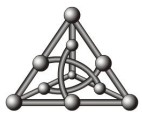
UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO E ESTATÍSTICA

Motor de Física de Corpos Rígidos em GPU com Arquitetura CUDA

Márcio Artacho Peres

ORIENTADOR: Professor Dr. Paulo A. Pagliosa



Campo Grande
2008

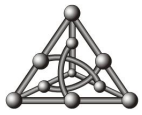
UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO E ESTATÍSTICA

Motor de Física de Corpos Rígidos em GPU com Arquitetura CUDA

Márcio Artacho Peres

Dissertação apresentada ao Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

ORIENTADOR: Professor Dr. Paulo A. Pagliosa



Durante a elaboração deste projeto o autor recebeu apoio financeiro da CAPES.

Campo Grande
2008

Aos meus pais Narciso e Maria Inês, irmã Vanessa e a minha namorada Magda pelo apoio e compreensão quando não pude lhes dar maior atenção.
Márcio

Agradecimentos

É difícil resumir com poucas palavras todos os agradecimentos necessários aos que contribuíram de alguma maneira para a finalização desse projeto. Agradeço a todos os professores e colegas do curso de Mestrado em Ciência da Computação da UFMS, pelas diversas conversas e discussões que, mesmo informais, contribuíram para o aprendizado.

Agradeço à minha família pela compreensão e pelo apoio que sempre tive, tanto no período universitário quanto nos anos anteriores. E obrigado Deus, pela força principalmente nos momentos de mais dificuldades.

Agradeço à Capes e à Fundect pelo apoio financeiro, e ao Professor Henrique Mongelli por ter disponibilizado, com recursos do CNPQ, o equipamento com suporte à CUDA, necessário para o desenvolvimento deste trabalho.

Agradeço aos meus companheiros de projeto, orientador Paulo Pagliosa e Alexandre Soares, ambos com colaboração fundamental para o desenvolvimento deste.

Resumo

Peres, M. A. *Um Motor de Física de Corpos Rígidos em GPU com Arquitetura CUDA*. Dissertação (Mestrado em Ciência da Computação), Universidade Federal de Mato Grosso do Sul, 2008.

O objetivo principal deste trabalho é o desenvolvimento de um detector de colisões e de um motor de física para simulação dinâmica de corpos rígidos usando unidades de processamento gráfico (GPUs) que oferecem suporte à CUDA (*Computer Unified Device Architecture*). Ambos os componentes integram um framework de animação dinâmica chamado AS, desenvolvido pelo Grupo de Visualização, Simulação e Games do DCT/UFMS. A implementação de um detector de colisões e de um motor de física em CUDA para AS permite que este execute toda ou parte da física em GPU, liberando a CPU para outras tarefas e propiciando a simulação de cenas com um número maior de atores em tempo real.

A simulação de uma cena em um instante de tempo começa com a determinação de todos os pontos de contato que ocorrem entre pares de atores. A detecção de colisões é dividida em uma fase geral e uma fase exata. Na primeira, o espaço da cena é dividido em uma estrutura de células de mesmo tamanho. Somente os atores que estão dentro ou interceptam uma determinada célula da estrutura e cujos volumes limitantes de interceptam podem colidir. Dado um par de atores potencialmente em contato, a fase exata verifica se estes de fato colidem através de cálculos de intersecção entre as formas que definem a geometria dos atores. A seguir, o motor de física computa as forças de restrição oriundas dos contatos e de *junções* entre os atores. Este é um *problema de complementaridade linear* (PCL) resolvido iterativamente com o algoritmo de Gauss-Seidel com sobre-relaxação sucessiva (SOR). As forças de restrição, somadas às forças externas aplicadas, são usadas pelo motor para determinação da velocidade e posição atualizadas de cada corpo rígido da cena, o que é feito pela integração numérica das equações de movimento através do método de Euler.

Os testes efetuados com a implementação paralela em CUDA do detector de colisão e do motor de física de AS demonstraram que a GPU pode ser empregada efetivamente na simulação dinâmica em tempo real de cenas constituídas de milhares de corpos rígidos com milhares de restrições, com desempenho até duas vezes mais eficiente que a versão para CPU dependendo da aplicação.

Palavras-chave: *GPGPU*, *CUDA*, *motor de física*.

Abstract

Peres, M. A. *Um Motor de Física de Corpos Rígidos em GPU com Arquitetura CUDA*. Master's Thesis, Universidade Federal de Mato Grosso do Sul, 2008.

The main purpose of this work is the development of a collision detector and a physics engine for dynamic simulation of rigid bodies on graphics processing units with support to CUDA (Computer Unified Device Architecture). Both the components are part of a framework for dynamic-based animation called AS, witch was developed by the researchers of the Group of Visualization, Simulation and Games of the DCT/UFMS. The implementation of a collision detector and a physics engine on CUDA enables AS to execute the physics loop partial or entirely on GPU, allowing the CPU to run other tasks and performing real time simulations of scenes made of a large number of actors.

The first step of simulating a scene at a given instant of time is to determine all contact points among the actors. The collision detection is divided in a broad and a narrow phase. In the broad phase the scene space is regularly subdivided in cells. Only the actors inside or intercepting a particular cell and whose bounding volumes intersect each other are able to collide. Given a pair of actors that potentially can be in contact, the narrow phase verifies if they actually collide by computing the intersection points among the geometric shapes of the actors. Following, the physics engine calculates the constraint forces to avoid the interpenetration of the contacting bodies and to maintain the *joints* between the actors of the scene. This is a *linear complementarity problem* (LCP) which is iteratively solved by using the Gauss-Seidel algorithm with successive over relaxation (SOR). The physics engine uses these constraint forces as well as the extern forces applied to the bodies in order to update the velocity and the position of the actors of the scene. This is performed with the employ of the Euler method to numerically integrate the equations of motion of the rigid bodies.

The results obtained from the parallel implementation on CUDA of the collision detector and the physics engine of AS shown that the GPU can be effectively used for real time simulation of scenes made of thousands of rigid bodies with thousands of constraints, with a speedup of two when compared to the version for CPU, depending on the application.

Keyword: *GPGPU*, *CUDA*, *physic engine.*

Conteúdo

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

1.1 Motivações e justificativas

Este trabalho envolve o estudo do estado-da-arte em tecnologias relacionadas à simulação dinâmica de corpos rígidos em tempo real, com ênfase no desenvolvimento de aplicações interativas baseadas em tais tecnologias, incluindo jogos digitais 3D. Um dos motivos de interesse em pesquisa e desenvolvimento de jogos digitais é que este é atualmente considerado, em muitos países, setor estratégico dentro da indústria tecnológica. Isto se deve ao fato de que jogos digitais podem ser desenvolvidos para servir como ferramentas interativas de visualização e simulação, usadas tanto para fins de entretenimento como para fins específicos (por exemplo, treinamento de pilotos em simuladores de vôo) em áreas diversas tais como engenharia (de estruturas, automobilística, aeroespacial), exploração de petróleo e medicina, entre outras aplicações.

A complexidade do desenvolvimento de um jogo digital 3D deve-se à sua própria natureza multi e interdisciplinar e ao fato de que se espera atualmente que estes sejam capazes de prover, em tempo real, o maior grau de realismo possível, tanto no aspecto gráfico como no aspecto de simulação. O interesse pelo acréscimo de realismo físico em jogos digitais é resultado não somente do aumento de velocidade das CPUs, mas também da evolução das unidades de processamento gráfico (*graphics processing units*, ou GPUs), as quais implementam em hardware muitas das funções de visualização. Além disso, espera-se das aplicações de simulação que estas sejam capazes de tratar de um número cada vez maior de objetos em tempo real, da ordem de centenas ou milhares, como ilustrado na Figura 1.1.

Tendo em vista a grande demanda computacional em simulações em tempo real, investiga-se a possibilidade de uso de hardware, além da CPU, para auxiliar na simulação. Algumas possibilidades são o uso da PPU (physics processing unit, ou unidade de processamento de física), o uso de uma CPU extra ou o uso de uma GPU.

PPU A primeira PPU do mercado, lançada pela Ageia [AGE05] no ano de 2005, é uma placa avulsa que conectada ao computador é capaz de auxiliar especificamente o processamento de física. Foi projetada para permitir cenas mais realistas com implementações de aceleração na dinâmica de corpos rígidos, detecção de colisões, dinâmica de fluidos, entre outras. A principal desvantagem da PPU é a necessidade de aquisição de um novo hardware.

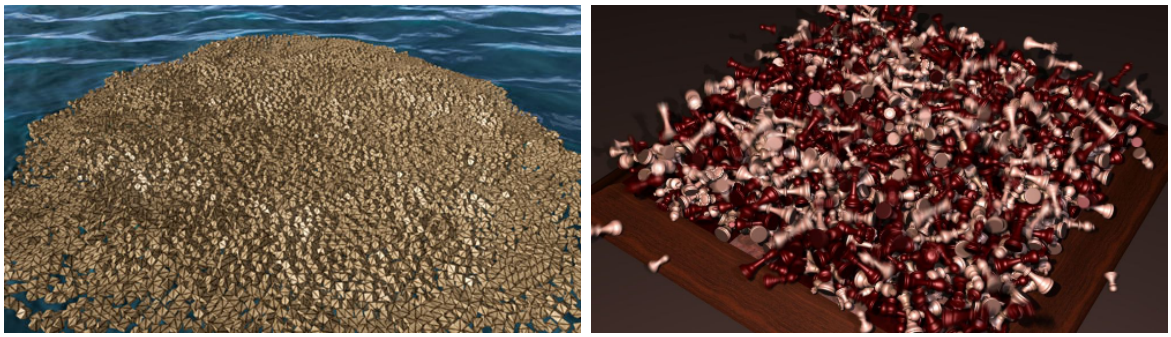


Figura 1.1: Exemplos de simulação de corpos rígidos. A imagem da esquerda ilustra a simulação de aproximadamente dez mil caixas flutuando na água, ou seja, além da colisão de corpos rígidos, há também simulação de fluidos. Na imagem da direita a simulação consiste em mil peças de xadrez sendo lançadas em um tabuleiro. (As imagens foram extraídas de *http://www.cemyuksel.com/research/waveparticles/* e *http://www.cs.ubc.ca/~dkaufman/*, respectivamente.)

CPU Atualmente, é comum CPUs contendo até quatro núcleos. Nesse caso, muitas vezes, três dos núcleos ficam ociosos devido à baixa demanda de processamento. Os núcleos extras da CPU poderiam ser responsáveis pelo processamento da física, mas a CPU iria necessitar de muitos núcleos para igualar a capacidade de processamento paralelo de uma PPU ou GPU. A Intel espera alcançar 64 núcleos em uma CPU em 2010, ultrapassando facilmente a capacidade das PPU e GPU mais recentes. No entanto, atualmente o poder computacional necessário para acelerar os cálculos da física não está nas CPUs.

GPU A terceira possibilidade é utilizar uma unidade de processamento gráfico, ou GPU, para servir de co-processador da simulação. O uso de GPU para aplicações não gráficas é uma área relativamente nova da computação denominada GPGPU (computação de propósito geral em GPU) que vem recebendo interesse crescente em diversos campos, tais como métodos numéricos, geometria computacional e biologia computacional, entre outros. Isto se deve a algumas características: relação entre custo e desempenho, aumento dos recursos de programação e capacidade de processamento (principalmente nas GPUs atuais) e arquitetura paralela. A título de comparação, um processador Intel Core 2 Duo E8400 tem capacidade teórica de execução de 48 GFLOPS, enquanto que uma GPU NVidia GeForce 8800 GTX, tem capacidade de 576 GFLOPS. Por isso optou-se pela utilização desse hardware para o presente estudo.

O Grupo de Visualização, Simulação e Games (GVSG) do DCT-UFMS desenvolveu um sistema de animação chamado AS [Oli06]. Este sistema foi revisado por [dS08] a fim de possibilitar seu uso em aplicações de simulação dinâmica interativas e em tempo real. As principais alterações em relação à versão original são a implementação de um laço principal que sincroniza a renderização, implementação de um detector de colisões e motor de física próprios (em CPU); além disso, conta com extensões de suporte à execução de scripts e ações e tratamento de eventos de entrada do usuário em tempo real.

Este trabalho pretende prover uma implementação do detector de colisões e do motor de física de AS para GPUs CUDA (Compute Unified Device Architecture) — uma nova arquitetura que permite utilizar a GPU como um multiprocessador paralelo capaz de resolver problemas a partir de programas escritos em uma linguagem baseada em C. O propósito geral é investigar a possibilidade de emprego eficiente deste tipo de GPU em aplicações desta natureza. A implementação destes componentes de AS em GPU teria a vantagem de liberar a CPU para outros tipos de processamento que poderiam ser executados em paralelo com

a GPU. Além disso, o trabalho é motivado pela possibilidade de se ter cenas mais realistas e com um número maior de objetos sendo simulados em tempo real, dado a capacidade de processamento mencionada acima.

1.2 Objetivos e contribuições

O objetivo geral deste trabalho é o desenvolvimento para GPUs CUDA de dois componentes de AS fundamentais para simulação dinâmica: o detector de colisões e o motor de física.

Os objetivos específicos são:

- Estudar os fundamentos matemáticos e computacionais necessários ao entendimento e desenvolvimento de um detector de colisões e dos componentes que compõem um motor de física.
- Estudar o modelo de programação CUDA.
- Verificar a possibilidade de uso eficiente de GPUs em aplicações interativas de simulação de corpos rígidos em tempo real, incluindo jogos digitais.
- Descrever em detalhes a implementação paralela do detector de colisões e do motor de física.

A principal contribuição do trabalho é a implementação *orientada a objetos* do motor de física em CUDA em si. Este é capaz de simular *estavelmente* corpos rígidos em contato com ou sem atrito, constituídos de múltiplas formas (esfera, caixa e cápsula) e unidos por vários tipos de junções (esférica, de revolução e fixa, na versão corrente). O número máximo de atores, junções e formas é limitado somente pela quantidade de memória disponível na GPU. O motor provê uma fundação de componentes que podem ser utilizados como fonte de ensino de graduação e pós-gradação e pesquisa em diversas disciplinas como Física e Computação Gráfica. A partir dos resultados obtidos neste trabalho, pode-se adicionar mais facilmente outras funcionalidades ao motor, tais como simulação de fluidos e corpos flexíveis. Também outros métodos de detecção de colisões podem ser considerados.

Interessante mencionar que, em fevereiro de 2008, a NVidia adquiriu a Ageia, proprietária do motor de física denominado PhysX [Cor08]. De acordo com a NVidia, além da versão do PhysX para CPU, ou seja, capaz de ser executado em qualquer computador, a empresa tem intenção de estudar, otimizar e implementar o software de física para GPUs com suporte a CUDA (a partir da série GeForce 8). Esta estratégia pode ter como justificativas o avanço da capacidade computacional das GPUs atuais somada a seu custo acessível, o que permite que estas possam se constituir como acessório relativamente comuns em computadores, não havendo, assim, a necessidade de aquisição de um novo hardware para o funcionamento do sistema.

Sendo assim, pode-se perceber a relevância do presente trabalho, visto que, apesar de um estudo acadêmico, este é um dos objetivos da NVidia com seu recente adquirido motor de física.

1.3 Resumo do trabalho

A simulação dinâmica de uma cena cujos atores são corpos rígidos consiste em, conhecidas as posições e velocidades de cada corpo rígido em cada instante de tempo t da simulação, determinar as novas posições e velocidades em um instante $t + \Delta t$, onde Δt é o passo de

tempo. Esta determinação é decorrente da integração da *equação de movimento* de cada corpo, a qual é formulada em termos da inércia e das forças atuantes no corpo, sendo estas compostas pelas forças externas e pelas forças de restrição decorrentes dos contatos e de *junções* entre dois corpos. A computação das forças de restrição pode ser formulada como um *problema de complementaridade linear* (PCL), como descrito no Capítulo 2. Os passos envolvidos na simulação dinâmica de uma cena em um instante de tempo t são:

1. Detecção de colisões, responsável pela determinação dos pontos de contato entre os atores da cena. O componente do sistema que efetua este passo é denominado detector de colisões.

2. Montagem do sistema correspondente ao problema de complementaridade linear.

3. Resolução do PCL, responsável pela determinação simultânea de todas as forças de restrição nos atores da cena.

4. Aplicação das forças totais e integração da equação de movimento a fim de se determinar a posição e velocidade de cada ator da cena. O componente do sistema que trata dos passos 2, 3, e 4 é chamado motor de física.

Neste trabalho ambos os componentes foram implementados para GPUs NVidia com arquitetura CUDA. O modelo de programação consiste em uma arquitetura com unidades de processamento paralelas, onde vários conjuntos de dados, ou *streams*, podem ser processados simultaneamente pelo mesmo conjunto de instruções, ou *kernel*, executados em unidades distintas. Para usufruir da capacidade dessa arquitetura é necessário que programas sejam escritos para execução de tarefas em paralelo. Os problemas mais adequados para serem solucionados nesse modelo são aqueles cujos dados do processamento são independentes, tornando a paralelização direta. Por outro lado, em problemas com interdependência entre os dados, algumas técnicas devem ser utilizadas para a subdivisão do problema em partes independentes, as quais são solucionadas de forma paralela.

Detector de colisões

O detector de colisão determina, para todos objetos da cena, se dois objetos colidem e em qual posição. A detecção de colisões é dividida em duas fases: fase geral e fase exata. A primeira consiste na aplicação de testes rápidos com o objetivo de descartar pares de objetos que não colidem, diminuindo, assim, os cálculos mais complicados feitos na fase exata. Esta trata da determinação precisa dos pontos de contato entre os pares de objetos provindos da primeira etapa.

A fase geral implementada no detector de colisões é baseada em um esquema de subdivisão espacial através de uma estrutura regular, denominada *grid*, em células. Cada objeto é associado a uma ou mais células, cuja dimensão, uniforme, deve ser maior que o maior objeto da cena. Os testes de colisão, para a criação dos pares de colisão, são efetuados entre objetos associados à mesma célula ou células adjacentes. De modo geral, percebe-se uma independência entre os dados a serem processados, principalmente durante a etapa de associação, permitindo o paralelismo. Uma implementação para GPU da fase geral foi elaborada com base em [Gra07].

Na fase exata há também independência, pois a determinação dos pontos de contato de um par de objetos não influencia na determinação dos outros. Portanto, todos os pares podem ser analisados de forma paralela. Uma versão para GPU também foi implementada.

Motor de física

O movimento de corpos rígidos geralmente não é irrestrito, mas condicionado a determinadas restrições, as quais podem ser devidas a *junções* e *contatos* entre os corpos. Por exemplo, dois corpos rígidos podem ser unidos de tal forma que um esteja restringido a girar ao redor do outro, como em uma dobradiça. Um outro exemplo é quando dois corpos rígidos chocam-se. O movimento subseqüente deve ser tal que os corpos se afastem um do outro, evitando assim a interpenetração. Em ambos os exemplos, além das forças externas às quais os corpos estão submetidos (gravidade, vento, etc.), ocorrem também forças de restrições que influenciam no movimento dos corpos, no primeiro caso, fazendo com que eles permaneçam unidos e no segundo, os repelindo e evitando a interpenetração.

Uma das funções mais importantes do motor de física é a determinação *simultânea*, em um dado instante de tempo, de todas as forças de resrições atuantes nos corpos rígidos de uma cena, oriundas de junções e contatos. Matematicamente, esta questão pode ser formulada como um problema de complementaridade linear, ou PCL [RWC92]. Uma vez determinados os pontos de contato pelo detector de colisões, o primeiro passo do motor de física é, a partir desses contatos e também das junções entre corpos, a montagem da matriz e vetores do PCL, conforme explicado no Capítulo 2.

O método de resolução do PCL implementado no trabalho é uma versão em paralelo do algoritmo iterativo do SOR (Successive Over Relaxation, ou Sobre Relaxação Sucessiva) [Erl04]. SOR (oriundo do método de *Gauss-Seidel*) é um método iterativo utilizado para determinação da solução (aproximada) de sistemas lineares, porém este pode ser estendido para resolução de PCL, com poucas modificações. Um método é iterativo quando fornece uma seqüência de aproximantes da solução, cada uma das quais obtida das anteriores pela repetição do mesmo processo. A cada iteração, a solução obtida torna-se mais próxima da solução exata. Apesar de sua convergência ser lenta para uma solução bastante precisa, soluções satisfatórias para animações podem ser alcançadas com poucas iterações. Umas das justificativas de utilização do método iterativo é a possibilidade de configuração da relação precisão dos resultados e tempo de processamento, através da determinação do número máximo de iterações a processar.

A solução para GPU é baseada no método de blocos que resumi-se na divisão do sistema de n linhas em $\lceil \frac{n}{m} \rceil$ blocos de m linhas cada. Os blocos são executados em paralelo (Jacobi) e as linhas internas ao bloco de forma seqüencial (Gauss-Seidel). Detalhes sobre o método e implementações são abordados no Capítulo 2.

A integração da equação de movimento é baseada em um método numérico de resolução de equações diferenciais ordinárias, ou EDOs, denominado método de Euler. Usou-se o método de integração de Euler por ser simples e eficiente numérica e computacionalmente, ou seja, soluções numéricas por ele encontradas são, pelo menos visualmente, convincentes do comportamento físico dos objetos e isso é realizado de forma rápida pelo computador. Uma discussão mais aprofundada sobre métodos de integração numérica pode ser encontrada em [BW97]. O processo de integração da equação de movimento pode ser eficientemente implementado com auxílio de GPU, uma vez que esta é totalmente independente entre os corpos, portanto pode ser executada em paralelo.

1.4 Revisão bibliográfica

Na literatura, há diversos trabalhos de diferentes áreas que abordam o uso da GPU [1]. Na programação das GPUs tradicionais, o trabalho de [PF05] apresenta as últimas técnicas para

[1] http://www.gpgpu.org.

a computação de propósito geral utilizando essa arquitetura.

