e que utilizam AVL, mostrados a seguir, são: *spheres-trees*, *AABB-trees*, *OBB-trees* e k-dops. Há outros algoritmos conhecidos e eficientes encontrados na literatura, mas são variações desses métodos fundamentais utilizados em detecção de colisão.

2.2.3.1 Sphere-trees

O método de *sphere-trees* é um dos mais simples, sendo esse um motivo por torná-lo muito popular em aplicações de RV. Hubbard [29] implementa esse método, onde o objeto é envolvido por uma esfera (nível 0) e, recursivamente, há uma união sucessiva de mais esferas fazendo com que elas se aproximem ao máximo da resolução do objeto. Considerando que as esferas são invariantes quanto à rotação, então, para um objeto rígido, a hierarquia é construída por meio de um pré-processamento e as transformações da hierarquia são as mesmas transformações lineares aplicadas ao objeto. A figura 2.8 mostra um exemplo de *sphere-trees*.

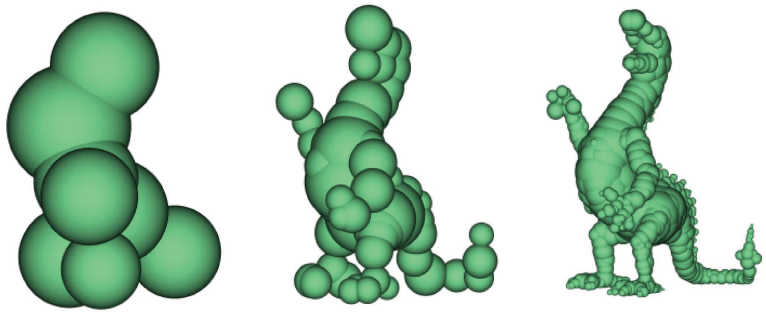

Figura 2.8: Exemplos do método de *sphere-trees*. Fonte: [9].

Bradshaw e O'Sullivan [9] mostram um exemplo de algoritmo de construção de uma *sphere-tree*. O algoritmo consiste em um número de camadas, no qual a camada de nível mais alto decompõe a construção em um número de sub-problemas, subdividindo o objeto em regiões. Essas regiões estão aproximadas por um grupo de esferas que são usadas adiante para dividir o objeto e então construir subárvores.

O nó raiz da *sphere-tree* é a menor esfera que possa envolver o objeto. O primeiro nível das esferas, os filhos do nó raiz, são construídos através de um algoritmo que determina um número de filhos que deve ser gerado para cada nó da árvore.

Essas esferas são então usadas para segmentar o objeto em um número de regiões. Cada região define a área do objeto que deve ser coberta por um grupo de esferas filhas. Essas esferas formam

o próximo nível da hierarquia como filhos das esferas que definiram a região [9].

O algoritmo 4 mostra a construção de uma *sphere-trees* em um pseudocódigo.

```
início
    Construir grupo de pontos que cubra o objeto;
    Construir diagrama de Voronoi, contendo número inicial de esferas;
    Construir uma esfera envolente, representando nó raiz da árvore;
    para cada nível da árvore faça
        para cada esfera contida no nível faça
            Determinar região do objeto coberta pela esfera pai;
            Atulizar diagrama de Voronoi para cobrir a região da esfera pai;
            Gerar grupo de esferas do diagrama de Voronoi, reduzindo a região da esfera
            pai;
        fim
    fim
fim
```

Algoritmo 4: Pseudocódigo da construção de uma *sphere-tree*.

2.2.3.2 AABB-trees

O método de árvores AABB basicamente divide o objeto em sub-regiões a partir de um AABB (nó raiz). Uma caixa AABB não pode ser colocada livremente no objeto, pois o AABB é orientado aos eixos de coordenadas do espaço.

Uma árvore AABB é construída de cima para baixo (*top-down*), por subdivisões recursivas. Em cada passo da recursão, o menor AABB do conjunto de primitivas é computado e o conjunto é dividido respeitando-se a escolha do melhor plano. O processo continua até que cada subconjunto contenha apenas um elemento. Sendo assim, uma AABB para um conjunto de primitivas tem n folhas e $n-1$ nós internos [61].

A imagem 2.9 mostra a seqüência para construção de uma *AABB-tree*.

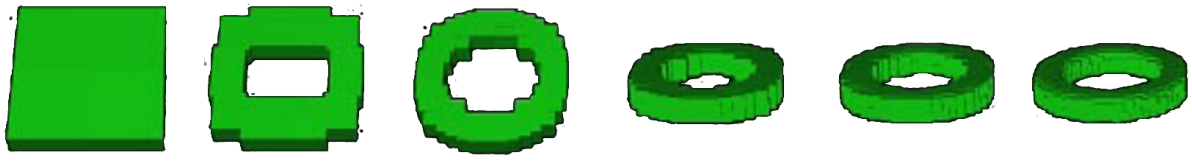

Figura 2.9: Construção de uma *AABB-tree* [22].

O nó raiz é uma caixa. A partir do segundo passo é adicionado um conjunto de caixas para que

estas cheguem próximo ao formato do objeto envolvido.

2.2.3.3 OBB-trees

OBB-tree é uma árvore onde cada nó possui uma OBB. Gottschalk [22] define árvores OBB da seguinte forma:

- a estrutura básica é uma árvore binária onde cada nó possui uma OBB.
- cada nó folha possui apenas um triângulo.
- pode ser criada a partir de abordagem *top-down* ou *bottom-up*

A utilização de árvores OBB representa um ganho em relação ao desempenho e precisão na detecção de colisão. Árvores OBB garantem um ajuste muito superior em relação a outros volumes envolventes.

A construção de uma árvore OBB possui dois componentes básicos: a colocação da OBB ao redor do grupo de polígonos para que estes tenham um encaixe justo e o agrupamento das OBB encaixados em uma árvore hierárquica.

A figura 2.10 mostra a construção de uma árvore OBB.

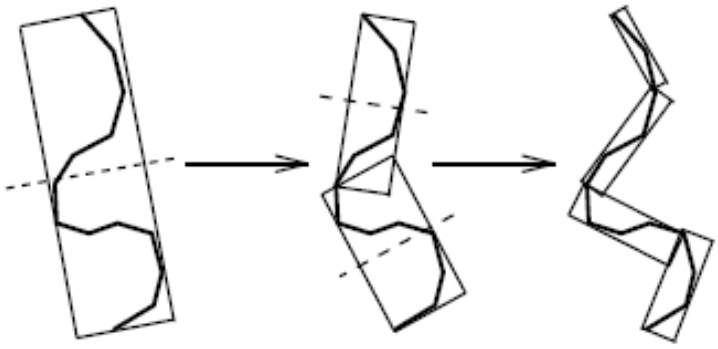

Figura 2.10: Exemplo do método de *OBB-trees* em 2D [24].

2.2.3.4 k-DOPS

A idéia do algoritmo de *k-dops*, segundo Klosowski et al. [35], é de construir uma hierarquia de orientação discreta de poliedros convexos cujas faces são determinadas por placas paralelas, no qual suas normais exteriores vêm de um pequeno número fixo de k-orientações.

A figura 2.11 mostra o método *k-dops* sendo aplicado a um modelo de avião.

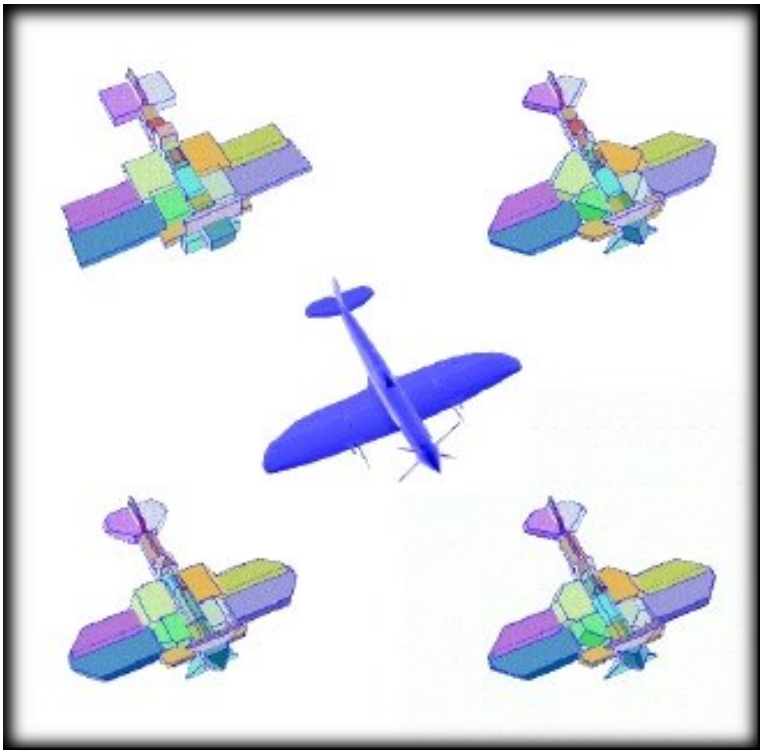

Figura 2.11: Exemplo do método de *k-dops*. Fonte: [24].

2.2.4 Computação Incremental da Distância

A distancia mínima é verificada incrementalmente entre um par de objetos, ou seja, a cada iteração do ambiente é feita uma verificação da distância entre os objetos. Quando essa distância se torna tão pequena a ponto de que os objetos possam se colidir, é quando há uma notificação ao ambiente sobre a detecção de colisão.

Há vários métodos propostos para a computação incremental da distância, bem como para a computação hierárquica da distância para corpos convexos em movimento [23], algoritmos eficientes para computação da distância incremental [40], algoritmos incrementais para detecção de colisão entre polígonos [51], entre outros.

2.2.5 Objetos Rígidos e Deformáveis

Algoritmos de detecção de colisão podem ser classificados em duas categorias, para objetos rígidos e para objetos deformáveis [5]. A grande diferença entre esses dois tipos de abordagens é que

os algoritmos para objetos rígidos normalmente são pré-computados e os para objetos deformáveis são processados em tempo de execução.

A detecção de colisão em objetos rígidos pode ser definida quando os objetos em cena não têm alteração em sua forma como uma característica do objeto. Há muitos métodos desenvolvidos para objetos rígidos, uma vez que são de implementação mais simples e não há necessidade de verificação da intra-colisão (colisão com o objeto e ele mesmo), pois não há deformação.

Como os métodos de colisão para objetos deformáveis são computados em tempo de execução, há outras abordagens específicas para esse tipo de colisão, os métodos que consideram ou não a intra-colisão. Lau et al. [38] afirmam que há poucos métodos de detecção de colisão em objetos deformáveis e, desses poucos, nem todos consideram a intra-colisão. A maioria desses algoritmos utiliza a abordagem da subdivisão hierárquica do objeto e que pode ser classificada em dois grupos: objetos representados por malha poligonal e por superfícies paramétricas.

Como dito anteriormente, métodos para objetos deformáveis, além de considerar a inter-colisão (entre dois objetos distintos), tratam a intra-colisão (auto-colisão) para que seja considerado um algoritmo robusto e de qualidade.

2.2.5.1 Malhas Poligonais

Em objetos representados por malhas poligonais, sua deformação é feita através da atualização da posição dos vértices da malha do polígono. A figura 2.12 mostra um exemplo de um modelo em 3D representado por malha poligonal.

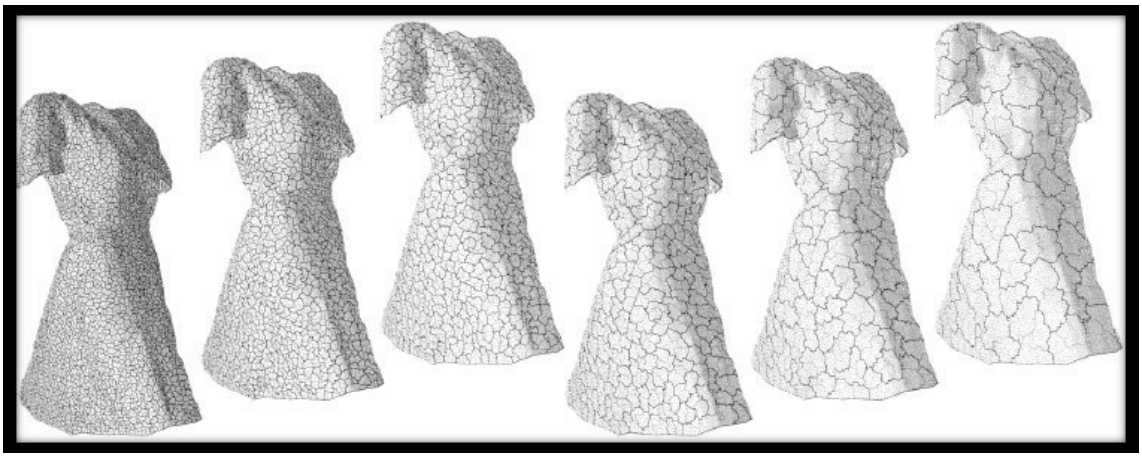

Figura 2.12: Objeto representado por malha poligonal. Fonte: [24].

O algoritmo proposto por Yang e Thalmann [64] é um método hierárquico utilizando *octree*, em

que cada nó da *octree* é uma caixa envoltória (*bounding box*) alinhada aos eixos globais (eixos do ambiente). Trata-se de um método simples, entretanto, se ocorrer uma deformação no objeto, toda a hierarquia precisa ser recalculada novamente.

Bergen [61] propõe um método no qual uma árvore AABB é construída para cada objeto, no qual consiste em uma árvore binária de caixas envoltórias alinhadas às coordenadas locais do objeto. Ao invés de reconstruir a árvore AABB, as caixas se realinham enquanto o objeto deforma, assim simplificando o processo de atualização das caixas. A sobreposição das áreas das caixas aumenta constantemente. Esse método assim como o método proposto por Yang e Thalmann [64] também requer que todos os nós sejam atualizados enquanto o objeto se deforma.

Volino e Thalmann [62] propõem um método no qual primeiramente uma hierarquia de polígonos é criada. As normais desses polígonos e o contorno do polígono gerado são analisadas para determinar se ocorre uma intra-colisão. Lau et al. [38] afirma que esse método é eficiente somente se operações simples de *bits* são efetuadas na análise das normais e que testes experimentais mostraram que o teste do contorno pode ser ignorado sem afetar os resultados da intra-colisão.

2.2.5.2 Superfícies Paramétricas

Objetos representados por superfícies paramétricas usualmente são deformados nos pontos de controle da superfície e ,deformações desse tipo, podem causar uma divisão da superfície caso se pretenda manter a qualidade do resultado da representação.

Herzen et al. [27] propõem um método de detecção de colisão entre dois objetos de superfícies paramétricas, mas o custo computacional desse algoritmo é muito alto e requer que as superfícies satisfaçam condições de continuidade (valores de Lipschitz).

Hughes et al. [30] mostra um método de colisão para superfícies baseadas em Bèzier e B-spline. O método constrói uma árvore AABB para cada superfície e utiliza uma pseudo-normal e um mapa Gaussiano para detectar intra-colisão e o método *sweep and prune* para detectar outras colisões. No entanto, a construção das pseudo-normais, o cálculo dos novos pontos da superfície e a atualização completa da árvore AABB têm que ser realizados em cada quadro durante a deformação do objeto. Além disso, esse método requer um alto número de equações para determinar se a superfície possui uma intra-colisão, sendo então de alto custo computacional.

Em linhas gerais, se um objeto é deformado baseado na atualização do modelo poligonal, os correspondentes métodos de detecção de colisão podem ser mais eficientes porque apenas envolvem

atualizar a hierarquia de limites enquanto o objeto se deforma [38].

Por outro lado, se o objeto é deformado baseado na atualização das superfícies, pontos precisam ser computados ou removidos dinamicamente. Normalmente, algoritmos de detecção de colisão relacionados a esse tipo de deformação são menos eficientes porque a área de colisão precisa ser reconstruída enquanto o objeto se deforma [38].

2.3 Deformação

A deformação de objetos é um recurso que auxilia a transformação de um AV em uma simulação mais próxima do mundo real. Em objetos 3D, a deformação é implementada em modelos que permitam a manipulação de sua estrutura (vértices e arestas) [13].

A deformação de um objeto é basicamente a mudança da estrutura poligonal de um modelo cujo resultado final é um novo modelo após a aplicação de uma tensão ou uma variação térmica no objeto.

A deformação é muito estudada em áreas como computação gráfica, animação e realidade virtual. Além disso, outras áreas como engenharia, entretenimento e projeto de ambientes têm utilizado objetos deformáveis para criar e editar superfícies de sólidos complexos [14].

As aplicações variam desde simulação de prospecção e extração de petróleo, simulação de desenvolvimento de carros, simulação de visitas a museus virtuais até simuladores de procedimentos médicos.

2.3.1 Métodos de Deformação

Métodos de deformação têm sido propostos e as três técnicas mais citadas na literatura de deformação de objetos em aplicações de RV são: Deformação de Forma Livre [20, 28], Métodos de Elementos Finitos [45] e Massa-Mola [53].

As próximas seções apresentam a explicação de cada uma dessas técnicas.

2.3.1.1 Deformação de Forma Livre

Deformação de forma livre (do inglês *Free Form Deformation* - FFD) importante método no auxilio de projetos geométricos e animações assistidos por computador. Esta técnica basicamente

consiste em deformar modelos geométricos solidos de uma maneira livre através de pontos de controle. Simplificadamente a deformação de forma livre consiste em envolver um objeto de qualquer representação grafica por um volume parametrizado. É feita então uma ligação entre o objeto e os pontos de controle do volume parametrizado [45].

O método FFD altera a estrutura do objeto deformando o espaço determinado por um volume parametrizado. O método cria uma representação geométrica (envolve o objeto com uma grade poligonal), tal que todas as transformações são aplicadas nesta representação, refletindo assim a deformação no objeto [28, 20].

A Figura 2.13 demonstra a deformação através da modificação na malha poligonal externa ao objeto.

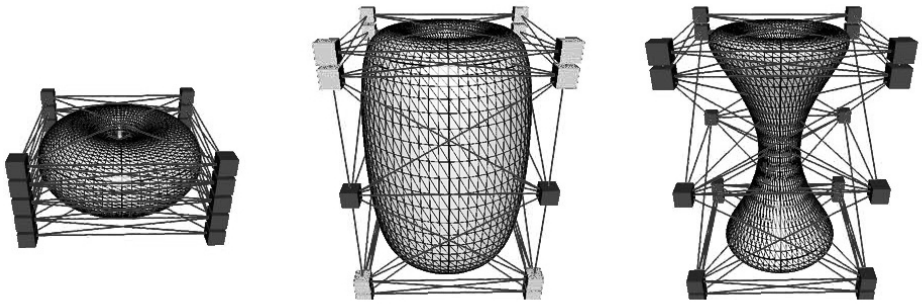

Figura 2.13: Exemplo de deformação com FFD [28].

Observa-se que as deformações são obtidas a partir de uma manipulação indireta dos pontos do modelo. Estabelecer pontos de controle como coeficiente da função de deformação faz com que cada movimento realizado sobre os mesmos interfira diretamente na localização dos vértices do modelo, conforme figura 2.14(c).

A deformação pode ser feita em outras bases polinomiais, tais como produto tensorial B-Splines ou produtos não tensoriais polinomiais de Bernstein [56].

Sederberg [56] diz que o método FFD não é uma técnica intuitiva, pois consiste em envolver os objetos deformados em um simples cubo para se ter o controle dos seus pontos, sendo que esses são manipulados para deformar o espaço circunvizinho e induzir deformações nos objetos.

Choi et al. [13] lembram que os movimentos de controle dos pontos e as correspondentes mudanças na estrutura dos objetos devem estar ligados para obter o resultado de deformações esperado. Mesmo sendo versátil, esse método é difícil para limitar as deformações para pequenas regiões, pois somente os objetos inseridos dentro das grades são deformados globalmente. Eles afirmam ainda

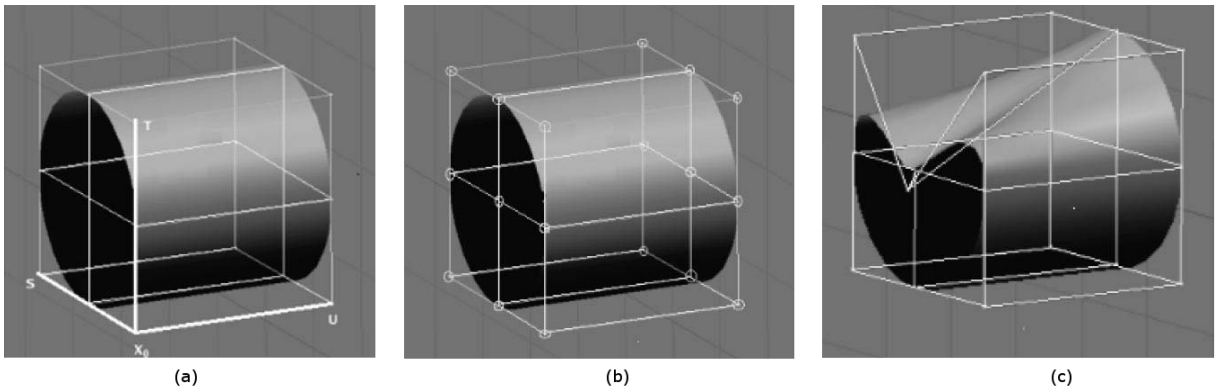

Figura 2.14: Passos do método FFD. (a) sistema local (S, T, U); (b) distribuição dos pontos de controle; (c) objeto deformado [12].

que o processo é considerado tedioso, pois sempre que são requeridas modificações subseqüentes são necessárias especificações explícitas para se aplicar deformações, ou seja, caso a deformação seja requerida em objetos pequenos, é necessário se estudar como será deformada a parte desejada.

