UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MARILENA MAULE

Aceleração da Deformação Interativa de Corpos Sólidos Usando GPU

Trabalho de Graduação.

Prof. Luciana Nedel
Orientadora

Dr. Anderson Maciel
Co-orientador

Porto Alegre, Dezembro de 2009.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 5
LISTA DE FIGURAS 6
RESUMO 7
ABSTRACT 8
1 INTRODUÇÃO 9
2 GPGPU E TRABALHOS RELACIONADOS 11
2.1 Programação em GPU 13
2.1.1 C for Graphics (Gg) 13
2.1.2 GLSL e HLSL 13
2.1.3 Arquitetura CUDA 14
2.2 Trabalhos Relacionados 16
2.2.1 Precisão Física 16
2.2.2 Aceleração dos Cálculos 16
3 SIMULAÇÃO FÍSICA DE CORPOS DEFORMÁVEIS 17
3.1 Sistema Massa-Mola 17
3.1.1 Simulação de Tecido 18
3.1.2 Simulação de Corpos com Volume 18
3.2 Integração Numérica 19
3.2.1 Integração de Euler 19
3.2.2 Integração de Verlet 20
4 IMPLEMENTAÇÃO 21
4.1 Fluxo de Execução 21
4.1.1 Simulação de Tecido 21
4.1.2 Simulação de Corpos com Volume 22
4.2 Organização da Memória 23
4.2.1 Simulação de Tecido 23
4.2.2 Simulação de Corpos com Volume 24
4.3 Detecção e Resposta a Colisões 25
4.3.1 Colisão com o Chão 25
4.3.2 Colisão com a Ferramenta 26
4.4 Dispositivo de Retorno de Força 27
4.5 Sombra 28
4.5.1 Desenho do Ponto de Vista da Luz 28
4.5.2 Desenho do Ponto de Vista da Câmera 29
4.6 Iluminação 30
5 RESULTADOS 31
5.1 Resultados Visuais 31
5.2 Simulação de Tecido 32

5.3 Simulação de Corpos com Volume 33
5.3.1 Compartilhamento 34
5.3.2 Escalabilidade 35
5.3.3 Interatividade 36
6 CONCLUSÕES 38
REFERÊNCIAS 40
ANEXO A - SHADERS 42

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
Cg	*C for Graphics*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
FPS	*Frames per Second*
GLSL	*OpenGL Shading Language*
GPGPU	*General-Purpose Computation on Graphics Processing Units*
GPU	*Graphics Processing Unit*
GUI	*Graphics User Interface*
HLSL	*High Level Shading Language*
MPP	*Massive Parallel Processing*
RAM	*Random Access Memory*
SP	*Scalar Processor*

LISTA DE FIGURAS

Figura 2.1 - Diagrama de blocos da GeForce 8800 [GF8] 11
Figura 2.2 - Operações de ponto flutuante e Largura de banda de memória [PG] 12
Figura 2.3 - Comparação da ocupação da área do *chip* [PG] 12
Figura 2.4 - Blocos de *threads* [PG] 14
Figura 2.5 - Camadas de execução [PG] 15
Figura 3.1 - (a) mola em repouso. (b) mola comprimida. (c) mola estendida. 17
Figura 3.2 - Esquemático do sistema massa-mola para tecido 18
Figura 3.3 - Esquemático do sistema massa-mola para corpos com volume 19
Figura 4.1 - Fluxograma da simulação de tecido 22
Figura 4.2 - Fluxograma da simulação de corpos com volume 22
Figura 4.3 - Etapa de cálculo da simulação de corpos com volume 23
Figura 4.4 - Organização da memória para simulação de tecidos 24
Figura 4.5 - Cálculo do vetor velocidade 25
Figura 4.6 - Projeção da partícula no plano da ferramenta 26
Figura 4.7 - Colisão com a superfície da ferramenta 27
Figura 4.8 - Graus de liberdade do Phantom Omni 27
Figura 4.9 - *Shadow Mapping*: Mapeamento entre coordenadas da luz e da câmera 29
Figura 5.1 - Interface gráfica da simulação de tecido 31
Figura 5.2 - Interface gráfica da simulação de corpos com volume 32
Figura 5.3 - Gráficos de desempenho dos métodos numéricos na simulação de tecido. Com destaque para a queda do desempenho com o aumento da quantidade de partículas. 33
Figura 5.4 - Gráfico da comparação de desempenho quando a GPU é compartilhada e dedicada. A GPU dedicada tem melhor desempenho e a diferença é acentuada no modelo mais moderno da placa. 34
Figura 5.5 - Gráfico da comparação de desempenho quando a quantidade de partículas aumenta. A queda de desempenho é mais acentuada na CPU, enquanto a GPU mais moderna apresenta pouca perda. 35
Figura 5.6 - Gráfico de comparação de desempenho em escalabilidade e compartilhamento entre os modelos de GPU. A GPU compartilhada apresenta maior queda de desempenho. 36
Figura 5.7 - Gráfico da oscilação do desempenho médio de quadros por segundo durante a execução. As elipses vermelhas destacam a melhora do desempenho nos períodos em que não houve contato da ferramenta com o modelo. 36
Figura 5.8 - Gráfico da oscilação de desempenho médio de iterações por segundo durante a execução. As elipses vermelhas destacam a melhora do desempenho nos períodos em que não houve contato da ferramenta com o modelo. 37

RESUMO

Aplicativos que visam simular o comportamento físico de corpos deformáveis são importantes ferramentas científicas, também exploradas em jogos e animações computacionais.

O primeiro passo para fazer uma simulação é modelar o corpo que se deseja simular, e quanto mais preciso o modelo, maior será o conjunto de dados gerados. Para gerar gráficos com movimentos suaves é necessário que sejam gerados pelo menos 30 quadros por segundo. Se, no entanto, se deseja gerar gráficos interativos e essa interação inclui produzir resposta háptica – como retorno de força, por exemplo – o desempenho requerido se eleva a cerca de 1000 quadros por segundo para produzir transições suaves.

Para atingir esse desempenho existem alternativas como CPUs rápidas com vários núcleos, *clusters* de processadores com uma rede de interconexão veloz e, recentemente, o uso do co-processador gráfico (GPU) que, além de eficiente é a alternativa mais barata. A possibilidade oferecida pelas atuais placas gráficas, de operar paralelamente sob vários dados, revigora os esforços voltados à simulação física.

Assim, este trabalho se dedica a explorar a utilização do paradigma massivamente paralelo, popularizado pelas GPUs, na aceleração dos cálculos envolvidos nas iterações de um sistema de simulação de corpos deformáveis. Um esquema para simulação baseada em física de corpos deformáveis foi implementado em duas versões, uma na CPU e outra na GPU. Para medir e comparar seu desempenho, foi desenvolvida uma aplicação gráfica interativa que permite a interação háptica com retorno de força de uma ferramenta virtual com corpos deformáveis. Os resultados obtidos ajudam a ilustrar as vantagens e desvantagens introduzidas pelo uso desses dispositivos.

Palavras-Chave: GPGPU, CUDA, Computação Gráfica, Deformação Interativa.

GPU Acceleration for Interactive Solid Bodies Deformation

ABSTRACT

Applications which intend to simulate the physical behavior of deformable bodies are important scientific tools, also used in games and computer animation.

The first step to make a physical simulation is to model the body which will be simulated. An accurate model will generate a large data set. At least 30 frames per second are necessary to generate graphics with smooth movements. However, if the goal is to generate interactive graphics and such interaction includes the rendering of haptic response – as force feedback, for example – the performance requirement grows up to around 1000 frames per second to obtain smooth transitions.

There is a number of alternative ways to achieve such performance, like fast CPUs with multi cores, CPU clusters with an ultra fast network and, lately, the use of the graphics processing unit (GPU), an efficient and non-expensive alternative. The state-of-the-art graphics cards offer the possibility of many cores operating in parallel over a huge amount of data, and this brings a new impulse to the physical simulation efforts.

Thus, this work explores the use of the GPU's massively parallel paradigm to accelerate the simulation of deformable bodies. A schema for physics-based simulation of deformable bodies has been implemented in two versions, one targeted to the CPU and the other to the GPU. An interactive graphics application has also been developed to measure and compare the performance of the two implementations. Such application also provides haptic interaction with force feedback. The results obtained help in illustrating advantages and disadvantages in the use of such devices as the GPU in general purpose computation.

Key-Words: GPGPU, CUDA, Graphics Computer, Interactive Deformation.

1 INTRODUÇÃO

A simulação de deformações é uma parte da computação gráfica que vem sendo explorada sob vários aspectos e em diferentes áreas de aplicação, como simulação de materiais, aplicações médicas e jogos. Um dos problemas enfrentados na construção de um sistema computacional com deformação é obter alto desempenho para prover um retorno visual e háptico em tempo interativo, ou seja, obter uma taxa de atualização dos dispositivos de tal forma que a animação e o retorno de força sejam suaves e contínuos. A taxa de atualização mínima requerida para gerar gráficos com movimentos suaves é de 30 quadros por segundo. Se, no entanto, se deseja oferecer interação háptica sobre os modelos, essa taxa mínima de atualização se eleva a cerca de 1000 quadros por segundo, impondo severas restrições ao desempenho dos algoritmos de simulação física.

Um dos métodos mais facilmente implementados e, por consequência, mais usados nas simulações físicas de corpos deformáveis é o sistema massa-mola [MS]. A interatividade nesse tipo de simulação costuma ser limitada pela quantidade de pontos de massa simulados, pois, a cada passo, o sistema de equações de integração precisa ser resolvido para cada ponto. À medida que a quantidade de dados aumenta, torna-se muito difícil manter uma taxa aceitável de iterações, tanto para interação visual como para interação háptica, cujos requisitos de desempenho são maiores.

Simulações físicas de modo geral manipulam uma enorme quantidade de pontos e parâmetros, formando um grande conjunto de dados que sofrerão as mesmas operações, ou seja, o mesmo código é executado sobre cada dado. Há certa independência entre os dados durante o cálculo de uma iteração, o que constitui uma condição interessante de ser tratada no paradigma de GPGPU (*General-Purpose Computation on Graphics Processing Units*) [GPGPU], o qual consiste na utilização de hardware gráfico para implementações de propósito geral.

Sendo o paradigma relativamente novo, ainda não estão claros que conjuntos de problemas podem tirar vantagem do seu poder de paralelismo, servindo este trabalho, também, para explorar o alcance do paradigma. Como as placas gráficas se tornaram uma opção de baixo custo para processamento paralelo, especialmente em aplicações que manipulam grandes volumes de dados, é proposta sua utilização para acelerar o processamento de forma a permitir a interação mantendo a coerência física.

A independência dos dados durante uma iteração do sistema massa-mola torna trivial sua paralelização e este é um fator importante e motivador do uso de GPUs para seu processamento. Enquanto a CPU é obrigada a processar quase que serialmente cada ponto, a GPU, com seu grande número de unidades de processamento, é capaz de manipular vários pontos paralelamente.

Nesse contexto, o objetivo desse trabalho é avaliar o desempenho obtido com o uso auxiliar da GPU, através do desenvolvimento de uma ferramenta para simulação física de objetos deformáveis, que se valha do poder de paralelismo das placas gráficas modernas para acelerar o processamento dos dados, e tornar o desempenho da aplicação mais próximo do tempo real, provendo melhor qualidade de interação háptica com o usuário.

Inicialmente são introduzidos alguns conceitos relacionados à GPGPU, ressaltando aspectos que motivaram seu uso. Em seguida, é descrito o modelo de simulação massa-mola, a maneira como foi implementado e as técnicas usadas para melhorar a qualidade das imagens geradas.

Por fim, os resultados desse trabalho visam ressaltar os aspectos positivos e negativos do uso de um co-processador gráfico na simulação física de corpos deformáveis, levando em consideração as técnicas empregadas na simulação e na qualidade dos resultados visual e háptico. Fazendo uma análise comparativa da média de iterações com e sem o uso da GPU.

2 GPGPU E TRABALHOS RELACIONADOS

Com o crescente avanço das tecnologias empregadas na fabricação do hardware, especialmente no que diz respeito à miniaturização de componentes, é cada vez maior a quantidade de transistores encapsulados dentro de um mesmo chip. No caso das placas de processamento gráfico, essa tecnologia está sendo usada para aumentar consideravelmente o número de unidades de processamento, dando a elas um elevado poder de paralelismo.