Na programação das GPUs atuais, Nguyen [Ngu07] mostra as recentes técnicas para a nova arquitetura de programação CUDA. Mais especificamente as partes do livro relacionadas com o presente trabalho são simulação física e computação em GPU.

Na detecção de colisão, Knott e Pai [KP03] apresentam um algoritmo que utiliza a GPU como co-processador diminuindo o volume de computação na detecção de colisão de poliedros. Este é linear ao número de objetos e não utiliza pré-processamento e nenhuma estrutura de dados especial. Sathe e Lake [SL06] propõem uma técnica para detecção de colisão de corpos rígidos nas GPUs atuais usando *cube-maps*. A criação de um *cube-map* para cada objeto é feita em um pré-processamento para aumentar a eficiência do algoritmo.

Recentemente, Vasely [Vas08] utilizou a GPU para implementação em paralelo de métodos iterativos para a solução de sistemas lineares esparsos, incluindo Jacobi e Gauss-Seidel. Segundo Vasely, o maior ganho da implementação no dispositivo comparado com uma implementação seqüencial foi de 3 vezes em sistemas da ordem de 10^6.

Alguns trabalhos abordam o uso da GPU para resolução de EDOs, notadamente aqueles relacionados à simulação de sistemas de partículas. Neste campo Kipfer e outros apresentam um método que inclui a detecção e tratamento de colisões entre partículas. Este usa a GPU para ordenar as partículas em uma fase geral de detecção de colisão [KSW04]. Em um trabalho simultâneo, Kolb e outros propõem um simulador de sistemas de partículas em GPU capaz de determinar com precisão a colisão de partículas com outras geometrias da cena [KLRS04].

Em 2006, NVidia e Havok inovaram com a física para *games* apresentando o Havok FX, uma API (*application programming interface*) para simulação de partículas e corpos rígidos executadas em GPU. A API ainda possui suporte à detecção e tratamento de colisões entre partículas e corpos rígidos bem como processamento e visualização da simulação em GPUs separadas (em sistemas com múltiplas GPUs).

Acreditando nessa idéia a Intel acabou por adquirir a empresa Havok, e pretende futuramente desenvolver seu próprio hardware gráfico. Parte desse investimento deve ser para manter sua posição no mercado mediante sua maior concorrente, a AMD, que em 2006 comprou uma das duas maiores fabricantes de hardware gráfico, a ATI.

1.5 Organização do texto

O restante do texto está estruturado em seis capítulos comentados a seguir.

Capítulo 2
Motor de física de AS

Este capítulo apresenta um resumo dos componentes do motor de física de corpos rígidos. O problema de complementaridade linear é apresentado juntamente com um método de resolução. Em seguida, é mostrada a implementação do solucionador de PCL, do solucionador de EDO e suas principais classes.

Capítulo 3
Detecção de colisão

Neste capítulo são mostrados as responsabilidades do detector de colisões. Detalhes de implementação do detector de colisões são apresentados, juntamente com suas classes fundamentais.

Capítulo 4
GPGPU com CUDA

Este capítulo define GPGPU. São introduzidos o modo de programação, a arquitetura e dificuldades tanto das GPUs tradicionais quanto da nova arquitetura para programação em GPUs CUDA. Uma implementação do solucionador de EDO utilizando GPGPU tradicional também é apresentada.

Capítulo 5
Implementação em CUDA

Este capítulo apresenta as principais classes da implementação em GPU do motor de física. O capítulo inicia com a implementação da fase geral do detector de colisões em CUDA. Em seguida apresenta a implementação do solucionador de PCL em CUDA.

Capítulo 6
Exemplos

A apresentação e análise dos resultados de aplicações exemplos de simulação executadas com o motor são mostradas neste capítulo. Tabelas com variações entre números de atores são apresentadas para ambas arquiteturas, com os respectivos *speedups*.

Capítulo 7
Conclusão

Neste capítulo são apresentadas as conclusões, comentários finais e sugestões de trabalhos futuros.

CAPÍTULO 2

Motor de Física do AS

2.1 Introdução

Este capítulo apresenta um resumo do motor de física de corpos rígidos de AS desenvolvido por [dS08], o qual foi implementado em GPU conforme descrito no Capítulo 5. As funcionalidades do motor são descritas a seguir. Maiores detalhes sobre dinâmica de corpos rígidos podem ser encontrados em [dS08].

Seja uma cena constituída de n corpos rígidos. O *estado* do i-ésimo corpo rígido no instante de tempo t é definido como:

$$\mathbf{S}_i(t) = \begin{bmatrix} \mathbf{X}_i \\ \mathbf{q}_i \\ \mathbf{V}_i \\ \boldsymbol{\omega}_i \end{bmatrix}, \tag{2.1}$$

onde $\mathbf{X}_i$ é a posição em coordenadas globais do *centro de massa* $\mathcal{C}_i$, $\mathbf{q}_i$ é um quaternion que representa a rotação de um sistema de referência local do corpo (com origem em $\mathcal{C}_i$) em relação ao sistema de coordenadas globais, $\mathbf{V}_i$ e $\boldsymbol{\omega}_i$ são *velocidade linear* e *velocidade angular*, respectivamente, do corpo i no instante t. A principal responsabilidade do motor de física é determinar o estado $\mathbf{S}_i(t + \Delta t)$ de cada corpo rígido i no instante $t + \Delta t$, onde Δt é um *passo de tempo* da simulação, em função do estado $\mathbf{S}_i(t)$ e das forças $\mathbf{F}_i(t)$ e torques $\boldsymbol{\tau}_i(t)$ resultantes que atuam no corpo em t. Essa determinação requer a integração numérica da *equação de movimento* de cada corpo rígido i:

$$\frac{d}{dt}\mathbf{S}_i(t) = \begin{bmatrix} \mathbf{V}_i(t) \\ \frac{1}{2}\mathbf{w}_i\mathbf{q}_i \\ m_i^{-1}\mathbf{F}_i(t) \\ \mathbf{I}_i^{-1}(\boldsymbol{\tau}_i(t) - \boldsymbol{\omega}(t) \times \mathbf{I}_i(t)\boldsymbol{\omega}(t)) \end{bmatrix}, \tag{2.2}$$

onde $\mathbf{w}_i$ é o quaternion $[0, \boldsymbol{\omega}_i]$, m_i representa a massa do corpo e a matriz 3×3 $\mathbf{I}_i$ é o *tensor de inércia* do corpo i. O componente do motor de física responsável pela resolução da equação de movimento e atualização do estado de cada corpo rígido é o solucionador de EDO, descrito na Seção 2.3.

As forças que atuam em um corpo rígido são resultantes da somatória de todas as forças externas (tais como força da gravidade) e forças de *restrições* . Uma restrição elimina um

ou mais *graus de liberdade* (degree of freedom, ou DOF) do corpo ou impedem que dois corpos se interpenetrem. (Um corpo rígido sem restrições possui seis graus de liberdade, três deslocamentos nas direções dos eixos principais de um sistema de coordenadas Cartesianas de referência e três rotações em torno dos eixos do sistema.)

A configuração de um sistema de n corpos em um instante de tempo t é o conjunto das posições $\mathbf{X}_i, 1 \leq i \leq n$, de todos os corpos do sistema em t. O conjunto de todas as possíveis configurações do sistema é chamado de *espaço de configurações*. No entanto, forças de restrições impostas a alguns corpos, impedem que um número de configurações sejam válidas, isto é, nem toda configuração do sistema pode ser atingida.

As restrições de movimento são descritas por uma ou mais condições expressas em função das posições dos corpos e do tempo. Uma restrição expressa por uma condição envolvendo uma igualdade:

$$C(\mathbf{x}(t)) = 0, \tag{2.3}$$

é dita ser *bilateral*, enquanto que uma restrição cuja condição é dada por uma desigualdade:

$$C(\mathbf{x}(t)) \geq 0, \tag{2.4}$$

é chamada *unilateral*.

Restrições são oriundas de *junções* entre dois corpos e também de *contatos* entre corpos.

Contato

Um contato entre dois corpos rígidos introduz uma restrição cuja condição é expressa por uma inequação, ou seja, uma restrição *unilateral*. Uma restrição de um contato k com atrito é formulada utilizando uma matriz *Jacobiana* definida como:

$$\mathbf{J}_C^k = \begin{bmatrix} \mathbf{J}_N^k & \mathbf{J}_{t_1}^k & \mathbf{J}_{t_2}^k \end{bmatrix}^T. \tag{2.5}$$

Dado que o contato ocorreu entre os corpos rígidos de índices i e j, com a normal de contato $\mathbf{N}^k$ (em relação a superfície do corpo rígido i) e $\mathbf{x}_i'^k$ e $\mathbf{x}_j'^k$ os vetores dos respectivos centro de massa $\mathbf{x}_{i,j}$ dos corpos ao ponto de contato, a restrição que impede a penetração dos corpos no ponto de contato k pode ser descrita como:

$$\begin{bmatrix} \mathbf{N}^k & \mathbf{x}_i'^k \times \mathbf{N}^k & -\mathbf{N}^k & -\mathbf{x}_j'^k \times \mathbf{N}^k \end{bmatrix} \begin{bmatrix} \mathbf{v}_i \\ \omega_i \\ \mathbf{v}_j \\ \omega_j \end{bmatrix} = \mathbf{J}_N^k \mathbf{u}^k \geq 0, \tag{2.6}$$

onde $\mathbf{u}^k$ contém as velocidades linear e angular de ambos os corpos do contato.

O princípio do trabalho virtual requer que a força de restrição seja ortogonal à restrição, ou seja:

$$\mathbf{Q}_C^k = \mathbf{J}_N^k \lambda_N^k, \tag{2.7}$$

onde $\mathbf{Q}_C^k$ é a força generalizada (definida pela força normal e o torque da força normal em relação ao centro de massa de cada corpo) e λ_N^k é o multiplicador de Lagrange. Note que $\mathbf{N}^k$ é considerado um vetor unitário, então λ_N^k é a magnitude da força normal $\mathbf{F}_N^k = \lambda_N^k \mathbf{N}^k$. O escalar λ_N^k de todos os K contatos deve ser definido tal que $\mathbf{F}_N^k$ previna a interpenetração dos corpos, seja uma força repulsiva e anule-se no momento da separação dos corpos, o que resulta em uma condição de complementaridade:

$$\mathbf{J}_N^k \mathbf{u}^k \geq 0 \text{ complementar a } \lambda_N^k \geq 0. \tag{2.8}$$

A condição de complementaridade deve ser entendida como:

$$\mathbf{J}_N^k \mathbf{u}^k \lambda_N^k = 0, k = 1, 2, ..., K. \tag{2.9}$$

Aplicando a pirâmide de atrito [Bar94], dado os versores ortogonais t_1 e t_2, os quais podem ser tomados arbitrariamente no plano tangente ao contato para atrito isotrópico, as velocidades tangenciais podem ser escritas como:

$$\begin{bmatrix} \mathbf{J}_{t_1}^k \\ \mathbf{J}_{t_2}^k \end{bmatrix} \mathbf{u}^k = \begin{bmatrix} [\mathbf{t}_1^k & \mathbf{x}_i'^k \times \mathbf{t}_1^k & -\mathbf{t}_1^k & -\mathbf{x}_j'^k \times \mathbf{t}_1^k] \\ [\mathbf{t}_2^k & \mathbf{x}_i'^k \times \mathbf{t}_2^k & -\mathbf{t}_2^k & -\mathbf{x}_j'^k \times \mathbf{t}_2^k] \end{bmatrix} \mathbf{u}^k. \tag{2.10}$$

As forças generalizadas de atrito são:

$$\begin{bmatrix} \mathbf{Q}_{t_1}^k \\ \mathbf{Q}_{t_2}^k \end{bmatrix} = [\mathbf{J}_{t_1}^k \quad \mathbf{J}_{t_2}^k] \begin{bmatrix} \lambda_{t_1}^k \\ \lambda_{t_2}^k \end{bmatrix}, \tag{2.11}$$

e as forças tangenciais:

$$\begin{aligned} \mathbf{F}_{t_1}^k &= \lambda_{t_1}^k \mathbf{t}_1^k, \\ \mathbf{F}_{t_2}^k &= \lambda_{t_2}^k \mathbf{t}_2^k, \end{aligned} \tag{2.12}$$

onde $\lambda_{t_1}^k$ e $\lambda_{t_2}^k$ são multiplicadores de Lagrange à determinar. De acordo com a lei de Coulomb, o atrito dinâmico ocorre quando a velocidade relativa tangencial é diferente de zero; neste caso, a força de atrito atinge seu valor máximo e uma direção oposta ao movimento. Isto significa que:

$$\mathbf{J}_{t_1}^k \mathbf{u}^k > 0 \Rightarrow \lambda_{t_1}^k = -\mu_k \lambda_N^k, \tag{2.13a}$$

$$\mathbf{J}_{t_1}^k \mathbf{u}^k < 0 \Rightarrow \lambda_{t_1}^k = \mu_k \lambda_N^k, \tag{2.13b}$$

$$\mathbf{J}_{t_1}^k \mathbf{u}^k = 0 \Rightarrow \lambda_{t_1}^k < |\mu_k \lambda_N^k|, \tag{2.13c}$$

onde μ_k é o coeficiente de atrito do k-ésimo contato. Este é uma propriedade do material de um corpo rígido, sendo o valor de μ_k uma combinação dos coeficientes de atrito dos corpos em contato. A última restrição, equação (2.13c), é o caso do atrito estático. Restrições similares são impostas para $\mathbf{J}_{t_2}^k$ e $\lambda_{t_2}^k$.

As equações (2.13) são condições mais gerais de complementaridade.

Agrupando-se as forças generalizadas de contato com atrito, tem-se:

$$\mathbf{Q}_C^k = \underbrace{\begin{bmatrix} \mathbf{J}_N^k \\ \mathbf{J}_{t_1}^k \\ \mathbf{J}_{t_2}^k \end{bmatrix}^T}_{(\mathbf{J}_C^k)^T} \underbrace{\begin{bmatrix} \lambda_N^k \\ \lambda_{t_1}^k \\ \lambda_{t_2}^k \end{bmatrix}}_{\lambda_C^k}. \tag{2.14}$$

Ou de maneira geral:

$$\mathbf{Q}_\mathsf{C} = \mathbf{J}_\mathsf{C}^T \lambda_\mathsf{C}. \tag{2.15}$$

Definindo-se $\lambda_{min}^k = [0 \quad -\mu_k \lambda_N^k \quad -\mu_k \lambda_N^k]$ e $\lambda_{max}^k = [\infty \quad \mu_k \lambda_N^k \quad \mu_k \lambda_N^k]$, as restrições do contato k podem ser escritas em uma notação unificada como:

$$\lambda_{C_l}^k = \lambda_{min_l}^k \Rightarrow (\mathbf{J}_C^k \mathbf{u}^k)_l \geq 0, \tag{2.16a}$$

$$\lambda_{C_l}^k = \lambda_{max_l}^k \Rightarrow (\mathbf{J}_C^k \mathbf{u}^k)_l \leq 0, \tag{2.16b}$$

$$\lambda_{min_l}^k < \lambda_{C_l}^k < \lambda_{max_l}^k \Rightarrow (\mathbf{J}_C^k \mathbf{u}^k)_l = 0, \tag{2.16c}$$

onde o índice $l = 1, 2, 3$ indica a linha da matriz ou vetor correspondente ($l = \mathbf{N}, \mathbf{t}_1, \mathbf{t}_2$).

A consideração do impacto no contato será tratada mais adiante.

Junção

Uma junção é uma conexão entre dois corpos ou entre um corpo e um ponto arbitrário, a qual resulta em forças fazendo com que o movimento de um corpo seja relativo ao outro ou ao ponto arbitrário, de acordo com o tipo de junção. Em AS são considerados três tipos de junções: esféricas, de revolução e fixa, mas outros tipos podem ser incorporados, sem grandes dificuldades, ao sistema.

Uma junção esférica força que dois pontos sobre dois corpos diferentes sejam coincidentes, removendo três graus de liberdade de cada corpo. Uma junção de revolução pode ser usada para representar uma dobradiça entre dois corpos: cinco graus de liberdade de cada corpo são removidos, restando uma rotação que se dá em torno do eixo da dobradiça. Uma junção fixa admite que três pontos não colineares sobre dois corpos diferentes sejam coincidentes, removendo seis graus de liberdade de cada corpo. Se as translações e rotações permitidas por estes tipos de junções não possuem limites, como é o caso deste trabalho, então as junções são definidas por restrições *bilaterais*.

Da mesma forma que uma restrição de contato introduz no sistema um conjunto de três restrições *unilaterais*, sendo uma referente à normal no ponto de contato e duas às tangenciais ou atrito, uma junção j introduz m^j restrições *bilaterais* no sistema, onde m^j representa a quantidade de graus de liberdade removidos de ambos os corpos pela junção.

O vetor de velocidades normais às superfícies representadas pelas funções de restrição da junção j é:

$$\mathbf{J}_J^j \mathbf{u}^j = 0. \tag{2.17}$$

As forças de restrição generalizadas da junção j podem ser descritas como:

$$\mathbf{Q}_J^j = \underbrace{\begin{bmatrix} \mathbf{J}_1^j \\ \vdots \\ \mathbf{J}_{m^j}^j \end{bmatrix}^T}_{(\mathbf{J}_J^j)^T} \underbrace{\begin{bmatrix} \lambda_1^j \\ \vdots \\ \lambda_{m^j}^j \end{bmatrix}}_{\lambda_J^j}, \tag{2.18}$$

onde $-\infty < \lambda_J^j < \infty$. Ou de um modo global:

$$\mathbf{Q}_\mathsf{J} = \mathbf{J}_\mathsf{J}^T \lambda_\mathsf{J}. \tag{2.19}$$

Unificando contatos e junções

Sejam N corpos rígidos, K contatos e J junções. O número total de restrições devido as junções e contatos é definido como:

$$R = \underbrace{\sum_{j=1}^{J} m^k}_{\text{junções}} + \underbrace{3K}_{\text{contatos}} \tag{2.20}$$

O vetor de velocidades generalizadas $\mathbf{u} \in \mathbb{R}^{6N}$ é definido:

$$\mathbf{u} = [\mathbf{v}_1, \omega_1, \mathbf{v}_2, \omega_2, \cdots, \mathbf{v}_N, \omega_N]^T. \tag{2.21}$$

A matriz Jacobiana $\mathbf{J} \in \mathbb{R}^{R\times 6N}$ devido as junções e contatos:

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}^1_{C_1} & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{J}^1_{C_N} \\ \vdots & & \vdots & & \vdots & & \vdots \\ \mathbf{J}^k_{C_1} & \cdots & \mathbf{J}^k_{C_i} & \cdots & \mathbf{J}^k_{C_j} & \cdots & \mathbf{J}^k_{C_N} \\ \vdots & & \vdots & & \vdots & & \vdots \\ \mathbf{J}^K_{C_1} & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{J}^K_{C_N} \\ \mathbf{J}^1_{J_1} & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{J}^1_{J_N} \\ \vdots & & \vdots & & \vdots & & \vdots \\ \mathbf{J}^j_{J_1} & \cdots & \mathbf{J}^j_{J_i} & \cdots & \mathbf{J}^j_{J_j} & \cdots & \mathbf{J}^j_{J_N} \\ \vdots & & \vdots & & \vdots & & \vdots \\ \mathbf{J}^J_{J_1} & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{J}^J_{J_N} \end{bmatrix}. \tag{2.22}$$

Note que $\mathbf{J}$ é uma matriz esparsa, cada linha possui somente doze elementos não nulos: $\mathbf{J}_{(C,J)_{i,j}}$, onde i, j são os índices dos corpos pertencentes a junção ou contato.

O vetor de multiplicadores de Lagrange $\lambda \in \mathbb{R}^R$ é agrupado como:

$$\lambda = [\lambda^1_C, \lambda^2_C, \cdots, \lambda^N_C, \lambda^1_J, \lambda^2_J, \cdots, \lambda^J_J]^T. \tag{2.23}$$

Com isso define-se que para qualquer restrição $r, 1 \leq r \leq R$, pode-se escrever:

$$\mathbf{w}^r = \mathbf{J}^r \mathbf{u}^r \tag{2.24}$$

tal que

$$\lambda^r = \lambda^r_{min} \Rightarrow \mathbf{w}^r \geq 0, \tag{2.25a}$$
$$\lambda^r = \lambda^r_{max} \Rightarrow \mathbf{w}^r \leq 0, \tag{2.25b}$$
$$\lambda^r_{min} < \lambda^r < \lambda^r_{max} \Rightarrow \mathbf{w}^r = 0. \tag{2.25c}$$

A Equação de movimento global pode ser descrita como:

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{Q}, \tag{2.26}$$

onde $\mathbf{M} \in \mathbb{R}^{6N\times 6N}$ é a matriz de massa generalizada dada por:

$$\mathbf{M} = \begin{bmatrix} \mathbf{1}m_1 & & & & 0 \\ & \mathbf{I}_1 & & & \\ & & \vdots & & \\ & & & \mathbf{1}m_N & \\ 0 & & & & \mathbf{I}_N \end{bmatrix}, \tag{2.27}$$

onde $\mathbf{1}$ é a matriz identidade, e $\mathbf{Q}$ é o vetor de forças externas generalizadas definido como:

$$\mathbf{Q} = [\mathbf{F}_1, \tau_1 - \omega_1 \times \mathbf{I}_1\omega_1, \cdots, \mathbf{F}_N, \tau_N - \omega_N \times \mathbf{I}_N\omega_N]^T. \tag{2.28}$$

Acrescentando as forças de restrição tanto de contato quanto de junções, vem:

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{Q} + \mathbf{Q}_\mathsf{C} + \mathbf{Q}_\mathsf{J}. \tag{2.29}$$

Substituindo as Equações (2.15) e (2.19), tem:

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{Q} + \mathbf{J}^T_\mathsf{C}\lambda_\mathsf{C} + \mathbf{J}^T_\mathsf{J}\lambda_\mathsf{J} = \mathbf{Q} + \mathbf{J}^T\lambda. \tag{2.30}$$

Na formulação baseada em velocidade, usa-se um passo do método de integração de Euler para aproximar a aceleração do sistema como sendo:

$$\dot{\mathbf{u}} = \frac{\mathbf{u}(t+\Delta t) - \mathbf{u}(t)}{\Delta t}, \tag{2.31}$$

onde $\mathbf{u}(t)$ é a velocidade no início do passo de tempo corrente da simulação, cujo intervalo é Δt, e $\mathbf{u}(t+\Delta t)$ é a velocidade no próximo passo de tempo. Substituindo na Equação (2.30) tem:

$$\mathbf{M}(\mathbf{u}(t+\Delta t) - \mathbf{u}(t)) = \mathbf{Q}\Delta t + \mathbf{J}^T\lambda\Delta t. \tag{2.32}$$

Isolando $\mathbf{u}(t+\Delta t)$ na equação acima obtém-se:

$$\mathbf{u}(t+\Delta t) = \mathbf{u}(t) + \mathbf{M}^{-1}\mathbf{J}^T\lambda\Delta t + \mathbf{M}^{-1}\mathbf{Q}\Delta t. \tag{2.33}$$

Note, na equação acima, que as forças do lado direito são multiplicados por Δt, ou seja, tais termos representam impulsos generalizados ao invés de forças.

