ANIMAÇÃO BASEADA EM FÍSICA USANDO PARTÍCULAS EM GPUS

Yalmar Temistocles Ponce Atencio

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientador: Claudio Esperança

Rio de Janeiro
Setembro de 2011

ANIMAÇÃO BASEADA EM FÍSICA USANDO PARTÍCULAS EM GPUS

Yalmar Temistocles Ponce Atencio

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Claudio Esperança, Ph.D.

Prof. Paulo Roma Cavalcanti, D.Sc.

Prof. Ricardo Guerra Marroquim, D.Sc.

Prof. Waldemar Celes Filho, D.Sc.

Prof. Esteban Walter Gonzalez Clua, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2011

Ponce Atencio, Yalmar Temistocles

Animação Baseada em Física usando Partículas em GPUs/Yalmar Temistocles Ponce Atencio. – Rio de Janeiro: UFRJ/COPPE, 2011.

XVI, 139 p.: il.; 29, 7cm.

Orientador: Claudio Esperança

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2011.

Referências Bibliográficas: p. 127 – 139.

1. Computação gráfica. 2. Simulação física de fluidos e sólidos usando partículas. 3. Processamento paralelo em GPUs. I. Esperança, Claudio. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

A toda minha família pelo apoio
e compreensão.

Agradecimentos

Gostaria de agradecer às pessoas que integram o Laboratório de Computação Gráfica (LCG) da COPPE/UFRJ. Aos professores Claudio Esperança, Antônio Oliveira, Paulo Roma e Ricardo Marroquim. Pelo apoio que me foi fornecido, conselhos e dúvidas sanadas. Em especial aos professores Claudio Esperança, meu orientador, por ter me ajudado em cada trapalhada que fiz durante o doutorado, e Antônio Oliveira por ter me ajudado a compreender diversos conceitos de matemática e física.

A minha esposa Zuria e minha filha Nicole pelo amor e apoio desinteressado durante este tempo.

Agradeço também aos meus pais e irmãos pelo apoio e compreensão para a conclusão desta tese.

Aos amigos do mestrado e doutorado: Alvaro, Saulo, Flávio, Felipe Moura e tantos outros; pelas conversas e troca de idéias.

Às secretárias do programa de Engenharia de Sistemas e Computação, pela ajuda nos documentos que precisei durante os cursos de mestrado e doutorado.

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), o Ministério das Relações Exteriores (MRE) por intermédio da Divisão de Temas Educacionais (DCE) e o Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) que tornaram possível minha seleção no Programa de Estudante-Convênio de Pós-Graduação (PEC-PG), suporte financeiro que recebi durante o doutorado.

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

ANIMAÇÃO BASEADA EM FÍSICA USANDO PARTÍCULAS EM GPUS

Yalmar Temistocles Ponce Atencio

Setembro/2011

Orientador: Claudio Esperança

Programa: Engenharia de Sistemas e Computação

Apresenta-se, nesta tese, abordagens para simulação baseada em física. Inicialmente, abordagens sequencias são apresentadas, entretanto, devido ás limitações em poder de processamento de CPUs atuais é possível simular apenas algumas dezenas de objetos em tempo real. Ainda utilizando múltiplos núcleos de processamento (*multi-core*), os resultados serão incrementados aproximadamente em 5-10 vezes, sendo assim possível simular algumas centenas de objetos em tempo real. Estas limitações, sugerem o emprego de arquiteturas que permitam processamento paralelo massivo. Este é o caso das GPUs atuais, pelo que um segundo grupo de abordagens; que tratam objetos sólidos (rígidos e deformáveis), líquidos (SPH), elásticos (tecidos) e a interação entre eles são implementados empregando esta plataforma. Nestas abordagens, que chamamos de abordagens GPGPU, todas ou algumas etapas foram realizadas na GPU aproveitando seu poder computacional. Este aproveitamento é possível, principalmente, pelo emprego de partículas para representar objetos, ou seja, objetos são representados por conjuntos de partículas. Esta representação permite utilizar o mesmo sistema de detecção de colisão (sistema unificado) para simular a dinâmica e interação de objetos de diferentes classes. Assim, é possível simular cenários com grandes quantidades de objetos, e de diferentes classes, em tempo real. O sistema de detecção de colisão unificado é linear no número de partículas, e a detecção de colisão entre partículas (esferas) é rápida, levando a obter ganhos consideráveis comparado com abordagens sequencias que usam malhas. Quando duas partículas colidem, forças de repulsão ou impulsos são computados, que posteriormente são aplicados aos objetos aos quais estas partículas pertencem. Para a renderização, objetos sólidos guardam referências para malhas trianguladas, que serão visualizadas na posição e orientação correspondente a cada instante de tempo. Já no caso de líquidos, é usada uma abordagem baseada no espaço-imagem

que permite aproximar a superfície do liquido sem necessidade de usar algoritmos complexos, como por exemplo, os conhecidos algoritmos de marcha.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

PHYSICALLY BASED ANIMATION USING PARTICLES ON GPUS

Yalmar Temistocles Ponce Atencio

September/2011

Advisor: Claudio Esperança

Department: Systems Engineering and Computer Science

In this work, we present some approaches for physically based simulation. Firstly, two sequential approaches are presented, however, due to limited computational power on current CPUs, only few dozens of objects can be simulated in real time. Still using multi-core processors, will be possible to simulate only some hundred of objects in real time. These limitations, suggest the use of massive parallel processing architectures. This is the case of current GPUs, so a second group of approaches, that allow the simulation of solid objects (rigid and deformable), liquids (SPH), elastics (cloth) and the interaction between them was implemented using that architecture. In these approaches, to which we called GPGPU approaches, all or some steps are done on the GPUs exploiting their computational power. This use is possible, mainly, due to use of particles to represent objects. This representation allows use the same collision detection system (called, unified collision detection) to simulate the dynamic and interaction between different kind of objects. Then, is possible to simulate scenes with hundreds and thousands of objects in real time. The unified collision detection is linear to number of particles, and collision detection between particles (spheres) is simple and fast, leading to high increase in performance compared with sequential approaches based on meshes. When two particles collide, forces or impulses are computed, which are applied to objects which belong them. For rendering, solid object have references to pre-loaded meshes, who are rendered on their corresponding position and orientations for each time step. On the liquid case, an image-space approach is used, which allows approximate the liquid surface without use complex algorithms, for instance, the well known marching techniques.

Sumário

Lista de Figuras **xii**

Lista de Tabelas **xvi**

1 Introdução **1**

2 Revisão bibliográfica **6**

2.1 Computação paralela e a unidade de processamento gráfico 6

2.1.1 Computação paralela 7

2.1.2 Computação de propósito geral em GPUs 8

2.2 Métodos de integração 13

2.2.1 Métodos de integração para o movimento 16

2.3 Detecção de colisão 16

2.3.1 Detecção de colisão grosseira 19

2.3.2 Detecção de colisão exata 20

2.4 Simulação baseada em física 26

2.4.1 Simulação baseada em física fazendo uso de GPUs 27

3 Duas abordagens para simulação baseada em física **28**

3.1 Simulação de objetos rígidos 28

3.1.1 Representação do objeto 29

3.1.2 Detecção de interferência 29

3.1.3 Integração 30

3.1.4 Tratamento de colisões 31

3.1.5 Tratamento de contatos 32

3.1.6 Grafo de contato 33

3.1.7 Propagação de choque 34

3.1.8 Separação dos objetos 34

3.2 Simulação de objetos deformáveis 35

3.2.1 Representação dos objetos 35

3.2.2 Pré-processamento 35

3.2.3 Detecção de colisões 36

3.2.4 Resposta às colisões . . . 37
3.2.5 Região de deformação . . . 40
3.2.6 Dinâmica baseada em casamento de formas . . . 42
3.2.7 Método de integração . . . 44
3.2.8 Deformação linear . . . 44
3.2.9 Deformação quadrática . . . 45
3.2.10 Extensões . . . 46

4 Detecção de Colisões em GPU 49
4.1 Detecção de colisão grosseira baseada em subdivisão do espaço . . . 49
4.1.1 Subdivisão espacial . . . 50
4.1.2 Subdivisão espacial paralela . . . 51
4.1.3 Implementação da subdivisão espacial paralela em CUDA . . . 53
4.1.4 Discussão . . . 57
4.2 Detecção de colisão grosseira baseado no esquema de varredura e poda 57

5 Simulação baseada em física usando partículas e a GPU 59
5.1 Detecção de colisão baseada em partículas . . . 60
5.2 Simulação de objetos rígidos . . . 61
5.2.1 Movimento de um corpo rígido . . . 61
5.2.2 Representação da forma dos objetos . . . 63
5.2.3 Detecção de colisão . . . 65
5.2.4 Resposta às colisões . . . 66
5.2.5 Implementação da simulação na GPU . . . 66
5.2.6 Simulação de objetos rígidos usando impulsos . . . 73
5.3 Simulação de objetos deformáveis . . . 78
5.3.1 Representação da forma dos objetos . . . 78
5.3.2 Detecção e resposta de colisão . . . 79
5.3.3 Atualização da forma dos objetos . . . 79
5.4 Hidrodinâmica suavizada com partículas em GPUs . . . 79
5.4.1 Equações para a dinâmica de fluidos . . . 80
5.4.2 Discretização do espaço ocupado pelo fluido . . . 80
5.4.3 Condição de Fronteira . . . 82
5.4.4 Implementação na GPU . . . 83
5.4.5 Atualização da Posição . . . 85
5.5 Interação de líquido e tecido . . . 86
5.5.1 Simulação de fluidos . . . 86
5.5.2 Simulação de tecido . . . 86
5.5.3 Uso da grade 3D . . . 87
5.5.4 Detecção de colisão . . . 88

5.5.5 Reação do líquido 89
5.5.6 Reação do tecido 90
5.5.7 Estruturas de dados utilizadas 91
5.6 Interação entre líquido e sólidos 91

6 Renderização da superfície de líquidos 93
6.1 Renderização usando *marching cubes* 93
6.2 Renderização de superfície usando o espaço-imagem 96
6.2.1 Profundidade da superfície do líquido 96
6.2.2 Suavização da textura de profundidade na superfície do líquido 97
6.2.3 Cálculo da espessura do líquido visível 98
6.2.4 Renderização do cenário 99

7 Resultados 100
7.1 Simulação de objetos sólidos em CPU 103
7.1.1 Simulação de objetos rígidos 103
7.1.2 Simulação de objetos deformáveis 108
7.2 Abordagens usando sistemas de partículas e GPUs 112
7.2.1 Esquema de detecção de colisões grosseiro 114
7.2.2 Simulação de objetos rígidos 115
7.2.3 Simulação de objetos deformáveis usando casamento de formas 120
7.2.4 Simulação de líquidos usando Hidrodinâmica suavizada com partículas 121
7.2.5 Simulação de objetos elásticos 122
7.2.6 Acoplamento entre objetos diferentes 123
7.3 Publicações e contribuições 124

8 Conclusões 125
8.1 Trabalhos futuros 126

Referências Bibliográficas 127

Lista de Figuras

2.1 Pipeline gráfico com funcionalidade fixa. 9
2.2 Pipeline gráfico programável. 9
2.3 Arquitetura do modelo unificado da NVIDIA. 10
2.4 Grade de blocos em CUDA. 11
2.5 Arquitetura do modelo de programação do CUDA. 12
2.6 método de detecção de colisão sweep and prune em 2D. 20
2.7 Hierarquia de caixas limitantes alinhadas com os eixos (AABB-Tree). 21
2.8 Hierarquia de esferas limitantes (Sphere-Tree). 21
2.9 Hierarquia de caixas limitantes orientadas (OBB-Tree). 22
2.10 Extração de camadas com faces visíveis desde a posição do observador. 24
2.11 Texturas guardam informação de colisão. 25
2.12 Encontrando pontos de colisão. 25

3.1 Contatos entre objetos empilhados. 29
3.2 Interferência entre objetos com arestas não proporcionais. 30
3.3 Exemplo de objetos empilhados. 33
3.4 Grafo de contato para cenas com objetos empilhados. 33
3.5 contatos entre objetos empilhados. 34
3.6 A verificação de interferência na etapa de filtragem grosseira coleta vértices dentro das esferas envolventes (a); a detecção de colisão exata é feita apenas nas células em colisão potencial (b). 37
3.7 Vértices borda (esquerda) e vértices internos (direita). 38
3.8 Pontos de contato com suas normais nas faces intersectadas. 39
3.9 Direções de penetração para os vértices colididos. 40
3.10 Tratamento de colisões assimétricas. 41
3.11 Determinação da superfície de contato. 42
3.12 (a) dois objetos em colisão, (b) depois uma superfície de contato é determinada e (c) deformada com o casamento de formas. 43
3.13 Tipos de deformação (imagem extraída de [1]). 46
3.14 Identificando células de borda (direita) do conjunto de células em colisão potencial (esquerda) para objetos em colisão. 47

3.15 Função distância para o modelo toro definida na caixa limitante. Linhas vermelhas definem uma distância negativa (fora do modelo) e linhas azuis definem uma distância positiva (dentro do modelo). . . . 48

4.1 Uma esfera envolvente intersectando até oito possíveis células. 50
4.2 Um exemplo de subdivisão espacial 2D para quatro objetos. 51
4.3 Separação de colisões potenciais em listas que correspondem ao tipo de célula. 52
4.4 Subdivisão espacial 3D por tipo de célula. 52
4.5 Volumes envolventes (esferas) contidos numa grade 3D de $3 \times 3 \times 3$. . 53
4.6 O conjunto de objetos da figura 4.5 é projetado nos planos YZ (esquerda), ZX (meio) e XY (direita). 54
4.7 Resultado da construção do array de identificadores de célula para o conjunto de objetos da figura 4.5. 55
4.8 O algoritmo Radix Sort aplicado ao array de identificadores de célula. 56
4.9 Inspecionando mudanças na lista ordenada de identificadores de células para localizar células em colisão. 56
4.10 Erros do esquema de subdivisão. 57

5.1 Mapeamento de partículas na grade. 61
5.2 Translação de um objeto rígido (esquerda) e rotação de um corpo rígido (direita). 61
5.3 Conjunto de partículas que representam um corpo rígido. 64
5.4 Partículas que representam um objeto: obtidas por voxelização (esquerda) e por posicionamento na superfície (direita). 65
5.5 Diferentes resoluções de partículas para representar um sólido (imagem 29-3 extraída de [2]). 65
5.6 Fluxograma da simulação de corpos rígidos na GPU 67
5.7 Construção e ordenação de arrays de índices de células e índices de objetos para um conjunto de partículas em 2D. 69
5.8 Resolvendo colisão entre objetos representados por partículas usando forças elásticas. 74
5.9 Resolvendo colisão partícula-partícula (esquerda) e colisão partícula-obstáculo (direita). 75
5.10 Corpo rígido representado por partículas, de tamanhos iguais (meio) e de tamanhos diferentes (direita). 76
5.11 Aproximação de um ponto de colisão na superfície da esfera. 77
5.12 Conjunto de partículas posicionadas nos vértices de uma malha. . . . 79
5.13 Distribuição uniforme de partículas dentro/atrás da superfície da fronteira (imagem extraída de [3]). 83

5.14 Fluxograma para uma iteração de simulação SPH. 84
5.15 Função de distância e densidade para a superfície da fronteira. 85
5.16 Molas que conectam partículas vizinhas para a modelagem de tecido (imagem extraída de [3]). 87
5.17 Duas grades são utilizadas para a interação entre líquido e tecido (imagem extraída de [3]). 88
5.18 Campo de força no tecido: descontínua (esquerda) e contínua (direita). As imagens foram extraídas de [3]. 89

6.1 Superfície de partículas de fluido utilizando a técnica *metaballs*. . . . 95
6.2 Superfície gerada usando a técnica marching cubes para um conjunto de 10k partículas. 95
6.3 Tempos gastos em cada etapa na geração de superfície com a técnica de marcha de cubos para 10k partículas (ver figura 6.2). 96
6.4 Computando a textura de profundidade das partículas visíveis. 97
6.5 Suavização do buffer de profundidade das partículas visíveis. 97
6.6 Transparência do líquido. 99

7.1 Um cenário simples envolvendo quatro objetos. 103
7.2 Simulando um cenário com 30 caixas empilhadas. 104
7.3 Simulação de um dominó com 300 peças. 104
7.4 Cenário com objetos complexos: vista frontal(esquerda) e vista superior(direita). 105
7.5 Tempo gasto em cada etapa da simulação para o experimento de empilhamento de caixas ($S1_RBI_{CPU}$). 105
7.6 Quadros por segundo para o experimento de empilhamento de caixas ($S1_RBI_{CPU}$). 106
7.7 Tempo gasto em cada etapa da simulação de 300 peças de dominó. . . 106
7.8 Quadros por segundo para a simulação de 300 peças de dominó. . . . 106
7.9 Tempo gasto em cada etapa da simulação para o cenário empilhamento de coelhos. 107
7.10 Quadros por segundo para a simulação de empilhamento de coelhos. . 107
7.11 simulação de uma cena com objetos diferentes (veja $S1_TDB_{CPU}$ na tabela 7.6). 110
7.12 Tempo gasto para cada sub-processo em cada passo de tempo. 110
7.13 Quantidade de primitivas em colisão (vértices em colisão potencial, faces, tetraedros e vértices em colisão real) para cada passo de tempo. 111
7.14 Experimentos com 8 (a), 18 (b) e 27 (c) esferas. 111
7.15 Simulação de 8 toros deformáveis com empilhamento. 112

7.16 Tempo gasto (em segundos) no decorrer da simulação com 8 toros utilizando os protótipos $S1_DB_{CPU}$, $S2_DB_{CPU}$ e $S3_DB_{CPU}$. . . . 112
7.17 Dois quadros do experimento S_BD_{GPU}. 114
7.18 Quadros por segundo para o experimento S_BP_{GPU} que é testado no $SistemaB$, $SistemaC$ e $SistemaD$. 114
7.19 Simulação de 693 objetos complexos (*Stanford bunny*). 115
7.20 Comportamento elástico quando os objetos batem com as paredes do cenário. 115
7.21 Tempo gasto em cada etapa de simulação para o experimento S_RBF_{GPU}. 116
7.22 Tempo de simulação vs. tempo de renderização. 116
7.23 Fps para obtidos em três GPUs diferentes. 117
7.24 Simulação de objetos grandes: visualização com partículas (esquerda), visualização com partículas e malhas (centro) e visualização apenas com malhas (direita). 117
7.25 Partículas interpenetradas gerando forças opostas. 118
7.26 Representação, com partículas de tamanho variable, para dois modelos *Stanford bunny*: pequeno (esquerda), grande (centro) e relação de tamanhos (direita). 118
7.27 Simulação de coelhos: chuva de 2904 coelhos pequenos(esquerda), 500 coelhos pequenos caem sobre 6 coelhos grandes(direita). 119
7.28 Tempo gasto em cada etapa de simulação para o experimento $S1_RBI_{GPU}$. 119
7.29 Dois quadros de simulação. Interpenetração entre objetos e obstáculos(paredes) não mais se verifica. 120
7.30 Simulação de 180 objetos deformáveis. 121
7.31 Tempo gasto em cada etapa de simulação para o experimento S_DB_{GPU}. 121
7.32 Tempo de simulação vs. tempo de renderização. 121
7.33 Simulação de líquido SPH com 100000 partículas. 122
7.34 Efeito de cáustica gerado por um tanque de agua. 122
7.35 Simulação de tecidos colidindo. 123
7.36 Bolas de líquido caindo num retalho de tecido pendurado pelas pontas. 123
7.37 Interação de objetos sólidos e líquido. 124

Lista de Tabelas

7.1 Configurações computacionais (PCs) empregadas para testar os protótipos implementados 101
7.2 Protótipos que empregam apenas CPU 102
7.3 Protótipos implementados que empregam CPU e GPU usando sistemas de partículas 102
7.4 Experimentos realizados no protótipo RBI_{CPU} 103
7.5 Tempo gasto, em *segundos*, para cada etapa da simulação do cenário com 30 *Stanford bunnies* (figura 7.3) 108
7.6 Experimentos realizados nos protótipos para simulação de objetos deformáveis 109
7.7 objetos com diferente resolução. 109
7.8 primitivas para experimentos com esferas. 111
7.9 Simulações realizadas para os protótipos usando sistemas de partículas e GPUs 113

Capítulo 1

Introdução

Animação baseada em física (*Physically Based Animation*) é um processo computacional que permite obter uma animação contínua da interação de objetos com plausibilidade física. Diversas abordagens têm sido sugeridas para lidar com diversos aspectos do mundo real, tais como simulação de sistemas de partículas, simulação de objetos rígidos, simulação de objetos deformáveis, simulação de fluidos (líquidos, fogo, fumaça, etc.) e o acoplamento destas. Estas abordagens são utilizadas em diversas aplicações, tais como: jogos interativos, simulações cirúrgicas, robótica, indústria do cinema, indústria automotiva, entre outras. Um dos aspectos mais estudados, quando se quer simular a interação física entre objetos, é o fato decorrente de que dois ou mais objetos não podem ocupar o mesmo lugar no mesmo instante de tempo. Durante a animação física estas situações devem ser detectadas e resolvidas. O processo para tratar estas situações, conhecido como *detecção de colisão*, não é um problema trivial, e tipicamente gasta a maior parte do esforço computacional realizado a cada instante de tempo durante a simulação.

Há um esforço constante da comunidade de pesquisadores para melhorar o tempo gasto na detecção de colisão e tornar as aplicações mais rápidas. Nos últimos anos, com a evolução das unidades de processamento gráfico (GPUs), novas técnicas de detecção de colisão foram apresentadas [4–10], explorando o *espaço-imagem*, contrastando com as técnicas tradicionais [11–15] que usam o espaço do objeto.

As técnicas que usam o espaço-imagem têm proporcionado melhor desempenho do que as técnicas tradicionais, devido ao alto poder de computação que as GPUs possuem. Entretanto, o processamento em espaço-imagem nas GPUs é limitado quanto ao tamanho e precisão numérica.

Além das técnicas que usam o espaço-imagem, técnicas baseadas em partículas têm se tornado bastante populares nos últimos anos [2, 3, 16, 17]. A base destas técnicas é a representação de objetos por conjuntos de partículas e a detecção de colisão simplesmente se resume a detectar interferência entre partículas, ou seja,

pequenas esferas. Estas técnicas se apóiam no uso de uma **grade 3D**[1] que permite a detecção de colisão entre partículas vizinhas. A grade é uma parte fundamental do método, já que permite uma detecção de colisão rápida, baseada no princípio de que uma partícula não pode colidir com outras além daquelas residentes na mesma célula(*voxel*) ou em células adjacentes.

A principal contribuição desta tese é a implementação de vários protótipos *completos* para simulação física. Para cada protótipo foram empregadas técnicas que se adaptam bem ao propósito estabelecido, e algumas foram modificadas para conseguir resultados visualmente aceitáveis. Inicialmente, dois protótipos utilizando estruturas de dados sequenciais foram implementados. Entretanto, considerando as limitações de hardware (CPU) atual, estas implementações conseguem simular apenas algumas dezenas de objetos em tempo real, pelo que foi necessário a procura por métodos e técnicas mais eficientes. Por outro lado, pelo nível de paralelismo que as GPUs fornecem, aplicações implementadas nesta arquitetura são dezenas de vezes mais rápidas do que seus equivalentes implementados para CPUs. Assim, foi implementado um segundo grupo de protótipos para simulação de objetos sólidos, elásticos, líquidos e a interação destes. Em alguns casos todos os processos da simulação são executados na GPU, já em outros casos é necessário o emprego da CPU para realizar cálculos adicionais. Uma característica comum deste grupo de protótipos é o emprego de partículas/esferas para representar os objetos. Assim, um mesmo sistema de detecção de colisão é usado para a interação de objetos de tipos diferentes. A seguir descreve-se de modo sucinto o conteúdo de cada capítulo.

O capítulo 2 apresenta conceitos e principais trabalhos relacionados com detecção de colisão, simulação baseada em física, métodos numéricos frequentemente empregados e conceitos de programação em GPUs.

No capítulo 3 são descritos dois protótipos para simulação de objetos sólidos. O primeiro, trata a simulação de corpos rígidos segundo, principalmente, a abordagem apresentada por Guendelman et al. [18]. Já o segundo, trata a simulação de objetos deformáveis, onde o método apresentado combina várias técnicas, a saber:

- objetos representados por malhas tedraedrais;
- detecção de colisão grosseira baseada na verificação de esferas envolventes para filtrar pares de objetos em colisão potencial;
- detecção de colisão exata empregando uma tabela de dispersão para mapear tetraedros [19];
- resposta a colisões computando a distância e direção de penetração [20] de todos os vértices colididos;

[1] Estrutura de dados usada para subdividir o espaço em células regulares.

- colisões assimétricas usando a técnica de *tratamento de contatos por projeção* de Jakobsen [21];
- a separação de objetos realizada usando as informações de colisão nos vértices colididos e um esquema de busca binária para levar estes vértices a uma superfície de contato;
- dinâmica baseada na técnica de casamento de formas de Müller [1];
- integração empregando um método semi-implícito de Euler [1].

O capítulo 4 aborda algoritmos e estruturas de dados para GPUs, descrevendo um esquema simplificado para detecção de colisão grosseiro baseado no trabalho apresentado por [22]. Nesta implementação percebemos que o método descrito em [22] não detecta todas as colisões, pelo que foi necessário reimplementá-lo fazendo as correções e adaptações necessárias. O alvo deste protótipo é detectar colisões em dezenas de milhares de volumes envolventes de tamanho variável em tempo real.

O capítulo 5 descreve a implementação de sete protótipos em GPU para simulação física de objetos (sólidos e líquido) representados por partículas. Cabe destacar que partículas são adequadas para representar objetos, dependendo da resolução, e por outro lado, a detecção de colisão entre partículas é simples é rápida. Como partículas são independentes entre si, o processamento de colisões pode ser realizado em paralelo de forma eficiente aproveitando o potencial das GPUs. De forma sumária esse capítulo aborda:

1. Um protótipo que trata a simulação de corpos rígidos implementado segundo a abordagem apresentada em [2]. Os objetos são representados por partículas de tamanho igual. Esta abordagem, no entanto, não é apropriada para simular corpos rígidos de geometria complexa, uma vez que as partículas que irão representar um objeto são obtidas utilizando um esquema de voxelização, sendo que, se se quer representar objetos com detalhe, será necessário gerar uma voxelização de resolução fina, levando a um gasto excessivo em número de partículas e por outro lado, caso se decida por rapidez, uma resolução grossa da voxelização não aproximará adequadamente o objeto, levando a comportamentos não desejados quando há colisão entre corpos rígidos.

2. Com o propósito de melhorar as deficiências da abordagem anterior, um segundo protótipo para simulação de corpos rígidos foi implementado. Esta versão segue o modelo físico baseado em impulsos apresentado em [18] e para melhorar a dinâmica e o desempenho, os objetos são representados por grupos reduzidos de partículas de tamanho variável, permitindo assim aproximar melhor suas formas. Esferas maiores são empregadas para preencher partes de

maior volume e esferas pequenas para partes delgadas. Ao se detectar colisão entre duas esferas, impulsos são aplicados em pontos localizados na superfície das esferas.

3. Um terceiro protótipo, que trata a simulação de corpos deformáveis, foi implementado segundo a abordagem apresentada em [1]. Entretanto, ao invés de usar nuvens de pontos, conjuntos de partículas são empregados para representar objetos, tratar colisões e computar deformações dos objetos a cada instante de tempo. Para a visualização, as deformações computadas são aplicadas a malhas de triângulos associadas aos objetos.

4. Aproveitando a técnica para detecção de colisões entre partículas, um protótipo para simulação de líquidos usando *Hidrodinâmica suavizada de partículas* (SPH) foi implementado. Este protótipo segue a abordagem apresentada em [17], entretanto, para evitar incompressibilidade foram empregados operadores propostos em [23].

5. Um quinto protótipo capaz de simular tecidos segundo a abordagem apresentada em [21] foi implementado. A contribuição neste protótipo é o tratamento de colisões e auto-colisões para retalhos de tecido. Um retalho de tecido é representado por uma grade de vértices interconectados por arestas, formando pequenos triângulos. Para o tratamento de colisões é empregada uma grade regular que mapeia triângulos de tecidos no espaço da simulação. Para detectar colisão é feita uma busca na vizinhança ao estilo da detecção de colisão de partículas.

6. A interação entre tecidos e líquidos foi implementada num sexto protótipo. Este esquema segue o trabalho apresentado em [3], onde o tratamento de colisões emprega duas grades, uma fina para mapear partículas (porções) de líquido e outra mais grossa para mapear triângulos de tecido.

7. Um sétimo protótipo, que permite a interação entre sólidos e líquido, foi implementado segundo a abordagem apresentada em [24]. Nesta implementação não houve contribuições importantes, apenas a junção dos protótipos primeiro e quarto.

O capítulo 6 apresenta duas técnicas para renderização da superfície de líquidos. Tipicamente a técnica de marcha de cubos (*Marching cubes*) [25] é utilizada. Esta, entretanto, dispende um tempo considerável para computar uma iso-superfície resultando não adequada para simulações em tempo real. Uma técnica que se adapta bem para visualizar superfícies de conjuntos grandes de partículas é a apresentada por Laan et al. [26]. A renderização da superfície segue um esquema baseado em

processamento de imagens, onde partículas que representam porcões de líquido são renderizados como esferas num estágio inicial. Num segundo estágio texturas com valores de normais e profundidade (*z-buffer*) são suavizadas. Finalmente, à imagem suavizada, correspondente à superfície visível do líquido, são aplicadas propriedades de transparência, refração, reflexão, sombra, iluminação para obter efeito de cáusticas.

No capítulo 7 são apresentados os resultados obtidos nos diferentes protótipos implementados, e por fim as conclusões e sugestões para trabalhos futuros são apresentadas no Capítulo 8.

Capítulo 2

Revisão bibliográfica

Neste capítulo são apresentados alguns conceitos importantes para a o trabalho desenvolvido na tese, a saber: conceitos de computação paralela e a Unidade de Processamento Gráfico, métodos de integração, métodos de detecção de colisão e métodos de simulação baseada em física para animar diferentes tipos de objetos.

No escopo de detecção de colisão, as técnicas e abordagens são classificadas em duas categorias: técnicas que usam o espaço do objeto e técnicas que usam o espaço-imagem. Existindo implementações sequenciais que se executam na **CPU**, assim como existem algoritmos para processamento paralelo que podem ser executados na CPU (com múltiplos núcleos de processamento) ou na **GPU**. Já na área de animação baseada em física são revisados trabalhos que usam modelos baseados em malhas, modelos baseados em pontos e modelos que usam partículas para representar os objetos.

2.1 Computação paralela e a unidade de processamento gráfico

No contexto de animação física, procura-se desenvolver técnicas eficientes para simular situações do mundo real, tais como: iluminação realista, movimento de fluidos (especialmente água), cabelos, fios, grama (vegetação em geral), objetos sólidos, entre outras. Estas tarefas de simulação exigem um esforço computacional considerável, o que sugere o emprego de arquiteturas computacionais de alto desempenho. De fato, o surgimento das unidades de processamento gráfico (UPGs, ou em inglês, GPUs), passaram a suprir poder computacional elevado a um custo relativamente modesto. Assim, a computação está evoluindo de um "processamento sequencial" na CPU para um "processamento paralelo" tanto na CPU (multi-core) quanto na GPU. Para viabilizar esse novo paradigma em computação, companhias como NVIDIA e AMD, especializadas no fabrico de hardware gráfico, empenham-se na criação de

novas arquiteturas de computação paralela que sejam utilizadas como verdadeiras plataformas para desenvolvimento de aplicativos gráficos e de propósito geral.

Em particular a tecnologia CUDA, desenvolvida pela NVIDIA, tem sido recebida com entusiasmo pela comunidade científica, beneficiando assim, o meio acadêmico e companhias de diversas áreas.

2.1.1 Computação paralela

Computação paralela é uma forma de computação onde os cálculos são executados de forma simultânea. Isto pode ser realizado de várias maneiras, variando de um paralelismo a nível bit-a-bit (ou seja, operações em múltiplos bits ao mesmo tempo), paralelismo a nível de instruções, paralelismo de dados (o mesmo cálculo é feito em dados diferentes) e paralelismo de tarefas (diferentes cálculos são realizados nos mesmos dados). Estas formas diferentes foram classificadas por Michael J. Flynn, quem criou o que hoje é chamado de taxonomia de Flynn [27].

- SISD: Uma instrução, um dado
- SIMD: Uma instrução, múltiplos dados
- MISD: Múltiplas instruções, um dado
- MIMD: Múltiplas instruções, múltiplos dados

Cabe ressaltar que a Unidade de Processador Gráfico (GPU) é um processador especializado, usado para acelerar a computação gráfica, e atualmente pode ser encontrada em diversos dispositivos acessíveis como computadores de mesa, computadores portáteis, telefones celulares, consoles de jogo, entre outros. Sua natureza paralela é desenhada para permitir o processamento simultâneo de grandes quantidades de operações de ponto flutuante, o que permite que diversos tipos de aplicações possam se beneficiar em desempenho.

Recentemente a GPU evoluiu para ser usada como um processador de propósito geral. Este tipo de abordagem é chamado de Computação de Propósito Geral em Unidades de Processamento Gráfico (GPGPU [28]). Na GPU, o número de operações de ponto flutuante por segundo (FLOPS) é até três ordens de grandeza maior que o obtido com CPUs convencionais. Outra vantagem das GPUs é que estas empregam arquiteturas de memória com grande largura de banda. Estes dois fatores combinados fazem da GPU uma plataforma de processamento paralelo barato, extremamente interessante para simular modelos computacionalmente pesados. Uma GPU atual, permite simular modelos físicos em tempo real, o que anteriormente apenas podia ser realizado por clusters de computadores.

2.1.2 Computação de propósito geral em GPUs

As GPUs modernas provêm um grande poder computacional para resolver problemas que são paralelizáveis, a saber: codificação-decodificação de video, reconhecimento de padrões em imagens, simulação física para jogos, visualização volumétrica, entre outros.

Abordagens recentes de GPGPU permitem criar aplicações de Computação de Alto Desempenho (HPC). Neste contexto, por exemplo, são tratados problemas de simulação de fluidos, dinâmica molecular, processamento de imagens médicas, entre outros, conseguindo assim que o pesquisador possa visualizar seus resultados rapidamente. Em alguns casos a rapidez de execução do algoritmo paralelo (executado numa GPU moderna) pode ser na ordem de dezenas de vezes mais rápido comparado com o algoritmo sequencial (executado numa CPU moderna). Entretanto, a GPU não resolve todos os problemas, já que há muitas limitações para que uma determinada aplicação possa ser executada de maneira eficiente, principalmente quando há dependência entre os dados a serem processados.

Unidade de processamento gráfico (GPU)

As GPUs evoluíram, em menos de uma década, de periféricos relativamente simples para unidades com alto poder de processamento, sendo seu uso quase indispensável em aplicações gráficas interativas. De fato, a evolução em rapidez e desempenho, junto com as melhorias na *programabilidade*, têm feito que as GPUs sejam consideradas verdadeiras plataformas computacionais, viabilizando a implementação de uma variedade de aplicações não gráficas.

Inicialmente, as GPUs foram projetadas para ter um *pipeline*[1] de funcionalidade fixa (ver Figura 2.1). Entretanto, notou-se a falta de flexibilidade para gerar efeitos complexos, como sombras, iluminação, entre outros. Com o propósito de superar esta limitação, o projeto Renderman [29] propôs a incorporação de um pipeline flexível. Os resultados obtidos por esta proposta foram de melhor qualidade frente aos obtidos usando apenas a funcionalidade fixa. Esta idéia foi adotada para projetar uma nova arquitetura [30] para as GPUs, adicionando unidades programáveis. Na primeira geração de placas gráficas, duas unidades programáveis foram introduzidas: a unidade de processamento de vértices e a unidade de processamento de fragmentos. Esta arquitetura, permite ao usuário escrever programas especializados chamados *shaders* [2], que são executados nos processadores de vértices (*vertex shader*) ou de

[1]Em GPU, pipeline se refere a uma sequência de etapas na qual é dividido o processo de renderização, desde o envio de primitivas, usando alguma API gráfica, até a saída na tela do computador.

[2]No campo da computação gráfica, um *shader* é um pequeno programa (conjuntos de instruções) usado para calcular efeitos de renderização na GPU.

fragmentos (*fragment shader*).

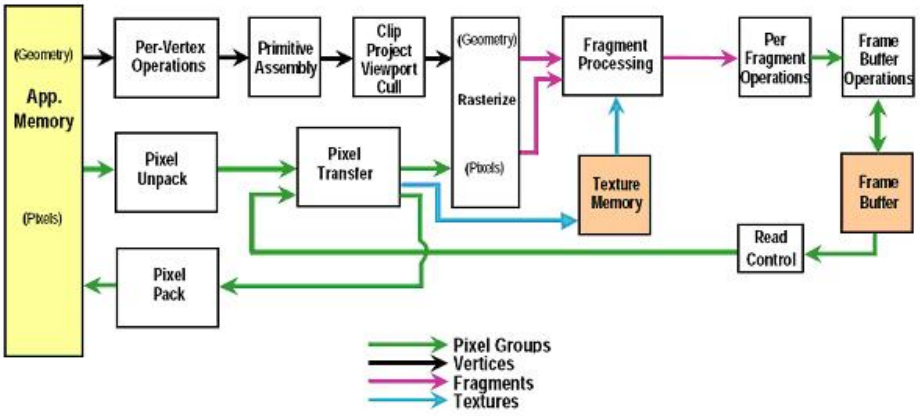


Figura 2.1: Pipeline gráfico com funcionalidade fixa.

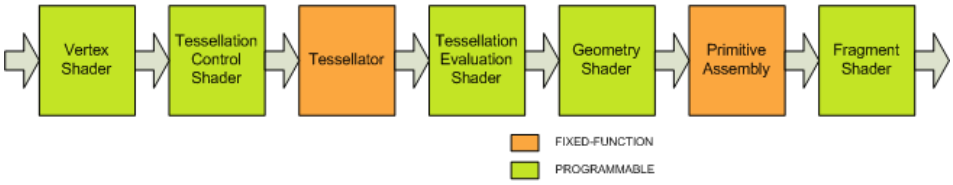


Figura 2.2: Pipeline gráfico programável.

Este pipeline com duas unidades de processamento, uma para vértices e a outra para fragmentos, foi substituído por um modelo unificado chamado *Shader Model 4.0*, permitindo que programas de vértices (*vertex programs*) e programas de fragmentos (*fragment programs*) usem o mesmo conjunto de instruções [31]. O modelo unificado inclui também programas de geometria (*geometry programs*), que permite alterar a geometria associada aos vértices de entrada (primitivas), e o novo shader para geração de mosaico (tessellation shader). A figura 2.2 mostra, no pipeline gráfico, a localização das partes programáveis e as partes que continuam fixas.

Arquitetura da GPU

As GPUs estão sendo utilizadas em diferentes classes de aplicações, frequentemente superando em desempenho CPUs modernas. Entretanto, a arquitetura das GPUs evoluíram numa direção diferente das CPUs, adotando uma abordagem onde os recursos da unidade de processamento são divididos em estágios à maneira de um pipeline. Assim, uma parte de um processador trabalha num estágio, alimenta diretamente com sua saída outra parte que trabalha com o estágio seguinte, como acontece, por exemplo, com os estágios de processamento de vértices e rasterização.

Nas primeiras versões de GPUs programáveis o conjunto de instruções do processador de vértices e do processador de fragmentos eram diferentes, e em estágios separados. Já nas GPUs modernas, esses processadores convergiram para uma arquitetura programável única (veja a Figura 2.3), obedecendo ao mesmo conjunto de instruções. Esta arquitetura é composta por vários processadores de fluxos (*streaming processors*), que trabalham em paralelo e dividem seu tempo no processamento de vértices, de geometria e de fragmentos.

A principal desvantagem deste modo de trabalho é o balanceamento de carga, onde o desempenho geral depende do estágio mais lento. Por exemplo, se o programa de vértices é complexo e o programa de fragmentos é simples, a rapidez de execução dependerá do programa de vértices. Em 2009, a NVIDIA apresentou o surgimento de uma nova geração de Processadores Gráficos com o codinome FERMI [32]. Estes processadores têm a capacidade de processar vários *kernels* [3] simultaneamente, levando a outro nível de paralelismo.

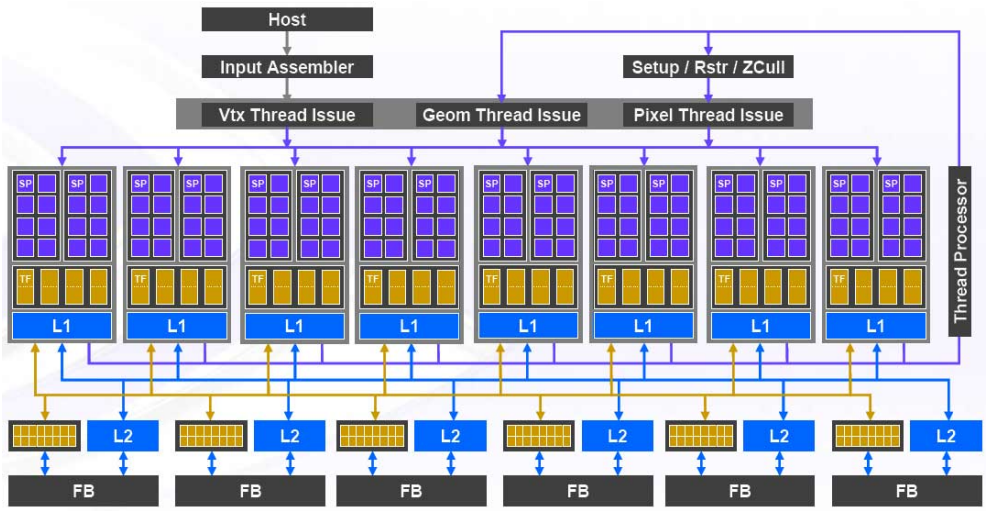


Figura 2.3: Arquitetura do modelo unificado da NVIDIA.

Programabilidade

Na programação das GPUs um aspecto importante é o acesso à memória, nas primeiras GPUs apenas podia ser feito através de busca em memória de textura (*texture look-up*). Em GPUs modernas, com o novo modelo unificado, o acesso à memória pode ser feito nos três tipos de shaders, entretanto, apenas operações de leitura (*gather*) são suportadas. Operações de escrita (*scatter*), podem ser feitas de forma limitada através de *vertex shaders*.

Em 2006, a geração de placas gráficas G80 da NVIDIA foi introduzida, assim como a tecnologia CUDA [33] (*Compute Unified Device Architecture*) para progra-

[3] *Kernel* no contexto de programação de GPUs significa *núcleo de processamento paralelo de fluxo*.

mação de GPUs, com o propósito de utilizar todos os recursos das placas gráficas de forma simples e eficiente. Em CUDA a implementação é dividida em trechos de código chamados *kernels*, que são executados em paralelo organizados por *threads*[4].

A serie G80 de GPUs e a tecnologia CUDA trouxeram o conceito de multi-processador. Cada multi-processador possui uma nova arquitetura SIMT (*Single Instruction Multiple Thread*) e mapeia cada linha de processamento (*thread*) num processador escalar. Uma unidade de multi-processamento SMIT cria, gerencia, agenda e executa linhas de processamento em grupos paralelos de 32 chamados de *warps*.

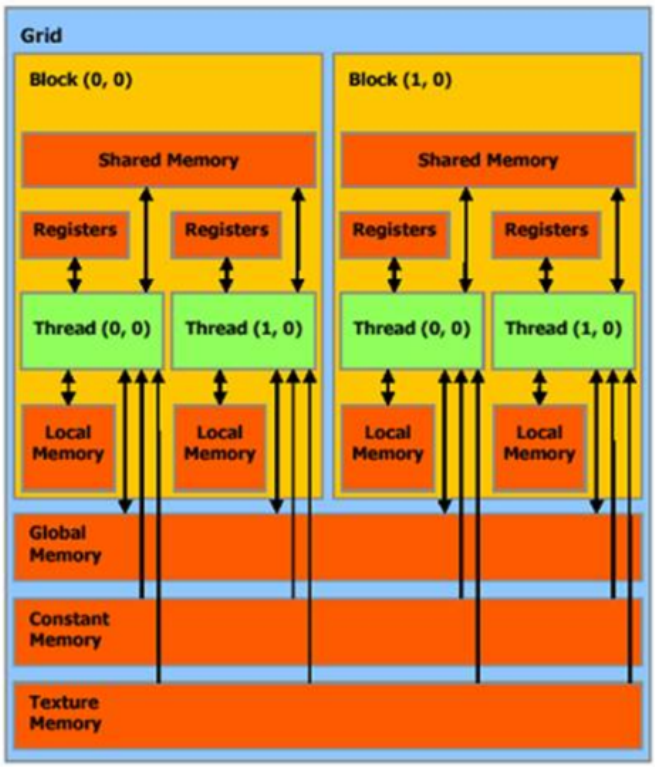


Figura 2.4: Grade de blocos em CUDA.

[4]Uma *thread*, ou seja, uma linha de processamento, no contexto da programação de GPUs é responsável pela tarefa mais simples a ser computada.