Com o aumento da demanda por processamento, vinda principalmente do mercado de jogos, e influenciada pelas novas especificações do DirectX10 (o qual acrescentou uma etapa de processamento geométrico ao *pipeline*), a nVidia deu início à programabilidade de partes do *pipeline* gráfico. Inicialmente era possível programar o processador de vértices e o processador de fragmentos [H05], possibilitando aceleração de algoritmos e dando maior realismo às cenas geradas.

A arquitetura da GeForce 8800 [GF8] foi a primeira a unificar o conjunto de processadores, resultando em mais liberdade de programação. Ela conta com 128 processadores agrupados em oito blocos de dezesseis SPs (*Scalar Processors*), como mostra a Figura 2.1. Enquanto nas arquiteturas anteriores, os processadores trabalhavam sobre os dados de forma vetorial, cada um dos novos SPs trata um dado por vez. Isso permitiu que a frequência de operação fosse praticamente dobrada de 600MHz para 1.35GHz.

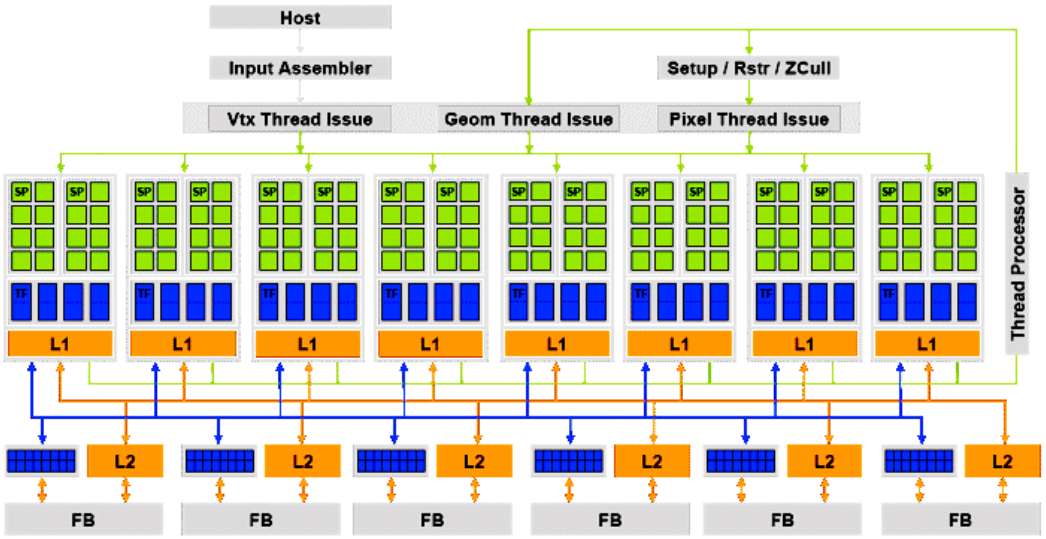

Figura 2.1 - Diagrama de blocos da GeForce 8800 [GF8]

O grande gargalo dos sistemas computacionais também tem um impacto forte sobre o desempenho das placas gráficas. Ou seja, o tempo de acesso à memória principal pode penalizar severamente a execução de determinados tipos de algoritmos (*io-bound*). Portanto, é imprescindível que os programadores dediquem algum tempo para se familiarizar com as singularidades dessa arquitetura, para poderem extrair o máximo de proveito do paralelismo oferecido.

A Figura 2.2 ilustra os gráficos que comparam o aumento do desempenho das GPUs, em relação ao aumento apresentado pelas CPUs, no que diz respeito à capacidade de transferência de memória e quantidade de números em ponto flutuante processados por segundo.

Figura 2.2 - Operações de ponto flutuante e Largura de banda de memória [PG]

Enquanto os processadores genéricos precisam de mais gerenciamento para manter o controle de fluxo e a *cache* de instruções e de dados, o chip gráfico é voltado ao processamento, ocupando o espaço da pastilha majoritariamente com unidades de execução, como mostra a Figura 2.3. Sendo especializado em processamento gráfico, ou seja, *cpu-bound* e altamente paralelo. Cada unidade executa o mesmo programa para um sub-conjunto dos dados de entrada, escondendo o atraso da memória.

Figura 2.3 - Comparação da ocupação da área do *chip* [PG]

Assim como outras arquiteturas dedicadas, a GPU é significativamente mais barata do que um processador convencional; o que incentiva seu uso não apenas no meio acadêmico, mas também em jogos e em servidores, por exemplo. Uma comparação do aumento de desempenho das GPUs e CPUs nos últimos anos é mostrada na Figura 2.2.

Em 2006, a nVidia lançou a linha Tesla, sua primeira placa voltada a processamento genérico, abrindo caminho para clusters de GPUs. Com isso as placas gráficas começam a competir pelo mercado de processamento de alto desempenho. O mais recente desenvolvimento da nVidia é a arquitetura Fermi, a qual promete ser um marco na

história do processamento paralelo, apresentando inovações como controle de erro e 512 núcleos.

Apesar das GPUs serem indicadas para paralelismo de granularidade diferente das CPUs elas são fortes concorrentes, existindo ambiciosos projetos de fusão, como o desenvolvido pela AMD [AMD_FUSION] que pretende encapsular as duas arquiteturas no mesmo *chip*. Ou seja, o futuro do processamento paralelo é promissor.

2.1 Programação em GPU

Assim que partes do *pipeline* gráfico tornaram-se programáveis, foram desenvolvidas novas linguagens de programação específicas para a arquitetura do hardware gráfico, caracterizada por várias unidades de processamento.

Essas primeiras linguagens são conhecidas como linguagens de *shader*. Foram criadas para dar mais flexibilidade ao *pipeline*, possibilitando a substituição de algumas etapas por algoritmos mais elaborados. Exigem que o programador tenha conhecimento de detalhes do *pipeline* gráfico, bem como dos comandos OpenGL ou DirectX para disparo e controle desse *pipeline*.

2.1.1 C for Graphics (Gg)

Desenvolvida pela nVidia em colaboração com a Microsoft, com o objetivo de possibilitar a programação dos processadores de vértice e de fragmento, a linguagem Cg [CG] é compatível com as APIs (*Application Programming Interface*) gráficas OpenGL e DirectX, e suportada pela maioria das placas gráficas.

Permite o desenvolvimento de programas em uma linguagem pseudo-C, que são portáveis, pois o compilador gera um código *assembler* que é compilado para binário pelo ambiente de execução (*Cg runtime*). Contudo, esses programas executam apenas nos processadores de vértices e/ou fragmentos, sendo necessário uma aplicação suporte sobre uma das APIs gráficas para ativar e controlar o *pipeline* gráfico.

Essa linguagem é usada no contexto desse trabalho para a implementação de um algoritmo de sombra, o qual melhora consideravelmente a qualidade da imagem em relação ao resultado gerado pelo *pipeline* fixo do OpenGL.

2.1.2 GLSL e HLSL

OpenGL Shading Language [GLSL], também chamada de GLslang, foi criada pelo OpenGL ARM. Considerada uma extensão do OpenGL e otimizada para a API. A *High Level Shading Language* [HLSL], por sua vez, foi criada pela Microsoft e otimizada para a API DirectX.

Ambas foram desenvolvidas com base na linguagem Cg, mas levando em conta os interesses dos grupos que desenvolvem as APIs gráficas. E, como a própria Cg, são voltadas ao desenvolvimento de algoritmos sobre a transformação de vértices e colorização de fragmentos, também sendo necessário o uso de programas hospedeiros na CPU. Apesar de poderem ser usadas com propósito genérico, são mais indicadas para programação de algoritmos gráficos.

2.1.3 Arquitetura CUDA

Desenvolvido pela nVidia, o modelo de programação paralela CUDA (*Compute Unified Device Architecture*) [CUDA] foi projetado para facilitar a programação de propósito geral no hardware gráfico sendo, portanto, a melhor escolha para o desenvolvimento deste trabalho.

As abstrações de programação providas pela linguagem baseiam-se em hierarquia de *threads*, memória compartilhada e barreiras de sincronização.

A grande vantagem de CUDA é ser independente de qualquer API gráfica e não obrigar o programador a manter um controle sobre o *pipeline* gráfico para que se dê o processamento nas unidades programáveis, ao contrário de outras linguagens voltadas para a arquitetura das GPUs. Contudo, assim como as outras linguagens, também precisa de um processo hospedeiro executando em CPU, além de ser suportada apenas por placas da nVidia.

Da mesma forma que o hardware gráfico é dividido em blocos de unidades de execução, a linguagem expõe claramente para o programador as *threads* divididas em blocos capazes de cooperar e compartilhar memória. Por sua vez, os blocos são organizados em *grids* uni ou bidimensionais, como mostra a Figura 2.4.

Figura 2.4 - Blocos de *threads* [PG]

Para executar uma função em GPU é necessária uma inicialização feita em CPU, seguida de uma invocação da função, chamada de função de *kernel*. Essa inicialização deve, basicamente, tratar da alocação da memória no dispositivo e definir os parâmetros de execução, como quantidade de blocos e *threads*, por exemplo. Depois da execução, os dados podem ser copiados para a memória de trabalho e exibidos. Uma ilustração da sintaxe utilizada na invocação de uma chamada de *kernel* em CUDA é a multiplicação de matrizes abaixo:

```
//-- Alocação de memória
                                        ...
cudaMalloc( (void**) &(C), mem_size);

//-- Chamada de kernel
dim3 dimBlock(16, 16);
dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
             (N + dimBlock.y - 1) / dimBlock.y);
matAdd<<<dimGrid, dimBlock>>>(A, B, C);

//-- Resgate do resultado
cudaMemcpy(C,resultado,mem_size,cudaMemcpyDeviceToHost);
```

Um bloco de 16x16 contém 512 *threads*, e a *grid* é criada de tal forma que cada *thread* será responsável pelo processamento de um elemento da matriz. A gerência das *threads* é feita pelo *runtime*, podendo a quantidade de *threads* exceder a de processadores.

Há três níveis de hierarquia de memória vistos por uma *thread*. Sua memória local, privada de acesso exclusivo; a memória compartilhada pelas *threads* do mesmo bloco e a memória global, acessada por todas as *threads* em execução. Há também duas memórias protegidas contra escrita, a memória de textura e a de constantes. Ambas são mantidas entre as várias execuções do *kernel* e otimizadas para acesso rápido.

CUDA possui chamadas de funções específicas para troca de dados entre a memória do dispositivo (*device*) e a memória do sistema (*host*), uma vez que assume a execução do *kernel* na GPU como um co-processador, enquanto o resto do programa executa as chamadas em CPU. Na Figura 2.5 é apresentado um esquemático extraído do *CUDA Programming Guide*:

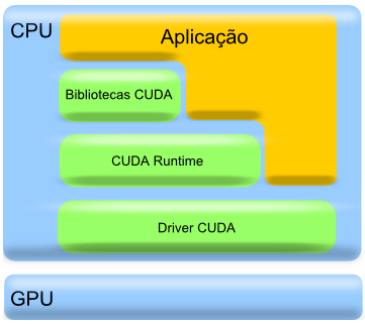

Figura 2.5 - Camadas de execução [PG]

A camada de aplicação suporta várias linguagens, entre elas C++, Java e Python.

A camada de bibliotecas é composta por funções matemáticas de mais alto nível.

O *runtime* é responsável, entre outros, pela distribuição e gerência das *threads* pelos processadores disponíveis.

2.2 Trabalhos Relacionados

Simulação física é uma área bastante pesquisada. Os métodos utilizados devem lidar com o compromisso entre desempenho e exatidão. Diversas otimizações já foram propostas, algumas sacrificando a exatidão física quando esta não é imprescindível, outras se conformando a uma baixa no desempenho reduzindo a possibilidade de interação.

A escolha da técnica usada depende do foco do trabalho e resultado ao qual se pretende chegar. Existem várias propostas de métodos para melhorar a qualidade das simulações de deformação, sejam elas voltadas à aceleração dos cálculos ou à precisão física. Ao contrário da maioria das otimizações, a implementação da simulação física na GPU permite uma considerável melhora no desempenho sem uma perda proporcional em exatidão física. Alguns exemplos são apresentados abaixo.

2.2.1 Precisão Física

Trabalhos com o objetivo de realizar uma simulação física confiável se voltam ao desenvolvimento de métodos acurados. Contudo, métodos com maior precisão são mais lentos. Alguns trabalhos exploram métodos implícitos de deformação a partir da convolução de formas rígidas, como [RJ07] e [SOG08]. Esses trabalhos são capazes de gerar uma simulação em tempo real, porém sofrem com a falta de precisão física, uma vez que a deformação é aproximada por um envoltório.