Substituindo na Equação (2.24) resulta:

$$\mathbf{w} = \underbrace{\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T}_{\mathbf{A}}\underbrace{\lambda\Delta t}_{\mathbf{x}} + \underbrace{\mathbf{J}(\mathbf{u}(t) + \mathbf{M}^{-1}\mathbf{Q}\Delta t)}_{\mathbf{f}} \geq 0. \tag{2.34}$$

Impacto pode ser adicionado à formulação do contato usando a lei de impacto de Newton, isto é, fornecendo, para o k-ésimo contato um coeficiente de restituição, $0 \leq \varepsilon_k \leq 1$. Considerando o impacto, para esse contato, a Equação (2.6) torna-se:

$$\mathbf{J}_N^k\mathbf{u}^k(t+\Delta t) \geq -\underbrace{\varepsilon_k\mathbf{J}_N^k\mathbf{u}^k(t)}_{b_k}. \tag{2.35}$$

Se $\varepsilon_k = 1$, o choque é perfeitamente elástico, e se $\varepsilon_k = 0$, os corpos permanecerão em contato de repouso após a colisão. Assim como o coeficiente de atrito, e ε_k é uma propriedade associada ao material de um corpo rígido, portanto seu valor pode ser tomado como uma combinação dos coeficientes de restituição dos corpos em contato.

Dessa forma é montado um vetor $\mathbf{b} \in \mathbb{R}^R$, cujo valor referente à restrição normal de um contato k é definido pela Equação (2.35), e o valor referente ao restante, tanto às restrições de contato devidas ao atrito e às junções é zero, como segue:

$$\mathbf{b} = [\underbrace{b_1 \quad 0 \quad 0 \quad b_2 \quad 0 \quad 0 \quad \cdots \quad b_K \quad 0 \quad 0}_{\text{contatos}} \quad \underbrace{0 \quad 0 \quad \cdots \quad 0}_{\text{junções}}]^T. \tag{2.36}$$

O vetor acima deve ser adicionado ao termo representado por $\mathbf{f}$ da Equação (2.34), reescrita:

$$\mathbf{w} = \underbrace{\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T}_{\mathbf{A}}\underbrace{\lambda\Delta t}_{\mathbf{x}} + \underbrace{\mathbf{J}(\mathbf{u}(t) + \mathbf{M}^{-1}\mathbf{Q}\Delta t) + \mathbf{b}}_{\mathbf{f}} \geq 0, \tag{2.37}$$

ou

$$\mathbf{w} = \mathbf{A}\mathbf{x} + \mathbf{f} \geq 0 \quad \text{complementar a} \quad \lambda_{min} \leq \lambda \leq \lambda_{max}. \tag{2.38}$$

A determinação do vetor $\mathbf{x}$, magnitudes dos impulsos generalizados de restrição, satisfazendo a Equação (2.37) é um problema de complementaridade linear (PCL), neste caso, misto ou generalizado, pois nem todas as variáveis estão sujeitas às condições de complementaridade. Os dados de entrada do solucionador de PCL do motor de física, descrito na Seção 2.2, são definidos pela Equação (2.37), e, após solucionado, inicia-se o cálculo do novo estado com a

atualização da velocidade dos corpos da cena em $t + \Delta t$ de acordo com a Equação (2.33). Definindo o vetor de posições generalizadas $\chi \in \mathbb{R}^{7N}$ como:

$$\chi = [\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2, \cdots, \mathbf{x}_N, \mathbf{q}_N]^T, \tag{2.39}$$

onde $\mathbf{q}_i = [s_i, (a_{i_x}, a_{i_y}, a_{i_z})] \in \mathbb{R}^4$ é o quaternion que representa a orientação do corpo i, como comentado no início dessa seção. Definindo:

$$\Theta_i = \frac{1}{2} \begin{bmatrix} -a_{i_x} & -a_{i_y} & -a_{i_z} \\ s_i & a_{i_z} & -a_{i_y} \\ -a_{i_z} & s_i & a_{i_x} \\ a_{i_y} & -a_{i_x} & s_i \end{bmatrix}, \tag{2.40}$$

e $\mathbf{D} \in \mathbb{R}^{7N \times 6N}$ como:

$$\mathbf{D} = \begin{bmatrix} \mathbf{1} & & & 0 \\ & \Theta_1 & & \\ & & \vdots & \\ & & & \Theta_N \\ 0 & & & \mathbf{1} \end{bmatrix}, \tag{2.41}$$

então o restante do cálculo do novo estado é feito com a atualização das posições de todos os corpos rígidos:

$$\chi(t + \Delta t) = \chi(t) + \mathbf{D}\mathbf{u}(t + \Delta t)\Delta t. \tag{2.42}$$

A determinação dos contatos para o motor de física é responsabilidade de um componente do sistema de animação denominado *detector de colisões*. Este componente de AS é descrito detalhadamente no Capítulo 3.

A Figura 2.1 mostra um diagrama de classes UML com as principais classes do motor.

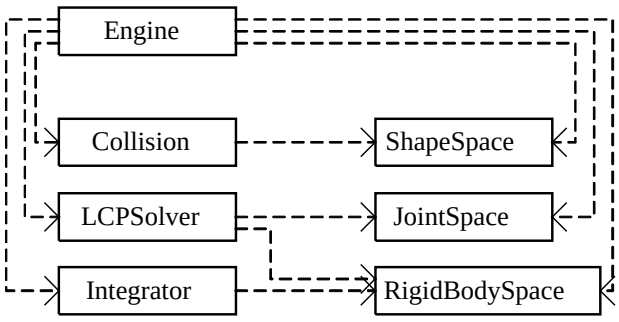


Figura 2.1: Diagrama UML do motor de física.

O sistema AS possui um objeto da classe `Engine` definido globalmente. Esse objeto tem a responsabilidade de organizar as invocações dos métodos responsáveis por cada etapa do motor de física. O construtor da classe cria as listas de dados utilizadas para o processamento e as referenciam com seus atributos: `rbSpace`, objeto da classe `RigidBodySpace`, representa a lista de corpos rígidos; `sSpace`, objeto da classe `ShapeSpace`, a lista de formas; e `jSpace`, objeto da classe `JointSpace`, a lista de junções. Essas classes serão detalhadas durante a apresentação das implementações.

O método `Engine:run()` é o método principal da classe, e tem como função invocar os métodos de ajuste e execução dos objetos referentes ao detector de colisões (`collision`), solucionador de PCL (`lcpSolver`) e solucionador de EDO (`integrator`). Esses atributos são objetos das classes `Collision`, `LCPSolver` e `Integrator`, respectivamente, instanciados pelo construtor. O método é ilustrado no pseudocódigo abaixo:

```
void Engine::run()
{
   collision.setInput(shapes,...);
   contacts = collision.run();

   lcpSolver.setInput(bodies, joints, contacts,...);
   lcpSolver.setProperties(...);
   constraintVel = lcpSolver.run();

   integrator.setInput(bodies);
   integrator.run(constraintVel,...);
}
```

2.2 Solucionador de PCL

Nessa seção é apresentado resumidamente o problema de complementaridade linear (PCL), métodos iterativos para a resolução e por fim a implementação do método empregado.

2.2.1 Definição de PCL

Definição (Problema de Complementaridade Linear). Dada uma matriz $\mathbf{A} \in \mathbb{R}^{n \times n}$, determinar $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{w} \in \mathbb{R}^n$ tal que

$$\mathbf{w} = \mathbf{A}\mathbf{x} - \mathbf{b}, \tag{2.43}$$

e para todos $i = 0, ..., n-1$, uma das três condições abaixo é satizfeita:

$$\mathbf{w}_i \geq 0, \tag{2.44}$$
$$\mathbf{x}_i \geq 0, \tag{2.45}$$
$$\mathbf{x}_i \mathbf{w}_i = 0. \tag{2.46}$$

A Equação (2.46) é a condição de complementaridade, a qual pode ser inicializada como:

$$\mathbf{x}^T \mathbf{w} = 0, \tag{2.47}$$

ou

$$\mathbf{x}^T (\mathbf{A}\mathbf{x} - \mathbf{b}) = 0. \tag{2.48}$$

Esta condição implica que $\mathbf{x}_i$ ou $\mathbf{w}_i$ devem ser zero.

Em um problema mais geral o elemento $\mathbf{x}_i$ pode ser submetido a limites específicos $lo \leq \mathbf{x}_i \leq hi$, e não mais $\mathbf{x}_i \geq 0$. Com essa condição surge o problema de complementaridade linear misto, ou PCLM (PCL *Misto*), o qual é determinar $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{w} \in \mathbb{R}^n$ tal que

$$\mathbf{w} = \mathbf{A}\mathbf{x} - \mathbf{b}, \tag{2.49}$$
$$lo \leq \mathbf{x} \leq hi \tag{2.50}$$

e para todos $i = 0, ..., n-1$, uma das três condições abaixo é satizfeita:

$$\mathbf{x}_i = lo_i, \mathbf{w}_i \geq 0, \tag{2.51}$$
$$\mathbf{x}_i = hi_i, \mathbf{w}_i \leq 0, \tag{2.52}$$
$$lo_i \leq \mathbf{x}_i \leq hi_i, \mathbf{w}_i = 0. \tag{2.53}$$

lo é o vetor de limites inferiores e hi é o vetor de limites superiores de $\mathbf{x}$. A Figura 2.2 ilustra o problema. Note que w_i não é forçado a 0 somente quando x_i está sobre o limite inferior ou superior.

Propriedades de PCLs são discutidas em detalhes em [RWC92], [Mur88] e [Bar92].

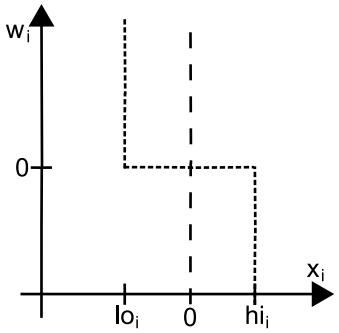


Figura 2.2: Ilustração da variável i do PCLM.

2.2.2 Método de solução

Os métodos de solução de problemas de complementaridade linear são divididos em *métodos de pivotamento* e *métodos iterativos* [Mur88]. Um dos primeiros algoritmos de pivotamento para a solução de PCLs é denominado *algoritmo de Lemke's*, utilizado por Baraff em [Bar91]. [Bar94] apresenta o *algoritmo de Dantzig*, um algoritmo de pivotamento para solucionar PCLs no contexto da dinâmica de corpos rígidos. Esses métodos resultam em uma solução precisa porém com alto custo computacional.

Os métodos iterativos, utilizados para a solução de um conjunto de equações lineares, tais como, *jacobi*, *Gauss-Seidel*, SOR(successive over relaxation), podem ser estendidos para a solução de PCLMs. Estes possuem menor utilização devido a lenta convergência a uma solução de alta precisão. Porém, para o propósito da animação em tempo real o principal interesse não é a precisão. Portanto, é necessário um método que em poucas iterações produza uma solução aproximada da solução exata para o problema. Essa é uma vantagem do método, pois algoritmos de pivotamento não possuem resultados intermediários. É evidente que a solução aproximada obtida irá resultar em pequenos erros na simulação, tais como pequenas penetrações. No entanto, nessas aplicações onde o passo de tempo (Δt) é limitado a um máximo, a ordem do erro será $O(\Delta t)$. [Erl04] e [Cat05] apresentam alguns desses algoritmos, dentre os quais o método SOR foi utilizado como base para a resolução do PCL implementado em AS.

Como resumido na Seção 1.3, os métodos iterativos fornecem uma seqüencia de soluções aproximadas, cada uma das quais obtida a partir das anteriores pela repetição do mesmo processo. A determinação da $(k+1)$-ésima solução é em função da k-ésima:

$$\mathbf{s}^{k+1} = f(\mathbf{s}^k). \tag{2.54}$$

O método inicia com uma solução $\mathbf{s}^0$ e gera a seqüência de soluções $\{\mathbf{s}^0, \mathbf{s}^1, \mathbf{s}^2, ...\}$, uma a cada iteração através da Equação (2.54). Após um número máximo de iterações k_max — e essa é a condição de parada adotada — a solução aproximada $\mathbf{s}^{k\_max}$ é definida e aceita como solução do sistema. Uma outra condição de parada utilizada nos métodos iterativos é a precisão da solução, porém essa é usada para resolver sistemas onde a precisão é fundamental, independente do custo computacional, o que contradiz com o propósito apresentado. O desempenho dos métodos iterativos estão relacionados com a dimensão do sistema e com a quantidade de iterações utilizadas.

Antes da apresentação do método SOR, são introduzidos dois métodos mais simples também utilizados para a resolução de sistemas lineares.

Um sistema linear é definido por:

$$\mathbf{Ax} = \mathbf{b}, \tag{2.55}$$

onde A é denominada matriz dos coeficientes, $\mathbf{b}$ é o vetor do termo independente e $\mathbf{x}$ é o vetor solução. Na primeira impressão, essa equação não é nada parecida com a Equação (2.54), porém com algumas mudanças nota-se a semelhança. Primeiramente a matriz $\mathbf{A}$ é decomposta na soma de três matrizes, sendo a matriz $\mathbf{L}$ triangular inferior, a matriz diagonal $\mathbf{D}$ e a matriz $\mathbf{U}$ triangular superior:

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}. \tag{2.56}$$

Substituindo a Equação (2.56) na Equação (2.55), tem-se:

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{b} \\ (\mathbf{L} + \mathbf{D} + \mathbf{U})\mathbf{x} &= \mathbf{b} \\ D\mathbf{x} &= \mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x} \\ \mathbf{x} &= \mathbf{D}^{-1}(\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}). \end{aligned} \tag{2.57}$$

A partir disso pode-se definir o processo iterativo como:

$$\mathbf{x}^{k+1} = \mathbf{D}^{-1}(\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}^k), \tag{2.58}$$

no qual nota-se que a determinação de $\mathbf{x}^{k+1}$ é em função de $\mathbf{x}^k$, condizendo com a Equação (2.54). O processo iterativo da Equação (2.58) é denominado método de Jacobi-Richardson.

Analisando a Equação (2.58), percebe-se que quando a variável $\mathbf{x}_i^{k+1}$ é computada, já se determinou as variáveis $\mathbf{x}_j^{k+1}$, $j < i$. Portanto, para a computação da variável $\mathbf{x}_i^{k+1}$ pode-se utilizar as variáveis mais recentes, resultando em uma convergência mais rápida para a solução. O novo esquema de atualização das variáveis é denominado método de Gauss-Seidel e o processo iterativo é definido por:

$$\mathbf{x}^{k+1} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{L}\mathbf{x}^{k+1} - \mathbf{U}\mathbf{x}^k). \tag{2.59}$$

Em termos computacionais, a principal diferença entre o método de Jacobi e o método de Gauss-Seidel é que no primeiro todas as variáveis podem ser atualizadas simultaneamente (modo paralelo), pois a computação de cada variável só depende a solução da iteração anterior. No método de Gauss-Seidel, a computação depende da solução anterior e solução atual já obtida (modo seqüencial).

Uma outra diferença do método Gauss-Seidel é a dependência da ordem na qual as variáveis são computadas. A reordenação do sistema entre as iterações altera a taxa de convergência do método, normalmente para melhor. Essas diferenças, geralmente, tornam o método Gauss-Seidel é mais eficiente que o método Jacobi.

O método das sobre-relaxações sucessivas (SOR) é uma variação do método de Gauss-Seidel pela introdução de um fator de relaxação. Essa modificação assume a forma de uma média ponderada entre a iteração prévia e a iteração de Gauss-Seidel calculada para cada componente, isto é:

$$\mathbf{x}_{k+1} = w\mathbf{x}_{k+1}^{Seidel} + (1 - w)\mathbf{x}_k, \tag{2.60}$$

onde, $0 < k < k\_max$, $\mathbf{x}_k^{Seidel}$ representa a iteração de Gauss-Seidel e w é o fator de relaxação. O objetivo principal é escolher um valor para w tal que acelere a convergência das iterações à solução. É aconselhável $0 < w < 2$. Se $w = 1$, o método de SOR se reduz ao método de Gauss-Seidel. Se $w < 1$ tem-se uma sob relaxação e se $w > 1$ tem-se uma sobre relaxação [BF97].

O processo iterativo denominado método SOR é obtido substituindo a Equação (2.59) na Equação (2.60):

$$\mathbf{x}^{k+1} = w(\mathbf{D}^{-1}(\mathbf{b} - \mathbf{L}\mathbf{x}^{k+1} - \mathbf{U}\mathbf{x}^k)) + (1 - w)\mathbf{x}^k, \tag{2.61}$$

ou, com outra organização:

$$\mathbf{x}^{k+1} = \mathbf{x}^k + w(\mathbf{D}^{-1}(\mathbf{b} - \mathbf{L}\mathbf{x}^{k+1} - \mathbf{U}\mathbf{x}^k) - \mathbf{x}^k). \tag{2.62}$$

Detalhes sobre métodos iterativos podem ser encontrados em [BF97].

$$\begin{aligned}
\mathbf{x}_1^{k+1} &= \mathbf{x}_1^k + w(\mathbf{d}_{11}^{-1}(\mathbf{b}_1 - \sum_{i=1}^{<1}\mathbf{l}_{1i}\mathbf{x}_i^{k+1} - \sum_{i=2}^{<n}\mathbf{u}_{1i}\mathbf{x}_i^k) - \mathbf{x}_1^k) \\
\mathbf{x}_2^{k+1} &= \mathbf{x}_2^k + w(\mathbf{d}_{22}^{-1}(\mathbf{b}_2 - \sum_{i=1}^{<2}\mathbf{l}_{2i}\mathbf{x}_i^{k+1} - \sum_{i=2}^{<n}\mathbf{u}_{2i}\mathbf{x}_i^k) - \mathbf{x}_2^k) \\
\mathbf{x}_3^{k+1} &= \mathbf{x}_3^k + w(\mathbf{d}_{33}^{-1}(\mathbf{b}_3 - \sum_{i=1}^{<3}\mathbf{l}_{3i}\mathbf{x}_i^{k+1} - \sum_{i=3}^{<n}\mathbf{u}_{3i}\mathbf{x}_i^k) - \mathbf{x}_3^k) \\
&\vdots \\
\mathbf{x}_n^{k+1} &= \mathbf{x}_n^k + w(\mathbf{d}_{nn}^{-1}(\mathbf{b}_n - \sum_{i=1}^{<n}\mathbf{l}_{ni}\mathbf{x}_i^{k+1} - \sum_{i=n}^{<n}\mathbf{u}_{ni}\mathbf{x}_i^k) - \mathbf{x}_n^k)
\end{aligned} \tag{2.63}$$

$$\begin{aligned}
\mathbf{x}_1^{k+1} &= \mathbf{x}_1^k + \dots \\
\mathbf{x}_2^{k+1} &= \mathbf{x}_2^k + \dots \\
\mathbf{x}_3^{k+1} &= \mathbf{x}_3^k + \dots \\
\mathbf{x}_4^{k+1} &= \mathbf{x}_4^k + \dots \\
\mathbf{x}_5^{k+1} &= \mathbf{x}_5^k + \dots \\
\mathbf{x}_6^{k+1} &= \mathbf{x}_6^k + \dots \\
\mathbf{x}_7^{k+1} &= \mathbf{x}_7^k + \dots \\
&\vdots \\
\mathbf{x}_n^{k+1} &= \mathbf{x}_n^k + \dots
\end{aligned} \tag{2.64}$$

Apresentado o método utilizado para solucionar o problema de complementaridade linear (PCL), a pergunta é: como resolver esse problema através de um método para solucionar sistemas de equações lineares?

Para solucionar esse problema utilizando o SOR basta adicionar uma etapa ao fim da computação de cada variável $\mathbf{x}_i$. Nessa etapa, a variável $\mathbf{x}_i$ é ajustada, tal que se exceder o limite inferior (lo) ou superior(hi) permitidos à ela, é projetada para o limite violado. Com isso o método passa a ser denominado *Projected* SOR. Os valores dos limites (lo e hi) são relacionados com o tipo de restrição (junções e contato) referente a variável do sistema.