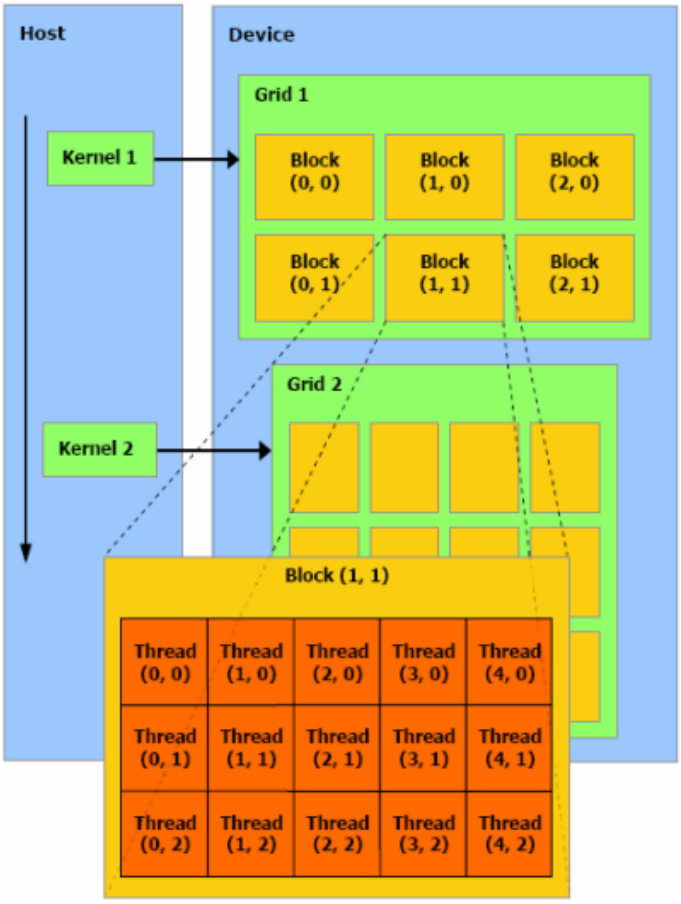


Figura 2.5: Arquitetura do modelo de programação do CUDA.

Em CUDA um bloco é definido por um lote de linhas de processamento que podem cooperar entre elas para compartilhar recursos de um multi-processador da GPU sincronizando eficientemente sua execução e o acesso a memória (ver Figura 2.4).

Numa aplicação, a quantidade de blocos e a quantidade de linhas de processamento por bloco a serem usados são definidos na CPU e passados como parâmetros à GPU. Esta configuração é chamada de grade (ver Figura 2.5) e apenas um kernel pode ser executado nela. Os kernels são similares aos shaders e executam as mesmas instruções para múltiplos dados.

Dependendo da aplicação, no kernel podem-se especificar pontos de sincronização, onde linhas de processamento num bloco são suspendidas até que todas as linhas de processamento no bloco cheguem ao ponto de sincronização.

Note-se que um multi-processador processa um bloco por vez e, se o número de multi-processadores na GPU é menor que o número de blocos na grade, blocos excedentes serão processados seqüencialmente.

Na tecnologia CUDA, assim como na programação tradicional em GPU, o maior inconveniente é a falta de ferramentas para depuração. Assim, por exemplo, resultados parciais de um processamento não podem ser inspecionados com facilidade, o

que torna a implementação tediosa e suscetível a erros.

Embora as GPUs tenham sido criadas para uso em aplicações gráficas 3D interativas, arquiteturas modernas foram desenhadas para suportar computação massiva em paralelo. A arquitetura das GPUs modernas, conformado por unidades programáveis, permite implementar aplicações de propósito geral em duas formas. A primeira baseada no uso de *shaders* que são executados nas partes programáveis do pipeline gráfico e a segunda baseada no uso de *kernels* na plataforma CUDA.

Uma aplicação na GPU empregando shaders tipicamente obedece ao seguinte padrão:

- primeiro, são enviadas primitivas geométricas para o processador de vértices. Neste estágio, um programa de vértices pode ser executado substituindo o processamento da funcionalidade fixa, que normalmente realiza cálculos de luz e transformações geométricas;

- na sequência, os vértices transformados são enviados para o processo de montagem de primitivas. As GPUs modernas incorporam o *geometry shader* neste estágio, permitindo modificar a geometria de entrada;

- a seguir, as primitivas são rasterizadas. Este processo permite gerar fragmentos para os pixels gerados pelas primitivas no espaço-imagem;

- a seguinte etapa, processamento de fragmentos, é considerada o motor de computação das GPUs, já que nele a maior parte do processamento é realizado. O *fragment shader*, programa de passos que substitui a funcionalidade fixa, pode ler dados da memória da GPU através de leitura de texturas, embora, possa escrever apenas na posição do fragmento que lhe foi atribuído;

- finalmente, o processo de composição, calcula um valor ou valores por fragmento e gera uma matriz de pixels chamado **frame buffer**. Este buffer normalmente é o resultado final da aplicação, e é exibido na tela do computador. Entretanto, os valores do buffer podem ser guardados em textura, usando por exemplo, a extensão *Frame Buffer Object* do OpenGL [34], para serem usados numa computação posterior. Aplicações complexas, frequentemente usam vários *shaders* em várias passadas (*multipass applications*) reutilizando valores obtidos em passos prévios.

2.2 Métodos de integração

Uma aplicação de simulação física requer a implementação de métodos que resolvem numericamente equações diferenciais que são derivadas durante a simulação. Existem uma grande variedade de métodos e cada um possui vantagens e desvantagens

a serem consideradas, sendo comum a escolha basear-se entre rapidez, acurácia e/ou estabilidade do método. Em geral, deseja-se que o método seja rápido, exato e robusto, embora, na maior parte de casos não seja possível obter as três características simultaneamente.

Dentre estes métodos, frequentemente são utilizados os baseados no método de Euler que, por sua vez, usam o teorema de Taylor para serem aproximados.

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

$$\begin{aligned} P_n(t_1, t_0) &= \frac{f^{(n)}(t_0)}{n!}(t_1 - t_0)^n + \frac{f^{(n-1)}(t_0)}{(n-1)!}(t_1 - t_0)^{n-1} + \cdots + f'(t_0)(t_1 - t_0) + f(t_0) \\ &= \sum_{k=0}^{n} \frac{f^{(k)}(t_0)}{k!}(t_1 - t_0)^k. \end{aligned} \tag{2.1}$$

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

- forma de Lagrange

$$R_n(t_1, t_0) = \frac{f^{(n+1)}(\bar{t})}{(n+1)!}(t_1 - t_0)^{n+1}, \quad \bar{t} \in (t_0, t_1). \tag{2.2}$$

- forma de Cauchy

$$R_n(t_1, t_0) = \frac{f^{(n+1)}(\bar{t})}{n!}(t_1 - \bar{t})^n(t_1 - t_0), \quad \bar{t} \in (t_0, t_1). \tag{2.3}$$

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

$$\dot{x}(t) \approx f(t, x(t)), \qquad \text{onde} \qquad x(t_0) = x_0. \tag{2.4}$$

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

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

$$x(t_{i+1}) = x(t_i) + \dot{x}(t_i)h + \ddot{x}(\bar{t})\frac{h^2}{2}, h = t_{i+1} - t_i \geq 0, \bar{t} \in [t_i, t_{i+1}]. \tag{2.5}$$

Pode-se notar que $x(t)$ é uma solução para a equação 2.4 e podemos trocar $x_i = x(t_i), \forall i$.

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

$$y_{i+1} = y_i + hf(t_i, y_i), \qquad i \geq 0. \tag{2.6}$$

Assim, pode-se ver que o método é impreciso e instável já que requer que o intervalo de tempo seja pequeno.

Um método também bastante popular é o proposto por Verlet [35]. Este método e suas variações se tornaram populares na indústria de jogos através do trabalho de Jakobsen [21], e também é baseado no teorema de Taylor.

$$x(t_{i+1}) = x(t_i) + \dot{x}(t_i)h + \frac{1}{2}\ddot{x}(t_i)h^2 + \frac{1}{6}x^3(t_i)h^3 + O(h^4) \tag{2.7}$$

$$x(t_{i-1}) = x(t_i) - \dot{x}(t_i)h + \frac{1}{2}\ddot{x}(t_i)h^2 - \frac{1}{6}x^3(t_i)h^3 + O(h^4). \tag{2.8}$$

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

$$x(t_{i+1}) = 2x(t_i) - x(t_{i-1})h + \ddot{x}(t_i)h^2 + O(h^4). \tag{2.9}$$

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

$$x(t_{i+1}) = 2x(t_i) - x(t_{i-1}) + \frac{h^2}{m}F(t_i, x(t_i), \dot{x}(t_i)), \qquad i \geq 1, \tag{2.10}$$

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

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

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

Além dos métodos baseados no teorema de Taylor, existem outros métodos conhecidos, por exemplo, o *m*étodo previsor-corretor e os *m*étodos de extrapolação. Uma explicação detalhada destes métodos pode ser encontrada no texto de

Eberly [36].

2.2.1 Métodos de integração para o movimento

Em 2005, Mueller et al. [1] apresentaram uma extensão do método de Euler para integração de velocidade e posição, chamado método semi-implícito de Euler. É chamado de semi-implícito por que primeiro é computada a velocidade e em seguida é computada a posição utilizando essa nova velocidade.

$$v(t+h) = v(t) + h\frac{f(t)}{m} \tag{2.11}$$

$$x(t+h) = x(t) + hv(t+h) \tag{2.12}$$

O método é apropriado para simulações com passo de tempo pequenos. Já para intervalos de tempo longos torna-se instável.

2.3 Detecção de colisão

Na última década houve uma rápida evolução em aplicações de simulação física e realidade virtual, principalmente pela evolução das GPUs e arquiteturas de múltiplos núcleos de processamento (*multicore*). Esta evolução ainda não é suficiente para algumas aplicações onde é necessário gerar animações em tempo real, já que o processo de detecção de colisão gasta a maior parte do processamento em cada instante de tempo durante a simulação. A detecção de colisão é uma componente fundamental em sistemas de simulação física, já que, disso depende a interação de objetos. No pior caso, pode haver $O(n^2)$ pares de objetos em colisão potencial. Por sua vez, para encontrar os pares de objetos que realmente estão em colisão e o lugar onde há colisão é necessário o uso de consultas de proximidade. O tempo gasto neste processo depende da complexidade geométrica dos objetos envolvidos. Diversas técnicas abordam o problema tentando diminuir o tempo gasto neste processo, destacando-se as técnicas que fornecem informação para o processo de resposta à colisão e a separação dos objetos interpenetrados.

A detecção de colisão frequentemente é dividida em duas etapas. A primeira é uma detecção de colisão grosseira (*broad phase*), cujo objetivo não é a exatidão dos resultados, mas apenas a eliminação de pares de objetos que garantidamente não colidem. A segunda é a detecção de colisão exata (*narrow phase*), que permite detectar colisões reais, ou seja, o local exato onde há colisão. Esta etapa frequentemente é a mais custosa já que depende da complexidade geométrica dos objetos em colisão.

As técnicas para detecção de colisão podem ser subdivididas em dois grupos:

baseados no espaço do objeto e baseados no espaço-imagem.

- As técnicas baseadas no espaço do objeto utilizam cálculos geométricos para detectar interferência entre objetos. Um expediente comum é o emprego de subdivisão espacial e subdivisão do objeto em partes. Como resultado, obtém-se estruturas de dados hierárquicas ou grades, facilitando a consulta a vizinhanças localizadas. Estas técnicas foram originalmente criadas para serem executadas em CPUs, mas com a evolução das GPUs, novos algoritmos foram propostos [37] para aproveitar a arquitetura paralela.

  Nesta fase tipicamente são usados esquemas de varredura e poda [11, 38, 39]. Neste sub conjunto de técnicas, recentemente foi apresentado um esquema [40] para ser executado em GPUs que permite tratar colisão em cenários com dezenas de milhares de objetos conseguindo, portanto, um novo patamar de desempenho para aplicações interativas. Estas técnicas, entretanto, foram desenhadas principalmente para serem usadas em sistemas que envolvem corpos rígidos, não existindo informação precisa acerca do seu uso em sistemas com objetos deformáveis. Já no caso de detecção de colisão exata, uma grande variedade de algoritmos foram apresentados, desde técnicas que usam hierarquias de volumes limitantes (esferas, elipses, caixas alinhadas com os eixos, caixas orientadas e politopos de orientação discreta) [11–15], subdivisão do espaço em grades [41] e hierarquias de subdivisão do espaço (Octrees, k-d Trees, BSP-Trees) [42]. Dentre elas, as hierarquias que usam caixas alinhadas com os eixos (AABBs) e esferas envolventes podem ser usadas em sistemas que envolvam objetos deformáveis [13, 15], já que são rápidas na atualização e eficientes em memória.

  Um método de detecção de colisão baseada em hierarquias balanceadas de AABBs é apresentado em [37]. O algoritmo mapeia as AABBs em texturas para serem usadas na GPU. Durante o percurso da hierarquia, consultas de visibilidade são usadas para contar o número de pares de AABBs sobrepostas. A varredura recursiva das hierarquias de AABBs é executada inteiramente na GPU, porém, o algoritmo não consegue atingir taxas interativas já que incorre num número excessivo de operações redundantes. Outra técnica similar é a apresentada por Choi [43], que aproveita o poder de processamento das GPUs para executar auto-colisões em modelos deformáveis, tais como tecidos.

- As técnicas baseadas no espaço-imagem (espaço definido para renderização) tipicamente usam representações discretizadas de projeções de objetos para acelerar consultas de colisão. Estas projeções usualmente são feitas renderizando as primitivas dos objetos no frame buffer. Como estas técnicas trabalham numa representação discretizada do espaço 2D, não garantem resulta-

dos exatos. Assim, a efetividade destes algoritmos é limitada à resolução do espaço-imagem definida ou que as GPUs suportam.

Estas técnicas não precisam de estruturas de dados complexas já que operam em imagens obtidas após o processo de renderização da geometria. Entretanto, como a verificação de interferência é feita na resolução do espaço-imagem, os resultados são aproximados e suscetíveis a erros de precisão. Um trabalho pioneiro é apresentado por Shinya et al. [44], que descreve uma técnica para detecção de colisão entre objetos rígidos, onde partes da superfície dos objetos são renderizadas em buffers de profundidade e guardadas em imagens, que posteriormente são empregados para executar testes de intersecção através de operações booleanas. Esta técnica é simples e rápida, porém suporta apenas objetos convexos.

Posteriormente, Heidelberger et al. [45] apresentaram uma extensão da técnica de Shinya et al., para tratar objetos deformáveis usando imagens de profundidade chamadas de *Layered Depth Images*. Nesta técnica as imagens são usadas para armazenar entradas e saídas de raios paralelos lançados a partir de diversos pontos de vista, podendo assim tratar objetos não convexos.

Técnicas mais atuais utilizam outros buffers da GPU[5] para verificação de interferência, principalmente com o propósito de suportar objetos de geometria e forma arbitrárias [10, 46].

Existem também técnicas que usam a GPU apenas para detectar interferência. Aproveitando a funcionalidade conhecida como *Occlusion Query*[6] que foi incorporada na primeira geração de GPUs programáveis [47, 48].

No entanto, as técnicas baseadas no espaço-imagem têm algumas desvantagens. Por exemplo, a necessidade de processar na CPU as imagens obtidas da GPU esbarram na limitação de transferência de dados entre **CPU-GPU-CPU** que depende das dimensões do espaço-imagem suportado pela GPU. A exceção das técnicas que permite filtrar pares de objetos em colisão potencial usando consultas de visibilidade [47, 49].

Outro problema é o custo de renderização, já que usualmente é necessário renderizar cada par de objetos em colisão potencial. Knott and Pai [5] propuseram um algoritmo que trata vários objetos numa renderização só, porém, a renderização é em modo aramado (*wireframe*), o que não garante uma detecção de colisões eficiente.

[5] A detecção de interferência frequentemente usa buffers de cor, de profundidade e/ou de estêncil.

[6] Esta funcionalidade realiza consultas de visibilidade, reportando o número de pixels visíveis após uma renderização.

Uma pesquisa estendida acerca das técnicas que usam o espaço-imagem é apresentada por Teschner et al. [50], mostrando que estas técnicas são adequadas para serem usadas em ambientes dinâmicos com objetos deformáveis, já que, em geral, não requerem de pré-processamentos ou estruturas de dados complexas tais como hierarquias de volumes limitantes. Técnicas para tratar múltiplos objetos numa renderização são apresentadas por Hang-Yong et al. [8, 9], conseguindo detectar também pontos de contato, que podem ser aproveitados no processo de resposta às colisões.

2.3.1 Detecção de colisão grosseira

Apesar de, verificações de intersecção usando volumes limitantes serem relativamente baratas, verificar todos os $\binom{n}{2}$ pares numa coleção de n volumes ainda expende uma computação considerável. Numa situação comum, onde poucos objetos estão em colisão, um algoritmo que consiga identificar os pares de objetos que se intersectam é preferível, em vez de um algoritmo de força bruta $O(n^2)$ que verifica todos os pares.

Uma das técnicas mais utilizadas é a de varredura e poda (*sweep and prune*) [11]. Esta técnica se baseia num algoritmo incremental que detecta um conjunto de volumes limitantes que se intersectam durante a simulação e, tem uma complexidade $\Omega(n \log n)$ no pior caso. A idéia geral reside na observação que se as projeções de dois objetos sobre uma linha reta não se intersectam, então pode-se garantir que os próprios objetos não se intersectam. Assim, o método consiste em calcular as projeções dos objetos da cena em um certo número de retas –normalmente são empregadas projeções sobre os eixos coordenados–. Para cada reta é mantida uma lista dos intervalos de projeção dos volumes limitantes (veja a Figura 2.6). Estas listas são atualizadas inserindo os intervalos de projeção, para cada passo de tempo, e ordenando-os pelo método de inserção. Um aspecto importante do método é a escolha do volume limitante, sendo usualmente empregadas as AABBs e esferas limitantes.

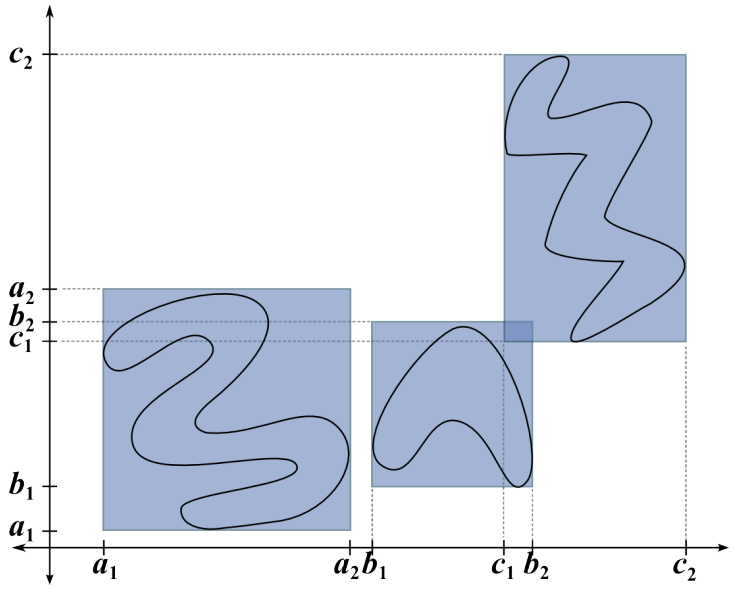


Figura 2.6: método de detecção de colisão sweep and prune em 2D.

No contexto de programação em GPUs, Le Grand [22] apresentou uma técnica de detecção de colisão em paralelo reduzindo drasticamente o número de pares de objetos em colisão potencial. A técnica usa esferas limitantes mapeadas numa grade 3D de forma tal que só esferas em células vizinhas podem colidir. Isto permite que o algoritmo seja implementado de maneira eficiente em paralelo. Recentemente, Liu et al. [40] apresentaram uma técnica baseada no esquema de varredura e poda para tratar colisão em dezenas de milhares de esferas ou caixas limitantes.

Métodos de particionamento do espaço, como os que serão mencionados na seção 2.3.2, podem também ser adaptados para serem usados na etapa de colisão grosseira. A técnica é especialmente útil quando usado em cenários com uma grande quantidade de objetos em movimento [51].

2.3.2 Detecção de colisão exata

Os métodos que abordam o problema de detecção de colisão exata podem ser classificados em cinco grupos: técnicas baseadas em hierarquias de volumes limitantes, técnicas de subdivisão espacial, técnicas que usam campos de distância, técnicas que usam o espaço-imagem e, recentemente, técnicas que se baseiam no uso de partículas.

Hierarquias de volumes limitantes

Estruturas de dados hierárquicas baseadas em volumes limitantes têm alcançado excelentes resultados em animação de objetos rígidos, já que podem ser construídas numa etapa de pré-processamento. No entanto, estas estruturas não são adequadas para a simulação de objetos deformáveis, onde é necessário uma atualização a cada

nova forma assumida pelo objeto durante a simulação. A idéia básica nesta abordagem é particionar o objeto recursivamente obtendo como resultado uma árvore. Os nós-folha referem-se às primitivas do objeto. Cada nó da árvore possui um volume limitante que garantidamente contém todas as primitivas contidas nos nós-filho descendentes. Este volume limitante é utilizado para fazer verificações rápidas de intersecção como condição para descer na hierarquia.

Como mencionado no inicio deste Capítulo, existem vários tipos de hierarquias de volumes limitantes e usualmente são verificadas de cima-para-baixo, testando recursivamente a interferência entre os volumes limitantes contidos em cada par de nós. Se ambos os nós são folhas, é feito um teste exato entre primitivas. Se um nó é folha e o outro é interno, o volume limitante do nó folha é testado contra o dos filhos do nó interno. Para otimizar este processo, se ambos os nós são internos, é recomendado verificar o nó com menor volume limitante versus os filhos do nó maior.

As hierarquias de volumes limitantes mais comuns utilizam caixas limitantes alinhadas com os eixos (ver Figura 2.7), esferas limitantes (ver Figura 2.8) e caixas limitantes de orientação arbitrária (ver Figura 2.9).

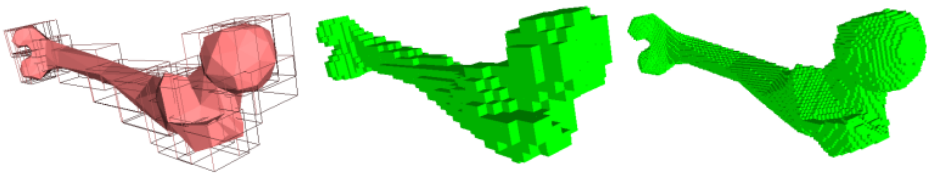


Figura 2.7: Hierarquia de caixas limitantes alinhadas com os eixos (AABB-Tree).

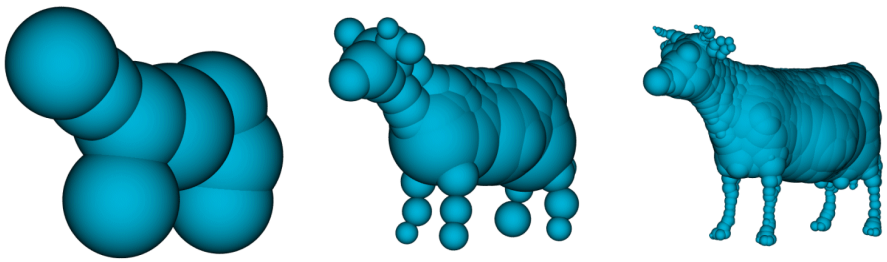


Figura 2.8: Hierarquia de esferas limitantes (Sphere-Tree).

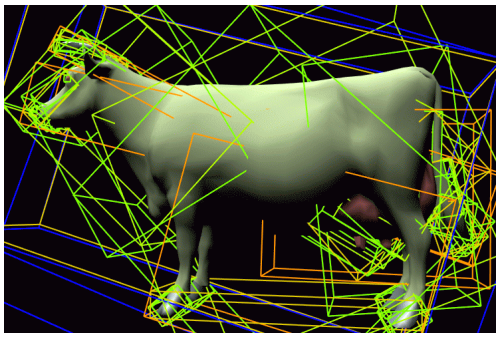


Figura 2.9: Hierarquia de caixas limitantes orientadas (OBB-Tree).

Subdivisão espacial

Em contraste com as técnicas de volumes limitantes que particionam o espaço do objeto, as técnicas de subdivisão espacial particionam em células (voxels) o espaço onde os objetos estão imersos, ou seja, o cenário onde os objetos interagem. Cada célula da partição mantém uma lista das primitivas que podem ser completa ou parcialmente contidas na célula. A principal consideração, em métodos para subdivisão espacial é a escolha de uma estrutura de dados flexível e eficiente na atualização e no uso de memória. Diversos esquemas de subdivisão espacial têm sido propostos, entre eles, as grades uniformes [41, 52], octrees [53], BSP-trees [51], kd-trees [54] e tabelas de dispersão espacial [19, 55].

Na construção destas estruturas de dados, a distribuição espacial dos objetos pode ou não ser considerada na geometria de subdivisão. Entre as primeiras incluem-se as BSP-Trees e kd-trees, por exemplo, que dividem o espaço 3D baseado na orientação e na posição de algumas primitivas escolhidas. Já em abordagens que não dependem dos objetos, como octrees e grades regulares, por exemplo, a subdivisão sempre ocorre segundo planos pré-determinados. Na verdade, a diferença reside na geometria de partição.

A detecção de colisão baseada em subdivisão espacial requer que as primitivas de todos os objetos sejam discretizadas em células. Assim, para cada célula que contém mais de uma primitiva, se faz uma verificação de intersecção a fim de encontrar colisões entre elas. Quando objetos deformáveis são considerados, é necessário atualizar a estrutura de dados a cada passo de tempo. Teschner et al. [19] apresentaram uma idéia simples para evitar atualizações desnecessárias. A idéia é manter em cada célula uma variável indicando o último instante de tempo (*timestamp*) em que foi atualizado. Mais tarde, as verificações de intersecção são feitas apenas nas células que foram atualizadas no passo de tempo corrente. Este processo é explicado em detalhe na seção 3.2.3.

Espaço-imagem

Diversos autores [4, 8–10, 44, 45, 56, 57] têm se dedicado à construção de algoritmos rápidos e confiáveis usando o espaço-imagem. Tipicamente, estas técnicas são baseadas em GPU e utilizam os buffers de profundidade, cor e estêncil.

Um algoritmo típico de detecção de colisão baseado no espaço-imagem envolve o seguintes passos: Inicialmente, computa-se uma AABB para cada objeto, e a cada passo de tempo um conjunto de objetos em colisão potencial (PCS, do inglês *Potentially Colliding Set*) é computado usando as AABBs.

O espaço-imagem é inicialmente determinado para conter todas as AABBs projetadas ortográficamente. Em seguida, usando a técnica conhecida como *depth peeling*[7], subconjuntos de faces no PCS são renderizados camada a camada (ver Figura 2.10) em *buffers* (de cor) que por sua vez são mapeados em texturas. Na figura, as texturas tex_1 e tex_2 foram preenchidas com valores referentes à cena. Na figura 2.10 a textura tex_1 contém os valores de profundidade das faces visíveis na primeira camada, a textura tex_2 contém os valores de profundidade das faces visíveis na segunda camada, e assim por diante. O processo pára quando não existem mais faces visíveis, isto é, o buffer reporta zero pixels renderizados. Note-se que neste exemplo foram usadas apenas três texturas, entretanto, a última textura não é utilizada uma vez que nenhum pixel foi renderizado, indicando que não existe mais objetos a serem renderizados.

[7]Esta técnica permite extrair camadas de geometria da cena desde o ponto de vista do observador. Inicialmente foi usada para criar efeitos de transparência [58].

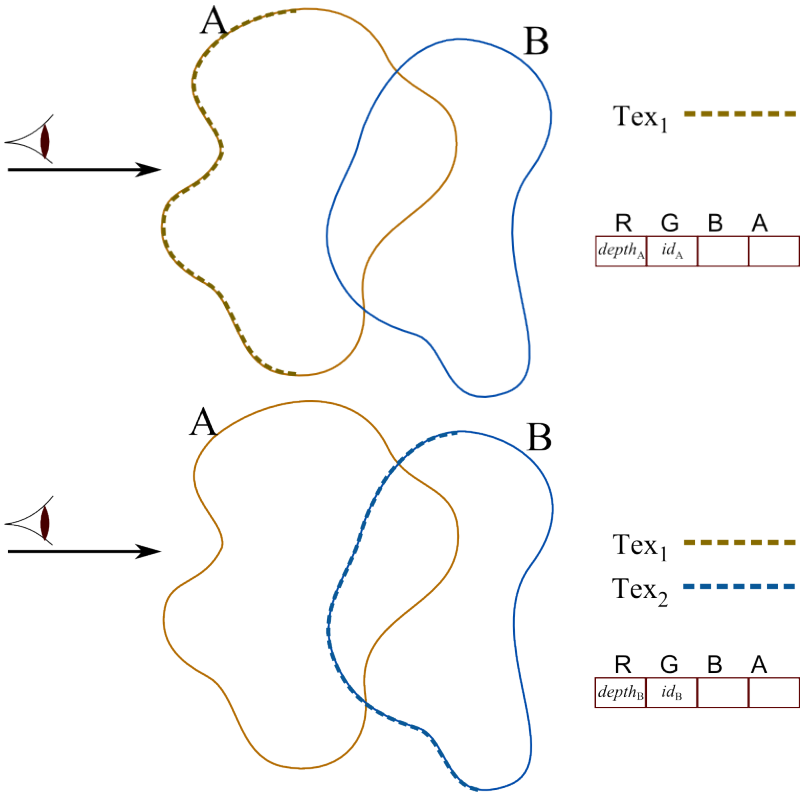


Figura 2.10: Extração de camadas com faces visíveis desde a posição do observador.

Para obter a primeira textura, é usado um programa de fragmentos que guarda nos canais de cor **R** e **G**, respectivamente, o valor da profundidade *depth* de cada pixel renderizado e o identificador do objeto *id* ao que pertence. As texturas seguintes são geradas usando a textura obtida na passagem anterior e um outro programa de fragmentos da seguinte forma.

- Seja tex_i a textura de entrada e tex_j a textura a ser gerada.
- Um pixel $p(u, v) \in tex_j$, que usa o texel[8] $t(u, v) \in tex_i$, é gerado apenas se $p.id \neq t.id$ e $p.depth < t.depth$.

Após ter extraído todas as camadas, o processo é repetido invertendo a direção de visão para atrás do cenário. Neste segundo passo, as texturas são usadas em ordem inverso e, os valores de profundidade e *id*s são guardados nos canais **B** e **A**. Assim, um pixel p das faces de trás pode ser encontrado na textura tex_1 ou na textura tex_2 (veja a Figura 2.11). Em ambos casos, o texel das faces da frente t é obtido usando a posição em coordenadas de textura do pixel p. Então o processo de detecção de colisão compara os *id*s em cada texel das texturas e reporta colisão apenas se estes forem diferentes.

[8]Nome dado a um pixel de imagem quando usada como textura.

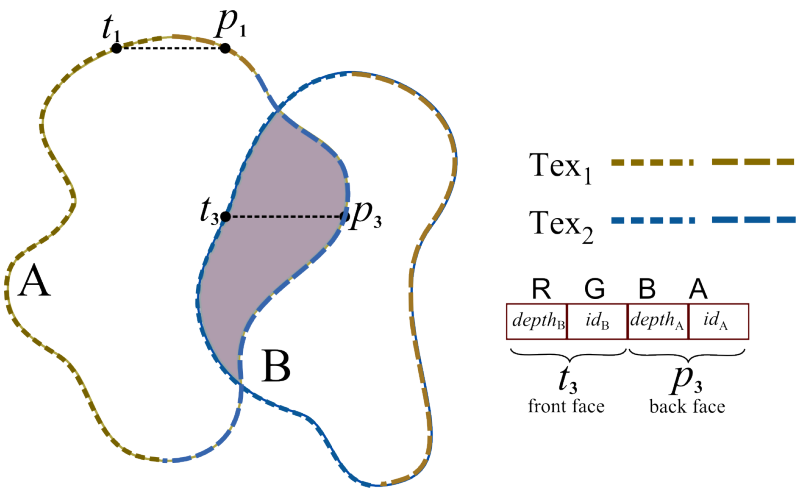


Figura 2.11: Texturas guardam informação de colisão.

Tipicamente técnicas de detecção de colisão apenas indicam se há ou não colisão, ignorando o local da colisão. Han-Young et al. [8] apresentaram uma extensão da técnica discutida acima de forma a obter pontos de colisão usando um programa de fragmentos que verifica em t e p, considerando a posição do observador, três situações: se p e t têm ids diferentes, se t não está atrás de p, e se a distância entre t e p é menor que uma pequena constante δ. A Figura 2.12 mostra os casos em que podem ser encontrados pontos de colisão.

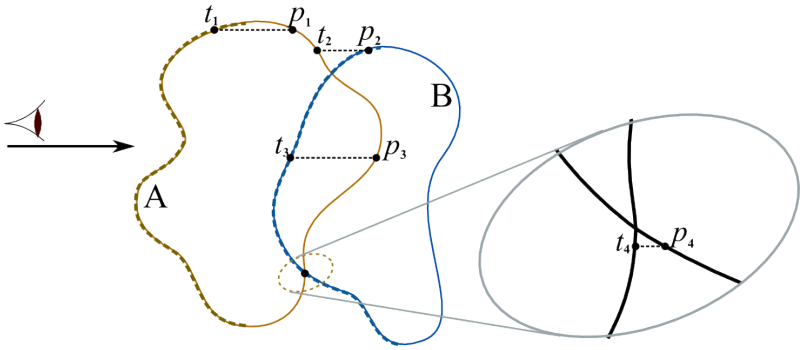


Figura 2.12: Encontrando pontos de colisão.

Após encontrados os pontos de colisão, os vetores normais para cada ponto podem também ser estimados usando pontos de colisão vizinhos não co-lineares. Estes pontos de colisão e vetores normais podem ser usados no esquema de simulação física baseado em impulsos como por exemplo o apresentado por Guendelman et al. [18].

Sistemas de partículas

Atualmente, métodos de simulação baseados em partículas tornaram-se bastante populares. Principalmente na criação de animação de fenômenos físicos, tais como: água [24, 59, 60], fumaça [60, 61], fogo [60], areia [16], fusão e interação com sólidos [24, 62], e outras extensões [63]. Entretanto, como relatado previamente, a

interação entre objetos requer um processo de detecção de colisão eficiente. Bell et al. [16] apresentam uma técnica eficiente para detectar colisões numa cena composta por uma grande quantidade de partículas. Nessa abordagem, uma partícula é representada por uma pequena esfera, assim a colisão se resume apenas à detecção de interferência entre esferas. Um método de força bruta consideraria $O(n^2)$ pares de partículas, o que é inaceitável para n da ordem de algumas dezenas de milhares de partículas. Para melhorar a eficiência, os autores usam uma grade espacial, onde cada voxel é um cubo cujo lado é igual ao diâmetro da maior esfera. Assim, o processo de detecção de colisão para cada partícula envolve apenas a avaliação das partículas que estão numa vizinhança de 27 voxels, isto é, as partículas dos 26 voxels adjacentes ao voxel onde a partícula avaliada se encontra e mais as partículas do mesmo voxel. Para aumentar a eficiência, é explorada a coerência temporal observando o fato de que as partículas avançam pequenas distâncias a cada passo de tempo e por conseguinte a grade muda muito pouco. Harada et al. [2] apresentam uma técnica (similar à técnica apresentada em [16]) que aproveita o poder computacional das GPUs conseguindo simular cenas com dezenas de milhares de partículas em tempo real. Parte desta tese se baseia nesta técnica que será descrita no Capítulo 5.

2.4 Simulação baseada em física

Simulação baseada em física é uma área intensamente investigada em computação gráfica, tendo sido publicados os primeiros trabalhos relevantes na década de 80 [64]. Trabalhos iniciais focam principalmente técnicas de massa-mola para modelos deformáveis usando métodos de integração explícita ou implícita. Por outro lado, diversos trabalhos se dedicam à simulação de objetos rígidos [18, 65–67]. Neste contexto, a maior dificuldade é tratar colisões com múltiplos contatos. Existem duas abordagens que tratam este problema: métodos por propagação [18] e métodos aplicando técnicas de otimização tais como métodos LCP [68] (*Linear Complementarity Problem*) qur também têm sido usados com sucesso para simular objetos quase-rígidos [69]. No entanto, estes métodos não garantem a existência de uma solução quando usado atrito no problema de múltiplos contatos. Este problema pode ser aliviado usando impulso-velocidade no lugar de usar força-aceleração [70]. O uso de impulso-velocidade na formulação da dinâmica de corpos rígidos provê uma forma natural para tratar mudanças no estado do corpo rígido. Uma característica desta abordagem é que interpenetrações detectadas num estagio inicial, inferem contatos num segundo estágio [18].

Já no contexto de modelos deformáveis, o primeiro artigo que trata deformações com restrições é de Provot [71]. Mais tarde Faure [72] estende esta abordagem para

animação de corpos articulados. Para simulação de objetos elásticos a abordagem apresenta por Jakobsen [73] tornou-se popular, por sua simplicidade, sendo adotada em aplicações interativas. Outras abordagens interessantes são encontradas em [73–75].

Recentemente a interação entre tipos de objetos diferentes tem despertado interesse na comunidade de pesquisadores, e diversos métodos foram apresentados:

- simulação de materiais granulares [16];
- simulação híbrida de objetos rígidos e deformáveis [3, 76, 77];
- acoplamento de objetos rígidos e líquidos [24, 78];
- acoplamento de objetos deformáveis e líquidos [79, 80];
- acoplamento de líquidos, objetos rígidos e objetos deformáveis [62];
- simulação unificada de objetos sólidos (cordas, tecidos e sólidos) [63, 81].

2.4.1 Simulação baseada em física fazendo uso de GPUs

Atualmente um processador gráfico (GPU) possui maior poder computacional que qualquer processador central (CPU) quando se trata de processar dados não dependentes, por exemplo, dados representados por matrizes. Em alguns casos, esse tipo de dados, pode ser computado várias dezenas de vezes mais rápido, dependendo da implementação. Por tal motivo, as GPUs tornaram-se uma importante ferramenta para a implementação de aplicações que precisam computar dados em formato de matrizes. Na área de simulação física o interesse é ainda maior, já que existe uma grande demanda de aplicações que possam ser executadas em tempo real. A etapa de detecção de interferência gasta um tempo considerável durante a simulação e quando se quer simular cenários com grande quantidade de objetos torna-se inviável. Portanto é necessário buscar métodos mais eficientes ou arquiteturas que permitam processamento de dados em paralelo.

Inicialmente as GPUs foram usadas para detecção de colisão, renderizando pares de objetos e inspecionando o buffer de profundidade para determinar se há ou não interferência [5]. Técnicas mais eficientes que permitem tratar vários objetos simultaneamente são apresentadas em [8, 9, 45]. Recentemente, técnicas sofisticadas que lidam com objetos de forma arbitrária [10, 46, 57] foram apresentadas. Por outro lado, com o propósito de obter melhor desempenho na execução, técnicas que permitem executar completamente todos os estágios de uma simulação na GPU foram apresentadas em [2, 3, 17, 24, 26, 60, 82].

Capítulo 3

Duas abordagens para simulação baseada em física

Simulação baseada em física é uma abordagem que tem a intenção de gerar animações realistas para toda classe de objetos, consistindo principalmente em: simulação de sistemas de partículas, simulação de objetos rígidos, simulação de objetos deformáveis, simulação de fluidos e o acoplamento destes.

Neste capítulo são descritos dois métodos de animação baseada em física que foram implementados com o propósito de fazer testes e comparações: o primeiro trata objetos rígidos e o segundo objetos deformáveis.

3.1 Simulação de objetos rígidos

O estudo da simulação física envolvendo objetos rígidos foi foco de diversas pesquisas por um longo tempo, entre elas as contribuições de Baraff et al. [83, 84], Popovic et al. [85], Guendelman et al. [18], entre outras.

O método descrito nesta seção baseia-se na técnica apresentada por Guendelman et al. [18] e que tornou-se bastante popular, sendo usada em diversas bibliotecas de física para jogos. Esta técnica permite uma simulação de objetos rígidos com resultados visualmente plausíveis, suportando múltiplos contatos e configurações com objetos empilhados.

O sucesso da técnica pode ser atribuída à combinação adequada das partes que a compõem: representação geométrica, detecção de interferência, integração, detecção de colisões, cálculo de contatos, grafo de contato, propagação de choque e a separação.

3.1.1 Representação do objeto

A representação do objeto compreende a representação da superfície e suas propriedades, tipicamente massa, centro de massa, densidade e tensor de inércia. A superfície do objeto usualmente é representada por uma malha triangulada e as propriedades são pré-computadas a partir dela. Para facilitar os cálculos, o espaço do objeto é configurado para que o centro de massa coincida com a origem e para que os eixos X, Y e Z estejam alinhados com os eixos principais do tensor de inércia. Observe que nesta configuração o tensor de inércia deve ser uma matriz diagonal, o que permite simplificar cálculos durante a fase de integração.

Com o objetivo de acelerar o processo de detecção de colisão, para cada objeto é computada uma função de distância que permite fazer consultas rápidas de colisão e interferência. Esta função pode ser representada por uma *grade* ou uma *octree*.

3.1.2 Detecção de interferência

O processo de detecção de colisão, como descrito no capítulo de revisão bibliográfica, compreende duas etapas: a primeira visa detectar pares de objetos em colisão potencial e a segunda que executa verificações de colisão exatas. Por simplicidade, para a etapa de filtragem grosseira, escolheu-se usar esferas envolventes permitindo verificação $\mathrm{O}(N^2)$ entre todos os objetos. Naturalmente, esta etapa pode ser melhorada usando métodos mais eficientes.

Já a etapa de detecção de colisão exata visa obter uma lista de pontos de colisão. Portanto, é usada a função de distância associada a cada objeto, que permite obter os vetores normais à superfície nos pontos de contato (ver a Figura 3.1). Posteriormente, as normais e pontos de contato são usados no cálculo de impulsos.

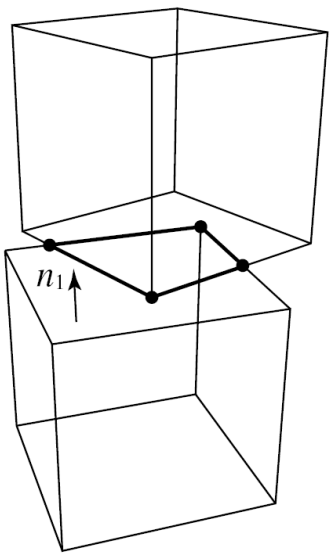


Figura 3.1: Contatos entre objetos empilhados.

Para cada par de objetos A e B em colisão potencial, os vértices $v_A(v_B)$ do objeto $A(B)$ são avaliados com a função de distância do objeto $B(A)$. Se este retorna sinal negativo, então o vértice $v_A(v_B)$ está dentro do objeto $B(A)$ e portanto, o vértice é inserido na lista de pontos em colisão. Esta técnica não é suficiente para detectar todas as colisões. Por exemplo, arestas compridas que possuem vértices muito distantes podem eventualmente se interpenetrar (veja a Figura 3.2). Para tratar este problema, nas arestas são atribuídos pontos extras que também são verificados para o caso de interferência.

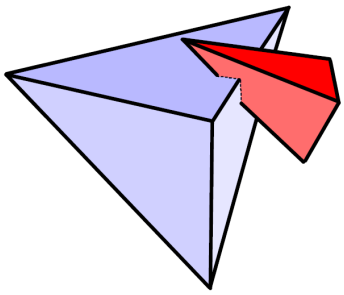


Figura 3.2: Interferência entre objetos com arestas não proporcionais.

Outro problema inerente da integração está relacionado com o intervalo de tempo (Δt) empregado. Por exemplo, se Δt é grande, objetos pequenos podem se atravessar. Este problema é tratado limitando o intervalo de tempo baseado nas velocidades (translacional e rotacional) e no tamanho dos volumes envolventes dos objetos sendo simulados.

3.1.3 Integração

No processo de integração, para cada objeto são usadas as equações:

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}, \quad \frac{d\mathbf{q}}{dt} = \frac{1}{2}\omega\mathbf{q}, \tag{3.1}$$

$$\frac{d\mathbf{v}}{dt} = \frac{\mathbf{f}}{m}, \quad \frac{d\mathbf{L}}{dt} = \tau, \tag{3.2}$$

onde $\mathbf{x}$ e $\mathbf{q}$ [1] são sua posição e a orientação, $\mathbf{v}$ e ω são suas velocidades linear e angular, $\mathbf{f}$ e τ são a força e o torque aplicadas sobre ele, m é sua massa e $\mathbf{L} = \mathbf{I}\omega$ é seu momento angular que depende de seu tensor de inércia. O tensor de inércia para cada passo de tempo é computado via $\mathbf{I} = \mathbf{R}\mathbf{I}_{body}\mathbf{R}^T$, onde $\mathbf{R}$ é a matriz de orientação e $\mathbf{I}_{body}$ é o tensor de inércia inicial. Como mencionado anteriormente, $\mathbf{I}_{body}$ é uma matriz diagonal, o que simplifica os cálculos. Existem diversos métodos para integração e seu uso depende do tipo de aplicação. Em particular, para uma

[1] $\mathbf{q}$ é um quatérnio

simulação com muitos objetos e múltiplos contatos é preferível usar o método de integração "para frente"[2] de Euler:

$$\mathbf{q}_{n+1} = \mathbf{q}_n + \frac{1}{2}\Delta t \omega_n \mathbf{q}_n, \tag{3.3}$$

aplicando-o na equação 3.1 de orientação.

A idéia desta abordagem é computar o intervalo de tempo seguinte e, subsequentemente, tratar as colisões e os contatos. De modo geral, as colisões produzem impulsos, que descontinuamente modificam a velocidade, e os contatos são associados a forças e acelerações. Assim, o algoritmo de integração é combinado com o processamento de colisões e reúne os seguintes passos:

- detecção de colisão;
- atualização das velocidades usando as equações de velocidade e torque;
- resolução de contatos;
- atualização das posições usando as equações de posição e orientação.