Existem técnicas mais exatas que priorizam a precisão física, como em [SLMS06] que usa o método de elementos finitos. Esse tipo de abordagem apresenta custo computacional bastante alto, o que compromete o desempenho, sendo pouco indicado para interação háptica.

2.2.2 Aceleração dos Cálculos

Métodos que objetivam acelerar os cálculos frequentemente exploram novas tecnologias e arquiteturas. Sejam elas supercomputadores, ou mesmo tecnologias alternativas como as modernas placas de processamento gráfico.

Trabalhos, como este, tentam explorar o uso das placas gráficas para acelerar os cálculos da simulação. Em [GW05] e [MHS05], por exemplo, são apresentadas técnicas de implementação do sistema massa-mola em GPU. Ambos sofrem de limitações impostas pelas linguagens de *shader*, sendo que o segundo usa conectividade implícita, o que limita o detalhamento do modelo.

O trabalho apresentado aqui prioriza o desempenho para oferecer interação háptica de qualidade. Acelerando os cálculos através do uso de métodos explícitos de deformação e do *hardware* gráfico, ao mesmo tempo em que provém liberdade quanto ao detalhamento do modelo.

3 SIMULAÇÃO FÍSICA DE CORPOS DEFORMÁVEIS

Existe uma série de maneiras de simular computacionalmente a deformação de corpos sólidos. Cada uma apresenta vantagens e desvantagens em termos de estabilidade, precisão e tempo de resposta. Sistemas implícitos, por exemplo, costumam oferecer uma boa precisão e estabilidade, contudo, são mais custosos. Os sistemas explícitos são mais rápidos, porém difíceis de parametrizar de modo a terem boa estabilidade.

O sistema escolhido foi o explícito porque seu baixo custo computacional a cada iteração provém melhor qualidade de resposta háptica.

As sessões a seguir descrevem o modelo massa-mola e as equações de integração numérica para resolução do sistema.

3.1 Sistema Massa-Mola

O sistema massa-mola é um método amplamente utilizado em simulações físicas, especialmente de tecidos e corpos moles. Ele consiste basicamente de uma série de partículas de certa massa interligadas através de molas de certa constante elástica.

Apesar do modelo massa-mola não possuir um controle preciso sobre as propriedades visco-elásticas dos materiais, ele é de fácil implementação, altamente paralelizável e apresenta um desempenho aceitável.

O princípio de funcionamento do sistema é bastante simples. Supondo uma partícula interligada a um ponto fixo, ou de massa infinita, através de uma mola. Uma vez que essa mola seja comprimida, surgirá uma força de reação de direção contrária que tende a afastar as partículas. Se a mola for esticada, a força de reação tenderá a reaproximar as partículas. Veja a Figura 3.1.

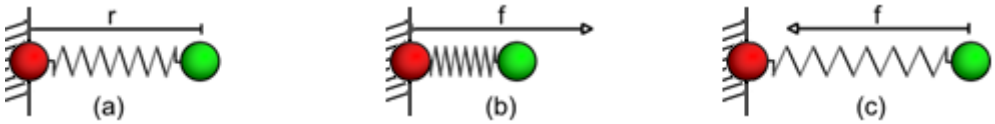

Figura 3.1 - (a) mola em repouso. (b) mola comprimida. (c) mola estendida.

As forças de reação da mola continuarão agindo até que esta volte à posição de repouso ou, caso haja alguma força externa, até que as forças convirjam para um estado de equilíbrio.

As forças atuantes do sistema são avaliadas para todas as partículas individualmente a cada passo da iteração, levando em consideração sua posição em relação às partículas com as quais está interligada pela mola, sendo estas consideradas fixas durante a avaliação pontual.

3.1.1 Simulação de Tecido

Supondo que todas as partículas são soltas e de massa finita, podendo uma interagir com as outras através das molas que as ligam. Em uma implementação inicial, o esquemático do sistema capaz de simular tecidos pode ser visto na Figura 3.2.

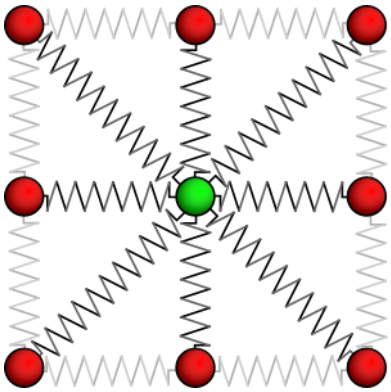

Figura 3.2 - Esquemático do sistema massa-mola para tecido

Nesse sistema cada partícula é ligada a, pelo menos, três outras partículas , e no máximo a oito vizinhas. A iteração do sistema para cada partícula considera todas as molas ligadas a ela.

Dentro do sistema massa-mola, a força exercida por cada mola i é calculada da seguinte maneira:

$$\boldsymbol{F_i = -k(l_i - r_i)\widehat{d_i}}$$

Onde $\widehat{\boldsymbol{d}}$ é o vetor direção da força, **k** é a constante elástica, **l** é o comprimento atual e **r** é o comprimento da mola em estado de repouso.

Partindo da Segunda Lei de Newton, $\boldsymbol{F = ma}$, obtemos a aceleração no ponto:

$$\boldsymbol{a = {\sum F_i}/{m_i}}$$

Esses dados são usados em um sistema de integração numérica para obter a nova posição da partícula no próximo instante de tempo.

3.1.2 Simulação de Corpos com Volume

A Figura 3.3 mostra o sistema de molas de um modelo tetraedraelizado de um fígado humano. Nesse modelo, cada partícula está ligada a uma variada quantidade de molas, o que não muda o modo como é feito o cálculo dos parâmetros da integração numérica, apenas afeta a organização da memória.

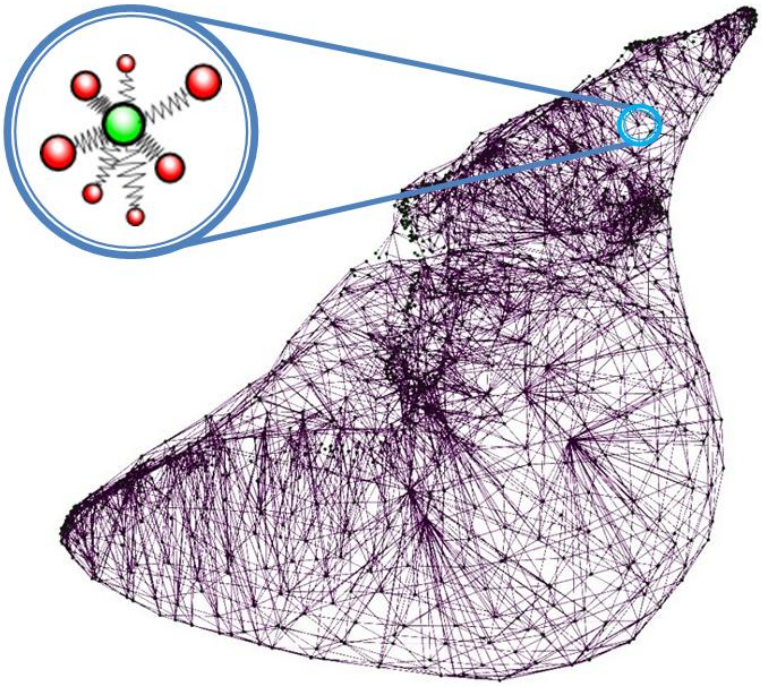

Figura 3.3 - Esquemático do sistema massa-mola para corpos com volume

3.2 Integração Numérica

A escolha do método de integração é impactante sobre o desempenho e convergência do sistema. É importante levar em consideração o custo-benefício de cada método, sempre tendo em mente o resultado ao qual se quer chegar. Nesse trabalho se optou por usar métodos mais simples visando obter desempenho para interação háptica.

Métodos mais apurados, como Runge-Kutta [RK], trazem maior precisão, porém demandam maior quantidade de cálculos fazendo com que a computação seja mais lenta. Enquanto métodos mais simples como Euler [EU] e Verlet [VE] proporcionam boa estabilidade a um menor custo computacional.

Para fins comparativos de desempenho, estabilidade e convergência, foram implementados dois métodos de integração numérica na simulação de tecidos, Euler e Verlet. Ambos os resultados foram satisfatórios, não havendo diferenças significativas sob os aspectos que interessam à aplicação, ou seja, média de iterações por segundo. Sendo assim, apenas o método de Euler foi mantido durante a implementação do sistema com volumes.

3.2.1 Integração de Euler

A integração de Euler é o mais simples dos métodos de integração numérica para equações diferenciais ordinárias. Ele assume que para uma dada parcela de tempo todos os valores da equação permanecem constantes, o quê nem sempre é verdade, e o faz menos acurado. O ajuste dos parâmetros do sistema é um problema desafiador, do qual depende a estabilidade da solução.

A implementação do método insere na equação uma constante de dissipação para emular o atrito e, consequentemente, a perda de energia do sistema, como mostram as equações a seguir,

$$a_i = {\sum \mathbf{F_{ij}}}/{\mathbf{m_i}} + \mathbf{g}$$

$$v_{i+1} = v_i\ (1\text{-}d) + a_i \Delta \mathbf{t}$$

$$x_{i+1} = x_i + v_{i+1}$$

onde, x_i representa a posição, v_i a velocidade, a_i a aceleração e m_i, a massa de cada partícula no instante de tempo **i**. Enquanto F_{ij} representa a força exercida pela mola **j** sobre a partícula **i**, d representa a porcentagem de energia que será dissipada e g a gravidade atuante sobre as partículas.

3.2.2 Integração de Verlet

A integração de Verlet é um método de integração numérica que tenta estimar um estado intermediário entre os intervalos de tempo. E, portanto, bastante usado não apenas na implementação de sistemas massa-mola, mas também em dinâmica de partículas para jogos.

A implementação desse método também insere na equação uma constante de dissipação para emular o atrito e, consequentemente, a perda de energia do sistema, como mostram as equações a seguir.

$$a_i = {\sum \mathbf{F_{ij}}}/{\mathbf{m_i}} + \mathbf{g}$$

$$x_{i+1} = x_i + (x_i - x_{i-1})\ (1\text{-}d) + a_i\ \Delta \mathbf{t^2}$$

Onde, x_i representa a posição, a_i a aceleração e m_i, a massa de cada partícula **i**. Enquanto F_{ij} representa a força exercida pela mola **j** sobre a partícula **i**, d representa a porcentagem de energia que será dissipada e g representa a gravidade.

Os resultados obtidos com a utilização da integração de Verlet são bastante similares aos resultados obtidos com a integração de Euler, como mostram os gráficos na seção de resultados. Sendo o desempenho bastante similar, para a segunda etapa do trabalho, apenas o método de Euler foi mantido.

4 IMPLEMENTAÇÃO

Essa seção trata dos métodos usados na implementação da aplicação. Foram usadas as linguagens de programação C/ C++, Cg e CUDA (versão 2.0), compiladas no ambiente Microsoft Visual Studio 2005 (VS8) com auxilio dos compiladores próprios das linguagens para GPU.

A paralelização do código em CPU foi feita através da API de programação paralela OpenMP, sendo que a quantidade de *threads* criadas é sempre igual à quantidade de núcleos do processador.

A criação da GUI (*Graphics User Interface*) foi feita com a biblioteca wxWidgets e o *rendering* com a API gráfica OpenGL.

4.1 Fluxo de Execução

Os fluxogramas a seguir descrevem, em alto nível, a sequência de operações usada nas implementações das simulações.

4.1.1 Simulação de Tecido

Para a simulação de tecido, o fluxo de execução não muda quando usado Euler ou Verlet, sendo que a única diferença é a interpretação da matriz intermediária.

Seguindo o esquemático da Figura 4.1, a etapa de inicialização consiste na alocação das matrizes e geração das coordenadas das partículas.

A cópia da matriz de posições estabelece uma barreira temporal. Em GPU, cada unidade de processamento gera o resultado para uma partícula e, a cada execução, as posições são copiadas de volta para a RAM. Em CPU, são criadas tantas *threads* quantos núcleos houver no processador, e a cada uma é dado um subconjunto de partículas. Como as matrizes de leitura e escrita não são as mesmas, não há necessidade de sincronização para leitura ou escrita de dados. Para gerar uma animação suave com um passo de integração pequeno, não feitos N cálculos para cada atualização do desenho.