Então, dado um PCL $(A, \mathbf{x}, \mathbf{b}, \mathbf{lo}, \mathbf{hi})$, e as configurações do SOR: fator w e um número máximo de iterações (k_max), o pseudocódigo do algoritmo é apresentado:

```
Algoritmo SolveLCP-SOR(A, x, b, lo, hi, w, k_max)
{
  for k=1; k<k_max; k++)
  {
    for i=0; i<n; i++)
    {
```

```
        deltaX = 0;
        for j=0; j<i; j++) deltaX += A(i,j)*x(j);
        for j=i+1; j<n; j++) deltaX += A(i,j)*x(j);
        deltaX = (b(i) - deltaX)/A(i,i);
        x(i) = x(i) + w(deltaX - x(i));
        if (x(i)>hi(i)) x(i) = hi(i);
        if (x(i)<lo(i)) x(i) = lo(i);
      }
    }
  }
```

De acordo com a Equação (2.37), tem-se:[1]

$$\begin{aligned} \mathbf{A} &= \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T + \mathbf{cfm}\Delta t \\ \mathbf{b} &= -\mathbf{J}(\mathbf{u} + \mathbf{M}^{-1}\mathbf{Q}\Delta t) + \mathbf{b}_{erro} \end{aligned} \tag{2.65}$$

onde, $\mathbf{J}$ é a matriz Jacobiana das restrições, $\mathbf{M}^{-1}$ é a inversa da matriz de massa generalizada dos corpos, $\mathbf{u}$ é o vetor de velocidades generalizadas dos corpos e $\mathbf{Q}$ são as forças externas generalizadas dos corpos. **cfm** é um vetor de forças de restrição mista (constraint force mixing) o qual é adicionado à diagonal da matriz do sistema com o objetivo de reduzir erros numéricos gerados pelo sistema e aumentar a estabilidade. O vetor $\mathbf{b}_{erro}$ representa o vetor $-b_{impacto}$ ((2.36)).

A fim de tornar o processo mais eficiente, [Erl04] cita algumas otimizações. Para evitar a multiplicação no segundo termo do lado direito da Equação (2.33), $\mathbf{M^{-1}J}^T\lambda\Delta t$, no processo de atualização da velocidade, o método solucionador de CLP retorna não somente $\mathbf{x} = \lambda\Delta t$, mas também $\mathbf{V'} = \mathbf{M^{-1}J}^T\mathbf{x} = \mathbf{M^{-1}J}^T\lambda\Delta t$ que representa as velocidades devidas às restrições. Isso é realizado utilizando as seguintes pré-computações:

$$\mathbf{d}_i = \frac{w}{\mathbf{A}_{i,i}}, \tag{2.66}$$

$$\mathbf{b}_i = \frac{w\mathbf{b}_i}{\mathbf{A}_{i,i}} = \mathbf{b}_i\mathbf{d}_i, \tag{2.67}$$

$$\mathbf{V'} = \mathbf{M}^{-1}\mathbf{J}^T\mathbf{x}^0, \tag{2.68}$$

$$\mathbf{J'} = \mathbf{M}^{-1}\mathbf{J}^T, \tag{2.69}$$

$$\mathbf{J}_{i,\cdot} = \mathbf{d}_i\mathbf{J}^T_{i,\cdot}, \tag{2.70}$$

onde $J_{i,\cdot}$ representa toda a linha i da matriz $\mathbf{J}$.

O uso desses valores pré-computados requer alterações no laço interno do algoritmo do SOR. Essas alterações são mostradas no pseudocódigo abaixo:

```
⋮
deltaX = b(i) - d(i)*x(i);
for (j=0; j<i; j++) deltaX -= J(i,j)*V'(j);
for (j=i+1; j<n; j++) deltaX -= J(i,j)*V'(j);
x(i) = x(i) + deltaX;
V' += J'(i,.)*deltaX;
⋮
```

Após a finalização do processo o vetor $\mathbf{V'}$ contém as velocidades (linear e angular), decorrentes das restrições, referentes a cada corpo rígido.

[1] A Equação (2.38) considera o vetor termo independente ($\mathbf{b}$) do sistema do lado esquerdo da inequação $A\mathbf{x} + \mathbf{b} \geq 0$. O sistema definido para o método SOR considera o vetor $\mathbf{b}$ do lado direito $A\mathbf{x} \geq \mathbf{b}$. Por isso, o sinal do vetor $\mathbf{b}$ deve ser invertido ao da Equação (2.38).

Uma outra otimização é em relação a multiplicação das matrizes $\mathbf{M}^{-1}$ e $\mathbf{J}^T$. Como comentado na Seção 2.1, cada linha da matriz $\mathbf{J}$ possui doze elementos diferentes de zero, sendo seis (1..6) relativos a um corpo (k) e seis (7..12) relativos ao outro (l). Dessa forma a matriz pode ser reduzida de $R \times 6N$ para $R \times 12$, onde R é o número de restrições e N o número de corpos. Com isso a coluna i da matriz resultante (de dimensão $R \times 12$) dessa multiplicação é computada:

$$\begin{aligned}
(\mathbf{M}^{-1}\mathbf{J}^T)_{1,i} &= \tfrac{1}{m_k}\mathbf{J}^T_{1,i}\ , \\
(\mathbf{M}^{-1}\mathbf{J}^T)_{2,i} &= \tfrac{1}{m_k}\mathbf{J}^T_{2,i}\ , \\
(\mathbf{M}^{-1}\mathbf{J}^T)_{3,i} &= \tfrac{1}{m_k}\mathbf{J}^T_{3,i}\ , \\
(\mathbf{M}^{-1}\mathbf{J}^T)_{4..6,i} &= I_k^{-1}\mathbf{J}^T_{4..6,i}\ , \\
(\mathbf{M}^{-1}\mathbf{J}^T)_{7,i} &= \tfrac{1}{m_l}\mathbf{J}^T_{7,i}\ , \\
(\mathbf{M}^{-1}\mathbf{J}^T)_{8,i} &= \tfrac{1}{m_l}\mathbf{J}^T_{8,i}\ , \\
(\mathbf{M}^{-1}\mathbf{J}^T)_{9,i} &= \tfrac{1}{m_l}\mathbf{J}^T_{9,i}\ , \\
(\mathbf{M}^{-1}\mathbf{J}^T)_{10..12,i} &= I_l^{-1}\mathbf{J}^T_{10..12,i}.
\end{aligned} \tag{2.71}$$

Onde m_k e m_l são as respectivas massas, e I_l^{-1} e I_k^{-1} as respectivas matrizes das inversas do tensor de inertia dos corpos k e l.

Calcular o elemento $A(i,i)$ também torna-se mais eficiente. Este corresponde ao produto vetorial entre a i-ésima linha da matriz $\mathbf{J}$ e a i-ésima coluna da matriz $(\mathbf{M}^{-1}\mathbf{J}^T)$, da seguinte forma:

$$\begin{aligned}
\mathrm{A}_{i,i} = \mathbf{J}_{i,\cdot}(\mathbf{M}^{-1}\mathbf{J}^T)_{\cdot,i} = {} & \mathbf{J}_{i,1}(\mathbf{M}^{-1}\mathbf{J}^T)_{1,i} + \mathbf{J}_{i,2}(\mathbf{M}^{-1}\mathbf{J}^T)_{2,i} + \\
& \mathbf{J}_{i,3}(\mathbf{M}^{-1}\mathbf{J}^T)_{3,i} + \mathbf{J}_{i,4}(\mathbf{M}^{-1}\mathbf{J}^T)_{4,i} + \\
& \mathbf{J}_{i,5}(\mathbf{M}^{-1}\mathbf{J}^T)_{5,i} + \mathbf{J}_{i,6}(\mathbf{M}^{-1}\mathbf{J}^T)_{6,i} + \\
& \mathbf{J}_{i,7}(\mathbf{M}^{-1}\mathbf{J}^T)_{7,i} + \mathbf{J}_{i,8}(\mathbf{M}^{-1}\mathbf{J}^T)_{8,i} + \\
& \mathbf{J}_{i,9}(\mathbf{M}^{-1}\mathbf{J}^T)_{9,i} + \mathbf{J}_{i,10}(\mathbf{M}^{-1}\mathbf{J}^T)_{10,i} + \\
& \mathbf{J}_{i,11}(\mathbf{M}^{-1}\mathbf{J}^T)_{11,i} + \mathbf{J}_{i,12}(\mathbf{M}^{-1}\mathbf{J}^T)_{12,i}.
\end{aligned} \tag{2.72}$$

2.2.3 Implementação

O solucionador de PCL é implementado através da classe `LCPSolver`. O diagrama da Figura 2.3 apresentada as classes utilizadas pelo solucionador.

Para iniciar a resolução do PCL é necessário, dado a lista de junções da cena (incluindo os contatos, pois estes são considerados uma junção temporária), criar a lista de restrições, a qual será necessária para a montagem do sistema a ser solucionado. Cada junção adiciona à lista uma quantidade de restrições, a qual não excede seis, de acordo com o seu tipo. No caso do contato, é adicionado à lista uma restrição relacionada com a normal e duas restrições relacionadas com o atrito. Como a quantidade de restrições é variável em cada junção, criou-se uma lista de índices que armazena o índice da primeira restrição de cada junção na lista de restrições [2].

É assumido, neste trabalho, que somente a montagem das restrições oriundas dos contatos é responsabilidade dessa classe. Portanto, as informações recebidas por essa classe são a lista

[2] A lista de índices foi criada para servir de base para uma possível reordenação do sistema, a qual altere somente a ordem entre as junções e não entre as restrições da mesma junção. Segundo [Erl04], uma reordenação do sistema aumenta sua eficiência, porém as restrições oriundas de uma mesma junção não devem ser reordenadas, pois possuem precedências, por exemplo em uma junção de contato a restrição referente à normal deve solucionada antes das restrições referentes ao atrito.

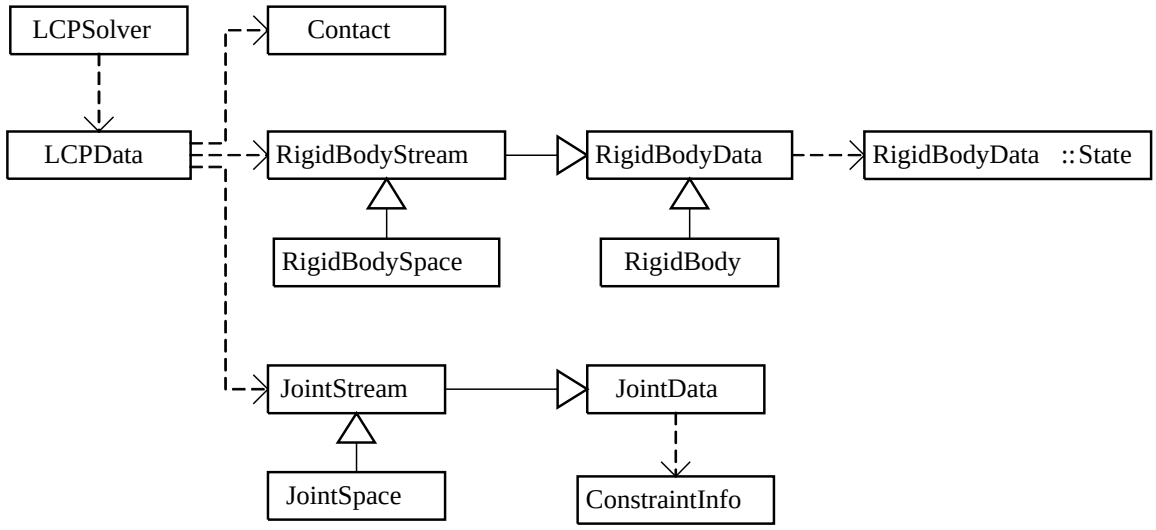


Figura 2.3: Diagrama UML do solucionador de PCL.

de corpos rígidos da cena, lista de contatos e a lista de restrições e índices das junções (exceto contato) previamente montadas.

As informações de uma restrição são representadas computacionalmente por uma estrutura denominada `ConstraintInfo`. Essa estrutura é composta por: dois índices dos corpos rígidos envolvidos na restrição (`body 1` e `body 2`), um vetor `J`, representando uma linha da matriz Jacobiana, contendo doze elementos, sendo seis para cada corpo, e ainda quatro valores ponto flutuante que consistem em pré-computação do lado direito da equação de restrição (`c`), força de restrição mista (`cfm`) e os limites inferior (`lo`) e superior (`hi`) da restrição, respectivamente. [dS08] apresenta detalhes da criação das restrições.

A lista de restrições e índices fazem parte de um repositório representado pela classe `JointStream`, a qual deriva de `JointData`. `JointData`, Figura 2.4, contém apenas os ponteiros que indicam o início da lista, já `JointStream` possui uma interface implementada para a manipulação dessas listas. A classe `JointSpace` deriva de `JointStream` e representa um espaço de junções na memória do computador.

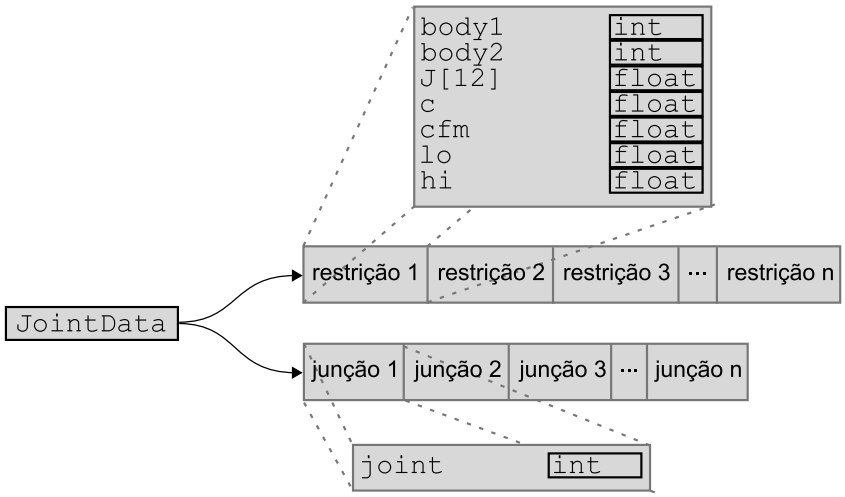


Figura 2.4: Conteúdo da lista de restrições e índices de junções.

A lista de corpos rígidos é representada pela classe `RigidBodyStream`, esta é derivada da classe `RigidBodyData`. `RigidBodyData`, Figura 2.5, possui um ponteiro para cada atributo de um corpo rígido, tais como estado (`x, q, v, w`), inversa da massa ($\mathtt{m}^{-1}$), inversa do tensor de inércia ($\mathtt{I}^{-1}$), força (`F`) e torque (`T`). Dessa maneira, `RigidBodyStream` contém uma lista para cada atributo, tal que, os atributos do corpo i estão armazenados na i-ésima posição das respectivas listas. Esses atributos podem ser acessados diretamente pelo corpo rígido que é representado pela classe `RigidBody`, a qual deriva de `RigidBodyData`. Na prática, o atributo `F` de um corpo rígido i aponta para a i-ésima posição da lista indicada por `F` de um objeto da classe `RigidBodyStream`. A classe `RigidBodySpace` deriva de `RigidBodyStream` e representa um espaço de corpos rígidos na memória do computador.

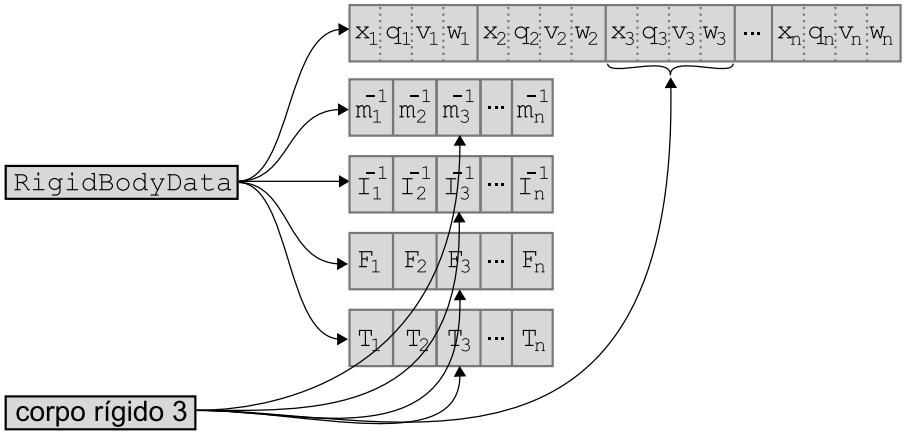


Figura 2.5: Conteúdo da lista de corpos rígidos e conteúdo de um corpo rígido.

A lista de contatos (`contacts`) é fornecida pelo detector de colisões e consiste em um vetor de objetos da classe `Contact`. Esses objetos armazenam as informações da geometria e superfície do contato.

Todos essas informações, tais como listas, contadores, parâmetros e vetores do método SOR, utilizadas pelo solucionador de PCL, são armazenadas em um atributo da classe `LCPData` declarado como `LCPSolver::data`. O objetivo é reduzir a quantidade parâmetros na chamada de métodos que não possuem acesso à classe, tais como métodos estáticos. Isso só tornou-se necessário devido a implementação em GPU compartilhar o uso desses métodos Seção 5.3.

O abastecimento das informações contidas em `data` é responsabilidade de dois métodos. O primeiro, `LCPSolver::setInput(rbStream, jStream, contacts, numberOfContacts)`, ajusta os dados referentes às entradas do solucionador, onde `rbStream` é a lista de corpos rígidos, `jStream` a lista de restrições e índices e `contacts` a lista de contatos contendo `numberOfContacts` contatos. Já o segundo, `LCPSolver::setInput(w, k_max, dt, g)`, ajusta os dados referentes ao método SOR, onde `w` é fator de relaxação e k_max é o número de iterações, e referentes ao sistema, onde `dt` é o passo de tempo da simulação e `g` é o vetor da aceleração da gravidade considerada na cena.

A montagem das restrições geradas pelos pontos de contato é implementada pelo método `LCPSolver::assemblyContactsInfo()` o qual percorre a lista de contatos criando e adicionando as restrições oriundas destes à lista recebida como entrada. As restrições (três) de cada contato são criadas através do método `computeContactInfo(c)`, onde `c` representa o contato:

```
void LCPSolver::assemblyContactsInfo()
{
   for each(Contact c in contacts)
```

```
      computeContactInfo(c, data);
  }
```

Uma vez montada a lista de restrições e índices das junções, inicia-se a resolução do PCL, através dos seguinte passos:

Pré-processamento consiste na aplicação das forças externas (forças da gravidade e aplicadas pelo usuário) à todos os corpos rígidos da cena. Além disso, as acelerações linear e angular de cada corpo rígido são pré-processadas para serem utilizadas posteriormente. O passo é implementado no método `LCPSolver::preProcess()`, cujo pseudocódigo é apresentado a seguir:

```
void LCPSolver::preProcess()
{
  for each(RigidBoby rb in rbStream)
    computePreProcess(rb, data);
}
```

Inicialização nesse passo são pré-computados os vetores **d** e **b**, representados pelas Equações (2.66) e (2.67), respectivamente, os quais possuem dimensão igual a quantidade de restrições da lista e a matriz $\mathbf{J}' = \mathbf{M}^{-1}\mathbf{J}^T$, da Equação (2.69). Ainda nesse passo, o vetor **J** de cada restrição é atualizado de acordo com a Equação (2.70). O passo é implementado no método `LCPSolver::SORLCPInit()`, cujo pseudocódigo é mostrado abaixo:

```
void LCPSolver::SORLCPInit()
{
  for each (ConstraintInfo constraint in jStream)
    computeSORLCPInit(constraint, data);
}
```

Processamento a solução do PCL através do método SOR otimizado descrito anteriormente é executada nesse passo. O método `LCPSolver::SORLCPSolver()` percorre todas junções da lista de junções k_max vezes. O método `LCPSolver::computeSORLCPSolver(joint)` executa o algoritmo para todas as restrições oriundas da junção `joint`, de acordo com o seguinte pseudocódigo:

```
void LCPSolver::SORLCPSolver()
{
  for each (Joint joint in jStream)
    computeSORLCPSolve(joint, data);
}
```

O método `LCPSolver::run()` é o principal da classe. Este é responsável pela invocação dos métodos referente às etapas do solucionador, conforme o código abaixo:

```
void LCPSolver::run()
{
  assemblyContactsInfo();
  preProcess();
  SORLCPInit();
  SORLCPSolver();
}
```

Os métodos `computePreProcess()`, `computeSORLCPInit()`, `computeContactInfo()` e `computeSORLCPSolve()` são declarados estáticos, por isso não possuem acesso aos atributos da classe. Então recebem o atributo `data` como parâmetro contendo as informações necessárias do PCL.

Uma vez solucionado o PCL e determinadas as forças, estas devem ser aplicadas aos respectivos corpos rígidos. Isso é feito juntamente com a integração da equação do movimento, descrita na próxima seção.

2.3 Solucionador de EDO

Para determinar o novo estado $\mathbf{S}_i(t + \Delta t)$ de cada corpo rígido i deve-se integrar a equação do movimento eq.motionequation. Esta é uma equação diferencial ordinária, cuja integração deve ser computada numericamente.

Métodos de integração numérica com foco especial em simulação de corpos rígidos são apresentados em [Ebe04]. O método de integração numérica mais simples e intuitivo é o método de Euler. Apesar da solução obtida pelo método conter erros numéricos devido a simplicidade, satisfazem a precisão necessária para a maioria dos simuladores de corpos rígidos.

O solucionador de EDO é implementado através da classe `Integrator` como ilustrado no diagrama da Figura 2.1.

O método `Integrator::setInput(rbStream)` é responsável por fornecer os dados de entrada ao solucionador, onde `rbStream` é a lista de corpos rígidos.

No método `Integrator::run(constraintVel, dt)` a lista de corpos rígidos é percorrida conforme o pseudocódigo abaixo. O método toma como argumento `constraintVel` contendo as velocidades (linear e angular), devidas às restrições, de cada corpo rígido, e `dt` que é o passo de tempo da simulação. Para cada corpo rígido o método `RigidBody::update()` é invocado a fim de realizar a integração numérica da equação de movimento.

```
void Integrator::run(constraintVel, dt)
{
  for each (RigidBody rb in rbStream)
    rb.update(constraintVel[rb], dt);
}
```

O pseudocódigo da integração numérica da equação de movimento, através do método de Euler, Equações (2.33) e (2.42), calculada para cada corpo, é mostrado a seguir. Os atributos `state`, `invMass` e `invInertia` são o estado, a inversa da massa e a inversa do tensor de inércia do corpo rígido, respectivamente.

```
void RigidBody::update(cVel, dt)
{
  state.V += cVel.linear + (invMass * F) * dt;
  state.w += cVel.angular + (invInertia * T) * dt;
  state.X += state.V * dt;
  state.q += Quat(state->w) * state->q * (0.5 * dt);
}
```

No código acima, a expressão `Quat(v)` resulta no quaternion [0,v].