3.1.4 Tratamento de colisões

Quando há muitos objetos em movimento, o tratamento eficiente de colisão é complexo e custoso, especialmente ao se considerar a ordem cronológica. O método apresentado por Guendelman et al. [18] resolve simultaneamente todas as colisões. Embora esta técnica não forneça os mesmos resultados que uma resolução de colisões em ordem cronológica, ela conduz a uma solução fisicamente plausível. As colisões são detectadas prevendo onde os objetos estarão no passo de tempo seguinte. Assim, os objetos são movidos temporariamente no passo de tempo seguinte e verifica-se a interferência. A mesma técnica é usada para detectar contatos e, se não há colisão, é desejável que os objetos sejam deslocados para a mesma posição em ambas as etapas. Para garantir este resultado, são usadas as novas velocidades para predizer as posições dos objetos em ambas as etapas. Por outro lado, enquanto as velocidades antigas são usadas para resolver as colisões, as velocidades novas são usadas para resolver os contatos. Por exemplo, para a fase de colisão, se a posição e a velocidade de um objeto são $\mathbf{x}$ e $\mathbf{v}$, a verificação de interferência é feita usando a posição predita $\mathbf{x}' = \mathbf{x} + \Delta t(\mathbf{v} + \Delta t \mathbf{g})$ e são aplicados impulsos utilizando a velocidade corrente. Durante o processamento de contatos, é usada a posição predita $\mathbf{x}' = \mathbf{x} + \Delta t \mathbf{v}'$ e impulsos são aplicados a esta nova velocidade $\mathbf{v}'$.

[2]Em inglês, forward integration method.

A estrutura do algoritmo consiste em mover todos os objetos rígidos para suas posições preditas e só então identificar e processar pares de objetos em colisão potencial. Como as velocidades dos objetos mudaram devido às colisões, entretanto, podem ocorrer novas colisões entre pares de objetos que não existiam anteriormente. Por conseguinte, o processo todo é repetido um número pequeno de vezes, por exemplo 5.

Para obter um resultado mais natural, a cada iteração os pares de objetos em colisão potencial são coletados numa lista e embaralhados de forma a evitar um processamento similar à iteração anterior.

Num ponto de colisão, a velocidade local de um objeto é $\mathbf{u} = \mathbf{v} + \omega \times \mathbf{r}$, onde $\mathbf{r}$ é o vetor que representa a **posição relativa** do ponto de colisão em relação ao centro de massa. Aplicando um impulso neste ponto de colisão, as velocidades linear e angular mudam de acordo com

$$\mathbf{v}' = \mathbf{v} + \frac{\mathbf{j}}{m} \tag{3.4}$$

e

$$\omega' = \omega + \mathbf{I}^{-1}(\mathbf{r} \times \mathbf{j}). \tag{3.5}$$

Logo, a nova velocidade no ponto de contato é

$$\mathbf{u}' = \mathbf{u} + K\mathbf{j}, \tag{3.6}$$

onde $K = \frac{\mathbf{1}}{m} + \mathbf{r}^{*T}\mathbf{I}^{-1}\mathbf{r}^*$, 1 é a matriz de identidade, e "*" indica a matriz simétrica[3] associada ao vetor. As velocidades nos pontos de contato serão utilizados mais tarde no tratamento de atritos estático e cinético.

3.1.5 Tratamento de contatos

Após ter processado todas as colisões, os objetos apresentam um comportamento plausível, porém, ainda podem existir colisões. Nesse caso é executado o processo de resolução de contatos. O objetivo deste processo é resolver as forças entre os objetos. Primeiro, as velocidades dos objetos são atualizadas e, como no processo de resolução de colisões, os contatos são detectados predizendo a posição dos objetos no passo de tempo seguinte, ignorando as forças de contato. A Figura 3.3 ilustra uma cena com objetos empilhados mostrando o processo de resolução de contatos. Após iniciada a simulação, pode-se ver que apenas a caixa de baixo terá interferência. Entretanto, após processar as forças nesta caixa, ela colidirá com a caixa

[3]Um vetor $\mathbf{u}$ possui uma matriz associada $\mathbf{u}^* = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_z & 0 \end{bmatrix}$, de forma que $\mathbf{u}^*\mathbf{v} = \mathbf{u} \times \mathbf{v}$.

de cima e, assim sucessivamente até a última caixa no topo da pilha. Em geral, este processo de propagação separa todos os objetos após algumas iterações. Naturalmente, em algumas situações o processo pode não convergir, como no caso de objetos empilhados. Este problema é tratado usando um grafo de contato.

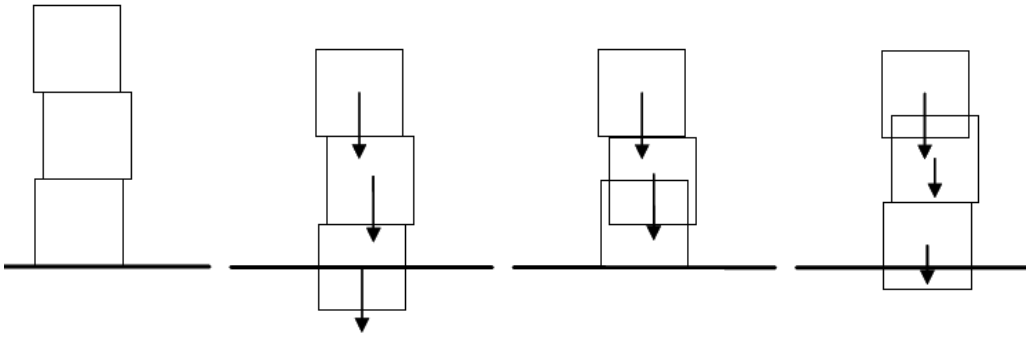


Figura 3.3: Exemplo de objetos empilhados.

3.1.6 Grafo de contato

Após a atualização de velocidades dos objetos é construído um grafo com o propósito de identificar objetos ou grupos de objetos que estão sobre outros objetos e determinar uma ordem apropriada. Os seguintes passos são necessários:

- construir o grafo. Para cada objeto i, computa-se uma nova posição usando $\mathbf{v}' = \mathbf{v} + \Delta t \mathbf{g}$, sendo as posições dos demais objetos ($j \neq i$) computadas usando as velocidades antigas. Adicionalmente, para cada objeto j intersectado por i nas posições candidato, criar uma aresta direcionada $j \leftarrow i$ indicando que i está sobre j;
- eliminar ciclos. O conjunto de pares no grafo deve ser único;
- atribuir uma ordem. Usando um método de ordenação topológica, atribuir a cada par de objetos um nível para o processamento de contato.

As Figuras 3.4 (a) e (b), mostram exemplos de grafos para duas cenas.

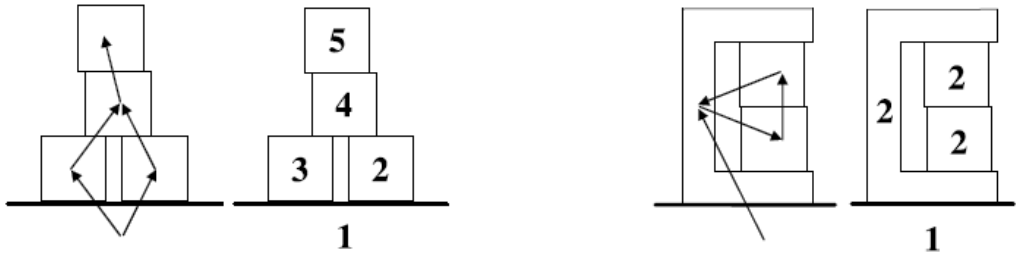


Figura 3.4: Grafo de contato para cenas com objetos empilhados.

3.1.7 Propagação de choque

Mesmo com a ajuda do grafo de contato, o modelo de propagação para os contatos requer muitas iterações para produzir um resultado visualmente realista, especialmente em situações onde há muitos objetos empilhados. A técnica de propagação de choque (Figura 3.5) inicia com os objetos de baixo, ou seja, objetos de menor nível no grafo de contatos. Logo, objetos processados terão massa infinita e velocidade zero evitando que objetos de cima empurrem objetos de baixo.

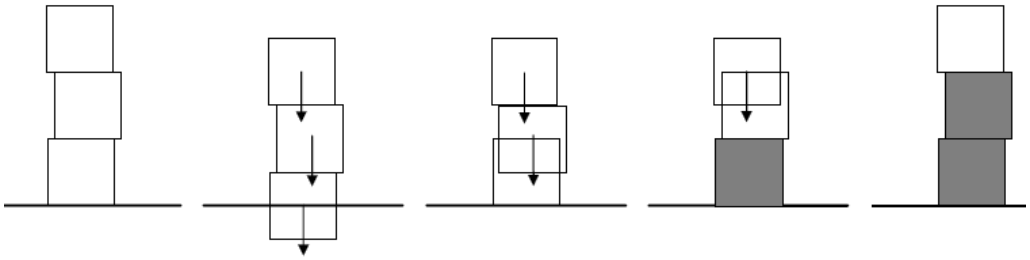


Figura 3.5: contatos entre objetos empilhados.

3.1.8 Separação dos objetos

O processo de separação segue com o cálculo das posições após terem sido feitos o tratamento de colisões e o tratamento de contatos usando o grafo de contatos e a propagação de choque. Para reduzir a margem de erro, a cada nível do grafo os objetos são separados gradualmente incrementando a fração de interpenetração. Em alguns casos ainda podem existir colisões por erros de arredondamento e dependências cíclicas no grafo.

Em resumo, a sequência de passos durante a simulação é:

1. detectar interferência(colisões) e aplicar impulsos;
2. computar um grafo de contato;
3. integrar as velocidades usando forças externas;
4. detectar interferência(contatos) e aplicar impulsos de contato inelástico;
5. aplicar propagação de choque;
6. integrar as posições.

3.2 Simulação de objetos deformáveis

Esta seção foca uma metodologia simplificada para animação de objetos deformáveis. Os objetos são representados por malhas tetraedrais, sendo seu uso fundamental no processo de detecção e resposta às colisões.

A metodologia combina várias técnicas, a saber:

- detecção de colisão grosseira (*broad phase*) baseada no uso das esferas envolventes dos objetos;
- detecção de colisão exata empregando uma grade regular e uma tabela de dispersão (*hash table*) conforme a abordagem apresentada por Teschner [19];
- resposta à colisões feita computando a profundidade de penetração [20] de todos os vértices colididos. Colisões assimétricas são tratadas usando a técnica de *tratamento de contato por projeção* de Jakobsen [21]. Finalmente, os objetos são separados usando a profundidade de penetração nos vértices colididos;
- dinâmica baseada na técnica de casamento de formas de Müller [1];
- integração empregando um método semi-implícito de Euler [1].

3.2.1 Representação dos objetos

Os objetos sao representados por malhas tetraedrais, principalmente para propósitos de processar adequadamente respostas às colisões. A malha tetraedral para cada objeto foi gerada empregando um triangulador 3D [4] baseado na triangulação de

3.2.2 Pré-processamento

Em primeiro lugar, é necessário definir uma grade regular que subdivide o espaço R^3, representando o mundo onde os objetos estão posicionados e onde se espera que colisões aconteçam. A grade é uma subdivisão do espaço em células de mesmo tamanho, onde cada célula aloja tetraedros dos objetos. Assim, o método detecta colisões fazendo uma busca nas células que intersectam os tetraedros dos objetos.

Ao invés de alocar explicitamente a grade, entretanto, um esquema mais eficiente consiste no mapeamento de suas células numa tabela menor usando uma função de dispersão (*hash function*). Assim, a estrutura de dados depende dos seguintes parâmetros:

[4] 3DMesher, de propriedade do LCG (Laboratório de Computação Gráfica do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ)

tamanho da tabela de espalhamento. O tamanho ótimo está relacionado com o número de primitivas (tetraedros) na cena e deve ser um número primo grande, para minimizar colisão de endereços na tabela;

tamanho da célula da grade. Deve estar relacionado com o tamanho dos tetraedros. Uma escolha razoável é empregar o comprimento médio das arestas dos tetraedros;

função de dispersão. Para o mapeamento eficiente das células na tabela de hash dispersão, uma função eficiente deve ser escolhida. Uma função que fornece boa distribuição é a seguinte:

$$h = hash(i, j, k) = (i\, c_i \oplus j\, c_j \oplus k\, c_k) \,\mathbf{mod}\, n, \tag{3.7}$$

onde $\oplus$ é a operação ou-exclusiva bit-a-bit, i, j e k são as coordenadas da célula da grade, c_i, c_j e c_k são números primos grandes e n é o tamanho da tabela de dispersão.

Adicionalmente, para cada objeto sua esfera envolvente mínima é computada. As esferas envolventes são usadas no estágio de detecção de colisão grosseira.

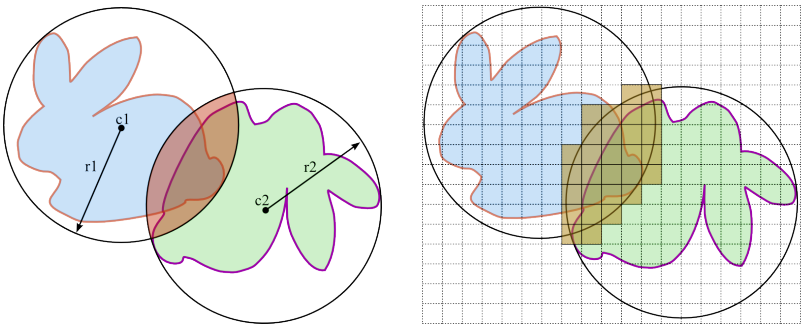
3.2.3 Detecção de colisões

Detecção de colisão grosseira

No método apresentado empregam-se esferas limitantes computadas para cada objeto. O objetivo desta fase é selecionar regiões na grade onde possa haver colisões. Usando as esferas limitantes, uma simples verificação é feita. Para cada par de objetos A e B, há colisão potencial se a distância entre os centros de suas esferas é menor que a soma de seus raios, isto é,

$$|c_1 - c_2| < r_1 + r_2. \tag{3.8}$$

Para cada par de objetos, os vértices envolvidos na região de colisão potencial ($v \subset A \cap B$) são coletados, já que serão usados para a detecção de colisão exata. Os vértices em colisão potencial são encontrados usando uma verificação de ponto dentro de esfera (veja a Figura 3.6).



(a) (b)

Figura 3.6: A verificação de interferência na etapa de filtragem grosseira coleta vértices dentro das esferas envolventes (a); a detecção de colisão exata é feita apenas nas células em colisão potencial (b).

Este processo é especialmente útil na atualização eficiente da grade. Apenas tetraedros nas regiões em colisão potencial são atualizados. Assim, um tetraedro t_A, associado ao objeto A, e cuja esfera envolvente colide com a esfera do objeto B, é atualizado apenas se ele intersecta com a esfera de B.

A eficiência desta técnica é auxiliada pelo uso de uma variável chamada *timestamp* (instante de tempo). A idéia é evitar uma atualização da grade a todo passo de tempo. Para tanto, quando a célula é atualizada, o timestamp é atualizado com o tempo corrente. Assim, a detecção de colisão exata entre primitivas (vértices e tetraedros) numa célula é realizada apenas se esta foi atualizada no tempo corrente.

Detecção de colisão exata

Uma vez que as regiões em colisão potencial são encontradas, primitivas que participam são enviadas para uma verificação exata. A verificação consiste em usar a função de dispersão que visita a célula associada ao vértice em colisão potencial para verificar se o mesmo está no interior de algum tetraedro na célula.

3.2.4 Resposta às colisões

O processo de resposta a colisões inicia-se computando a profundidade de penetração de todos os vértices colididos. Em seguida, regiões de deformação para todos os objetos são computadas. Finalmente, os objetos são separados aproximadamente na metade da profundidade de penetração dos vértices colididos usando uma técnica de busca binária.

Profundidade de penetração

A profundidade de penetração entre objetos que se sobrepõem é a mínima translação que deve ser aplicada para separá-los. De modo geral, a resposta à colisão de um par

de objetos pode ser computada a partir dessa informação. Entretanto, para objetos deformáveis, deve ser computada a profundidade de penetração de todos os vértices colididos. A malha tetraedral é usada para auxiliar esta tarefa, já que a informação de incidência pode ser usada para determinar a região de deformação e computar as forças de penalidade, permitindo assim uma resposta realista à colisão.

A idéia é processar os vértices colididos, classificando-os pelas profundidades de penetração. Primeiramente, os vértices próximos à superfície do objeto interpenetrado são avaliados. Em seguida, esta informação é propagada, usando a informação de incidência da malha tetraedral, para os vértices mais profundos. Como resultado, a profundidade de penetração d e a direção $\vec{r}$ são estimadas para todos os vértices em colisão.

Classificação dos vértices colididos: se um vértice colidido tem um ou mais vértices incidentes não colididos, este é um *vértice borda*, caso contrário, é um *vértice interno* (veja a Figura 3.7).

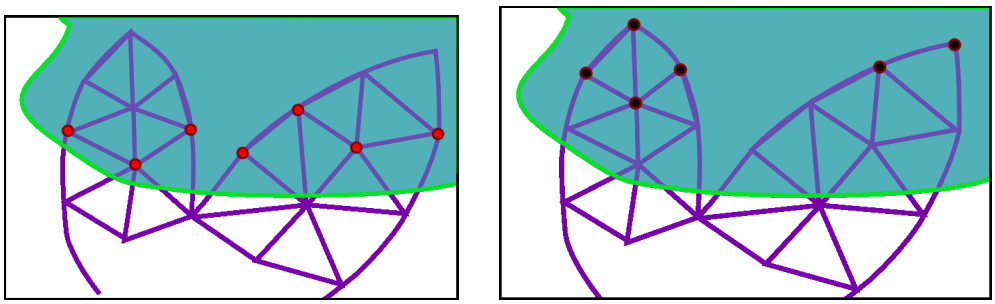


Figura 3.7: Vértices borda (esquerda) e vértices internos (direita).

Profundidade de penetração dos *vértices borda*: a computação da profundidade de penetração dos *vértices borda* depende do cálculo de seus *pontos de contato* e das normais nestes pontos. A idéia consiste em identificar arestas que possuem um vértice colidido e o outro não. Estas arestas são chamadas de *arestas de interseção*. Além disso, o ponto exato de interseção da aresta com a superfície do objeto interpenetrado deve ser encontrado. Este processo pode ser feito eficientemente visitando as células da grade intersectadas pela aresta e então verificar a interseção com todos os triângulos da superfície. O teste entre triângulos-arestas é feito em dois passos. Primeiro, é verificado se o plano de suporte do triângulo corta a aresta. Caso positivo, as coordenadas baricêntricas (t_1, t_2, t_3) do ponto de interseção p são computadas. Então, p está dentro do triângulo se $t_i > 0$ para $i = 1, 2, 3$. Adicionalmente, o vetor normal à superfície $\vec{n}$ é interpolado linearmente a partir das normais dos vértices do triângulo $(\vec{n}_1, \vec{n}_2, \vec{n}_3)$, isto é, $\vec{n} = \sum t_i \vec{n}_i$.

Se uma aresta de interseção corta mais de um triângulo, é escolhido o ponto mais próximo do vértice colidido. Os pontos de interseção são chamados de *pontos*

de contato. Finalmente, estes dados são usados para computar a profundidade de penetração dos vértices da borda (veja a Figura 3.8).

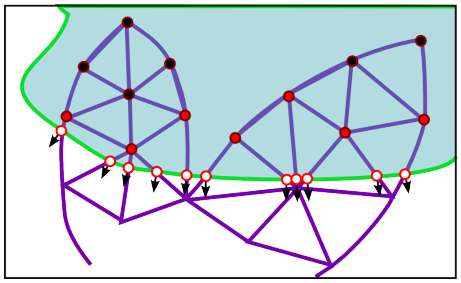


Figura 3.8: Pontos de contato com suas normais nas faces intersectadas.

Para cada vértice v, sua profundidade de penetração $d(v)$ e sua direção de penetração $\vec{r}(v)$ são computadas da seguinte maneira: para cada aresta de interseção e_i incidente em v, um peso w_i entre o ponto de contato p_i e v é dado por:

$$w(p_i, v) = \frac{1}{\|p_i - v\|^2}. \tag{3.9}$$

Logo, $d(v)$ e $\vec{r}(v)$ são computados usando as equações:

$$d(v) = \frac{\sum_{i=1}^{k} w(p_i, v)(p_i - v) \cdot \vec{n}_i}{\sum_{i=1}^{k} w(p_i, v)}, \tag{3.10}$$

$$\vec{r}(v) = \frac{\sum_{i=1}^{k} w(p_i, v)\vec{n}_i}{\sum_{i=1}^{k} w(p_i, v)}, \tag{3.11}$$

onde k é o número de arestas de interseção incidentes em v e p_i e $\vec{n}_i$ são o i-ésimo ponto e normal a serem avaliados, respectivamente.

Profundidade de penetração dos vértices internos: após o processamento de todos os vértices da borda, a profundidade de penetração destes vértices é empregada para computar a profundidade de penetração dos vértices internos u_i. Este processo é feito por propagação, isto é, uma vez que a primeira camada de vértices colididos é processada (*vértices borda*), os vértices internos vizinhos por aresta são processados. Assim, a computação da profundidade de penetração é feita por níveis, até que a informação de penetração seja determinada para todos os vértices colididos.

Dado um vértice interno u, sua profundidade de penetração $d(u)$ e sua direção de penetração $\vec{r}(u)$ são computadas usando pesos. Primeiro, para um vértice interno não processado u, incidente num vértice já processado v_j, um peso é computado segundo:

$$\mu(u, v_j) = \frac{1}{\|v_j - u\|^2}. \tag{3.12}$$

Logo, $d(u)$ e $\vec{r}(u)$ são computados usando a média ponderada das contribuições de todos os vértices incidentes processados segundo as equações:

$$d(u) = \frac{\sum_{j=1}^{k} \mu(u, v_j)((v_j - u) \cdot \vec{r}(v_j) + d(v_j))}{\sum_{j=1}^{k} \mu(u, v_j)}, \tag{3.13}$$

$$\vec{r}(u) = \frac{\sum_{j=1}^{k} \mu(u, v_j)\vec{r}(v_j)}{\sum_{j=1}^{k} \mu(u, v_j)}, \tag{3.14}$$

onde k é o número de vértices processados incidentes em u. A Figura 3.9 mostra as direções de penetração dos vértices colididos.

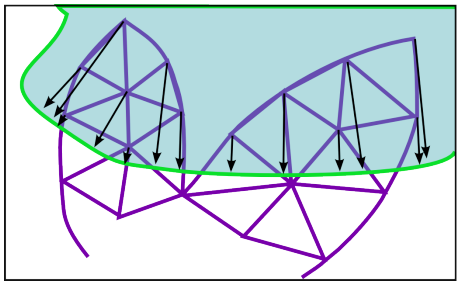


Figura 3.9: Direções de penetração para os vértices colididos.

Adicionalmente, é necessário encontrar um *triângulo de contato* para cada *vértice borda* v. Este triângulo é uma face na superfície do objeto interpenetrado, que intersecta o raio com origem em v e a direção $\vec{r}_v$.

3.2.5 Região de deformação

Após a computação da profundidade de penetração para todos os vértices colididos, é executado o processo de separação para obter uma situação sem interseção. Para este propósito, calcula-se uma região de deformação que é definida pelos vértices em colisão e os vértices das faces de contato (triângulos na superfície), que não necessariamente são vértices em colisão (x_i, x_j e x_k na Figura 3.10). Colisões com esta configuração são chamadas de colisões assimétricas.

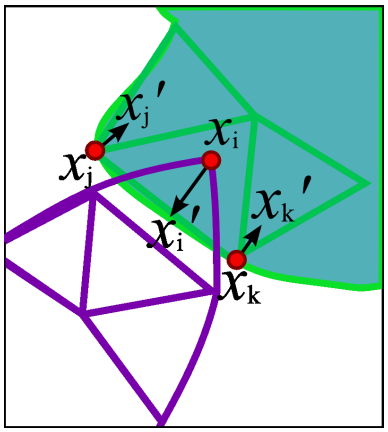


Figura 3.10: Tratamento de colisões assimétricas.

O tratamento deste tipo de colisão nesta abordagem diverge do método relativamente complexo proposto em [86]. Emprega-se aqui a técnica simplificada para manipulação de colisões por projeção descrita em [21].

A idéia consiste em projetar vértices em colisão fora do obstáculo, movendo-os até alcançarem um estado livre de interseção. Por exemplo, na Figura 3.10, o vetor de deslocamento de x_i é a sua profundidade de penetração e os vetores de deslocamento dos vértices não colididos (x_j e x_k) são:

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

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

onde x_i é o vértice em colisão, x_i' é sua projeção na aresta de contato x_j-x_k, $\vec{s_j}$ e $\vec{s_k}$ são os vetores de deslocamento para x_j e x_k respectivamente, e α_1 e α_2 representam os fatores proporcionais de deslocamento para os vértices na aresta de contato, sujeito a $\alpha_1 + \alpha_2 = 1$. Note que uma aresta de contato em 2D corresponde a um triângulo de contato em 3D e as constantes α_i são áreas proporcionais. Por exemplo, se o triângulo de contato tem seus vértices x_k, x_j e x_l e x_i' é a projeção do vértice colidido no triângulo, então $A = \mathit{Area}(x_j, x_k, x_l)$, $\alpha_1 = \mathit{Area}(x_k, x_l, x_i')/A$, $\alpha_2 = \mathit{Area}(x_j, x_l, x_i')/A$ e $\alpha_3 = \mathit{Area}(x_j, x_k, x_i')/A$.

Uma vez que todos os vetores de deslocamento para os vértices em colisão tenham sido computados, uma superfície de contato é definida. Conceitualmente, a superfície de contato deveria ser equidistaste dos vértices em colisão e, assim, uma estimativa inicial de suas posições é obtida transladando os vértices colididos até a metade do comprimento de seus vetores de deslocamento. Esta estimativa é refinada por meio de um esquema de busca binária [86]: se x^1 é a estimativa inicial para um vértice na superfície, e s é seu vetor de deslocamento associado, então a i-ésima estimativa

para esse vértice é dado por:

$$x^i \leftarrow x^{i-1} \pm \frac{1}{2^{(i+1)}} s, \tag{3.15}$$

onde a direção do movimento é determinada de forma a aproximar o vértice de superfície de contato (ver Figura 3.11). Na prática, três ou quatro iterações são suficientes para produzir um superfície de contato adequada.

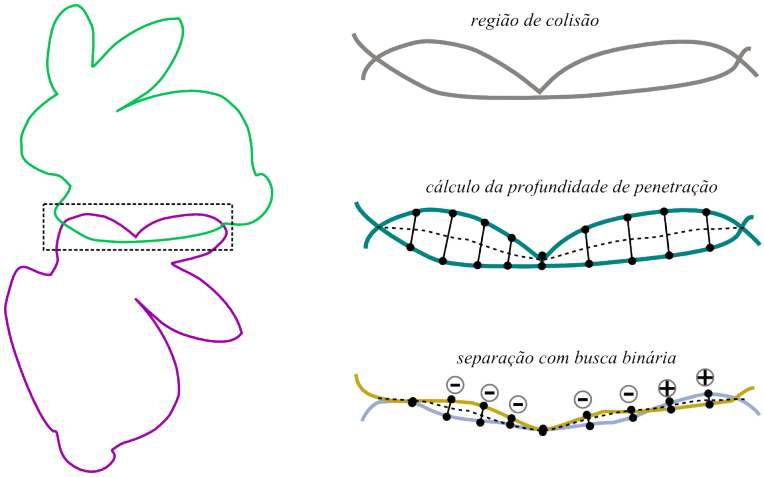


Figura 3.11: Determinação da superfície de contato.

3.2.6 Dinâmica baseada em casamento de formas

Para estimar o comportamento dinâmico dos objetos colididos, a técnica de casamento de formas descrita em [1] é usada. A idéia é fazer corresponder vértices em seu estado inicial $\mathbf{x}_i^0$ com as posições no estado corrente $\mathbf{x}_i$, isto é, a posição dos vértices como resultado do movimento ou uma deformação. O casamento de formas consiste em determinar uma matriz de transformação afim usando otimização por mínimos-quadrados. Esta transformação é um modelo para uma deformação plausível[5] e proporciona uma form.a simples para que os objetos recuperem suas formas originais (veja a Figura 3.12). Uma vez que a transformação é computada, as posições $\mathbf{g}_i$ alvo são computadas para cada vértice. Estas são usadas para computar a nova velocidade e a nova posição de cada partícula para o passo seguinte de tempo.

[5]Plausível, neste contexto, significa que pode ser aceitável mas não necessariamente correto.

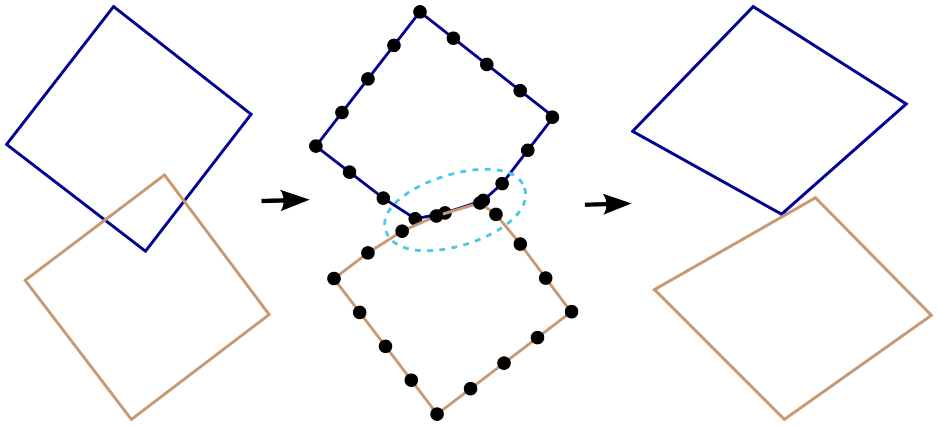


Figura 3.12: (a) dois objetos em colisão, (b) depois uma superfície de contato é determinada e (c) deformada com o casamento de formas.

O algoritmo tem duas componentes principais: encontrar a transformação rígida ótima que aproxima uma nova posição e orientação para o objeto (problema de casamento de formas) e mover os vértices para a posição objetivo aplicando um modelo de deformação linear.

Considerando os pesos das partículas, o problema de casamento de formas consiste em encontrar vetores de translação $\mathbf{t}$ e $\mathbf{t}_0$ e a matriz de rotação $\mathbf{R}$ que minimiza:

$$\sum_i w_i(\mathbf{R}(x_i^0 - \mathbf{t}_0) + \mathbf{t} - \mathbf{x}_i)^2. \tag{3.16}$$

Para simplificar, é usado $w_i = m_i$, onde m_i é a massa da partícula i. Os vetores de translação ótimos entre os centros de massa da posição inicial e da posição atual são dados por

$$\mathbf{t}_0 = \mathbf{x}_{cm}^0 = \frac{\sum_i m_i \mathbf{x}_i^0}{\sum_i m_i} \quad e \quad \mathbf{t} = \mathbf{x}_{cm} = \frac{\sum_i m_i \mathbf{x}_i}{\sum_i m_i}. \tag{3.17}$$

A rotação ótima $\mathbf{R}$ pode ser encontrada determinando uma matriz de transformação linear $\mathbf{A}$. Note que $\mathbf{A}$ não necessariamente é ortonormal. Considerando as posições relativas das partículas definidas por:

$$\mathbf{q}_i = \mathbf{x}_i^0 - \mathbf{x}_{cm}^0 \quad e \quad \mathbf{p}_i = \mathbf{x}_i - \mathbf{x}_{cm}, \tag{3.18}$$

uma seleção para o problema de correspondência pode ser estimado por mínimos quadrados. Assim, a matriz $\mathbf{A}$ é encontrada minimizando o termo

$$\sum_i m_i(\mathbf{A}\mathbf{q}_i - \mathbf{p}_i)^2. \tag{3.19}$$

Igualando a zero as primeiras derivadas desta expressão em relação aos coefici-

entes de $\mathbf{A}$, obtemos:

$$\mathbf{A} = (\sum m_i \mathbf{p}_i \mathbf{q}_i^T)(\sum m_i \mathbf{q}_i \mathbf{q}_i^T)^{-1} = \mathbf{A}_{pq}\mathbf{A}_{qq}, \tag{3.20}$$

onde $\mathbf{A}_{pq}$ é uma matriz de correlação e $\mathbf{A}_{qq}$ é uma matriz simétrica que contém apenas informação de escala. Por conseguinte, para encontrar $\mathbf{R}$, é necessário encontrar a componente de rotação $\mathbf{A}_{pq}$. Segundo [1], é usado um método [87] de decomposição polar para obter $\mathbf{R} = \mathbf{A}_{pq}\mathbf{S}^{-1}$, onde $\mathbf{S} = \sqrt{\mathbf{A}_{pq}^T\mathbf{A}_{pq}}$ é uma matriz simétrica. Para obter $\mathbf{S}^{-1}$, e a matriz dada por $\mathbf{A}_{pq}^T\mathbf{A}_{pq}$ é diagonalizada usando de 5 a 10 rotações de Jacobi [88].

Finalmente, todas as partículas são deslocadas para suas posições alvo computadas como:

$$\mathbf{g}_i = \mathbf{R}(\mathbf{x}_i^0 - \mathbf{x}_{cm}^0) + \mathbf{x}_{cm}, \tag{3.21}$$

e o processo é repetido para todo passo de tempo.

3.2.7 Método de integração

Como em [1], para a integração usamos um método de Euler modificado, que inclui uma parte explícita (atualização da velocidade) e uma parte implícita (atualização da posição). Assim, o esquema de integração é dado por:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \alpha\frac{\mathbf{g}_i(t) - \mathbf{x}_i(t)}{\Delta t} + \frac{\Delta t}{m_i}f_{ext}(t), \tag{3.22}$$

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t\mathbf{v}_i(t + \Delta t), \tag{3.23}$$

onde Δt é o passo de tempo, m_i é a massa, $\mathbf{v}_i(t + \Delta t)$ é a velocidade após o passo de tempo, $\mathbf{v}_i(t)$ é a velocidade atual, $\alpha \in [0, 1]$ é um parâmetro que controla a rigidez do objeto, $\mathbf{g}_i(t)$ é a posição alvo para o vértice $\mathbf{x}_i(t)$ é a massa da partícula e $\mathbf{f}_{ext}(t)$ são as forças externas (gravidade, vento, entre outras).

3.2.8 Deformação linear

O método descrito até aqui consegue apenas pequenas deformações na forma rígida do objeto. Para ter uma maior liberdade de deformação no movimento, pode-se usar a matriz de deformação linear $\mathbf{A}$. Esta matriz descreve a melhor transformação linear da forma inicial que caça com a forma atual em termos de mínimos quadrados. Em vez de usar $\mathbf{R}$ na equação 3.21 para computar $\mathbf{g}_i$, usa-se a combinação $\beta\mathbf{A}+(1-\beta)\mathbf{R}$, onde β é um parâmetro adicional de controle. A presença de $\mathbf{R}$ na soma garante que a forma do objeto não seja totalmente deformada. Para garantir que o volume

seja conservado, divide-se $\mathbf{A}$ por $\sqrt[3]{det(\mathbf{A})}$ garantindo que $det(\mathbf{A}) = 1$. Diferente da deformação rígida, onde apenas é necessário computar $\mathbf{A}_{pq}$, nesta deformação é necessário computar a matriz $\mathbf{A}_{qq} = (\sum_i m_i \mathbf{q}_i \mathbf{q}_i^T)^{-1}$. No entanto, esta matriz simétrica pode ser pre-computada.

3.2.9 Deformação quadrática

Note que o modelo de deformação apresentado na sub-seção 3.2.8 consegue apenas deformações lineares, ou seja, efeitos de estiramento e achatamento. Para obter efeitos de torção e flexão devemos re-definir as matrices de deformação para:

$$\mathbf{g}_i = \beta[\mathbf{AQM}] + (1-\beta)[\mathbf{R}\ \tilde{\mathbf{0}}\ \mathbf{0}]\tilde{\mathbf{q}}_i + \mathbf{x}_{cm}, \tag{3.24}$$

onde $\mathbf{g}_i \in \Re^3$, $\tilde{\mathbf{q}} = [q_x, q_y, q_z, q_x^2, q_y^2, q_z^2, q_x q_y, q_y q_z, q_z q_x]^T \in \Re^9$. $\mathbf{A} \in \Re^{3\times3}$ contem os coeficientes para os termos lineares, $\mathbf{Q} \in \Re^{3\times3}$ os coeficientes para os termos quadráticos puros e $\mathbf{M} \in \Re^{3\times3}$ os coeficientes para os termos misturados. Com $\tilde{\mathbf{A}} = [\mathbf{AQM}] \in \Re^{3\times9}$ deve-se minimizar $\sum_i m_i(\tilde{\mathbf{A}}\tilde{\mathbf{q}}_i - \mathbf{p}_i)^2$. A transformação quadrática ótima torna-se:

$$\tilde{\mathbf{A}} = (\sum_i m_i \mathbf{p}_i \tilde{\mathbf{q}}_i^T)(\sum i m_i \tilde{\mathbf{q}}_i \tilde{\mathbf{q}}_i^T)^{-1} = \tilde{\mathbf{A}}_{pq}\tilde{\mathbf{A}}_{qq}. \tag{3.25}$$

Novamente, a matriz simétrica $\tilde{\mathbf{A}}_{qq} \in \Re^{9\times9}$ assim como $\tilde{\mathbf{q}}_i$ podem ser pre-computadas. De forma análoga ao caso do modelo de deformação linear, pode-se variar a flexibilidade usando $\beta\tilde{\mathbf{A}} + (1-\beta)\tilde{\mathbf{R}}$ no computo dos pontos objetivo, onde $\tilde{\mathbf{R}} \in \Re^{3\times9} = [\mathbf{R}\ \mathbf{0}\ \mathbf{0}]$. A figura 3.13 mostra os tipos de deformação.

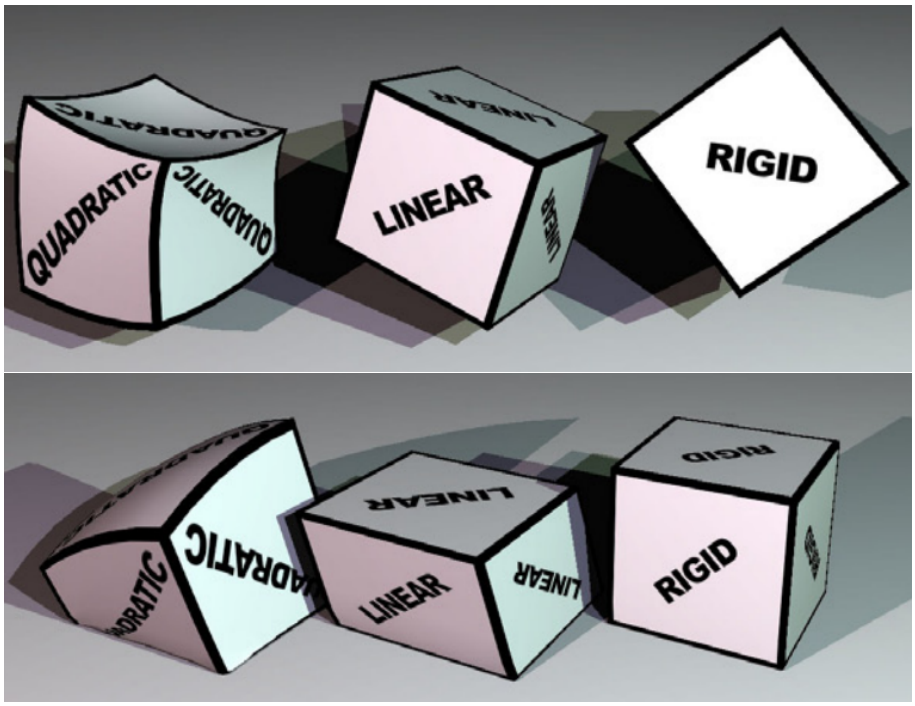


Figura 3.13: Tipos de deformação (imagem extraída de [1]).

3.2.10 Extensões

A implementação inicial, descrita na seção anterior, onde foram utilizadas malhas tetraedrais [89] para determinar a separação dos objetos após colisões, permite simular objetos deformáveis adequadamente, entretanto, simular cenas com uma quantidade grande de objetos resulta custoso e inviável para ser executado em tempo real (ver a seção 7.1.2 no capítulo de resultados). Isto motivou à procura por novas ideais e técnicas pelo que foram implementadas duas abordagens adicionais que foram apresentadas como contribuições no artigo [90].

Propagação baseada nas células de borda

Uma otimização simples é detectar o grupo de células da grade que tem faces dos objetos em colisão (ver figura 3.14). Estas células são chamadas de *células de borda*. O processo de detecção de colisão visita unicamente as células de borda, e os vértices em colisão encontrados são propagados utilizando o método previamente descrito.

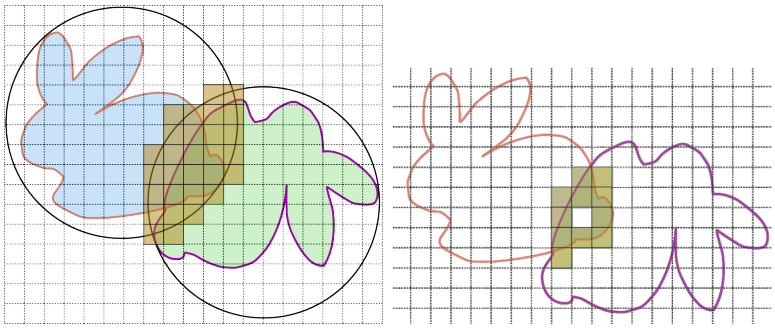


Figura 3.14: Identificando células de borda (direita) do conjunto de células em colisão potencial (esquerda) para objetos em colisão.

Usando funções de distância

Ainda com a otimização usando células de borda, a técnica de detecção de colisão utilizada demanda um tempo considerável. O cálculo da profundidade de penetração depende da resolução e qualidade da malha tetraedral, o que sugere o uso de algum método alternativo. Uma técnica que se adapta bem, neste contexto, é o uso de funções de distância [18], que mapeiam, para cada ponto p no interior do objeto, o ponto q mais próximo da superfície do objeto. Assim, é possível estimar em tempo constante a profundidade e direção do vetor de penetração para qualquer ponto no interior do objeto (ver figura 3.15). Logo, se A e B são dois objetos numa possível colisão, para cada vértice v_A (v_B) é aplicado uma transformação inversa à transformação atual do objeto B (A), que permite avaliar, no espaço do objeto B (A), se o vértice v_A (v_B) está dentro ou fora. Caso esteja dentro, há colisão e são retornados a profundidade e direção de penetração. Finalmente, estes valores são utilizados no processo de separação de objetos.

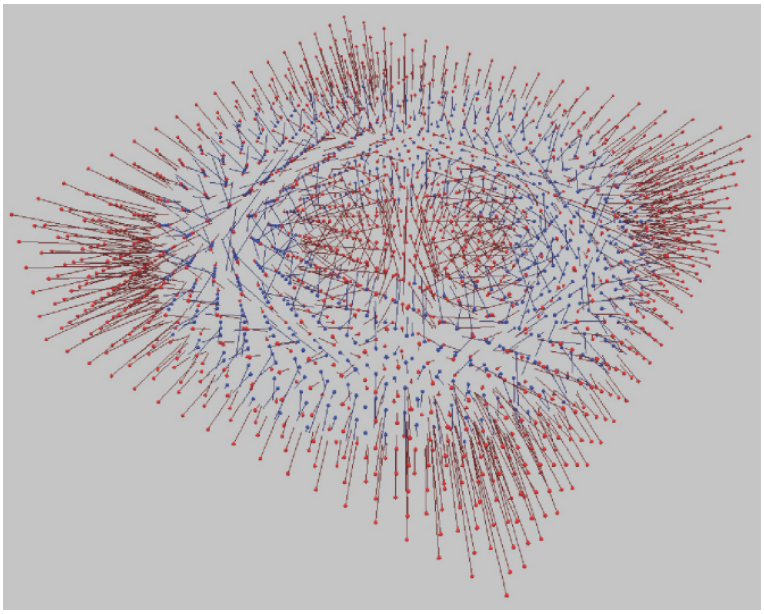


Figura 3.15: Função distância para o modelo toro definida na caixa limitante. Linhas vermelhas definem uma distância negativa (fora do modelo) e linhas azuis definem uma distância positiva (dentro do modelo).

Capítulo 4

Detecção de Colisões em GPU

Em aplicações de simulação física onde há grande quantidade de objetos, o processo de detecção de colisão frequentemente é dividido em duas etapas (descritas na seção 2.3). Atualmente, esta técnica também tem adquirido relevância em implementações em GPU, já que há uma demanda constante na manipulação de maiores quantidades de objetos em movimento. Em particular, a técnica de simulação física em GPU baseada em partículas (usada neste trabalho) requer o uso de dezenas de milhares de partículas para obter uma animação realista, e no caso de aplicações que simulam milhares de objetos complexos em movimento, uma etapa de detecção de colisão grosseira pode ser vantajoso.

Neste capítulo discutimos o uso de técnicas de detecção de colisão grosseira em paralelo, com implementações para GPUs. A primeira técnica útil, baseia-se numa grade grosseira que sub-divide o espaço ocupado pelos objetos [22]. Esta técnica é adequada para ser implementada em paralelo. Uma segunda técnica, recentemente apresentadas [40], implementa o método de varredura e poda (*sweep and prune*) [11] mas em paralelo para GPUs.

4.1 Detecção de colisão grosseira baseada em subdivisão do espaço

Uma implementação de força bruta para n objetos consiste em realizar $n(n-1)/2$ testes de colisão, tendo uma complexidade de $O(n^2)$. Portanto é necessário buscar algoritmos alternativos, tais como o de varredura e eliminação (descrito no capítulo 2) ou subdivisão espacial (descrito na sub-seção 4.1.1). Esses algoritmos podem atingir uma complexidade média $O(n \log n)$ usando um esquema de indexação, porém, sua complexidade no pior caso ainda é $O(n^2)$. Parte do trabalho realizado nesta tese baseia-se na técnica de subdivisão espacial usando grades, já que pode ser implementada em paralelo e assim aproveitar o poder computacional das GPUs [22].

4.1.1 Subdivisão espacial

Esta técnica particiona o espaço numa grade uniforme, de tal forma que uma célula da grade é maior que o diâmetro da maior esfera envolvente dos objetos. Cada célula contém uma lista de objetos cujos centróides estão contidos nela. Então, um teste de colisão entre dois objetos é executado apenas se eles pertencem à mesma célula ou a duas células adjacentes. Alternativamente, pode-se atribuir a cada célula uma lista de todos os objetos cujos volumes envolventes intersectam a célula. Neste caso, um objeto pode aparecer em até 2^d células, onde d é a dimensão da subdivisão espacial (por exemplo, $d = 3$ para um mundo 3D), conforme ilustrado na figura 4.1.