Na etapa de cálculo, a matriz de posições das partículas é copiada para uma matriz auxiliar. Os cálculos para cada partícula são distribuídos entre as *threads* que acessam os dados da matriz auxiliar. Por fim, o resultado dos cálculos é a nova posição da partícula, escrita na matriz original das posições.

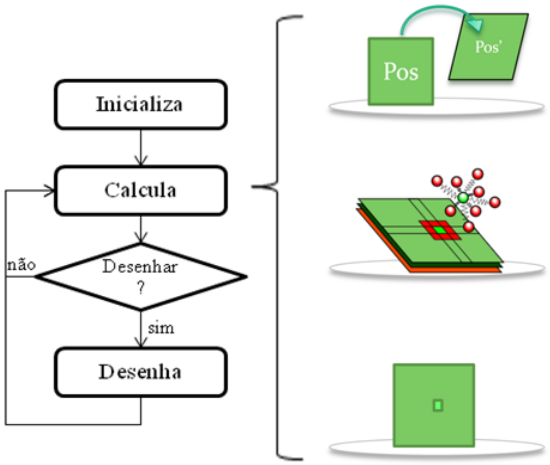

Figura 4.1 - Fluxograma da simulação de tecido

Ao processar uma partícula i, o mesmo índice é usado para acessar as matrizes intermediária, auxiliar e de posições, atuais e constantes (Seção 4.2).

4.1.2 Simulação de Corpos com Volume

Para simulação de corpos com volume apenas a integração de Euler foi mantida. Seguindo o esquemático da Figura 4.2, a etapa de inicialização lê os arquivos contendo os dados do modelo tridimensional. Ao todo são três arquivos contendo as posições das partículas, os triângulos que compõem a superfície do modelo e os tetraedros que definem a conectividade entre as partículas, ou seja, as molas.

Diferentemente do fluxo de execução da simulação de tecido, a simulação de volume apenas recupera os valores das posições da memória da GPU para a RAM quando é necessário desenhar a cena.

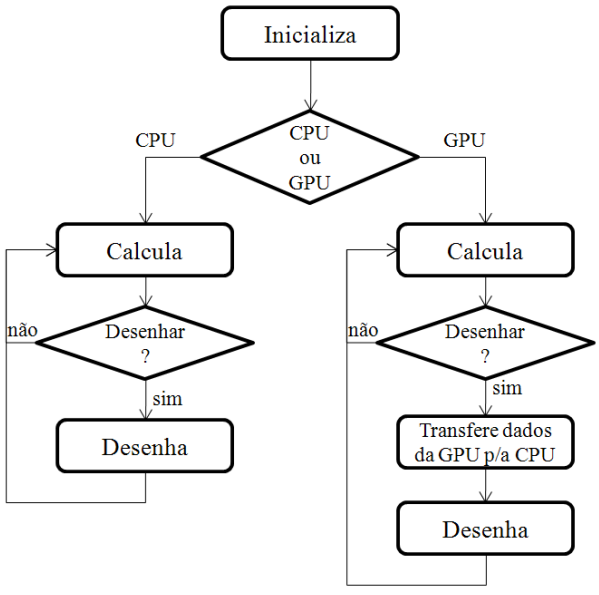

Figura 4.2 - Fluxograma da simulação de corpos com volume

Devido ao aumento da complexidade do modelo simulado, a etapa de cálculo ocupa mais memória para armazenar a conectividade das partículas. Assim como na simulação de tecidos, o índice da partícula é usado para acessar seus dados no vetor de posições, auxiliar e de conectividade (Seção 4.2). Para acessar os vizinhos da partícula **i**, é feito um acesso à posição **i** do vetor de conectividade, que fornece os dados para acesso à lista de vizinhos. Na lista de vizinhos são recuperados os índices das partículas que se conectam à partícula **i** e, com esses índices, são acessadas as matrizes de posição. Veja a Figura 4.3.

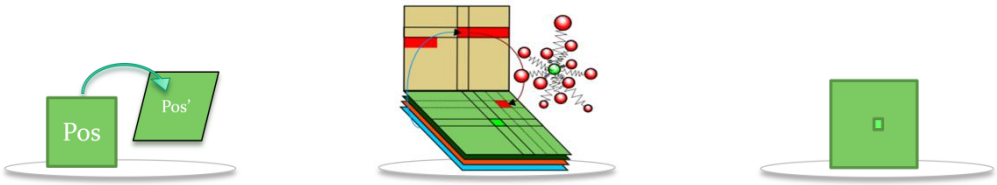

Figura 4.3 - Etapa de cálculo da simulação de corpos com volume

Como última fase da etapa de cálculo é feita a detecção de colisão, descrita na Seção 4.3. A resposta de força enviada ao dispositivo háptico é feita por uma *callback* independente, como descrito na Seção 4.4.

4.2 Organização da Memória

A maneira como os dados são organizados e acessados na memória pode comprometer o desempenho da solução. Isso porque a velocidade de acesso à memória não acompanhou a aceleração dos microprocessadores que, mesmo com o uso de *caches*, podem perder vários ciclos a espera de um dado. Portanto, há necessidade de estruturas e programação eficientes que permitam acesso aos dados o mais rápido possível.

Levando isso em consideração, para as diferentes etapas do trabalho também foram diferentes as técnicas usadas, baseadas nos trabalhos [DCN06] e [OLG07], e descritas a seguir.

4.2.1 Simulação de Tecido

O armazenamento dos dados da simulação de tecido foi estruturado em três matrizes, cujas dimensões correspondem diretamente às quantidades de partículas do retângulo de tecido a ser simulado. Essas matrizes são:

- Matriz de posições iniciais: essa matriz é constante e representa o estado inicial do corpo. A distância entre duas partículas nessa matriz corresponde ao comprimento das molas em repouso, usado no cálculo das forças do sistema durante as iterações.

- Matriz de posições: essa matriz armazena as posições que sofrerão os efeitos das forças atuantes no sistema. A cada passo seus dados são iterados e é sobre eles que é feito o desenho e a detecção de colisão.

- Matriz auxiliar: essa matriz armazena as novas posições, resultantes da iteração do sistema, criando uma barreira temporal que garante a constância dos dados durante a iteração e, ao mesmo tempo, eliminando uma dependência de dados na execução paralela entre as partículas.

- Matriz intermediária: essa matriz armazena os dados intermediários necessários à integração numérica. Na integração de Euler ela contém as velocidades de cada partícula, enquanto na integração de Verlet ela contém as posições das partículas no tempo *t-1*.

As molas são representadas implicitamente através da vizinhança de oito nas matrizes, como mostra a Figura 4.4.

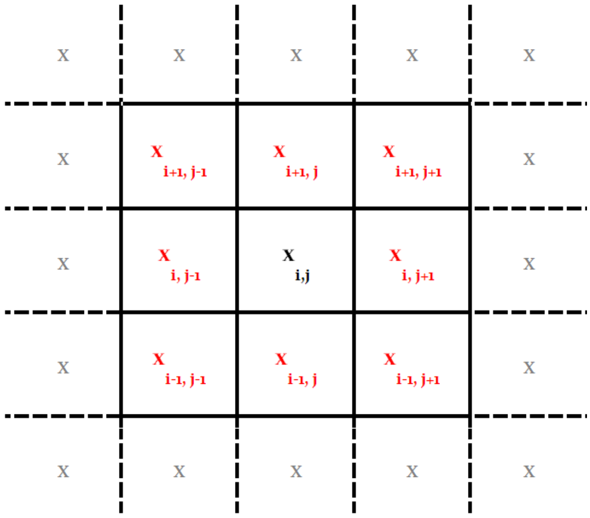

Figura 4.4 - Organização da memória para simulação de tecidos

Ao avaliar a partícula $\boldsymbol{x_{ij}}$, é calculado um somatório das forças geradas por cada uma das molas entre os índices vizinhos, semelhante a um *kernel* de convolução.

4.2.2 Simulação de Corpos com Volume

Assim como na simulação de tecidos, a simulação de corpos com volume também usa quatro *arrays* para guardar os dados iterados. Como para esta etapa foi mantido apenas o método de Euler, a matriz intermediária corresponde apenas à velocidade.

Para um corpo com volume o sistema se torna mais complexo, não permitindo mais a declaração implícita das molas. A estrutura que antes armazenava a posição das partículas, e cuja vizinhança dessa posição determinava as molas ligadas à partícula, agora é separada em dois vetores:

- Vetor de conectividade: para cada partícula **i**, armazena a quantidade total de vizinhos e o índice do primeiro vizinho no vetor de vizinhos.

- Vetor de vizinhos: essa matriz armazena sequencialmente as listas índices dos vizinhos de cada partícula.

Além desses, também foi criado um vetor que relaciona cada partícula à normal do plano que implementa a ferramenta de interação háptica. Esse vetor guarda, para a iteração anterior, um valor inteiro que informa se a partícula se encontrava na frente ou

atrás do plano. Essa informação é importante no tratamento da detecção de colisão do objeto com a ferramenta.

Adicionalmente, temos as estruturas usadas no desenho. Uma lista de normais dos vértices da superfície e uma lista de triângulos, contendo os índices das partículas que constituem seus vértices.

4.3 Detecção e Resposta a Colisões

A detecção de colisão costuma ser uma das etapas mais custosas da simulação física, especialmente quando trabalhamos com corpos deformáveis, nos quais as mudanças topológicas requerem atualizações constantes nas estruturas de dados.

Foram estudados vários métodos de detecção de colisão como estruturas baseadas em hierarquias e particionamento do espaço, métodos probabilísticos e métodos baseados em imagens em [JP04], [MBT07], [KHM98] e [TKZ04]. Contudo, a ferramenta implementada é tão simples que o teste de colisão tem um custo muito baixo, não compensando o gasto com a atualização de uma estrutura. Assim, os testes são feitos para todas as partículas. Caso houvesse mais objetos na cena, ou também ferramentas mais complexas, a quantidade de testes poderia crescer exponencialmente e seria indispensável o uso de uma estrutura apropriada.

4.3.1 Colisão com o Chão

O chão da cena é definido como o plano y=0, assim sendo, para determinar se uma das partículas do sistema atravessou o chão, basta testar a componente y da sua posição. Se essa componente for menor do que zero, significa que a partícula deve ter sua posição corrigida da seguinte forma:

- Sua componente y é zerada e
- Sua velocidade é recalculada.

O vetor velocidade é decomposto em suas componentes: horizontal e vertical, como mostra a Figura 4.5.

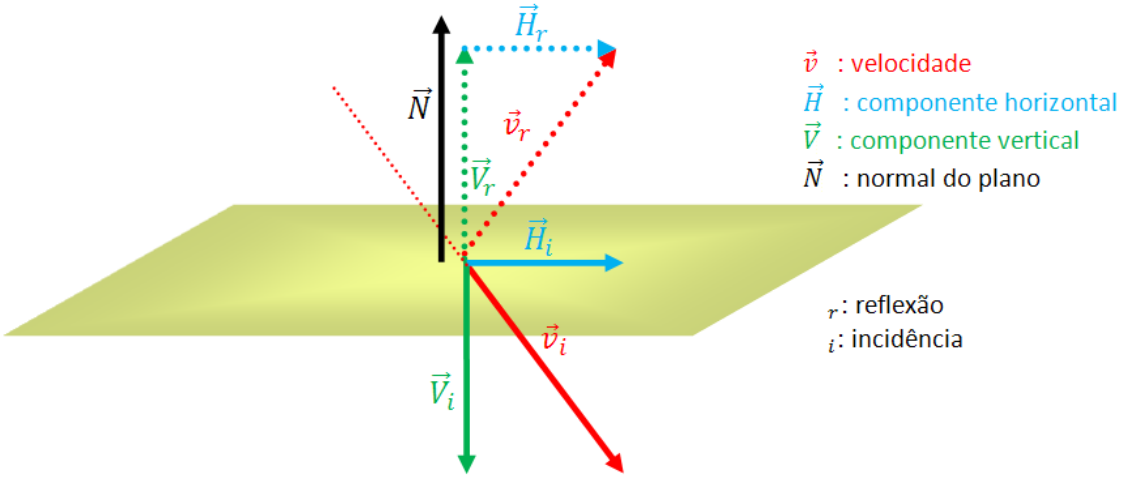

Figura 4.5 - Cálculo do vetor velocidade

Na reflexão a componente $\vec{V}$ é invertida, onde $\vec{V}_i$ é a projeção do vetor velocidade na normal do plano. Sendo que a projeção de um vetor $\vec{v}_1$em outro vetor $\vec{v}_2$ é igual a $|\vec{v}_1|\ \cos(\alpha)\ \widehat{v_2}$, e que $\cos(\alpha) = \left(\frac{\vec{v}_1 \cdot \vec{v}_2}{|\vec{v}_1||\vec{v}_2|}\right)$, temos:

$$\vec{V}_r = -\vec{V}_i = |\vec{v}_i|\left(\frac{\vec{v}_i \cdot \vec{N}}{|\vec{v}_i|}\right)\vec{N} \rightarrow \vec{V}_r = -(\vec{N} \cdot \vec{v}_i)\vec{N}$$

A componente $\vec{H}$ permanece constante e sua reflexão é definida como:

$$\vec{H}_r = \vec{H}_i = \vec{v}_i - \vec{V}_i \rightarrow \vec{H}_r = \vec{v}_i - (\vec{N} \cdot \vec{v}_i)\vec{N}$$

Dessa forma, o vetor velocidade resultante após a colisão é a soma das suas componentes após a reflexão:

$$\vec{v}_r = \vec{V}_r + \vec{H}_r \rightarrow \vec{v}_i - 2(\vec{N} \cdot \vec{v}_i)\vec{N}$$

4.3.2 Colisão com a Ferramenta

A ferramenta de interação foi implementada de forma a representar uma pá virtual com a qual o usuário pode interagir com o modelo 3D, obtendo retorno de força.