Esta é o última etapa referente ao motor de física. A partir desta, o novo estado $\mathbf{S}_i(t+\Delta t)$ de cada corpo rígido i estão atualizados, e seus respectivos atores poderão ser renderizados.

2.4 Comentários finais

Neste capítulo apresentou-se um resumo dos componentes do motor de física de AS. Este é responsável pela determinação do estado de cada corpo rígido de uma cena ao longo da animação. O solucionador de PCL é o componente do motor que toma como entrada todas as restrições sobre os corpos rígidos em um instante t, sejam oriundas das junções ou de contatos entre os corpos. Então o componente determina simultaneamente as forças que satisfazem todas as restrições a fim de manter os corpos unidos pelas junções e evitar que os corpos em contato se interpenetrem. Em CPU, o solucionador de PCL é um objeto da classe `LCPSolver`. Determinadas as forças de restrições, a equação de movimento é integrada numericamente através do componente solucionador de EDO, o qual implementa para isso o método de Euler. Em CPU, o solucionador de EDO é um objeto da classe `Integrator`. O resultado da integração é o novo estado de cada corpo rígido.

O detector de colisões é o componente cuja função é determinar, e fornecer ao PCL, os pontos de contatos a partir da lista de formas dos atores da cena. Em CPU, o detector de colisões é um objeto da classe `Collision`.

CAPÍTULO 3

Detecção de Colisão

3.1 Introdução

Neste capítulo é introduzido o detector de colisões, componente do sistema de animação responsável por determinar os contatos e fornecê-los ao motor de física.

O detector de colisões é um componente fundamental na simulação física para a interatividade e demonstração do funcionamento do sistema em tempo real. São decorrentes de colisões a maioria das forças que atuarão sobre os objetos na simulação. Ele é responsável, principalmente, por determinar se dois objetos colidem e em qual posição, e isso deve ser realizado mantendo taxas interativas aceitáveis para manter o realismo da simulação. No contexto desse trabalho, detecção de colisões refere-se ao processo de comparação de pares de objetos, descobrindo se estão colidindo e calculando os exatos pontos de contato.

Uma forma de determinar quais objetos estão em contato é testando todos os objetos entre si com um algoritmo de *força bruta*. Obviamente esta técnica torna-se inviável quando se trata de aplicações em tempo real. Para contatos entre n objetos, o número de testes requeridos é da ordem de $O(n^2)$. O número de *pares de colisão* é $n(n-1)/2$ (um par de objetos que requer um teste de colisão). O número de testes é claramente muito alto para sistemas em tempo real. A situação se agrava ainda mais se ao invés de considerar n como o número de objetos, considerá-lo como o número de primitivos que descrevem os objetos (triângulos na maioria das vezes). Pode-se ter em uma cena milhares de triângulos agrupados em objetos, tornando impossível determinar, dessa forma, o conjunto de pares de colisão em tempo real.

Por outro lado, na maioria dos casos, muitos desses objetos não possuem nenhuma chance de colidir, pois podem, por exemplo, estar muito distantes uns dos outros. Por isso, para evitar testes desnecessários, a detecção de colisão é dividida em duas fases denominadas fase geral (*broad-phase*) e fase exata (*narrow-phase*). É trabalho da fase geral reduzir o número de interseções requeridas e isso é feito através de algumas técnicas citadas na Seção 3.2. A fase exata deve identificar os pares de colisão que se interceptam e calcular os pontos de interseção. Esta é tratada na Seção 3.3. A proposta de utilizar duas fases no processo de detecção de colisão é bastante eficiente quando muitos dos objetos não estão colidindo, o que acontece na prática, pois diversos pares de objetos serão descartados durante a fase geral.

3.2 Fase geral

Na fase geral, os testes de colisão são conservativos — usualmente baseados somente em técnicas com volumes envolventes (VE) — porém rápidos a fim de descartar eficientemente pares de objetos que não colidem. A idéia dos VEs é envolver cada objeto da cena com um primitivo geométrico (exemplos: esfera, caixa, cilindro, etc). Uma condição importante é sempre tentar minimizar o volume dos volumes envolventes. Ao minimizar-se o volume, diminui-se a chance de dois volumes envolventes estarem colidindo e seus objetos internos não. Se dois VEs se interceptam então o respectivo par de objetos é adicionado a um conjunto de pares que serão testados durante a fase exata. Se o par de VEs não se intercepta então os objetos não se interceptam. Calcular se dois objetos complexos estão se interceptando possui um custo computacional muito maior que calcular o par de VEs, o que explica a eficiência dessa fase. Os tipos de VEs mais comuns são esferas, caixas alinhadas aos eixos do sistema de coordenadas utilizado (AABB) e caixas orientadas (OBB). O algoritmo utilizado na fase geral da detecção de colisões desse trabalho considera esferas envolventes para seus testes.

Esferas envolventes

O uso de esferas envolventes como VE é muito comum pois seu teste de interseção com outras esferas envolventes é bastante simples e rápido. Além disso, sua dimensão (raio) não necessita de alterações caso o objeto mude sua orientação. O único problema das esferas envolventes é o fato que nem sempre se consegue um ajuste adequado. Por exemplo, no caso dos primitivos básicos, o ajuste da esfera envolvente para um cubo é mais eficiente que para uma cápsula, Figura 3.1.

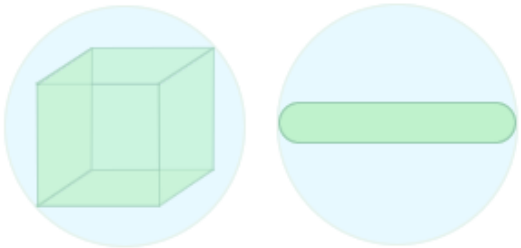


Figura 3.1: Esferas envolventes: cubo e cápsula.

O cálculo de interseção é baseado no conceito de que esferas não se sobrepõem se a distância entre seus centros C_1 e C_2 é maior que a soma de seus raios r_1 e r_2:

$$\|C_1 - C_2\| > r_1 + r_2, \tag{3.1}$$

ou

$$(C_1 - C_2) \cdot (C_1 - C_2) > (r_1 + r_2)^2. \tag{3.2}$$

A Equação (3.2) é uma otimização da Equação (3.1) substituindo o cálculo da raiz quadrada por uma multiplicação, aumentando assim a eficiência do cálculo.

Apesar da simplicidade do cálculo de interseção da esfera envolvente, a implementação por força bruta da fase geral realiza $n(n-1)/2$ testes de colisão, onde n é o número de objetos, ou seja, na ordem de $O(n^2)$ testes, como citado anteriormente. Algoritmos alternativos, tais como *sort and sweep* e *subdivisão espacial*, os quais levam em consideração a coerência espacial

entre os objetos, são mais eficientes pois desconsideram testes entre objetos distantes, com isso atingem, na média, complexidade $O(n \log n)$ [Gra07].

Sort and Sweep

Uma das técnicas para tornar a fase geral rápida e eficiente é o algoritmo *sort and prune* [BW97]. No algoritmo, o volume envolvente (normalmente esfera) de cada objeto i é projetado em um dos eixos x, y e z, definindo um intervalo de colisão unidimensional $[c_i, f_i]$ do objeto ao longo desse eixo, onde c_i marca o começo do intervalo de colisão e f_i marca seu final. Dois objetos cujos intervalos de colisão não se sobrepõem não colidem.

As marcas dos n objetos são inseridas em uma lista com $2n$ entradas. Em seguida, essa lista é ordenada em ordem ascendente. Por fim, a lista é percorrida do começo ao fim. Quando uma marca c_i é descoberta na lista, o objeto i é adicionado a uma lista de objetos ativos. Assim que uma marca f_i é descoberta, o objeto i é removido da lista de objetos ativos. Os testes de colisão são aplicados somente entre o objeto i e todos os objetos pertencentes à lista de objetos ativos no momento em que o objeto i é adicionado à lista de objetos ativos. Um exemplo com três objetos é ilustrado na Figura 3.2.

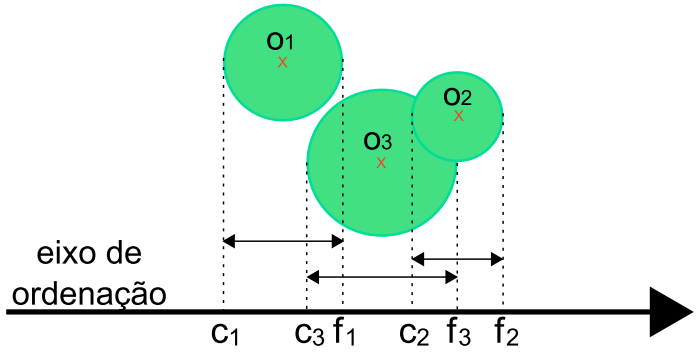


Figura 3.2: Algoritmo *Sort and Sweep* para três objetos.

Sort and Sweep não é complexo de se implementar e é um bom ponto de partida para um algoritmo de fase geral de detecção de colisão. Adicionalmente, devido à coerência espacial entre um quadro da simulação e o próximo, pode-se utilizar um algoritmo de ordenação de complexidade $O(n^2)$ tal como *ordenação por inserção* de uma maneira eficiente, pois este tem seu desempenho melhorado para $O(n)$ quando aplicado a listas semi-ordenadas.

Subdivisão espacial

Outro método utilizado na fase geral é a *subdivisão espacial*, a qual particiona uniformemente o espaço em uma grade, ou *grid*, tal que todas as células do *grid* possuem dimensão igual, no mínimo, ao tamanho do maior objeto. Cada célula possui uma lista contendo todos os objetos cujos centróides a ela pertencem. Um teste de colisão entre dois objetos é aplicado somente se eles pertencem à mesma célula ou então a duas células adjacentes.

Alternativamente — e esse é o método considerado no restante da seção — pode-se atribuir a cada célula uma lista de todos os objetos cujos volumes envolventes a interceptam. Nesse caso um objeto pode aparecer no máximo em 2^d células, onde d é a dimensão da subdivisão espacial (por exemplo, $d = 3$ para uma cena 3D), como ilustrado na Figura 3.3.

Um teste de colisão é aplicado entre dois objetos somente se eles aparecem na mesma célula e pelo menos um deles possui seu centróide na célula. Por exemplo, na Figura 3.4, um teste de colisão é aplicado entre os objetos O_1 e O_2, pois ambos possuem seus centróides

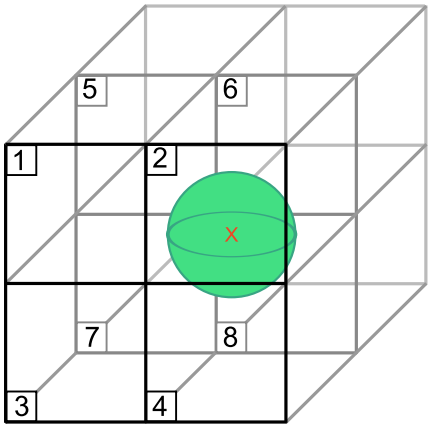


Figura 3.3: Objeto no espaço 3D intersectando as $2^3 = 8$ possíveis células.

pertencentes à célula 1, e entre os objetos O_2 e O_3, pois ambos aparecem na célula 5 e O_3 possui seu centróide na célula 5. Porém nenhum teste de colisão é aplicado entre os objetos O_2 e O_4, pois ambos aparecem na célula 2, mas nenhum dos centróides está na célula 2.

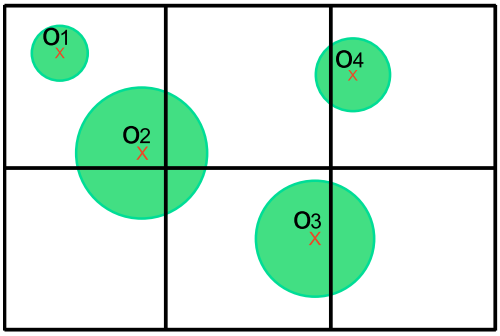


Figura 3.4: Exemplo de subdivisão espacial 2D com quatro objetos.

A implementação mais simples da divisão espacial cria uma lista de identificadores de objetos com uma *hashing* dos identificadores das células nas quais eles residem, ordena essa lista pelo ID da célula, e em seguida percorre a lista identificando grupos de células iguais, aplicando testes de colisão entre todos os objetos que compartilham a mesma célula.

A necessidade da célula ser no mínimo tão grande quanto o volume envolvente do maior objeto da cena pode causar computação desnecessária no caso de uma grande disparidade na dimensão dos objetos. Um exemplo poderia ser um grande planeta rodeado por milhares de pequenos asteróides. Para tratar isso [Mir96] utiliza *grids* hierárquicos, não abordado aqui.

Subdivisão espacial foi o método utilizado na implementação da fase geral da detecção de colisão desse trabalho. Além de uma implementação seqüencial descrita na Seção 3.4, há também uma implementação em paralelo para execução em GPU descrita com mais detalhes na Seção 5.2.

Existem alguns trabalhos relacionados à detecção de colisão em paralelo. Em [LK02] é sugerido um algoritmo de detecção de colisão baseado em particionamento espacial. O algoritmo não implementa hierarquia de volumes envolventes, mas sim volumes de dimensão constante, o que afeta relativamente o desempenho. Porém, para corpos rígidos o tempo gasto é recompensado com a construção das estruturas, uma vez que estes não sofrem alterações em seus volumes envolventes durante a simulação.

Em [AS01] são apresentados algumas versões de algoritmos de detecção de colisão em

paralelo. Alguns problemas e soluções com o balanceamento de carga e sugestões dos autores sobre técnicas de redução do *overhead* de comunicação são abordados.

[GW07] sugerem um algoritmo paralelo para detecção de colisão cuja idéia principal é manter a escalabilidade e portabilidade considerando ainda o princípio da localidade e balanceamento de carga do sistema. O algoritmo apresentado, segundo os autores, possui a vantagem de poder ser implementado em diferentes modelos de máquinas paralelas através de simples adaptações, diferenciando-se dos demais destinados a modelos específicos de máquinas paralelas.

O algoritmo da fase geral implementado nesse trabalho, comentado na Seção 5.2, foi basesdo de [Gra07] e foi um dos únicos encontrados para o propósito de implementação em GPU utilizando CUDA.

3.3 Fase exata

Na fase exata, os testes de colisão são precisos — normalmente são computados os pontos de contato, profundidades e normais. Deste modo, essa tarefa é mais custosa, porém é aplicada somente aos pares do conjunto de possíveis colisões. O custo computacional considerado nessa fase é relativo às formas dos objetos consideradas na colisão. No presente trabalho são consideradas somente formas primitivas como esferas, caixas, cápsulas, e planos para fins de detecção de colisões, pois isso simplifica bastante os cálculos dessa fase. Por outro lado, quando se usa, por exemplo, malhas de triângulos para a definição dos objetos, um cuidado maior deve ser tomado. No caso das malhas ao invés de testar todos os triângulos de um par de objetos, algumas técnicas são utilizadas para que os testes sejam executados de forma reduzida e organizada, resultando em uma forma eficiente para a determinação dos contatos.

Utilizar somente formas primitivas não significa a ausência de suporte a objetos com forma definida através de malha de triângulos em uma cena. Isso é possível aproximando a forma do objeto com uma combinação de formas primitivas, a qual será considerada pelo detector de colisões. A forma mais precisa (definida através de malha) do objeto será considerada somente para renderização. [dS08] apresenta detalhes dessa implementação considerada em AS.

3.4 Implementação em CPU

A implementação da fase geral desse trabalho considera o algoritmo de subdivisão espacial introduzido na Seção 3.2. O componente detector de colisões de AS é implementado de acordo com o diagrama da Figura 3.5.

O AS permite a criação de corpos compostos por uma ou várias formas, ou *shapes*. Essas formas são tratadas individualmente na detecção de colisão, portanto o principal dado de entrada do detector de colisões é a lista de formas. Um objeto cuja classe deriva da classe abstrata `Shape` representa uma forma. Os atributos declarados nesta classe são `typeId`, que representa o tipo da forma (esfera, caixa, cápsula ou plano), `bodyId`, o índice do corpo ao qual a forma pertence, e `boundRadius`, o raio da esfera envolvente da forma. O centro da esfera envolvente é dado pelo atributo `globalPose`, o qual armazena a posição e orientação em relação às coordenadas globais da forma.

A classe `ShapeStream` representa um repositório de formas contendo uma interface para manipulação das mesmas, Figura 3.6. Essa classe deriva de `ShapeData` a qual declara somente os ponteiros da lista de formas e de *poses* globais. O motivo dessa separação será

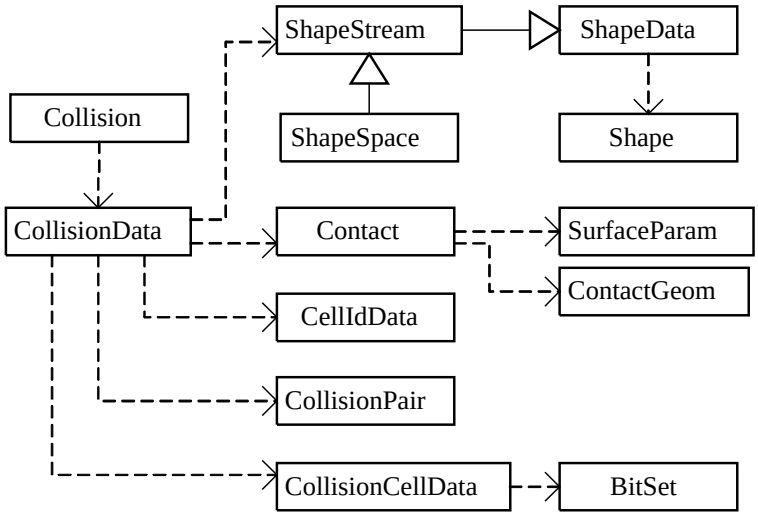


Figura 3.5: Diagrama UML do detector de colisão.

explicado durante a detecção de colisão em CUDA Seção 5.2. A classe `ShapeSpace` deriva de `ShapeStream` e representa um espaço de formas na memória do computador.

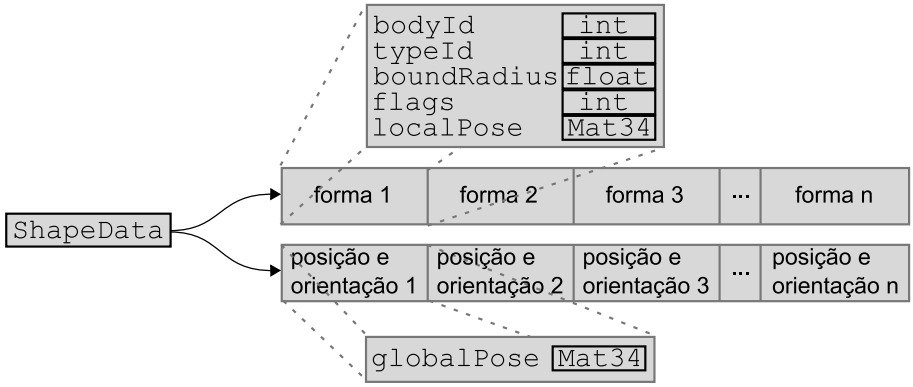


Figura 3.6: Conteúdo da lista de formas.

A classe responsável pelo detector de colisões é `Collision`. Esta possui o método `Collision::run()` o qual simplesmente invoca os métodos `Collision::runBroad()` e `Collision::runNarrow()` responsáveis pelos processos da fase geral e fase exata, respectivamente. O método `Collision::setInput()` é utilizado para ajustar a lista de formas, e alguns outros parâmetros detalhados a seguir. Todos os dados necessários referentes ao detector de colisões estão encapsulados no atributo `Collision::data` do tipo `CollisionData`. Isso foi feito para melhorar a eficiência do detector de colisão em CUDA.

Fase geral

Antes de inicializar os detalhes da implementação da fase geral do detector de colisões, algumas considerações são feitas para auxiliar a compreensão do algoritmo. Visto o algoritmo de subdivisão espacial descrito na Seção 3.2, umas das considerações adotadas é a dimensão da célula ser maior que o maior volume (esfera é considerado nesse trabalho) envolvente das formas. Uma outra consideração é cada célula possuir um rótulo, ou *label*, de acordo com sua posição na grade, para evitar a ocorrência de problemas de duplicação de pares de colisão. Esse rótulo é um número tal que células de mesmo rótulo devem estar separadas por uma

célula. São utilizados 2^d rótulos diferentes para células em um espaço de d dimensões. Em um espaço 3D, oito rótulos (1 a 8) são necessários para rotular as células, como ilustrado na Figura 3.7.

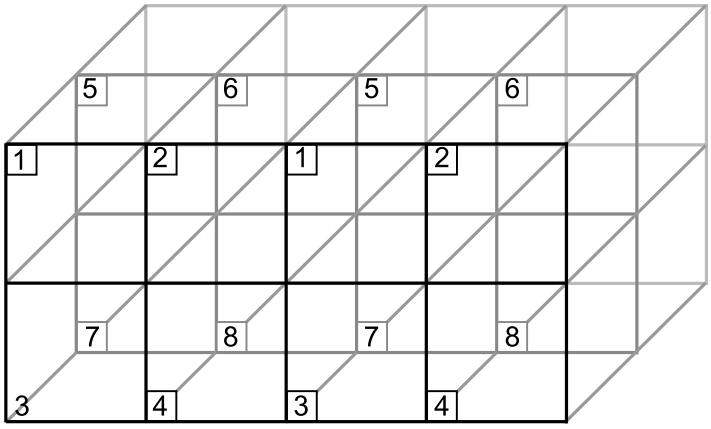


Figura 3.7: Rótulos de células no espaço 3D.