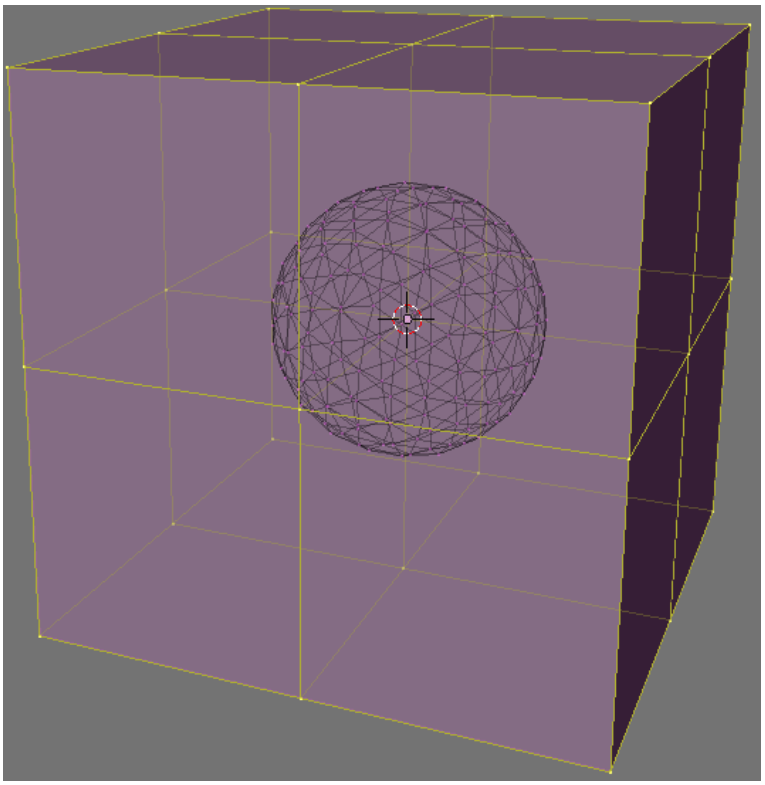


Figura 4.1: Uma esfera envolvente intersectando até oito possíveis células.

Um teste de colisão entre dois objetos é realizado apenas se eles aparecem na mesma célula e pelo menos um deles tem seu centróide nessa célula. Por exemplo, na figura 4.2, um teste de colisão é realizado entre os objetos O_1 e O_2, pois ambos têm seus centróides na célula 1, e entre os objetos O_2 e O_3 porque ambos aparecem na célula 5 e O_3 tem o seu centróide na célula 5. Mas nenhum teste de colisão é realizado entre os objetos O_2 e O_4 que aparecem na célula 2, já que seus centróides não estão na célula 2.

A implementação mais simples de subdivisão espacial consiste em mapear para cada objeto as células por ele interceptadas. É criada, então, uma lista com todos pares (objeto, célula), a qual é subsequentemente ordenada por célula. Desta forma, objetos que ocupam uma mesma célula aparecem em trechos contíguos nessa lista,

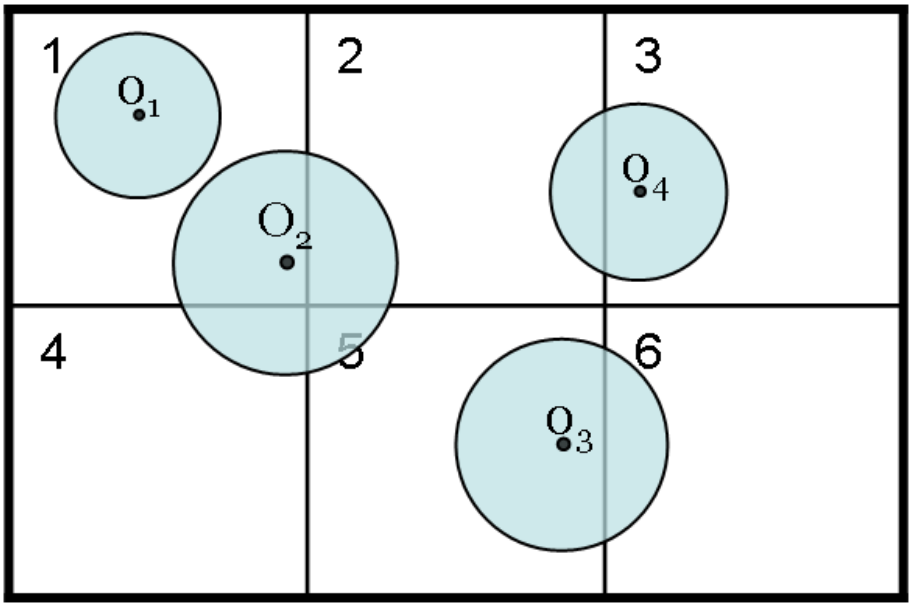


Figura 4.2: Um exemplo de subdivisão espacial 2D para quatro objetos.

podendo, então, ser testados entre si. A necessidade de que uma célula para ser maior que o volume limitante do maior objeto pode causar computação desnecessária quando há uma disparidade de tamanho entre os objetos. Um exemplo disso seria um grande planeta cercado por milhares de asteróides pequenos. Uma solução para esse tipo de problema é usar grades hierárquicas, como proposto por Mirtich [91].

4.1.2 Subdivisão espacial paralela

A paralelização do algoritmo de subdivisão espacial não é trivial como a versão sequencial, já que um objeto pode estar envolvido em mais de um teste de colisão simultaneamente, isto quando intersecta várias células que são processadas em paralelo. Portanto, deve existir um mecanismo para evitar que dois ou mais processos computacionais atualizem estados de um dado objeto ao mesmo tempo. Este problema pode ser tratado explorando a separação espacial das células. Uma vez que cada célula é garantidamente é maior que o volume limitante do maior objeto, então uma célula será processada separadamente das outras células que estão sendo processadas ao mesmo tempo. Assim, um objeto que está contido em alguma célula será atualizado uma única vez. Em 2D, para cobrir todas as células deve haver quatro passagens computacionais como ilustrado na Figura 4.3: todas as células com o mesmo rótulo numérico (de 1 a 4), ou tipo de célula, podem ser processados na mesma passada. A figura ilustra um caso simples onde o objeto O_2 tem duas colisões potenciais que devem ser tratadas separadamente. Em 3D, oito passagens são necessárias, já que há $2 \times 2 \times 2$ repetições de células numeradas de 1 a 8, conforme ilustrado na Figura 4.4.

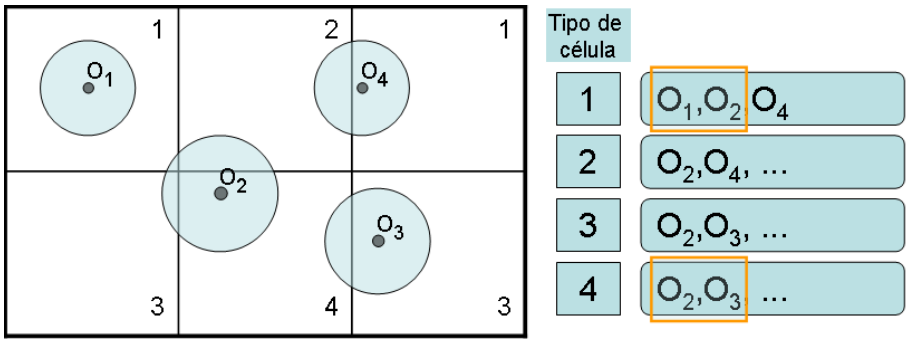


Figura 4.3: Separação de colisões potenciais em listas que correspondem ao tipo de célula.

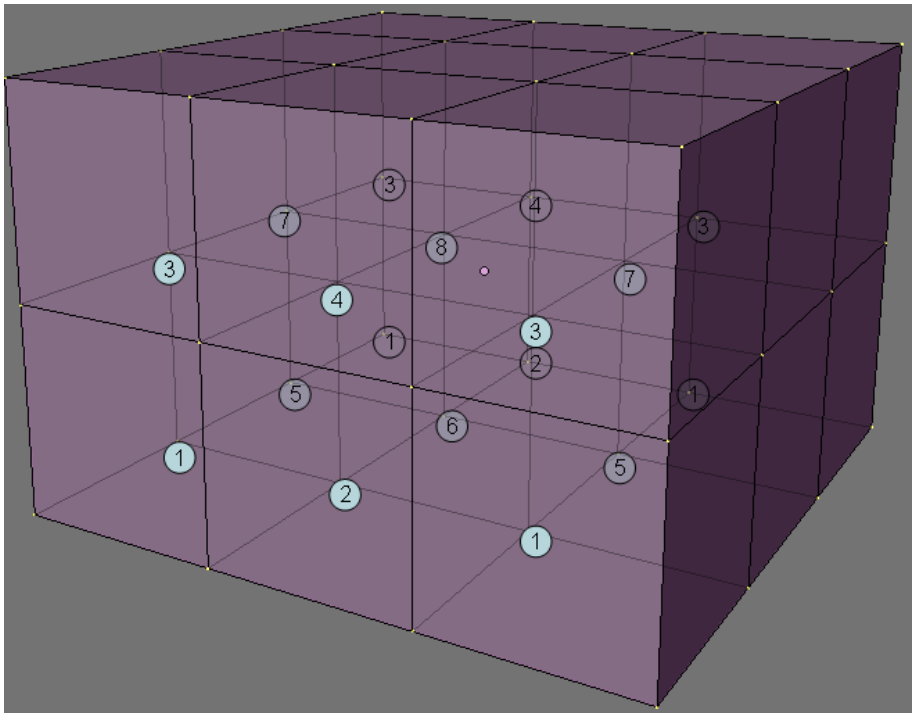


Figura 4.4: Subdivisão espacial 3D por tipo de célula.

Um problema decorrente do processamento paralelo é testar colisão entre dois objetos várias vezes. Isso pode acontecer quando dois objetos ocupam uma determinada célula, porém, seus centróides não estão nessa célula. Este problema pode ser tratado da seguinte maneira:

- Associando a cada objeto 8 de controle, onde d é a dimensão. Os d bits registram o tipo de célula onde reside seu centróide (célula H), e os outros 2^d bits especificam os tipos de células que intersectam seu volume limitante (células P).

4.1.3 Implementação da subdivisão espacial paralela em CUDA

O algoritmo compreende as seguintes etapas:

- Inicialmente são construídas as listas de identificadores de objeto e identificadores de célula na memória da GPU.
- Durante o teste de colisão as listas de objetos e células são ordenadas utilizando o algoritmo *RadixSort*[92].
- Finalmente as listas são percorridas para encontrar os pares de objetos em colisão.

Para explicar o processo que envolve a construção das listas de identificadores de células e objetos será empregada a configuração ilustrada nas figuras 4.5 e 4.6.

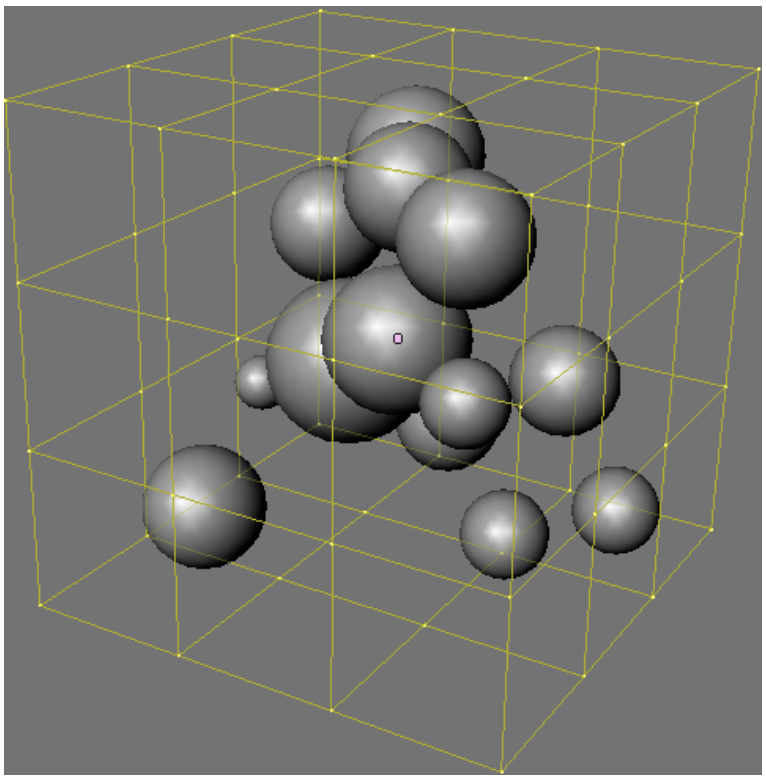


Figura 4.5: Volumes envolventes (esferas) contidos numa grade 3D de $3 \times 3 \times 3$.

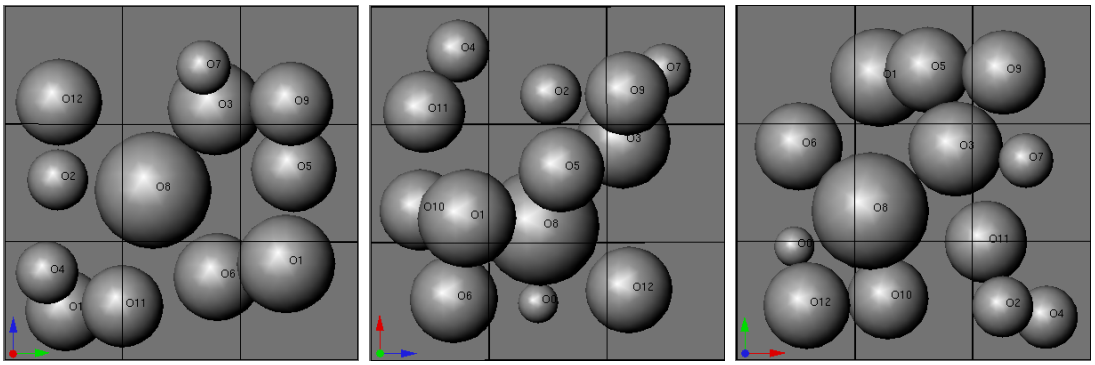


Figura 4.6: O conjunto de objetos da figura 4.5 é projetado nos planos YZ (esquerda), ZX (meio) e XY (direita).

Construção das listas de identificadores de objetos e células

Os atributos de cada objeto são: o ID (que é usado para acessar propriedades do objeto e a construção da lista de identificadores de célula), seus bits de controle (usados para armazenar vários estados como o tipo de célula H e outros tipos de células ocupadas, como mencionado no final da seção 4.1.1), e até 2^d identificadores de células que podem ser intersectadas. Estes atributos são armazenados em dois arrays distintos de $(n \times 2^d)$ elementos:

1. O primeiro array contém os identificadores de célula do primeiro objeto, seguido pelo identificador de célula do segundo objeto, e assim por diante.

2. O segundo array contém os identificadores de objeto e bits de controle para cada objeto (um para cada célula que ocupa).

Depois que os arrays foram alocados, o primeiro passo para realizar a detecção de colisão é preencher o array de índices de células. Para cada objeto, há células (diferentes da célula H) que seu volume limitante intersecta (denominadas células P). Como indicado anteriormente, se para um objeto O_i uma célula C_k é P, e também é P para outro objeto O_j, então O_i e O_j não colidem na célula C_k, pois seus centróides estão separados numa distância maior que diâmetro da maior esfera envolvente. Os identificadores de células H são calculados usando uma função *hash* das coordenadas de seus centróides. Este cálculo é feito da seguinte forma:

```
unsigned cell_id = (unsigned)(pos.x / CELLSIZE) << XSHIFT |
                   (unsigned)(pos.y / CELLSIZE) << YSHIFT |
                   (unsigned)(pos.z / CELLSIZE) << ZSHIFT;
```

onde *pos* é a posição do objeto, $CELLSIZE$ é a dimensão de uma célula e $[XYZ]SHIFT$ são constantes usadas para mapear coordenadas 3D em 1D. O identificador da célula H é armazenado no array de células que um objeto intersecta

como o primeiro elemento. Simultaneamente, o elemento correspondente no array de identificadores de objeto é atualizado com o identificador do objeto e atribuímos um bit de controle para indicar que se trata da célula H do objeto. Em seguida, as células P de cada objeto são armazenadas no array de identificadores de célula, testando se o volume limitante intersecta algumas das $3^d - 1$ células vizinhas. Dado que o tamanho da célula é maior que o maior volume limitante, um objeto pode tocar não mais do que 2^d células. Isto significa que um objeto tem uma célula H e que pode ter até $2^d - 1$ células P. Se o objeto tem menos de $2^d - 1$ células P, os identificadores de células extra são marcadas como *null* para indicar que não são válidos.

Objetos	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}
Células H	001	120	201	112	200	121	010	212	111	222	100	200	002
Células P	011	110		111			020		010	121	000	100	001
		121		121			011		110	122		110	
		020		122					001	221		210	
		021		212					101				
		111		211					011				
				222									
				221									

Figura 4.7: Resultado da construção do array de identificadores de célula para o conjunto de objetos da figura 4.5.

Ordenação do array de identificadores de células

Este array é ordenado pelos identificadores de células. Naturalmente, cada identificador de célula aparece várias vezes no array, e queremos que elementos com o mesmo tipo de célula estejam juntos (com a célula H na frente), já que isso simplifica o processamento de células em colisão. Como o array de identificadores de célula já foi ordenado, primeiro pelo *id* de objeto e segundo pelo tipo de células (H ou P), o algoritmo de ordenação deve ser um método de ordenação estável que garanta, no final do processo de ordenação, que identificadores de células iguais continuem na mesma ordem que se encontravam no início. O método de ordenação em paralelo *RadixSort* cumpre com essas exigências e a figura 4.8 ilustra o resultado da ordenação aplicado ao array de identificadores de célula da figura 4.7.

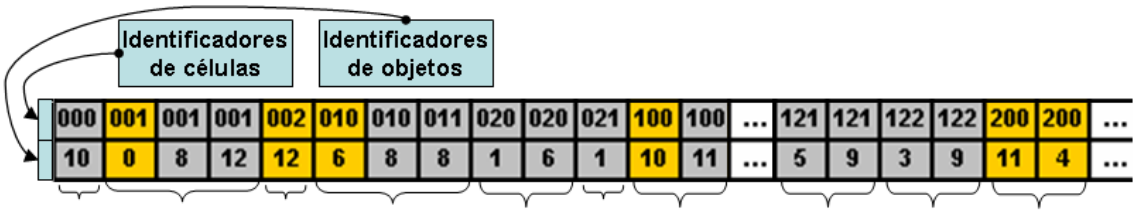


Figura 4.8: O algoritmo Radix Sort aplicado ao array de identificadores de célula.

Criando a lista de células de colisão

Uma célula é de colisão quando contém mais de um objeto, e requer processamento na fase de detecção de colisão exata. A lista de todas as células de colisão é criado a partir do array de identificadores de células da seguinte forma: o array ordenado de identificadores de células é percorrido para verificar mudanças, isto é, a ocorrência que marca o final da lista de ocupantes de uma célula e o início da outra. Este processo é descrito em detalhe em [22]). No final do processo, cada célula de colisão guarda seu início, o número de células H, e o número de células P, como mostrado na Figura 4.9. Esta lista pode, opcionalmente, ser dividida em 2^d listas diferentes de células, e o processo de detecção de colisões ocorre em 2^d passos.

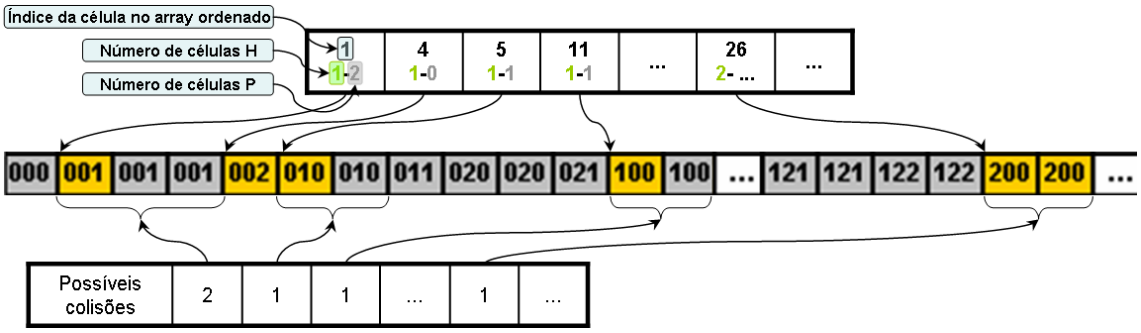


Figura 4.9: Inspecionando mudanças na lista ordenada de identificadores de células para localizar células em colisão.

Percurso da lista de células em colisão

Cada lista de células em colisão é percorrida, e as células em colisão são selecionadas para serem processadas, conforme ilustrado na Figura 4.9. Em aplicações tais como a dinâmica molecular ou simulação de partículas, colisões potenciais podem ser processadas imediatamente, permitindo que o percurso sirva como a fase exata. Para uma simulação onde as partículas são constituintes de objetos (veja a seçao) o percurso apenas guarda todas as colisões que passam pelo teste de sobreposição de volumes limitantes.

Logo, como há uma *thread* processando cada célula de colisão, o potencial da GPU será aproveitado apenas se existe uma grande quantidade de células em colisão,

pelo que esta abordagem é adequada para simular a interação de grandes quantidades de objetos.

4.1.4 Discussão

Observamos que o método apresentado por Le Grand [22] incorre num grande erro ao afirmar que "um teste de colisão entre dois objetos é realizado se pelo menos um deles tem seu centróide na célula onde existe a colisão". Segundo o autor, o esquema suporta qualquer configuração fazendo que o tamanho da célula seja 1.5 vezes maior que o diâmetro da maior esfera. A Figura 4.10 demonstra que não importa quanto o diâmetro das esferas seja reduzido, o erro continuará aparecendo.

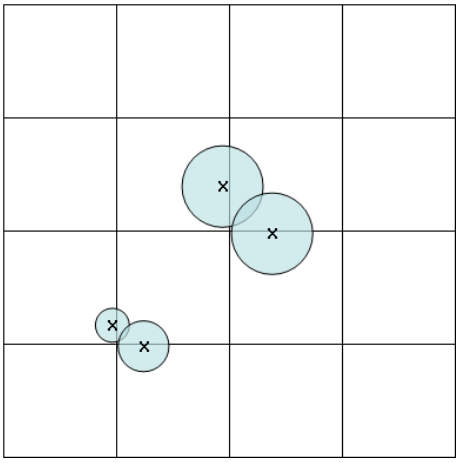


Figura 4.10: Erros do esquema de subdivisão.

Na prática, o método proposto pelo autor realmente não detecta todas as colisões, mas isso não invalida totalmente o método, já que pode ser corrigido. Entretanto, a correção acarretará um gasto adicional de processamento devido às verificações requeridas.

4.2 Detecção de colisão grosseira baseado no esquema de varredura e poda

Este método implementa em paralelo uma técnica já bastante conhecida, a de varredura e poda. Embora uma implementação direta em paralelo não seja possível, os autores fizeram algumas modificações para aliviar as limitações da técnica sequencial. Também a técnica *analise da componente principal* (PCA) apresentada por [107] é usada para determinar o melhor eixo de varredura. Assim, o algoritmo principal é executado nas seguintes etapas:

1. Os objetos O_i são projetados no eixo escolhido para obter a lista de intervalos $I = (m_1, M_1), (m_2, M_2), ..., (m_n, M_n)$, onde m (M) é o início (final) do intervalo ocupado pelo objeto O_i.

2. A lista I é ordenada pelas primeiras componentes usando o algoritmo *Radix-Sort*, obtendo a lista ordenada L.

3. Para cada O_i: Primeiro, a lista L é percorrida no intervalo I_i até que $M_i < m_j$ para algum objeto O_j, onde $i < j$. Logo, se $m_j \in I_i$ para algum objeto O_j, então se verifica se ha sobreposição entre os objetos O_i e O_j, caso positivo o par (O_i, O_j) é acrescentado na lista P.

4. A lista P representa o conjunto em colisão potencial que devera ser analisado para determinar se realmente há colisão entre os pares de objetos.

O método resulta particularmente vantajoso por que não apresenta limitações como no caso do método apresentado na seção anterior, onde o tamanho dos objetos devem ser de determinado tamanho em relação à subdivisão do espaço. Detalhes sobre a implementação deste método podem ser encontrados no artigo apresentado por Liu et al.[40].

Capítulo 5

Simulação baseada em física usando partículas e a GPU

Utilizando técnicas de simulação baseada em física podem-se gerar animações realistas e de alta qualidade. Por exemplo, animar um sólido que cai num recipiente ou superfície não é difícil utilizando algum software para animação 3D. Porém, animar milhares de sólidos caindo e batendo uns nos outros demanda um trabalho considerável, ou torna-se inviável, já que o comportamento físico pode não ser correto. Assim, a simulação é a única solução prática quando há uma grande quantidade de objetos que interagem num cenário. Entretanto, dependendo da quantidade de objetos e da complexidade do cenário, a computação da simulação pode-se tornar custosa fazendo necessário o uso de técnicas para acelerar a simulação.

Existe uma grande demanda por métodos computacionais para simulação baseada em física que possam ser executadas em tempo real, principalmente pela industria de jogos e do cinema, onde é necessário animar cenários com alta qualidade e realismo. Por outro lado, aplicações em tempo real são muito mais exigentes, já que é necessário computar uma grande quantidade de quadros por segundo para obter uma animação contínua.

Atualmente usam-se GPUs para acelerar simulações baseadas em física, e técnicas como autômatos celulares [93], dinâmica euleriana de fluidos [17, 94], simulação de tecidos baseado em massa-mola [95] entre outras foram propostas na literatura. Uma característica comum nestas técnicas é que os objetos sendo simulados são representados por partículas e o espaço ocupado por eles é representado por uma grade regular que é uma estrutura de dados adequada para aproveitar o paralelismo e poder computacional das GPUs.

Este capítulo descreve em detalhe técnicas de simulação baseada em partículas apresentadas por Harada [2, 3, 17, 24], originalmente implementadas empregando *shaders* mas que foram adaptadas para a tecnologia CUDA, permitindo assim aproveitar melhor o poder computacional das GPUs e acelerar a simulação física.

5.1 Detecção de colisão baseada em partículas

Harada [2, 24] apresenta uma técnica de detecção de colisões para simulações que usam grandes quantidades de partículas, como é requerido no caso de uma simulação com milhares de sólidos, simulação de líquidos ou uma combinação destas. Esta técnica se baseia no gerenciamento de uma grade espacial que serve para localização rápida de partículas, isto é, cada voxel da grade guarda as partículas que têm o centro localizado dentro do voxel. Conforme proposto por Harada et al. [2], essa técnica é limitada devido ao uso de texturas e ao fato de que cada voxel (representado por um pixel RGBA) pode guardar referências para apenas quatro partículas (uma em cada canal) por vez. Para superar esta limitação, nesta tese adotamos a técnica descrita por Green [96], onde cada voxel admite uma quantidade arbitrária de partículas.

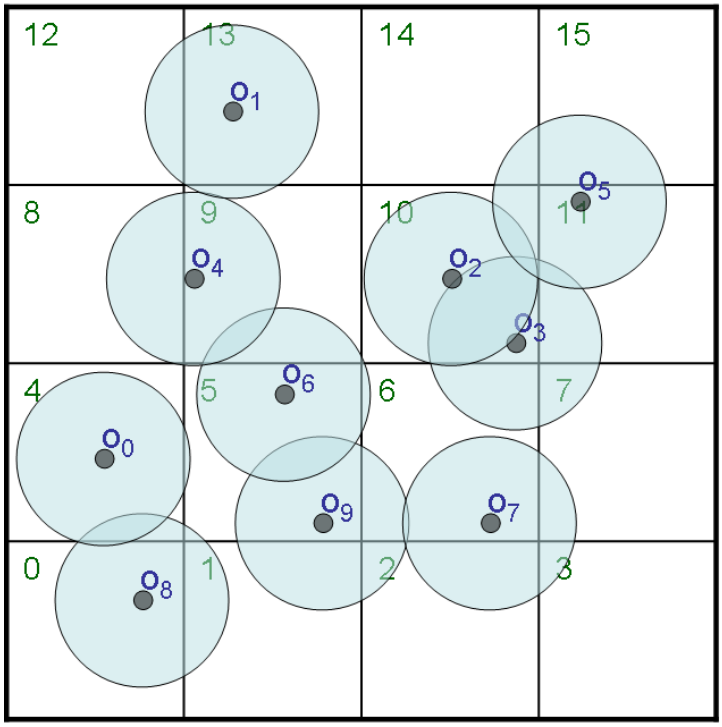
Usando esta técnica, a detecção de colisão, para cada passo de tempo, envolve duas etapas.

Atualização da grade: A grade é um array 3D onde cada célula(voxel) contém uma lista de índices de partículas (veja a figura 5.1). Uma célula é representada por um cubo de largura e altura iguais ao diâmetro da partícula.

Cada partícula é mapeada na célula segundo a sua posição. O mapeamento é simples usando a função $(g_x, g_y, g_z) = (\lfloor \frac{p_x}{D} \rfloor, \lfloor \frac{p_y}{D} \rfloor, \lfloor \frac{p_z}{D} \rfloor)$, onde g_x, g_y e g_z são coordenadas da grade 3D; p_x, p_y e p_z são as coordenadas da posição da partícula; e D é o diâmetro da partícula.

Detecção de colisão: A detecção de colisão se resume à verificação de intersecção entre esferas, ou seja, há colisão se a distância entre os centros de duas partículas é menor do que seu diâmetro.

A grade 3D auxilia este processo restringindo o teste a partículas que residem na mesma célula ou em células adjacentes. Assim, a execução deste processo tem complexidade quase linear no número de partículas, isto por que é improvável encontrar um grande número de partículas ocupando o mesmo voxel.



Índice de Célula	Índices de Partículas
0	*8*
1	
2	
3	
4	*0*
5	*6, 9*
6	*7*
7	
8	
9	*4*
10	*2, 3*
11	*5*
12	
13	*1*
14	
15	

Figura 5.1: Mapeamento de partículas na grade.

5.2 Simulação de objetos rígidos

5.2.1 Movimento de um corpo rígido

A técnica apresentada em [2] computa o movimento de um corpo rígido em duas partes: a translação e a rotação, como ilustrado na figura 5.2. A translação descreve o movimento do centro de massa, enquanto que a rotação descreve como o corpo rígido gira em torno do seu centro de massa. Uma descrição detalhada pode ser encontrada em [97].

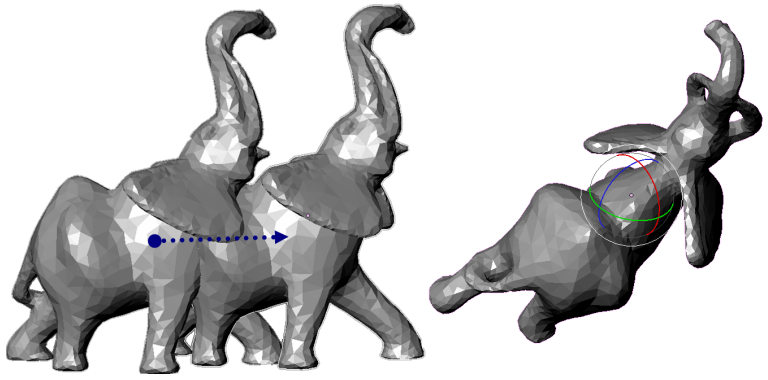


Figura 5.2: Translação de um objeto rígido (esquerda) e rotação de um corpo rígido (direita).

Translação

O movimento do centro de massa é trivial, podendo ser considerado similar ao movimento de uma partícula. Assim, quando uma força F atua sobre um corpo rígido, esta força pode causar uma variação do seu momento linear P. A relação entre P e F é:

$$\frac{dP}{dt} = F. \tag{5.1}$$

Logo, a velocidade v do corpo rígido é encontrada dividindo P pela massa M do corpo rígido:

$$v = \frac{P}{M}. \tag{5.2}$$

A velocidade é a derivada da posição x do centro de massa em relação ao tempo

$$\frac{dx}{dt} = v. \tag{5.3}$$

Rotação

A força F que atua em algum ponto de um corpo rígido, diferente do centro de massa, gera uma mudança no momento angular L. Esta variação depende da posição relativa r, que é o ponto de atuação em relação ao centro de massa. Assim, a derivada de L em relação ao tempo é o torque, que por sua vez é o produto vetorial entre r e F:

$$\frac{dL}{dt} = r \times F. \tag{5.4}$$

A velocidade angular ω, cuja magnitude é a velocidade de rotação, é a derivada do momento angular L:

$$\omega = I(t)^{-1}L, \tag{5.5}$$

onde a matriz 3×3 $I(t)$ é o tensor de inércia no tempo t e $I(t)^{-1}$ é sua inversa. O tensor de inércia é um valor físico, que indica a resistência do corpo ao movimento de rotação. O tensor de inércia muda de acordo com a orientação do corpo rígido, então deve ser recomputada a cada passo de tempo. A matriz $I(t)^{-1}$ é calculada usando a seguinte equação:

$$I(t)^{-1} = R(t)I(0)^{-1}R(t)^T, \tag{5.6}$$

onde $R(t)$ é a matriz de rotação que descreve a orientação de um corpo rígido no tempo t.

Tendo a velocidade angular ω, o passo seguinte é calcular a derivada da orientação do corpo rígido em relação ao tempo usando a velocidade angular ω. Para guardar a orientação, tipicamente são usadas duas formas de representação: matrices de rotação ou quatérnios. Quatérnios têm sobre matrices diversas vantagens; por exemplo, matrices de rotação, tendem a acumular componentes não rotacionais ao logo do tempo durante a simulação. Portanto, erros numéricos que se somam durante combinações repetidas de matrizes de rotação podem levar a uma distorção do corpo rígido. Quatérnios, por sua vez, não têm graus de liberdade para o movimento não rotacional. Além disso, a representação é adequada para ser usada em GPUs, pois é constituído por quatro números que podem ser armazenados armazenados nos canais RGBA de um pixel ou de modo semelhante numa variável CUDA de tipo *vector4*. Dessa forma aproveita-se a arquitetura da GPU nos cálculos, já que o custo de operações primitivas na GPU é igual se utilizado uma variável *vector4* ou um *float* simples.

Um quatérnio $q = [s, v_x, v_y, v_z]$ representa uma rotação de s radianos em torno de um eixo $v = (v_x, v_y, v_z)$. Para simplificar, um quatérnio pode ser representado como $q = [s, v]$.

Um objeto cuja orientação é representado por q, e que é submetido a uma alteração de sua orientação dada por uma velocidade ω durante um intervalo de tempo dt, tem sua orientação alterada pela equação:

$$dq = (\cos(\frac{\theta}{2}), a\sin(\frac{\theta}{2})) \tag{5.7}$$

onde $a = \omega/\|\omega\|$ é o eixo de rotação e $\theta = \|\omega dt\|$ é o ângulo de rotação. Então o quatérnio no tempo $t + dt$ é atualizado da seguinte forma:

$$d(q + dt) = q \times dq \tag{5.8}$$

5.2.2 Representação da forma dos objetos

O cálculo do movimento de corpos rígidos é relativamente simples, como visto acima, entretanto para simular a interação entre objetos precisamos tratar possíveis colisões. Na abordagem proposta por Harada, os corpos rígidos são representados por conjuntos de partículas (pequenas esferas) [16, 98], como ilustrado na figura 5.3. Isto pode ser feito usando uma grade 3D que discretiza o espaço ocupado pela caixa limitante do corpo rígido, e assumindo que existe uma partícula em cada voxel da grade. Logo, são usadas apenas partículas cujos centros estejam localizados no interior do corpo rígido. Observe que esta técnica poderá ser utilizada apenas para corpos rígidos representados por superfícies fechadas ou funções implícitas.

Harada [2] descreve uma técnica baseada no método *depth-peeling*, que é execu-

tada na GPU permitindo agilizar a obtenção das partículas que conformam o corpo rígido. Entretanto, esta técnica, dependendo da resolução, pode gerar uma grande quantidade de esferas, podendo não ser necessárias todas elas. Para obter apenas esferas que estejam próximas da superfície do corpo rígido, pode-se usar a técnica de traçado de raios (*ray-casting*) paralelos a algum eixo. Para cada raio lançado, partindo de fora do objeto, são detectados os voxels que intersectam a superfície (entrada e saída) do corpo.

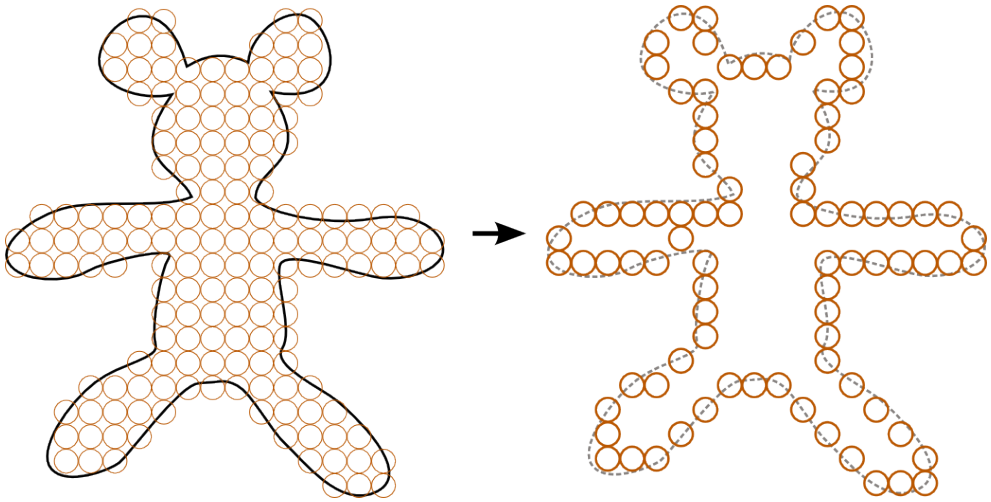


Figura 5.3: Conjunto de partículas que representam um corpo rígido.

Observa-se que a voxelização resulta numa aproximação da superfície do corpo rígido, isto é, as partículas que representem o corpo rígido não definem adequadamente a superfície (ver figura 5.3). Assim, quando os objetos são de geometria complexa, a simulação pode ocasionar reações de colisão inesperadas.

Uma alternativa é alocar esferas no interior do corpo, numa distância r da superfície. Por outro lado, é recomendável que as esferas sejam posicionadas a uma distância que varia entre r e $2r$, desta forma obtém-se uma boa densidade de partículas sem impactar no consumo de memória. A figura 5.4 ilustra a diferença entre as duas técnicas.

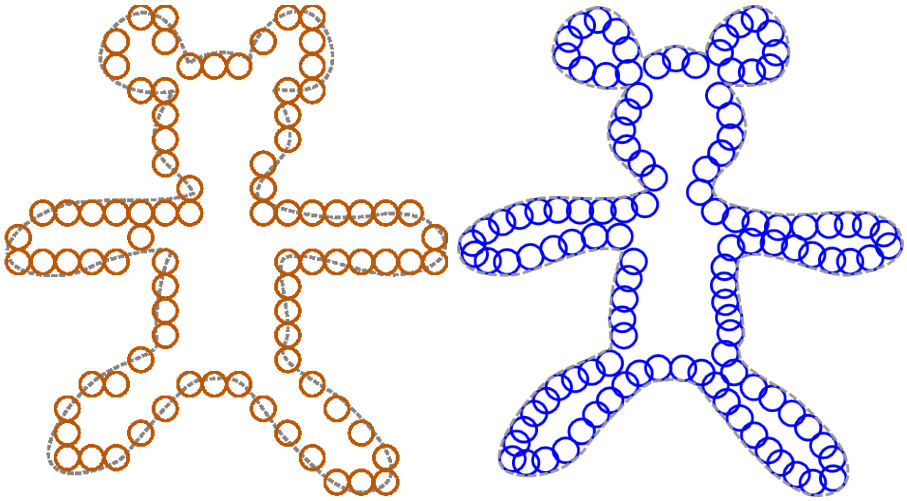


Figura 5.4: Partículas que representam um objeto: obtidas por voxelização (esquerda) e por posicionamento na superfície (direita).

5.2.3 Detecção de colisão

Como um corpo rígido é representado por um conjunto de partículas, é utilizada a técnica de detecção de colisão para sistemas de partículas descrita na seção 5.1. Harada [2] mostra que esta técnica é adequada para ser implementada na GPU aumentando o desempenho da simulação com relação a uma implementação em CPU. Esta abordagem dispensa uma etapa *broad phase*, uma vez que objetos são representados por conjuntos relativamente pequenos de partículas e são testadas quanto a colisões fazendo uso da grade. A vantagem desta representação de objetos, por conjuntos de partículas, é a controlabilidade de desempenho e precisão da simulação. Aumentar a resolução dos objetos empregando maior quantidade de partículas (veja figura 5.5) aumenta o custo computacional diminuindo o desempenho da simulação, no entanto, consegue-se uma representação mais precisa da forma dos objetos e o realismo da simulação melhora. Desta forma, podemos ajustar a resolução dos objetos dependendo da finalidade.

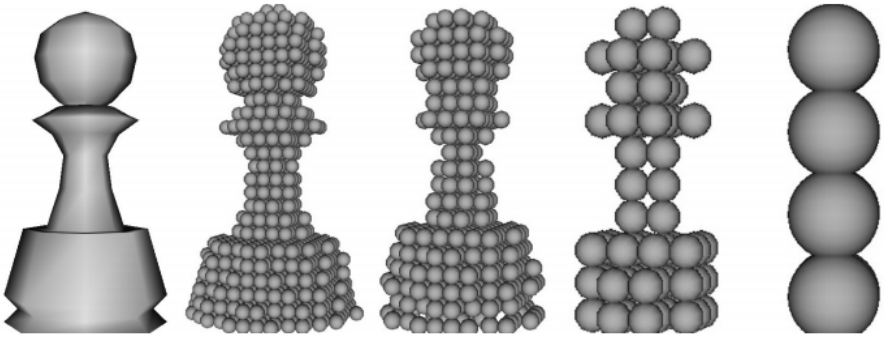


Figura 5.5: Diferentes resoluções de partículas para representar um sólido (imagem 29-3 extraída de [2]).

5.2.4 Resposta às colisões

O processo de detecção de colisão verifica se uma partícula colide com seus vizinhos locais. Se há colisão, forças de repulsão são calculadas aplicando o método de elementos discretos (DEM [1]). A força de repulsão $f_{i,s}$, é modelada por uma mola linear, e a força de amortecimento $f_{i,d}$, modelada por um amortecedor, que dissipa a energia entre as partículas. Assim, para uma partícula i que colide com uma partícula j, as forças são calculadas pelas seguintes equações:

$$f_{i,s} = -\kappa(d - |r_{ij}|)\frac{r_{ij}}{|r_{ij}|}, \tag{5.9}$$

$$f_{i,d} = \eta v_{ij}, \tag{5.10}$$

onde κ, ∇, d, r_{ij} e v_{ij} são: o coeficiente de restituição da mola, coeficiente de amortecimento, diâmetro da partícula, posição relativa da partícula e velocidade relativa da partícula p_j em relação à partícula p_i, respectivamente. A força de cisalhamento $f_{i,t}$ também é modelada como uma força proporcional à velocidade tangencial relativa $v_{ij,t}$:

$$f_{i,t} = \kappa_t v_{ij,t}, \tag{5.11}$$

onde a velocidade tangencial relativa é calculada como:

$$v_{ij,t} = v_{ij} - (v_{ij}\frac{r_{ij}}{|r_{ij}|})\frac{r_{ij}}{|r_{ij}|} \tag{5.12}$$

Então, a força total F e o torque total T aplicados a um corpo rígido correspondem ao somatório das forças e torques exercidos sobre as partículas do corpo rígido:

$$F = \sum_i (f_{i,s} + f_{i,d} + f_{i,t}), \tag{5.13}$$

$$T = \sum_i (r_i \times (f_{i,s} + f_{i,d} + f_{i,t})), \tag{5.14}$$

onde r_i é a posição relativa atual da partícula i em relação ao centro de massa.

5.2.5 Implementação da simulação na GPU

Uma iteração da simulação consiste de cinco etapas (a figura 5.6 mostra o processo todo para um passo de tempo).