Da mesma forma com que é feita a colisão com o chão, a colisão com a ferramenta também é equacionada em termos da normal do plano que representa a superfície da pá. Assim, a posição da partícula após a colisão é o ponto em que ela se projeta sobre o plano, dada por $\boldsymbol{P_n = P_o - \vec{N}d}$, onde $\boldsymbol{P_n}$ é a nova posição, $\boldsymbol{P_o}$ é a posição antiga, $\vec{N}$ é a normal do plano que representa a ferramenta e $\boldsymbol{d}$ é a distância de $\boldsymbol{P_o}$ ao plano, como mostra a Figura 4.6.

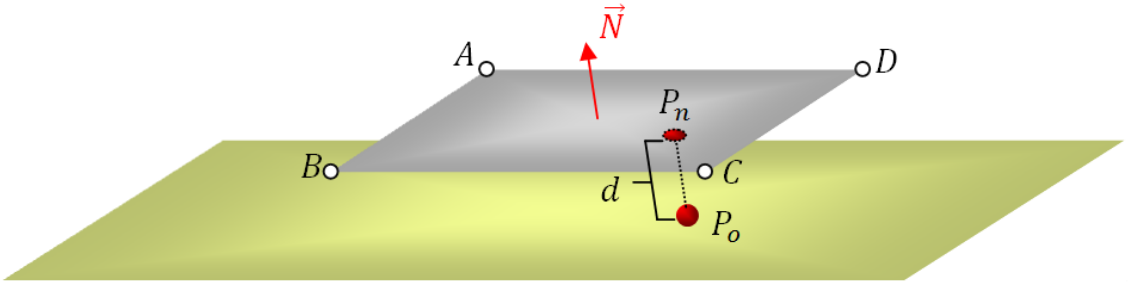

Figura 4.6 - Projeção da partícula no plano da ferramenta

Na Figura 4.7 podemos visualizar o teste que delimita a região retangular que corresponde à superfície da ferramenta. São criados quatro vetores partindo de cada um dos quatro cantos da ferramenta em direção ao ponto projetado na superfície do plano. Esses vetores são então normalizados e testados contra as bordas da ferramenta.

Por exemplo, veja a Figura 4.7, o vetor que liga o ponto **B** ao ponto $\boldsymbol{P_n}$ forma um ângulo $\boldsymbol{\alpha}$ com o vetor que liga o ponto **B** com o ponto **C**. E assim, para cada um dos cantos da ferramenta o ângulo é formado entre os vetores que partem do canto em questão e chegam, um ao ponto projetado e o outro ao próximo canto em sentido anti-horário. Se algum desses ângulos for maior do que 90º, então não há colisão.

Em termos de implementação, foi usado o cosseno do ângulo, dado pelo produto escalar entre os vetores normalizados. Se para todos os produtos o resultado for maior do que zero, então há colisão.

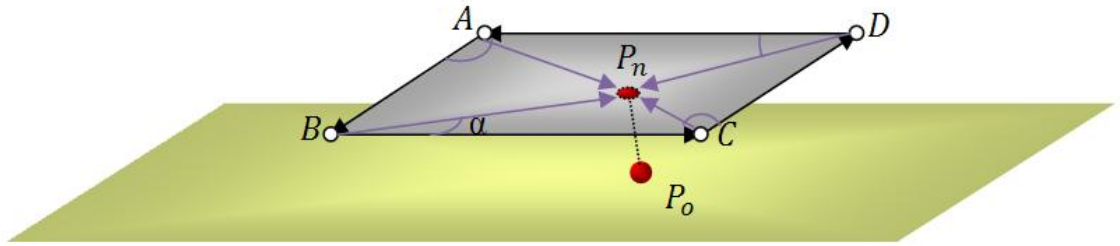

Figura 4.7 - Colisão com a superfície da ferramenta

A força de retorno para interação é calculada a partir da redução das distâncias de penetração das partículas no plano da ferramenta. O vetor de força gerado segue a orientação da normal desse plano e tem o comprimento da maior penetração. Na seção 4.4 é apresentado o dispositivo que foi usado nos experimentos para *renderizar* o retorno de força.

4.4 Dispositivo de Retorno de Força

O dispositivo de interface háptica usado foi o Phantom Omni, fabricado pela empresa SensAble. Ele é capaz de fornecer retorno de força de até 5N, provendo ao usuário seis graus de liberdade de movimento. A Figura 5.1 ilustra os movimentos possíveis de serem feitos com esse dispositivo. Mais informações sobre esse dispositivo podem ser encontradas na página web da empresa [PO].

Figura 4.8 - Graus de liberdade do Phantom Omni

A programação para esse dispositivo é feita através de uma biblioteca na linguagem C++, fornecida pela própria SensAble. Ela fornece várias rotinas e funções de *callback* para intermediar a entrada e saída de dados.

Nesse trabalho, o dispositivo é usado para movimentar uma pá virtual, onde a empunhadura representa o cabo da pá. Para tal, os dados providos pelo controlador do dispositivo são a matriz de transformação da empunhadura e a posição do ponto em que ela se prende ao braço mecânico. Com essas informações, são definidos uma normal e

quatro pontos, que representam respectivamente o plano da pá e a delimitação da superfície desse pano.

A normal e os pontos da ferramenta são transformados de acordo com a matriz fornecida pelo controlador e escaladas por uma matriz constante que define o tamanho do espaço de trabalho. Com esses dados obtém-se a pá virtual tal como pode ser vista na Figura 4.7.

Após a detecção de uma colisão do objeto com a ferramenta, como descrito no capítulo 3.4, é calculado o valor do retorno de força. Esse retorno é representado na forma de um vetor tridimensional, cuja norma representa a quantidade de força que será aplicada pelo dispositivo. Essa quantidade é obtida através da maior distância de penetração das partículas na ferramenta, e a direção do retorno acompanha a normal do plano.

O dispositivo recebe esse vetor de força através de uma *callback* da biblioteca, onde além da força aplicada pelo modelo, também é feito o cálculo da resistência do chão da cena sobre a ferramenta. Ambas as forças são somadas e truncadas, se necessário, na maior força suportada pelo dispositivo. A função que obtém os dados vindos do dispositivo também é chamada dentro dessa *callback*.

O Phantom Omni, por ser um equipamento relativamente barato, não possui uma frequência de atualização tão elevada, chegando ao máximo de 1kHz. Os experimentos descritos no capítulo a seguir demonstram que os *desktops* atuais suportam freqüências de operação superiores a essa.

4.5 Sombra

A sombra é um elemento de cena muito importante para expressar a relação de proximidade entre os objetos. Através dela, o usuário pode ter uma ideia mais exata da distribuição espacial dos objetos e, com isso, interagir de forma mais intuitiva com uma cena de três dimensões exibida em um dispositivo de apenas duas dimensões.

O algoritmo foi implementado usando a linguagem *C for Graphics*, a qual permite a substituição dos programas que executam nos processadores de vértices e de fragmentos. Uma vez substituído o programa padrão para uma parte do *pipeline* gráfico, é preciso que o novo programa se encarregue de efetuar as operações que se deseja manter, como transformação de vértices, clipping e iluminação. Portanto, uma vez que se fez necessária a implementação de um modelo local de iluminação para processar os fragmentos, escolheu-se um algoritmo com resultados visuais melhores do que o padrão implementado pelas placas.

O algoritmo de sombra escolhido chama-se *Shadow Mapping* [PK09]. Esse algoritmo consiste em duas etapas de *rendering*, uma do ponto de vista da luz, e outra do ponto de vista da câmera.

4.5.1 Desenho do Ponto de Vista da Luz

Uma câmera virtual é posicionada de modo a visualizar a parte da cena que se deseja iluminar pelo ponto de luz. A matriz *Modelviewproj_light* é atualizada de acordo com as coordenadas definidas por essa câmera virtual, os programas básicos que processam

os vértices e fragmentos no *pipeline* gráfico são substituídos por programas específicos, e então a cena é desenhada.

No programa de vértices, transformamos as coordenadas de acordo com as matrizes definidas pela câmera virtual da luz. A diferença para o programa padrão é a saída de uma textura contendo as posições transformadas, que serão usadas no processamento dos fragmentos no desenho do ponto de vista da câmera.

No programa de fragmentos é efetuado o *clipping* e, ao contrário do programa padrão, ao invés de desenhar as cores do fragmento, a profundidade do pixel é o valor salvo na saída do programa.

O resultado desse desenho é salvo em uma textura que servirá de entrada para a próxima passada de desenho.

4.5.2 Desenho do Ponto de Vista da Câmera

Nesse passo, a câmera virtual que indica o ponto de vista do usuário é quem define as coordenadas de visualização. A matriz *Modelviewproj* é atualizada de acordo com as coordenadas dessa câmera virtual, e a matriz *Modelviewproj_light* segue inalterada. Os programas padrão são substituídos por novos programas, a textura resultante do passo anterior é passada como parâmetro para o processo de fragmentos e a cena é desenhada.

No programa de vértices, transformamos as coordenadas de acordo com a matriz *Modelviewproj* e passamos para o programa de fragmentos uma textura contendo as coordenadas transformadas pela *Modelviewproj_light*.

No programa de fragmentos é efetuado o clipping das coordenadas extraídas da textura, essas então são usadas para acessar a textura vinda do passo anterior (que contém o mapa de profundidade do ponto de vista da luz). O valor obtido é usado para comparar as profundidades e determinar se um dado fragmento visto pela câmera também é visto pela luz; se sim, o pixel está iluminado, senão, está em sombra.

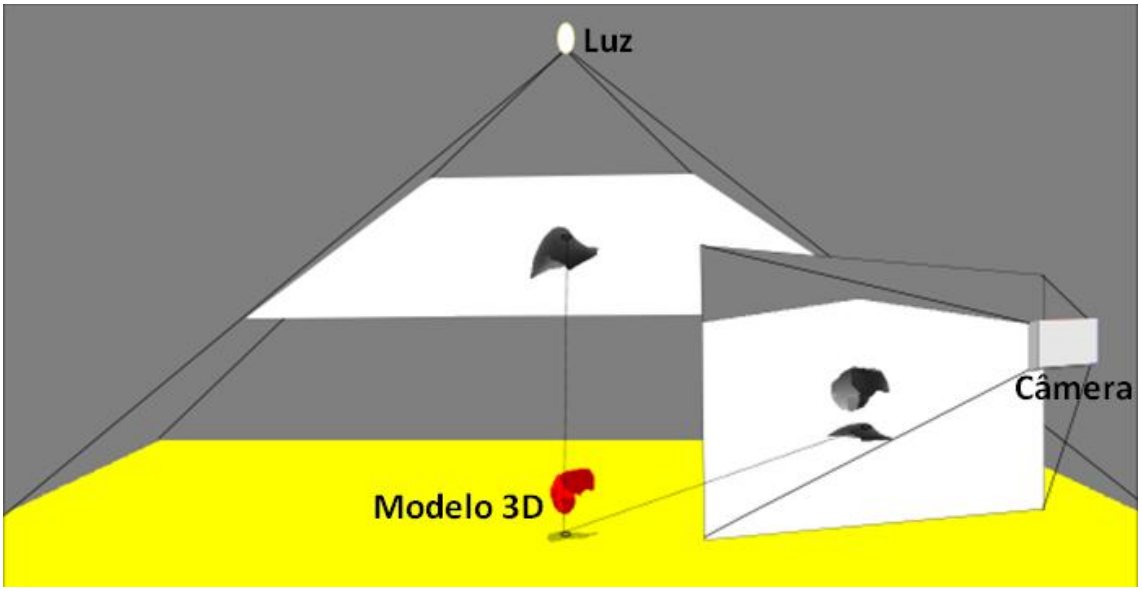

Figura 4.9 - *Shadow Mapping*: Mapeamento entre coordenadas da luz e da câmera.