Como o método FFD desconsidera propriedades físicas para deformação de objetos, essa técnica acaba sendo limitada e custosa, pois não permite a manipulação dos objetos que compõem a cena, não sendo então uma técnica satisfatória para aplicações de RV.

2.3.1.2 Método de Elementos Finitos

O Método de Elementos Finitos (do inglês *Finite Element Method* - FEM) utiliza técnicas matemáticas para encontrar soluções aproximadas para equações diferenciais parciais. A técnica subdivide o domínio do problema, resultando em partes menores de malhas poligonais. A técnica tem por objetivo reduzir complicadas equações diferenciais em um grupo de equações algébricas que poderão ser solucionadas numericamente [45].

2.3.1.3 Massa-Mola

O método Massa-Mola (do inglês, *Mass Spring* - *MS*) está relacionado aos sistemas do tipo massa-mola considerados na área de Mecânica. Ele permite a remodelagem de objetos através de pontos de massa conectados por molas. Cada um dos pontos de massa é o mapeamento de um ponto específico do objeto. O deslocamento desses pontos descreve a deformação do objeto. A figura 2.15 ilustra o método massa-mola.

A mola é uma estrutura que possui elasticidade permitindo assim a sua deformação quando uma

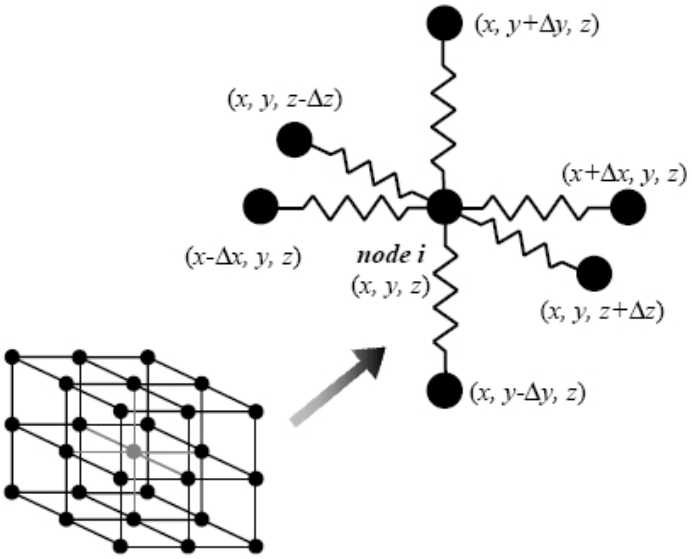

Figura 2.15: Demonstração da conexão por molas do nó central com seus vizinhos [13].

pressão é exercida sobre ela, possui uma força linear e é baseada na lei de Hooke [53]:

Técnica de fácil implementação e compreensão, permitindo também a interatividade e simlação em tempo real [28]. Uma das desvantagens é se o modelo possui uma alta complexidade com grandes quantidades de vértices pode haver uma sobrecarga de memória, tornando o processamento e a manipulação desses objetos na cena lentos e os resultados n ao sejam satisfatórios [28].

Sistemas massa-mola têm sido usados em várias aplicações tais como animações faciais, representações de ações musculares e simulações de procedimentos médicos [20]. Em aplicações de simulação de ambiente real, os objetos deformáveis compostos por este método permitem a incorporação de propriedades inerentes ao objetos reais a partir da inserção de constantes de mola e massa. A determinação dessas constantes não simples devido às propriedades do material [20].

Ainda nesses sistemas, os pontos de massa e os ligamentos de molas podem formar diferentes tipos de polígonos, tais como triângulos ou quadrados. No entanto, na grande maioria das aplicações destinadas à simulação de objetos com realismo, a estrutura triangular é a mais indicada, uma vez que permite a composição de geometrias mais suaves [45].

2.4 Estudo das ferramentas: JOGL e JAVA 3D

Com o crescimento da área de Realidade Virtual (RV),ferramentas de auxílio ao desenvolvimento de aplicações têm surgido nos últimos anos e várias delas têm sido disponibilizadas gratuitamente para

que aplicações de baixo custo sejam produzidas. A RV abrange um amplo conjunto de aplicações, desde jogos até simuladores para auxílio em treinamento médico.

Dessa forma, surge a necessidade de se buscar ferramentas de baixo custo que proporcionem recursos para aumentar o desempenho da aplicação e tornar o ambiente muito próximo ao mundo real. Aplicações voltadas ao treinamento médico, por exemplo, possuem uma grande necessidade de que a aplicação responda a uma ação pré-determinada em tempo real.

Há várias ferramentas disponíveis que possibilitam aplicações em RV, dentre elas a GL4Java, Java3D, LWJGL (*Lightweight Java Game Library*) e JOGL. Para a análise descrita, apenas as ferramentas Java 3D e JOGL foram consideradas por haver um conjunto muito grande de ferramentas possíveis de serem utilizadas e também porque o *framework* de teste foi escrito em Java 3D.

2.4.1 JAVA 3D

A API Java 3D (J3D) consiste em uma hierarquia de classes Java que serve como interface para o desenvolvimento de sistemas gráficos de renderização tridimensionais [33].

Elas possui construtores de alto nível que permitem a criação e manipulação de objetos geométricos, especificados em um universo virtual. Também possibilita a criação de universos virtuais com uma grande flexibilidade, em que as cenas são representadas por meio de grafos e os detalhes de sua visualização são gerenciados automaticamente [57, 58].

Assim, o desenvolvimento de um programa J3D se resume na criação de objetos e no seu posicionamento em um grafo de cena, que os combina em uma estrutura de árvore, podendo assim ser tratados tanto individualmente quanto em grupo.

Os grafos de cena são responsáveis pela especificação do conteúdo do universo virtual e pela forma como este é visualizado.

Um grafo de cena é uma estrutura de dados que descreve uma cena tridimensional na forma de nós e arestas. Os nós guardam todas as propriedades dos objetos presentes numa cena, como geometria, cor, transparência e textura. As arestas guardam as relações entre os nós. Alguns motivos podem levar ao uso de J3D como: alto nível de abstração, importação direta de modelos criados com outras ferramentas para a sua representação interna, definição de som 3D, facilidade para utilização de dispositivos de RV, integração com a Internet e possibilidade de conexão com sistemas de banco de dados, além da gratuidade e portabilidade da linguagem Java [57, 58].

A J3D foi construída com o objetivo de criar uma API que fosse independente de plataforma,

semelhante à *Virtual Reality Modeling Language* (VRML). Atualmente, a J3D consiste em uma API baseada nas bibliotecas gráficas OpenGL e DirectX e os programas podem ser escritos como aplicação, *applet* ou ambas. Nos últimos anos, várias aplicações foram desenvolvidas usando J3D, tais como jogos, comércio eletrônico, visualização de dados e elaboração de interfaces.

DirectX é uma biblioteca desenvolvida pela Microsoft para computadores baseados no sistema operacional Windows, que permite à equipe de desenvolvimento o acesso a recursos avançados de hardware sem a necessidade de escrever códigos específicos para esse fim [44]. Assim, a J3D torna-se uma biblioteca de mais alto nível de abstração quanto à programação gráfica quando comparada ao OpenGL ou DirectX.

Na J3D, há três meios de se implementar um AV, os quais são chamados de *universos*: o *SimpleUniverse*, o *VirtualUniverse* e o *ConfiguredUniverse*, detalhados a seguir. A classe *SimpleUniverse* permite que se crie facilmente um ambiente mínimo de usuário para um programa J3D ser executado. Elas cria todos os objetos necessários na parte *View* do grafo de cena. Especificamente, esta classe cria um nó *Locale*, um único nó do tipo *ViewingPlatform* e um objeto *Viewer* (ambos com valores padrões). Para muitas aplicações básicas de J3D, a *SimpleUniverse* provê todas as funcionalidades necessárias às aplicações. Aplicações mais sofisticadas podem requerer mais controles para adquirir funcionalidades extras [57, 58].

O *VirtualUniverse* consiste em uma lista de objetos Locale sendo que cada um deles contém uma coleção de nós de grafo de cenas existentes no universo. Uma aplicação ou *applet* pode ter mais do que um *VirtualUniverse*, mas muitas aplicações precisam somente de um. Universos virtuais são entidades separadas sendo que nenhum nó pode existir em mais de um universo virtual ao mesmo tempo, assim como os objetos em um universo virtual não são visíveis e nem interagem com objetos dentro de qualquer outro universo virtual. Um objeto do tipo *VirtualUniverse* define métodos para enumerar seu *Locale* e os remover do universo virtual [57, 58].

A classe *ConfiguredUniverse* cria todos os objetos *View* necessários do grafo de cena. Especificamente, ela cria um *Locale*, um ou mais *ViewingPlatforms* e pelo menos um objeto de *Viewer*. O *ConfiguredUniverse* pode criar um ambiente baseado nos conteúdos de um arquivo de configuração. Isso permite que uma aplicação seja executada sem mudança por uma gama larga de configurações, como janelas em PC convencionais, óculos 3D, telas de imersão únicas ou múltiplas telas, ou instalações de RV, inclusive em *Cave´s*, além de exibições *HMD´s* que permitem os seis graus de liberdade [57, 58].

Pelo menos um *Viewer* deve ser provido pela configuração. Se um *ViewingPlatform* não é provido, um padrão será criado e o *Viewer* será anexado a ele. Um arquivo de configuração pode ser especificado diretamente passando a URL a um construtor do *ConfiguredUniverse*. Alternativamente, um *ConfigContainer* pode ser criado primeiro a partir de um arquivo de configuração e, então, passar a um construtor do *ConfiguredUniverse* apropriado. Se um arquivo de configuração ou *container* não for criado, o *ConfiguredUniverse* cria um ambiente *viewing* padrão da mesma forma como o *SimpleUniverse*. Se forem informados um ou mais objetos *Canvas3D*, ele os usará em vez de criar o novo. Todos os construtores disponíveis para o *SimpleUniverse* também estão disponíveis no *ConfiguredUniverse* [57, 58].

A principal vantagem do J3D é a utilização dos grafos de cena. Com ela, o projeto de cena é enfatizado, em vez da renderização. Um grafo de cena naturalmente comporta elementos de gráficos completos como estruturas geométricas tridimensionais, comportamentos, modelos de iluminação e detecção de colisão, entre outros. No nível de implementação em J3D, o grafo de cena pode ser utilizado para agrupar formas com propriedades semelhantes e toda otimização que o usuário necessitar precisa ser feita em APIs de níveis mais baixos.

J3D possui ainda otimizações no grafos de cena e, em baixo nível, é construído sobre OpenGL ou DirectX, o que mostra o comprometimento com o desempenho. J3D permite ao programador agir sobre o grafo de cena, de forma total (chamada de modo imediato) ou parcial (chamada de modo misto). O modo imediato dá ao programador maior controle sobre a renderização e o gerenciamento de cena.

A implementação da J3D sobre DirectX e OpenGL possui um alto nível de portabilidade, reduzindo tempo e custos de desenvolvimento. Como J3D é uma API Java, ela possui como base a orientação a objetos e faz uso de todas as características do Java, como *threads*, tratamento de exceções, entre outras. J3D foi amplamente utilizada nos últimos anos e sua distribuição é acompanhada com cerca de 40 exemplos e vários tutoriais. Há também muitos artigos disponíveis que descrevem a utilização da J3D, incluindo alguns importantes como parte do projeto Mars Lander Hover [5] e um visualizador de imagens cerebrais integradas [50].

Os modelos de vistas do Java 3D separam o mundo virtual do mundo real, facilitando a reconfiguração das aplicações e a utilização de vários dispositivos de saída. Por meio de compressão geométrica, a J3D facilita a transmissão de grandes quantidades de dados entre aplicações, como jogos multiusuários.

J3D é muito lenta para jogos e aplicações que requerem um alto nível de renderização. Em 2002, foram realizados alguns experimentos e concluiu-se que, em um mesmo aplicativo construído com as linguagens C++, GL4Java e J3D, a versão em J3D é cerca de 2.5 vezes mais lenta, embora possuísse um tamanho (em número de classes) muito menor [43].

Apesar do alto nível de programação da J3D, há uma deficiência no nível de renderização. Por exemplo, J3D não pode realizar sombreamento de pixel e vértice. Pelo grafo de cena ser de alto nível, torna-se mais difícil aumentar a velocidade de aplicações desenvolvidas em J3D, diferentemente de aplicações construídas diretamente com DirectX e OpenGL.

Ao ocultar o baixo nível de programação da API gráfica, dificulta-se para o programador encontrar erros da API e codificar *drivers*.

2.4.2 JOGL

API JOGL (Java OpenGL) provê a ligação entre a biblioteca gráfica OpenGL e a linguagem Java. A versão atual do pacote consegue utilizar todos os métodos da biblioteca OpenGL até a versão 2.0. Ela incorpora características de outros *bindings* com GL4Java, LWJGL e Magician [33].

A API JOGL permite o acesso a muitas funcionalidades existentes na linguagem C, como a biblioteca de sistema de janelas GLUT, além de integrar facilmente as APIs Java que são padrões de gerenciamento de janelas como a AWT e a Swing [32].

Ela estabelece o GLCanvas como principal *widget* AWT, um componente que suporta aceleração por hardware e que tem por objetivo ser o *widget* primário utilizado em uma aplicação. O GLJPanel é o *widget* Swing que suporta aceleração por hardware.

Por meio da utilização do GLJPanel, o JOGL é integrado ao Swing e o componente GLJPanel que se sobrepõe aos componentes do Swing. JOGL renderiza a cena diretamente sobre uma área OpenGL utilizada pelo *pipeline* Java2D OpenGL, oferecendo aceleração máxima e eliminando a releitura de um *frame buffer*, utilizado em algumas outras APIs (e até mesmo no JOGL em versões anteriores).

Esse componente JOGL (GLJPanel) é tão rápido quanto o GLCanvas e oferece completa integração com o Java2D. Com isso, pode-se [11]:

1. sobrepor componentes Swing (menus leves, tooltipos e outros *widgets*) no topo da renderização OpenGL.

2. usar gráficos 3D onde se utilizaria normalmente um *widget* Swing (dentro de um Jtable ou

JTree).

3. desenhar gráficos OpenGL 3D acima da renderização Java2D. Jgears [] oferece um exemplo em que o fundo é desenhado com Java2D GradientPaint, enquanto um GLJPanel translúcido com um fundo zero-alfa desenha uma engrenagem.

4. desenhar gráficos Java2D sobre renderização OpenGL 3D. Jgears novamente oferece um exemplo deste caso: os ícones e o contador de quadros por segundo são desenhados sobre as engrenagens utilizando Java2D.

Mesmo em aplicações que não utilizavam essas características (GLJPanel), quase nenhuma mudança é necessária, devendo-se apenas adicionar um GLEventListener a um GLJPanel ao invés de um GLCanvas [55].

Levando-se em conta o fator desempenho, o componente GLCanvas ainda oferece um melhor resultado que o GLJPanel, pois o mecanismo de *swapping* implementado neste componente ainda é mais rápido que o encontrado no Swing, apesar de, nas versões recentes, a velocidade do GLJPanel ter sido melhorada com a inclusão do Java2D/OpenGL *pipeline* (neste caso, implementado internamente no Java2D por uma operação de cópia de pixel [11]), discutido anteriormente [15]. Entretanto, o GLCanvas pode ser utilizado em quase todos os tipos de aplicações, exceto aqueles que utilizam JinternalFrames.

O acesso à base da API do C OpenGL é realizado por meio da Java Native Interface (JNI), diferentemente de outras bibliotecas de *wrappers* que apenas expõem o OpenGL procedural através de alguns métodos em algumas classes, ao invés de mapear as funcionalidades do OpenGL no paradigma de orientação a objeto.

A maioria do código da JOGL foi gerado automaticamente dos arquivos de cabeçalho do C OpenGL por meio de uma ferramenta de conversão Gluegen que foi desenvolvida especialmente para facilitar a criação da JOGL [16, 32, 39, 31]. Conforme [32], o Java Web Start é recomendado como veículo de distribuição para aplicações do tipo JOGL. Há um instalador *applet* dentro do pacote JOGL que possibilita a extensão de *applets* não assinadas que usam JOGL em navegadores e JREs tão antigas quanto à versão 1.4.2, que foi a primeira versão do Java suportada pelo JOGL.

A API JOGL vem sendo bastante difundida hoje em dia, principalmente na área de jogos eletrônicos, onde teve sua origem [15]. Possui algumas aplicações científicas, mas ainda é pouco encontrado nessas áreas [1].

As vantagens da JOGL são, na maioria, as mesmas encontradas na biblioteca OpenGL, já que a JOGL é apenas GL. JOGL provê um modelo gráfico procedural. O modelo procedural é mais intuitivo a programadores gráficos com mais experiência, já que historicamente os algoritmos e métodos são procedurais.

JOGL provê um acesso direto ao *pipeline* de renderização. Ele permite que os programadores especifiquem exatamente como os gráficos devem ser renderizados. Outra vantagem da JOGL é sua otimização em hardware e software, o que torna possível sua utilização em computadores baratos, jogos de vídeo ou supercomputadores de última geração.

A proximidade da abordagem procedural em programação gráfica pode ser um ponto fraco para muitos programadores Java. Para programadores que receberam suas primeiras instruções em Java utilizando o conceito de orientação a objeto, os métodos procedurais da JOGL não combinam bem com a abordagem orientada a objeto. Alguns fabricantes otimizam seus produtos com extensões proprietárias (de OpenGL), fazendo com que seja necessário utilizá-las para uma determinada plataforma, o que diminui a portabilidade e eficiência para as outras plataformas.

Enquanto interfaces C para OpenGL são fáceis de se encontrar, interfaces Java ainda não são padronizadas e não são amplamente disponíveis. A exposição de detalhes internos do processo de renderização do OpenGL geralmente reduz a legibilidade dos programas gráficos, o que dificulta as atividades de desenvolvimento e manutenção dos programas.

CAPÍTULO 3

METODOLOGIA DO TRABALHO

Este capítulo descreve a metodologia para criação do ambiente virtual interativo, incluindo as classes e métodos para detecção de colisão e deformação de objetos, seus diagramas e pseudocódigos correspondentes.

3.1 Detecção de Colisão dos Objetos

A criação de um método de detecção de colisão de objetos é um grande desafio, pois na literatura já existem muitos métodos propostos e com boa qualidade. Apesar do grande número de algoritmos para a detecção de colisão, não há ainda um método que combine satisfatoriamente dois aspectos importantes na detecção de colisão: precisão e desempenho.

Com a idéia de simular um ambiente de treinamento médico em um AV, como uma ferramenta de baixo custo, a necessidade de se ter uma precisão próxima à real é alta, já que a ferramenta tem como objetivo treinar futuros profissionais da área da saúde. Para isso, o desempenho da ferramenta deve ser alto para permitir a simulação em tempo real dos eventos que ocorrem no AV.

Para a simulação do ambiente de treinamento médico, implementou-se um método de deformação para demonstrar a eficácia do ambiente e se as características de precisão e desempenho são atingidas.

O método de colisão foi baseado no algoritmo desenvolvido por Antonio [2] e o método de deformação foi adaptado do *framework* ViMet [47], no qual a classe de deformação desenvolvida deverá ser futuramente integrada. Para a construção do AV, utilizou-se o pacote JOGL [31, 32].

O método de colisão compreende a divisão espacial, a computação da distância incremental e a interpolação de faces. A figura 3.1 mostra um diagrama com as principais etapas do método de colisão desenvolvido, enumeradas a seguir:

- os objetos são inseridos na cena, ou seja, no AV;
- o AV ou cena é dividido em octantes, utilizando o conceito de divisão espacial (*octrees*);
- os objetos na AV são distribuídos nos octantes;

- a cada movimento do objeto é utilizada a computação incremental da distância para verificar se os objetos estão no mesmo octante;
- teste de colisão face a face.

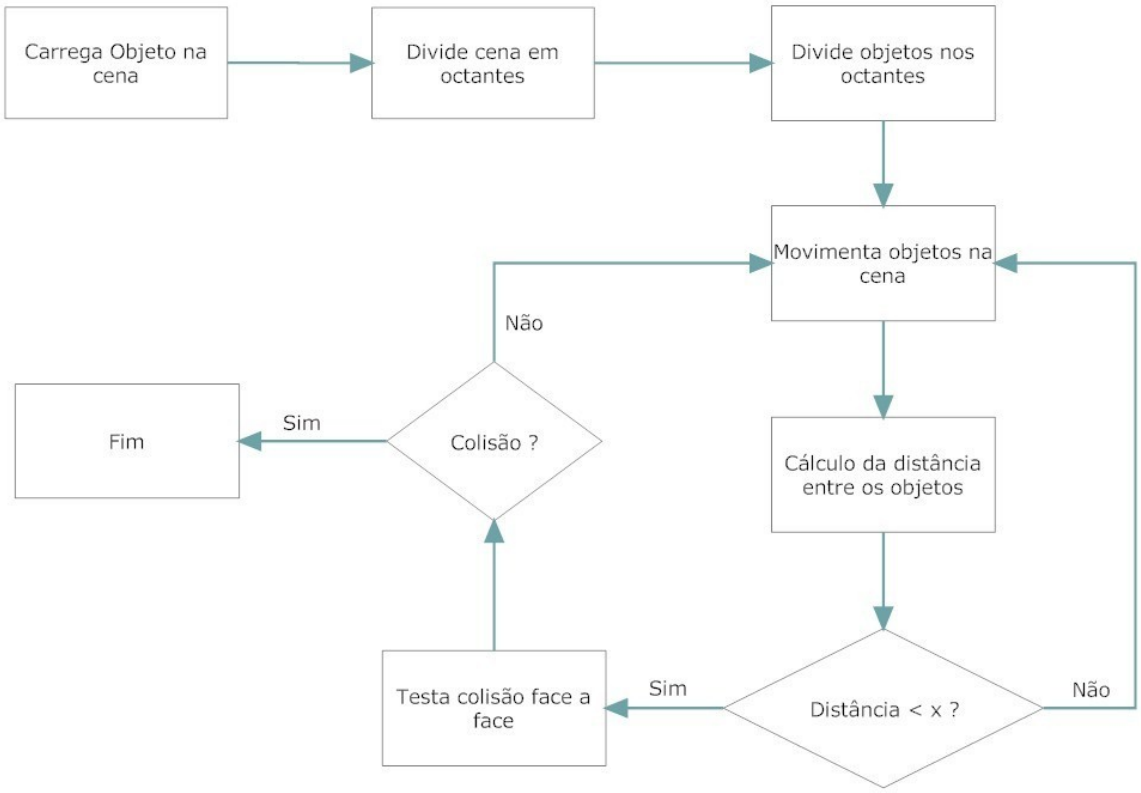

Figura 3.1: Diagrama ilustrando os passos do método de colisão.