[1] O DEM é um método para simulação de materiais granulares [99]

1. Cálculo de valores de partículas
2. Atualização da grade
3. Detecção e resposta a colisões
4. Cálculo de momentos
5. Cálculo de posição e orientação dos corpos rígidos

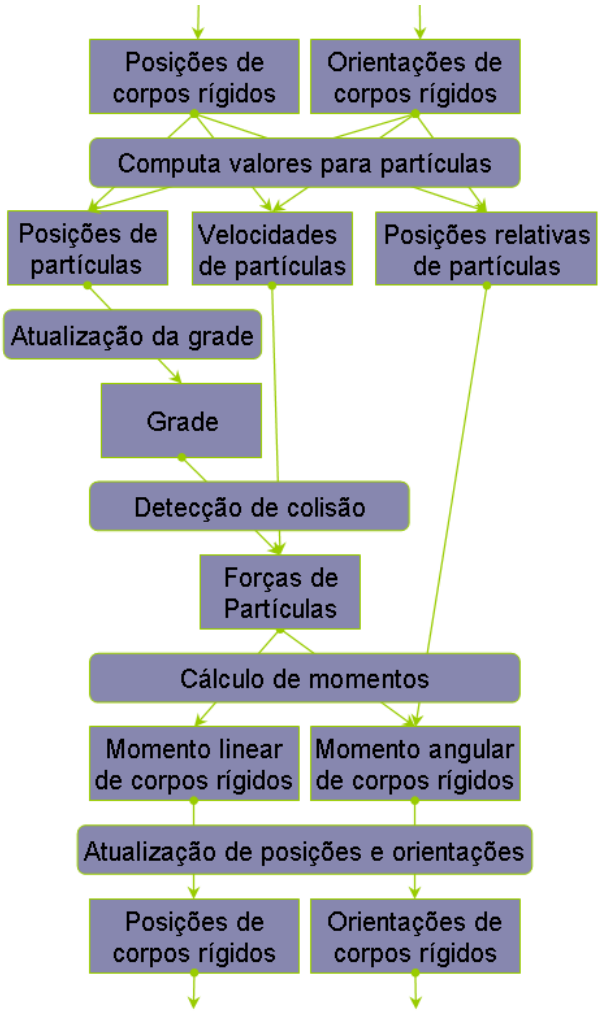


Figura 5.6: Fluxograma da simulação de corpos rígidos na GPU

Estruturas de dados utilizadas

Diferentemente da implementação apresentada por Harada [2], onde os valores físicos são guardados em *texturas*, neste trabalho, utilizamos arrays de vetores de escalares (*vector4*). Para esta implementação são necessárias as seguintes informações:

1. para cada partícula i pertencente ao objeto j
   - posições p_i

- posições relativas r_i
- velocidades v_i
- forças f_i
- torque τ_i
- índice j do corpo ao qual pertence

2. para corpos rígidos

- posições X_j
- orientações Q_j
- velocidades lineares V_j
- velocidades angulares ω_j

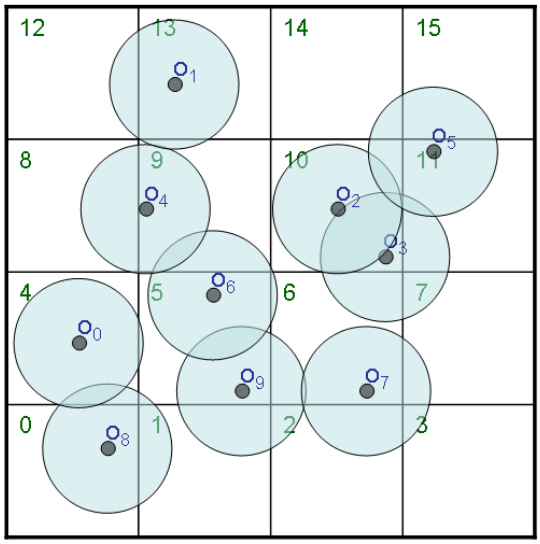
Note que para cada partícula p_i é atribuído um índice j que permite identificar o corpo rígido O_j ao qual pertence.

Geração e atualização da grade

A grade é representada por quatro arrays de inteiros. Os dois primeiros, C_L e P_L, de tamanho igual ao número de partículas, guardam o índice *hash* da célula onde a partícula i se encontra e o índice da partícula i respectivamente (veja o mapeamento na figura 5.7). O terceiro e o quarto array, S_L e E_L, de tamanho igual ao número de células, guardam (para cada célula) o início e o fim de um intervalo de índices dos arrays C_L e P_L. Cada intervalo de índices $S_{L_k}...E_{L_k}$ corresponde a todas as posições onde se encontram partículas cujos centros residem na célula k.

A atualização da grade consiste em ordenar os arrays C_L e P_L pelos índices de células, ou seja, usando como chave os valores de C_L. O algoritmo de ordenação utilizado é o *RadixSort* [22], mas também poderia ser utilizado o algoritmo *BitonicSort* ou qualquer outro algoritmo eficiente de ordenação em paralelo.

Uma segunda etapa envolve a varredura do array C_L para detectar o intervalo de partículas que estão alojadas em cada célula. A figura 5.7 mostra o resultado dos processos de construção e ordenação de arrays de índices de células e índices de objetos. Logo, os intervalos/grupos de partículas contidas em cada voxel são encontrados.



Índices dos arrays	Arrays desordenados (Id célula, id objeto)	Arrays Ordenados	(inicio, fim)
0	4, 0	0, 8	0, 1
1	13, 1	4, 0	
2	10, 2	5, 6	
3	10, 3	5, 9	
4	9, 4	6, 7	1, 2
5	11, 5	9, 4	2, 4
6	5, 6	10, 2	4, 5
7	6, 7	10, 3	
8	0, 8	11, 5	
9	5, 9	13, 1	5, 6
10			6, 8
11			8, 9
12			
13			9, 10
14			
15			

Figura 5.7: Construção e ordenação de arrays de índices de células e índices de objetos para um conjunto de partículas em 2D.

Cálculo de valores físicos para as partículas

Primeiro os valores físicos das partículas são calculados, permitindo assim inferir a força e torque aplicados sobre os corpos. Para tanto, computa-se a posição relativa com relação ao centro de massa do corpo ao qual pertence. Isto é feito multiplicando a posição relativa inicial pela orientação atual do corpo.

$$r_i = Q_j r_i^0 Q_j^*, \tag{5.15}$$

onde Q_j é a orientação atual do corpo e Q_j^* é a orientação anterior do corpo. Logo, a posição de uma partícula x_i é calculada a partir do centro de massa do objeto e a posição relativa atual da partícula r_i como segue:

$$x_i = X_j + r_i \tag{5.16}$$

A velocidade v_i da partícula é analogamente calculada usando as velocidades linear V_j e angular ω_j do corpo rígido:

$$v_i = V_j + \omega_j \times r_i \tag{5.17}$$

Detecção de colisão e reação

Como descrito no capítulo 5.1, o processo de detecção de colisão de partículas é limitado à busca na vizinhança da localização de cada partícula. Simplesmente calcula-se a distância entre duas partículas, que ocupam a mesma célula ou células vizinhas, e se verifica esta distância. Se é menor que o diâmetro há colisão e as

forças de reação entre as partículas são calculadas usando as equações 5.9, 5.10 e 5.11.

Em CUDA este processo é realizado utilizando um *kernel*, onde para cada partícula, se busca colisões no grupo de partículas que se encontram na vizinhança de 27 voxels (veja o código 5.1).

Listing 5.1: Kernel *CUDA* para calcular forças entre partículas

```
__global__ void
computeForcesD(float4* forceArray,    /* output: força */
               float4* torqueArray    /* output: torque */)
{
   uint index = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;
   if (index >= params.numParticles) return;

   // leitura de dados das texturas
   float4 posData = FETCH(particlePos, index);
   float4 velData = FETCH(particleVel, index);
    float3 pos = make_float3(posData);
   float3 vel = make_float3(velData);

   // necessario para o cálculo do torque [t = f x (ppos - RBpos)]
   uint RBindex = (uint)posData.w;
   float3 RBpos = make_float3(FETCH(RBPos, RBindex));

   // posiçao na grade
   int3 gridPos = calcGridPos(pos);

   // acumula forças e torques
   float3 force = make_float3(0.0f);
   float3 torque = make_float3(0.0f);

   float3 f;  // força temporaria

   // busca na vizinhança
   for(int z=-1; z<=1; z++) {
      for(int y=-1; y<=1; y++) {
         for(int x=-1; x<=1; x++) {
            int3 neighbourPos = gridPos + make_int3(x, y, z);
            uint gridHash = calcGridHash(neighbourPos);

            // percorre todas as partículas no voxel
            uint startIndex = FETCH(cellStart, gridHash);

            if (startIndex != 0xffffffff) {       // célula nao vazia
               uint endIndex = FETCH(cellEnd, gridHash);
               for(uint j=startIndex; j<endIndex; j++) {
                  if (j != index) {   // apenas partículas diferentes desta
                     float4 posj = FETCH(particlePos, j);
                     uint RBj = posj.w;
                     if (RBi != RBj){
                        float3 pos2 = make_float3(posj);
                        float4 velData = FETCH(particleVel, j);
                        float3 vel2 = make_float3(velData);
```

```
                        // colisao entre dois esferas
                        f = collideSpheres(pos, pos2, vel, vel2, params);

                        force += f;
                        torque += cross(pos - RBpos, f);
                    }
                }
            }
        }

    }
  }
}

f = collideWalls(pos, radius, params);
force += f;
torque += cross(pos - RBpos, f);

// adiciona gravidade
volatile uint2 hash = FETCH(particleHash, index);
force +=  params.pmass*params.gravity;
torque += cross(pos - RBpos, params.pmass*params.gravity);

// atualiza força e torque
forceArray[hash.y] = make_float4(force, 1.0f);
torqueArray[hash.y] = make_float4(torque, 1.0f);
}
```

Cálculo do momento

Após o cálculo das forças sobre as partículas, usamos as equações 5.1 e 5.4 para calcular os momentos. O cálculo da variação do momento linear de um corpo rígido requer da soma de forças que agem sobre as partículas que pertencem ao corpo rígido. Esta operação é feita num *kernel*, onde para cada corpo rígido, serão somados os valores de força e torque de todas suas partículas. Se o número de partículas por corpo rígido for grande, pode-se usar algum algoritmo de redução em paralelo (ver [100]). De modo semelhante, o cálculo da variação do momento angular requer a posição relativa das partículas em relação ao centro de massa. Como as posições já foram computadas numa etapa anterior é conveniente executar os cálculos do momento angular e linear no mesmo *kernel*. (veja o código 5.2).

Listing 5.2: Kernel *CUDA* para calcular o momento

```
__global__ void
computeMomentaD(float4* RBVelArray,     // velocidade linear
                float4* RBwVelArray,    // velocidade angular
                float dt)
{
    uint index = __umul24(blockIdx.x,blockDim.x) + threadIdx.x;
    if (index >= params.numRBs) return;
```

```
    volatile float4 RBVelData = RBVelArray[index];
    volatile float4 RBwVelData = RBwVelArray[index];

    float3 force = make_float3(FETCH(RBForce, index));    // Força
    float3 torque = make_float3(FETCH(RBTorque, index)); // Torque

    float RBmass = params.pmass*RBwVelData.w;     // massa do corpo

    // Atualiza a velocidade do corpo rígido
    // v = V + f*dt/m
    float3 vel = make_float3(RBVelData) + force*(dt/RBmass);

    RBVelArray[index] = make_float4(vel*delta, RBVelData.w);

    float3x3 R;
    quat2Matrix(FETCH(RBOri, index), &R);

    // tensor de inércia
    float3x3 invI;
    for (uint i = 0; i < 3; i++){
        for (uint j = 0; j < 3; j++) {
            invI.m[i][j] = 0.0f;
        }
    }

    invI.m[0][0] = 1.0/Ixx;
    invI.m[1][1] = 1.0/Iyy;
    invI.m[2][2] = 1.0/Izz;

    // Computa a inverse do tensor de inércia no tempo t
    float3x3 R_invI, RT, invIt;
    mulMatrix(&R, &invI, &R_invI);
    transpose(&R, &RT);
    mulMatrix(&R_invI, &RT, &invIt);

    // Computa a velocidade angular
    float3 wvel = make_float3(RBwVelData) + mulMatrixVector(&invIt, torque)*dt;

    RBwVelArray[index] = make_float4(wvel*delta, RBwVelData.w);

}
```

Cálculo da posição e orientação dos corpos rígidos

Uma vez computados os momentos, linear e angular, deve-se atualizar a posição e orientação dos corpos. Ambos cálculos podem ser realizados no mesmo kernel (veja o código 5.3).

Listing 5.3: Kernel *CUDA* para atualizar a posição e orientação dos objetos

```
__global__ void
updateRBStateD(float4* RBPosArray, // input/output: posiçoes de objetos
               float4* RBOriArray, // input/output: orientaçoes de objetos
               float dt)
{
```

```
    uint index = __umul24(blockIdx.x,blockDim.x) + threadIdx.x;
    if (index >= params.numRBs) return;

    float4 RBOriData = RBOriArray[index];

    // atualiza a posiçao
    RBPosArray[index] += make_float4(make_float3(FETCH(RBVel, index)), 0.0f)*dt;

    // atualiza a orientaçao
    float3 w = make_float3(FETCH(RBwVel, index))*dt;

    float lw = dot(w, w);

    if (lw > 0.0001f) {
        lw = sqrtf(lw);
        w = w*(1.0/lw);

        // metade do ângulo
        float a  = lw*0.5f;

        float4 dq = make_float4( w*sin(a), cos(a) );
        float4 Q = mulQuaternion(dq, RBOriData);
        RBOriArray[index] = normalizeQuaternion(Q);
    }

}
```

Renderização

Cada corpo rígido está associado a uma malha poligonal e é renderizado usando a posição X_j do centro de massa e a orientação Q_j. Note que as malhas poligonais correspondentes aos modelos já estão carregados na memória da GPU bastando, por tanto, apenas invocar uma instância quando se quer mostrar o objeto.

5.2.6 Simulação de objetos rígidos usando impulsos

A implementação baseada em forças, descrita na seção 5.2, gerou resultados de comportamento físico plausível, entretanto, observou-se que a modelagem da reação a colisões para objetos complexos, usando forças de repulsão, gera um comportamento elástico nos objetos quando há colisão com obstáculos ou outros objetos (ver figura 7.20). Este problema é inerente ao método, uma vez que o cálculo do momento depende da distância de interpenetração que há entre uma partícula e um obstáculo (ver equações 5.9, 5.10, 5.12, 5.13 e 5.14).

O problema se agrava uma vez que uma partícula, que faz parte de um objeto, não pode se mover de forma independente mas pode penetrar obstáculos (ou outro objeto) gerando forças elásticas muito grandes, dependendo da profundidade de penetração. A figura 5.8 ilustra duas situações onde os resultados após a colisão são completamente diferentes. Note-se que o objeto é o mesmo e percorre a mesma

distância, apenas, devido ao processo de amostragem no tempo, a posição está ligeiramente deslocada. No primeiro caso se observa que, num tempo t, o objeto está muito próximo do obstáculo; no seguinte passo de tempo $t + \Delta t$ uma partícula entra totalmente no obstáculo gerando uma força muito grande, proporcional à distância de penetração; e no seguinte passo de tempo $t + 2\Delta t$ o objeto será jogado para fora do obstáculo a uma velocidade grande. Entretanto, no segundo caso, no tempo t o objeto se encontra não tão próximo do obstáculo; no seguinte passo de tempo $t + \Delta t$ uma partícula interpenetra medianamente o obstáculo gerando uma força suficiente para repelir o objeto fora do obstáculo, porem a força será menor que no primeiro caso; e no passo de tempo $t + 2\Delta t$ o objeto assumira uma velocidade menor que no primeiro caso.

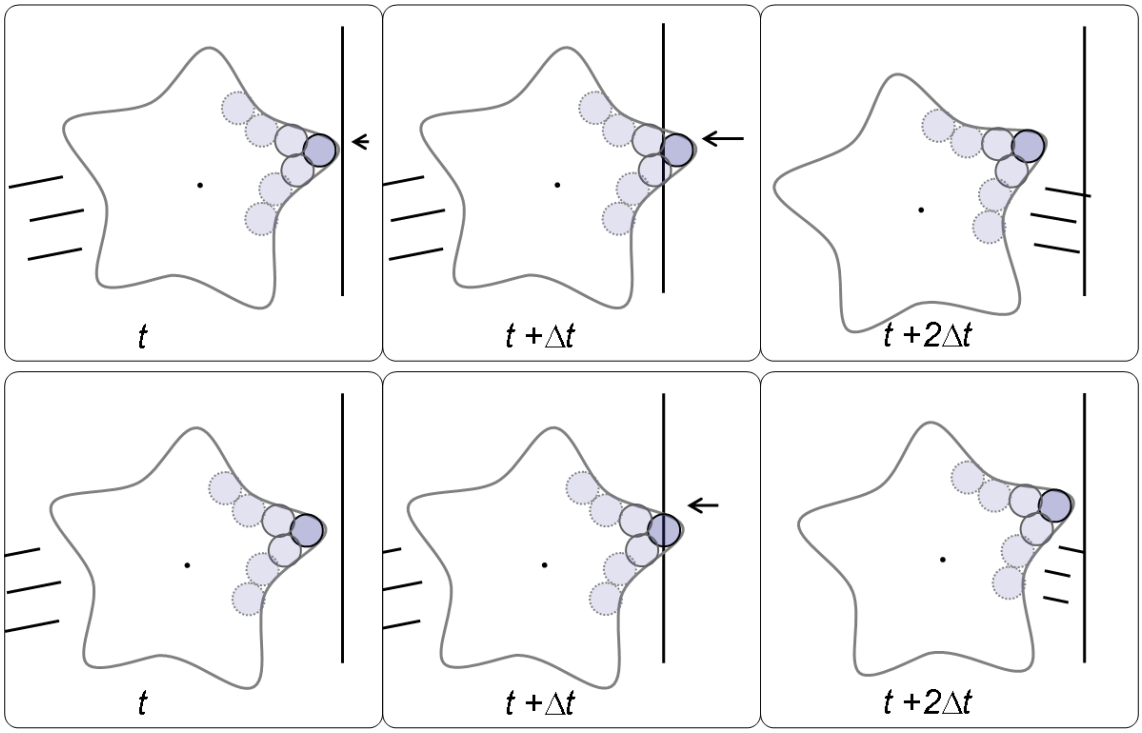


Figura 5.8: Resolvendo colisão entre objetos representados por partículas usando forças elásticas.

Pode-se notar, portanto, que este método depende, principalmente, da posição e a velocidade com que os objetos se movem e não apenas do intervalo de tempo. Para resolver este problema é necessário que as colisões sejam tratadas de um modo diferente. Na literatura existe o que é chamado de "detecção de colisão contínua" (*Continuous collision detection*, em inglês) [101], que tenta prevenir de forma robusta a interpenetração entre objetos, ainda quando estes se movem a grandes velocidades. Entretanto, o sistema de detecção de colisões usado neste trabalho é um sistema discreto, ou seja, avalia o estado dos objetos em tempos regulares.

Observe que o tratamento de colisões para um conjunto de partículas indepen-

dentes é plausível, isto por que a distância de interpenetração, dependendo do diâmetro, é pequena e por tanto não gera forças muito grandes. Por outro lado, quando há colisão entre uma partícula e um obstáculo, uma força proporcional à profundidade de penetração será gerada, e se a distância de penetração for muito grande, a força fará com que a partícula seja disparada para fora do obstáculo (ver figura 5.9). Este resultado, pode ser aliviado empregando um artifício simples que consiste em projetar a posição da partícula perpendicularmente para fora da superfície do obstáculo, é também refletir na superfície do obstáculo o vetor velocidade. Adicionalmente podemos reduzir a magnitude da velocidade que deve ser dissipada devido ao impacto. Considerando estes aspectos, esta técnica funciona bem para partículas soltas e pode ser aceitável.

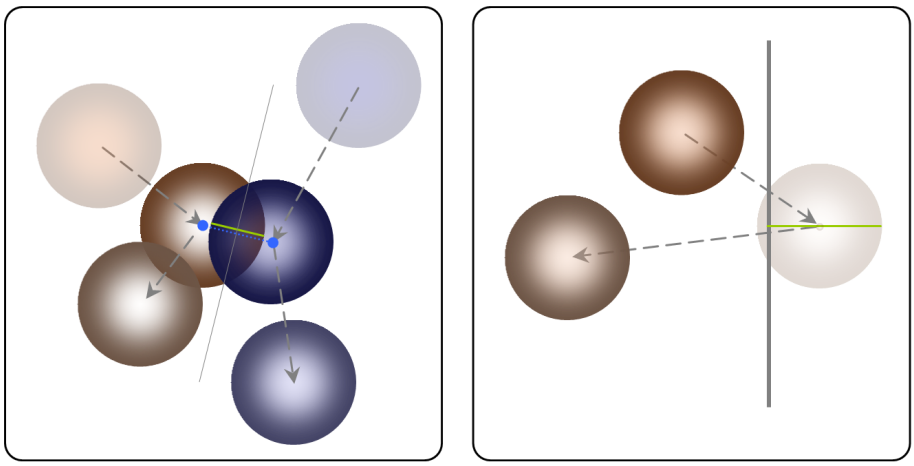


Figura 5.9: Resolvendo colisão partícula-partícula (esquerda) e colisão partícula-obstáculo (direita).

Uma vez observado que o uso de forças elásticas produz resultados indesejáveis, a implementação desta segunda versão segue a formulação para tratamento de colisões apresentada por Guendelman et al. [18], descrito na seção 3.1 e para a detecção de colisão é usada a técnica descrita na seção 5.2.

Foi observado também que a representação por partículas para objetos que possuem características finas, tais como, vincos e bordas afiadas é pouco adequada usando a técnica descrita na seção anterior, já que para aproximar razoavelmente um objeto que possui essa geometria é necessário empregar partículas menores, o que aumentará consideravelmente a quantidade de partículas. Nesses casos podemos adotar partículas de tamanhos diferentes para aproximar melhor a forma dos objetos, porem, gastando um número reduzido de partículas. A figura 5.10 ilustra um exemplo desta nova representação de partículas para objetos. A peça de xadrez é representada por 73 partículas de tamanho variável, já com a representação proposta por Harada seria necessário usar 748 partículas e já com a otimizando, apresentada na seção anterior, utilizando apenas partículas na superfície seria necessário 415

partículas. Além de ser necessário uma grande quantidade de partículas, para se ter uma animação razoável que evite a interpenetração entre objetos, seria necessário reduzir o intervalo de tempo Δt, o que diminuirá considerávelmente o número de quadros por segundo.

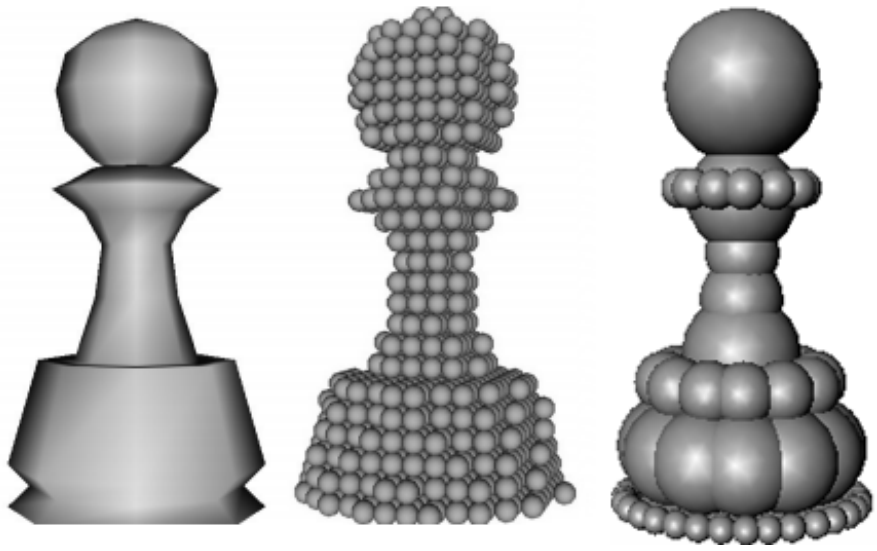


Figura 5.10: Corpo rígido representado por partículas, de tamanhos iguais (meio) e de tamanhos diferentes (direita).

Detecção de colisões

O esquema de detecção de colisões apresentado na seção 5.1 é utilizado, no entanto, neste caso, as partículas são esferas de tamanhos diferentes. Assim, utilizamos uma grade onde o tamanho de cada célula (*voxel*) é igual ao diâmetro da maior partícula. O processo de detecção continua igual, considerando apenas uma maior carga computacional, devido ao aumento no número de partículas que podem ocupar uma dada célula.

Resposta às colisões

O processo de detecção de colisão verifica se uma partícula colide com seus vizinhos locais. Quando há colisão, um impulso j é calculada para ser aplicada conforme as equações 3.4 e 3.5. Note que o ponto de colisão para aplicar um torque é um ponto na superfície da esfera em colisão, que está localizado na reta que passa pelos centros das esferas em colisão. A figura 5.11 ilustra este conceito.

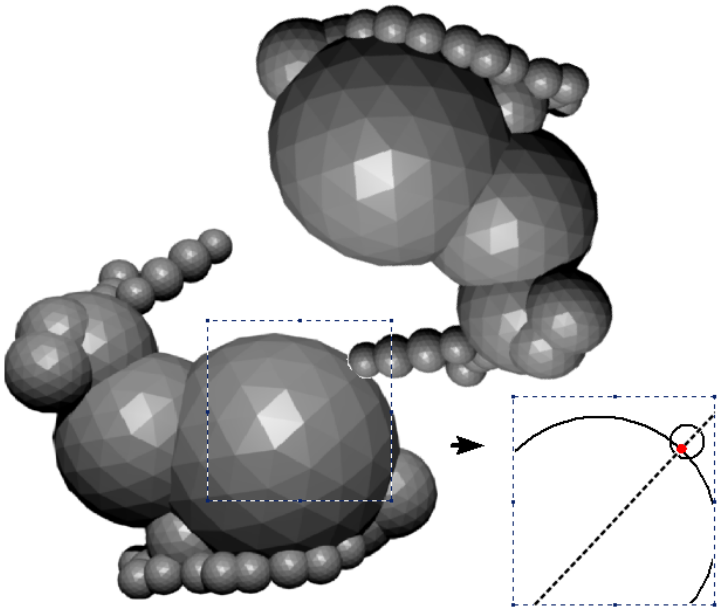


Figura 5.11: Aproximação de um ponto de colisão na superfície da esfera.

Simulação

A simulação de um passo de tempo consiste no processamento das seguintes etapas:

1. guardar estados dos objetos;
2. integrar posições e velocidades dos objetos;
3. atualizar partículas;
4. atualizar a grade;
5. detectar e processar colisões;
6. recuperar estados guardados dos objetos;
7. integrar velocidades dos objetos;
8. detectar e processar contatos;
9. integrar posições e orientações.

Em resumo, as diferenças com a técnica apresentada na seção 5.2 são:

- As colisões são detectadas predizendo para onde os objetos se moverão no seguinte passo de tempo, movendo-os temporariamente ali para verificar interferência [18]. Se há colisão um impulso é aplicado utilizando a velocidade corrente. O mesmo esquema é utilizado para tratar contatos, com a exceção de usar a velocidade nova para aplicar um impulso.

- As partículas são de tamanhos diferentes, o que permite ter uma interação mais natural entre objetos, uma vez que as partículas estão posicionadas conforme a superfície do objeto.

- Como as partículas (esferas) podem ter tamanho variável, algumas podem ocupar uma grande parte do objeto. Por tanto, ao invés de aplicar forças no centro das partículas, impulsos são aplicados em pontos na superfície das esferas (figura 5.11).

Renderização

Como no caso descrito na seção anterior, cada corpo rígido está associado a uma malha poligonal e é renderizado usando a posição X_j do centro de massa e a orientação Q_j utilizando malhas poligonais correspondentes aos modelos que já estão carregados na memória da GPU.

5.3 Simulação de objetos deformáveis

O método de simulação baseado em física usando partículas apresentado na seção 5.2 gera uma animação realista e de comportamento físico plausível. Assim, é possível estender essa abordagem para simular objetos deformáveis utilizando o método apresentado na seção 3.2, onde os objetos são representados por pontos. A idéia é utilizar partículas no lugar de pontos para representar os objetos.

5.3.1 Representação da forma dos objetos

A representação utilizada na seção 5.2 pode também ser utilizada na simulação de objetos deformáveis. Neste caso, são usadas partículas (pequenas esferas de tamanho igual) posicionadas nos vértices da malha. A figura 5.12 mostra a representação de um objeto por um conjunto de esferas posicionadas nos vértices da malha.

Figura 5.12: Conjunto de partículas posicionadas nos vértices de uma malha.

5.3.2 Detecção e resposta de colisão

Apenas forças de repulsão que permitem separar as partículas precisam ser consideradas. Logo, estas forças são utilizadas no esquema de integração semi-implícito apresentado na seção 3.2, na parte de simulação de objetos deformáveis.

5.3.3 Atualização da forma dos objetos

A abordagem para simulação de objetos deformáveis apresentada no capítulo 3.2 permite encontrar uma transformação ótima que aproxima uma nova posição e orientação para um conjunto de pontos que representa um objeto. Uma vez obtida a transformação, os pontos são movidos para a posição objetivo aplicando um modelo de deformação.

5.4 Hidrodinâmica suavizada com partículas em GPUs

Na simulação computacional de fluidos a carga computacional é bastante elevada, especialmente quando se quer simular fluidos de superfície livre em tempo real. Para este propósito, o método conhecido como Hidrodinâmica Suavizada com Partículas (do inglês, *Smoothed Particles Hydrodynamics-SPH*) [102, 103], é bastante utilizado

para simular fluidos de superfície livre, já que os cálculos das equações da mecânica de fluidos são muito mais rápidos de ser realizados do que os métodos tradicionais Eulerianos que usam grades e/ou elementos finitos. O método SPH pode ser implementado eficientemente na GPU e permite uma interação mais natural com outros objetos na cena, como sólidos e objetos elásticos. Mais ainda, com GPUs atuais podem-se obter ganhos significativos em tempo de execução comparado com CPUs. Este ganho é obtido pelo uso de uma grade uniforme para mapear as partículas, o que permite fazer uma busca rápida, e encontrar partículas vizinhas (ver seção 5.2). Como resultado, a simulação de fluido usando o método SPH é acelerado, e líquidos com dezenas de milhares de partículas podem ser simulados em tempo real.

5.4.1 Equações para a dinâmica de fluidos

As equações que governam o escoamento incompressível de fluido são a equação de conservação de massa e da equação de conservação de momento, chamadas de equações de Navier-Stokes.

$$\frac{D\rho}{Dt} = 0, \tag{5.18}$$

$$\frac{DU}{Dt} = -\frac{1}{\rho}\nabla P + v\nabla^2 U + g, \tag{5.19}$$

onde ρ, U, P, v, g são a densidade, velocidade, pressão, coeficiente de viscosidade e aceleração gravitacional, respectivamente.

5.4.2 Discretização do espaço ocupado pelo fluido

SPH é um método de interpolação para sistemas de partículas, onde quantidades de campo, que são definidas nas posições de partículas, podem ser avaliadas em qualquer lugar no espaço ocupado pelo fluido. Para este propósito, SPH distribui quantidades numa vizinhança local de cada partícula usando uma função radial simétrica de suavização. Assim, uma quantidade escalar ϕ é interpolada numa localização x por uma soma ponderada de contribuições de todas as partículas numa vizinhança de raio h.

$$\phi(x) = \sum_j m_j \frac{\phi_j}{\rho_j} W(x - x_j, h), \tag{5.20}$$

onde j percorre todas as partículas na vizinhança e m_j, ρ_j, x_j, ϕ_j são a massa, densidade, posição e uma medida do campo para a partícula j, respectivamente. A função $W(x - x_j, h)$ é uma função de suavização no raio h.

Os termos de massa e densidade de partículas aparecem na equação 5.20 uma

vez que uma partícula i representa um volume dado por m_i/ρ_i. Enquanto que a massa m_i é constante, e em nosso caso igual para todas as partículas, a densidade ρ_i varia e precisa ser computada a cada passo de tempo. Assim, podemos computar a densidade da partícula na posição x substituindo ϕ_j por ρ_j na equação 5.20:

$$\rho(x) = \sum_j m_j W(x - x_j). \tag{5.21}$$

A pressão do fluido é calculada usando a equação:

$$p = p_0 + \kappa(\rho - \rho_0), \tag{5.22}$$

onde p_0, ρ_0 são a pressão de repouso e a densidade, respectivamente e κ é uma constante.

Logo, para calcular a conservação de momento, os operadores gradiente e laplaciano devem ser modelados, já que são usados para resolver as forças de pressão e de viscosidade nas partículas. Assim, a força de pressão F_p e a força de viscosidade F_v são computadas como:

$$F_p = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W_p(r_{ij}), \tag{5.23}$$

$$F_v = v \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W_v(r_{ij}), \tag{5.24}$$

onde $r_{ij} = r_j - r_i$ é o vetor de posição relativa e r_i e r_j são as posições das partículas i e j, respectivamente. As funções de peso W utilizadas por Müller[59] são também utilizadas nesta abordagem. As funções ponderadas para pressão, viscosidade e outras condições são.

$$\nabla W_p(r) = \frac{45}{\pi h^6}(h - |r|)^3 \frac{r}{|r|} \tag{5.25}$$

$$\nabla W_v(r) = \frac{45}{\pi h^6}(h - |r|) \tag{5.26}$$

$$W(r) = \frac{315}{64\pi h^9}(h^2 - |r|^2)^3 \tag{5.27}$$

Nestas funções o valor fora do raio h é 0.

5.4.3 Condição de Fronteira

Pressão

A força de pressão leva o líquido a uma densidade constante, quando se está computando as equações de fluxo incompressível, mantendo as partículas a uma distância d, que é a distância de repouso. Na fronteira, se assume que uma partícula i está a uma distância $\|r_{iw}\|$, sendo $\|r_{iw}\| < h$. A força de pressão empurra a partícula i de volta para uma distância d na direção de $n_w(r_i)$, que é o vetor normal na superfície da fronteira. Assim, a força de pressão $F_{p,i}$ é modelada como:

$$\begin{aligned} F_p &= m_i \frac{\Delta x_i}{dt^2} \\ &= m_i \frac{(d - |r_{iw}|)n(r_i)}{dt^2} \end{aligned} \tag{5.28}$$

Densidade

Quando uma partícula está perto da superfície da fronteira a uma distância menor que h, a contribuição da fronteira para a densidade da partícula é estimada supondo que há partículas dentro da superfície, e é calculada como:

$$\rho_i(r_i) = \sum_{j \in fluid} m_j W(r_{ij}) + \sum_{j \in wall} m_j W(r_{ij}). \tag{5.29}$$

A distribuição de partículas na fronteira é determinada supondo que estão alocadas uniformemente de forma perpendicular à superfície e a curvatura média é 0, como mostrado na figura 5.13. Assim, a contribuição da fronteira é uma função da distância $\|r_{iw}\|$.

$$\rho_i(r_i) = \sum_{j \in fluid} m_j W(r_{ij}) + Z^{\rho}_{wall}(|r_{ij}|) \tag{5.30}$$

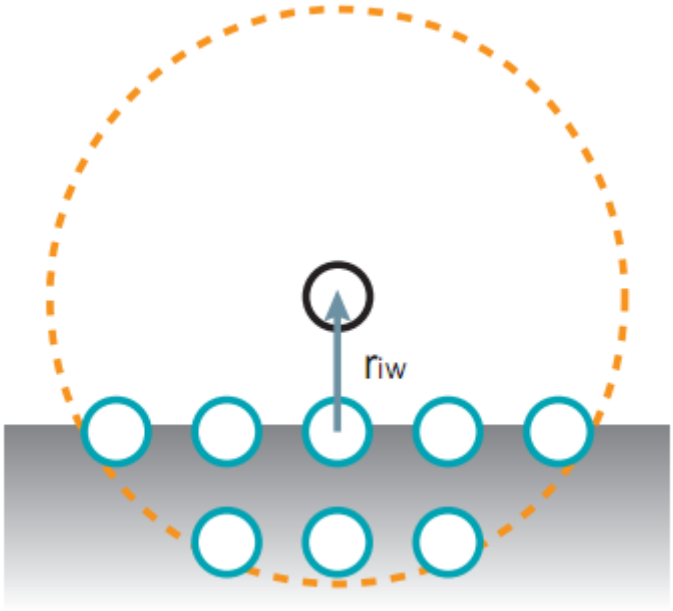


Figura 5.13: Distribuição uniforme de partículas dentro/atrás da superfície da fronteira (imagem extraída de [3]).

A função de ponderação $Z^{\rho}_{wall}(\|r_i w\|)$ da superfície da fronteira depende da distância $\|r_i w\|$ que pode ser pré-computada e usada diretamente na simulação. Logo, o valor da função numa posição arbitrária é calculada por uma interpolação linear. De modo semelhante, para obter a distância de cada partícula à superfície da fronteira é usada uma função de distância que é pré-computada e guardada na memória da GPU para ser utilizada durante a simulação.

5.4.4 Implementação na GPU

A cada passo de tempo, a simulação SPH é realizada em quatro etapas.

1. Atualização da grade
2. Computação da densidade
3. Atualização da velocidade
4. Atualização das posições das partículas

O fluxograma é mostrado na Figura 5.14.

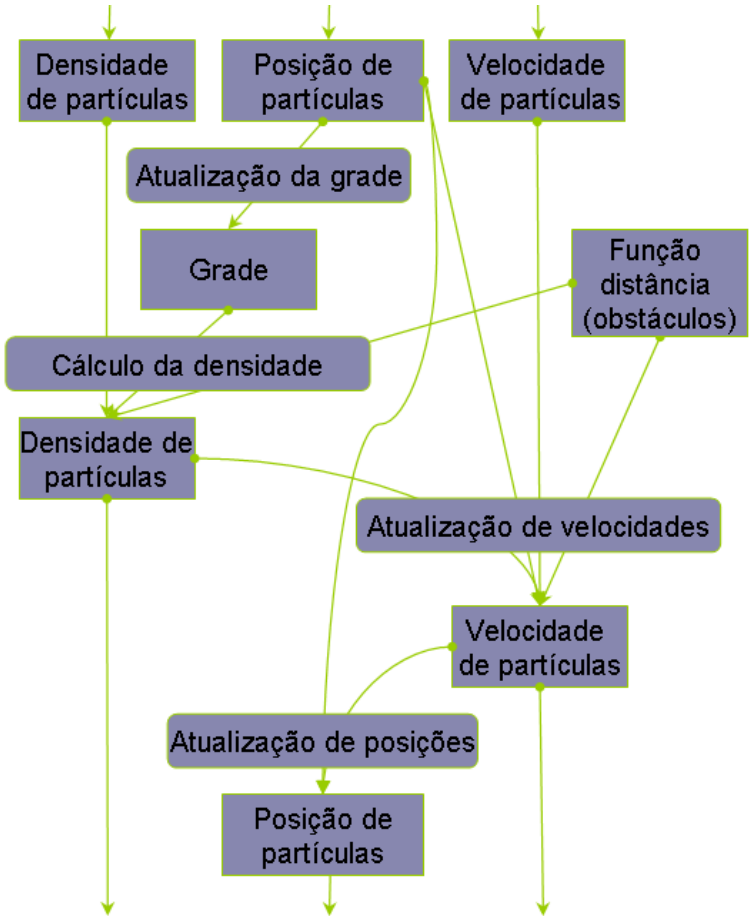


Figura 5.14: Fluxograma para uma iteração de simulação SPH.

Estrutura de Dados

De modo semelhante à simulação de corpos rígidos apresentada na seção 5.2, para simular líquido são armazenados na memória da GPU os seguintes valores:

- Atributos das partículas do fluído
  - Posição.
  - Velocidade.
  - Densidade.
- Atributos para o domínio.
  - Grade.
  - Função de distância e densidade para obstáculos.

Geração e atualização da grade

A geração e atualização da grade é feita como na simulação de corpos rígidos (seção 5.2, onde é utilizado o algoritmo de ordenação *RadixSort* que permite agrupar

partículas pelo índice da célula (voxel) onde se encontram. Uma vez ordenadas as partículas, para cada partícula, apenas será necessário computar a contribuição de funções ponderadas de partículas que se encontram numa vizinhança de 27 voxels e que estejam dentro do raio $\|r_{ij}\| < h$.

Cálculo da densidade

Para calcular a densidade de cada partícula, a equação 5.27 é usada. Os índices de partículas vizinhas à partícula i podem ser encontrados fazendo uso da grade, que mantém um conjunto de partículas por voxel. Então, a densidade da partícula i é calculada através da soma ponderada de valores físicos das partículas vizinhas. Por outro lado, se a partícula i está perto de uma fronteira ou obstáculo, a uma distância menor que o raio h, a contribuição da fronteira à densidade da partícula i é obtida da função de distância e densidade pré-computada. Este processo é ilustrado na figura 5.15, onde a contribuição de densidade da fronteira é adicionada à densidade da partícula i.

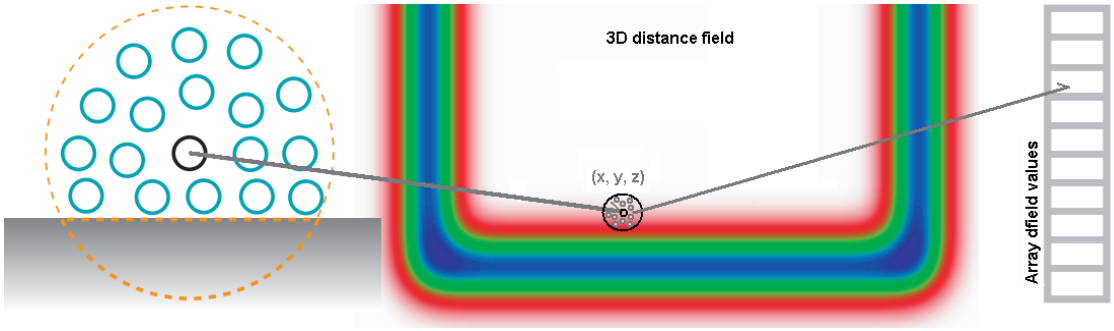


Figura 5.15: Função de distância e densidade para a superfície da fronteira.

Atualização da velocidade

Para calcular as forças de pressão e de viscosidade, as partículas vizinhas têm de ser buscadas novamente. O procedimento é o mesmo que para o cálculo da densidade. Estas forças são calculadas através das equações 5.23 e 5.24. Adicionalmente, caso uma partícula esteja perto da fronteira, a força de pressão da superfície é calculada utilizando a função de distância pré-computada.

5.4.5 Atualização da Posição

Tendo a velocidade atualizada, a posição é calculada com um esquema de integração de Euler explícito.

$$x_i^{'} = x_i + v_i dt, \tag{5.31}$$

onde x_i e v_i são a posição anterior e a velocidade anterior da partícula i, respectivamente.

5.5 Interação de líquido e tecido

Como já foi mencionado, a dinâmica de fluidos é complexa de ser modelada e por conseguinte criar animações manualmente torna-se inviável. Mais ainda, quando se quer simular a interação de objetos sólidos e fluido a complexidade aumenta. Embora a simulação baseada em física tenha sido amplamente estudada na área de mecânica computacional, frequentemente os métodos desenvolvidos neste campo não são aplicáveis para aplicações em tempo real, já que nessa área se coloca grande ênfase na precisão dos resultados, mas não na velocidade de execução. Assim, o método *SPH* (descrito na seção anterior) tenta simplificar os métodos exatos a fim de aumentar a eficiência computacional e executar simulações em tempo real. Nesta seção apresenta-se uma técnica que permite a interação entre líquido e objetos elásticos (tecidos) representados por malhas de triângulos.

5.5.1 Simulação de fluidos

Na seção 5.4 foram apresentadas as equações que governam o fluxo incompressível de fluido (equações 5.18 e 5.19). O fluido é representado por um conjunto de partículas e as equações governantes são resolvidas usando o método *SPH*. Apesar do método *SPH* não resolver a equação 5.18 devido à compressibilidade inerente ao método, essa compressibilidade pode ser reduzida para efeitos de conservação do volume. Os valores físicos para uma partícula na posição x são calculados pela soma ponderada dos valores das partículas à sua volta utilizando as equações 5.20, 5.25 e 5.27.

5.5.2 Simulação de tecido

A técnica de simulação para objetos elásticos utilizado é um modelo *massa-mola* onde as partículas estão conectadas por molas, conforme mostrado na figura 5.16. Para a integração do tempo, o método de Verlet [21] é empregado. Este método é adequado para simulação em tempo real, já que é menos custoso computacionalmente que outros métodos e tem melhor estabilidade numérica do que o método explícito de Euler. Assim, a posição x^{t+dt} da partícula i no tempo $t+dt$ é calculado pela seguinte equação:

$$x_i^{t+dt} = x_i^t + dt(x_i^t - x_i^{t-dt}) + \frac{F_i}{m_i}dt^2, \tag{5.32}$$

onde m_i, d, dt são a massa da partícula i, coeficiente de amortecimento e passo de tempo, respectivamente. Uma força externa F_i consiste da gravidade $m_i g$, força de mola $F_{i,spring}$, e à força do fluido $F_{i,fluid}$. A força da mola $F_{i,spring}$ é calculada como o somatório de todas as molas incidentes na partícula i:

$$F_{i,spring} = \sum_{j \in N_{adj}} k_{adj}(|x_{ij} - l_{adj}|)\frac{x_{ij}}{|x_{ij}|}, \tag{5.33}$$

onde k_{adj}, l_{adj} e x_{ij} são o coeficiente de mola, o comprimento de repouso das molas ligando as partículas adjacentes, e a posição relativa da partícula j em relação à partícula i, respectivamente.

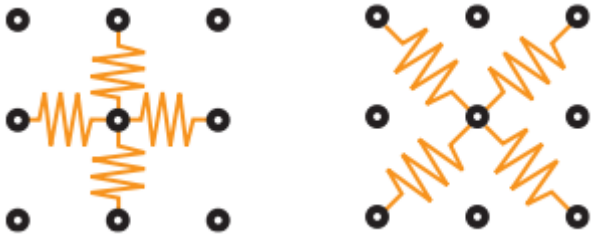


Figura 5.16: Molas que conectam partículas vizinhas para a modelagem de tecido (imagem extraída de [3]).

5.5.3 Uso da grade 3D

Para a interação entre as partículas do líquido e objetos elásticos é usada a técnica apresentada em [3], onde para cada partícula procura-se o triângulo mais próximo do objeto elástico. Para implementar esta idéia, é necessário o uso de outra grade uniforme onde são guardados identificadores de triângulos de objetos elásticos. Nesta segunda grade, cada voxel armazena índices de triângulos cujo centróide se encontra dentro do voxel. Assim, duas grades são usadas, uma para mapear partículas de fluido e outra para mapear triângulos de objetos elásticos (ver figura 5.17). Nesta implementação, queremos simular a interação de tecidos com o líquido, assim, os triângulos são de tamanho e forma iguais no estado de repouso, e durante a simulação, se espera que não sofram mudanças consideráveis devido às propriedades físicas do tecido.

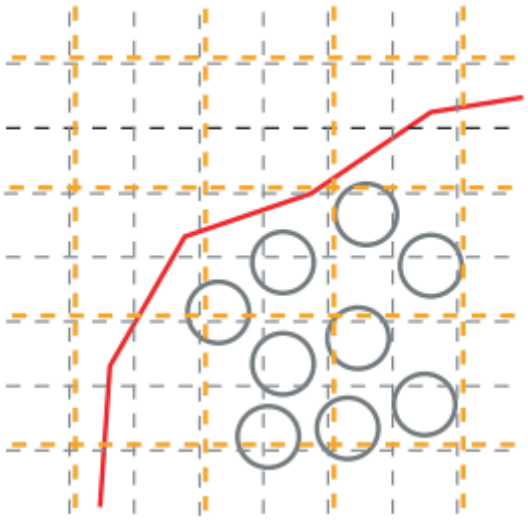


Figura 5.17: Duas grades são utilizadas para a interação entre líquido e tecido (imagem extraída de [3]).