A fase geral do detector de colisões considera somente os volumes envolventes das formas em seus testes. Por isso, para simplificar, nesta seção a palavra objeto refere-se à esfera envolvente de uma forma de um ator.

O problema de duplicação de pares de colisão ocorre se ambos os objetos pertencerem a um conjunto de células, porém seus centróides se localizarem em células (rótulos) diferentes. A solução para isso é feita da seguinte maneira:

1. É associado a cada forma um conjunto de $d + 2^d$ bits de controle; d bits armazenam o rótulo da célula na qual o centróide da forma reside, chamada de *célula R* (*home cell*); 2^d bits especificam os rótulos de células interceptadas pela esfera envolvente da forma, chamadas *células I* (*phanton cell*).

2. A esfera envolvente de cada forma tem seu raio multiplicado por `sqrt(2)` e a célula da grade deve ser pelo menos 1.5 maior que a maior esfera envolvente escalonada.

Devido a essas considerações um teste de colisão é aplicado entre dois objetos pertencentes à mesma célula, se pelo menos um é residente na célula. A Figura 3.8 mostra os testes de colisão para um espaço 2D. Antes de aplicar o teste de colisão entre dois objetos pertencentes a uma célula de rótulo L, é preciso verificar se o rótulo da célula residente de um dos objetos, L', é menor que L e está entre os tipos de células que são comuns entre ambos objetos (obtido através da operação `AND` com seus 2^d bits de controle). Se esse caso ocorrer, o teste pode ser ignorado, pois:

- ou os objetos compartilham uma célula de rótulo L', teste que já foi realizado na análise da célula de rótulo L';
- ou os objetos não compartilham uma célula de rótulo L', os volumes envolventes não se sobrepõem devido ao item 2 mencionado acima.

Para um melhor entendimento alguns casos são ilustrados na Figura 3.9, onde se tem os objetos e seus respectivos volumes envolventes expanndidos.

No primeiro caso, objetos O_1 e O_2, o teste é ignorado na análise da célula de rótulo $L = 3$, pois ambos os objetos compartilham as células de rótulo 1 e 3, e o rótulo da célula residente do O_1 é $L' = 1$. O teste é aplicado na análise da célula de rótulo $L = 1$.

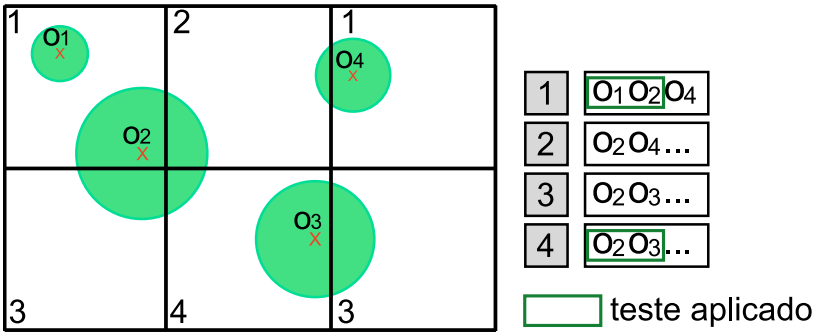


Figura 3.8: Testes de colisão considerados.

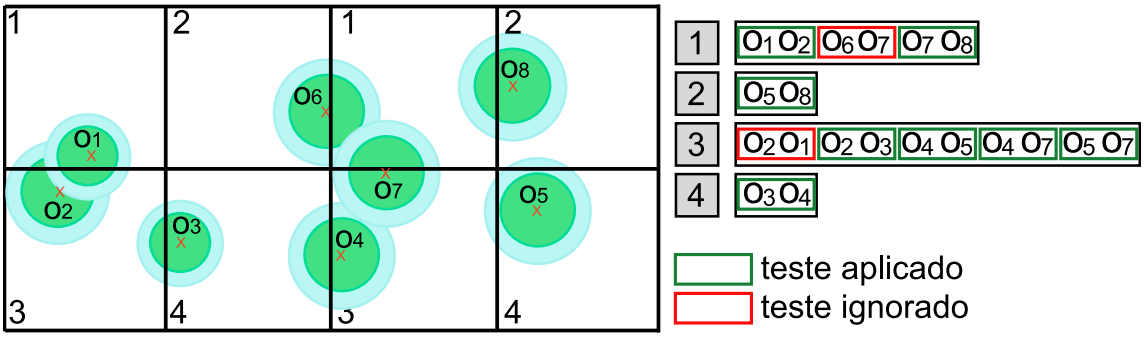


Figura 3.9: Evitando testes de colisão duplicados.

No segundo caso, objetos O_3 e O_4, o teste é ignorado na análise da célula de rótulo $L = 4$, pois ambos os objetos compartilham as células de rótulo 3 e 4, e o rótulo da célula residente do O_4 é $L' = 3$. O teste não será aplicado em nenhum momento, pois tem-se a garantia da não sobreposição entre os objetos.

No terceiro caso, objetos O_4 e O_5, o teste é aplicado na análise da célula de rótulo $L = 3$, pois ambos os objetos compartilham as células de rótulo 3 e 4, porém o rótulo da célula residente do O_5 é $L' = 4$ que é maior que $L = 3$. Visualmente pode-se verificar que não há sobreposição, porém o teste será aplicado pois não satisfaz a condição imposta.

No quarto caso, objetos O_6 e O_7, o teste só seria aplicado na análise da célula de rótulo $L = 1$, porém é ignorado devido a uma outra condição: ambos os objetos possuem seus tipos de célula residente diferente do $L = 1$, em outras palavras interceptam a célula, porém não possuem seus centróides na mesma. Essa ilustração é o caso no qual a sobreposição se torna mais próxima de ocorrer e a garantia é dada pelo aumento do volume, pois verificando visualmente os volumes expandidos se tocam, porém os objetos não. No mesmo caso estão os objetos O_6 e O_8, no qual a não sobreposição se torna clara.

A implementação utiliza basicamente o vetor de identificadores de células. Cada elemento do vetor, organizado como uma estrutura `CellIdData`, representa a relação entre um objeto, ou melhor, esfera envolvente do objeto e uma célula da grade:

```
struct CellIdData
{
   uint cellID;
   uint objID;
   bool hCell;
   uint label;
}
```

`objId` armazena o índice do objeto na lista de objetos, `hCell` representa o tipo de relação com a célula (R ou I), `cellId` armazena o identificador da célula e `label` representa seu rótulo.

A saída do algoritmo é a lista de pares de colisão contendo todos os possíveis pares de formas cujas esferas envolventes interceptam-se. Cada elemento desse vetor é representado pela estrutura `CollisionPair`, a qual simplesmente contém os índices de ambas as formas do par.

```
struct CollisionPair
{
  uint shape1;
  uint shape2;
}
```

A primeira etapa da implementação é a criação do vetor de identificadores da célula. Para tal criação os objetos são percorridos para a determinação das células na qual reside e as quais interceptam. Primeiro é determinada sua célula R e em seguida, a partir dessa, são determinadas as células I de cada objeto.

O identificador (`cellID`) da célula R de cada objeto é um *hash* das coordenadas do seu centróide em relação ao início da grades, que é calculado como:

```
int3 gridPos;
gridPos.x = (int)floor((pos.x - WORLD_ORIGIN.x) / CELLSIZE);
gridPos.y = (int)floor((pos.y - WORLD_ORIGIN.y) / CELLSIZE);
gridPos.z = (int)floor((pos.z - WORLD_ORIGIN.z) / CELLSIZE);

uint hashR = ((gridPos.x) << XSHIFT) |
       ((gridPos.y) << YSHIFT) |
       ((gridPos.z) << ZSHIFT);
}
```

`gridPos` representa os índices nas três coordenadas da célula R, `pos` representa a posição do objeto, `CELLSIZE` é a dimensão da célula, `WORLD_ORIGIN` é a posição inicial da grade, e `XSHIFT`, `YSHIFT` e `ZSHIFT` são constantes predefinidas que determinam quantos bits são atribuídos ao *hash* para cada coordenada da dimensão. `CELLSIZE`, `WORLD_ORIGIN` e (`XSHIFT`, `YSHIFT`, `ZSHIFT`) são fornecidos como parâmetros, em adição a lista de formas, através do método `Collision:setInput()`.

Após a determinação da célula R do objeto, inicia-se a criação das células I do mesmo. Os identificadores (`cellID`) das células I de cada objeto são determinados testando a interseção da esfera envolvente do objeto com algumas (feito de forma organizada não é necessário testar todas) das $3^d - 1$ células vizinhas imediatas da célula R do objeto. Após verificada a interseção com a célula o identificador (*hash*) é calculado:

```
uint hashI = hashR +  (((gridPos.x + offX) << XSHIFT) |
                       ((gridPos.y + offY) << YSHIFT) |
                       ((gridPos.z + offZ) << ZSHIFT));
```

No trecho de código acima, `offX`, `offY` e `offZ` $\in \{-1, 0, 1\}$, são os deslocamentos nas coordenadas X, Y e Z, respectivamente, referente a célula R do objeto, cujas combinações alcançam todas as células vizinhas imediatas dessa.

Assim que essas células forem criadas, devem ser armazenadas em alguma posição do vetor. A forma mais simples de armazenamento das células no vetor é seqüencialmente. Dessa forma, tem-se primeiro as células referentes ao objeto 1, seguidas das células do objeto 2, etc.

Como na ordenação do vetor células com o mesmo identificador devem ser ordenadas por seu tipo (primeiro células R depois células I), então deve-se ordenar primeiro por tipo e depois por identificador da célula, executando assim, duas chamadas ao processo de ordenação.

Para evitar duas ordenações o armazenamento é feito tal que as células R sejam alocadas antes das células I. Então, seja n o número de objetos, nas primeiras n posições do vetor são armazenadas as células R. As respectivas células I são armazenadas de forma seqüencial a partir da posição n do vetor.

Devido à necessidade de manter células R antes de células I após a ordenação, a escolha do algoritmo é, portanto, limitada a uma ordenação estável. Uma ordenação estável garante que no fim do processo de ordenação identificadores iguais das células aparecerão na seqüência ordenada na mesma ordem em que estavam no início do processo, ou seja, células R antes de células I. Para a ordenação estável, o algoritmo utilizado é o *radix sort*, Figura 3.10.

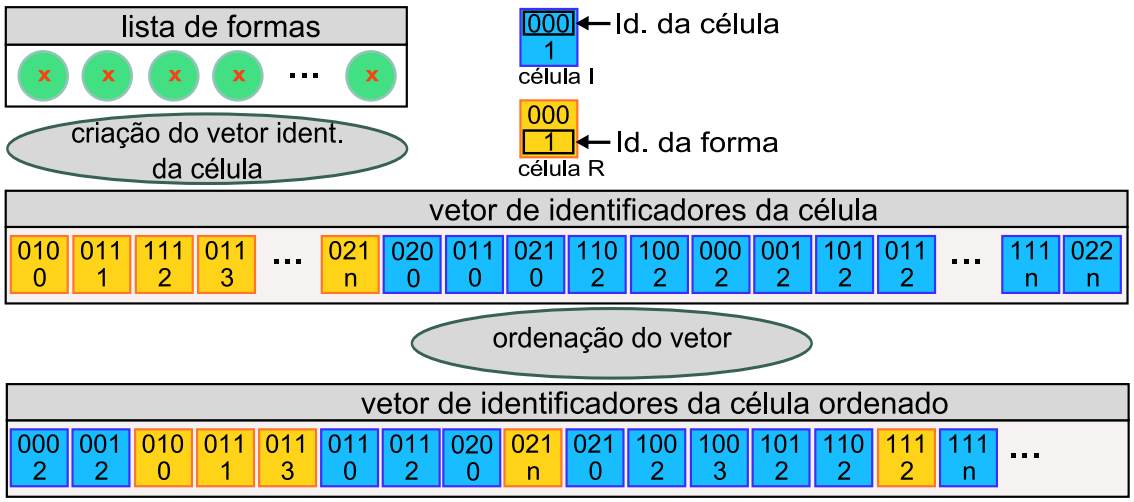


Figura 3.10: Vetor de identificador de células ordenado.

Por fim, o vetor ordenado é percorrido verificando-se os grupos de células com identificadores iguais, chamados células de colisão. Uma célula de colisão é um conjunto de células (com mesmo identificador) que requer processamento de testes de colisão, isto é, contém $r \geq 1$ células R e $i \geq 0$ células I onde $r + i \geq 2$. A idéia inicial dessa fase é efetuar os testes de colisão entre todos objetos da célula de colisão (força bruta). Porém os testes de colisão são efetuados somente entre células R $\times$ R e células R $\times$ I. Os testes de colisão das células I $\times$ I podem ser ignorados devido as considerações de escala na esfera envolvente feitas anteriormente. Para facilitar a montagem dos testes foi feita a ordenação de células R antes de células I dentro de uma célula de colisão.

Antes de aplicar o teste de colisão entre as células R $\times$ I deve-se fazer a seguinte verificação: seja L o rótulo da célula R e L' o rótulo da célula I, se $L' < L$ e o objeto da célula R intercepta L', então o teste é ignorado. Esta é uma simplificação do teste visto anteriormente, pois nesse caso tem-se a garantia do teste entre uma célula R e uma célula I.

Resumindo, um teste de colisão é aplicado entre todas as células R $\times$ R, pois trata-se dos objetos cujos centróides pertencem à célula, e entre as células R $\times$ I, caso não seja verdadeiro o teste acima.

Outra consideração importante é que, como AS permite a criação de corpos rígidos compostos por várias formas e estas são tratadas individualmente na detecção de colisões, então os testes entre as formas de um mesmo corpo devem ser ignorados e isso é verificado através da comparação do atributo `bodyId`.

Um teste de colisão consiste em comparar a interseção entre as esferas envolventes (não aumentadas) de duas formas. Em caso de interseção, um par de colisão é criado e adicionado à lista de pares para posteriormente ser analisado durante a fase exata.

A algoritmo da fase geral é mais eficiente quando as formas por ele consideradas possuem dimensões relativamente parecidas. Porém, nas simulações podem existir corpos cujas dimensões são muito diferentes da maioria dos outros. Esses corpos normalmente são estáticos, porém essenciais à simulação. Como exemplo pode-se citar um plano, uma caixa com diversas formas, etc. Pensando nisso criou-se um tipo de forma denominado ilimitado (*unbounded*). Esta possui a esfera envolvente muito maior que as demais formas ou infinita (no caso do plano), por isso não são consideradas no algoritmo da fase geral. Assumi-se que formas ilimitadas pertencem a um corpo estático.

Como as formas ilimitadas são estáticas não colidem com outras do mesmo tipo. Portanto, os testes de colisão que são feitos para considerar esse novo tipo de forma são testes entre as formas "normais", ou com limites e formas ilimitadas. Para isso, para cada forma com limites, percorre-se a lista de formas ilimitadas testando a colisão entre as formas, e em caso de interseção, o par de colisão é criado e adicionado à lista de pares. O teste de colisão deve ser ignorado no caso da forma ilimitada se tratar de um plano, pois o teste é sempre positivo devido sua esfera envolvente possuir raio infinito.

As formas *unbounded* são armazenadas na mesma lista, no entanto iniciam-se no fim do vetor e o preenchimento é feito em direção ao início. Já as formas *bounded* iniciam-se no começo do vetor e são preenchidas em direção ao fim.

Os passos descritos até no momento são decorrente da fase geral da detecção de colisão e pertencem ao método `Collision::runBroad()`. Após a determinação dos possíveis pares de colisão através da fase geral, é necessário a determinação dos pontos de contato e isso é feito na fase exata.

Fase exata

Na fase exata, o método responsável `Collision::runNarrow()` tem a função de percorrer todos os pares de colisão invocando o método `Collision::collide(Contact* c, Shape*, Shape*)`. Esse método invoca o método adequado para a determinação dos pontos de contato da colisão de acordo com os tipos das formas do par de colisão analisado.

`Contact` é uma estrutura que armazena os dados de um contato, dentre eles dados geométrico e da superfície. Os dados geométricos do contato, organizados na estrutura `ContactGeom`, são dois índices para ambos os corpos participantes do contato (`body1`, `body2`) e posição (`pos`), normal (`normal`) e profundidade (`depth`) do ponto de contato. As informações da superfície, organizados na estrutura `SurfaceParam`, são principalmente coeficiente de restituição (`bounce`) e coeficiente de atrito (`mu`). Somente dados geométricos são determinados nessa fase. Mesmo nessa análise detalhada é possível que as formas não se colidam, nesse caso somente as esferas envolventes se interceptaram e nenhum ponto de contato é criado. A saída da fase exata é uma lista de pontos de contato.

```
struct Contact
{
   SurfaceParameters surface;
   ContactGeom geom;
}

struct Contact
{
   Vec3 pos;
```

```
    Vec3 normal;
    float depth;
    int body1;
    int body2;
}

struct SurfaceParameters
{
    float bounce;
    float mu;
    ...
}
```

O sistema oferece suporte a primitivos esferas, caixas, cápsula e planos, portanto tem-se implementado as respectivas funções de determinação dos contatos entre: esfera-esfera, esfera-caixa, esfera-cápsula, esfera-plano, caixa-caixa, caixa-cápsula, caixa-plano, cápsula-cápsula, cápsula-plano. Essas funções foram extraídas das implementações do [Smi00] e de [Eri04].

Uma vez determinados os contatos entre as formas, os quais servirão de entrada para o motor de física, são encerradas as tarefas do detector de colisões.

3.5 Comentários finais

O detector de colisões e sua implementação em CPU foram apresentados nesse capítulo. Este componente é fundamental na simulação física uma vez que são decorrentes dele os contatos considerados pelo PCL. O detector de colisões foi dividido em duas fases. A primeira, denominada fase geral, foi criada para diminuir os testes entre formas e consiste em uma verificação rápida dos pares de formas que têm possibilidade de colisão. Somente estes serão analisados em detalhes durante a fase exata, na qual é verificado se há efetivamente contato entre corpos e, nesse caso, determinadas as propriedades dos pontos de contato.

CAPÍTULO 4

GPGPU com CUDA

4.1 Introdução

As GPUs são projetadas para acelerar a renderização de gráficos tridimensionais, sendo utilizada, principalmente, em jogos digitais, simuladores, aplicações CAD, entre outras. Tradicionalmente, o conjunto de operações suportadas pela GPU é limitado a transformações simples, instruções de cor e luminosidade. No entanto, o crescimento da demanda para o realismo nos jogos digitais tem permitido a GPU a oferecer maior programabilidade para suportar operações com gráficos arbitrários.

Devido a esse crescimento acelerado do poder de processamento das GPUs comparado às CPUs, há um maior interesse no uso da GPU para computação de propósito geral. Esse campo é conhecido como GPGPU (*general purpose computing on GPU*). As GPUs mais recentes são significantemente mais potentes que os processadores atuais, e o poder de computação está crescendo a uma taxa maior que a das CPUs. Existem duas razões para isso:

1. A GPU é um processador de propósito especial. Assim, ela está apta a destinar mais transistores à computação que a CPU, a qual deve ser capaz de executar qualquer tipo de programa. Dado, para ambos chips, a mesma quantidade de transistores, a GPU oferece maior poder computacional por chip.

2. GPUs são construídas baseadas na arquitetura SIMD[1] — *single instruction, multiple data*— ou seja, uma arquitetura com unidades de processamento paralelas (ou núcleos), onde vários conjuntos de dados podem ser processados simultaneamente, pelo mesmo conjunto de instrução ou kernel, executados em unidades distintas. Portanto, cada geração pode melhorar seu desempenho com relação à anterior adicionando mais núcleos de processamento. Já as CPUs, tradicionalmente, possuem arquitetura SISD — *single instruction single data* — ou seja, são baseadas na arquitetura tradicional de *von Neumann*, com processamento linear e seqüencial. Apesar das CPUs tenderem, atualmente, para o recurso de dois ou mais núcleos, as GPUs ainda contam com uma vantagem significante nessa área por enquanto.

Para levar vantagem de alguma arquitetura *multi-núcleo* é necessário que programas sejam escritos para execução de tarefas em paralelo. Em alguns problemas, nos quais há

[1] Considera-se SIMD as operações sobre vértices e MIMD — *multiple instruction, multiple data* as operações sobre pixels.

independência dos dados a paralelização se torna direta. Por outro lado, em problemas com interdependência entre os dados, algumas técnicas são utilizadas para a subdivisão do problema em partes independentes, as quais são solucionadas de forma paralela. Esse também é o processo adotado pela GPU, porém esta provê um modo de programação alternativo, que em alguns casos torna a implementação mais difícil.

O poder computacional das GPUs poderia ser aproveitado com mais facilidade em outras áreas da computação, além da computação gráfica, não fossem algumas barreiras impostas. A principal vem do fato de a evolução deste tipo de hardware estar intimamente relacionada com o desenvolvimento de jogos digitais, sendo assim, tanto o ambiente como o próprio modelo de programação estão altamente restringidos e orientados a satisfazer as necessidades inerentes a este modelo de software, exigindo-se, desta forma, conhecimento por parte do programador, da arquitetura subjacente a este tipo de hardware, de forma a possibilitar um uso eficiente do mesmo.

Entretanto, o recente aparecimento de certas APIs (*application programming interfaces*), tais como CUDA (*Compute Unified Device Architecture*) da NVidia e CTM (*Close to Metal*) da ATI, contribuíram neste aspecto, proporcionando aos programadores uma interface suficientemente baixo nível, que lhes permite desenvolver ferramentas e aplicações de alto nível.

Em seguida, são apresentados alguns mecanismos de funcionamento das GPU's. Na seção Seção 4.2 apresentado-se o *pipeline* gráfico das GPUs tradicionais e a maneira de como é feita a GPGPU nesses hardwares. A seção Seção 4.2 mostra o funcionamento e alguns detalhes da API CUDA, arquitetura utilizada no projeto. Por fim, na seção Seção 4.3, apresenta um trecho de código em CUDA.