Nesta classe de detecção de colisão são implementadas funções (métodos) que são responsáveis por carregar os objetos na cena, dividir a cena em octantes, dividir objetos ou faces nos octantes, cálculo de colisão e verificação da colisão entre os objetos.

Para carregar um objeto na cena, o ambiente desenvolvido utiliza modelos no formato OBJ [48] ou 3ds [4], que consiste em um arquivo contendo informações básicas do modelo 3D: vértices, faces e normais. Com esses dados é possível recriar o modelo no ambiente. Os vértices são guardados em uma lista que contém as coordenadas x, y e z. O apêndice A descreve mais detalhadamente esses formatos.

Para a lista de faces, apenas os índices das faces do objeto são armazenados. Por exemplo, se os vértices V_1, V_2, V_3 pertencem à face F_1 com V_1 na posição 1 da lista de vértices, V_2 na posição 2 e V_3 na posição 3, então a lista de faces recebe a seguinte informação: $F_1[1].x = 1$,$F_1[1].y = 2$ e $F_1[1].z = 3$. Assim, o armazenamento de informações é reduzido e também é eliminada a redundância

de informações (coordenadas repetidas de vértices entre diferentes faces).

O cálculo das normais é efetuado para cada face dos objetos. O vetor normal (x_N, y_N, z_N) de cada face é armazenado em uma lista de normais. O cálculo das normais pode ser expresso como:

$$x_N = (V_1.y * V_2.z) - (V_1.z * V_2.y) \tag{3.1}$$

$$y_N = (V_1.z * V_2.x) - (V_1.x * V_2.z) \tag{3.2}$$

$$z_N = (V_1.x * V_2.y) - (V_1.y * V_2.x) \tag{3.3}$$

Após carregado o objeto na cena, a mesma é dividida em octantes. Essas partes servirão para identificar onde o objeto (ou parte dele) se encontra, facilitando a verificação da colisão.

A divisão da cena é feita de acordo com a posição dos objetos, tal que os objetos mais afastados na cena são detectados e suas coordenadas máximas e mínimas são usadas na subdivisão. A figura 3.2 ilustra a divisão da cena.

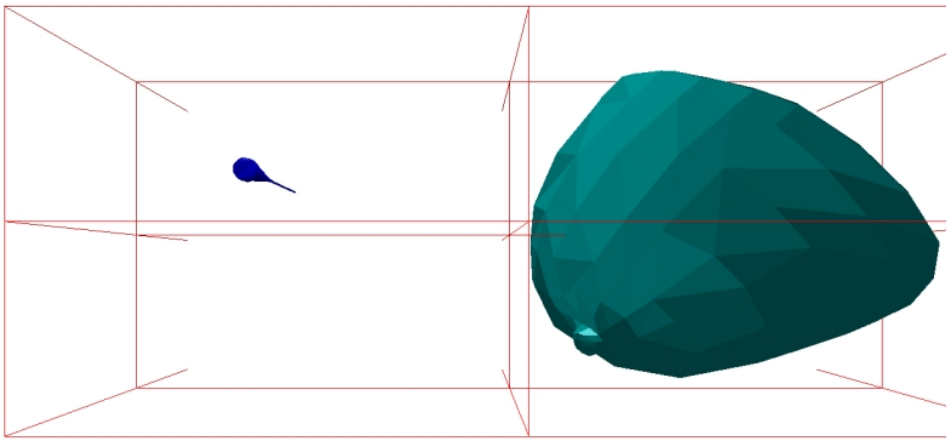

Figura 3.2: Divisão do espaço utilizando *octree*.

Após a divisão da cena, o posicionamento dos objetos em cada octante é realizado. Se um objeto ocupar mais que um octante, as faces do objeto são divididas entre os octantes.

Ao movimentar um objeto na cena é possível verificar como esses octantes são dinamicamente alterados. Se dois objetos estiverem no mesmo octante, uma nova subdivisão do mesmo é feita até que o objeto ou suas faces não compartilhem esse octante com outro objeto. Nessa fase é realizado o cálculo da distância entre os objetos. Se a distância entre eles for menor que DIÂMETRO_MÍNIMO ou o número de subdivisões da *octree* for maior que ALTURA_MÁXIMA, sendo que DIÂMETRO_MÍNIMO e ALTURA_MÁXIMA são constantes definidas na implementação,

verifica-se realmente há colisão entre os objetos.

Para verificar a colisão entre os objetos, o método inclui os seguintes passos: verificação da coplanaridade dos triângulos, teste entre os pontos dos triângulos e teste de um ponto sobre o triângulo. Esse método foi baseado no algoritmo de Antonio [2].

Os pseudocódigos de 5 a 8 mostram os passos seguidos para a obtenção da colisão no ambiente virtual.

```
início
    Main() {
    constroiCena();
    enquanto renderCena faça
        divideCenaEmOctree();
        renderizaCena();
        verificaColisao();
    fim
    }
fim
```

Algoritmo 5: Módulo principal.

```
início
    divideCenaEmOctree() {
    calculaRaizOctree;
    subdivideOctree posicionaFacesNaOctree;
    }
fim
```

Algoritmo 6: Módulo de divisão da cena e posicionamento de faces na *octree*.

```
início
    verificaColisao() {
    verificaAlturaMaximaOctree;
    verificaDiametroOctree;
    verificaColisaoFaceFace();
    }
fim
```

Algoritmo 7: Módulo de verificação de colisão.

```
início
    verificaColisaoFaceFace() {
    verificaCoplanaridade;
    testaArestasEntreTriangulos;
    testaPontoNoTriangulo;
    }
fim
```

Algoritmo 8: Módulo de colisão baseado em [2].

3.2 Deformação dos Objetos

O método de deformação é composto pelos seguintes passos, os quais são ilustrados no diagrama da figura 3.3:

- método recebe faces em colisão;
- busca do ponto mais próximo de colisão entre os objetos;
- procura pelos vértices imediatamente adjacentes ao ponto escolhido (localizados na camada 1);
- procura vizinhos da camada anterior; esse passo é realizado recursivamente até que o número de camadas definidas para a deformação seja atingida. A escolha do número de camadas é feita através do cálculo da força exercida sobre o ponto;
- deforma o objeto no ponto de colisão e em suas camadas subseqüentes.

A classe de deformação simula a alteração na forma de um objeto reposicionando os vértices do mesmo e de regiões vizinhas. O conceito de massa-mola é utilizado como método de deformação. Para a obtenção da fórmula de força da mola é preciso considerar que os nós de massa do objeto estão conectados por mola [13], como visto na figura 3.4.

Cada mola deve obedecer à elasticidade de corpos (lei de Hooke), que calcula a deformação causada pela força exercida sobre um corpo, tal que a força é igual ao deslocamento da massa a partir do seu ponto de equilíbrio multiplicada pela característica constante da mola ou do corpo que

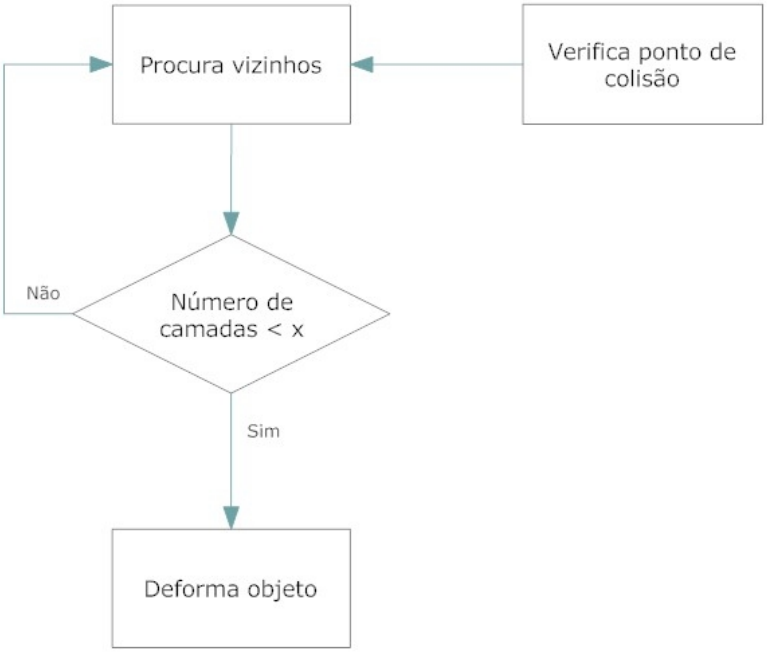

Figura 3.3: Diagrama ilustrando os passos do método de deformação.

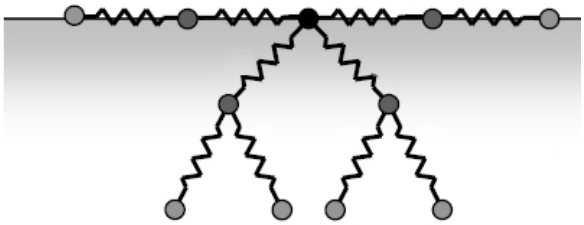

Figura 3.4: Massas conectadas por molas [13].

sofrerá deformação. Vale notar que a força produzida pela mola é diretamente proporcional ao seu deslocamento do estado inicial.

A equação 3.4 representa o cálculo da deformação, dada por

$$F = k\,\Delta l \tag{3.4}$$

em que F é a força elástica (Newton), k é uma constante positiva denominada constante elástica da mola (Newton/metro) e Δl é a deformação da mola (metros). A constante elástica traduz a rigidez da mola. Quanto maior for a constante elástica da mola, maior será sua dureza.

Essa expressão é sempre calculada quando a mola sai de seu estado de equilíbrio (estado natural da mola) e passa a estar comprimida ou esticada. A lei de Hooke pode ser utilizada desde que o limite elástico do material não seja excedido. O comportamento elástico dos materiais segue o regime elástico na lei de Hooke apenas até um determinado valor de força. Após este valor, a relação de proporcionalidade deixa de ser definida. Se essa força continuar a aumentar, o corpo perde a sua elasticidade e a deformação passa a ser permanente (inelástico), chegando a romper o material.

A figura 3.5 ilustra uma mola com uma extremidade ligada a uma parede e a outra ligada a um corpo, sendo l_0 o comprimento natural da mola e por l o seu comprimento quando é esticada ou encolhida. A variável x relaciona-se a esses comprimentos através de $x = l - l_0$.

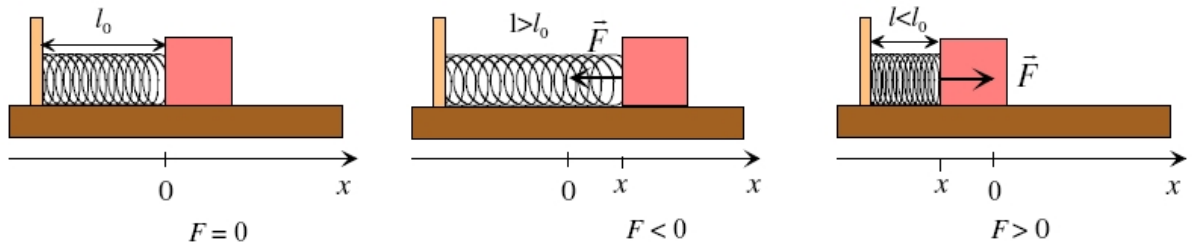

Figura 3.5: Demonstração da ação da lei de Hooke (massa-mola) [19].

O método de deformação possui etapas de processamento, sendo elas: escolha do ponto de colisão, busca dos vizinhos do ponto de colisão e deformação das camadas. Uma camada é definida pelos vizinhos do ponto de colisão.

A partir das faces detectadas no método de colisão, realiza-se o cálculo da distância Euclidiana entre os pontos das faces, tal que a menor distância é escolhida como nó raiz para receber a de-

formação. Após essa escolha é feita uma pesquisa na lista de faces buscando-se os nós vizinhos do nó raiz para a formação da primeira camada. A busca pelos vizinhos é feita recursivamente até ser interrompida pelo método de deformação.

Os vértices que estão diretamente conectados ao vértice raiz pertencem à primeira camada de deformação. A segunda camada é composta pelos vértices que estão conectados aos vértices da primeira camada, ou seja, vértices secundários ao vértice raiz, e assim sucessivamente. Essas informações, ou seja, quais vértices pertencem a quais camadas, são armazenadas em uma estrutura de dados definida como vizinhos.

A figura 3.6 ilustra a escolha de vizinhos ao nó raiz e a formação das camadas. O nó raiz é representado pela cor azul, primeira camada em rosa, segunda camada em roxo e terceira camada em verde.

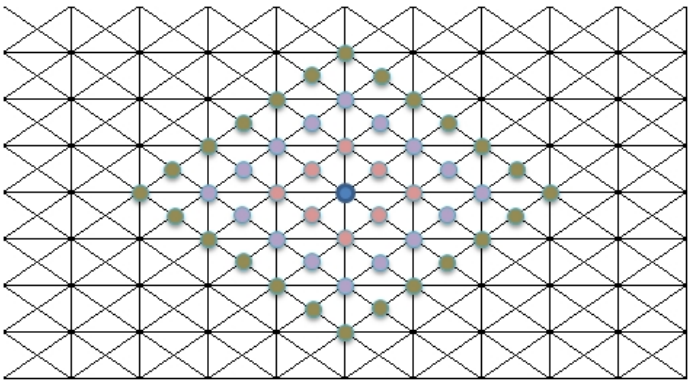

Figura 3.6: Formação de camadas na malha poligonal.

Ao ser aplicada uma força F no nó raiz, o comportamento dinâmico deste é regido pelas equações de Newton (Segunda Lei de Newton, $F = ma$). A aceleração é a derivada da velocidade v, em relação ao tempo, que, por sua vez, é a derivada de x em relação ao tempo

$$a = \frac{dv}{dt} = \frac{d^2x}{dt^2} \tag{3.5}$$

Combinando a equação de Newton com a expressão da força, tem-se

$$m\frac{d^2x}{dt^2} = -k\ \Delta t \tag{3.6}$$

Com isso, tem-se a determinação da variação sobre o tempo, implicando assim a posição final da mola em uma determinada variação do tempo [13]. Para deformar um objeto, é utilizada a

equação [13]

$$m_i \frac{d^2 u_i}{dt^2} + \alpha_i \frac{du_i}{dt} + \sum_{j \in \text{nós conectados}} \frac{k_{ij}(\|r_{ij}\| - l_{ij})}{\|r_{ij}\|} r_{ij} = F_i \tag{3.7}$$

em que m_i, u_i e α_i são, respectivamente, massa, posição e constante de amortecimento do nó raiz. O vetor que representa a distância entre o nó i e o nó j é dado por $r_{ij} = u_j \; u_i$ e l_{ij} é o tamanho natural da mola que conecta o nó i ao nó j.

Usando o método de diferenças finitas, seja Δt uma etapa da variação do tempo. A posição do nó i no instante $t + \Delta t$ é dada por $u_i(t + \Delta t)$ e esta é calculada tomando-se a posição do nó i no instante corrente t e a sua posição no instante anterior $t - \Delta t$.

As constantes de massa-mola, força e amortecimento são inseridas manualmente. A quantidade de camadas afetadas na deformação depende diretamente da força exercida no nó raiz e também dos parâmetros de constante da mola e amortecimento, uma vez que são eles que permitem uma modelagem matemática das estruturas físicas do objeto.

À medida em que os vértices são deslocados pela força externa, a força das molas afeta os vértices vizinhos. Assim, a deformação é propagada para as camadas, ou seja, quando um vértice se desloca da posição P para P', juntamente com ele são deslocados os demais vértices com os quais ele possui ligações.

Dessa forma, quando um ponto é deslocado, sua posição deve ser recalculada através da fórmula

$$\frac{k_{ij}(\|r_{ij}\| - l_{ij})}{r_{ij}} r_{ij} \tag{3.8}$$

que é o calculo da força de uma única mola, tendo como componentes específicos k que corresponde à constante da mola, r representando o vetor de distância entre os pontos i e j, e l que é o tamanho natural da mola.

Após os cálculos, esse valor se reflete no restante da fórmula apresentada anteriormente, deslocando os vértices vizinhos e gerando assim uma deformação do ponto. Quando o resultado deste cálculo é um valor maior que zero, isto significa que a força deve ser propagada para as próximas camadas; caso contrário, significa que a camada atual é a última a ser processada. A figura 3.7 ilustra o comportamento da força em relação às camadas.

O algoritmo 9 apresenta as principais etapas do método de deformação.

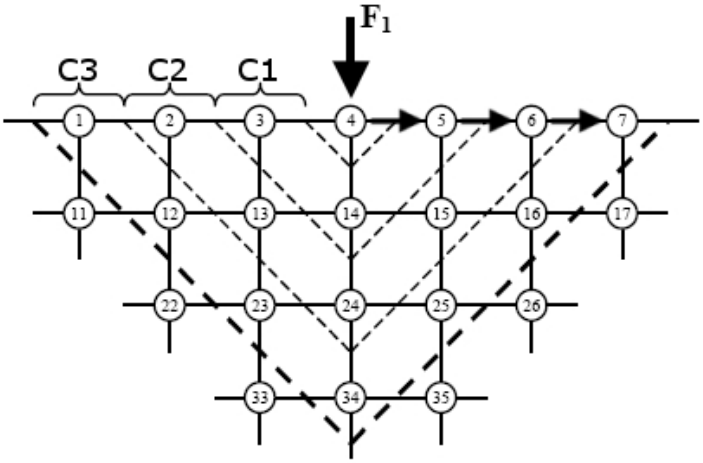

Figura 3.7: Força sendo aplicada ao nó raiz e sua propagação [13].

```
início
    deformacao(face1, face2) {
    selecionaNoRaiz(face1, face2);
    selecionaVizinhos;
    calculaForca;
    enquanto !Termina faça
        selcionaCamada;
        selecionaVizinhos;
        calculaForca;
    fim
    }
fim
```

Algoritmo 9: Módulo de deformação.

O apêndice B descreve as principais classes e métodos relacionados à detecção de colisão e à deformação de objetos que foram desenvolvidos no trabalho.

CAPÍTULO 4

RESULTADOS EXPERIMENTAIS

Neste capítulo são descritos os resultados da implementação dos métodos de colisão e deformação, incluindo as características da plataforma de desenvolvimento, os experimentos realizados, a captura das imagens e uma discussão dos resultados obtidos.

A partir desses experimentos, tornou-se possível analisar o desempenho e a precisão dos métodos desenvolvidos e que foram incorporados à ferramenta.

4.1 Critérios para os Testes

Alguns parâmetros foram definidos para a realização dos testes. Em aplicações cuja interação é efetuada com dispositivos não convencionais, tais como luvas e capacete, esses atributos pode ser dinamicamente ajustados.

No método de colisão foram realizados testes com os parâmetros diâmetro mínimo e altura máxima, enquanto no método de deformação, os valores da força aplicada foram alteradas. Os valores para a constante da mola (K), constante de amortecimento (D) e a massa (M) permaneceram os mesmos ao longo de todos os testes.

Os experimentos foram realizados de duas formas, um com apenas o método de colisão e o outro com a colisão e a deformação integradas. Como o ambiente possibilita a visualização dos objetos no ambiente com as malhas poligonais destacadas (*wireframe*) e esse recurso pode alterar a média de quadros por segundo (FPS, *frames per second*), os testes também mostraram a média de FPS com a visualização das malhas poligonais.

Os objetos carregados na cena estão no formato 3ds [4] ou obj [48] que consiste em um arquivo com informações sobre os vértices, faces e vetores normais dos polígonos que formam os objetos. O apêndice A descreve mais detalhadamente esses formatos.

Duas plataformas diferentes foram testadas nos experimentos para avaliar o desempenho dos métodos: um computador com sistema operacional Windows Vista com processador Intel Core2Duo T5250 1.50 GHz, 3 GB de memória RAM e placa de vídeo Intel x3100 *on-board* e um computador

com sistema operacional Linux Debian 2.6.18-6-686 com processador Intel Pentium 4 Xeon 3.20GHz, 2GB de memória RAM e placa de vídeo G-Force4 MX4000 de 64 bits.

4.2 Detecção de Colisão enre Objetos

Esta seção apresenta os resultados do método de colisão de objetos nas plataformas citadas anteriormente. Duas categorias de objetos foram utilizadas nos testes, a primeira formada por objetos regulares simples e a segunda formada por objetos complexos com grande número de faces poligonais.