5.5.4 Detecção de colisão

A detecção de colisão para a simulação do líquido foi explicada na seção 5.4; entretanto, para a interação entre líquido e tecido devem ser detectadas interferências entre partículas e triângulos. Primeiramente, para cada partícula do líquido, procura-se o triângulo mais próximo, pois a distância ao tecido é a distância ao triângulo mais próximo do tecido. Embora esta operação possa ser computacionalmente custosa, a segunda grade permite reduzir a busca ao conjunto de triângulos cujos índices estão armazenados em voxels adjacentes ao voxel na qual a partícula se encontra. Assumindo que o tecido tem uma determinada espessura, a força de interação entre uma partícula de fluido e um triângulo de tecido é modelado como uma força proporcional à profundidade de penetração. No entanto, o campo de força no domínio computacional torna-se descontínuo nas conexões de polígonos quando apenas a distância a uma face da malha é computada. O caso é ilustrado no lado esquerdo da figura 5.18). Esta descontinuidade pode produzir movimentos indesejados das partículas do fluido. O problema pode ser superado, calculando a distância exata ao tecido. Entretanto, uma vez que o cálculo de distância exata ao triângulo requer calcular distâncias a arestas e vértices, aumentando o custo computacional, optou-se utilizar uma distância aproximada ao tecido. Assumindo que existem n triângulos $T = t_0, t_1, \cdots, t_{n-1}$ próximos a uma partícula, calculamos distâncias d_c para os centros gravitacionais de triângulos na vizinhança, o triângulo com menor valor d_c é escolhido como o triângulo $t_{closest}$. Considerando que os triângulos possuem três vértices $v_{0,j}$, $v_{1,j}$ e $v_{2,j}$, $t_{closest}$ é encontrado pela seguinte equação.

$$t_{closest} = min_{t_j \in T}(x - \frac{v_{0,j} + v_{1,j} + v_{2,j}}{3})^2 \tag{5.34}$$

A detecção de colisão e auto-colisão entre tecidos segue o mesmo principio. Para cada vértice de um tecido é feita uma busca pelo triângulo mais próximo (fazendo uso da segunda grade). Se a distância é menor que uma constante δ, é gerada uma força que permite separar o vértice do triângulo, na direção da normal do triângulo a uma distância δ. O cálculo desta força é feita usando a idéia apresentada no trabalho de Jakobsen [21] e é aplicada no passo de integração.

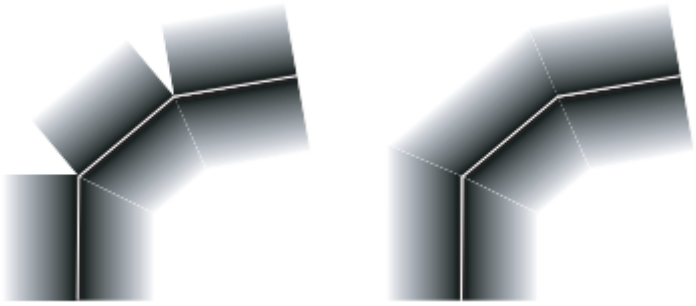


Figura 5.18: Campo de força no tecido: descontínua (esquerda) e contínua (direita). As imagens foram extraídas de [3].

Assumindo que os vértices do tecido têm massa igual, a posição do centro gravitacional pode ser calculada como a média dos três vértices, e a distância d_p para o triângulo $t_{closest}$ é calculada a partir das posições de vértices da seguinte forma.

$$d_p = |(v_{1,j} - v_{0,j}) \times (v_{2,j} - v_{0,j})\dot{(}x - v_{0,j})| \tag{5.35}$$

Como d_p é contínua no domínio computacional, o campo de força calculado é também contínuo. Logo, uma partícula de líquido colide com o tecido, quando a distância ao tecido é menor que a distância de repouso ε entre partículas.

5.5.5 Reação do líquido

O termo de pressão do líquido funciona como uma força que faz a densidade do líquido manter-se constante, ou seja, tenta manter as partículas na distância de repouso. Quando a distância d para o tecido é menor do que a distância de repouso ε, uma força é introduzida para levar a partícula de volta à distância ε. A força $f_{p,i}$ é calculada usando o vetor normal n do triângulo da seguinte forma:

$$f_{p,i} = m_i \frac{\varepsilon - d}{dt^2} n \tag{5.36}$$

Quando o tecido está dentro do raio de uma partícula, deve haver uma força de viscosidade. Suponha que as partículas são colocadas sobre a superfície de um tecido

e têm velocidades v e densidades ρ uniformes. A força de viscosidade do tecido é modelada como:

$$f^v is_{i,wall} = -\mu \frac{m}{\rho}(v_j - v_i) \sum_{j \in wall} \nabla W_v(r_{ij}) \tag{5.37}$$

Assumindo que as partículas no tecido são colocados de maneira uniforme, o tecido pode ser considerado como uma fronteira e a densidade que contribui à soma ponderada de uma partícula i é um valor pré-computado.

Assim, a contribuição para a densidade e viscosidade são estimadas. Dado que não existem partículas no tecido, a densidade de uma partícula de fluido perto do tecido é um valor pequeno, o que leva a uma concentração de partículas perto do tecido. A contribuição para a densidade do tecido é calculado pela equação 5.38 (assumindo que temos no termo de viscosidade, a contribuição da densidade $\rho_{i,wall}$ como uma soma ponderada destas partículas).

$$\rho_{i,wall} = m \sum_{j \in wall} W(r_{ij}) \tag{5.38}$$

5.5.6 Reação do tecido

A força que exerce o fluido num triângulo do tecido atua na direção do gradiente de pressão do fluido. Os movimentos de translação e rotação de um triângulo são obtidos a partir da força calculada. No entanto, a força não induz um movimento de rotação quando a direção é perpendicular ao triângulo. Nesse caso, a força do líquido pode ser calculada como uma força que trabalha no centro gravitacional do triângulo. Quando um tecido é composto por um número suficientemente grande de pequenos triângulos, a variação da pressão de um fluido sobre o triângulo é insignificante, e a direção do gradiente de pressão pode ser aproximado pela direção do vetor normal ao triângulo. Assim, o movimento rotacional induzido pelo fluido é dispensado.

A força F_i no triângulo i, que é a soma das forças f_j negativas que colidem com as partículas do fluido, é calculada como:

$$F_i = -\sum_{j \in colliding} f_j. \tag{5.39}$$

Considerando a descrição acima, a força do fluido não induz um movimento de rotação nos triângulos. Assim, a força F_i é distribuída uniformemente entre os vértices $f_{i,0}$, $f_{i,1}$ e $f_{i,2}$ pertencentes ao triângulo i.

$$f_{i,0} = f_{i,1} = f_{i,2} = \frac{F_i}{3} \tag{5.40}$$

Adicionalmente, a cada vértice de tecido, são somadas as forças geradas por colisões e auto-colisões entre tecidos.

5.5.7 Estruturas de dados utilizadas

Os valores físicos para o fluido e o tecido são armazenados em arrays de float4:

- Propriedades físicas das partículas do fluido
  - Posições.
  - Velocidades.
  - Densidades.
- Propriedades físicas para o tecido
  - Array de posições de vértices.
  - Array de posições anteriores de vértices.
  - Array de forças.
- Detecção de colisão, cálculo de pressão e viscosidade e atualização da velocidade.
  - Grade que guarda índices de partículas de líquido.
  - Função de distância e densidade para obstáculos.
  - Grade que guarda índices de triângulos de tecido.

5.6 Interação entre líquido e sólidos

Nas seções 5.4 e 5.2 foram descritas como são implementadas simulações de líquidos e corpos rígidos respectivamente. Note que em ambas as simulações, de líquido e corpos rígidos, é utilizado o mesmo sistema de detecção de colisão baseado no uso de uma grade uniforme. Assim, podemos simplesmente computar forças entre partículas e mudar as propriedades físicas do líquido e dos corpos rígidos que interagem na simulação.

No processo de detecção de colisão, partículas de corpos rígidos são consideradas como partículas de fluido, e o cálculo da força para cada partícula de corpo rígido consiste em computar $f_{i,fluid}$ e $f_{i,rigid}$, onde $f_{i,fluid} = f_i^{press} + f_i^{vis}$. Logo, cada corpo rígido j recebe uma força total F_j e um torque T_j que é o somatório de forças e

torques de todas as partículas que o compõem (equação 5.41).

$$\begin{aligned} F_j &= \sum_{i \in RB_j} (f_{i,rigid} + f_{i,fluid}) \\ T_j &= \sum_{i \in RB_j} r_i^{'} \times (f_{i,rigid} + f_{i,fluid}) \end{aligned} \tag{5.41}$$

Obtendo estas forças e torques, as posições e orientações dos objetos são atualizadas como descrito na subseção 5.2.1, e as partículas também atualizadas para suas novas posições.

Já no caso de partículas de líquido, a atualização de propriedades físicas é realizada utilizando o método SPH descrito na seção 5.4.

Capítulo 6

Renderização da superfície de líquidos

Métodos de simulação baseado em partículas tem se tornado populares para criar animações de fenômenos físicos tais como movimento de líquidos, mistura com sólidos, derretimento de sólidos, etc. Entretanto, extrair e visualizar uma superfície que delimita o volume de partículas não é trivial e resulta num processo custoso, especialmente quando há uma grande quantidade de partículas. De modo geral, um líquido pode fluir por todo o cenário, exceto quando está contido em algum tipo de recipiente. Frequentemente, nos métodos eulerianos, o domínio do fluido é definido como uma grade finita que permite resolver as equações de movimento e aproximar a superfície do líquido. Entretanto, uma simulação aceitável para esse método requer uma grade de resolução fina, o que resulta custoso no processamento e uso de memória. Assim, os métodos que usam partículas se adaptam melhor para aplicações interativas.

A seguir são apresentados métodos de extração de superfície comumente utilizados. No entanto, damos ênfase ao método apresentado por Laan [26], que é um método baseado no espaço da imagem, e apresenta melhores resultados para simulação de líquidos densos.

6.1 Renderização usando *marching cubes*

Nosso objetivo, nesta parte do trabalho, é a renderização da superfície de um líquido representado por partículas. Entretanto, uma renderização eficiente, visualmente realista e em tempo real ainda é difícil de conseguir. O primeiro desafio é encontrar a superfície do líquido. A maioria de técnicas adotam o uso de uma grade regular para manter uma isosuperfície atualizada, a cada passo de tempo durante uma simulação. Assim, a renderização da superfície do líquido é equivalente à renderização de uma

isosuperfície.

A base para a renderização de uma isosuperfície é uma representação discreta do campo escalar correspondente à densidade do fluido. A técnica de *marcha de cubos* [25], utiliza uma amostragem da função de densidade realizada nos vértices de uma grade regular. Foi realizado uma implementação desta técnica onde, além da grade que guarda os valores do campo escalar do volume do fluido, uma grade que representa um campo vetorial correspondente ao gradiente dos vértices da superfície é usada.

Esta implementação foi feita na plataforma CUDA e consta de 5 etapas.

- A grade que representa o campo escalar da densidade do fluido é gerada usando as partículas SPH.
- A grade é avaliada nos vértices dos voxels e computa o número de vértices que cada voxel ira gerar.
- Os voxels que geraram triangulaçoes não nulas são selecionados.
- Os voxels não nulos são avaliados para gerar triângulos. dependendo da quantidade de vertices gerados no passo 2. Neste passo são atualizados dois arrays: de posições e normais de vértices da malha gerada. Não se sabe a priori quantos vértices serão gerados, a cada passo de tempo, por tanto, se usa um valor máximo estimado a partir das dimensões da grade.
- Renderização da malha de triângulos. Os arrays de posições e normais estão ordenados sequencialmente e mapeados na memória da GPU, e cada intervalo de três elementos, nos arrays de posições e normais, representa um triângulo. Assim, basta executar uma chamada à função $glDrawArrays(GL_TRIANGLES, 0, totalVerts)$.

A pesar de a técnica de marcha de cubos ser bastante utilizada para visualizar dados volumétricos estáticos, onde novos dados de isosuperfícies são gerados raramente, para a visualização de líquidos representados por conjuntos de partículas, não resulta ser uma técnica adequada. Como o tempo gasto para a geração/atualização da grade que representa a densidade volumétrica do fluido é muito alto, esta se converte num gargalo, comprometendo o desempenho da simulação. Um exemplo simples é mostrado na figura 6.1, onde 1000 partículas são renderizadas usando a técnica de *metaballs*. Pode se notar que a superfície das pequenas esferas (que representam partículas) se mostram facetadas, o que não é adequado para renderização de líquidos. Um outro exemplo (ver figura 6.2) tenta simular líquido num tanque cilíndrico. A simulação contém 10K partículas roda a 25fps numa GeForce GTS450.

Estes resultado, demonstram que, a pesar do processamento paralelo, o uso da técnica de marcha de cubos não é uma alternativa para ser usada em simulações de tempo real que requeiram grandes quantidades de partículas. O gráfico da figura 6.3 mostra os tempos gastos em cada etapa da geração de superfície usando a técnica de *metaballs* e marcha de cubos para o exemplo com 10K partículas. Note que o processo de geração de triângulos gasta muito mais tempo que o resto de processos, reduzindo consideravelmente o número de quadros por segundo numa simulação de líquido.

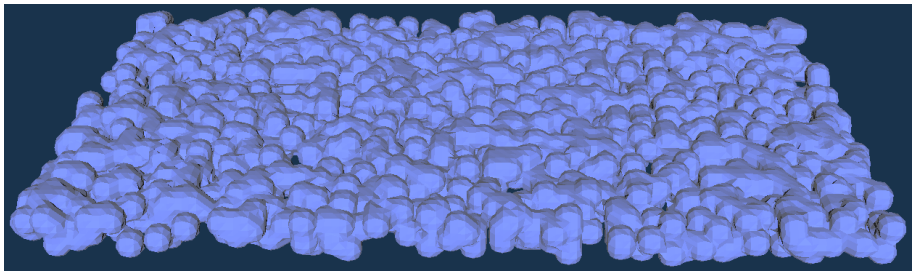


Figura 6.1: Superfície de partículas de fluido utilizando a técnica *metaballs*.

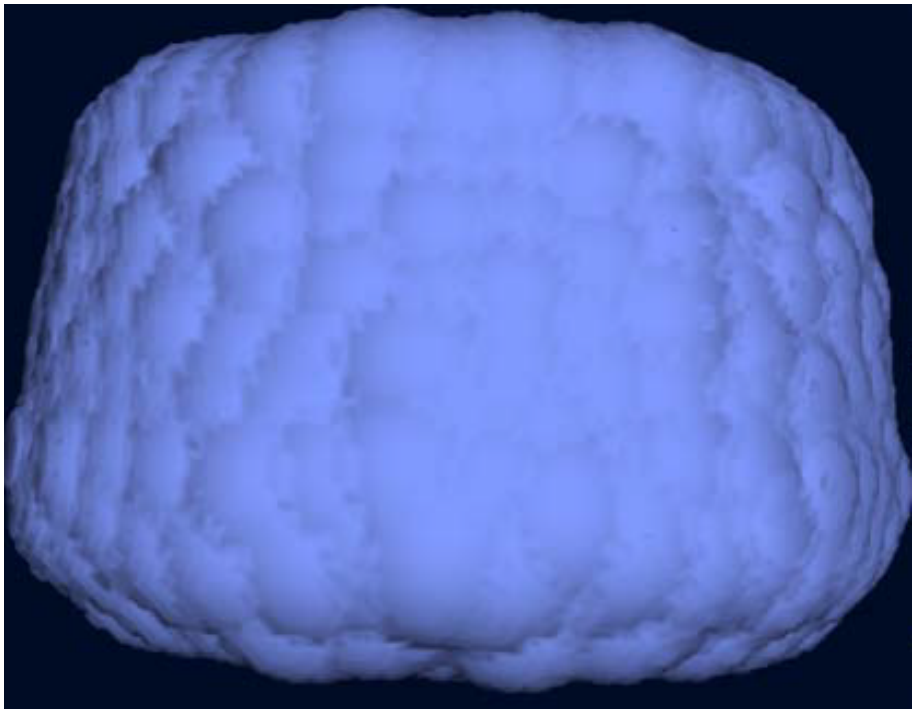


Figura 6.2: Superfície gerada usando a técnica marching cubes para um conjunto de 10k partículas.

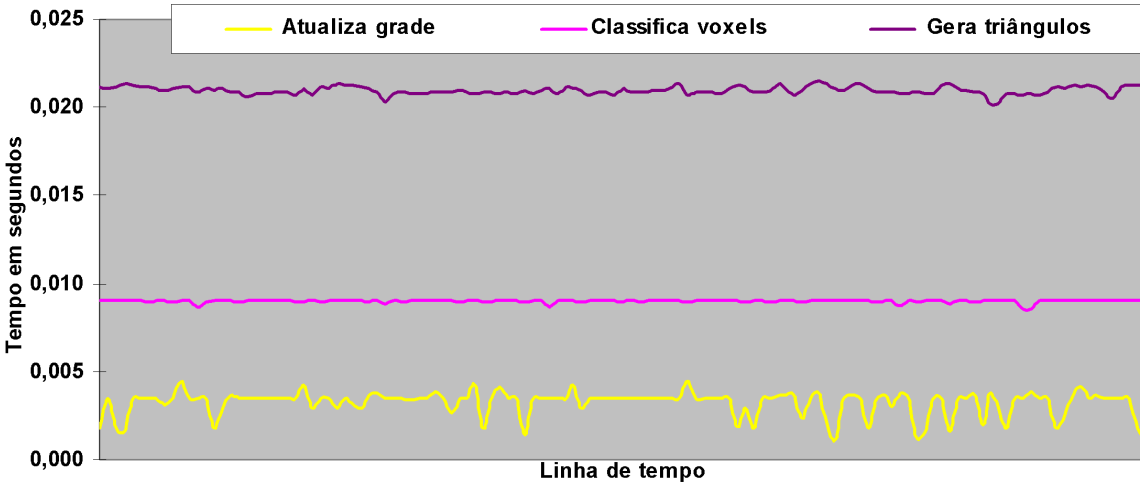


Figura 6.3: Tempos gastos em cada etapa na geração de superfície com a técnica de marcha de cubos para 10k partículas (ver figura 6.2).

6.2 Renderização de superfície usando o espaço-imagem

Laan et al. [26], apresentam uma abordagem para renderização de um líquido simulado com SPH utilizando várias dezenas de milhares de partículas em tempo real. O método não requer uma poligonização de uma isosuperfície. A renderização é feita apenas para áreas visíveis da superfície do líquido. Esta abordagem, em resumo, é uma adaptação do método para renderização de nuvens de pontos baseado em *splats*[1]. Esta abordagem, compreende as seguintes etapas.

- rasterizaçao da distância ao observador (profundidade) das partículas visíveis que representam a superfície do líquido numa imagem,
- suavização da imagem de profundidade da superfície,
- cálculo da espessura do líquido visível,
- renderização do líquido misturando-o com os objetos do cenário.

6.2.1 Profundidade da superfície do líquido

Primeiramente as partículas são renderizadas como pequenas esferas e o buffer de profundidade é gerado guardando valores das partículas que estão mais próximos à posição da camera. Para melhorar o desempenho, as esferas são renderizadas como *point sprites*[2] computando valores de profundidade num *fragment shader* (ver figura

[1]Primitiva geométrica representada por uma pequena elipse. Frequentemente, usada para renderização de superfícies, quando não existe uma malha de polígonos.

[2]É uma pequena imagem, que contém um quadrilátero contendo um formato de esfera

6.4). A saída deste processo é armazenada numa textura que é utilizada no processo de suavização.

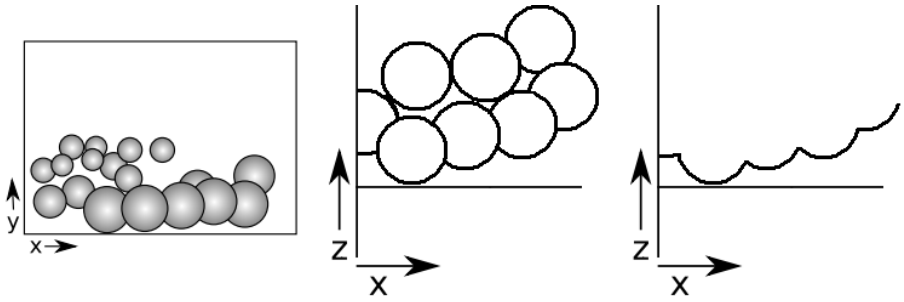


Figura 6.4: Computando a textura de profundidade das partículas visíveis.

6.2.2 Suavização da textura de profundidade na superfície do líquido

A textura com valores de profundidade da superfície do líquido deve ser suavizada, já que a renderização das esferas gera uma superfície do líquido com pequenas protuberâncias dependendo do tamanho das esferas. Estas protuberâncias podem ser disfarçadas utilizando um filtro gaussiano de suavização unilateral ou bilateral. A figura 6.5 ilustra a suavização de um conjunto de partículas após a computação da normal para cada pixel.

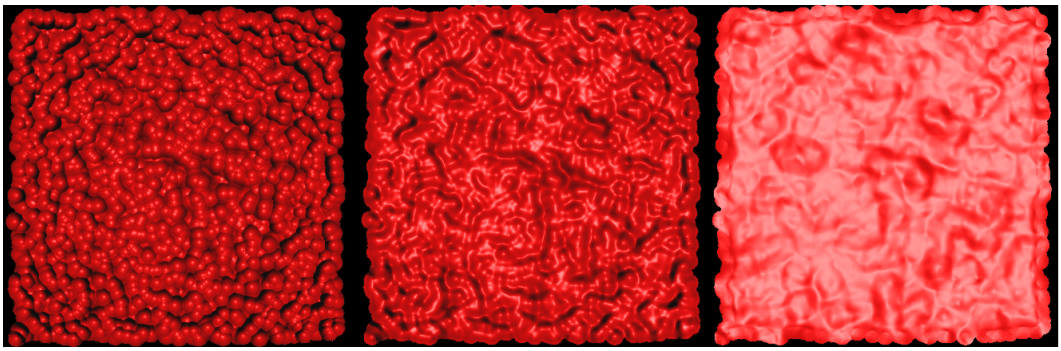


Figura 6.5: Suavização do buffer de profundidade das partículas visíveis.

O código abaixo permite realizar a suavização do buffer de profundidade. Note que os parâmetros r, $blurScale$ e $blurDepthFalloff$, podem ser mudados para obter resultados diferentes.

Listing 6.1: Shader GLSL para suavizar o buffer de profundidade das partículas SPH

```
uniform sampler2D particlesDepthTex;  // Contains the depth

const float scale = 1.0;
```

```
uniform float windowWidth;
uniform float windowHeight;
uniform float r = 3.0;
uniform float blurScale = 0.5;
uniform float blurDepthFalloff = 2;

const float zBackground = 0.01;

void main() {
    float center = texture2D(particlesDepthTex, gl_TexCoord[0].xy).x;
    vec2 texelSize = vec2 (scale/windowWidth, scale/windowHeight);
    if (center < zBackground) {
        gl_FragColor = vec4(0,0,0,1);
    } else {
        float sum = 0;
        float wsum = 0;
        for(float y=-r;y<=r;y+=1.0) {
            for(float x=-r;x<=r;x+=1.0) {
                float r2 = length(vec2(x,y));
                if (r2>r) continue;
                r2 *= blurScale/r;

                // Spatial gaussian weight
                float w = exp(-r2*r2);

                // Depth gaussian weight
                float g = 0;

                float sample = texture2D(particlesDepthTex, gl_TexCoord[0].xy + ↩
                    vec2(x, y)*texelSize).x;
                if (sample >= zBackground) {
                    r2 = (sample - center) * blurDepthFalloff;
                    g = exp(-r2*r2);
                }
                sum += sample * w * g;
                wsum += w * g;
            }
        }
        if (wsum > 0.0) {
            sum /= wsum;
        }
        gl_FragColor.x = sum;
    }
}
```

6.2.3 Cálculo da espessura do líquido visível

Num cenário com líquido, se espera que objetos sejam menos visíveis dependendo da quantidade de líquido que existe desde a posição da câmera. Para conseguir este efeito, a profundidade dos objetos ocluídos pelo líquido deve ser computada. Assim, a espessura do líquido é obtida fazendo a diferença entre as profundidades da superfície do líquido e a profundidade do objeto mais proximo à posição da camera. A figura 6.6 ilustra a transparência do líquido.

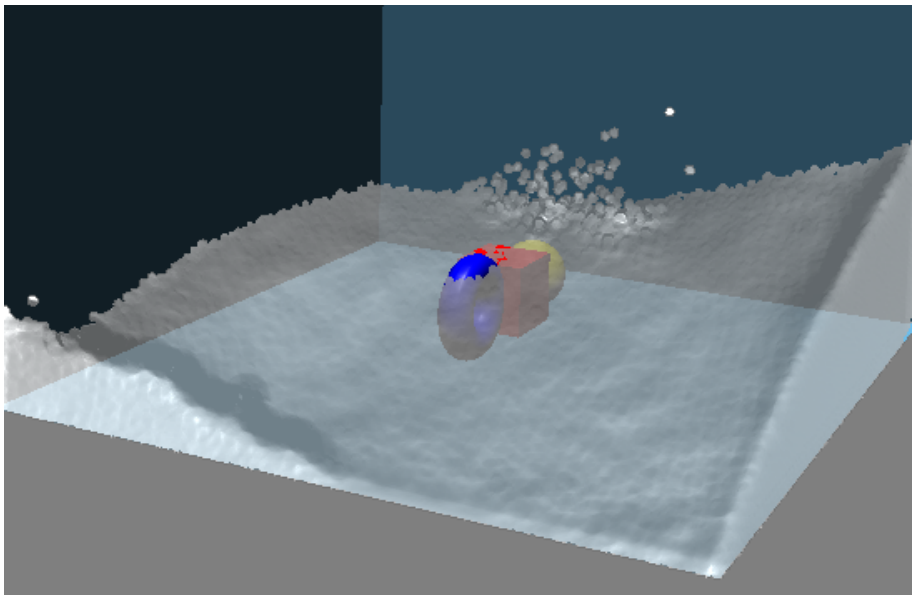


Figura 6.6: Transparência do líquido.

6.2.4 Renderização do cenário

Logo, os resultados parciais sao compostos para obter a imagem final. A cor na superfície visível do líquido é combinada com a cor dos objetos dentro e atrás do líquido, dependendo da distancia de profundidade, usando componentes *alpha*. Adicionalmente podemos aplicar uma equação *Fresnel* com componentes de reflexão e refração, como ilustrado na figura 7.34.

Capítulo 7

Resultados

Nesta tese foram apresentadas técnicas para animação baseada em física usando métodos tradicionais (capítulo 3,) bem como métodos baseados em partículas (capítulos 4 e 5). Resumindo, as técnicas implementadas foram:

- Abordagens tradicionais
  - Simulação de objetos rígidos usando impulsos 3.1.
  - Simulação de objetos deformáveis usando casamento de formas 3.2.
- Abordagens usando sistemas de partículas e GPUs.
  - Esquema de detecção de colisões grosseiro 4.1.
  - Simulação de objetos rígidos usando forças de repulsão 5.2.
  - Simulação de objetos rígidos usando impulsos 5.2.6.
  - Simulação de objetos deformáveis usando casamento de formas 5.3.
  - Simulação de líquidos usando SPH 5.4.
  - Simulação de objetos elásticos 5.5.
  - Acoplamento entre objetos líquidos e sólidos 5.6.

Ferramentas utilizadas

A implementação dos protótipos foi realizada usando as linguagens $C/C++$, $C-CUDA$ e $GLSL$; empregando as bibliotecas OpenGL [34] e GLUT [104]. Os protótipos foram compilados utilizando Ms-VC++ e o compilador NVidia CUDA nvcc. No sistema operacional Linux foi utilizado o GNU gcc.

Sistemas utilizados para testar os protótipos

Para medir o desempenho dos protótipos implementados foram utilizadas estações de trabalho em quatro configurações (PCs), que são apresentados na tabela 7.1.

Tabela 7.1: Configurações computacionais (PCs) empregadas para testar os protótipos implementados

Nome	Processador central	Memória CPU	Processador gráfico	Memória GPU
SistemaA	Intel Core 2 Duo E6750 2.66GHz	2Gb DDR2	NVidia GeForce 7600GT	512Mb DDR2
SistemaB	Intel Core i3-530 2.93GHz	4Gb DDR3	NVidia GeForce 8600GTS	512Mb DDR3
SistemaC	Intel Core i5-760 2.80GHz	4Gb DDR3	NVidia GeForce GT240	512Mb DDR3
SistemaD	Intel Core i7-960 3.20GHz	4Gb DDR3	NVidia GeForce GTS450	1Gb DDR5

Protótipos implementados

Desde o início deste trabalho foram implementados diversos protótipos para simulação baseada em física, que são divididos em dois grupos: os implementados que usam somente CPU e os que usam CPU e GPU. As tabelas 7.2 e 7.3 descrevem a aplicação para cada protótipo desenvolvido.

Tabela 7.2: Protótipos que empregam apenas CPU

Nome	Descrição da implementação	Linguagens usadas	Plataforma
RBI_{CPU}	Simulação de objetos rígidos usando impulsos	$C++$	CPU
$TDB1_{CPU}$	Simulação de objetos deformáveis usando casamento de formas e malhas tetraedrais para detecção de colisões	$C++$	CPU
$TDB2_{CPU}$	Otimização da simulação de objetos deformáveis usando propagação nas bordas	$C++$	CPU
$DBDF_{CPU}$	Otimização da simulação de objetos deformáveis usando funções de distância	$C++$	CPU

Tabela 7.3: Protótipos implementados que empregam CPU e GPU usando sistemas de partículas

Nome	Descrição da implementação	Linguagens usadas	Plataforma
BP_{GPU}	Detecção de colisão grosseira	$C++$ $C-CUDA$	GPU
RBF_{GPU}	Simulação de objetos rígidos usando forças de repulsão	$C++$ $C-CUDA$	GPU/CPU
RBI_{GPU}	Simulação de objetos rígidos usando impulsos	$C++$ $C-CUDA$	GPU/CPU
DB_{GPU}	Simulação de objetos deformáveis usando casamento de formas	$C++$ $C-CUDA$	GPU/CPU
Cl_{GPU}	Simulação de tecidos usando integração de Verlet	$C++$ $C-CUDA$	GPU/CPU
SPH_{GPU}	Simulação de líquidos usando SPH	$C++$ $C-CUDA$	GPU
ALL_{GPU}	Simulação para interação de líquidos e sólidos	$C++$ $C-CUDA$	GPU

Observe que alguns dos protótipos utilizam ambas plataformas CPU e GPU, isto é, a CPU é utilizado para fazer cálculos adicionais, e não apenas para invocar chamadas para executar processos na GPU.

7.1 Simulação de objetos sólidos em CPU

Estas abordagens foram implementadas usando estruturas de dados, algoritmos sequenciais e técnicas que são executados apenas em CPU.

7.1.1 Simulação de objetos rígidos

Foi implementado um protótipo, usando a abordagem de animação de objetos rígidos descrita na seção 3.1, que chamamos de RBI_{CPU}. Neste protótipo os objetos podem ser de complexidade arbitrária, e são tratadas configurações de empilhamento e múltiplos contatos. As figuras 7.1, 7.2, 7.3 e 7.4 mostram resultados obtidos usando este protótipo. Para medir os tempos, foi utilizado o *SistemaB*.

Simulações realizadas

A tabela 7.4 resume os experimentos realizados usando o protótipo RBI_{CPU}.

Tabela 7.4: Experimentos realizados no protótipo RBI_{CPU}

Nome	Protótipos utilizados	Descrição da simulação
$S1_RBI_{CPU}$	RBI_{CPU}	30 caixas caem em queda livre conseguindo uma configuração de empilhamento (figura 7.2).
$S2_RBI_{CPU}$	RBI_{CPU}	Simulação de uma espiral com 300 peças de domino (figura 7.3).
$S3_RBI_{CPU}$	RBI_{CPU}	Simulação da interação de 30 *Stanford bunnies* (figura 7.4).

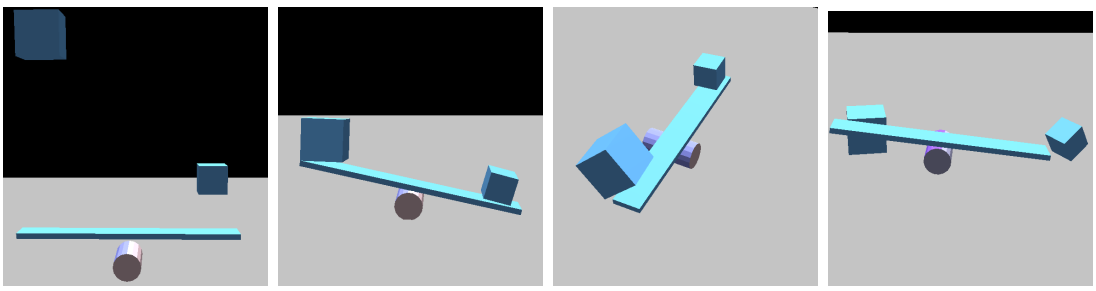


Figura 7.1: Um cenário simples envolvendo quatro objetos.

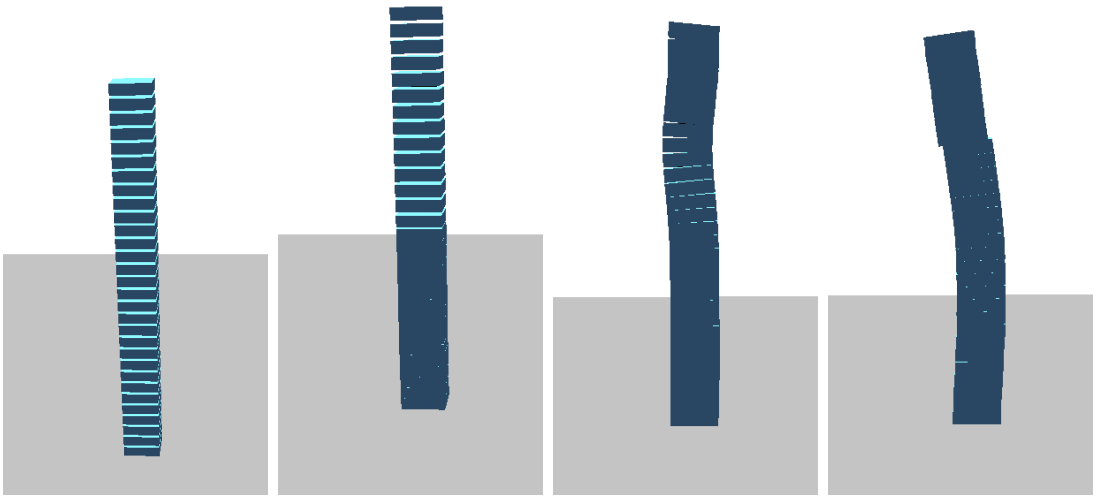


Figura 7.2: Simulando um cenário com 30 caixas empilhadas.

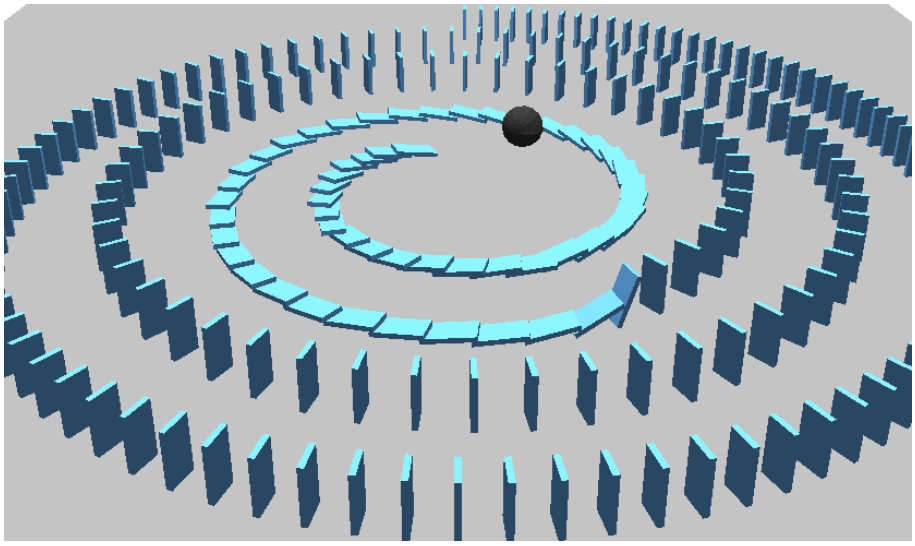


Figura 7.3: Simulação de um dominó com 300 peças.

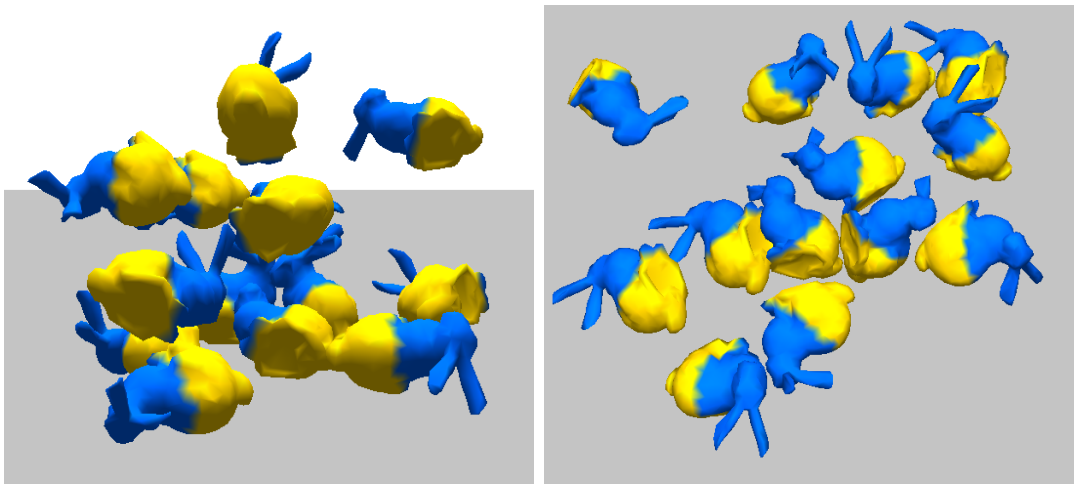


Figura 7.4: Cenário com objetos complexos: vista frontal(esquerda) e vista superior(direita).

O experimento $S1_RBI_{CPU}$ simula um cenário com 30 caixas sendo empilhadas. Neste exemplo uma caixa é um objeto simples que consta de 8 vértices e 12 triângulos. O gráfico da figura 7.5 mostra o desempenho da simulação quando o número de caixas aumenta de 1 até 30, e o gráfico da figura 7.6 mostra o número de quadros por segundo.

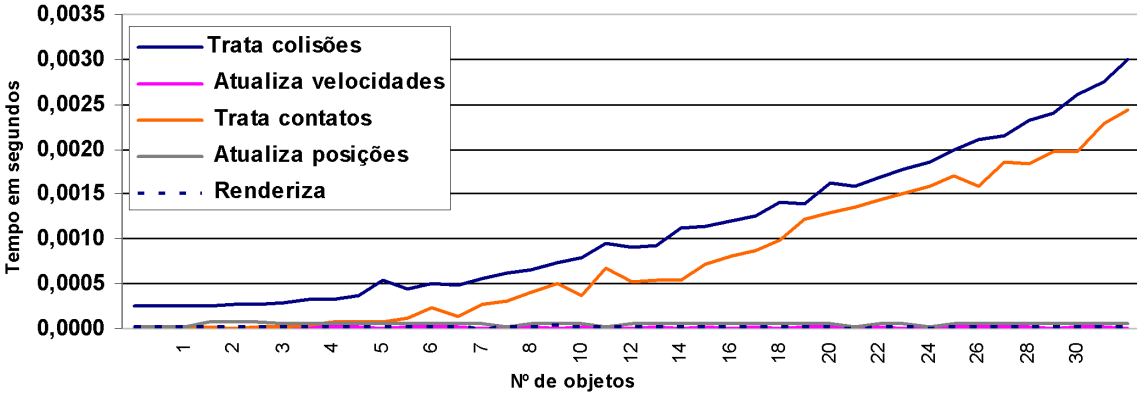


Figura 7.5: Tempo gasto em cada etapa da simulação para o experimento de empilhamento de caixas ($S1_RBI_{CPU}$).

Note que as linhas que não aparecem claramente na realidade correspondem a valores muito próximos de zero, o que representa tempos negligíveis com relação aos tempos gastos nos processos de detecção e tratamento de colisões. O experimento por ser relativamente simples, não apresenta um custo elevado de computação e é executado a taxas altas de quadros por segundo.

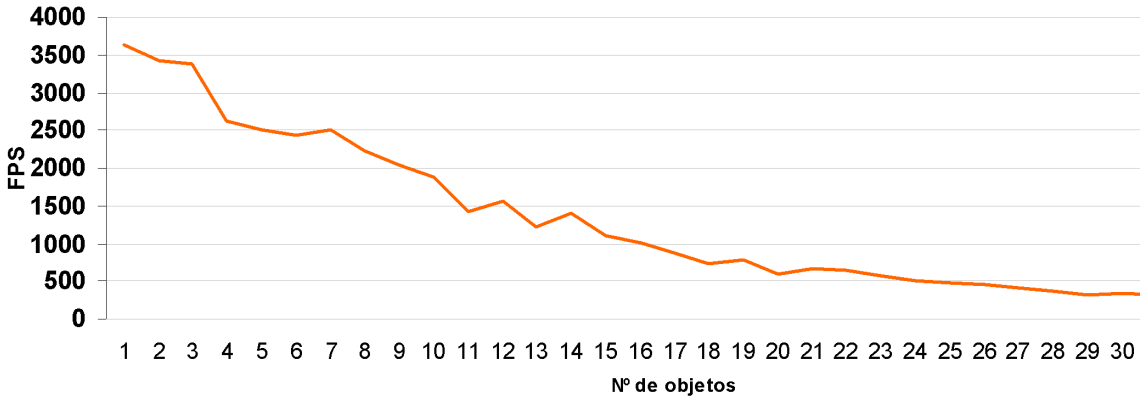


Figura 7.6: Quadros por segundo para o experimento de empilhamento de caixas ($S1_RBI_{CPU}$).

O experimento $S2_RBI_{CPU}$ simula um cenário que é constituído de uma espiral formada por 300 peças de dominó, que caem empurrando seus vizinhos na sequência. Cada peça de dominó é um paralelepípedo de 8 vértices e 12 triângulos. O gráfico da figura 7.7 mostra o desempenho da simulação no decorrer do tempo, e o gráfico da figura 7.8 mostra o número de quadros por segundo.

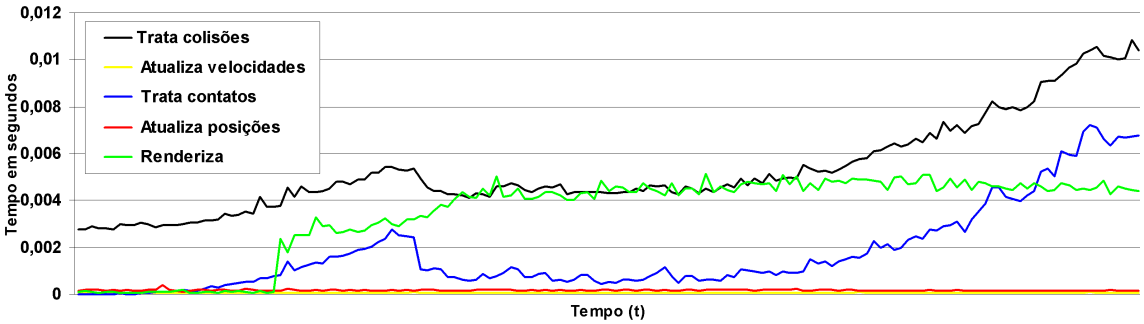


Figura 7.7: Tempo gasto em cada etapa da simulação de 300 peças de dominó.

Figura 7.8: Quadros por segundo para a simulação de 300 peças de dominó.

Este experimento, apesar de utilizar objetos de geometria similar a do exemplo anterior, contém uma quantidade maior de objetos pelo que a taxa de quadros por segundo cai consideravelmente. Também podemos notar que o tempo de renderização aumenta, pela quantidade de objetos que precisam ser transformados antes

de serem exibidos. É importante mencionar que o protótipo faz uso de funções de distância para a detecção de colisões, ao invés de fazer cálculos diretos de distância entre caixas 3D.

No experimento $S3_RBI_{CPU}$ é realizado uma simulação de 30 modelos (*Stanford bunny*) sendo amontoados. Neste exemplo, cada modelo é de 461 vértices e 918 triângulos. O gráfico da figura 7.9 mostra o desempenho da simulação quando o número de objetos aumenta, e o gráfico da figura 7.10 mostra o número de quadros por segundo.

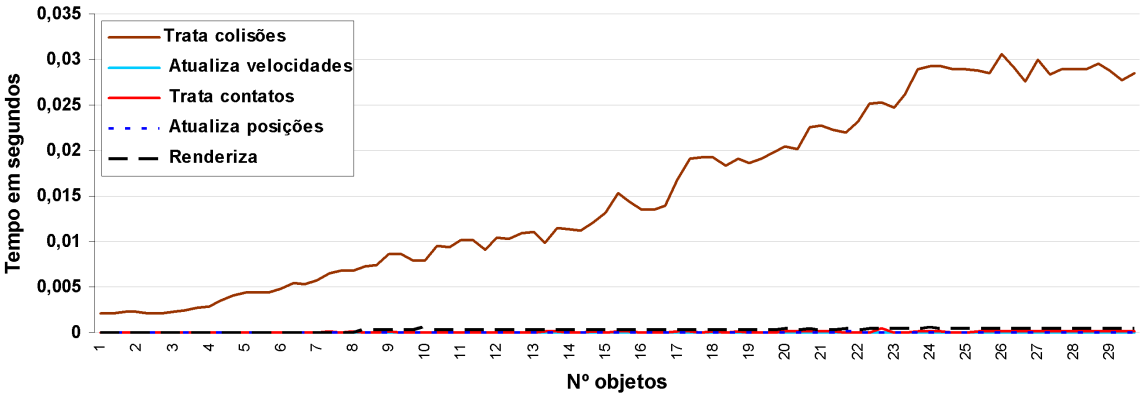


Figura 7.9: Tempo gasto em cada etapa da simulação para o cenário empilhamento de coelhos.