A Figura 4.9 mostra o mapeamento de um fragmento visto pela luz para seu correspondente visto pela câmera, e a avaliação do mapa de profundidade que determina se o fragmento está em sombra.

4.6 Iluminação

Para fazer a iluminação local da cena, foi usado o algoritmo de *Phong Shading* [CGT]. Adicionar a implementação do algoritmo de *Phong* aos programas da segunda etapa do algoritmo de sombra é simples. Ele é bastante similar ao algoritmo de *Gouraud Shading* (*Smooth* em OpenGL), a diferença básica é que, enquanto *Gouraud* colore os vértices e interpola as cores internas, *Phong* interpola as normais e calcula a cor de cada fragmento.

Ao programa de vértices foram enviadas as normais, que passam inalteradas ao programa de fragmentos assim como a posição do vértice que será usada pelo algoritmo.

No programa de fragmentos, foram adicionados os parâmetros que controlam a iluminação, o material dos objetos da cena e a posição do observador. A cor do fragmento é dada pela soma das componentes de emissão, ambiente, especular e difusa vindas da interação das componentes do material com as componentes do ambiente e da fonte de luz.

```
//-- Definição da cor do fragmento
cor = emissao + ambiente + difusa + especular
```

A componente de emissão é definida pelas características do tipo de material que queremos desenhar, enquanto que as demais componentes são dadas por:

```
//-- Definição das contribuições das componentes da luz
ambiente = Ka * luzAmbiente
difusa = Kd * cor_fonte_luz * Max( dot( L ,  N), 0 )
especular = Ks * cor_fonte_luz * Max( dot( H, N ), 0 )^brilho
```

Sendo:

- P: posição do fragmento (não transformada pela *modelviewproj*).
- N: vetor normal do fragmento.
- L: vetor de direção do feixe de luz.
- H: vetor de direção da visualização.
- Ke: constante que define a luz emitida pelo material.
- Ka: constante que define a quantidade de luz ambiente refletida pelo material.
- Kd: constante que define a componente difusa do material.
- Ks: constante que define a componente especular do material.

5 RESULTADOS

Nessa seção são apresentados os resultados obtidos no trabalho. Inicialmente tem-se os resultados gráficos, seguidos da comparação de desempenho.

A comparação de desempenho foi dividida em duas etapas. Uma considerando os métodos numéricos sob o ponto de vista da simulação de tecidos. A outra, comparando o desempenho da solução para corpos com volume em três ambientes diferentes.

5.1 Resultados Visuais

Para facilitar a interação do usuário, foi implementada uma interface gráfica que permite a alteração dos parâmetros da simulação e da visualização. Na simulação de tecido não houve grande preocupação com a qualidade gráfica, como pode ser visto na Figura 5.1.

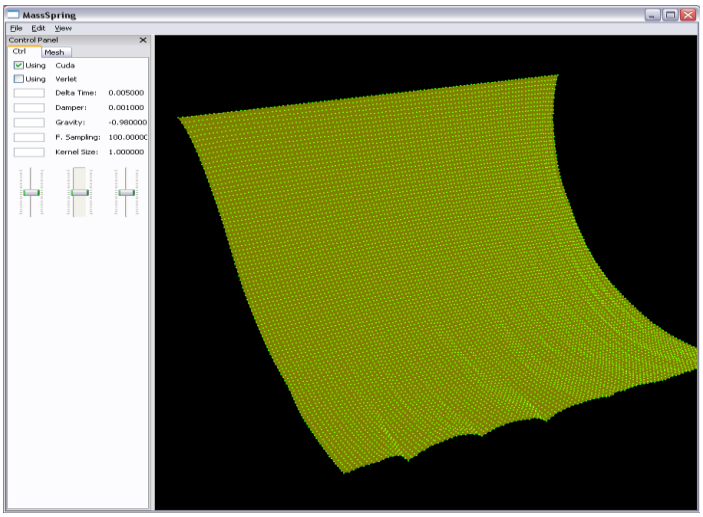

Figura 5.1 - Interface gráfica da simulação de tecido

Para a simulação de corpos com volume foram usadas técnicas para aumentar o realismo da cena, proporcionando relação de proximidade entre os objetos da cena, como mostra a Figura 5.2.

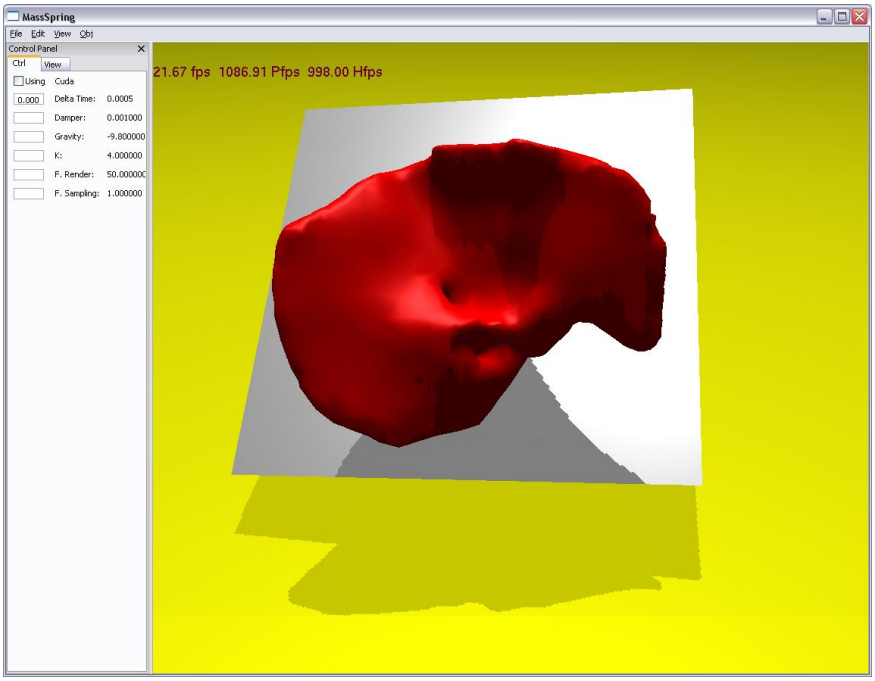

Figura 5.2 - Interface gráfica da simulação de corpos com volume

5.2 Simulação de Tecido

Como primeira parte do trabalho, foi implementada a simulação de tecidos usando um sistema massa-mola como descrito anteriormente. Os testes iniciais tiveram o propósito de indicar se o algoritmo estava evoluindo no caminho certo, e também verificar o tempo de execução dos métodos numéricos.

Ambos os métodos de integração foram implementados em duas versões, uma para executar em CPU, outra para executar em GPU. Mantidas as devidas ressalvas quanto às linguagens e técnicas específicas referentes às arquiteturas empregadas em cada caso, as implementações do algoritmo podem ser consideradas equivalentes.

Como pode ser visto nos gráficos da Figura 5.3, é insignificante a diferença de desempenho entre os métodos, uma vez que ambos usam a mesma quantidade de memória e executam operações similares.

Comparando os resultados obtidos sob o ponto de vista das arquiteturas, notamos que a taxa de iterações por segundo na GPU já é significativamente maior do que na CPU. Além disso, podemos ver que a perda de desempenho com o aumento da quantidade de partículas simuladas é menos acentuada na GPU, o que motivou o andamento da etapa envolvendo corpos com volume.

Uma implementação desse algoritmo adaptada ao uso de clusters poderia escalar melhor com o aumento no número de nós, contudo há o atraso inserido pela comunicação entre os nós, e o custo da infra-estrutura necessária é elevado. Enquanto isso, as GPUs possuem um grande número de unidades de processamento, que não são tão eficientes quanto um processador de propósito geral é para qualquer tarefa, mas sua grande quantidade faz com que problemas paralelizáveis a grão fino tenham um grande ganho de desempenho.

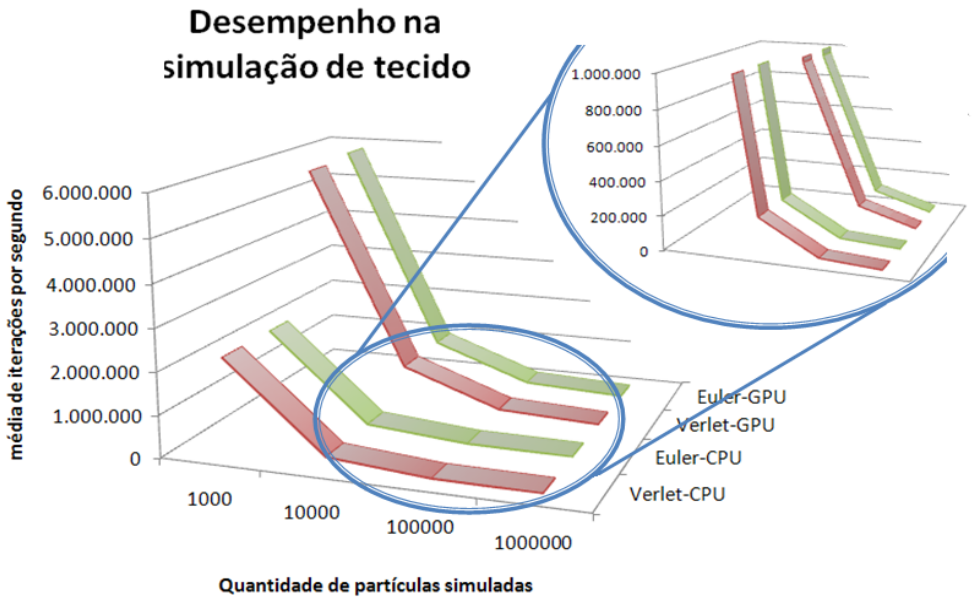

Figura 5.3 - Gráficos de desempenho dos métodos numéricos na simulação de tecido. Com destaque para a queda do desempenho com o aumento da quantidade de partículas.

5.3 Simulação de Corpos com Volume

A segunda etapa do trabalho trata de corpos com volumes e configurações mais complexas do sistema de molas. Os testes nessa etapa foram mais completos, avaliando diferentes aspectos como escalabilidade e impacto das técnicas usadas.

Para tal, foram escolhidas GPUs lançadas em dois períodos diferentes da linha GeForce, com duas unidades em cada máquina utilizada, permitindo a análise do impacto do compartilhamento da placa com o vídeo do sistema.

O parâmetro usado para fazer as comparações foi a taxa média de iterações do sistema massa-mola realizadas por segundo, durante 60 segundos de simulação. Quanto maior for essa taxa, melhor é a qualidade dos estímulos háptico e visual.

As configurações de máquina usadas nos testes foram:

- **Configuração G84**
  - **Sistema Operacional:** Win32 5.1.2600 Service Pack 3
  - **Processador:** Intel® Core™ 2 Quad Q6600 (2.4 GHz)
  - **Memória RAM:** 2.0 GB
  - **Placa gráfica:** GeForce 8600 GT (2 unidades)
  - **Memória gráfica:** 256 MB
  - **Frequência dos núcleos:** 1180 MHz
  - **Quantidade de núcleos:** 32

- **Configuração GT200**
  - **Sistema Operacional:** Win32 5.1.2600 Service Pack 3
  - **Processador:** Intel® Core™ 2 Quad Q9550 (2.8 GHz)
  - **Memória RAM:** 3.0 GB
  - **Placa gráfica:** GeForce 260 GTX (2 unidades)
  - **Memória gráfica:** 856 MB
  - **Frequência dos núcleos:** 1242 MHz
  - **Quantidade de núcleos:** 216

5.3.1 Compartilhamento

O primeiro teste visa avaliar o impacto do compartilhamento da placa gráfica entre os cálculos físicos e a saída gráfica, ou seja, quando temos apenas uma GPU. Na Figura 5.4 temos a comparação dos resultados da execução dos testes nos dois modelos de GPU com dois objetos 3D. Uma das fileiras mostra o impacto das técnicas quando usamos para os cálculos a mesma GPU que processa o vídeo do sistema, enquanto na outra fileira temos valores alcançados quando uma unidade é reservada apenas para os cálculos do sistema de simulação física.