4.2.1 Testes com Objetos Simples

Estes testes utilizaram objetos com baixo número de vértices e faces, a saber: Diamante com 18 vértices e 8 faces e Rosca com 119 vértices e 140 faces. A figura 4.1 mostra os dois objetos em suas posições iniciais na cena após serem carregados no ambiente.

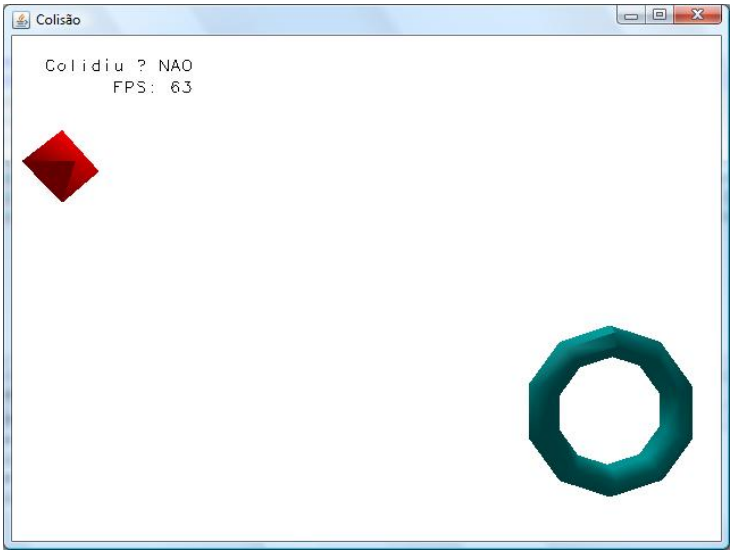

Figura 4.1: Objetos simples em estado inicial.

Na plataforma Windows, a taxa média de renderização para esses objetos estáticos na cena e sem colisão foi igual a 62 FPS e na plataforma Linux, a taxa média de renderização foi de 61 FPS, ou seja, taxas similares.

Desde a etapa inicial da aplicação que é carregar o objeto na cena, o cálculo da divisão do espaço (*octree*) é realizada. Esse cálculo consiste em verificar a posição dos objetos na cena e dividir a mesma em octantes, posicionando os objetos ou parte dele (faces do objeto) nos octantes.

A borda externa da *octree* é calculada pela posição dos objetos, verificando-se as coordenadas X, Y e Z mínimos e máximos dos vértices e, assim, determinando a caixa envolvente aos objetos.

A figura 4.2 mostra a mesma figura com a imagem da *octree* sobreposta aos objetos, podendo visualizar o posicionamento dos objetos em octantes diferentes. A *octree* é usada na etapa de verificação de colisão entre os objetos. Quando o número de nós-filhos da *octree* atinge o valor definido na aplicação ou o valor do octante em questão for menor que a distância mínima também definida na aplicação, a colisão então passa a ser verificada face a face.

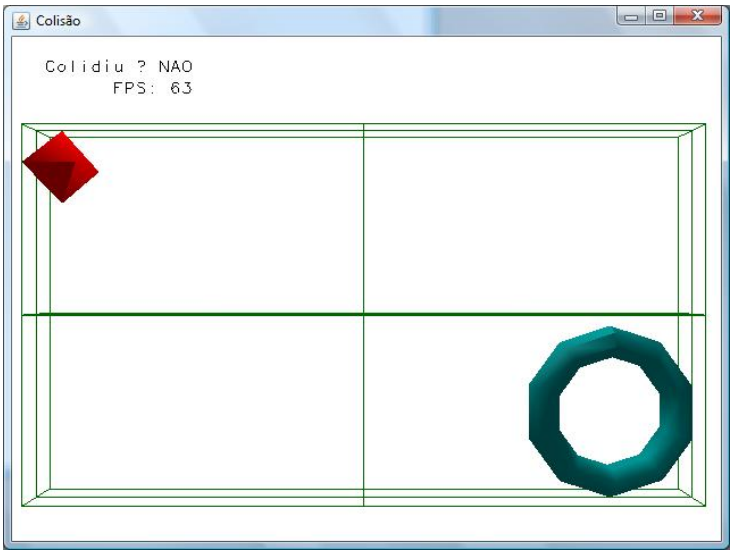

Figura 4.2: Objetos em estado inicial com *octree* sobreposta.

Ao adicionar a *octree* na visualização da cena não é notada diferença de processamento da cena anterior, que havia em sua visualização apenas os objetos Diamante e Rosca. Assim, a taxa média de renderização verificada na cena para a plataforma Windows foi novamente de 62 FPS e de 61 FPS para a plataforma Linux.

A figura 4.3 ilustra a visualização da mesma cena anterior com as malhas poligonais destacadas. A visualização das malhas poligonais influencia a taxa de renderização da aplicação, pois detalhes do objeto que não podem ser visualizados nas outras cenas, podem ser agora vistos.

A valor médio de FPS obtido para a visualização da cena com destaque as malhas poligonais foi de 62 FPS para ambas as plataformas Windows e Linux.

Com a movimentação de um objeto em direção a outro objeto, a aplicação pode dividir a cena em espaços menores com a *octree* até a possível colisão entre os objetos (verificação face a face). A figura 4.4 ilustra a aproximação dos objetos e a subdivisão do espaço, mas sem haver a colisão.

A taxa média de renderização apenas com o deslocamento dos objetos sem se colidirem e apenas

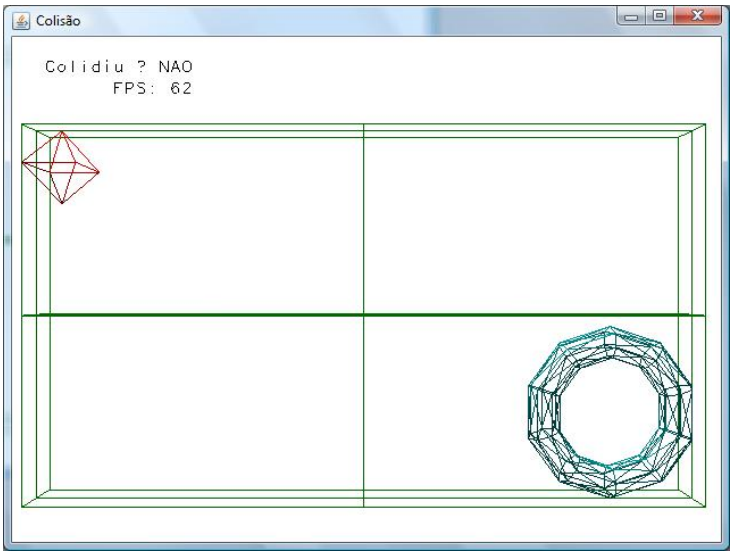

Figura 4.3: Objetos em estado inicial com malhas poligonais visíveis.

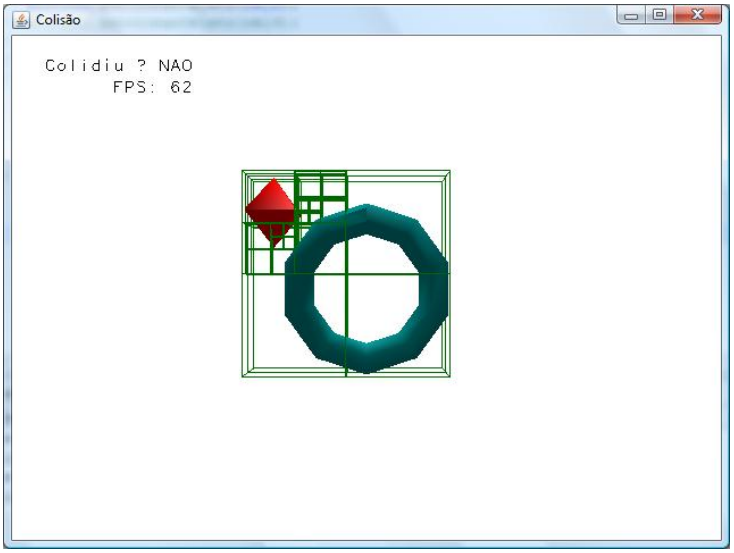

Figura 4.4: Movimentação de um objeto na cena.

com o recálculo do particionamento da cena foi igual a 62 FPS nas plataformas Windows e Linux, portanto, taxa similar para o teste anterior com os objetos estáticos na cena.

A figura 4.5 ilustra a cena de movimentação de um dos objetos com a visualização das malhas poligonais. Ao se comparar esta cena com a cena anterior, tem se que a taxa média de renderização não foi alterada, apesar das faces posteriores dos objetos Rosca e Diamante que estavam escondidas e não renderizadas aparecerem nesta cena.

Como um dos objetos escolhidos possui um orifício (Rosca), fez-se se então um teste no qual se introduz o Diamante nessa lacuna para verificar se há uma falsa detecçao de colisão. A realização desse teste é importante pois um grande desafio encontrado nos métodos de colisão é a detecção de

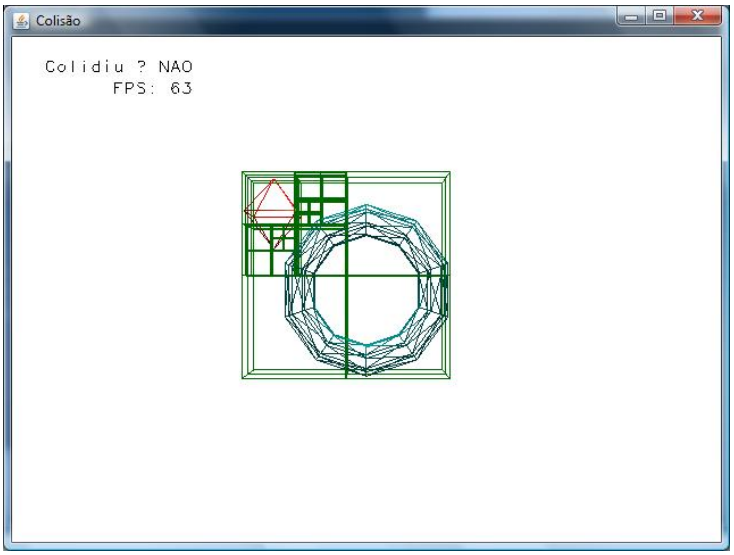

Figura 4.5: Movimentação na cena com malhas poligonais visíveis.

colisão em objetos côncavos, onde o espaço interno do objeto pode não colidir com outro objeto.

Nas figuras 4.6 a 4.8, o objeto Diamante está localizado exatamente no centro do objeto Rosca, demonstrando, assim, que a verificação de colisão é realizada corretamente.

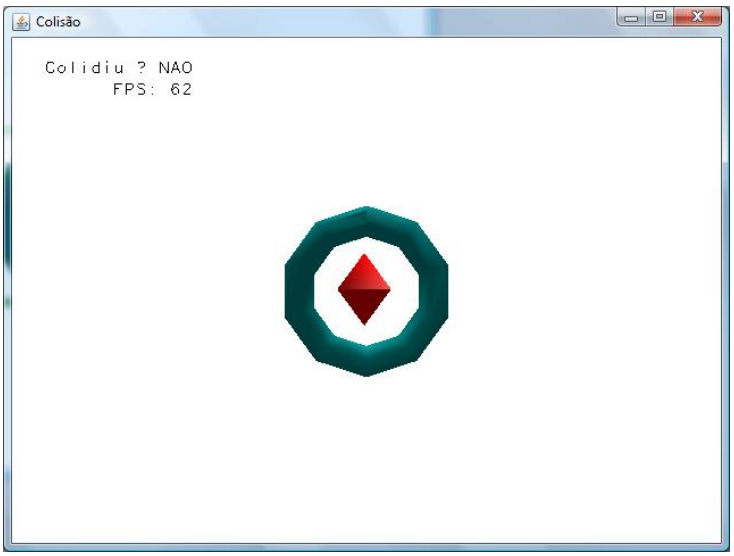

Figura 4.6: Objetos sem colisão.

Tem-se que, durante todo o processo de movimentar o objeto Diamante ao orificio do objeto Rosca, não há uma diminuição na quantidade média de renderização tanto para a plataforma Windows quanto para plataforma Linux, mantendo-se assim a média de 62 FPS para ambas as plataformas.

É importante resaltar que a movimentação do objeto Diamante através do objeto Rosca tem a mesma média de processamento que a simples movimentação do objeto Diamante pela cena.

A detecção de colisão é indicada para o usuário através de uma mensagem na cena, indicando

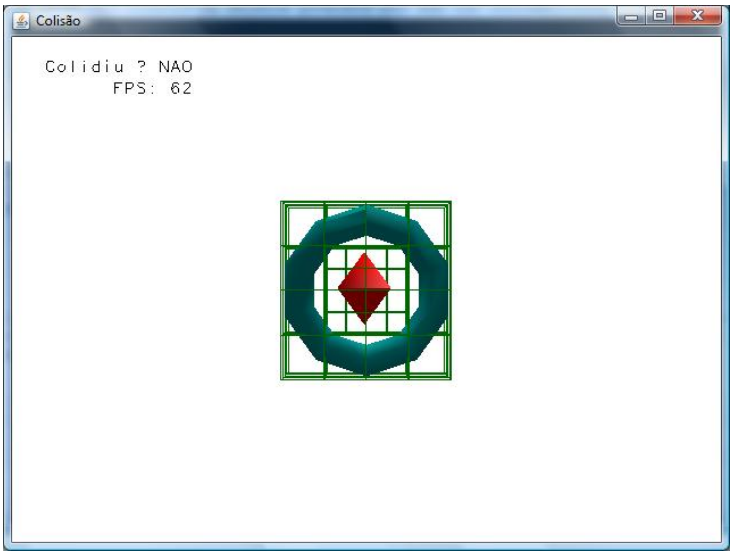

Figura 4.7: Objetos sem colisão sobrepostos com *octree*.

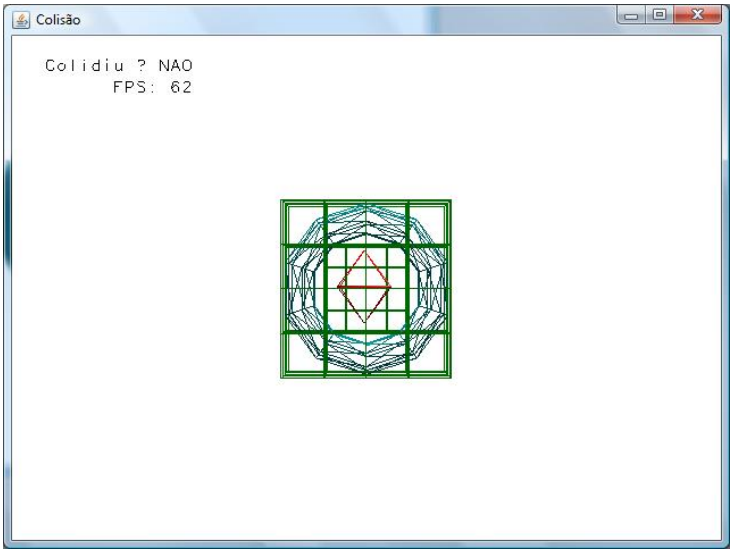

Figura 4.8: objetos sem colisão com malhas poligonais visíveis.

se houve ou não a colisão dos objetos na cena. O processo de detectar colisão no ambiente segue os passos de subdivisão hieráquica do espaço, distância entre os objetos e verificação face a face.

A figura 4.9 mostra os objetos antes de se colidirem, observando-se que existe um espaço mínimo entre os objetos. Esta cena tem como taxa média de renderização 62 FPS para a plataforma Windows e de 60 FPS para a plataforma Linux.

A figura 4.10 ilustra a colisão dos objetos na cena, enquanto a figura 4.11 mostra os objetos se colidindo em malhas poligonais. A taxa média de renderização foi de 62 FPS na plataforma Windows e de 60 FPS na plataforma Linux para a cena da figura 4.10 e uma taxa média de renderização de 60 FPS para a plataforma Windows e de 58 FPS para a plataforma Linux para a cena da figura 4.11.

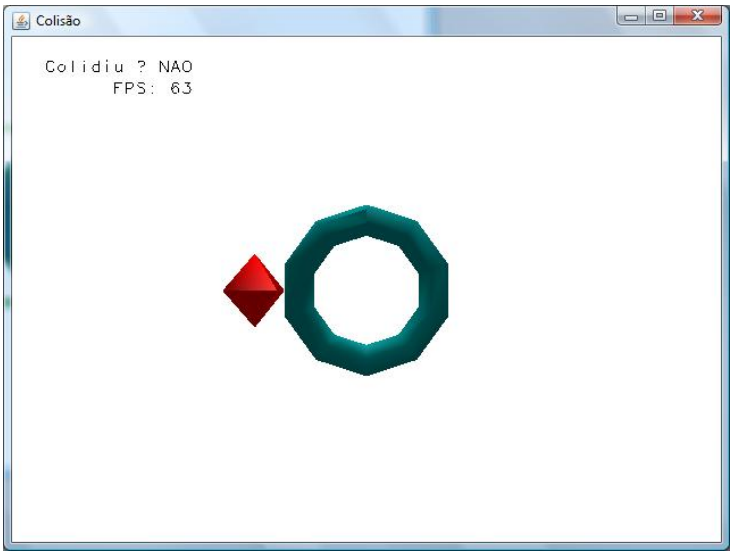

Figura 4.9: Objetos antes da colisão.

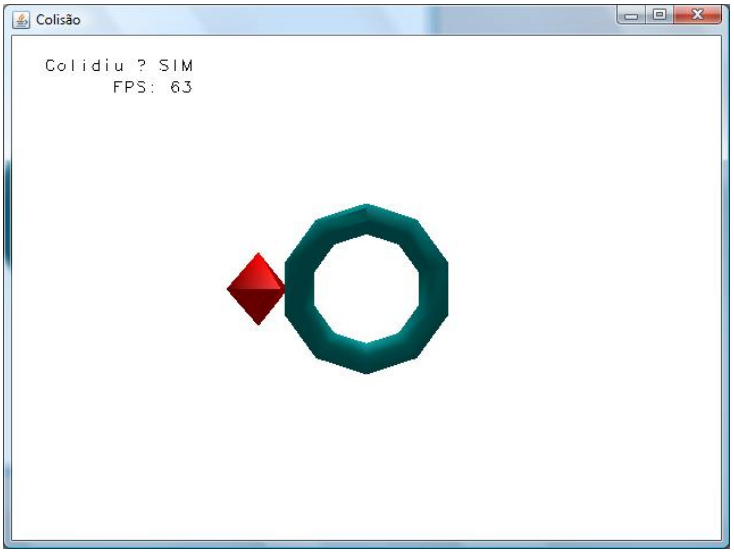

Figura 4.10: Objetos se colidindo.

A figura 4.12 ilustra uma ampliação da região de colisão. Nesta visualização é possivel mostrar a proximidade atingida entre os objetos para se obter uma resposta de colisão, tendo assim uma precisão bem próxima ao real.

4.2.2 Testes com Objetos Complexos

Nestes testes foram utilizados objetos para simulação do procedimento de punção em exames de biópsia. Este procedimento consiste na extração de pequenas partes de tecidos do órgão sob investigação. O material coletado é então enviado para exames patológicos para a elaboração do diagnóstico.

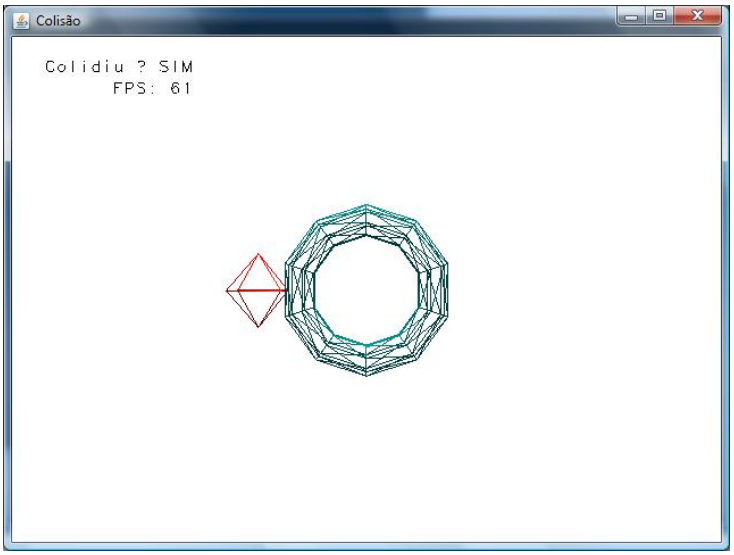

Figura 4.11: Colisão entre objetos com malhas poligonais visíveis.

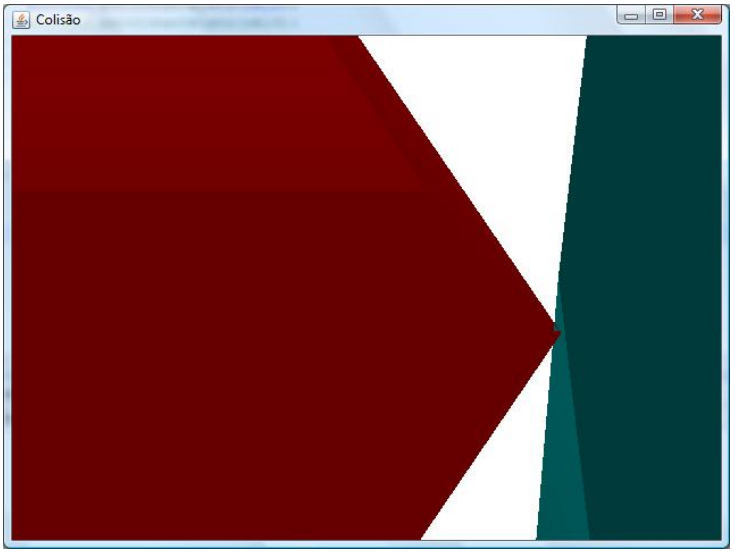

Figura 4.12: Detecção de colisão com ampliação.