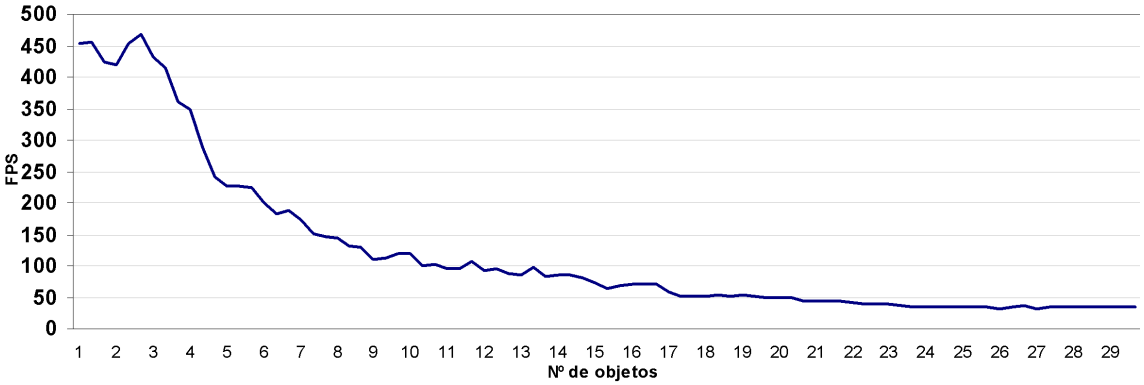


Figura 7.10: Quadros por segundo para a simulação de empilhamento de coelhos.

Observe que a taxa de quadros por segundo cai bastante com relação aos experimentos anteriores, já que o processo de detecção de colisões gasta muito mais tempo, pois, para cada par (A e B) de objetos em colisão potencial é necessário testar cada vértice de A (B) na função distância de B (A) e viceversa, para encontrar possíveis interpenetrações.

A tabela 7.5 mostra tempos para cada etapa da simulação deste experimento. Os dados foram coletados quando a simulação apresenta a maior taxa de processamento,

isto é, quando há o maior número de objetos em colisão. Os testes foram executados em três sistemas de computação.

Tabela 7.5: Tempo gasto, em *segundos*, para cada etapa da simulação do cenário com 30 *Stanford bunnies* (figura 7.3)

CPU-GPU	Trata colisões	Atualiza velocidades	Trata contatos	Atualiza posições	Renderiza	fps
SistemaB	0.0289940	0.0000070	0.0001370	0.0000310	0.0004770	34
SistemaC	0.0199010	0.0000037	0.0000463	0.0000178	0.0003059	49
SistemaD	0.0158810	0.0000021	0.0000237	0.0000102	0.0002447	62

Os resultados obtidos para esta implementação sequencial, em CPU, revelam que o desempenho aumenta pouco para arquiteturas mais modernas, o que motiva a procura por métodos mais eficientes e o uso de algoritmos que possam ser executados em paralelo.

7.1.2 Simulação de objetos deformáveis

Uma segundo grupo de protótipos foi implementado para simulação de objetos deformáveis (seção 3.2). Nestes protótipos foram empregados *Vertex Buffer Objects* do OpenGL para aprimorar o desempenho da simulação, uma vez que a forma dos objetos muda a cada instante de tempo. A tabela 7.6 descreve as simulações realizadas com estes protótipos. Para testar a eficiência destes experimentos foi utilizado o *SistemaA*.

Simulações realizadas

Tabela 7.6: Experimentos realizados nos protótipos para simulação de objetos deformáveis

Nome	Protótipos utilizados	Descrição da simulação
$S1_TDB_{CPU}$	$TDB1_{CPU}$	Simulação de 9 modelos: 3 *ducks*, 2 *Stanford bunnies* e 3 esferas (ver figura 7.11). A tabela 7.7 mostra a quantidade de elementos para cada modelo.
$S2_TDB_{CPU}$	$TDB1_{CPU}$	Simulação de 8, 18, e 27 esferas (ver figura 7.14).
$S3_DB_{CPU}$	$TDB1_{CPU}$, $TDB2_{CPU}$ e $DBDF_{CPU}$	Simulação de empilhamento de 8 toros (ver figura 7.15).

Com o propósito de avaliar o desempenho da técnica apresentada foram simulados alguns experimentos (ver as Figuras 7.11, 7.12 e 7.13). Os experimentos tentam medir o desempenho do protótipo implementado em função da quantidade de objetos e a complexidade de suas geometrias. A Tabela 7.7 mostra a resolução dos objetos usados nas simulações (número de vértices, número de faces e tetraedros).

Tabela 7.7: objetos com diferente resolução.

Objeto	vértices da superfície	total de vértices	faces	tetraedros
Stanford bunny	436	510	868	1750
duck	424	519	846	1819
esfera	386	729	768	2560

Nos experimentos da tabela 7.6, o coeficiente de rigidez α é de 0.8 (os objetos são quase rígidos).

Adicionalmente foram coletadas informações sobre o número de primitivas envolvidas na colisão (ver figura 7.13) que são processados a cada passo de tempo, e o porcentagem gasto em cada sub-processo (7.12) em milisegundos (ms).

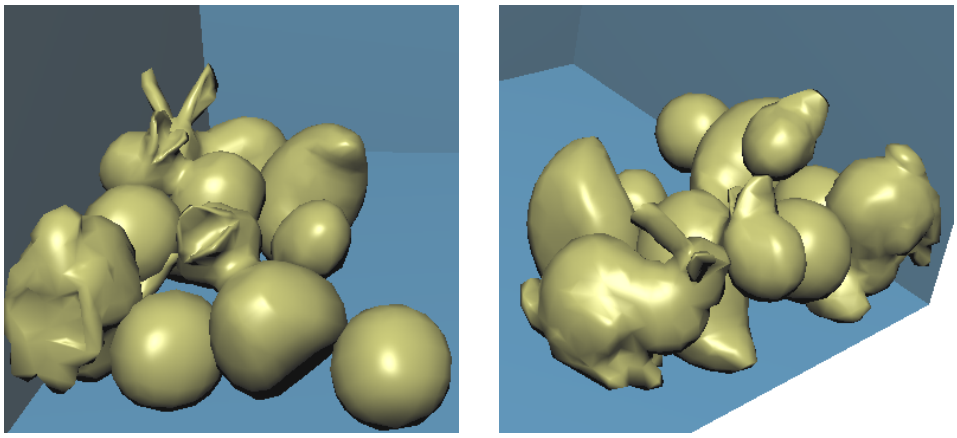


Figura 7.11: simulação de uma cena com objetos diferentes (veja $S1_TDB_{CPU}$ na tabela 7.6).

O experimento $S1_TDB_{CPU}$ simula a interação entre modelos de geometria diferente. A cena contém 2952 vértices (se considera apenas vértices da superfície) e 9917 tetraedros que são animados a uma taxa média de 30 fps.

O gráfico da figura 7.12 mostra, em *ms*, o tempo gasto em cada sub-processo referente à simulação $S1_TDB_{CPU}$. Note que a detecção de colisão, em ambos estágios, gasta muito mais tempo do que os outros sub-processos. O tempo gasto na computação da profundidade de penetração é quase constante e no casamento de formas é insignificante.

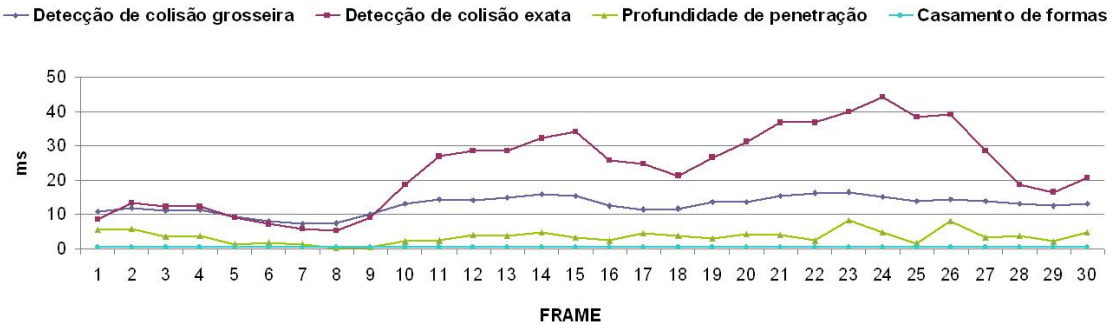


Figura 7.12: Tempo gasto para cada sub-processo em cada passo de tempo.

O gráfico da figura 7.13 mostra de forma mais precisa o comportamento dos algoritmos de detecção de colisão. Note que a detecção de colisão grosseira filtra significativamente o número de primitivas em colisão sendo que, dessas primitivas, apenas o 10% efetivamente estão em colisão em media.

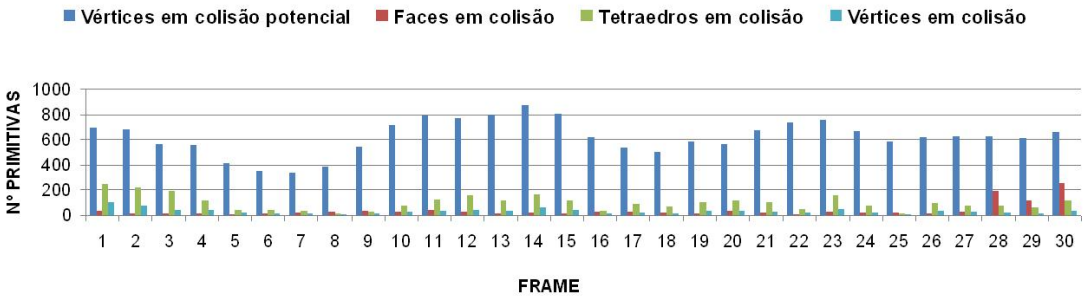
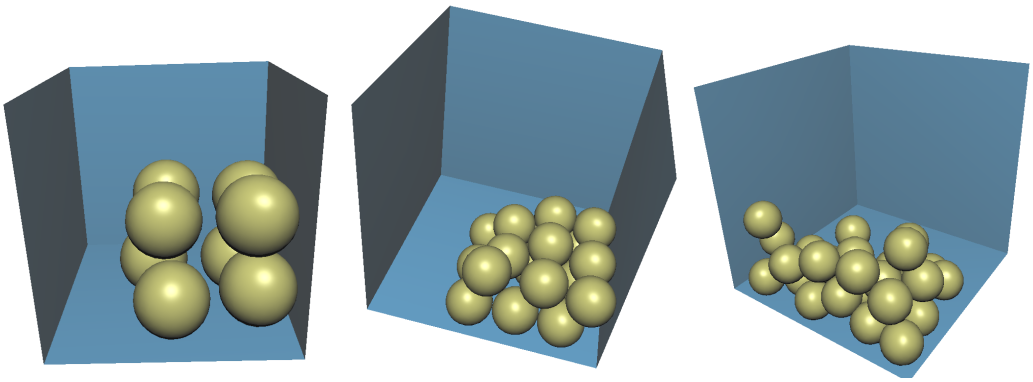


Figura 7.13: Quantidade de primitivas em colisão (vértices em colisão potencial, faces, tetraedros e vértices em colisão real) para cada passo de tempo.

Outros três experimentos (ver $S2_TDB_{CPU}$ na tabela 7.6) foram conducidos, envolvendo um número variado de esferas. A figura 7.14, mostra simulações com cenas de 8, 18 e 27 esferas. O número de primitivas contidas em cada cena assim como a taxa de quadros por segundo (fps) são mostrados na Tabela 7.8.

Tabela 7.8: primitivas para experimentos com esferas.

Número de esferas	vértices	tetraedros	média de fps
8	5832	20480	62
18	19683	46080	41
27	19683	69120	22



(a) (b) (c)

Figura 7.14: Experimentos com 8 (a), 18 (b) e 27 (c) esferas.

Após ter testado o protótipo implementado, os resultados obtidos não podem ser considerados apropriados para simulação em tempo real quando se quer animar uma quantidade grande de objetos. Isto motivou a procura de novas técnicas e idéias, pelo que foram implementados dois protótipos adicionais.

A simulação $S3_DB_{CPU}$, trata um cenário com 8 toros sendo empilhados, e foi testado nos três protótipos para simulação de objetos deformáveis. O gráfico da figura 7.16 mostra uma comparação do tempo gasto no decorrer da simulação para cada protótipo implementado. Observa-se que o protótipo de propagação usando células da borda consegue ser aproximadamente 20% mas rápido do que a versão original, uma vez que o número de células a ser visitado é menor. Entretanto, no protótipo usando funções de distância o tempo foi reduzido em até 10 vezes, já que o teste de colisão é realizado diretamente na função distância do outro objeto, e não precisa de estruturas de dados auxiliares para detectar interferência.

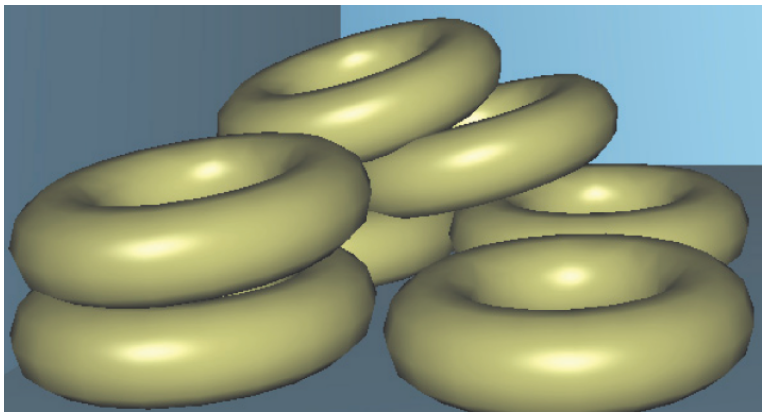


Figura 7.15: Simulação de 8 toros deformáveis com empilhamento.

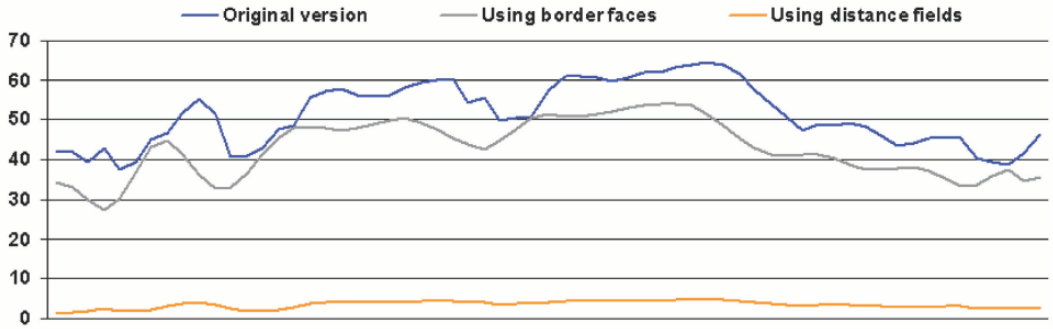


Figura 7.16: Tempo gasto (em segundos) no decorrer da simulação com 8 toros utilizando os protótipos $S1_DB_{CPU}$, $S2_DB_{CPU}$ e $S3_DB_{CPU}$.

7.2 Abordagens usando sistemas de partículas e GPUs

Estas abordagens, diferentemente das abordagens tradicionais, empregam algoritmos paralelos que podem ser executados em GPU, agilizando todo o processo da simulação e, consequentemente, capazes de obter resultados a taxas interativas.

Simulações realizadas

A tabela 7.9 apresenta um conjunto de experimentos realizados para os protótipos da tabela 7.3.

Tabela 7.9: Simulações realizadas para os protótipos usando sistemas de partículas e GPUs

Nome	Protótipo utilizado	Descrição da simulação
S_BP_{GPU}	BP_{GPU}	Detecção de colisão grosseira para volumes limitantes de 50K objetos. Para ver o resultado é utilizado esferas que interagem, sendo que a separação é realizada usando forças de repulsão elástica.
S_RBF_{GPU}	RBF_{GPU}	Simulação de 693 modelos (*Stanford bunny*) contidos numa caixa de lado 2. Cada modelo é representado por 95 partículas, e cada partícula é uma esfera de raio $\frac{1}{64}$. A grade usada é um cubo de lado 2 subdivido em $64 \times 64 \times 64$ voxels.
$S1_RBI_{GPU}$	DBI_{GPU}	Uma chuva com milhares de objetos (*Stanford bunny*) é simulada. A cena é composta de 2904 objetos contidos numa caixa de lado 64. Cada modelo contém 35 partículas, e as partículas são esferas de raio variável $0.159 < r < 1$. A grade usada para a detecção de colisão mapeia o espaço ocupado pela caixa que contém os objetos, subdivida em $64 \times 64 \times 64$ voxels.
$S2_RBI_{GPU}$	DBI_{GPU}	A interação de objetos (*Stanford bunny*) de diferentes tamanhos é simulada. Neste experimento, uma cena com 500 objetos pequenos que caem sobre 6 objetos grandes é simulado. Os modelos, pequeno e grande, são formados por 35 e 208 partículas, respectivamente. A configuração da cena é igual à do exemplo acima (tamanho das partículas, e propriedades da grade).
S_DB_{GPU}	DB_{GPU}	Simulação de 180 objetos (50 *Stanford bunnies* e 130 caixas) deformáveis que interagem numa caixa de lado 64. Os objetos, caixa e *Stanford bunny*, são constituídos de 98 e 208 partículas, respectivamente. Cada partícula é uma esfera de raio $\frac{1}{8}$ e a grade usada para a detecção de colisão é igual à do exemplo $S1_RBI_{GPU}$.

7.2.1 Esquema de detecção de colisões grosseiro

A implementação do esquema descrito na seção 4.1 foi testado com configurações de $3K$ até $50K$ objetos em colisão potencial. A figura 7.17 mostra dois quadros de um cenário com $50K$ objetos interagindo. Neste exemplo foram usadas esferas de diferentes tamanhos que interagem usando o esquema de integração de Euler. O gráfico da figura 7.18 mostra o número de quadros por segundo para este exemplo.

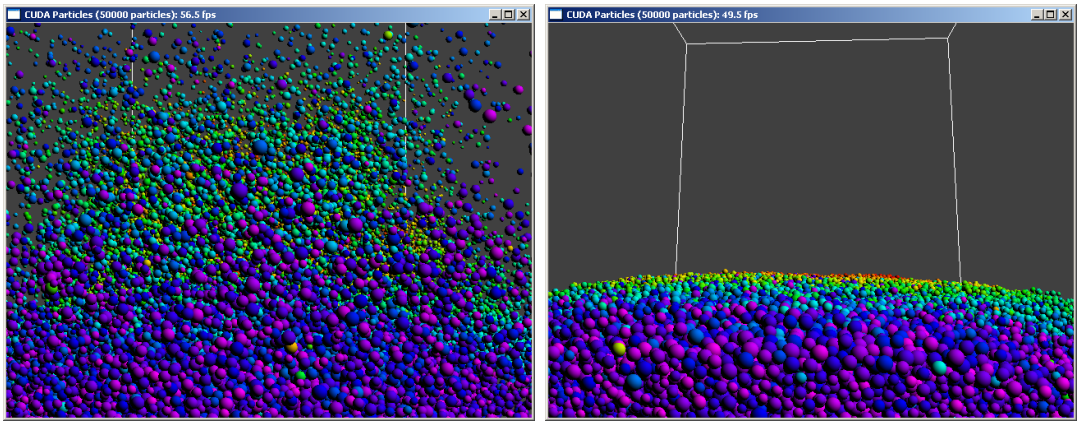


Figura 7.17: Dois quadros do experimento S_BD_{GPU}.

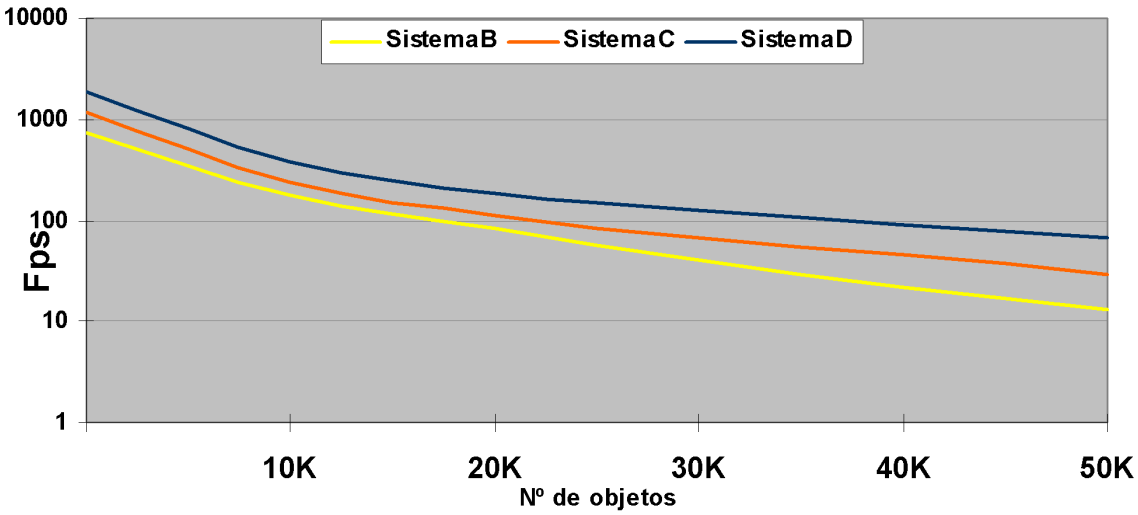


Figura 7.18: Quadros por segundo para o experimento S_BP_{GPU} que é testado no *SistemaB*, *SistemaC* e *SistemaD*.

Neste experimento simples, as colisões são tratadas diretamente, como colisões entre esferas, gerando forças de repulsão. Entretanto, a extensão do esquema para tratar objetos mais complexos não é muito difícil. Uma opção seria usar o esquema de simulação de corpos rígidos apresentado na seção 3.1.

7.2.2 Simulação de objetos rígidos

Nesta parte foram implementados dois protótipos: o primeiro baseado em forças segundo as abordagens de Harada [2] e Baraff [97], e o segundo baseado em impulsos, segundo a abordagem descrita na seção 3.1.

Simulação de objetos rígidos baseada em forças de repulsão

Os resultados obtidos são mostrados na figura 7.19, onde se ilustra um cenário com 693 modelos (*Stanford bunny*) sendo amontoados numa caixa. Para a renderização dos objetos foi utilizada uma malha de 3405 vértices e 6806 triângulos.

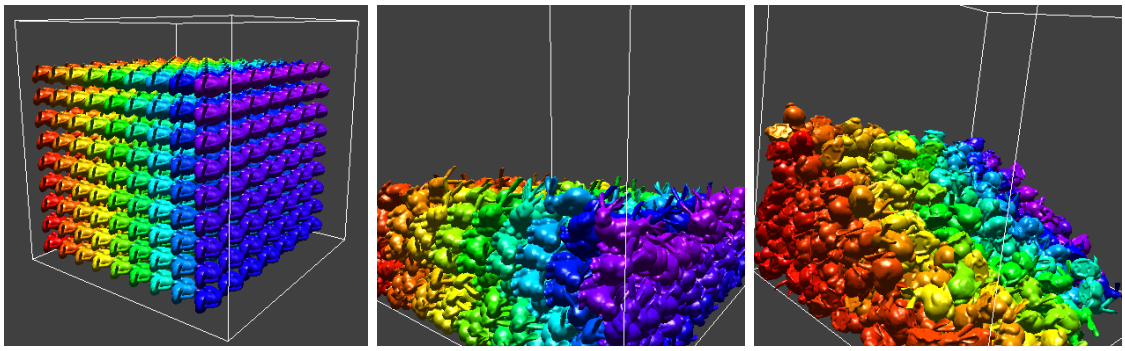


Figura 7.19: Simulação de 693 objetos complexos (*Stanford bunny*).

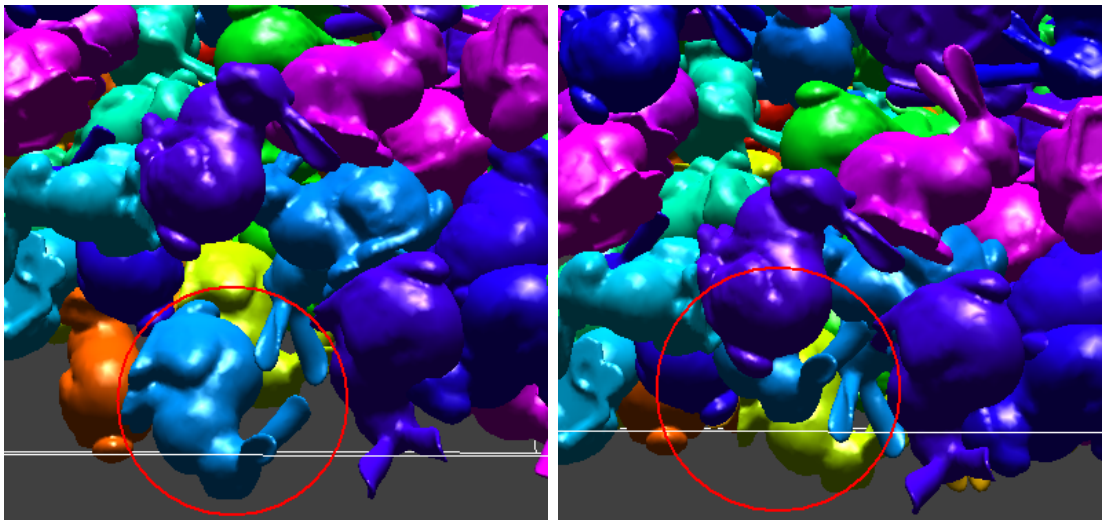


Figura 7.20: Comportamento elástico quando os objetos batem com as paredes do cenário.

O gráfico da figura 7.21 mostra o tempo gasto em cada etapa, para cada passo de tempo, no decorrer da simulação. Estes resultados foram obtidos usando o *SistemaD* (ver tabela 7.1).

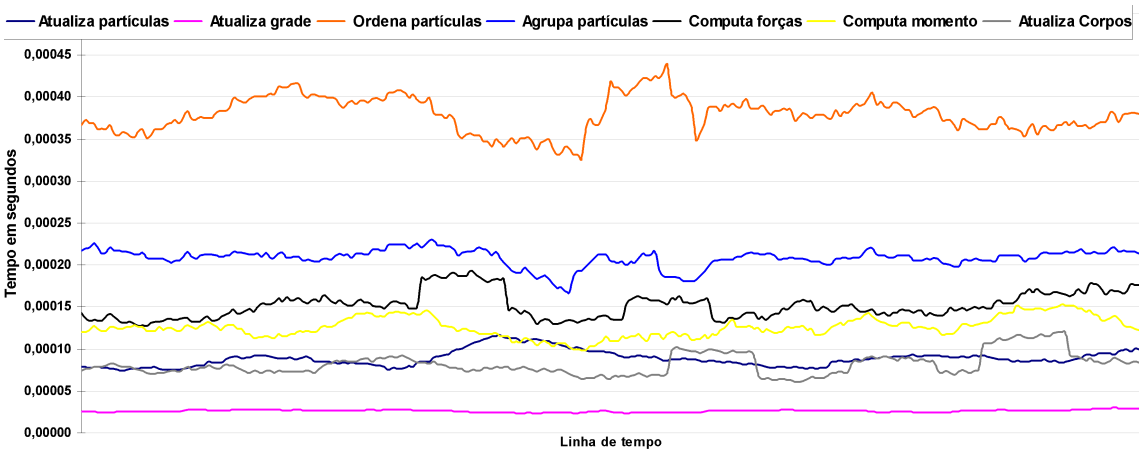


Figura 7.21: Tempo gasto em cada etapa de simulação para o experimento S_RBF_{GPU}.

No gráfico da figura 7.22 são comparados o tempo de integração vs. o tempo de renderização, para cada passo de tempo, no decorrer da simulação. Observa-se que o tempo de renderização é bastante alto comparado com o tempo de processamento nos cálculos da simulação. Isto se deve a que é necessário transferir, da memória da GPU, os dados que representam os estados dos objetos (*arrays* de posições e orientações), para a memória da CPU, e poder aplicar a transformação correspondente de forma a visualizar cada objeto na posição e orientação correta.

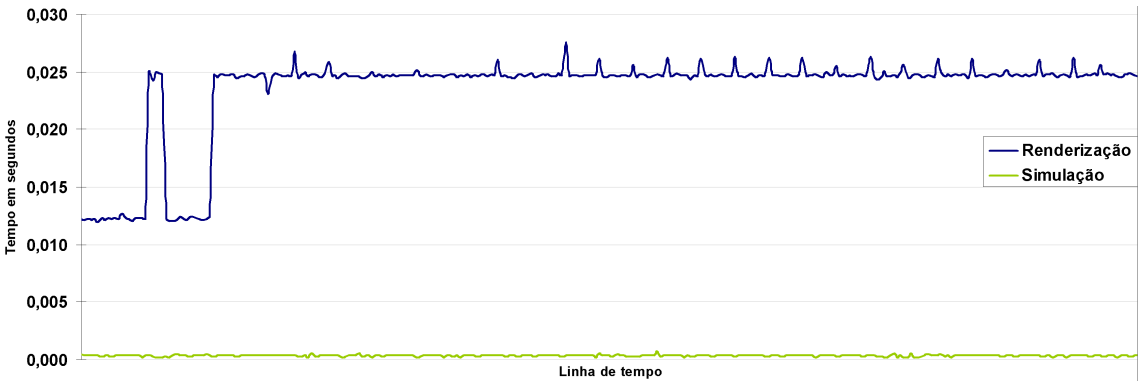


Figura 7.22: Tempo de simulação vs. tempo de renderização.

O gráfico da figura 7.23, entretanto, mostra o desempenho em fps para três GPUs diferentes. Nota-se que o comportamento é bastante similar, apenas com a diferença de rapidez devido ao poder computacional das três GPUs empregadas.

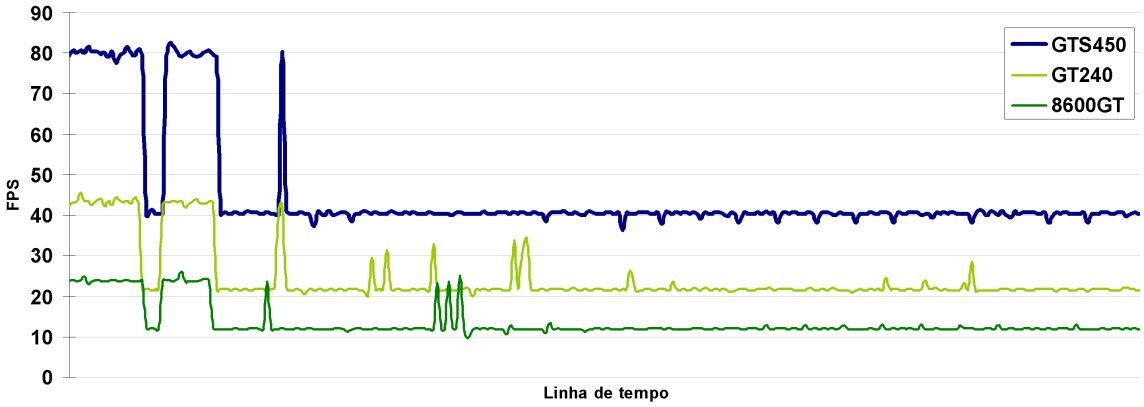


Figura 7.23: Fps para obtidos em três GPUs diferentes.

Esta implementação, baseada em forças de repulsão, gerou resultados de comportamento físico plausível. Observou-se, no entanto, que nesta modelagem, a reação a colisões sobre objetos complexos, gera um comportamento elástico quando os objetos colidem com obstáculos ou paredes (ver figura 7.20). Este problema é inerente ao método, o que motivou a implementação de uma versão baseada em impulsos.

Outra desvantagem desta abordagem surge quando se quer simular objetos grandes, já que é necessário o emprego de um número elevado de partículas diminuindo consideravelmente o desempenho e a quantidade de objetos que podem ser simulados em tempo real. A figura 7.24 mostra um quadro da simulação de uma cena composta por 6 *Stanford bunnies*, onde cada objeto é composto por 4936 partículas de igual diâmetro.

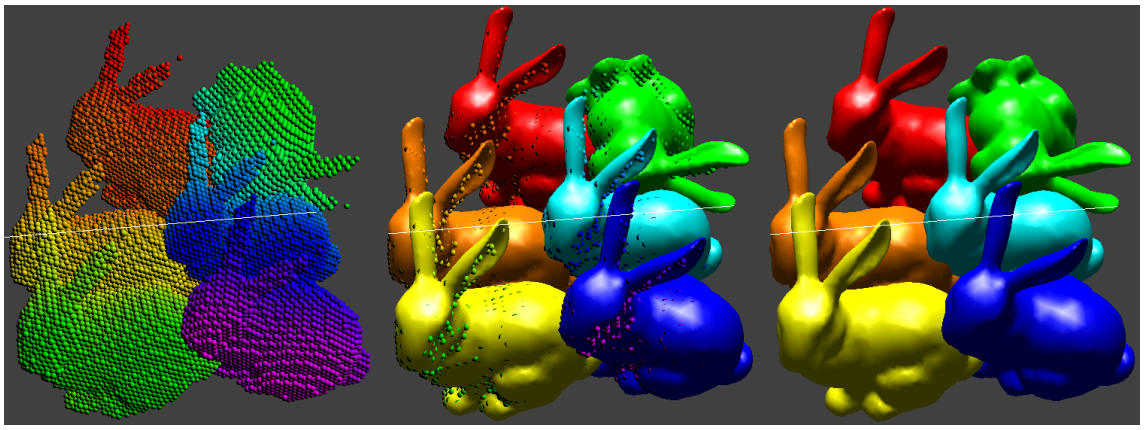


Figura 7.24: Simulação de objetos grandes: visualização com partículas (esquerda), visualização com partículas e malhas (centro) e visualização apenas com malhas (direita).

Neste experimento se observa que a qualidade da simulação é comprometida uma vez que as partículas têm um tamanho muito inferior ao tamanho dos objetos, sendo assim comum observar-se interpenetrações onde uma partícula de um dado

objeto colide com partículas do outro objeto mas desde o interior, o que dificulta a determinação do eixo de repulsão entre as partículas (ver figura 7.25).

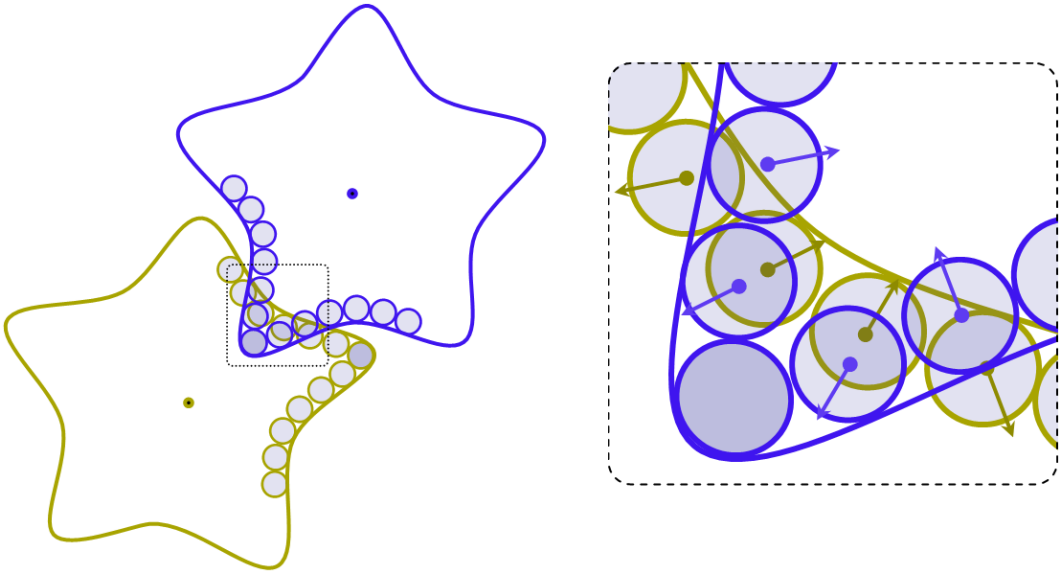


Figura 7.25: Partículas interpenetradas gerando forças opostas.

Simulação de objetos rígidos usando impulsos

A segunda versão implementada segue o método apresentado por Guendelman et al. [18], descrito na seção 5.2.6. Nesta implementação, a técnica de Harada [2] foi estendida para usar partículas de tamanhos diferentes, o que permite aproximar melhor a forma dos objetos. Nos experimentos $S1_RBI_{GPU}$ e $S2_RBI_{GPU}$, foram usados dois modelos: o primeiro (*small bunny*) representado por 35 partículas e o segundo (*big bunny*) representado por 208 partículas. A figura 7.26 ilustra a representação por partículas para os modelos *small bunny* e *big bunny*.

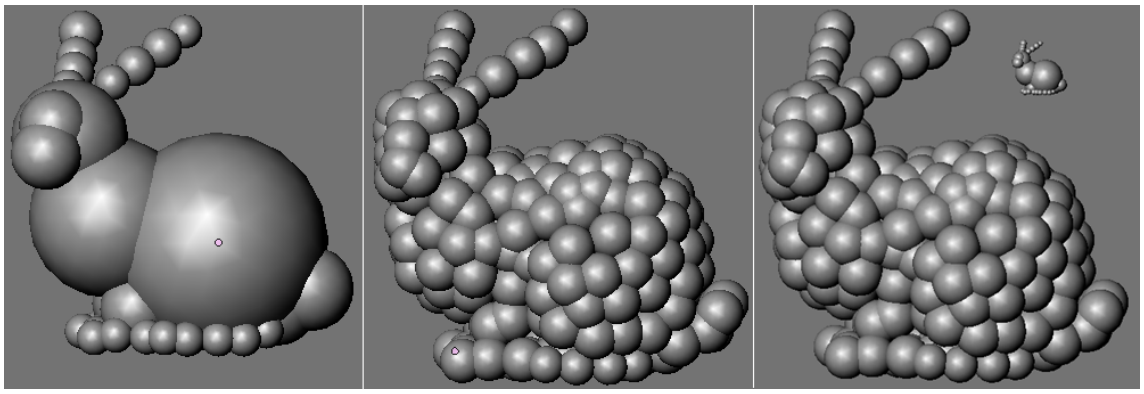


Figura 7.26: Representação, com partículas de tamanho variable, para dois modelos *Stanford bunny*: pequeno (esquerda), grande (centro) e relação de tamanhos (direita).

Estes experimentos foram conduzidos no *SistemaD*, onde o primeiro que simula

um cenário de chuva de objetos, visa mostrar a eficiência do método proposto, conseguindo ser executado em aproximadamente $22fps$. Já o segundo experimento, com objetos de tamanhos diferentes, visa mostrar a qualidade do método e é executado em aproximadamente $80fps$. Note que, no protótipo S_RBF_{GPU}, este experimento não tem um bom desempenho, uma vez que o objeto grande é constituído por 50 vezes a quantidade de partículas do objeto pequeno, além de permitir interpenetrações devido à relação de tamanho entre partículas e objetos grandes.

O gráfico da figura 7.28 mostra o tempo gasto em cada etapa, para cada passo de tempo, no decorrer da simulação para o experimento $S1_RBI_{GPU}$. Se observa que o processo de ordenação de partículas por índices de célula, fundamental para o processo de detecção de colisão, gasta quase 30% do tempo total num passo de simulação.

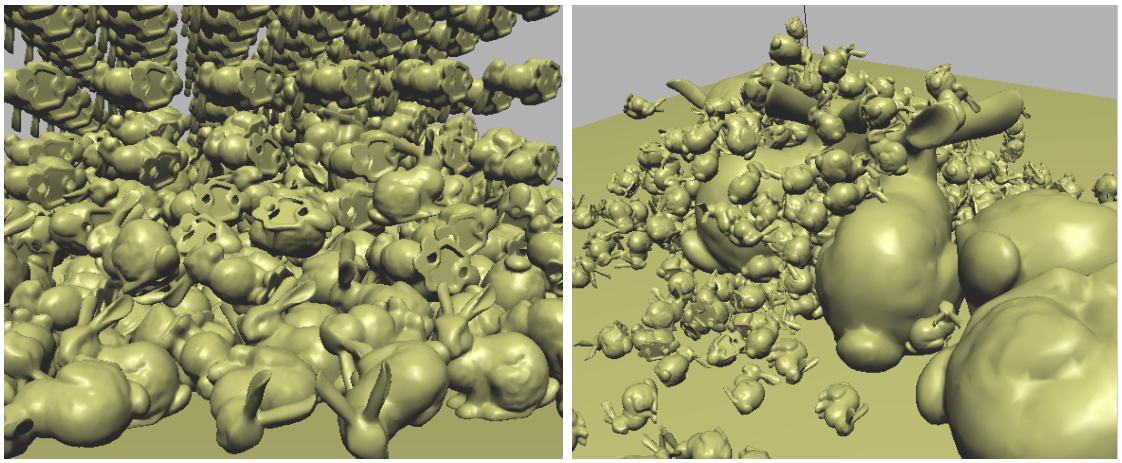


Figura 7.27: Simulação de coelhos: chuva de 2904 coelhos pequenos(esquerda), 500 coelhos pequenos caem sobre 6 coelhos grandes(direita).

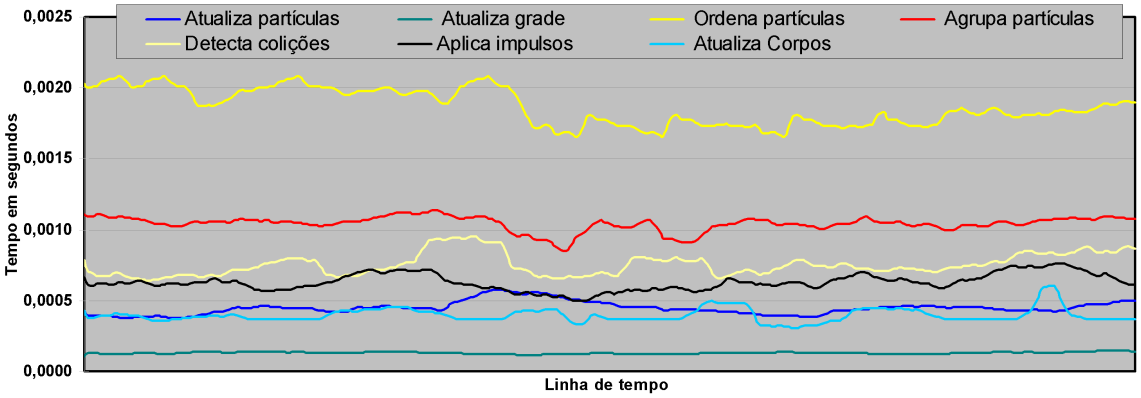


Figura 7.28: Tempo gasto em cada etapa de simulação para o experimento $S1_RBI_{GPU}$.

A figura 7.29 mostra dois instantes de tempo da simulação $S1_RBI_{GPU}$. Pode-se notar que o comportamento elástico obtido com o protótipo RBF_{GPU} não mais

se verifica, o que demonstra que o protótipo baseado em impulsos é mais apropriado para simulação de objetos rígidos.

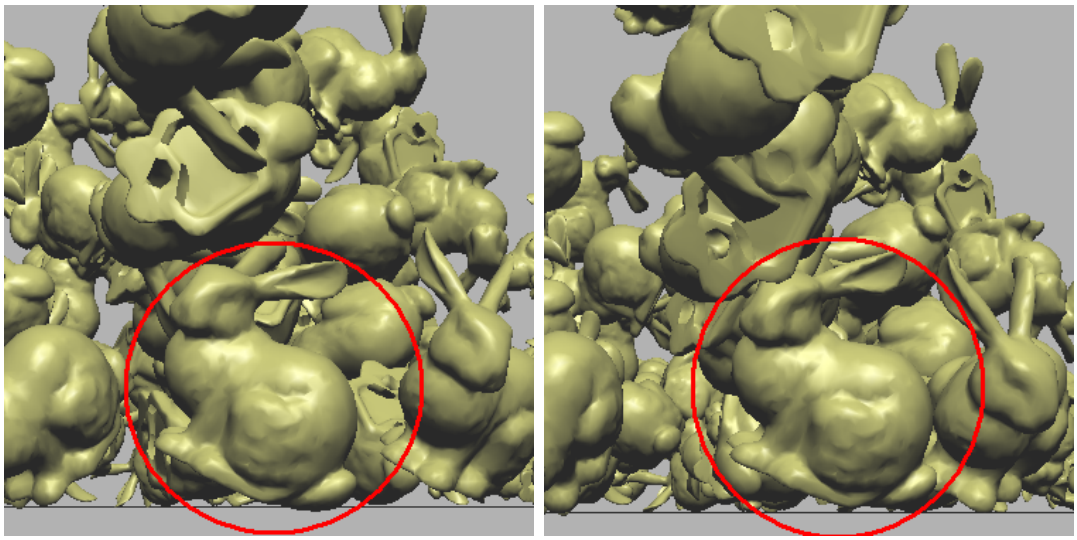


Figura 7.29: Dois quadros de simulação. Interpenetração entre objetos e obstáculos(paredes) não mais se verifica.