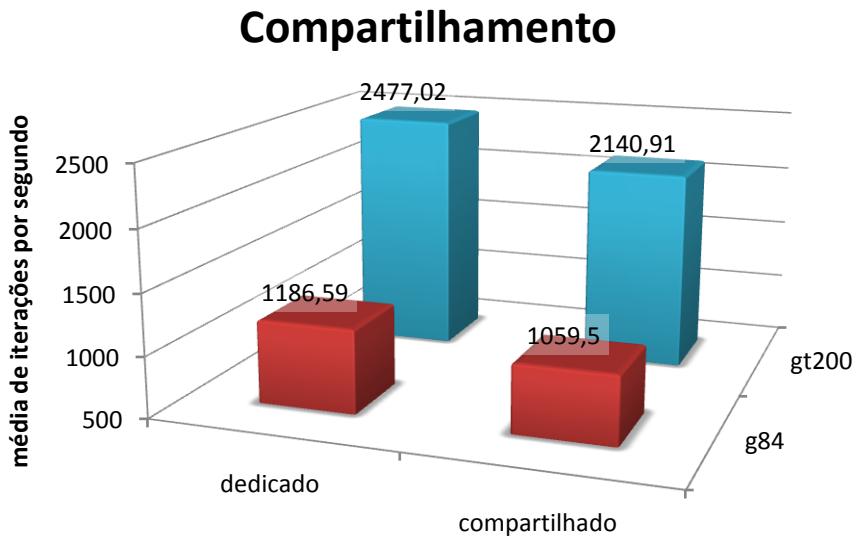

Figura 5.4 - Gráfico da comparação de desempenho quando a GPU é compartilhada e dedicada. A GPU dedicada tem melhor desempenho e a diferença é acentuada no modelo mais moderno da placa.

Uma placa gráfica, usada para iterar a simulação física enquanto também se encarrega de fazer o desenho e gerenciar a saída gráfica do sistema, tem penalidades extras como trocas de contexto e compartilhamento da memória.

Visto que o desempenho é melhor em GPUs dedicadas, os testes a seguir são feitos sob essa forma de execução.

5.3.2 Escalabilidade

O teste de escalabilidade visa verificar o desempenho quando a quantidade de partículas no sistema massa-mola cresce. A Figura 5.5 mostra o comportamento das arquiteturas para esse teste.

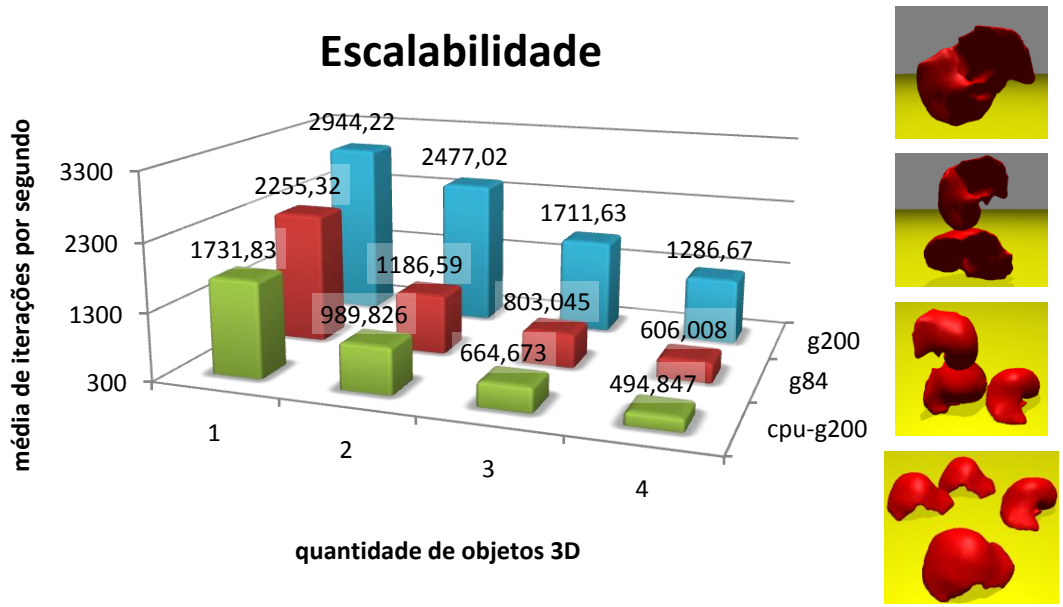

Figura 5.5 - Gráfico da comparação de desempenho quando a quantidade de partículas aumenta. A queda de desempenho é mais acentuada na CPU, enquanto a GPU mais moderna apresenta pouca perda.

Colocados lado a lado os gráficos de desempenho da CPU e dos dois modelos de GPU para a versão completa nota-se que, enquanto a CPU perde desempenho drasticamente quando aumenta a quantidade de modelos 3D no sistema, a perda apresentada pela GPU é consideravelmente menos acentuada. Isso demonstra que a grande quantidade de unidades de processamento (núcleos) presentes na placa gráfica compensa a baixa frequência de operação em cada uma delas. Comparando os dois modelos entre si, pode-se ver melhor o impacto da quantidade de núcleos no chip, uma vez que a frequência de processamento entre os núcleos dos modelos não é muito diferente.

Na Figura 5.6 têm-se os gráficos de desempenho quando se aumenta a quantidade de dados processados na GPU dedicada e na GPU compartilhada. Nota-se que, com o aumento da demanda por recursos, o modo compartilhado até mesmo da G200 apresenta maior perda de desempenho em relação ao modo dedicado da mesma placa.

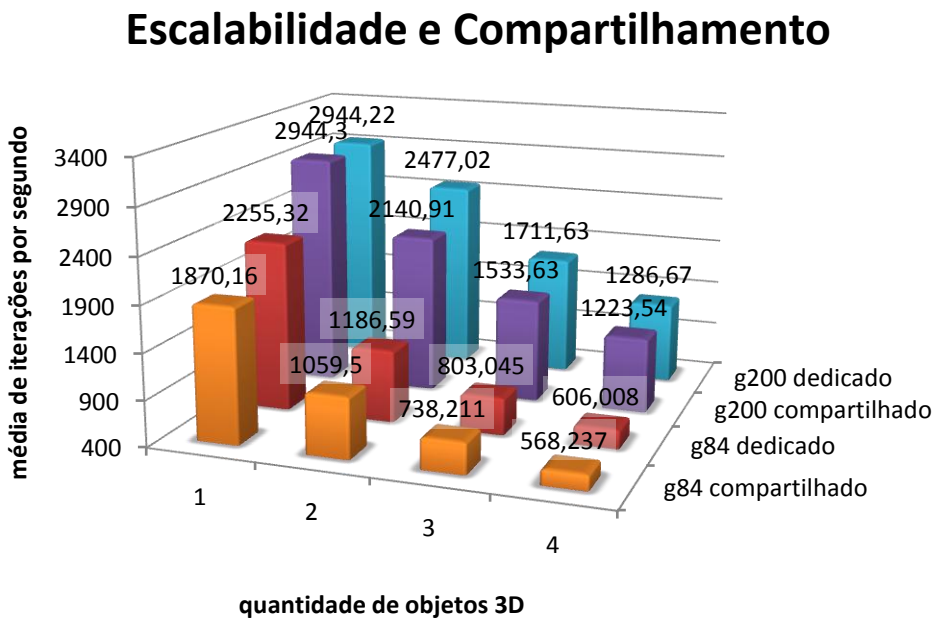

Figura 5.6 - Gráfico de comparação de desempenho em escalabilidade e compartilhamento entre os modelos de GPU. A GPU compartilhada apresenta maior queda de desempenho.

5.3.3 Interatividade

Para os testes de interatividade foi usado um sistema contendo apenas um objeto 3D. Como os objetos visam emular um sistema com mais partículas, e não têm colisão entre si, a escalabilidade das arquiteturas se mantém como mostrada anteriormente.

O gráfico apresentado na Figura 5.7 mostra a frequência média de desenho, medida a cada 2 segundos da simulação na sua versão completa, havendo interação através do dispositivo háptico.

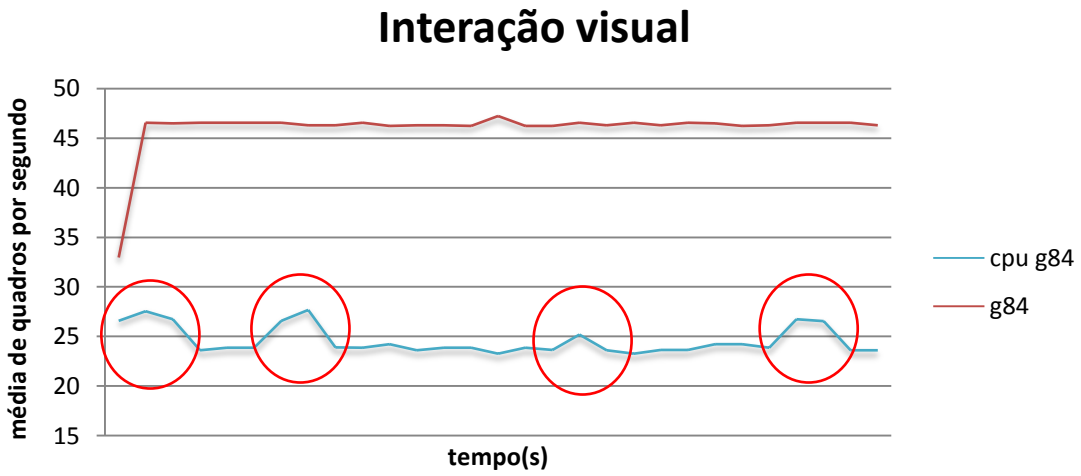

Figura 5.7 - Gráfico da oscilação do desempenho médio de quadros por segundo durante a execução. As elipses vermelhas destacam a melhora do desempenho nos períodos em que não houve contato da ferramenta com o modelo.

Tanto no retorno gráfico quanto no retorno háptico, podemos notar que a colisão com a ferramenta de interação afeta muito mais o desempenho em CPU, como podemos ver na Figura 5.8.

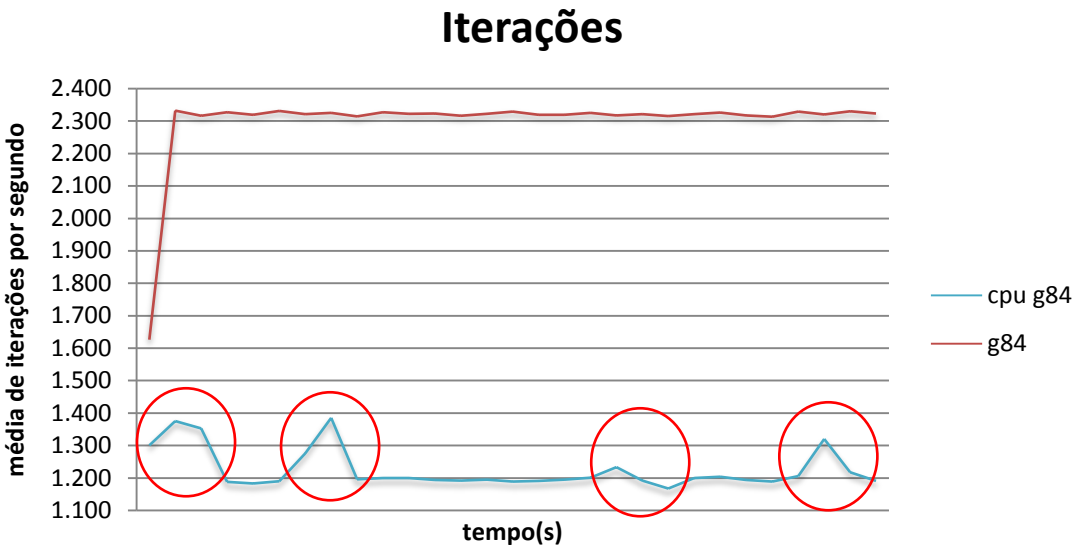

Figura 5.8 - Gráfico da oscilação de desempenho médio de iterações por segundo durante a execução. As elipses vermelhas destacam a melhora do desempenho nos períodos em que não houve contato da ferramenta com o modelo.