Dois objetos foram modelados: um para representar o instrumento médico (neste trabalho, uma Seringa) e o outro para representar o órgão (neste trabalho, uma Mama). A quantidade de polígonos presentes na cena foi de 964 vértices e 1.817 faces para a Seringa e de 8.872 vértices e 17.490 faces para a Mama.

A figura 4.13 mostra os objetos em suas posições iniciais após serem carregados no ambiente. Inicialmente, a taxa de renderização está em torno de 28 FPS na plataforma Windows e de 27 FPS na plataforma Linux.

A figura 4.14 mostra os objetos na cena em posição inicial com a representação *octree*, onde se pode notar que, mesmo estando em sua posição inicial, a aplicação já determina em qual octante

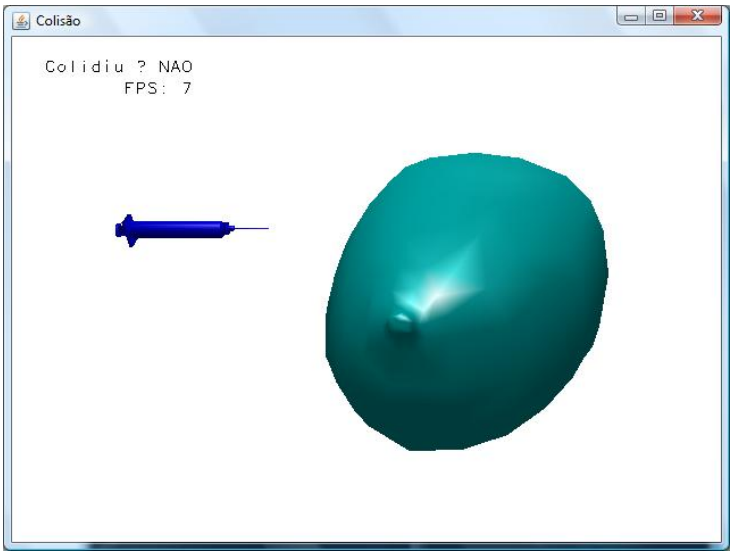

Figura 4.13: Objetos em posições iniciais.

cada objeto ou parte dele está inserido. Pode-se observar que, já neste primeiro momento, uma parte da mama estava no mesmo octante que a seringa. Com isso, esse octante foi dividido em outros 8 octantes, fazendo com que cada octante contenha apenas partes do mesmo objeto. A taxa média de renderização foi compativel com a mostrada na cena anterior de 28 FPS na plataforma Windows e de 27 FPS na plataforma Linux.

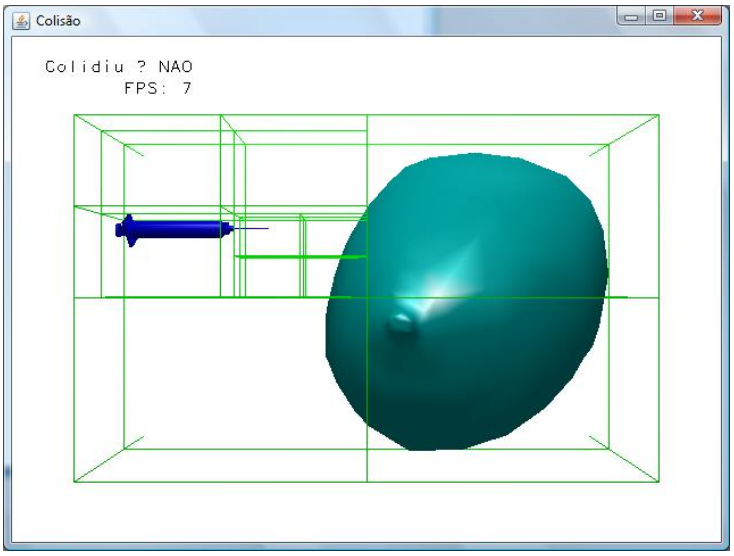

Figura 4.14: Objetos em posições iniciais com sobreposição da *octree*.

Ao apresentar os objetos com malhas poligonais visíveis, tem-se uma taxa média de 16 FPS na plataforma Windows e uma média de 15 FPS na plataforma Linux. A figura 4.15 mostra a cena com os objetos em suas posições iniciais.

Ao movimentar um dos objetos na cena, sem que haja uma colisão entre eles, com a subdivisão

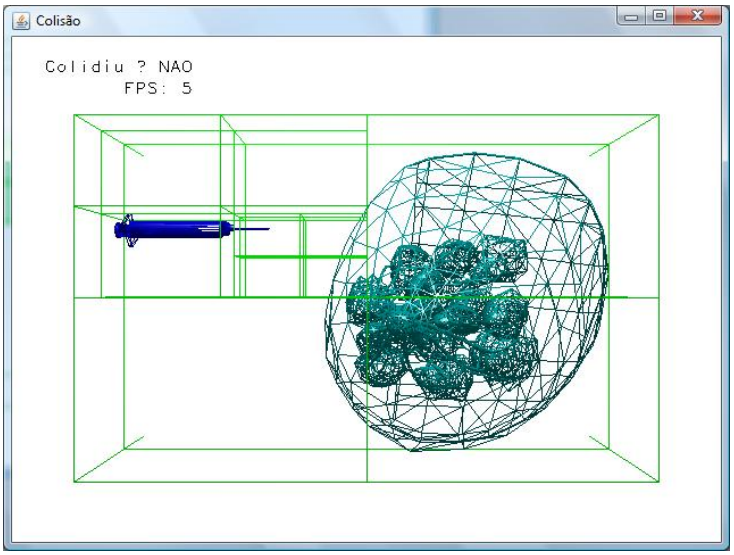

Figura 4.15: Objetos em posições iniciais com malhas poligonais visíveis.

hierárquica do espaço (*octree*) na cena, obteve-se uma média de 14 FPS para a plataforma Windows e de 13 FPS para a plataforma Linux. A figura 4.16 ilustra essa movimentação mostrando também a subdivisão da cena.

Pode-se observar também na na figura 4.16 que a divisão da cena foi bem detalhada, posicionando partes do mesmo objeto em vários octantes, sendo que, para se detectar a colisão, é necessária uma maior subdivisão.

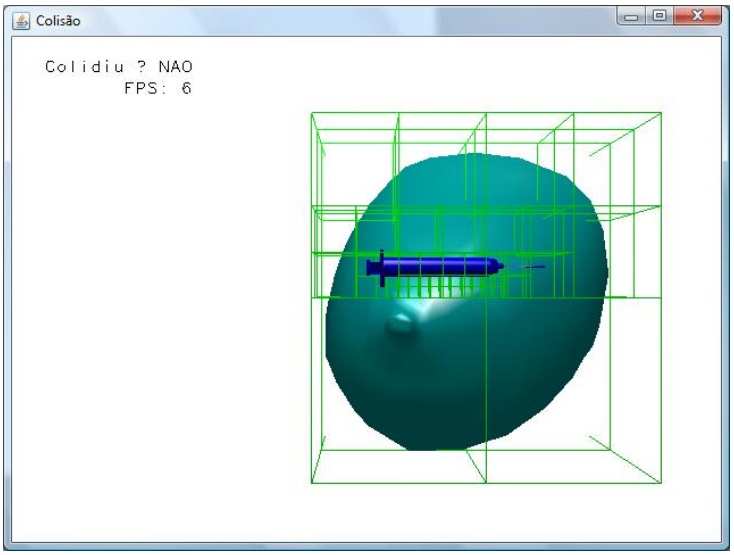

Figura 4.16: Movimentação de objeto na cena.

Havendo uma colisão entre os objetos, verifica-se uma taxa de 24 FPS para a plataforma Windows e de 23 FPS para Linux. A figura 4.17 ilustra o momento da colisão entre os objetos. A colisão é verificada após a divisão recursiva dos octantes até que se atinja um tamanho mínimo. Após essa

divisão, o cálculo de sobreposição de faces é efetuado. Havendo colisão entre as faces ou se um valor pré-especificado de distância mínima for alcançado entre as faces, então a colisão é detectada.

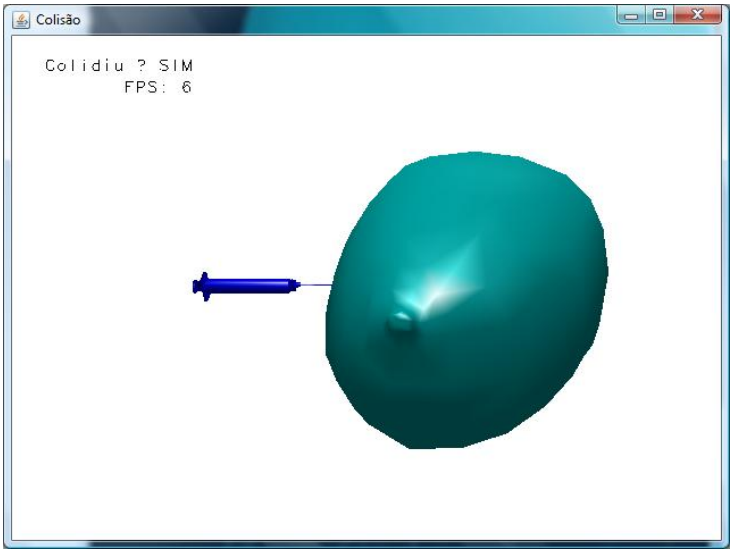

Figura 4.17: Colisão entre objetos.

A figura 4.18 mostra a cena com sobreposição da *octree* durante a detecção da colisão dos objetos, podendo ser observada a quantidade de octantes.

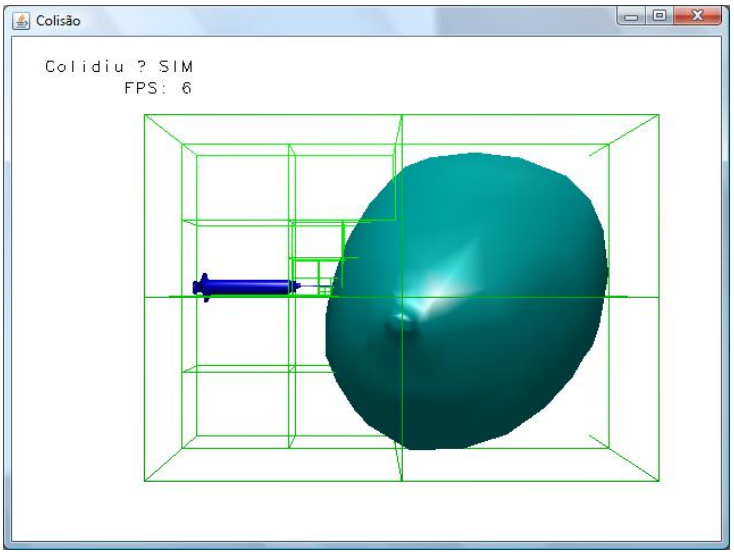

Figura 4.18: Colisão entre os objetos com sobreposição da *octree*.

A figura 4.19 mostra uma região ampliada próxima ao ponto de colisão detectado pelo método proposto. Nesta visualização, pode-se verificar a proximidade dos objetos, tendo então que a colisão ocorre muito próxima à face real do outro objeto.

A figura 4.20 mostra uma ampliação da região de detecção com as malhas poligonais visíveis. Nesta visualização, pode ser observado com melhor precisão que a extremidade do objeto Seringa

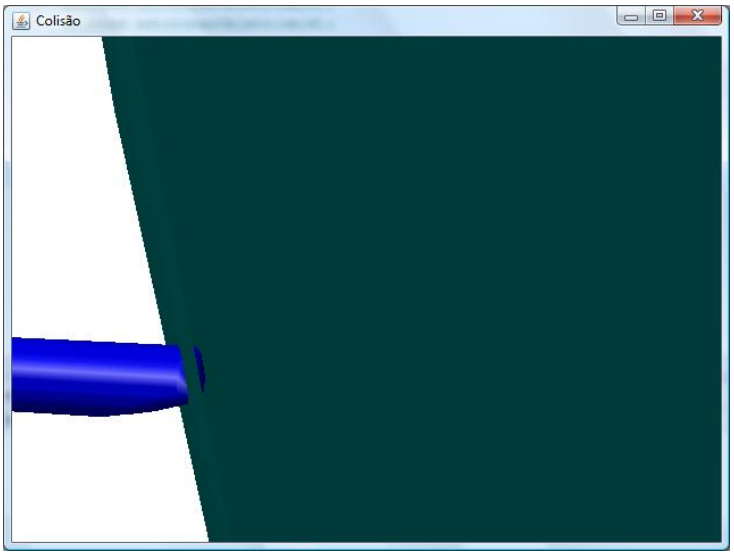

Figura 4.19: Detalhe da colisão.

praticamente toca a face do objeto Mama.

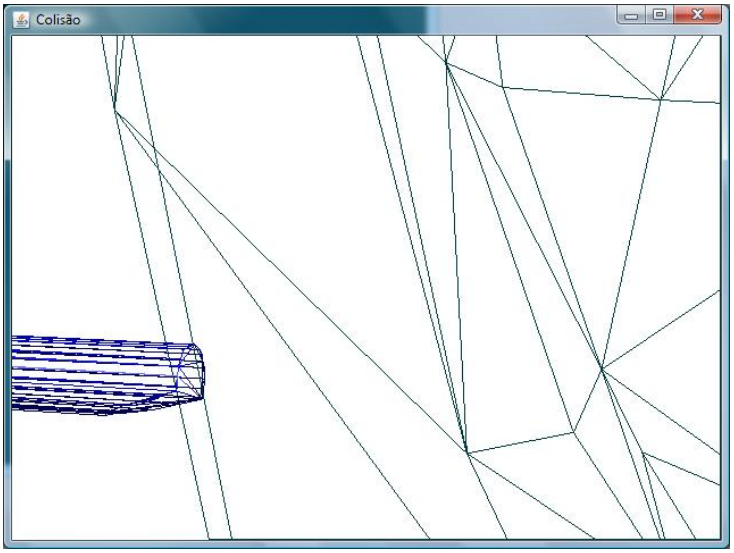

Figura 4.20: Detalhe da colisão com malhas poligonais visíveis.

4.3 Deformação dos Objetos

Nesta seção são mostrados os testes realizados no ambiente desenvolvido com a adição do método de deformação. O cálculo de deformação é feito apenas após a colisão, por esse motivo são mostrados apenas os resultados do cálculo da deformação uma vez que o desempenho da fase anterior a este cálculo não é alterado.

Os valores da constante da mola (K), constante de amortecimento (D), massa (M) e força

(F), definidos no capítulo 3, não foram alterados durante os testes. Os valores considerados nos experimentos foram $K = 0.3$, $M = 300.0$, $D = 0.7f$ e $F = (4.0f, 0.0f, 0.0f)$, sendo que F é considerada para os três eixos (X, Y e Z).

4.3.1 Testes com Objetos Simples

Os mesmos objetos simples da etapa de colisão, ou seja, o Diamante e a Rosca, foram utilizados nos testes de deformação.

Quando os objetos se colidem então os cálculos do método de deformação são efetuados, sendo que para a plataforma Windows e para Linux foram obtidas taxas médias de renderização de 34 e 32 FPS, respectivamente.

A figura 4.21 ilustra os objetos momentos antes da colisão e deformação. Essa visualização é importante para a comparação visual do objeto em seu estado inicial (estado atual) com a visualização do objeto após a deformação.

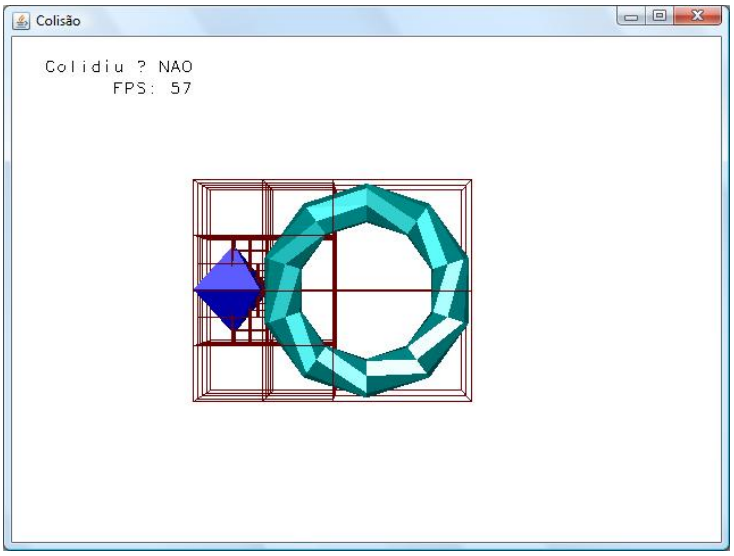

Figura 4.21: Objetos antes de se colidirem.

A figura 4.22 ilustra a deformação do objeto Rosca. É possivel notar que, como a quantidade de faces é pequena, quando uma face se move, a estrutura original do objeto já sofre uma grande transformação. Para uma deformação mais acentuada é necessário que as faces do objeto sejam de um tamanho consideravelmente menor do que o apresentado.

A figura 4.23 mostra os objetos momentos antes de se colidirem em outro ponto da Rosca. A taxa média obtida na plataforma Windows foi de 63 FPS, enquanto para a plataforma Linux foi de 61 FPS.

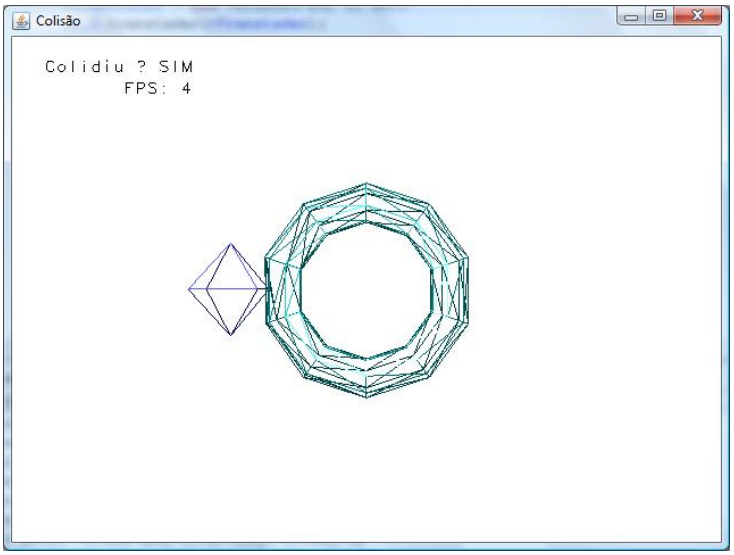

Figura 4.22: Deformação do objeto Rosca.

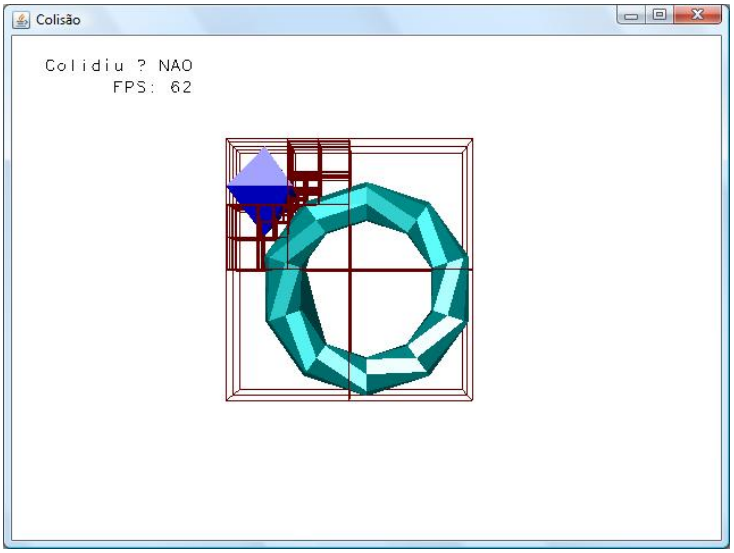

Figura 4.23: Objetos antes de se colidirem.

A figura 4.24 mostra a colisão entre os objetos em outro ponto, envolvendo mais vértices e faces. A taxa média de FPS reduz-se consideravelmente, ficando entre 5 a 10 FPS na plataforma Windows e entre 3 a 10 FPS na plataforma Linux.

4.3.2 Teste com Objetos Complexos

Nos testes com objetos complexos, utilizou-se novamente a agulha e a mama. Na cena foi utilizado o comando GL_FLAT do OpenGl para mudar a razão de aspecto dos objetos na cena, destacando-se melhor a deformação dos objetos. A figura 4.25 ilustra os objetos Seringa e Mama antes de se colidirem, tal que os objetos estão envolvidos na representação *octree*.

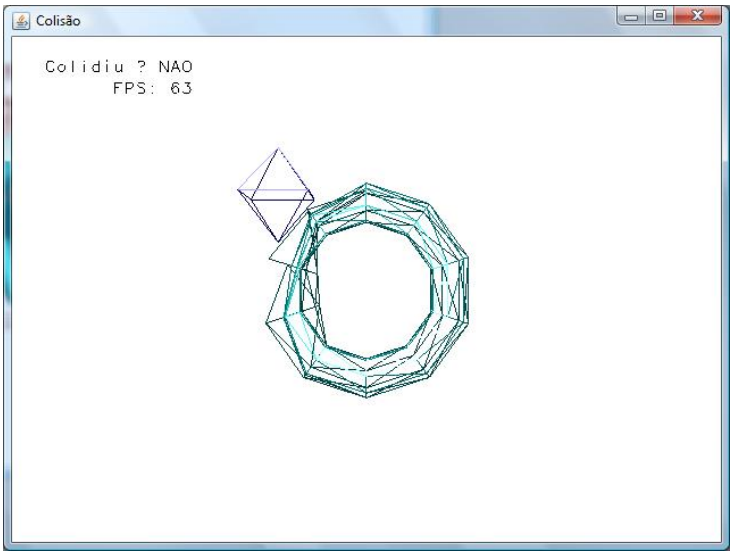

Figura 4.24: Deformação de mais faces do objeto Rosca.