4.2 Pipeline gráfico

O *pipeline* gráfico refere-se à seqüência de operações para transformar um modelo poligonal de uma cena tridimensional em uma imagem renderizada na tela. A GPU é um dispositivo físico com capacidade de executar essas operações. A programação sobre este modelo é feita através de APIs gráficas, tais como, OpenGL e Direct3D, implementadas pelos fabricantes de hardware.

O *pipeline* gráfico inicia-se recebendo da CPU uma lista de vértices interconectados e, então, através do processador de vértices, aplica operações de transformação tais como escala e rotação, e também iluminação para determinar a cor de cada vértice. Os vértices são rasterizados para resultar nos fragmentos (termo utilizado para descrever um pixel antes de ser renderizado na tela) individuais dos quais, através do processador de fragmentos, a cor final é computada. Rasterização é o processo de transformar uma imagem formada por vértices em uma imagem aproximada por fragmentos. A cor de cada fragmento é determinada por interpolação entre as cores dos vértices ou possivelmente valores de uma textura em memória. Em jogos de computador, as texturas normalmente contém ilustrações as quais são aplicadas na superfície dos objetos, porém as texturas podem ser usadas para armazenar quaisquer valores de ponto flutuante de 32 bits, como no caso de programação genérica.

Por fim, os fragmentos visíveis a partir da posição da câmera são selecionados e armazenados em uma área do hardware gráfico denominada *frame buffer*, para posteriormente serem visualizados na tela. Para executar essas tarefas o hardware gráfico provê diversos processadores de vértice e de fragmento, sendo o último normalmente em maior número.

Tradicionalmente, cada estágio desse *pipeline* possuía funcionalidade fixa, porém com a evolução das GPUs, alguns estágios passaram a ser programáveis. Os estágios programáveis do *pipeline* permitem que um desenvolvedor crie o algoritmo usado pelo estágio por meio da

escrita de *shaders*. Os *shaders* são um conjunto de comandos que aceitam recursos gráficos de entrada, executam uma série de instruções (usando registradores temporários, constantes e acesso a leitura de texturas) sobre esses recursos e apresentam o resultado. Os *shaders* podem ser escritos em assembly (linguagem de montagem) ou em linguagens específicas de alto nível, tais como: HLSL (*High Level Shading Language*) para Direct3D, ou GLSL (*OpenGL Shading Language*) para OpenGL, que permitem a programação de *shaders* em nível de algoritmos. A implementação de shaders em uma aplicação gráfica, permite maior flexibilidade ao programador para tratar os dados dentro do pipeline.

Um *shader* de vértice é executado por um processador de vértice. Este é associado a um vértice, do qual possui permissão para manipular os atributos tais como, posição, normal, cor, coordenada de textura, etc. Neste estágio não é possível trocar um tipo primitivo nem criar, destruir ou inverter a ordem de seus vértices.

Um *shader* de fragmento é executado por um processador de fragmento. Este é associado a um pixel, e é responsável basicamente por determinar a cor final e o valor de profundidade para os dados do pixel. Os dados de entrada, normalmente, são passados em forma de textura, e cada processador acessa a posição referente à coordenada de textura do fragmento, porém é permitido o acesso a qualquer posição da textura por qualquer processador de fragmento. Por outro lado, a escrita em um pixel só pode ser feita pelo processador de fragmento associado a ele e o resultado de um pixel não pode ser usado por outro processador no mesmo passo de renderização.

O suporte à instruções de desvio nos processadores de fragmento é limitado, e algumas operações possuem um custo computacional elevado. Isso ocorre devido à execução simultânea da mesma instrução por todos os processadores de fragmento. Se alguma condição de desvio é avaliada diferentemente entre quaisquer processadores, ambos os lados do desvio são avaliados por todos os processadores. Por isso, o desempenho total depende da divergência do desvio. Laços, que não podem ser expandidos durante a compilação, também causam problemas similares.

Geralmente na programação genérica utilizam-se somente *shader* de fragmento. Algumas soluções, no entanto, requerem o uso de ambos, o que é possível, pois são aplicados em momentos distintos do pipeline. Além disso, tem-se a opção de renderização em múltiplos passos com diferentes *shaders*, onde o resultado corrente armazenado no *frame buffer* é utilizado no passo seguinte.

O modelo de programação utilizado no *pipeline* gráfico é normalmente abstraido como programação em fluxo ou *streaming programming*. Nesse modelo, o programa executado pelas diversas unidades de processamento (de vértice ou fragmento) são denominados núcleos (ou *kernels*), que atuam sobre o fluxo de dados (*data stream*).

O procedimento básico para a computação genérica utilizando a GPU é descrito a seguir:

- Um quadrilátero com certa dimensão é criado e passado para o processador de vértice. Este é rasterizado formando os fragmentos que são passados para os processadores de fragmentos.
- Um *shader* de fragmento é aplicado a cada fragmento independentemente pelo processador de fragmento associado a ele.
- A saída dos processadores de fragmentos é armazenada em um *frame buffer* que pode ser copiado para CPU ou usado como entrada em um próximo passo.

Alcançar todo o poder computacional contido na GPU e utilizá-lo eficientemente para aplicações não gráficas, através desse procedimento, é uma tarefa custosa devido a alguns fatores:

- A programação da GPU é feita através de API's gráficas, exigindo do programador conhecimento adicional na área de computação gráfica.
- O desempenho é reduzindo devido à largura de banda da memória e ao ajuste de diversos parâmetros gráficos necessários somente para o funcionamento da API.
- A flexibilidade de programação, comumente oferecida em CPU, é reduzida devido à incapacidade de escrita em qualquer parte da memória principal da GPU (executar *scatter*), embora seja possível ler em qualquer parte da mesma (executar *gather*).
- A GPU possui limitações quanto ao número de texturas de entrada (dados de entrada) e número de *frame buffers* de saída (dados de saída).

Aplicação

As dificuldades mensionadas acima foram encontradas durante o início deste projeto com a implementação do componente do motor de física denominado solucionador de EDO. O método de integração numérica da equação de movimento de cada corpo rígido utilizado foi do tipo Runge-Kutta de quarta ordem [Ebe04].

A fim de possibilitar o uso de GPU para propósito geral, foi desenvolvido um framework cujas classes bases foram projetadas para serem independentes de uma linguagem de *shader* particular. A partir dessas classes bases, não discutidas aqui, podem-se derivar classes específicas para uma determinada linguagem de *shader*. No diagrama de classes da Figura 4.1 são mostradas as principais classes do framework para programação de GPU com GLSL, a linguagem de shader adotada na implementação.

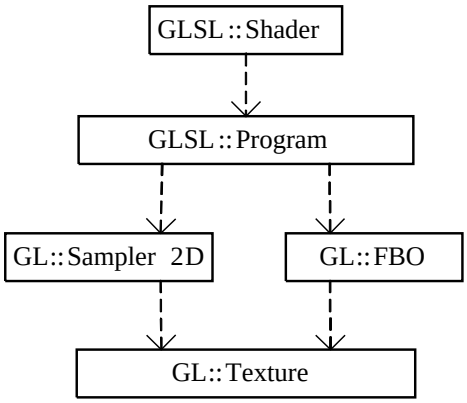


Figura 4.1: Principais classes para GPGPU com GLSL.

Um objeto da classe `GLSL::Shader` representa um shader em GLSL, isto é, um módulo de código executável em GPU. O construtor da classe toma como argumentos o nome e um inteiro que identifica se o objeto sendo criado é um shader de vértice ou de fragmento. O principal método da classe `loadSourceFromFile(fileName)` carrega o código fonte do shader a partir do conteúdo do arquivo `fileName`.

Um objeto da classe `GLSL::Program` é um contêiner de shaders GLSL. Os resultados de processamento de um programa GLSL são armazenados em um objeto da classe `GL::FBO` (*frame buffer object*), descrita adiante. `GLSL::Program` declara métodos públicos para obtenção e definição do FBO usado por um programa. Além desses, a classe contém métodos para adicionar shaders, inserir o programa no pipeline da GPU e executar o programa em GPU. Esses métodos são controlados através de estados pois suas execuções devem ser ordenadas.

Um objeto da classe `GL::Texture` representa uma textura da OpenGL. Na implementação corrente do framework apenas texturas 2D cujos texels (elemento da textura) podem armazenar 1, 2, 3 ou 4 números reais de 32 bits são consideradas. O construtor da classe toma como argumento um descritor de textura, o qual é uma estrutura contendo inteiros que define a largura, altura e o formato dos texels (1, 2, 3, ou 4 números reais) necessários para a criação da textura. Os principais métodos públicos de `GL::Texture` permitem escrever dados de um buffer em memória para uma textura e ler os dados armazenados em uma textura para um buffer em memória. Texturas são usadas em um programa GLSL como repositório de dados de entrada e saída.

Um objeto da classe `GL::Sampler2D` representa uma unidade de textura que pode ser passada como parâmetro para um programa GLSL. O número máximo de unidades de textura usada em um programa é dependente do hardware; a implementação do solucionador de EDO em GPU proposto supõe que este número seja pelo menos seis. Os atributos de uma unidade de textura são um identificador inteiro não negativo menor que o número máximo de unidades de textura possível e uma referência da textura 2D contendo os dados de entrada, ambos definidos pelos argumentos do construtor da classe. Os principais métodos públicos são `setTexture(texture)`, que define `texture` como sendo a textura 2D da unidade de textura receptora da mensagem, e `write(x,y,w,h,data)`, que escreve a partir do texel `(x,y)` da textura 2D da unidade de textura `w` × `h` texels da área de memória endereçada por `data`. Um objeto dessa classe é usado para definir os dados de entrada que serão passados para um programa GLSL.

Um objeto da classe `GL::FBO` representa um FBO, uma extensão da OpenGL usada na implementação para renderização em texturas as quais armazenam os resultados de processamento de um programa GLSL. O número de texturas que um FBO pode renderizar simultaneamente depende da GPU. A implementação do solucionador de EDO em GPU proposto supõe que pelo menos quatro texturas possam ser anexadas a um FBO.

O construtor de `GL::FBO` toma como argumentos um descritor de textura e um inteiro igual ao número de texturas a serem anexadas ao FBO (limitado ao valor máximo aceito pelo hardware). O construtor cria as texturas a partir do descritor dado e as anexa ao FBO. A classe tem três principais métodos públicos. Dois métodos são para habilitar as texturas do FBO para escrita, e o método para troca de textura da unidade de textura com uma textura do FBO, usado para fazer com que os dados de saída de um programa sejam usados como dados de entrada do mesmo ou de outro programa GLSL. Além desses, há também o método `read(i,x,y,w,h,data)`, o qual lê a partir do texel `(x,y)` da `i`-ésima textura do FBO `w` × `h` texels para a área de memória endereçada por `data`, usado para capturar os dados de saída gerados por um programa GLSL.

O integrador da equação de movimento implementado em GPU é um objeto da classe `ODESolverGPU`, derivada de `GLSL::Program`. O programa possui apenas um shader de fragmento e usa para saída um FBO com quatro texturas com largura igual a 1, altura igual a n, onde n é o número de corpos rígidos na cena, e formato de texel de 4 números reais de 32 bits (necessário para armazenamento de quaternions). O shader aplica o método de Runge-Kutta de quarta ordem e gera, em cada texel da i-ésima linha de cada uma das texturas do FBO, $1 \leq i \leq n$, as variáveis $\mathbf{x}_i$, $\mathbf{q}_i$, $\mathbf{V}_i$ e

A principal razão pela qual motiva a evolução das GPUs com múltiplos núcleos e memórias ampla largura de banda é o fato destas serem especializadas na computação intensiva e altamente paralelizável, que representa exatamente o que é a renderização gráfica. Isto explica a razão pela qual uma maior quantidade de transistores é alocada para o processamento de dados ao invés de serem alocados para caching de dados e controle de fluxo, como ilustrado na Figura 4.2.

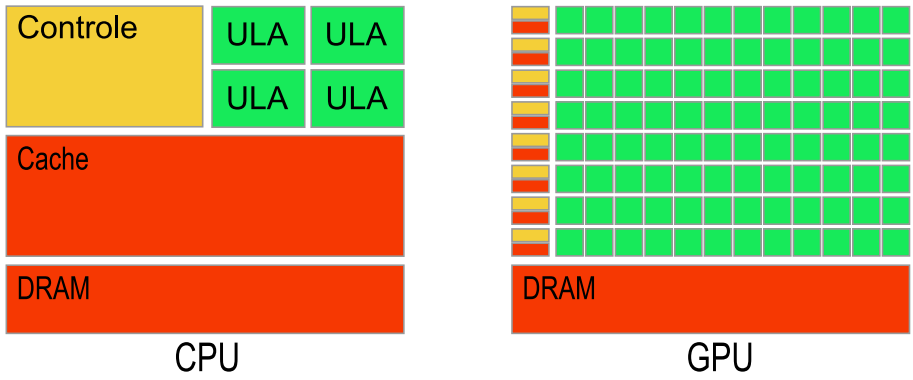


Figura 4.2: Transistores, caching de dados e controle de fluxo: CPU e GPU.

Mais especificamente, a GPU está especialmente destinada para lidar com problemas que podem ser tratados sob a forma de computações de dados em paralelo de grande intensidade aritmética. Dado que um mesmo programa é executado recorrentemente sobre cada elemento de dados, não existe a necessidade de um sofisticado controle de fluxo. Por outro lado, estas características permitem que a latência nos acessos de memória seja disfarçada com cálculos, evitando-se com isso o uso de caches de grandes dimensões.

O processamento de dados em paralelo mapeia os elementos de dados para processamento em threads em paralelo. Diversas aplicações que processam grandes conjuntos de dados podem usar um modelo de programação paralelo para aumentar o desempenho da computação. Tal é verificado no exemplo da renderização 3D onde grandes conjuntos de pixels e vértices são distribuídos em várias threads paralelas. Este método é aplicável noutros contextos como processamento de imagens ou vídeos, estendendo-se ainda a outras áreas de aplicação fora da computação gráfica, tais como na biologia computacional, computação financeira ou até em simulações de modelos físicos.

CUDA (Compute Unified Device Architecture)

CUDA é uma nova arquitetura de hardware e software que permite tratar de computação que utiliza paralelismo de dados, sem ter de transformá-la para funcionar numa API gráfica. CUDA oferece às aplicações intensivas computacionalmente acesso ao poder de processamento das GPUs através de uma nova interface de programação. Fornecendo maior desempenho e simplificando o software de desenvolvimento usando a linguagem C, CUDA permite criar soluções inovadoras para problemas com grande intensidade de dados. Essa tecnologia está disponível para GPUs NVidia a partir da Série 8 da GeForce, Quadro FX 5600/4600 e GPU Tesla.

O software CUDA é composto de diversas camadas como ilustrado na Figura 4.3: um controlador de hardware, uma API e a sua runtime e duas bibliotecas matemáticas de alto nível de uso comum, CUFFT e CUBLAS, cujos detalhes podem ser encontrados em [Cor06]. Esta API possui a vantagem de oferecer uma extensão à linguagem de programação C comentada nas próximas seções.

CUDA fornece endereçamento de memória DRAM, que permite uma maior flexibilidade de programação: ambas as operações *gather* e *scatter*. Do ponto de vista da programação, isto se traduz na possibilidade de ler e escrever dados em qualquer parte da memória, como em CPU, algo que nas GPU mais antigas era impossível, como dito anteriormente.

Além disso, oferece ainda cache de dados paralelos ou memória compartilhada no processador com acessos de leitura e escrita bastante eficientes, as quais podem ser utilizadas pelas

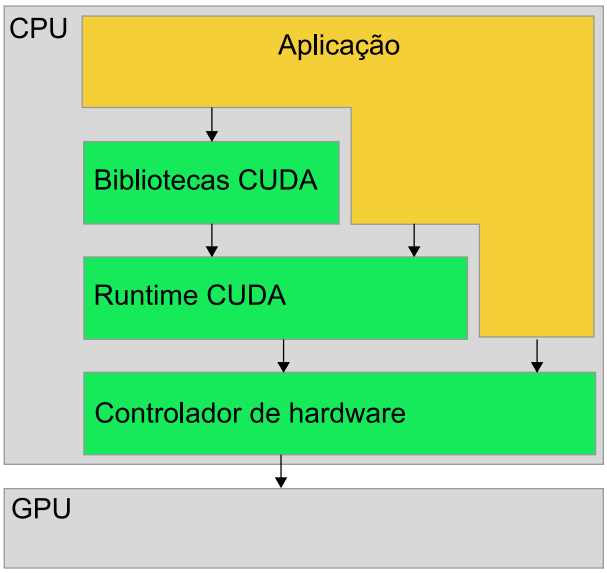


Figura 4.3: Camadas do software CUDA.

threads para compartilhar dados. As aplicações podem tirar proveito disto, minimizando o *overfetch* e *round-trips* à DRAM e conseqüentemente diminuindo a dependência à largura de banda.

A GPU como processador auxiliar multitarefa

Com programação em CUDA, a GPU é vista como um dispositivo computacional capaz de executar um alto número de threads em paralelo, operando como um co-processador da CPU. Assim, porções de dados (ou funções) paralelizáveis e que consumam grandes quantidades de recursos computacionais são carregados na GPU, aliviando a carga da CPU. Para tal, é necessário que a função seja isolada e compilada para o hardware específico da GPU, resultando então em um programa, denominado *kernel*, que é mais tarde carregado e executado pelo dispositivo.

Tanto o host quanto o dispositivo mantém suas próprias memórias, possibilitando, no entanto, que um possa efetuar cópias de dados de uma memória para a outra, através de chamadas otimizadas presentes na API, que utilizam o acesso direto à memória (DMA).

Grid, Blocos e Threads

O grupo de threads que executa um kernel é organizado como um *grid* de blocos de threads, ilustrado na Figura 4.4.

Um *bloco* é um grupo de threads que podem compartilhar dados através da memória compartilhada e sincronizar suas execuções para coordenar o acesso a memória. Pontos de sincronização no kernel podem ser especificados, nos quais as threads do bloco são suspensas até que todas alcançam o ponto de sincronização.

Cada thread é identificada por um número inteiro da thread, *ID da thread*, o qual é o número da thread interno ao bloco. Para facilitar endereçamentos complexos baseados no ID da thread, a aplicação pode também especificar um bloco como um vetor bi ou tri-dimensional de tamanho arbitrário e identificar cada thread através de um índice de 2 ou 3 componentes. Para um bloco bi-dimensional de tamanho (D_x, D_y), o ID da thread de índice

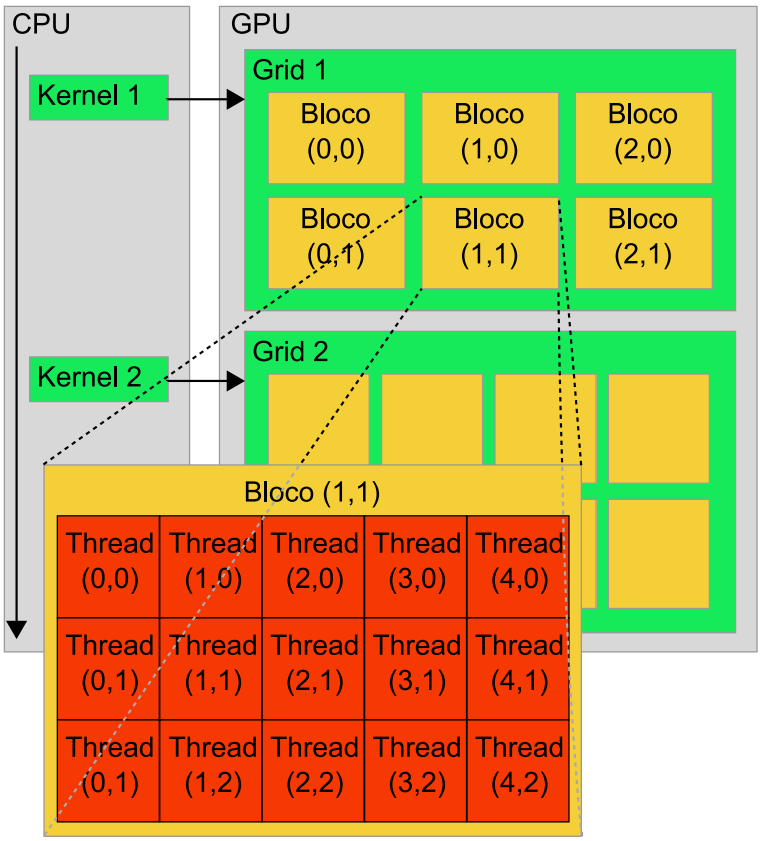


Figura 4.4: Grupo de threads.

(x, y) é $(x + y \times D_x)$ e para um bloco tri-dimensional de tamanho (D_x, D_y, D_z), o ID da thread de índice (x, y, z) é $(x + y \times D_x + z \times D_x \times D_y)$.

Existe um limite máximo do número de threads que um bloco pode conter [Cor06]. No entanto, blocos da mesma dimensão que executam o mesmo kernel podem ser agrupados em um *grid* de blocos, fazendo com que o número de threads para uma única invocação de um kernel seja muito maior. Isso tem o custo de uma menor cooperação entre as threads, pois threads em blocos diferentes de um mesmo *grid* não podem se comunicar e/ou sincronizar uma com a outra. A vantagem deste modelo é permitir que kernels sejam executados sem recompilação por diversos dispositivos com diferentes capacidades. Um dispositivo deve executar todos os blocos de um *grid* seqüencialmente se contiver pouca capacidade, ou em paralelo se a capacidade for adequada, ou como usualmente em uma combinação de ambos.

Cada bloco é identificado por um número inteiro do bloco, *ID do bloco*, o qual é o número do bloco interno ao *grid*. Para facilitar endereçamentos complexos baseados no ID do bloco, a aplicação pode também especificar um *grid* como um vetor bi-dimensional de tamanho arbitrário e identificar cada bloco usando um índice de 2 componentes. Para um *grid* bi-dimensional de tamanho (D_x, D_y), o ID do bloco de índice (x, y) é $(x + y \times D_x)$.