Os pontos de pico no gráfico da CPU nas Figuras 5.7 e 5.8 são os períodos de tempo em que não houve contato da ferramenta com o modelo 3D. Enquanto que na GPU o desempenho mantém-se estável durante a simulação, demonstrando que o impacto do cálculo da colisão em GPU é desprezível.

6 CONCLUSÕES

Esse trabalho apresentou o desenvolvimento de um experimento de simulação de corpos deformáveis com retorno háptico e a análise comparativa da sua execução com o auxílio de co-processadores gráficos. As comparações levaram em conta a disponibilidade da unidade gráfica, a escalabilidade das arquiteturas e o comportamento durante a interação.

A partir da pesquisa inicial, concluiu-se que:

- Linguagens de *Shader* são boas para melhorar o resultado visual e, apesar de poderem ser usadas para computação genérica, são dependentes de APIs e estritamente ligadas ao *pipeline* gráfico.
- A linguagem de programação CUDA é a melhor opção para programação genérica, pois independe de APIs gráficas e oferece uma abstração de mais alto nível sobre o que acontece internamente ao *pipeline* gráfico.
- O realismo introduzido na cena pelas técnicas de *Shadow Mapping* e *Phong Shading*, apesar de simples, aumenta a qualidade do retorno visual, oferecendo melhor noção de profundidade e relação entre os objetos.

A partir das simulações realizadas, pode-se concluir que:

- Unidades gráficas dedicadas, por possuírem maior disponibilidade de recursos, apresentam desempenho melhor do que unidades compartilhadas, especialmente quando a demanda por esses recursos cresce.
- Com o aumento da complexidade do modelo 3D usado, o desempenho apresentado pela GPU escala consideravelmente melhor do que o da CPU para a mesma complexidade. Isso indica que o uso do co-processador gráfico para acelerar os cálculos permite um detalhamento maior do objeto que se deseja simular.
- A qualidade da interação háptica pode ser melhorada com o auxílio das placas gráficas programáveis, que apresentaram poucas oscilações das taxas de iteração.

Como trabalhos futuros, sugere-se:

- Estudo do impacto de técnicas de manutenção de volume para contornar limitações do sistema massa-mola.
- Análise do impacto de uma estrutura de dados para detecção de colisão, a fim de suportar o aumento real da quantidade de elementos na cena, com colisão entre eles.
- Estudo do desempenho sob outras formas de deformação, envolvendo mudanças topológicas, como em simulação de corte, por exemplo.

As conclusões obtidas nesse trabalho podem ser usadas como mais uma motivação para o uso de placas gráficas em aplicações que demandam desempenho.

REFERÊNCIAS

[AMD_FUSION] AMD, **The Future is Fusion**. Disponível em: http://fusion.amd.com. Acessado em: nov. 2009.

[CG] NVIDIA, C for Graphics. Disponível em: http://developer.nvidia.com/page/cg_main.html. Acessado em: jun. 2009.

[CGT] FERNANDO, R., KILGARD, M. J., **The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.** Boston: 2006. cp. 5.

[CPP] HALFHILL, Tom R., **Parallel Processing With CUDA: Nvidia's High-Performance Computing Platform Uses Massive Multithreading**. January 28, 2008. Microprocessor Report. Reed Electronics Group

[CUDA] NVIDIA, **Compute Unified Device Architecture.** Disponível em: http://www.nvidia.com/cuda. Acessado em: jun. 2009.

[DCN06] DIETRICH, C. A., COMBA, J. L. D., NEDEL, L. P., **Storing and Accessing Topology on the GPU: A Case Study on Mass-Spring Systems.** ShaderX 5 - Advanced Rendering Techniques, 2006. Edited by Wolfgang Engel. Pages 565-578. ISBN 1-58450-499-4.

[EU] WEISSTEIN, E. W. **Euler Integral.** From MathWorld. Disponível em: http://mathworld.wolfram.com/EulerIntegral.html. Acessado em: jun. 2009.

[GF8] NVIDIA, **GeForce 8800 Architecture Technical Brief.** Disponível em: http://www.nvidia.com/object/IO_37100.html. Acessado em: nov. 2009.

[GLSL] **OpenGL Shading Language**. Disponível em: http://www.opengl.org/documentation/glsl. Acessado em: jun. 2009.

[GPGPU] HARRIS, M. et al. **General-Purpose Computation On Graphics Hardware.** Disponível em: http://www.gpgpu.org. Acessado em: jun. 2009.

[GW05] GEORGII J., WESTERMANN R.: **Mass-Spring System on the GPU**. Simulation Modelling Practice and Theory. 2005. vol.13 p.693–702.

[H05] HARRIS, M., **Mapping computational concepts to GPUs**. In ACM SIGGRAPH 2005 Courses. Los Angeles, California, July 31 – August 4, 2005.

[HLSL] **High Level Shading Language**. Disponível em: http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx. Acessado em: jun. 2009.

[JP04] JAMES D. L., PAI D. K., THALMANN D.: **BD-Tree: Output-Sensitive Collision Detection for Reduced Deformable Models.** Proc of. ACM SIGGRAPH, 2004.

[KHM98] KLOSOWSKI, J. T., HELD, M., MICHEL, J. S. B., SOWIZRAL, H., ZIKAN, K., **Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs.** IEEE Transactions on Visualization and Computer Graphics. Volume 4, Issue 1 (January 1998) Pages: 21 – 36.

[MBT07] MACIEL A., BOULIC R., THALMANN D.: **Efficient Collision Detection within Deforming Spherical Sliding Contact**. Proc. of IEEE Transactions on Visualization and Computer Graphics, vol. 13, no. 3, pp. 518-529, May/Jun, 2007.

[MHS05] MOSEGAARD J., HERBORG P., SORENSEN T. S.: **A GPU Accelerated Spring Mass System for Surgical Simulation.** 13th Medicine Meets Virtual Reality, Long Beach, USA. Studies in Health Technology and Informatics 2005; 111:342-8.

[MS] **Mass-Spring-Damper Systems The Theory.** Disponível em: http://mathinsite.bmth.ac.uk/pdf/msdtheory.pdf. Acessado em: jun. 2009.

[OLG07] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E., PURCELL, T., **A Survey of General-Purpose Computation on Graphics Hardware**. Computer Graphics Forum, volume 26, number 1, 2007, pp. 80-113.

[PG] NVIDIA, **nVidia Cuda Programming Guide.** Disponível em: http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf. Acessado em: jun. 2009.

[PO] SENSABLE, Phantom Omni. Disponível em: www.sensable.com/haptic-phantom-omni.htm. Acessado em: nov. 2009.

[RJ07] RIVERS A.R., JAMES D.L.: **Fast Lattice Shape Matching for Robust Real-Time Deformation**. Proc. of ACM SIGGRAPH, 2007.

[RK] WEISSTEIN, E. W. **Runge-Kutta Method.** From MathWorld. Disponível em: http://mathworld.wolfram.com/Runge-KuttaMethod.html. Acessado em: jun. 2009.

[PK09] PAMPLONA, V. F., KUHN, G. R., **Implementando Shadow Map.** Disponível em: http://grkuhn.googlepages.com/ShadowMap.pdf. Acessado em: nov. 2009.

[SLMS06] DE S., LIM Y., MANIVANNAN M., SRINIVASAN M. A.: **Physically Realistic Virtual Surgery using the Point-Associated Finite Field (PAFF) Approach**. Proc. of First Joint Eurohaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, 2005.

[SOG08] STEINEMANN D., OTADUY M. A., GROSS M.: **Fast adaptive shape matching deformations**. Proc. of ACM SIGGRAPH, 2008.

[TKZ04] TESCHNER, M., KIMMERLE, S., ZACHMANN, G., HEIDELBERGER B., RAGHUPATHI L., FUHRMANN, A., CANI, M. P., FAURE, F., THALMANN N., STRASSER. W., **Collision Detection for Deformable Objects.** Eurographics State-of-the-Art Report (EG-STAR), page 119--139 – 2004.

[VE] ERCOLESSI F. **Verlet Integration.** Disponível em: http://www.fisica.uniud.it/~ercolessi/md/md/node21.html. Acessado em: jun. 2009.

ANEXO A - SHADERS

```
//-- Programa de vértices para o rendering do ponto de vista da luz
void main(  float4      i_position        : POSITION,
            float4      i_color           : COLOR,
            out float4  o_color           : COLOR,
            out float4  o_position_light  : TEXCOORD0,
            out float4  o_position        : POSITION,
            const uniform float4x4 modelViewProj
)
{
      o_position        = mul(modelViewProj, i_position);
      o_position_light  = mul(modelViewProj, i_position);

      o_color = i_color;
}

//-- Programa de fragmentos para o rendering do ponto de vista da luz
void main(  float4      i_color           : COLOR,
            float4      i_position_light  : TEXCOORD0,
            out float4  o_color           : COLOR)
{
      //-- clip
      i_position_light = i_position_light/i_position_light.w;
      //-- Coordenadas em 0-1
      o_color.xyzw = (i_position_light.z + 1.0) * 0.5;
}

//-- Programa de vértices para o rendering do ponto de vista da câmera
void main(  float4      i_position        : POSITION,
            float4      i_color           : COLOR,
            float3      i_normal          : NORMAL,
            out float4  o_color           : COLOR,
            out float4  o_position        : POSITION,
            out float4  o_light_position  : TEXCOORD0,
            out float3  o_normal          : TEXCOORD1,
            out float3  o_objectPos       : TEXCOORD2,
            const uniform float4x4 modelViewProj,
            const uniform float4x4 modelView,
            const uniform float4x4 modelViewProjLight
)
{
      o_color     = i_color;
      o_position  = mul(modelViewProj, i_position);
      o_light_position  = mul(modelViewProjLight, i_position);

      o_normal = i_normal;
      o_objectPos = i_position.xyz;
}
```

```
//--  Programa de fragmentos para o rendering do ponto de vista da
      câmera
void main(  float4      i_color           : COLOR,
            float4      i_light_position  : TEXCOORD0,
            float3      i_normal          : TEXCOORD1,
            float3      i_position        : TEXCOORD2,
            out float4  o_color           : COLOR,
            const uniform sampler2D shadowMap,
            uniform float3 globalAmbient,
            uniform float3 lightColor,
            uniform float3 lightPosition,
            uniform float3 eyePosition,
            uniform float3 Ke,
            uniform float3 Ka,
            uniform float3 Kd,
            uniform float3 Ks,
            uniform float  shininess)
{
//-- SHADOW MAPPING
      //-- clip and tarnsform to 0-1
      i_light_position = (i_light_position/i_light_position.w + 1.0)
                         * 0.5;
      //-- find where shadow is
      float4 map = tex2D (shadowMap, i_light_position.xy);
      float depth = map.w;
      float obj_test = abs(map.g - i_color.g) < 0.05;
      float sm_test = depth < (i_light_position.z - 0.01);
      float4 sm_color;
      sm_color.xyz = i_color.xyz - (0.3 * sm_test);

//-- PHONG ILUMINATION MODEL
      float3 P = i_position.xyz;
      float3 N = normalize(i_normal);
      //-- Compute emissive term
      float3 emissive = Ke;
      //-- Compute ambient term
      float3 ambient = Ka * globalAmbient;
      //-- Compute the diffuse term
      float3 L = normalize(lightPosition - P);
      float diffuseLight = max(dot(L, N), 0.0);
      float3 diffuse = Kd * lightColor * diffuseLight;
      //-- Compute the specular term
      float3 V = normalize(eyePosition - P);
      float3 H = normalize(L + V);
      float specularLight = pow(max(dot(H, N), 0.0), shininess);
      float difTest = (diffuseLight > 0.0);
      specularLight *= difTest;
      float3 specular = Ks * lightColor * specularLight;
      //-- put it all together
      float4 p_color;
      p_color.xyz = emissive + ambient + diffuse
                  + specular * (1.0 - sm_test );
      //-- mix colors
      o_color.xyz = p_color.xyz * 0.7 + sm_color.xyz*0.3;
      o_color.w = i_color.w;
}
```