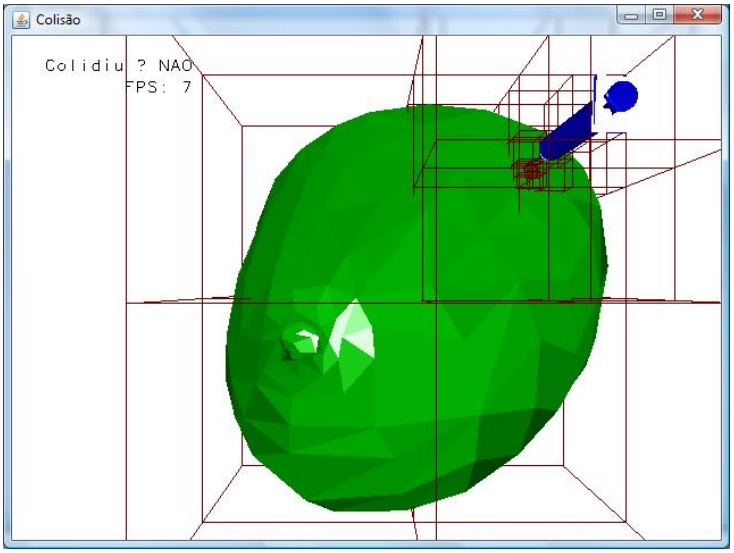

Figura 4.25: Objetos antes de se colidirem.

Para facilitar a visualização da cena, a figura 4.26 mostra a mesma imagem anterior mas com destaque para as malhas poligonais.

As figuras 4.27 e 4.28 mostram uma seqüencia de deformação das faces do objeto Mama após a colisão. A média de renderização obtida nas plataforma Windows e Linux foram de 17 FPS e 16 FPS, respectivamente.

4.4 Comentários Finais

A partir dos experimentos realizados, pode-se efetuar um comparação entre os resultados obtidos pelo protótipo para objetos simples e complexos nas plataformas Windows e Linux. Valores médios

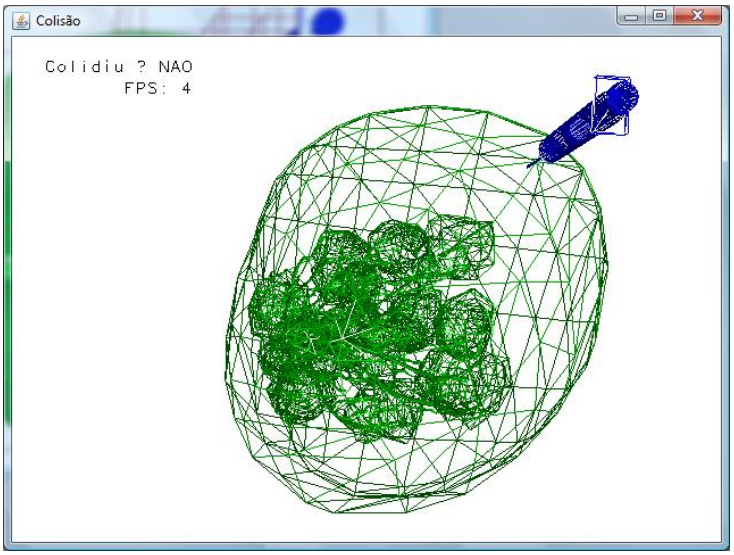

Figura 4.26: Objetos antes da colisão com malhas poligonais visíveis.

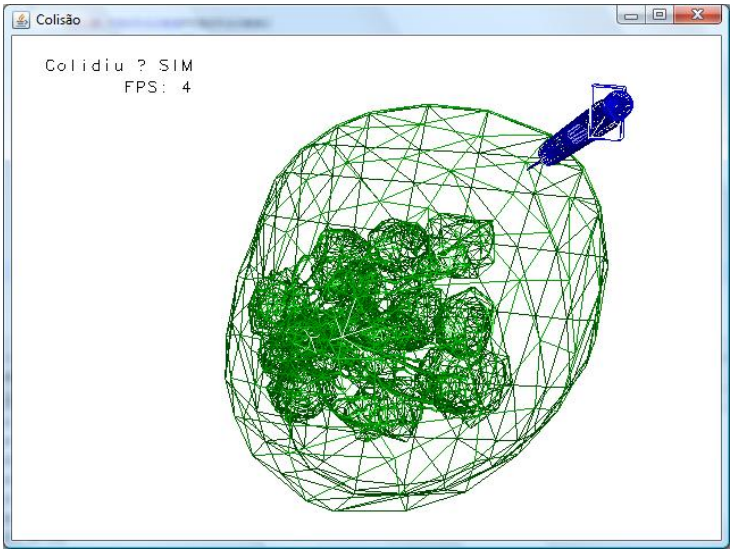

Figura 4.27: Deformação de uma face do objeto Mama.

para a taxa de renderização são mostrados na tabela 4.1.

	Colisão		**Deformação**	
Média FPS	Objetos Simples	Objetos Complexos	Objetos Simples	Objetos Complexos
Windows	62	24	34	17
Linux	60	23	32	16

Tabela 4.1: Comparação entre valores de FPS para objetos simples e complexos nas plataformas Windows e Linux.

Com base nesses dados é possível verificar que, com objetos simples, a aplicação tem um tempo de resposta adequado em termos de taxa de renderização. Como o sistema visual humano é capaz de perceber animações em tempo real a taxas próximas de 24 FPS, o ambiente demonstrou atingir

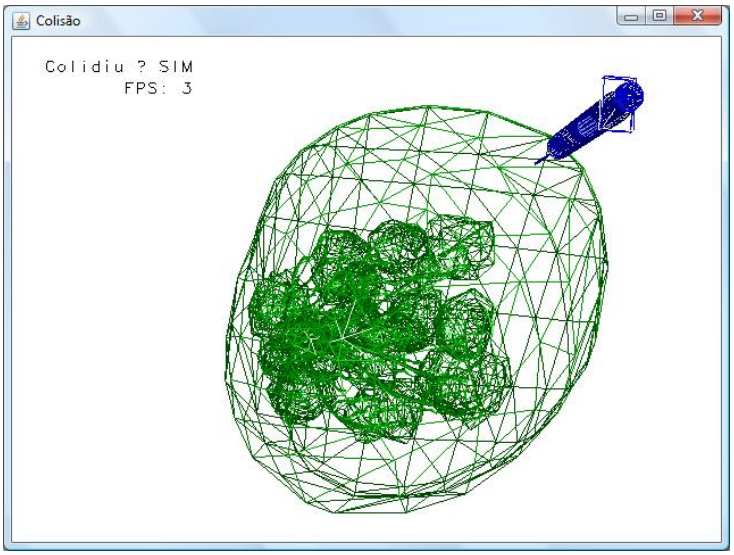

Figura 4.28: Deformação de uma face do objeto Mama.

taxas médias suficientemente altas para garantir interação em tempo real para os objetos simples com os métodos de colisão e deformação dos objetos.

Para objetos complexos, a taxa de renderização foi mais baixa, porém, ainda assim o ambiente permitiu que o usuário interagisse satisfatoriamente com os objetos (notar que movimentação dos objetos possui taxas médias em torno de 24 FPS).

Para a deformação foram coletadas as posições iniciais das faces envolvidas na colisão e a posição final das mesmas, mostrando assim o grau de deformação.

A figura 4.29 ilustra uma seqüência de deformação do objeto Mama. A figura 4.29(a) ilustra o objeto em sua forma inicial, a figura 4.29(b) mostra o momento da colisão entre os objetos Seringa e Mama e início da deformação, enquanto a figura 4.29(c) exibe a objeto após a deformação.

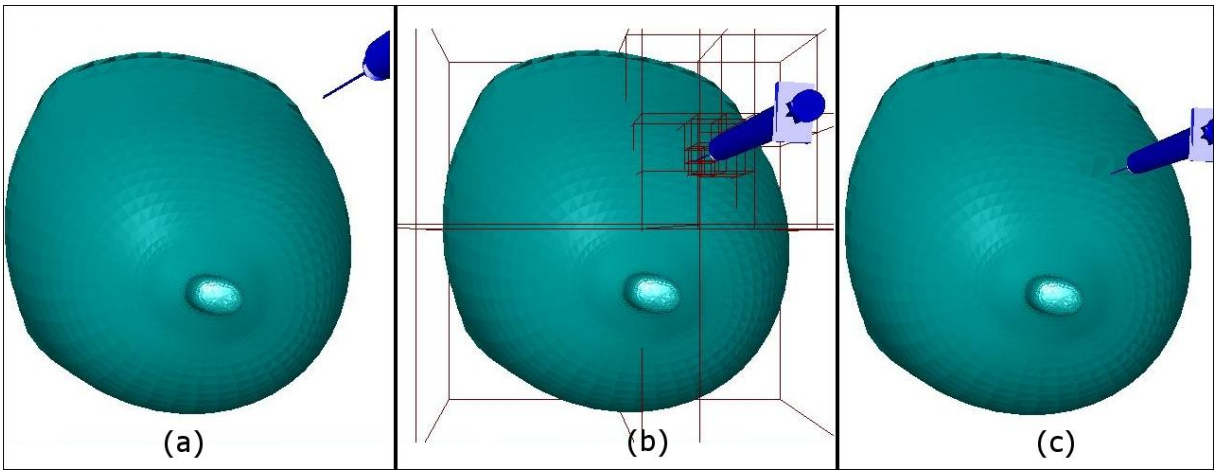

Figura 4.29: Seqüência de deformação do objeto Mama.

A tabela 4.2 exibe, na primeira coluna, o índice correspondente ao vértice do objeto Mama que foi modificado. As três próximas colunas mostram a posição (x, y, z) desse vértice em seu estado original e as três últimas colunas exibem a posição final (x, y, z) do vértice após a deformação.

Índice do Vértice	**Início - Sem deformação**			**Final - Com deformação**		
1033	2.790618	7.593750	8.822466	2.3826656	6.484224	7.376596
303	0.649441	7.817082	9.075597	0.5543608	6.674912	7.749357
1473	2.781990	9.013126	7.983518	2.3754556	7.696336	6.816769

Tabela 4.2: Resultado da deformação do objeto Mama.

Para validação dos métodos, outros objetos foram utilizados nos experimentos. A figura 4.30(a) ilustra o início de uma seqüencia de deformação do objeto Nádegas, mostrando o mesmo com seus vértices originais. A figura 4.30(b) mostra o momento da colisão entre os objetos. A figura 4.30(c) exibe o final da deformação ocorrida pelo contato entre os objetos Seringa e Nádegas. A tabela 4.3 mostra as posições iniciais e finais dos vértices deformados.

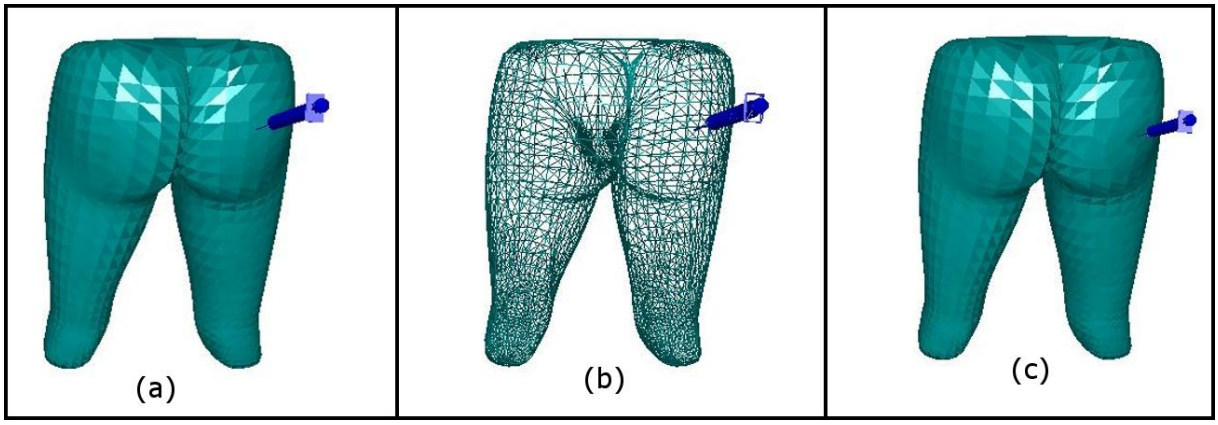

Figura 4.30: Seqüência de deformação do objeto Nádegas.

Índice do Vértice	**Início - Sem deformação**			**Final - Com deformação**		
846	5.399398	5.716644	8.164954	4.610506	4.881189	6.815141
195	6.668903	6.022062	7.506981	5.694274	5.142079	6.4099874
426	5.476997	4.670556	7.786890	4.676944	3.988847	6.6491313

Tabela 4.3: Resultado da deformação do objeto Nádegas.

A figura 4.31 ilustra uma seqüência de deformação do objeto Perna. A figura 4.31(a) apresenta o objeto com seus vértices originais, a figura 4.31(b) o momento da colisão entre os objetos Seringa e Perna e, finalmente, a figura 4.31 mostra o objeto Perna após a deformação. A tabela 4.4 mostra as posições iniciais e finais dos vértices deformados.

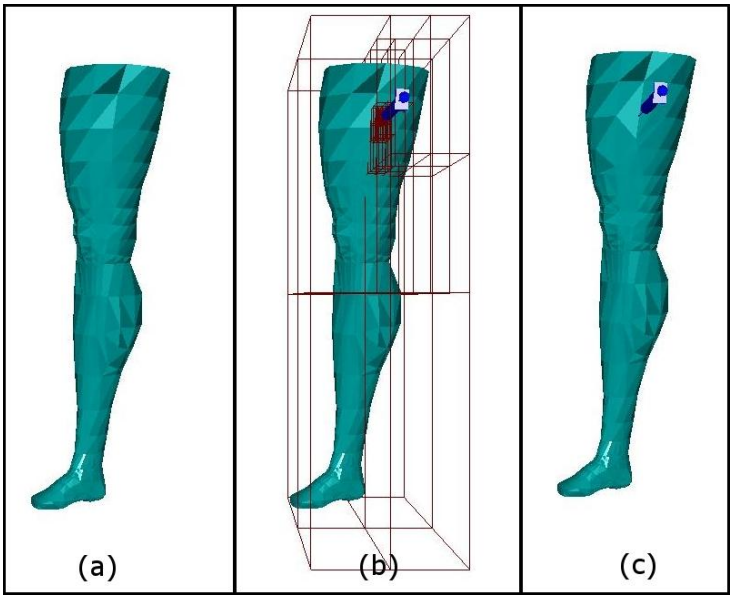

Figura 4.31: Seqüência de deformação do objeto Perna.

Índice do Vértice	**Início - Sem deformação**			**Final - Com deformação**		
389	-0.3521803	11.213284	11.378992	-0.3007779	9.573436	9.559213
375	-1.7953256	9.217928	10.524809	-1.5335817	7.872452	8.987516
362	-1.5345905	7.555994	9.426216	-1.5238687	7.503209	9.360363

Tabela 4.4: Resultado da deformação do objeto Perna.

CAPÍTULO 5

CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou o desenvolvimento de um protótipo de ambiente virtual interativo para treinamento médico. Métodos de detecção de colisão e de deformação dos objetos foram incorporados ao ambiente.

Devido à futura integração dos métodos no *framework* ViMet, uma estrutura de classes foi projetada, tornando mais simples também a inclusão de novas funcionalidades ao ambiente. O desenvolvimento utilizou pacotes livres, abertos e multiplataforma. Esses objetivos foram satisfatoriamente alcançados a partir da utilização da linguagem de programação Java.

A subdivisão hierárquica do espaço baseada em *octrees* foi utilizada na implementação do método de detecção de colisão dos objetos. O método se mostrou preciso e com adequadas taxas de renderização, permitindo uma simulação com realismo.

O uso da deformação massa-mola permitiu a simulação de alteração na forma dos objetos que se colidem com o uso de malhas poligonais, o que proporciou maior realismo à cena contida no ambiente.

Embora o trabalho possa ser melhorado em vários aspectos, conforme propostas de trabalhos futuros citadas a seguir, os experimentos demonstraram que a abordagem atingiu os principais objetivos propostos.

Trabalhos Futuros

Algumas diretrizes de pesquisa podem ser sugeridas a partir dos resultados obtidos neste trabalho. Com a integração das classes e métodos desenvolvidos no *framework* ViMet, o trabalho poderá trazer ainda mais benefícios a aplicações de treinamento médico. Além da simulação dos exames de punção mamária, outros tipos de procedimento médico podem ser simulados.

Embora os quesitos de precisão e desempenho tenham sido adequadamente alcançados, novos testes devem ser realizados para otimizar os métodos por meio da modificação dos parâmetros utilizados. Em particular, sugere-se um estudo mais detalhado dos parâmetros da deformação massa-

mola, buscando a obtenção de maior realismo em casos de tecidos de órgãos humanos.

Uma comparação com a recente biblioteca Cybermed [36] para desenvolvimento de aplicações médicas é planejada como atividade futura. Pretende-se também realizar testes em conjunto com profissionais da área de saúde para auxiliar na validação das técnicas e melhorar o realismo da simulação.

Finalmente, outra proposta de trabalho futuro é a incorporação de equipamentos não convencionais, melhorando a interação entre o usuário e o ambiente.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Ananthuni, R., Karki, B. B., Bollig, E. F., da Silva, C. R. S. e Eelebacher, G. A Web-Based Visualization and Reposition Scheme for Scientific Data. In *Proceedings of the 2006 International Conference on Modeling Simulation and Visualization Methods* (Las Vegas, NV, Estados Unidos, Junho 2006), pp. 311–317.

[2] Antonio, F. *Faster Line Segment Intersection*. Academic Press Professional, Inc., San Diego, CA, Estados Unidos, 1992, pp. 199–202.

[3] Aukstakalnis, S. e Blatner, D. *The Art and Science of Virtual Reality*. Peatchpit Press, Berkeley, CA, Estados Unidos, 1992.

[4] AutoDesk. Autodesk 3ds Max, 2008. `http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=5659302`.

[5] Backes, P. G., Norris, J. S., Powell, M. W. e Vona, M. A. Multi-mission Activity Planning for Mars Lander and Rover Missions. *IEEE Aerospace Conference 2* (2004), 877–886.

[6] Basdogan, C. e Ho, C.-H. Principles of Haptic Rendering for Virtual Environments, 2008. `http://network.ku.edu.tr/~cbasdogan/Tutorials/haptic_tutorial.html`.

[7] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[8] Bentley, J. L. K-d trees for semidynamic point sets. In *SCG '90: Proceedings of the sixth annual symposium on Computational geometry* (New York, NY, Estados Unidos, 1990), ACM Press, pp. 187–197.

[9] Bradshaw, G. e O'Sullivan, C. Sphere-tree Construction using Dynamic Medial Axis Approximation. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer animation* (New York, NY, Estados Unidos, 2002), pp. 33–40.

[10] Brandão, E. J. R., Trentin, M. A. S., Lebedeff, T. B., Mortari, M. I. M., Oro, N. T. e Pasqualotti, A. A Realidade Virtual como Proposta de Ensino-Aprendizagem de Matemática para Deficientes Auditivos. In *IV Congresso RIBIE* (Brasília-DF, 1998).

[11] Campbell, C. Java2D/JOGL Interoperability, 2008. http://weblogs.java.net/blog/campbell/archive/2005/09/java2djogl_inte_1.html.

[12] Campos, S. F. e dos Santos Machado, L. Métodos de Deformação em Sistemas Interativos. *Science of Computing 1*, 2 (Janeiro 2008), 15–32.

[13] Choi, K. S., Sun, H., Heng, P. A. e Cheng, J. C. Y. A Scalable Force Propagation Approach for Web-based Deformable Simulation of Soft Tissues. In *Web3D '02: Proceedings of the seventh international conference on 3D Web technology* (Tempe, AZ, Estados Unidos, 2002), pp. 185–193.

[14] Costa, I. F. e Balaniuk, R. Lem-an approach for real time physically based soft tissue simulation. *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on 3* (2001), 2337–2343 vol.3.

[15] Davidson, A. *Killer Game Programming in Java*. O'Reilly Media Inc., Maio 2005.

[16] Day, B. Advantages and disadvantages to Java-to-OpenGL bindings vs. the Java 3D API implementation from Sun, Outubro 2008. http://www.javaworld.com/javaworld/jw-05-1999/jw-05-media.html.

[17] de Berg, M., van Kreveld, M., Overmars, M. e Schwarzkopf, O. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, New York, NY, Estados Unidos, 2000.

[18] Eberly, D. H. *3D Game Engine Design, Second Edition: A Practical Approach to Real-Time Computer Graphics (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, Estados Unidos, 2006.

[19] Fiolhais, M. Física Geral I - Elementos de Física (Notas de Aula), 2008. http://nautilus.fis.uc.pt/personal/mfiolhais/FGbio/FGbio_0607.htm.

[20] Gibson, S. F. F. e Mirtich, B. A Survey of Deformable Modeling in Computer Graphics. TR-97, Mitsubishi Electric Research Laboratories, Cambridge, MA, Estados Unidos, 1997.