7.2.3 Simulação de objetos deformáveis usando casamento de formas

O protótipo implementado segue a abordagem descrita na seção 3.2. Entretanto, esta implementação paralela usando partículas leva a um ganho considerável na velocidade de execução, conseguindo simular centenas de objetos deformáveis em tempo real. No experimento S_DB_{GPU} (um quadro é ilustrado na figura 7.30) é simulada a interação de caixas e modelos *Stanford bunny* sendo amontoados numa caixa. A simulação, que é constituída de 130 caixas e 50 *Stanford bunny*, roda aproximadamente a $98fps$ no $SistemaD$. Os resultados de desempenho obtidos para este experimento, são relativamente baixos comparados com os obtidos na simulação de objetos rígidos. Isto se deve a que neste esquema é necessário computar deformações para os objetos a cada passo de tempo, além do cálculo de normais, realizado num shader de geometria, que é necessário para renderizar a superfície dos objetos. Para a renderização dos objetos foram utilizadas malhas de triângulos, onde uma caixa é representada por 386 vértices e 768 triângulos, e um *Stanford bunny* é representado por 2503 vértices e 4968 triângulos

O gráfico da figura 7.31 mostra tempos gastos em cada etapa da simulação, a cada quadro. Já o gráfico da figura 7.32 faz uma comparação entre o tempo gasto na simulação de um passo de tempo vs. o tempo gasto na renderização para cada quadro.

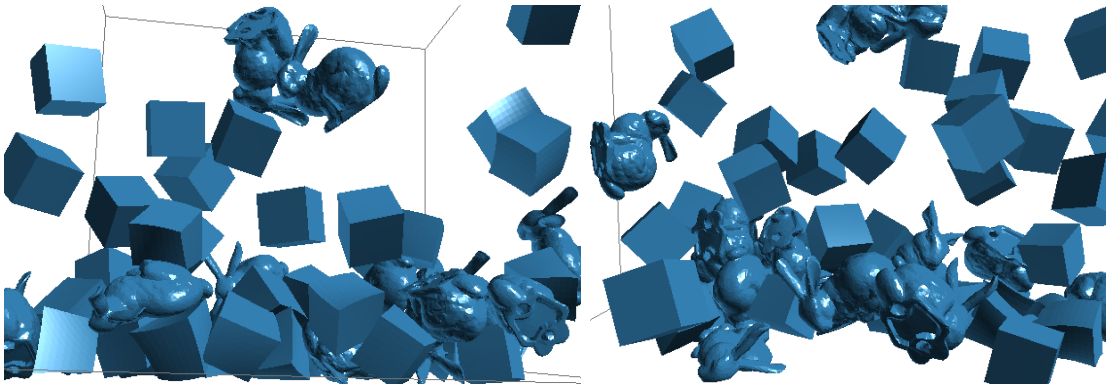


Figura 7.30: Simulação de 180 objetos deformáveis.

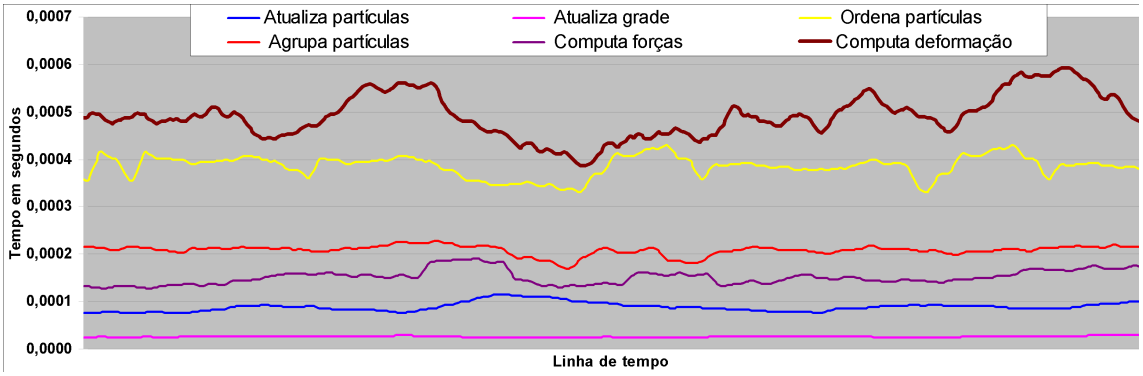


Figura 7.31: Tempo gasto em cada etapa de simulação para o experimento S_DB_{GPU}.

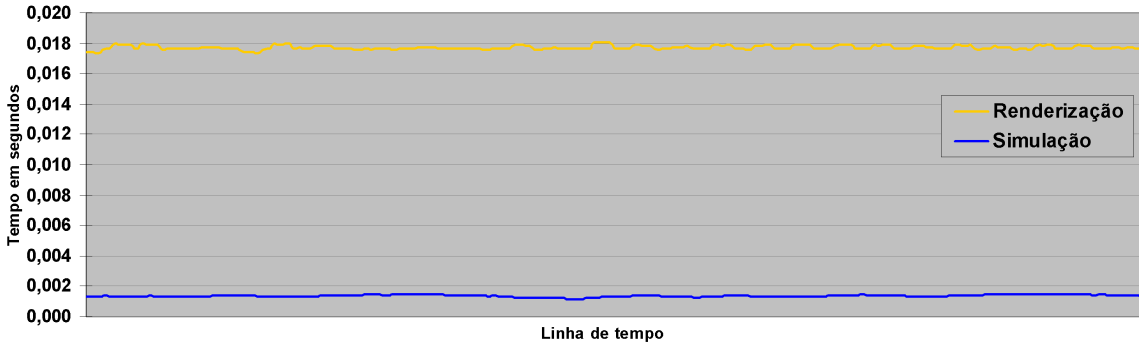


Figura 7.32: Tempo de simulação vs. tempo de renderização.

7.2.4 Simulação de líquidos usando Hidrodinâmica suavizada com partículas

O protótipo SPH_{GPU} foi implementado para simular fluidos utilizando o método SPH. A figura 7.33 ilustra a simulação de um tanque com líquido usando o protótipo. No experimento, o líquido consta de $100K$ partículas e roda aproximadamente a $60fps$ no $SistemaD$.

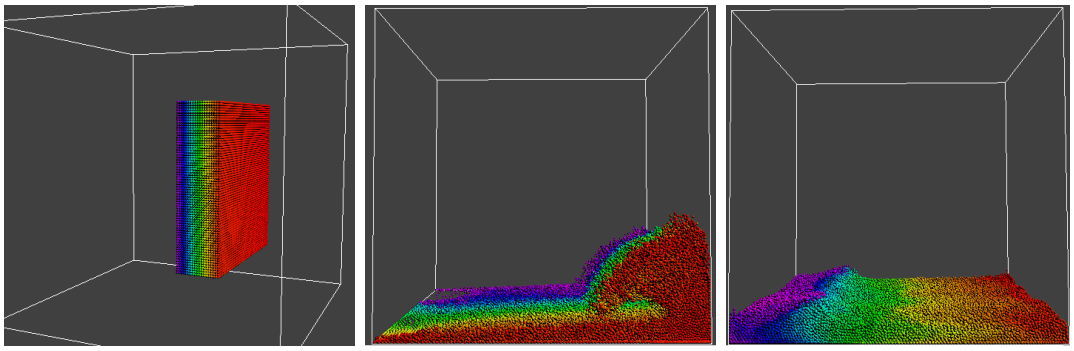


Figura 7.33: Simulação de líquido SPH com 100000 partículas.

Também foi implementada uma técnica para renderizar a superfície da água, descrita no capítulo 6. A figura 7.34 ilustra o resultado obtido aplicando uma função *fresnel* com sombra, reflexão e refração gerando o efeito de cáustica. Neste experimento, o líquido transparente (água) é representado por $10K$ partículas, rodando a $100fps$ aproximadamente no $SistemaD$.

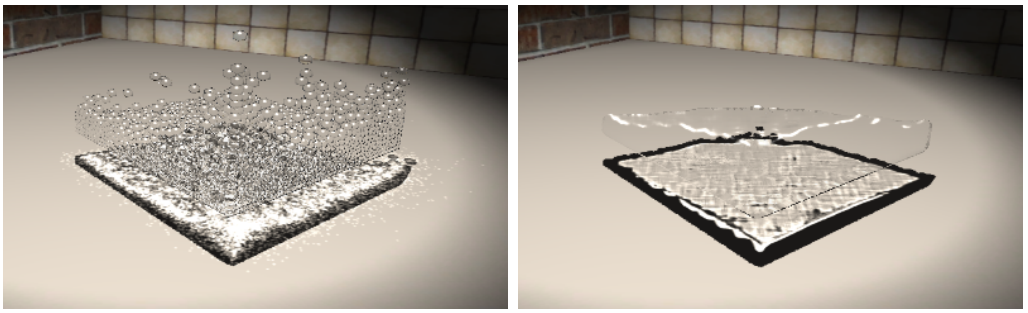


Figura 7.34: Efeito de cáustica gerado por um tanque de agua.

7.2.5 Simulação de objetos elásticos

Implementou-se um protótipo para simular tecidos baseado no esquema apresentado por Jakobsen [21]. No experimento da figura 7.35 se deixa cair um tecido sobre outro pendurado. O exemplo mostra a detecção e resposta a colisões e auto-colisões entre os retalhos de tecido. Cada retalho de tecido é representado por 289 vértices e 512 triângulos.

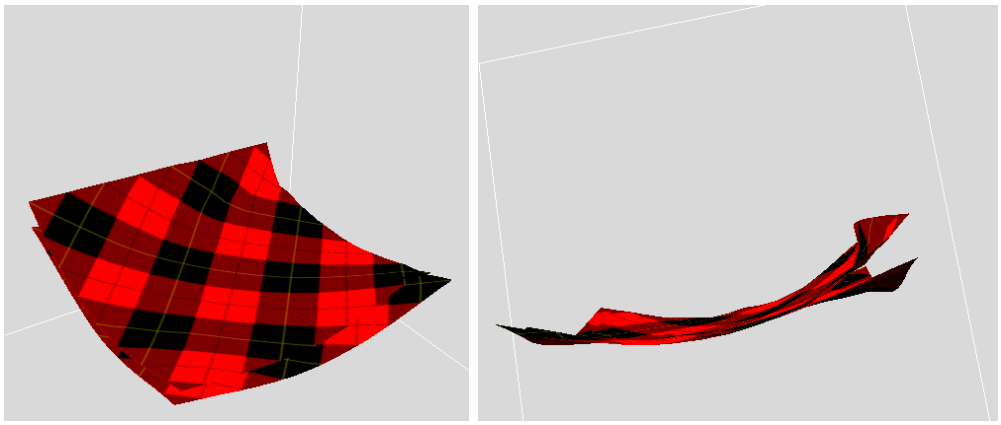


Figura 7.35: Simulação de tecidos colidindo.

7.2.6 Acoplamento entre objetos diferentes

A abordagem de sistemas de partículas e a grade espacial para detectar colisões no método apresentado no capítulo 5, é adequado para simular a interação entre objetos de tipos diferentes já que as forças geradas quando há colisão entre partículas são aplicadas respectivamente aos objetos aos quais pertencem. Entretanto, no caso de simulação de tecidos a detecção de colisão é ligeiramente diferente, mas segue o mesmo princípio. Assim, é possível combinar sistemas de partículas interagindo com retalhos de tecido. A figura 7.36 mostra a eficiência deste método, simulando bolhas de agua caindo sobre um retalho de tecido pendurado. O experimento roda a aproximadamente $55fps$ com $20K$ partículas.

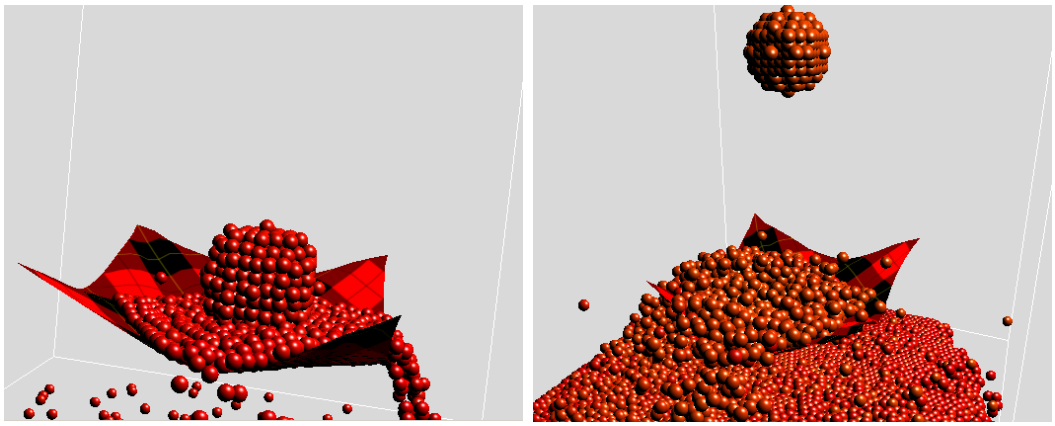


Figura 7.36: Bolas de líquido caindo num retalho de tecido pendurado pelas pontas.

A figura 7.37 mostra a interação de objetos rígidos com líquido SPH. Os experimentos rodam em aproximadamente 20 fps.

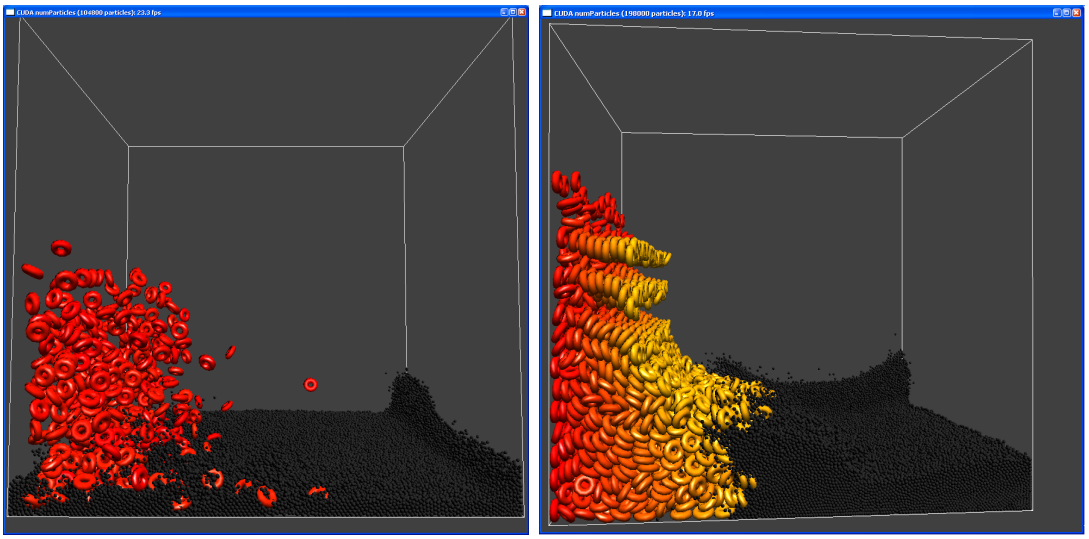


Figura 7.37: Interação de objetos sólidos e líquido.

7.3 Publicações e contribuições

As principais contribuições deste trabalho são:

- A escolha e utilização de diversas técnicas mas adequadas para simulação física em tempo real.
- A implementação de protótipos completos para simulação de objetos sólidos e líquidos.
- A interação de objetos de tipos diferentes pelo uso de partículas na representação de objetos e o uso da mesma grade para o processo de detecção de colisão. A exceção de tecidos, onde é necessário o uso de uma grade maior para mapear triângulos de tecido.

Algumas destas abordagens foram publicados em congressos e revistas:

- O esquema para simulação de objetos deformáveis descrito na seção 3.2 foi publicado no Simpósio Brasileiro de Jogos e Entretenimento Digital 2008 [89].
- Observando as limitações em desempenho do esquema anterior, duas extensões foram propostas (veja a seção 3.2.10) que geraram uma publicação no journal "Computers & Entertainment" [90].
- Um esquema baseado em partículas "Physically Based Simulation Using Particle Systems" foi publicado como poster no Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens 2008.
- Uma aplicação do esquema SPH implementado, foi publicado no Simpósio Brasileiro de Computação Gráfica e Processamento de imagens 2010 [105].

Capítulo 8

Conclusões

Nesta tese apresentamos abordagens de simulação baseada em física. Primeiramente abordagens que empregam algoritmos sequenciaiss foram implementadas e testadas. Os resultados obtidos demonstraram que há limitações em desempenho quando se quer simular cenas com grande quantidade de objetos. Entretanto, fazendo uso do potencial que possuem as GPUs modernas, foi possível implementar protótipos para simulação de objetos sólidos e líquidos capazes de lidar com grande quantidade de objetos a taxas interativas. Esses resultados se devem principalmente ao uso de partículas para representar objetos, à simplicidade deste tipo de estrutura e ao processamento paralelo realizado a cada etapa da simulação. Foi possível simular desde centenas até milhares de objetos em tempo real, e por outro lado, o esquema baseado em partículas permite a interação de objetos de diversos tipos num mesmo cenário de modo natural.

Foram realizados experimentos para testar o desempenho em cada protótipo implementado, permitindo concluir que o método baseado em sistemas de partículas possui diversas vantagens sobre os métodos sequenciais, principalmente por aproveitar em boa parte o poder computacional das GPUs modernas e obter um ganho considerável no tempo de execução a cada etapa da simulação. O método baseado em sistemas de partículas é eficiente e flexível, podendo ser utilizado para simular diversos tipos de objetos tais como: objetos rígidos, objetos deformáveis, líquidos, e a interação entre mesmos. O sucesso desse método se deve principalmente ao uso de uma grade 3D uniforme que mapeia o espaço onde os objetos interagem. Esta idéia também é utilizada para simular objetos elásticos como tecidos. Já neste caso, triângulos sao guardados em células de uma grade secundária. Os resultados obtidos demonstram que o método é adequado para simular o comportamento físico de objetos e, principalmente, permite simular cenários com grandes quantidades de objetos em movimento.

Com relação ao métodos conhecidos, apresenta-se contribuições no uso de partículas de diferentes tamanhos para representar objetos rígidos de geometria complexa

e a utilização de impulsos ao invés de forças de repulsão. Estas variações permitiram obter resultados mais apropriados para simulação de corpos rígidos, e a interação entre objetos de diferentes tamanhos tornou-se possível de modo mais natural.

Por outro lado, a técnica para detecção de colisões numa etapa grosseira (*broad phase*) apresentada também mostrou que pode ser empregada para alcançar ganhos consideráveis quando se deseja simular grandes quantidades de objetos.

8.1 Trabalhos futuros

Como trabalhos futuros podem ser considerados os seguintes assuntos:

- Melhorar a eficiência no processo de detecção de colisão a nível macro eliminando a limitação do tamanho dos objetos [40].
- Emprego de partículas orientadas [106] e assim representar sólidos com menor quantidade de partículas.
- Cálculos físicos mais apropriados para tratamento de colisões entre objetos deformáveis.
- Melhorar os resultados no método SPH (evitar compressibilidade).
- Suporte para objetos articulados (tratar restrições).
- Interface para que o usuário possa interagir com objetos.
- Multi-GPU para simular maiores quantidades de objetos.

Referências Bibliográficas

[1] MÜLLER, M., HEIDELBERGER, B., TESCHNER, M., et al. "Meshless deformations based on shape matching". In: *Proceedings of SIGGRAPH'05*, pp. 471–478, New York, NY, USA, 2005. doi: http://graphics.ethz.ch/~brunoh/research.html.

[2] HARADA, T. "GPU Gems 3". cap. Rigid Bodies on GPU, pp. 611–632, Addison Wesley, 2007. Disponível em: <`http://www.iii.u-tokyo.ac.jp/~takahiroharada/`>.

[3] TAKAHIRO HARADA, SEIICHI KOSHIZUKA, Y. K. "Real-time Fluid Simulation Coupled with Cloth". 2007.

[4] BACIU, G., WONG, W. S. "Image-Based Techniques in a Hybrid Collision Detector", *IEEE Transactions on Visualization and Computer Graphics*, v. 09, n. 2, pp. 254–271, 2003. ISSN: 1077-2626. doi: http://doi.ieeecomputersociety.org/10.1109/TVCG.2003.10012.

[5] KNOTT, D., PAI, D. "CinDeR: Collision and interference detection in real-time using graphics hardware". 2003. Disponível em: <`citeseer.ist.psu.edu/knott03cinder.html`>.

[6] FAN, Z., WAN, H., GAO, S. "Simple and rapid collision detection using multiple viewing volumes". In: *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry*, pp. 95–99, New York, NY, USA, 2004. ACM. ISBN: 1-58113-884-9. doi: http://doi.acm.org/10.1145/1044588.1044605.

[7] TESCHNER, M., KIMMERLE, S., HEIDELBERGER, B., et al. "Collision Detection for Deformable Objects". In: *Computer Graphics Forum*, pp. 61–81. Eurographics Association, Eurographics Association and Blackwell Publishing, 2005. Disponível em: <`http://citeseer.ist.psu.edu/teschner04collision.html`>.

[8] JANG, H.-Y., JEONG, T., HAN, J. "GPU-based Image-space Approach to Collision Detection among Closed Objects". In: *PG 2006: Proceedings of*

the 2006 Pacific Conference on Computer Graphics and Applications, pp. 242–251, 2006.

[9] JANG, H., HAN, J. "Image-Space Collision Detection Through Alternate Surface Peeling". In: *LNCS 4841: Proceedings of the 2008 International Symposium on Visual Computing*, pp. 66–75, 233 Spring Street, New York, USA, 2007. Springer.

[10] ALLARD, J., FAURE, F., COURTECUISSE, H., et al. "Volume Contact Constraints at Arbitrary Resolution", *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)*, v. 29, n. 3, aug 2010. Disponível em: <`http://codrt.fr/allardj/pub/2010/AFCFDK10`>. http://www.sofa-framework.org/projects/ldi.

[11] COHEN, J. D., LIN, M. C., MANOCHA, D., et al. "I-COLLIDE: an interactive and exact collision detection system for large-scale environments". In: *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pp. 189–ff., New York, NY, USA, 1995. ACM. ISBN: 0-89791-736-7. doi: http://doi.acm.org/10.1145/199404.199437.

[12] GOTTSCHALK, S., LIN, M. C., MANOCHA, D. "OBB-Tree: a hierarchical structure for rapid interference detection". In: *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 171–180, New York, NY, USA, 1996. ACM. ISBN: 0-89791-746-4. doi: http://doi.acm.org/10.1145/237170.237244.

[13] VAN DEN BERGEN, G. "Efficient collision detection of complex deformable models using AABB trees", *J. Graph. Tools*, v. 2, n. 4, pp. 1–13, 1997. ISSN: 1086-7651.

[14] BRADSHAW, G., O'SULLIVAN, C. "Adaptive medial-axis approximation for sphere-tree construction", *ACM Trans. Graph.*, v. 23, n. 1, pp. 1–26, 2004. ISSN: 0730-0301. doi: http://doi.acm.org/10.1145/966131.966132.

[15] JAMES, D. L., PAI, D. K. "BD-Tree: Output-Sensitive Collision Detection for Reduced Deformable Models", *ACM Transactions on Graphics (SIGGRAPH 2004)*, v. 23, n. 3, ago. 2004.

[16] BELL, N., YU, Y., MUCHA, P. J. "Particle-based simulation of granular materials". In: *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 77–86, New York, NY, USA, 2005. ACM. ISBN: 1-7695-2270-X. doi: http://doi.acm.org/10.1145/1073368.1073379.

[17] HARADA, T., KOSHIZUKA, S., KAWAGUCHI, Y. "Smoothed Particle Hydrodynamics on GPUs". pp. 63–70, 2007.

[18] GUENDELMAN, E., BRIDSON, R., FEDKIW, R. "Nonconvex rigid bodies with stacking". In: *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pp. 871–878, New York, NY, USA, 2003. ACM. ISBN: 1-58113-709-5. doi: http://doi.acm.org/10.1145/1201775.882358.

[19] TESCHNER, M., HEIDELBERGER, B., MÜLLER, M., et al. "Optimized Spatial Hashing for Collision Detection of Deformable Objects". In: *Proceedings of Vision, Modeling, Visualization VMV'03Proceedings of SPM 2005*, pp. 47–54, 2003. doi: http://graphics.ethz.ch/~brunoh/publications.html.

[20] HEIDELBERGER, B., TESCHNER, M., KEISER, R., et al. "Consistent Penetration Depth Estimation for Deformable Collision Response". In: *Proceedings of Vision, Modeling, Visualization VMV'04*, pp. 157–164, Stanford, USA, 2004. ISBN: 339–346. doi: http://graphics.ethz.ch/~brunoh/publications.html.

[21] JAKOBSEN, T. "Advanced Character Physics". In: *Proceedings, Game Developer's Conference 2001*, SJ, USA, 2001. GDC Press. doi: http://www.teknikus.dk/tj/gdc2001.htm.

[22] GRAND, S. L. "GPU Gems 3". cap. Broad-Phase Collision Detection with CUDA, pp. 697–721, Addison Wesley, 2007.

[23] HOSSEINI, S. M., MANZARI, M. T., HANNANI, S. K. "A fully explicit three-step SPH algorithm for simulation of non-Newtonian fluid flow", *International Journal of Numerical Methods for Heat & Fluid Flow*, v. 17, n. 7, pp. 715–735, 2007. ISSN: 0961-5539. doi: 10.1108/09615530710777976. Disponível em: <http://dx.doi.org/10.1108/09615530710777976>.

[24] HARADA, T., KOSHIZUKA, S., KAWAGUCHI, Y. "Real-time Coupling of Fluids and Rigid Bodies". 2007.

[25] LORENSEN, W. E., CLINE, H. E. "Marching cubes: A high resolution 3D surface construction algorithm", *SIGGRAPH Comput. Graph.*, v. 21, n. 4, pp. 163–169, 1987. ISSN: 0097-8930. doi: http://doi.acm.org/10.1145/37402.37422.

[26] VAN DER LAAN, W. J., GREEN, S., SAINZ, M. "Screen space fluid rendering with curvature flow". In: *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 91–98, New York, NY, USA, 2009. ACM. ISBN: 978-1-60558-429-4. doi: http://doi.acm.org/10.1145/1507149.1507164.

[27] FLYNN, M. "Some Computer Organizations and Their Effectiveness", *IEEE Trans. Comput.*, v. C-21, pp. 948+, 1972. Disponível em: <http://en.wikipedia.org/wiki/Flynn's_taxonomy>.

[28] NVIDIA™. "GPGPU GGeneral PPurpose Computation on GPU". 2004. http://www.gpgpu.org.

[29] SLUSALLEK, P., PFLAUM, T., SEIDEL, H.-P. "Implementing RenderMan - Practice, Problems, and Enhancements", *Computer Graphics Forum (Proc. Eurographics '94)*, v. 3, pp. 443–454, 1994. Disponível em: <citeseer.ist.psu.edu/slusallek94implementing.html>.

[30] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., et al. "A Survey of General-Purpose Computation on Graphics Hardware", *Computer Graphics Forum*, v. 26, n. 1, pp. 80–113, 2007. Disponível em: <http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x>.

[31] OWENS, J. D. "GPU Computing". In: *Proceedings of the IEEE*, v. 96-5, pp. 879–899, 2008.

[32] NVIDIA. "Fermi: NVIDIA's Next Generation CUDA Compute Architecture". 2009. Disponível em: <$http://www.nvidia.com/content/PDF/fermi_white_papers\NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf$>.

[33] NVIDIA™. "CUDA Environment – Compute Unified Device Architecture". 2007. http://www.nvidia.com/object/cuda_home.html.

[34] OPENGL, SHREINER, D., WOO, M., et al. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. USA, Addison-Wesley Professional, August 2005. ISBN: 0321335732. Disponível em: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321335732>.

[35] VERLET, A. "Computer Experiments on Classical Fluids: I. Thermodymamic properties of Leonard-Jones Molecules", *Phys. Review*, v. 159, pp. 98–103, 1967.

[36] EBERLY, D. *Game Physics*. New York, NY, USA, Elsevier Science Inc., 2003. ISBN: 1558607404.

[37] GRESS, A., ZACHMANN, G. *Object-Space Interference Detection on Programmable Graphics Hardware*. Relatório Técnico CG-2004-1, University Bonn, Informatikk II, Bonn, Germany, may 2004. Disponível em: <http://www.gabrielzachmann.org/>.

[38] HUDSON, T. C., LIN, M. C., COHEN, J., et al. "V-COLLIDE: Accelerated Collision Detection for VRML". In: Carey, R., Strauss, P. (Eds.), *VRML 97: Second Symposium on the Virtual Reality Modeling Language*, New York City, NY, 1997. ACM Press. Disponível em: <citeseer.ist.psu.edu/hudson97vcollide.html>.

[39] EHMANN, S. A., LIN, M. C. "Accelerated proximity queries between convex polyhedra by multi-level Voronoi marching". In: *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2101–2106, 2000.

[40] LIU, F., HARADA, T., LEE, Y., et al. "Real-time collision culling of a million bodies on graphics processing units". In: *ACM SIGGRAPH Asia 2010 papers*, SIGGRAPH ASIA '10, pp. 154:1–154:8, New York, NY, USA, 2010. ACM. ISBN: 978-1-4503-0439-9. doi: http://doi.acm.org/10.1145/1866158.1866180. Disponível em: <http://doi.acm.org/10.1145/1866158.1866180>.

[41] GANOVELLI, F., DINGLIANA, J., O'SULLIVAN, C. "BucketTree: Improving Collision Detection Between Deformable Objects". In: *Spring Conference in Computer Graphics (SCCG)*, pp. 156–163, 2000. Disponível em: <citeseer.ist.psu.edu/ganovelli00buckettree.html>.

[42] SAMET, H. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2006. ISBN: 0123694469.

[43] CHOI, Y.-J., KIM, Y. J., KIM, M.-H. "self-CD: Interactive Self-collision Detection for Deformable Body Simulation Using GPUs". In: *AsiaSim*, pp. 187–196, 2004.

[44] SHINYA, M., FORGUE, M.-C. "Interference detection through rasterization". In: *The Journal of Visualization and Computer Animation*, v. 2, pp. 132–134, 1991.

[45] HEIDELBERGER, B., TESCHNER, M., GROSS, M. H. "Detection of Collisions and Self-collisions Using Image-space Techniques". In: *WSCG*, pp. 145–152, 2004.

[46] FAURE, F., BARBIER, S., ALLARD, J., et al. "Image-based collision detection and response between arbitrary volume objects". In: *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, pp. 155–162, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. ISBN: 978-3-905674-10-1. Disponível em: <http://portal.acm.org/citation.cfm?id=1632592.1632615>.

[47] GOVINDARAJU, N. K., REDON, S., LIN, M. C., et al. "CULLIDE: interactive collision detection between complex models in large environments using graphics hardware". In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, p. 204, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1198555.1198785.

[48] MANOCHA, D. "Quick-CULLIDE: Fast Inter- and Intra-Object Collision Culling Using Graphics Hardware". In: *VR '05: Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality*, pp. 59–66, 319, Washington, DC, USA, 2005. IEEE Computer Society. ISBN: 0-7803-8929-8. doi: http://dx.doi.org/10.1109/VR.2005.62.

[49] GOVINDARAJU, N. K., KNOTT, D., JAIN, N., et al. "Interactive collision detection between deformable models using chromatic decomposition", *ACM Trans. Graph.*, v. 24, n. 3, pp. 991–999, 2005. ISSN: 0730-0301. doi: http://doi.acm.org/10.1145/1073204.1073301.

[50] TESCHNER, M., HEIDELBERGER, B., MANOCHA, D., et al. "Collision Handling in Dynamic Simulation Environments". In: *Eurographics Tutorial # 2*, pp. 1–4, Dublin, Ireland, 29 August 2005. Eurographics Association.

[51] LUQUE, R. G., JO A. L. D. C., FREITAS, C. M. D. S. "Broad-phase collision detection using semi-adjusting BSP-trees". In: *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pp. 179–186, New York, NY, USA, 2005. ACM. ISBN: 1-59593-013-2. doi: http://doi.acm.org/10.1145/1053427.1053457.

[52] ZHANG, D., YUEN, M. M. F. "Collision Detection for Clothed Human Animation". In: *PG '00: Proceedings of the 8th Pacific Conference on Computer Graphics and Applications*, p. 328, Washington, DC, USA, 2000. IEEE Computer Society. ISBN: 0-7695-0868-5.

[53] BANDI, S., THALMANN, D. "An Adaptive Spatial Subdivision of the Object Space for Fast Collision Detection of Animated Rigid Bodies", *Computer Graphics Forum*, v. 14, n. 3, pp. 259–270, 1995. Disponível em: <citeseer.ist.psu.edu/bandi93adaptive.html>.

[54] TELLER, S. J., SÉQUIN, C. H. "Visibility preprocessing for interactive walkthroughs". In: *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pp. 61–70, New York, NY, USA, 1991. ACM. ISBN: 0-89791-436-8. doi: http://doi.acm.org/10.1145/122718.122725.

[55] LEFEBVRE, S., HOPPE, H. "Perfect spatial hashing". In: *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pp. 579–588, New York, NY, USA, 2006. ACM. ISBN: 1-59593-364-6. doi: http://doi.acm.org/10.1145/1179352.1141926.

[56] BACIU, G., WONG, W., SUN, H. "RECODE: An Image-based Collision Detection Algorithm", *The Journal of visualization and Computer Animation*, v. 10, n. 4, pp. 181–192, 1998. ISSN: 1049-8907. doi: http://www3.interscience.wiley.com/cgi-bin/abstract/68501005/ABSTRACT.

[57] FAURE, F., BARBIER, S., ALLARD, J., et al. "Image-based Collision Detection and Response between Arbitrary Volumetric Objects". In: *ACM Siggraph/Eurographics Symposium on Computer Animation (SCA)*, pp. 155–162, jul. 2008.

[58] EVERITT, C. "Interactive order-independent transparency". 2001.

[59] MÜLLER, M., CHARYPAR, D., GROSS, M. "Particle-based fluid simulation for interactive applications". In: *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN: 1-58113-659-5.

[60] CRANE, K., LLAMAS, I., TARIQ, S. "Real Time Simulation and Rendering of 3D Fluids". In: Nguyen, H. (Ed.), *GPUGems 3*, cap. 30, Addison-Wesley, 2007.

[61] COHEN, J. M., TARIQ, S., GREEN, S. "Interactive fluid-particle simulation using translating Eulerian grids". In: *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10,

pp. 15–22, New York, NY, USA, 2010. ACM. ISBN: 978-1-60558-939-8. doi: http://doi.acm.org/10.1145/1730804.1730807. Disponível em: <http://doi.acm.org/10.1145/1730804.1730807>.

[62] ROBINSON-MOSHER, A., SHINAR, T., GRETARSSON, J., et al. "Two-way coupling of fluids to rigid and deformable solids and shells", *ACM Trans. Graph.*, v. 27, n. 3, pp. 1–9, 2008. ISSN: 0730-0301. doi: http://doi.acm.org/10.1145/1360612.1360645.

[63] MARTIN, S., KAUFMANN, P., BOTSCH, M., et al. "Unified simulation of elastic rods, shells, and solids". In: *SIGGRAPH '10: ACM SIGGRAPH 2010 papers*, pp. 1–10, New York, NY, USA, 2010. ACM. ISBN: 978-1-4503-0210-4. doi: http://doi.acm.org/10.1145/1833349.1778776.

[64] TERZOPOULOS, D., PLATT, J., BARR, A., et al. "Elastically deformable models". In: *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pp. 205–214, New York, NY, USA, 1987. ACM. ISBN: 0-89791-227-6. doi: http://doi.acm.org/10.1145/37401.37427.

[65] BARAFF, D. "Linear-time dynamics using Lagrange multipliers". In: *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 137–146, New York, NY, USA, 1996. ACM. ISBN: 0-89791-746-4. doi: http://doi.acm.org/10.1145/237170.237226.

[66] STEWART, D. E. "Rigid-Body Dynamics with Friction and Impact", *SIAM Rev.*, v. 42, n. 1, pp. 3–39, 2000. ISSN: 0036-1445. doi: http://dx.doi.org/10.1137/S0036144599360110.

[67] KAUFMAN, D. M., SUEDA, S., JAMES, D. L., et al. "Staggered projections for frictional contact in multibody systems". In: *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pp. 1–11, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1457515.1409117.

[68] BARAFF, D. "Analytical methods for dynamic simulation of non-penetrating rigid bodies", *SIGGRAPH Comput. Graph.*, v. 23, n. 3, pp. 223–232, 1989. ISSN: 0097-8930. doi: http://doi.acm.org/10.1145/74334.74356.

[69] PAULY, M., PAI, D., GUIBAS, L. "Quasi-rigid objects in contact". In: *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 109–119, Aire-la-Ville, Switzerland,

Switzerland, 2004. Eurographics Association. ISBN: 3-905673-14-2. doi: http://doi.acm.org/10.1145/1028523.1028539.

[70] KRY, P. G., PAI, D. K. "Continuous contact simulation for smooth surfaces", *ACM Trans. Graph.*, v. 22, n. 1, pp. 106–129, 2003. ISSN: 0730-0301. doi: http://doi.acm.org/10.1145/588272.588280.

[71] INSTITUT, X. P., PROVOT, X. "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior". In: *In Graphics Interface*, pp. 147–154, 1996.

[72] FAURE, F. "Interactive Solid Animation Using Linearized Displacement Constraints". In: *9 th Eurographics Workshop on Computer Animation and Simulation*. e, 1998.

[73] JAKOBSEN, T. "Gamasutra- Features: Advanced Character Physics". 2003.

[74] MÜLLER, M., HEIDELBERGER, B., HENNIX, M., et al. "Position based dynamics", *J. Vis. Comun. Image Represent.*, v. 18, n. 2, pp. 109–118, 2007. ISSN: 1047-3203. doi: http://dx.doi.org/10.1016/j.jvcir.2007.01.005.

[75] GOLDENTHAL, R., HARMON, D., FATTAL, R., et al. "Efficient simulation of inextensible cloth", *ACM Trans. Graph.*, v. 26, n. 3, pp. 49, 2007. ISSN: 0730-0301. doi: http://doi.acm.org/10.1145/1276377.1276438.

[76] SIFAKIS, E., SHINAR, T., IRVING, G., et al. "Hybrid simulation of deformable solids". In: *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 81–90, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN: 978-1-59593-624-4.

[77] SHINAR, T., SCHROEDER, C., FEDKIW, R. "Two-way coupling of rigid and deformable bodies". In: *SCA '08: Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 95–103, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. ISBN: 978-3-905674-10-1.

[78] CARLSON, M., MUCHA, P. J., TURK, G. "Rigid fluid: animating the interplay between rigid bodies and fluid". In: *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pp. 377–384, New York, NY, USA, 2004. ACM. doi: http://doi.acm.org/10.1145/1186562.1015733.

[79] CHENTANEZ, N., GOKTEKIN, T. G., FELDMAN, B. E., et al. "Simultaneous coupling of fluids and deformable bodies". In: *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 83–89, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. ISBN: 3-905673-34-7.

[80] LENAERTS, T., ADAMS, B., DUTRÉ, P. "Porous flow in particle-based fluid simulations". In: *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pp. 1–8, New York, NY, USA, 2008. ACM. ISBN: 978-1-4503-0112-1. doi: http://doi.acm.org/10.1145/1399504.1360648.

[81] STAM, J. "Nucleus: Towards a unified dynamics solver for computer graphics". pp. 1 –11, aug. 2009. doi: 10.1109/CADCG.2009.5246818.

[82] HARADA, T., KOSHIZUKA, S., KAWAGUCHI, Y. "Sliced data structure for particle-based simulations on GPUs". In: *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, GRAPHITE '07, pp. 55–62, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-912-8. doi: http://doi.acm.org/10.1145/1321261.1321271. Disponível em: <`http://doi.acm.org/10.1145/1321261.1321271`>.

[83] BARAFF, D. "Curved surfaces and coherence for non-penetrating rigid body simulation". In: *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pp. 19–28, New York, NY, USA, 1990. ACM. ISBN: 0-201-50933-4. doi: http://doi.acm.org/10.1145/97879.97881.

[84] BARAFF, D. "Interactive Simulation of Solid Rigid Bodies", *IEEE Comput. Graph. Appl.*, v. 15, n. 3, pp. 63–75, 1995. ISSN: 0272-1716. doi: http://dx.doi.org/10.1109/38.376615.

[85] POPOVIĆ, J., SEITZ, S. M., ERDMANN, M., et al. "Interactive manipulation of rigid body simulations". In: *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 209–217, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN: 1-58113-208-5. doi: http://doi.acm.org/10.1145/344779.344880.

[86] SPILLMANN, J., TESCHNER, M. "Contact Surface Computation for Coarsely Sampled Deformable Objects". In: *Proceedings of Vision, Modeling, Visualization VMV'05*, pp. 16–18, Stanford, USA, 2005. doi: http://cg.informatik.uni-freiburg.de/people/spillma/research.htm.

[87] SHOEMAKE, K., DUFF, T. "Matrix animation and polar decomposition". In: *Proceedings of the conference on Graphics interface '92*, pp. 258–264, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN: 0-9695338-1-0.

[88] SUMALI, H. *A New Adaptive Array of Vibration Sensors*. Tese de Doutorado, Mechanical Engineering Virginia Polytechnic Institute and State University, Virginia, USA, 1992.

[89] ALZAMORA, G., ATENCIO, Y. P., ESPERANÇA, C. "Simulation of Deformable Bodies Based on Tetrahedral Meshes and Shape Matching". In: *Proceedings of SBGames'08 - VII Brazilian Symposium on Computer Games and Digital Entertainment*. Sociedade Brasileira de Computação, SBC, 2008. ISBN: 85-766-9204-X.

[90] ATENCIO, Y., ALZAMORA, G., ESPERANÇA, C. "Enhanced physically-based animation of deformable bodies using shape-matching", *Comput. Entertain.*, v. 7, pp. 52:1–52:19, January 2010. ISSN: 1544-3574. doi: http://doi.acm.org/10.1145/1658866.1658871. Disponível em: <`http://doi.acm.org/10.1145/1658866.1658871`>.

[91] MIRTICH, B. V. *Impulse-based dynamic simulation of rigid body systems*. Tese de Doutorado, 1996. AAI9723116.

[92] SATISH, N., HARRIS, M., GARLAND, M. "Designing efficient sorting algorithms for manycore GPUs", *Parallel and Distributed Processing Symposium, International*, v. 0, pp. 1–10, 2009. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2009.5161005.

[93] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., et al. "A Survey of General-Purpose Computation on Graphics Hardware". In: *Eurographics 2005, State of the Art Reports*, pp. 21–51, ago. 2005.

[94] HARRIS, M. "Fast fluid dynamics simulation on the GPU". In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, p. 220, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1198555.1198790.

[95] ZELLER, C. "Cloth simulation on the GPU". In: *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1187112.1187158. Disponível em: <`http://doi.acm.org/10.1145/1187112.1187158`>.

[96] GREEN, S. *Particle Simulation using CUDA*. In: Report, NVIDIA, 2007.

[97] BARAFF, D. "An introduction to physically based modeling: Rigid body simulation - unconstrained rigid body dynamics". In: *In An Introduction to Physically Based Modelling, SIGGRAPH '97 Course Notes*, p. 97, 1997.

[98] TANAKA, M., SAKAI, M., ISHIKAWAJIMA-HARIMA, et al. "Rigid body simulation using a particle method". In: *ACM SIGGRAPH 2006 Research posters*, SIGGRAPH '06, New York, NY, USA, 2006. ACM. ISBN: 1-59593-364-6. doi: http://doi.acm.org/10.1145/1179622.1179775. Disponível em: <http://doi.acm.org/10.1145/1179622.1179775>.

[99] MISHRA, B. K. "A review of computer simulation of tumbling mills by the discrete element method: Part II–Practical applications", *International Journal of Mineral Processing*, v. 71, n. 1-4, pp. 95 – 112, 2003. ISSN: 0301-7516. doi: DOI:10.1016/S0301-7516(03)00031-0. Disponível em: <http://www.sciencedirect.com/science/article/B6VBN-48J45N7-1/2/c584739799c6b54f2d73e85e5e167718>.

[100] HARRIS, M., SENGUPTA, S., OWENS, J. D. "Parallel Prefix Sum (Scan) with CUDA". In: Nguyen, H. (Ed.), *GPU Gems 3*, Addison Wesley, ago. 2007.

[101] REDON, S., KHEDDAR, A., COQUILLART, S. "Fast continuous collision detection between rigid bodies". In: *Proc. of Eurographics (Computer Graphics Forum)*, p. 2002, 2002.

[102] DESBRUN, M., PAULE GASCUEL, M. "Smoothed Particles: A new paradigm for animating highly deformable bodies". In: *In Computer Animation and Simulation S̃96 (Proceedings of EG Workshop on Animation and Simulation*, pp. 61–76. Springer-Verlag, 1996.

[103] PAIVA, A., PETRONETTO, F., LEWINER, T., et al. *Meshless fluid simulation: introduction to SPH methods*. Rio de Janeiro, IMPA, august 2009.

[104] KILGARD, M. J. "The OpenGL Utility Toolkit (GLUT) Programming Interface". 1996.

[105] COUTINHO, B. B., OLIVEIRA, A. A. F., ATENCIO, Y. P., et al. "Rain Scene Animation through Particle Systems and Surface Flow Simulation by SPH". In: *Proceedings of the 2010 23rd SIBGRAPI Conference on Graphics, Patterns and Images*, SIBGRAPI '10, pp. 255–262, Washington, DC, USA, 2010. IEEE Computer Society. ISBN: 978-0-7695-4230-0. doi: http://dx.doi.org/10.1109/SIBGRAPI.2010.42. Disponível em: <http://dx.doi.org/10.1109/SIBGRAPI.2010.42>.

[106] MÜLLER, M., CHENTANEZ, N. "Solid simulation with oriented particles", *ACM Trans. Graph.*, v. 30, pp. 92:1–92:10, August 2011. ISSN: 0730-0301. doi: http://doi.acm.org/10.1145/2010324.1964987. Disponível em: <http://doi.acm.org/10.1145/2010324.1964987>.

[107] JOLLIFFE, I. *Principal Component Analysis*. Springer Verlag, 1986.