[21] Giesecke, S. Seminar Algorithmen für Computerspiele: Räumliche Sortierung, 2008. http://www.vis.uni-stuttgart.de/eng/teaching/lecture/ws04/seminar_spiele/bsp/.

[22] Gottschalk, S., Lin, M. C. e Manocha, D. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics And Interactive Techniques* (New York, NY, Estados Unidos, 1996), ACM Press, pp. 171–180.

[23] Guibas, L. J., Hsu, D. e Zhang, L. H-Walk: Hierarchical Distance Computation for Moving Convex Bodies. In *SCG '99: Proceedings of the Fifteenth Annual Symposium on Computational Geometry* (Miami Beach, FL, Estados Unidos, 1999), pp. 265–273.

[24] Hadap, S., Eberle, D., Volino, P., Lin, M. C., Redon, S. e Ericson, C. Collision Detection and Proximity Queries. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes* (Los Angeles, CA, Estados Unidos, 2004), p. 15.

[25] Hancock, D. Viewpoint: Virtual Reality in Search of Middle Ground. *IEEE Spectrum 32*, 1 (Janeiro 1995), 68.

[26] Hearn, D. e Baker, M. P. *Computer Graphics, C Version*. Prentice Hall, 1996.

[27] Herzen, B. V., Barr, A. H. e Zatz, H. R. Geometric Collisions for Time-dependent Parametric Surfaces. In *SIGGRAPH '90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (Dallas, TX, Estados Unidos, 1990), pp. 39–48.

[28] Hirota, G., Maheshwari, R. e Lin, M. C. Fast Volume-Preserving Free Form Deformation using Multi-level Optimization. In *SMA '99: Proceedings of the Fifth ACM Symposium on Solid Modeling and Applications* (Ann Arbor, MI, Estados Unidos, 1999), pp. 234–245.

[29] Hubbard, P. M. Collision Detection for Interactive Graphics Applications. *IEEE Transactions on Visualization and Computer Graphics 1*, 3 (1995), 218–230.

[30] Hughes, M., DiMattia, C., Lin, M. C. e Manocha, D. Efficient And Accurate Interference Detection For Polynomial Deformation. In *CA '96: Proceedings of the Computer Animation* (Washington, DC, Estados Unidos, 1996), IEEE Computer Society, p. 155.

[31] JOGL. Java OpenGL, Outubro 2006. `http://en.wikipedia.org/wiki/Java_OpenGL`.

[32] jogl.dev.java.net. JOGL - User's Guide, Outubro 2008. `https://jogl.dev.java.net/nonav/source/browse/*checkout*/jogl/doc/userguide/index.html?rev=HEAD&contenttype=text/html`.

[33] Kera, M., Bruno Filho, J. W. e Pedrini, H. Análise Comparativa entre Ferramentas para Aplicações em Realidade Virtual. *II Workshop de Aplicações de Realidade Virtual* (2006), 13–16.

[34] Kirner, C. e Pinho, M. Uma Introdução à Realidade Virtual. In *SIBGRAPI - Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens* (Campos do Jordão, SP, Outubro 1996). `http://grv.inf.pucrs.br/Pagina/TutRV/tutrv.htm`.

[35] Klosowski, J. T., Held, M., Mitchell, J. S., Sowizral, H. e Zikan, K. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (1998), 21–36.

[36] LabTEVE. Laboratório de Tecnologias para o Ensino Virtual e Estatística - CyberMed, 2008. `http://www.de.ufpb.br/~labteve/projetos/cybermed_.html`.

[37] Latta, J. N. e Oberg, D. J. A Conceptual Virtual Reality Model. *IEEE Computer Graphics and Applications 14*, 1 (1994), 23–29.

[38] Lau, R. W., Chan, O., Luk, M. e Li, F. W. LARGE: A Collision Detection Framework for Deformable Objects. In *VRST '02: Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (Hong Kong, China, 2002), pp. 113–120.

[39] Le, K. The Red Book Examples Using JOGL, Outubro 2008. `http://ak.kiet.le.googlepages.com/theredbookinjava.html`.

[40] Lin, M. C. e Canny, J. F. Efficient Algorithms dor incremental Distance Computation. *IEEE Conference on Robotics and Automation 2* (1991), 1008–1014.

[41] Machado, L. e Zuffo, M. Development and Evaluation of a Simulator of Invasive Procedures in Pediatric Bone Marrow Transplant. *Studies In Health Technology And Informatics 94* (2003), 193–195.

[42] Machado, L., Zuffo, M., Moraes, M. e Lopes, R. Modelagem Tátil, Visualização Estereoscópica e Aspectos de Avaliação em um Simulador de Coleta de Medula Óssea. In *IV Simpósio de Realidade Virtual* (Florianópolis-SC, Outubro 2001), pp. 23–31.

[43] Marner, J. Evaluating Java for Game Development. Dissertação de Mestrado, Universidade de Copenhagen, Dinamarca, Março 2002.

[44] Microsoft Corporation. DirectX, Outubro 2008. http://www.microsoft.com/Windows/directx/productinfo/overview/default.mspx.

[45] Moore, P. e Molloy, D. A Survey of Computer-Based Deformable Models. In *IMVIP 2007: International Machine Vision and Image Processing Conference* (Setembro 2007), pp. 55–66.

[46] Morie, J. F. Inspiring the Future: Merging Mass Communication, Art, Entertainment and Virtual Environments. *ACM SIGGRAPH Computer Graphics 28*, 2 (1994), 135–138.

[47] Oliveira, A. C. M. T. G. ViMeT - Projeto e Implementação de um Framework para Aplicações de Treinamento Médico usando Realidade Virtual. Dissertação de Mestrado, Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, SP, 2007.

[48] O'Reilly. Online Mirror of the Encyclopedia of Graphics File Formats. http://www.fileformat.info/format/wavefrontobj/.

[49] O'Reilly. Online Mirror of the Encyclopedia of Graphics File Formats. http://www.fileformat.info/format/3ds.

[50] Poliakov, A. V., Albright, E., Hinshaw, K. P., Corina, D. P., Ojemann, G., Martin, R. F. e Brinkley, J. F. Server-based Approach to Web Visualization of Integrated Three-dimensional Brain Imaging Data. *American Medical Informatics Association 12*, 2 (2005), 140–151.

[51] Ponamgi, M. K., Manocha, D. e Lin, M. C. Incremental Algorithms for Collision Detection Between Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics 3*, 1 (1997), 51–64.

[52] Preparata, F. P. e Shamos, M. I. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., New York, NY, Estados Unidos, 1985.

[53] Provot, X. Deformation Constraints in a Mass-spring Model to Describe Rigid Cloth Behavior. In *Proceedings of Graphics Interface* (Quebec, Canadá, 1995), pp. 147–154.

[54] Ritter, J. An Efficient Bounding Sphere. In *Graphics Gems*. Academic Press Professional, Inc., San Diego, CA, Estados Unidos, 1990, pp. 301–303.

[55] Russel, K. Java2D/JOGL Interoperability Demo, 2008. http://www.javagaming.org/forums/index.php?topic=10813.0.

[56] Sederberg, T. W. e Parry, S. R. Free-form deformation of solid geometric models. *Computer Graphics 20*, 4 (Agosto 1986), 151–160.

[57] Sun Microsystems. Tutorials and Code Camps, Java 3D API Tutorial, Outubro 2006. http://developer.java.sun.com/developer/onlineTraining/java3d.

[58] Sun Microsystems. The Java 3DTM API Specification Version 1.3, Maio 2007. http://java.sun.com/products/javamedia/3D/forDevelopers/J3D_1_3_API/j3dguide.

[59] Thibault, W. C. e Naylor, B. F. Set Operations on Polyhedra using Binary Space Partitioning Trees. *ACM SIGGRAPH Computer Graphics 21*, 4 (1987), 153–162.

[60] Valerio Netto, A., dos Santos Machado, L. e Oliveira, M. C. F. Realidade Virtual - Definições, Dispositivos e Aplicações. *Revista Eletrônica de Iniciação Cientifica 2*, 1 (2002).

[61] van den Bergen, G. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools 2*, 4 (1997), 1–13.

[62] Volino, P. e Thalmann, N. M. Efficient Self-collision Detection on Smoothly Discretized Surface Animations using Geometrical Shape Regularity. *Computer Graphics Forum 13*, 3 (1994), 155–166.

[63] von Schweber, L. e von Schweber, E. Cover Story: Realidade Virtual. *PC Magazine Brasil 5*, 6 (1995), 50–73.

[64] Yang, Y. e Thalmann, N. An Improved Algorithm for Colision Detection in Cloth Animation with Human Body. *Proceedings of Pacific Graphics Conference* (1993), 237–251.

APÊNDICE A

FORMATO DOS ARQUIVOS

Este apêndice descreve brevemente os dois formatos para representação poligonal dos objetos que foram utilizados no ambiente.

A.1 Formato OBJ

O formato *Wavefront OBJ* [48] é um padrão para representação de dados poligonais na forma ASCII. A organização do formato é descrita a seguir:

- Vértices:
  - v: vértices geométricos
  - vt: vértices de texturas
  - vn: vértices normais
  - vp: vértices de espaço paramétricos
- Atributos de curvas e superfícies:
  - deg: Graus
  - bmat: Matriz base
  - step: Tamanho do passo
  - cstype: Tipo de curva ou superfície
- Elementos:
  - p: Ponto
  - l: Linha
  - f: Face
  - curv: Curva
  - curv2: Curva 2D
  - surf: Superfície
- Indicações de curvas e superfícies:
  - parm: Valores de parâmetros
  - trim: Laço exterior de corte
  - hole: Laço interior de corte
  - scrv: Curva especial
  - sp: Ponto especial
  - end: Indicação de fim
- Conectividade entre superfícies:
  - con: Conexão

- Agrupamento:
  - g: Nome do grupo
  - s: Grupo de Alisamento (Smoothing group)
  - mg: Grupo de fundir (Merging group)
  - o: Nome do objeto
- Atributos de Renderização e exposição (Display):
  - bevel: Bevel interpolation
  - c_interp: Interpolação de cor
  - d_interp: Dissolve a interpolação
  - lod: Nível de detalhamento
  - usemtl: Nome do material
  - mtllib: Biblioteca do material
  - shadow_obj: Sombra do objeto
  - trace_obj: Ray tracing
  - ctech: Técnica de aproximação de curva
  - stech: Técnica de aproximação de superfície

A.2 Formato 3ds

O formato 3ds [49] é um arquivo binário e proprietário da Autodesk [4]. Como a organização desse arquivo é muito extensa, apenas as partes utilizadas no ambiente desenvolvido são descritas:

ID Name Data Type Description

- Identificador: 4d4d
  - Nome: M3DMAGIC
  - Tipo de dados: ND
  - Descrição: Número mágico do arquivo 3ds.
- Identificador: 3d3d
  - Nome: MDATA
  - Tipo de dados: WORD
  - Descrição: Conector de sub arquivos 3ds.
- Identificador: 4000
  - Nome: NAMED_OBJECT
  - Tipo de dados: CHAR[]
  - Descrição: nome do objeto em ASCII
- Identificador: 4110
  - Nome: POINT_ARRAY
  - Tipo de dados: SHORT
  - Descrição: Número de pontos do objeto

- Identificador: 4120
  Nome: FACE_ARRAY
  Tipo de dados: SHORT
  Descrição: Número de faces do objeto
- Identificador: 4140
  Nome: TEX_VERTS
  Tipo de dados: SHORT
  Descrição: Número de vértices do objeto

Dependência dos identificadores:

- O identificador 4110 (POINT_ARRAY) tem a seguinte definição de ponto:

```
typedef struct _POINT {
   FLOAT x, y, z;
} POINT;
```

- O identificador 4120 (FACE_ARRAY) tem a seguinte definição para face:

```
typedef struct _FACE {
   SHORT vertice1, vertice2, vertice3, flags;
} FACE;
```

- O identificador 4140 (TEX_VERTS) tem a seguinte definição para os vértices:

```
typedef struct _VERTEX {
   FLOAT x,y;
} VERTEX;
```

APÊNDICE B

CLASSES E MÉTODOS

Este apêndice apresenta os códigos fontes utilizados no protótipo referentes às classes e métodos para detecção de colisão e deformação de objetos.

Classe Geometria

Esta classe é responsável pelos cálculos de detecção de colisão dos objetos no ambiente.

```
public class Geometria {
	public static final float PRECISAO = 0.0001f;
	public static final float ERRO = 0.000001f;

	Vetor3f v0, v1, v2, normal, e0, e1, e2, metadeCaixa;
	Vetor3f E1, E2, N1, N2, V0, V1, V2, U0, U1, U2;
	float min, max, p0, p1, p2, rad, Ax, Ay, Bx, By, Cx, Cy, e, d, f;
	short i0, i1;

	// atribui às variáveis min e max o menor e maior valor, respectivamente,
	// das variáveis x1, x2 e x3
	public void MINMAX(float x0,float x1,float x2,float min,float max) {
		min = max = x0;
		if (x1<min)
			min=x1;
		if (x1>max)
			max=x1;
		if (x2<min)
			min=x2;
		if (x2>max)
			max=x2;

		this.min = min;
		this.max = max;
	}

	// testes no eixo X
	public boolean TESTE_EIXO_X01(float a, float b, float fa, float fb) {
		p0 = a*v0.y - b*v0.z;
		p2 = a*v2.y - b*v2.z;
		if (p0<p2) {
			min=p0;
			max=p2;
		}
		else {
			min=p2;
			max=p0;
		}
		rad = PRECISAO + fa * metadeCaixa.y + fb * metadeCaixa.z;
		if (min>rad || max<-rad)
			return false;
		return true;
	}

	public boolean TESTE_EIXO_X2(float a, float b, float fa, float fb) {
		p0 = a*v0.y - b*v0.z;
		p1 = a*v1.y - b*v1.z;
		if (p0<p1) {
			min=p0;
			max=p1;
		}
		else {
			min=p1;
			max=p0;
		}
		rad = PRECISAO + fa * metadeCaixa.y + fb * metadeCaixa.z;
		if (min>rad || max<-rad)
			return false;
		return true;
	}
```

```
// testes no eixo Y
public boolean  TESTE_EIXO_Y02(float a, float b, float fa, float fb) {
        p0 = -a*v0.x + b*v0.z;
        p2 = -a*v2.x + b*v2.z;
        if (p0<p2) {
                min=p0;
                max=p2;
        }
        else {
                min=p2;
                max=p0;
        }
        rad = PRECISAO + fa * metadeCaixa.x + fb * metadeCaixa.z;
        if (min>rad || max<-rad)
                return false;
        return true;
}

public boolean TESTE_EIXO_Y1(float a, float b, float fa, float fb) {
        p0 = -a*v0.x + b*v0.z;
        p1 = -a*v1.x + b*v1.z;
        if (p0<p1) {
                min=p0;
                max=p1;
        }
        else {
                min=p1;
                max=p0;
        }
        rad = PRECISAO + fa * metadeCaixa.x + fb * metadeCaixa.z;
        if (min>rad || max<-rad)
                return false;
        return true;
}

// testes no eixo Z
public boolean  TESTE_EIXO_Z12(float a, float b, float fa, float fb) {
        p1 = a*v1.x - b*v1.y;
        p2 = a*v2.x - b*v2.y;
        if (p2<p1) {
                min=p2;
                max=p1;
        }
        else {
                min=p1;
                max=p2;
        }
        rad = PRECISAO + fa * metadeCaixa.x + fb * metadeCaixa.y;
        if (min>rad || max<-rad)
                return false;
        return true;
        }

        public boolean  TESTE_EIXO_Z0(float a, float b, float fa, float fb) {
        p0 = a*v0.x - b*v0.y;
        p1 = a*v1.x - b*v1.y;
        if (p0<p1) {
                min=p0;
                max=p1;
                }
        else {
                min=p1;
                max=p0;
                }
        rad = PRECISAO + fa * metadeCaixa.x + fb * metadeCaixa.y;
        if (min>rad || max<-rad)
                return false;
        return true;
        }

// ordena tal que a <= b
public void  ORDENAR(float a,float b) {
        if (a>b) {
                float c;
                c=a;
                a=b;
                b=c;
        }
}

// teste aresta-aresta baseado no algoritmo de Franlin Antonio
// "Faster Line Segment Intersection", in Graphics Gems III, pp. 199-202
public boolean  EDGE_EDGE_TEST(Vetor3f V0,Vetor3f U0,Vetor3f U1) {
        Bx=U0.operatorArray(i0)-U1.operatorArray(i0);
        By=U0.operatorArray(i1)-U1.operatorArray(i1);
        Cx=V0.operatorArray(i0)-U0.operatorArray(i0);
        Cy=V0.operatorArray(i1)-U0.operatorArray(i1);
        f=Ay*Bx-Ax*By;
        d=By*Cx-Bx*Cy;
        if ((f>0 && d>=0 && d<=f) || (f<0 && d<=0 && d>=f)) {
```

```
                e=Ax*Cy-Ay*Cx;
                if (f>0) {
                        if (e>=0 && e<=f)
                                return true;
                }
                else {
                        if (e<=0 && e>=f)
                                return true;
                }
        }
        return false;
}

// testa aresta U0,U1 contra V0,V1
// testa aresta U1,U2 contra V0,V1
// testa aresta U2,U1 contra V0,V1
public void TESTA_ARESTAS(Vetor3f V0,Vetor3f V1,Vetor3f U0,Vetor3f U1,Vetor3f U2) {
        Ax=V1.operatorArray(i0)-V0.operatorArray(i0);
        Ay=V1.operatorArray(i1)-V0.operatorArray(i1);
        EDGE_EDGE_TEST(V0,U0,U1);
        EDGE_EDGE_TEST(V0,U1,U2);
        EDGE_EDGE_TEST(V0,U2,U0);
}

// testa se T1 está completamente dentro de T2
// verifica se V0 está dentro de tri(U0,U1,U2)
public boolean PONTO_NO_TRIANGULO(Vetor3f V0,Vetor3f U0,Vetor3f U1,Vetor3f U2) {
        float a,b,c,d0,d1,d2;
        a=U1.operatorArray(i1)-U0.operatorArray(i1);
        b=-(U1.operatorArray(i0)-U0.operatorArray(i0));
        c=-a*U0.operatorArray(i0)-b*U0.operatorArray(i1);
        d0=a*V0.operatorArray(i0)+b*V0.operatorArray(i1)+c;

        a=U2.operatorArray(i1)-U1.operatorArray(i1);
        b=-(U2.operatorArray(i0)-U1.operatorArray(i0));
        c=-a*U1.operatorArray(i0)-b*U1.operatorArray(i1);
        d1=a*V0.operatorArray(i0)+b*V0.operatorArray(i1)+c;

        a=U0.operatorArray(i1)-U2.operatorArray(i1);
        b=-(U0.operatorArray(i0)-U2.operatorArray(i0));
        c=-a*U2.operatorArray(i0)-b*U2.operatorArray(i1);
        d2=a*V0.operatorArray(i0)+b*V0.operatorArray(i1)+c;

        if (d0*d1>0.0) {
                if (d0*d2>0.0)
                        return true;
        }
        return false;
}

public boolean CALCULAR_INTERVALOS(float VV0,float VV1,float VV2,float D0,float D1,float D2,
                        float D0D1,float D0D2,float A,float B,float C,float X0,float X1) {
        if (D0D1>0.0f) {
                // here we know that D0D2<=0.0
                // that is D0, D1 are on the same side, D2 on the other or on the plane
                A=VV2; B=(VV0-VV2)*D2; C=(VV1-VV2)*D2; X0=D2-D0; X1=D2-D1;
        }
        else if (D0D2>0.0f){
                // here we know that d0d1<=0.0
                A=VV1; B=(VV0-VV1)*D1; C=(VV2-VV1)*D1; X0=D1-D0; X1=D1-D2;
        }
        else if (D1*D2>0.0f || D0!=0.0f){
                // here we know that d0d1<=0.0 or that D0!=0.0
                A=VV0; B=(VV1-VV0)*D0; C=(VV2-VV0)*D0; X0=D0-D1; X1=D0-D2;
        }
        else if (D1!=0.0f) {
                A=VV1; B=(VV0-VV1)*D1; C=(VV2-VV1)*D1; X0=D1-D0; X1=D1-D2;
        }
        else if (D2!=0.0f) {
                A=VV2; B=(VV0-VV2)*D2; C=(VV1-VV2)*D2; X0=D2-D0; X1=D2-D1;
        }
        else {
                // triangles are coplanar
                return coplanar_tri_tri(N1,V0,V1,V2,U0,U1,U2);
        }
        return false;
}

public boolean ColisaoTrianguloCubo(Vetor3f[] cubo, Vetor3f v0, Vetor3f v1, Vetor3f v2) {
        // Precisao de 6 casas decimais
        return InterseccaoTrianguloCubo(CentroCubo(cubo), MeioCubo(cubo), v0, v1, v2);
}

public boolean ColisaoTrianguloTriangulo(Vetor3f V0, Vetor3f V1, Vetor3f V2,Vetor3f U0,
                                         Vetor3f U1, Vetor3f U2) {
        this.V0 = V0;
        this.V1 = V1;
        this.V2 = V2;
        this.U0 = U0;
        this.U1 = U1;
        this.U2 = U2;
```

```
N1 = new Vetor3f();
N2 = new Vetor3f();

float du0, du1, du2, dv0, dv1, dv2, d1, d2;
Vetor3f D;
Vetor2f isect1, isect2;
float du0du1, du0du2, dv0dv1, dv0dv2;
short indice;
float vp0, vp1, vp2;
float up0, up1, up2;
float bb, cc, max;

D = new Vetor3f();
isect1 = new Vetor2f();
isect2  = new Vetor2f();

E1 = V1.operatorSub(V0);
E2 = V2.operatorSub(V0);
N1.Produto(E1, E2);
d1 = -(N1.operatorMult(V0));

du0 = (N1.operatorMult(U0)) + d1;
du1 = (N1.operatorMult(U1)) + d1;
du2 = (N1.operatorMult(U2)) + d1;

if (Math.abs(du0) < ERRO)
du0 = 0.0f;
if (Math.abs(du1) < ERRO)
du1 = 0.0f;
if (Math.abs(du2) < ERRO)
        du2 = 0.0f;

        du0du1 = du0 * du1;
        du0du2 = du0 * du2;

        if (du0du1 > 0.0f && du0du2 > 0.0f)
                return false;

        E1 = U1.operatorSub(U0);
        E2 = U2.operatorSub(U0);
        N2.Produto(E1, E2);
        d2 = -(N2.operatorMult(U0));

        dv0 = (N2.operatorMult(V0)) + d2;
        dv1 = (N2.operatorMult(V1)) + d2;
        dv2 = (N2.operatorMult(V2)) + d2;

        if (Math.abs(dv0) < ERRO)
                dv0 = 0.0f;
        if (Math.abs(dv1) < ERRO)
                dv1 = 0.0f;
        if (Math.abs(dv2) < ERRO)
                dv2 = 0.0f;

        dv0dv1 = dv0 * dv1;
        dv0dv2 = dv0 * dv2;

        if (dv0dv1 > 0.0f && dv0dv2 > 0.0f)
                return false;

        D.Produto(N1, N2);

        max = Math.abs(D.operatorArray((short)0));
        indice = 0;
        bb = Math.abs(D.operatorArray((short) 1));
        cc = Math.abs(D.operatorArray((short)2));

        if (bb > max) {
                max = bb;
                indice = 1;
        }
        if (cc > max) {
                max = cc;
                indice = 2;
        }

        vp0 = V0.operatorArray(indice);
        vp1 = V1.operatorArray(indice);
        vp2 = V2.operatorArray(indice);

        up0 = U0.operatorArray(indice);
        up1 = U1.operatorArray(indice);
        up2 = U2.operatorArray(indice);

        float a, b, c, x0, x1;
        a = 0;
        b = 0;
        c = 0;
        x0 = 0;
        x1 = 0;
```

```
                CALCULAR_INTERVALOS(vp0, vp1, vp2,dv0, dv1, dv2,dv0dv1, dv0dv2,a, b, c, x0, x1);

                float d, e, f, y0, y1;
                d = 0;
                e = 0;
                f = 0;
                y0 = 0;
                y1 = 0;
                CALCULAR_INTERVALOS(up0, up1, up2,du0, du1, du2,du0du1, du0du2,d, e, f, y0, y1);

                float xx, yy, xxyy, tmp;
                xx = x0 * x1;
                yy = y0 * y1;
                xxyy = xx * yy;

                tmp = a * xxyy;
                isect1.x = tmp + b * x1 * yy;
                isect1.y = tmp + c * x0 * yy;

                tmp = d * xxyy;
                isect2.x = tmp + e * xx * y1;
                isect2.y = tmp + f * xx * y0;

                ORDENAR(isect1.x, isect1.y);
                ORDENAR(isect2.x, isect2.y);

                if (isect1.y < isect2.x || isect2.y < isect1.x)
                        return false;

                return true;
}


public Vetor3f CentroCubo(Vetor3f[] cubo) {
        Vetor3f centro = new Vetor3f();

        centro.x = (cubo[3].x + cubo[0].x) / 2;
        centro.y = (cubo[4].y + cubo[0].y) / 2;
        centro.z = (cubo[0].z + cubo[1].z) / 2;

        return centro;
}


public Vetor3f MeioCubo(Vetor3f[] cubo) {
        // não é um ponto, mas é um vetor de 3 floats
        Vetor3f metade = new Vetor3f();

        metade.x = Math.abs(cubo[3].x - cubo[0].x) / 2;
        metade.y = Math.abs(cubo[4].y - cubo[0].y) / 2;
        metade.z = Math.abs(cubo[0].z - cubo[1].z) / 2;

        return metade;
}


private boolean InterseccaoTrianguloCubo(Vetor3f centroCaixa,Vetor3f metadeCaixa,Vetor3f vertice0,
                                         Vetor3f vertice1, Vetor3f vertice2) {
        Vetor3f vAbsolutoAresta = new Vetor3f();
        normal = new Vetor3f();

        v0 = vertice0.operatorSub(centroCaixa);
        v1 = vertice1.operatorSub(centroCaixa);
        v2 = vertice2.operatorSub(centroCaixa);

        e0 = v1.operatorSub(v0);
        e1 = v2.operatorSub(v1);
        e2 = v0.operatorSub(v2);

        vAbsolutoAresta.x = Math.abs(e0.x);
        vAbsolutoAresta.y = Math.abs(e0.y);
        vAbsolutoAresta.z = Math.abs(e0.z);

        this.metadeCaixa = metadeCaixa;
        if (!TESTE_EIXO_X01(e0.z, e0.y, vAbsolutoAresta.z, vAbsolutoAresta.y))
                return false;
        if (!TESTE_EIXO_Y02(e0.z, e0.x, vAbsolutoAresta.z, vAbsolutoAresta.x))
                return false;
        if (!TESTE_EIXO_Z12(e0.y, e0.x, vAbsolutoAresta.y, vAbsolutoAresta.x))
                return false;


        vAbsolutoAresta.x = Math.abs(e1.x);
        vAbsolutoAresta.y = Math.abs(e1.y);
        vAbsolutoAresta.z = Math.abs(e1.z);

        if (!TESTE_EIXO_X01(e1.z, e1.y, vAbsolutoAresta.z, vAbsolutoAresta.y))
                return false;
        if (!TESTE_EIXO_Y02(e1.z, e1.x, vAbsolutoAresta.z, vAbsolutoAresta.x))
                return false;
        if (!TESTE_EIXO_Z0 (e1.y, e1.x, vAbsolutoAresta.y, vAbsolutoAresta.x))
```

```
                return false;

        vAbsolutoAresta.x = Math.abs(e2.x);
        vAbsolutoAresta.y = Math.abs(e2.y);
        vAbsolutoAresta.z = Math.abs(e2.z);

        if (!TESTE_EIXO_X2(e2.z, e2.y, vAbsolutoAresta.z, vAbsolutoAresta.y))
                return false;
        if (!TESTE_EIXO_Y1(e2.z, e2.x, vAbsolutoAresta.z, vAbsolutoAresta.x))
                return false;
        if (!TESTE_EIXO_Z12(e2.y, e2.x, vAbsolutoAresta.y, vAbsolutoAresta.x))
                return false;


        MINMAX(v0.x, v1.x, v2.x, min, max);
        if (min > (metadeCaixa.x + PRECISAO) || max < (-metadeCaixa.x - PRECISAO))
                return false;

        MINMAX(v0.y, v1.y, v2.y, min, max);
        if (min > (metadeCaixa.y + PRECISAO) || max < (-metadeCaixa.y - PRECISAO) )
                return false;

        MINMAX(v0.z, v1.z, v2.z, min, max);
        if (min > (metadeCaixa.z + PRECISAO) || max < (-metadeCaixa.z - PRECISAO))
                return false;

        normal.Produto(e0, e1);

        if (!InterseccaoFaceCubo(normal, v0, metadeCaixa))
                return false;

        return true;
}

private boolean InterseccaoFaceCubo(Vetor3f normal, Vetor3f vertice, Vetor3f cubo) {
        Vetor3f vmin, vmax;

        vmin = new Vetor3f();
        vmax = new Vetor3f();
        float v;

        v = vertice.x;
        if (normal.x > 0.0f) {
                vmin.x = -cubo.x - v;
                vmax.x = cubo.x - v;
        }
        else {
                vmin.x = cubo.x - v;
                vmax.x = -cubo.x - v;
        }

        v = vertice.y;
        if (normal.y > 0.0f){
                vmin.y = -cubo.y - v;
                vmax.y = cubo.y - v;
        }
        else {
                vmin.y = cubo.y - v;
                vmax.y = -cubo.y - v;
        }

        v = vertice.z;
        if (normal.z > 0.0f){
                vmin.z = -cubo.z - v;
                vmax.z = cubo.z - v;
        }
        else {
                vmin.z = cubo.z - v;
                vmax.z = -cubo.z - v;
        }

        if (((normal.operatorMult(vmin)) - PRECISAO) >  0.0f)
                return false;
        if (((normal.operatorMult(vmax)) + PRECISAO) >= 0.0f)
                return true;

        return false;
}

private boolean coplanar_tri_tri(Vetor3f N,  Vetor3f V0, Vetor3f V1, Vetor3f V2,
                        Vetor3f U0, Vetor3f U1, Vetor3f U2) {
        Vetor3f A = new Vetor3f();

        // primeiro projeta em um plano de eixo alinhado que minimiza a área
        // dos triângulos, computa índices: i0, i1
        A.x = Math.abs(N.x);
        A.y = Math.abs(N.y);
        A.z = Math.abs(N.z);

        if (A.x > A.y) {
                if (A.x > A.z){
```

```
                        i0 = 1;    // A.x é maior
                        i1 = 2;
                }
                else {
                        i0 = 0;    // A.z é maior
                        i1 = 1;
                }
        }
        else {   // A.x<=A.y
                if (A.z > A.y){
                        i0 = 0;    // A.z é maior
                        i1 = 1;
                }
                else {
                        i0 = 0;    // A.y é maior
                        i1 = 2;
                }
        }

        // testa todas arestas de triângulo 1 contra as areastas do triângulo 2
        TESTA_ARESTAS(V0, V1, U0, U1, U2);
        TESTA_ARESTAS(V1, V2, U0, U1, U2);
        TESTA_ARESTAS(V2, V0, U0, U1, U2);

        // finalmente, testa se tri1 está totalmente em tri2 ou vice-versa
        PONTO_NO_TRIANGULO(V0, U0, U1, U2);
        PONTO_NO_TRIANGULO(U0, V0, V1, V2);

        return false;
    }
}
```

Classe MassSpring

Esta classe é responsável pelos de cálculos deformação dos objetos no ambiente.

```
public class MassSpring {
    private Vetor3f[] vertices;
    private Face[] faces;

    private Vetor3f[] vert_backup;
    private int[] indices;
    public float t_final;
    private Neighbors neighbor;
    private float k, m, d;
    public final float t=0.5f;
    private boolean termina;
    public Point3f tmp = new Point3f();

    // Construtor da Classe
    public MassSpring(Face[] faces,Vetor3f[] vertices) {
        this.vertices=vertices;
        this.faces = faces;

        neighbor = new Neighbors(vertices,faces);

        this.vertices = vertices;
        vert_backup = vertices;

    }

    public void deform(Vetor3f[] obj1_face, Vetor3f[] obj2_face, Vetor3f offset) {
        k = 0.3f;//p.getConstSpring();
        m = 300.0f;//p.getMass();
        d = 0.7f;//p.getDamping();
        //Point3f f = p.getForce();
        Vetor3f f = new Vetor3f(4.0f, 0.0f, 0.0f);

        int i = neighbor.getClicado1(obj2_face[0], offset);

        // Deformação nas camadas

        // medir tempo ->> início
        long inicio;
        long total;
        inicio = System.currentTimeMillis();
        // medir tempo ->> fim

        // início do tempo
        for (float t=0;t<3;t+=0.5) {
            // deformação do ponto I
```

```
                Point3d fs = calculaSomaFS(i,neighbor.getConnectedNodes(i));
                Point3f fr = new Point3f();
                fr.x = (float)f.x - (float)fs.x;
                fr.y = (float)f.y - (float)fs.y;
                fr.z = (float)f.z - (float)fs.z;
                calculaNP(i,fr);

                // deformação das camadas
                termina = false;
                int c=0;
                while (!termina) {
                        Vector camada = neighbor.getLayer(c++);
                        for (int l=0;l<camada.size();l++) {
                                int ponto = ((Integer)camada.get(l)).intValue();
                                fs = calculaSomaFS(ponto,neighbor.getConnectedNodes(ponto));
                                // Point3f tmp = new Point3f();
                                tmp.x = (float) fs.x;
                                tmp.y = (float) fs.y;
                                tmp.z = (float) fs.z;

                                System.out.println("forca entre " + ponto + " e " +
                                                        neighbor.getConnectedNodes(ponto) + " = " + tmp);

                                if (Math.abs(tmp.x)<0.05) termina = true;
                                if (Math.abs(tmp.y)<0.05) termina = true;
                                if (Math.abs(tmp.z)<0.05) termina = true;
                                calculaNP(ponto,tmp);
                        }
                } // fim tempo

                // medir tempo ->> início
                total = System.currentTimeMillis() - inicio;
                // System.out.println((total/1000.0)+" segundos");
                // System.out.println("Numero de Camadas: " + neighbor.getNumLayers());
                // medir tempo ->> fim
                try {
                  System.gc();
                } catch(Exception e) {
                  e.printStackTrace();
                }
        }
} //fim do método

// Deformação nas camadas

// Cálculo da fórmulas matemáticas - Newton e Hooke

public Point3d calculaSomaFS(int ponto, Vector vizinhos) {
        Point3d soma = new Point3d(0,0,0);
        Point3d t;
        for (int j=0;j<vizinhos.size();j++) {
            // calcula a força de uma mola (um ponto a outro)
            t = calculaFS(ponto, ((Integer)vizinhos.get(j)).intValue());
            soma.x += t.x;
            soma.y += t.y;
            soma.z += t.z;
        }
        return soma;
}

public Point3d calculaFS(int ponto, int outroPonto) {
        //Lei de Hooke

        double xr = 0;
        double yr = 0;
        double zr = 0;
        double x,y,z,n;

        // Lij
        x = vert_backup[outroPonto].x - vert_backup[ponto].x;
        y = vert_backup[outroPonto].y - vert_backup[ponto].y;
        z = vert_backup[outroPonto].z - vert_backup[ponto].z; ;

        // distância Euclidiana
        double l = Math.sqrt((x*x)+(y*y)+(z*z));

        // Rij
        x = vertices[outroPonto].x - vertices[ponto].x;
        y = vertices[outroPonto].y - vertices[ponto].y;
        z = vertices[outroPonto].z - vertices[ponto].z;

        // distância
        double modulo_dist = Math.sqrt((x*x)+(y*y)+(z*z));

        n = (k*(modulo_dist-l))/modulo_dist;

        // multiplicação final com o somatório
        xr = x*n;
        yr = y*n;
        zr = z*n;
```

```
                // Fim da Lei de Hooke

                Point3d resultado_R = new Point3d(xr,yr,zr);
                return resultado_R;
        }

        public void calculaNP(int ponto, Point3f f)//passa uma força só--> Point3d fs {
                float fx = f.x;
                float fy = f.y;
                float fz = f.z;

                // início - parte da deformação

                // Lei de Newton
                // xinicial
                double xi = vert_backup[ponto].x;
                // xcorrente
                double xc = vertices[ponto].x;
                // força mola age sobre os outros pontos
                double xnovo = ((((fx /*- xr*/)*(t*t))+ 2*m*xc)/(m+(d*t))) - xi;

                // yinicial
                double yi = vert_backup[ponto].y;
                // ycorrente
                double yc = vertices[ponto].y;
                double ynovo = ((((fy /*- yr*/)*(t*t))+ 2*m*yc)/(m+(d*t))) - yi;

                // zinicial
                double zi = vert_backup[ponto].z;
                // zcorrente
                double zc = vertices[ponto].z;
                double znovo = ((((fz /*- zr*/)*(t*t))+ 2*m*zc)/(m+(d*t))) - zi;

                //cast nas variáveis para ficarem com tipos compatíveis
                Vetor3f Ponto_Novo = new Vetor3f((float)xnovo,(float)ynovo,(float)znovo);

                // System.out.println(Ponto_Novo);
                vertices[ponto] = Ponto_Novo;

                // shape.atualiza(vertices);

                // fim da Lei de Newton
                // fim - parte da deformação
        }

        public Point3f getForce() {
                return tmp;
        }
}
```

Classe Neighbors

Esta classe é responsável pelos cálculos de deformação entre faces vizinhas ao ponto de colisão.

```
public class Neighbors {
        public Vetor3f vertices[];
        public Face[] faces;
        private int clicado;
        private Vector camadas,viz;

        int termina;

        public Neighbors(Vetor3f[] vertices, Face[] face) {
                viz = new Vector();
                camadas = new Vector<Vector>();

                this.vertices = vertices;
                this.faces = face;
        }

        public int getClicado1(Vetor3f pos, Vetor3f offset) {
                clicado=0;
                camadas.clear();

                System.out.println("Procurando ponto: "+ pos.x + "," + pos.y + "," + pos.z);

                double d[]=new double[vertices.length];

                for (int i=0;i<vertices.length;i++) {
                        double x = (pos.x)-(vertices[i].x + offset.x);
                        double y = (pos.y)-(vertices[i].y + offset.y);
```

```
                double z = (pos.z)-(vertices[i].z + offset.z);

                d[i] = Math.sqrt((Math.pow(x,2))+(Math.pow(y,2))+(Math.pow(z,2)));

        }

        double menor = d[0];
        for (int cont=1;cont<d.length;cont++) {
          if (d[cont] < menor) {
                menor=d[cont];
                clicado=cont;
          }
        }

        System.out.println("Ponto mais proximo no obj: " + vertices[clicado].x  + "," +
                           vertices[clicado].y + "," + vertices[clicado].z );
        System.out.println("Na posicao: " + clicado);

        return clicado;
}

public int getClicado(Vetor3f pos) {
        clicado=0;
        camadas.clear();

        System.out.println("Procurando ponto: "+ pos.x + "," + pos.y + "," + pos.z);

        double d[]=new double[vertices.length];

        for (int i=0;i<vertices.length;i++) {
                double x = (pos.x)-(vertices[i].x);
                double y = (pos.y)-(vertices[i].y);
                double z = (pos.z)-(vertices[i].z);

                d[i] = Math.sqrt((Math.pow(x,2))+(Math.pow(y,2))+(Math.pow(z,2)));
        }

        double menor = d[0];
        for (int cont=1;cont<d.length;cont++) {
          if (d[cont] < menor) {
                menor=d[cont];
                clicado=cont;
          }
        }

        System.out.println("Ponto mais proximo no obj: " + vertices[clicado].x  + "," +
                           vertices[clicado].y + "," + vertices[clicado].z );
        System.out.println("Na posicao: " + clicado);

        return clicado;
}

public Vector getConnectedNodes(int point) {
        int i=0;

        while (i <faces.length && faces[i].x != point &&  faces[i].y != point && faces[i].z != point) {
                // System.out.println(i + " - " +  faces[i].x +" - " + faces[i].y + " - " +
                //   faces[i].z +" - "+ point);
                i++;
        }

        viz = new Vector();

        if (faces[i].x == point) {
                incluirVetor(viz,faces[i].y,point);
                incluirVetor(viz,faces[i].z,point);
        } else if (faces[i].y == point) {
                        incluirVetor(viz,faces[i].x,point);
                        incluirVetor(viz,faces[i].z,point);
                } else if (faces[i].z == point) {
                                incluirVetor(viz,faces[i].x,point);
                                incluirVetor(viz,faces[i].y,point);
                }
        i++;
        return viz;
}

private void incluirVetor(Vector viz, int vizinho, int point) {
        boolean achou=false;

        if (vizinho==point)
                achou=true;

        for (int i=0;(i<viz.size())&&(!achou);i++) {
                if (((Integer)viz.get(i)).intValue()==vizinho)
                        achou=true;
        }

        if (!achou) {
                viz.addElement(new Integer(vizinho));
        }
```

```
	}

	public Vector getLayer(int c) {
		if (camadas.size()<=c)
			calculaCamada(c);
		return (Vector)camadas.get(c);
	}

	private void calculaCamada(int c) {
		if (c==0) {
			camadas.addElement(new Vector());
			procurarVizinhos(clicado,c);
		}
		else {
			int ca = c-1;
			camadas.addElement(new Vector());
			for (int i=0;i<((Vector)camadas.get(ca)).size();i++) {
				procurarVizinhos(((Integer)((Vector)camadas.get(ca)).get(i)).intValue(),c);
			}
		}
	}

	private void procurarVizinhos(int pontoBase, int camada) {
		int i=0;

		while (faces[i].x != pontoBase && faces[i].y != pontoBase && faces[i].z != pontoBase &&
		       i < faces.length )
			i++;
		if (faces[i].x == pontoBase) {
				incluirMatriz(faces[i].y,camada);
				incluirMatriz(faces[i].z,camada);
			} else if (faces[i].y == pontoBase) {
				incluirMatriz(faces[i].x,camada);
				incluirMatriz(faces[i].z,camada);
			} else if (faces[i].z == pontoBase) {
				incluirMatriz(faces[i].x,camada);
				incluirMatriz(faces[i].z,camada);
			}
		i++;
	}

	public int getNumLayers() {
		return camadas.size();
	}

	private void incluirMatriz(int vizinho, int camada) {
		// verificar se vizinho ja existe na matriz
		boolean achou=false;

		if (vizinho==clicado) {
			achou = true;
			// System.out.println("clicado");
		}

		for (int c=0;c<=camada;c++) {
			for (int i=0;i<((Vector)camadas.get(c)).size();i++) {
				// se o valor do vizinho for igual ao valor já existente, não insere
				if (((Integer)((Vector)camadas.get(c)).get(i)).intValue()== vizinho) {
					achou = true;
				}
			}
		}
		// aqui vai inserir o valor caso nao existe na matriz
		if (!achou) {
			// inclui na matriz
			((Vector)camadas.get(camada)).addElement(new Integer(vizinho));
		}
	}
}
```