Implementação do Hardware

O dispositivo é implementado como um conjunto de multiprocessadores, como ilustrado na Figura 4.5. Cada multiprocessador contém uma arquitetura SIMD (*single instruction, multiple data*): em um dado ciclo de clock, cada processador do multiprocessador executa a mesma instrução, porém sobre dados diferentes. Cada multiprocessador possui memória local de quatro tipos diferentes:

- Um conjunto de registradores locais (de 32 bits) por processador.
- Uma memória compartilhada à qual todos processadores possuem acesso — (espaço de memória compartilhada).
- Um cache de memória constante, somente de leitura, compartilhado por todos os processadores — (espaço de memória constante).
- Um cache de textura, somente de leitura, compartilhado por todos os processadores — (espaço de memória de textura).

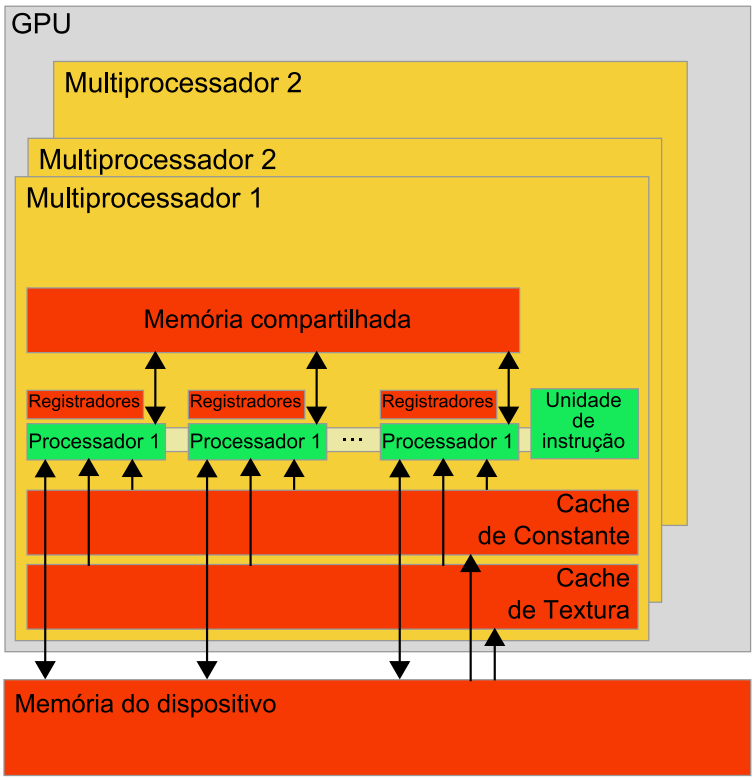


Figura 4.5: Modelo do hardware.

Os espaços de memória local e global são implementados como regiões do dispositivo de leitura e escrita e não possuem cache.

Modo de execução

Um *grid* de blocos de threads é executado no dispositivo por escalonamento dos blocos para execução nos multiprocessadores. Cada multiprocessador processa grupos de blocos, sendo um grupo após o outro. Um bloco é processado por somente um multiprocessador, portanto o espaço de memória compartilhada reside na memória compartilhada do *chip*, tornando o acesso à memória mais eficiente.

A quantidade de blocos que cada multiprocessador pode processar em um grupo depende da quantidade de registradores por thread e quanto de memória compartilhada por bloco estão sendo requisitados por um dado kernel, visto que os registradores e a memória compartilhada do multiprocessador são divididos por todas as threads do grupo de blocos.

Os blocos processados por um multiprocessador em um grupo são referenciados como ativos. Cada bloco ativo é dividido em grupos SIMD de threads denominados *warps*. O

tamanho da *warp* é definido como o número de threads pertencentes à *warp*, e é executado por um multiprocessador em uma forma SIMD. As warps ativas — todas pertecentes aos blocos ativos — são escalonadas periodicamente por um escalonador de threads que permuta de uma *warp* para outra a fim de maximizar o uso do recurso computacional do multiprocessador. Uma *meia-warp* é a primeira ou a segunda metade de uma *warp*.

A maneira de como um bloco é dividido em warps é sempre a mesma: cada *warp* contém threads consecutivas, relacionado aos identificadores das threads, com a primeira *warp* contendo o identificador de thread 0. A ordem do fluxo das *warps* internas a um bloco é indefinido, porém suas execuções podem ser sincronizadas, coordenando acessos à memória global e compartilhada.

A ordem do fluxo dos blocos internos a um *grid* é indefinido e não há nenhum mecanismo de sincronização entre blocos, portanto threads de blocos diferentes de um mesmo *grid* não podem se comunicar seguramente uma com a outra através da memória global durante a execução de um *grid*.

Detalhando a execução em nível de instrução, se uma instrução não atômica, executada por uma *warp*, escreve em uma mesma posição na memória global (ou compartilhada) por mais de uma das threads pertencentes à *warp*, o número de escritas serializadas que ocorrem na posição e a ordem em que elas ocorrem é indefinida, porém ao menos uma das escritas é garantida obter sucesso. Quando há a mesma situação para uma instrução atômica, a qual lê, modifica e escreve em uma mesma posição da memória, cada leitura, modificação e escrita naquela posição ocorrem e elas são serializadas, porém a ordem na qual ocorrem é indefinida.

A situação acima ocorre pois uma função atômica realiza uma operação de leitura-modificação-escrita atômica sobre uma palavra de 32 bits da memória global. Como exemplo, `atomicAdd()`, função da API que lê uma palavra de 32 bits de algum endereço da memória global, adiciona um inteiro e escreve o resultado de volta ao mesmo endereço. A operação é atômica no sentido de garantir que será realizada sem interferência de outras threads, isto é, nenhuma outra thread pode acessar este endereço até que a operação esteja finalizada. A restrição às operações atômicas é operar somente sobre inteiros de 32 bits com ou sem sinal.

Modelo da Memória

Uma thread executada no dispositivo tem acesso à memória DRAM e memória local do *chip* somente através dos *espaços de memória*, como ilustrado na Figura 4.6, que podem ser:

- registradores — leitura-escrita por thread;
- memória local — leitura-escrita por thread;
- memória compartilhada — leitura-escrita por bloco;
- memória global — leitura-escrita por *grid*;
- memória constante — somente leitura por *grid*;
- memória de textura — somente leitura por *grid*.

Os espaços de memória global, constante e de textura podem ser lidos ou escritos pelo host, ou CPU, e são persistentes durante a execução do kernel na mesma aplicação.

Tendo em vista que a memória global tem maior latência e menor largura de banda comparada à memória local, essa deve ter acesso minimizado. Um padrão ideal de programação é armazenar, durante a execução, os dados da memória global na memória compartilhada, em outras palavras, cada thread de um bloco:

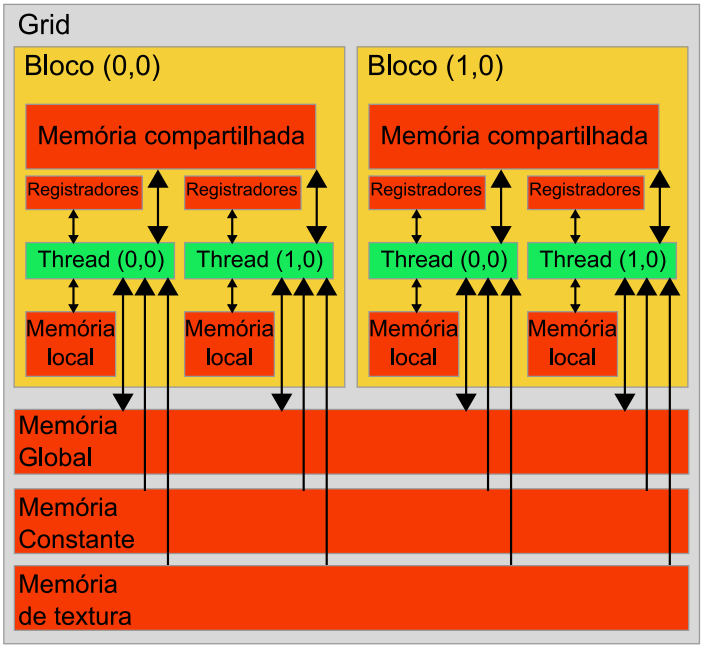


Figura 4.6: Modelo de memória.

- carrega os dados da memória global para a memória compartilhada;
- sincroniza todas as threads do bloco para uma leitura correta dos dados;
- processa os dados na memória compartilhada;
- sincroniza novamente, se necessário, para confirmar todos os resultados na memória compartilhada;
- escreve os resultados de volta na memória global.

Para que esse padrão seja eficiente necessita-se de soluções que fazem diversos acessos ao mesmo dado durante a execução do kernel, porém o que normalmente acontece é uma leitura única do dado, processamento e escrita, não havendo a necessidade de cópia do mesmo. Além disso, o padrão requer que o problema possa ser dividido em blocos, pois é somente por bloco que as threads podem ser sincronizadas. Um exemplo em que o problema é resolvido na forma ideal com relação ao acesso a memória é a multiplicação de matrizes apresentada em [Cor06].

O espaço de memória constante possui cache, então um acesso a ela custa igual a um acesso à memória global somente quando ocorrer uma falha na cache; caso não ocorra, o custo é o semelhante ao acesso a um registrador. O espaço de memória de textura funciona da mesma forma do supracitado e possui uma otimização da cache para localidade espacial 2D, fazendo com que threads do mesmo *warp* que lêem uma textura cujos endereços são próximos obtenham um melhor desempenho.

O espaço de memória compartilhada, por estar no *chip*, é mais rápido que os espaços de memória global e local. Desconsiderando a existência de conflitos de acessos, para todas as threads de uma *warp*, acessar a memória compartilhada é tão rápido quanto acessar um registrador. Registrador é a memória que possui o acesso mais eficiente, porém atrasos podem ocorrer devido às dependências de leitura após escrita e aos conflitos de acessos. Otimizações possíveis são feitas pelo compilador para evitar conflitos, diferentemente da memória compartilhada, onde quem deve evitar os conflitos, através de organização nos acessos, é o programador.

Uma extensão da linguagem de programação C

O objetivo da interface de programação CUDA é providenciar uma forma simples de programadores familiarizados com a linguagem C desenvolverem programas, sem grandes complicações, para execução em GPU.

A interface de programação oferece um conjunto de extensões à linguagem C, que permitem ao programador selecionar partes de um código fonte para execução na GPU e em uma biblioteca runtime que é dividida em três partes:

- uma componente do host, que é executada no host e fornece funções para controlar e acessar um ou mais dispositivos a partir deste;
- uma componente do dispositivo (GPU), que é executada no dispositivo (GPU) e fornece funções específicas deste;
- um componente comum a ambos (CPU e GPU), que fornece tipos de vetores e um subconjunto da biblioteca padrão C que é permitida tanto no código do host quanto código do dispositivo.

As únicas funções das bibliotecas do C padrão que são permitidas na GPU são aquelas fornecidas pelo componente comum.

Dentre as extensões da linguagem de programação C, estão inicialmente os qualificadores do tipo de função, os quais servem para especificar onde a função será executada, no host ou no dispositivo e de onde será a invocação, do host ou do dispositivo. Os qualificadores do tipo de função são:

- `__device__` função executada no dispositivo e invocada somente do dispositivo.
- `__global__` declara a função como sendo um kernel e esta é executada no dispositivo, porém invocada somente do host.
- `__host__` função executada no host e invocada somente pelo host. Declarar uma função somente com o qualificador `__host__` é equivalente a declarar sem nenhum qualificador, em ambos os casos a função será compilada somente para o host. Porém, se usado junto com o qualificador `__device__`, a função será compilada para ambos, host e dispositivo.

Existem algumas restrições quanto aos qualificadores, sendo as mais relevantes:

- `__device__` e `__global__` não permitem recursão, não podem conter variáveis estáticas em seus corpos e não podem conter número variável de argumentos.
- `__global__` e `__host__` não podem ser usados juntos.
- Para uma invocação a uma função __global__ é necessário especificar uma configuração de execução, onde se define o número de blocos e threads que irão executar a função (kernel). Esta invocação é assíncrona e retorna antes que o dispositivo tenha completado sua execução. Porém, para confirmar através do host o término da execução de uma função no dispositivo pode-se usar a função `cudaThreadSynchronize()` após a invocação do kernel.

Além destes, existem também os qualificadores do tipo de variáveis, listados a seguir:

- `__device__` declara variável que reside no dispositivo. Pode ser utilizada juntamente com um dos qualificadores citados a seguir, para especificar a qual espaço de memória

a variável pertence. Se nenhum estiver presente, a variável pertencerá ao espaço de memória global. Possui o tempo de vida da aplicação e é acessível por todas as threads do *grid*.

- `__constant__`, opcionalmente usada com `__device__`, declara uma variável que reside na memória constante, possui o tempo de vida da aplicação e é acessível por todas as threads do *grid*.

- `__shared__`, opicionalmente usada com `__device__`, declara uma variável que reside na memória compartilhada do bloco de threads, possui o tempo de vida do bloco e é acessível por todas as threads do bloco.

Uma outra extensão da linguagem C é especificação da configuração de execução para uma chamada de uma função `__global__`. Essa configuração define as dimensões do *grid* e blocos que irão ser usados para executar a função no dispositivo. A especificação é feita inserindo uma expressão na forma `<<<` D_g`,` D_b`,` N_s`,` S `>>>` entre o nome da função e a lista de argumentos, onde:

- D_g é do tipo `dim3` — estrutura composta por três inteiros sem sinal, sendo eles x, y e z, respectivamente —, e determina a dimensão do *grid*, tal que $D_g.x \times D_g.y$ é igual ao número de blocos sendo executados; $D_g.z$ não é usado pois o *grid* é bi-dimensional.

- D_b é do tipo `dim3` e especifica a dimensão do bloco de threads, tal que $D_b.x \times D_b.y \times D_b.z$ é igual ao número de threads por bloco.

- N_s e S são argumentos opcionais e indicam, resumidamente, a quantidade de memória compartilhada alocada dinamicamente por bloco e os dados na forma de stream utilizado no kernel, respectivamente. Mais detalhes em [Cor06].

Como exemplo, a função declarada como: `__global__ void Func(float* param);` pode ser invocada desta maneira: `Func <<< Dg, Db >>>(param);` A função poderá falhar caso D_g ou D_b forem maiores que o máximo permitido para o dispositivo.

A quarta e última extensão são variáveis pré-definidas somente de leitura que especificam as dimensões do *grid* e do bloco, tal como os índices do bloco e da thread:

- `gridDim` variável do tipo `dim3` contendo a dimensão do *grid*.

- `blockIdx` variável do tipo `uint3` contendo o índice do bloco no *grid*.

- `blockDim` variável do tipo `dim3` contendo a dimensão do bloco.

- `threadIdx` variável do tipo `uint3` contendo o índice da thread no bloco.

Compilação com o NVCC

O compilador fornecido pela NVIDIA para a utilização da linguagem CUDA é o NVCC. Este fornece diversas opções de comandos e os executa invocando uma coleção de ferramentas as quais implementam os diferentes estágios da compilação. A tarefa básica do NVCC consiste em separar código do host do código do dispositivo e compilar este em uma forma binária, em um arquivo objeto *cubin* (*CUDA binary*). O código do host gerado tem como saída um código C para ser compilado usando outra ferramenta ou diretamente como um código objeto invocando o compilador do host durante o último estágio da compilação.

A aplicação pode ignorar o código do host gerado e carregar e executar o objeto *cubin* no dispositivo usando a API do controlador CUDA, ou pode ligar com o código de host gerado,

o que inclui o objeto *cubin* como global e contém a tradução da configuração de execução em um código de inicialização CUDA para carregar e executar cada kernel compilado. Tanto para o código do host quanto para o código do dispositivo é permitido C++, porém o dispositivo contém algumas limitações como, por exemplo, recursão e ponteiros de função. Uma descrição detalhada sobre o NVCC pode ser encontrada em [Cor06].

Existem duas diretivas de compilação introduzidas pelo compilador NVCC: `__noinline__` e `#pragma unroll`.

Uma função `__device__` é, por padrão, inline. A diretiva `__noinline__` é usada para notificar o compilado que, se possível, não expanda a função em linha.

O compilador, por padrão, estende os pequenos laços com um número conhecido de iterações. A diretiva `#pragma unroll` é usada para controlar a maneira de estender um laço. É colocada logo antes do laço e é aplicada somente neste laço. Opcionalmente é seguida de um número que especifica quantas vezes o laço deve ser estendido. Por exemplo:

```
#pragma unroll 5
```

for (**int** i = 0; i < n; i++) o laço será estendido 5 vezes. O programador deve assegurar que isso não irá afetar a corretude do programa (por exemplo, se `n` for menor que 5). Se o número 1 for especificado o compilador não estende o laço e se nenhum número for especificado o laço é completamente estendido caso o número de iterações for constante.

Vetores pré definidos

Os tipos `charn`, `ucharn`, `shortn`, `ushortn`, `intn`, `uintn`, `longn`, `ulongn`, `floatn`, onde $n \in \{1, 2, 3, 4\}$, são derivados dos tipos básicos de dados. Estes são definidos como estruturas e seus componentes são acessados através dos campos `x`, `y`, `z` e `w` respectivamente. Todos esses tipos contêm um inicializador na forma `make_<tipo>`. O exemplo cria um vetor do tipo `int2` com os valores `(x, y)`:

```
int2 make_int2(int x, int y);
```

Além desses, tem-se o tipo `dim3` que é um tipo de vetor de inteiros baseado no `uint3` e usado para especificar dimensões. Na definição de uma variável do tipo `dim3`, qualquer componente não especificado é inicializado com 1, por exemplo:

```
dim3 vardim(64);
```

A variável `vardim` armazena os valores 64, 1 e 1 em seus componentes `x`, `y`, e `z`, respectivamente.

4.3 Exemplo de código CUDA

Nesta seção é apresentado um exemplo de implementações da soma de duas matrizes em ambas arquiteturas. A soma de duas matrizes A e B de dimensão $n \times m$ resulta em uma matriz C de mesma dimensão com $C_{i,j} = A_{i,j} + B_{i,j}$, onde $0 \leq i < n$ e $0 \leq j < m$. Percebe-se o uso da função `add(...)` tanto em CPU quanto em GPU. Isso é permitido devido à presença dos qualificadores `__host__` e `__device__` na função. Dessa forma evita-se a replicação de código.

Programa C para CPU

```
__host__ __device__ float add(float a, float b)
{
   return a + b;
}
void op_matrix_cpu(float *a, float *b, float *c, int n, int m)
{
   int i, j, index;
   for (i=0; i<n; i++)
      for (j=0, index = i*m; j<m; j++, index++)
         c[index] = add(a[index], b[index])
}
void main()
{
   ...
   add_matrix_cpu(a, b, c, n, m);
}
```

Programa C para CUDA

```
__global__ void op_matrix_gpu(float *a, float *b, float *c, int n, int m)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x
   int j = blockIdx.y*blockDim.y + threadIdx.y
   if (i<n && j<m)
   {
      int index = i*m+j;
      c[index] = add(a[index], b[index])
   }
}
void main()
{
   ...
   dim3 dimBlock(blockSizeX, blockSizeY);
   int gridSizeX = ceil((float)n/blockSizeX);
   int gridSizeY = ceil((float)m/blockSizeY);
   dim3 dimGrid(gridSizeX, gridSizeY);
   add_matrix_gpu <<< dimGrid, dimBlock >>> (a, b, c, n, m);
}
```

4.4 Comentários finais

Neste capítulo apresentou-se o uso da GPU para propósito geral. Esse campo é conhecido como GPGPU e tem evoluído muito nos últimos anos. Nas GPUs tradicionais a computação de propósito geral é efetuada utilizando APIs gráficas, tais como OpenGL. Assim, para desenvolver, o programador deve entender de computação gráfica, como o funcionamento do pipeline gráfico. Além disso, o desempenho é afetado com ajustes referentes à API gráfica.

Com o surgimento das APIs tais como CUDA ou CTM o desenvolvimento de GPGPU se tornou mais simples. Maior flexibilidade de programação, uso da linguagem C e maior largura de banda das memórias são características da API CUDA. Os multiprocessadores do hardware são representados por blocos e os respectivos processadores executam threads pertencentes ao bloco. Devido à facilidade de expansão de multiprocessadores, as GPUs atuais tentem a crescer cada vez mais.

CAPÍTULO 5

Implementação em CUDA

5.1 Introdução

Neste capítulo é apresentada a implementação em GPU do componente de AS denominado motor de física, Seção 5.3. As principais funções do motor de física são: a determinação das forças de restrições (originadas de junções e contatos) que atuam nos corpos rígidos de uma cena, em um dado instante de tempo, e a integração da equação de movimento para determinação do estado atualizado de cada corpo rígido, em função das forças e da massa do corpo. Outro componente cuja implementação em paralelo também é mostrada nesse capítulo é o detector de colisões, Seção 5.2. Este é responsável pela determinação dos pontos de contato entre os corpos rígido da cena durante a simulação e fornecimento destes ao motor de física.

A implementação foi elaborada para GPUs com suporte à arquitetura CUDA. Nesta, prezou-se umas das principais características da arquitetura CUDA que é o uso da linguagem C para desenvolvimento dos kernels. Com isso, classes, estruturas e métodos são implementados com o objetivo de serem utilizados tanto em CPU quanto no dispositivo. As implementações do detector de colisão e do motor de física para CPU e GPU são baseadas em um mesmo algoritmo; portanto, na medida do possível, os códigos foram elaborados para serem utilizados em ambas arquiteturas.

A Figura 5.1 mostra um diagrama de classes UML com as principais classes do motor em CUDA.

A classe `CUDAEngine` é derivada de `Engine` (uma explicação mais detalhada dessa classe pode ser encontrada em [dS08]) e determinada a organizar as invocações dos métodos responsáveis por cada etapa do motor de física em CUDA. Para que a GPU tenha acesso às listas de corpos, formas e junções da cena, essas devem ser alocadas na memória do dispositivo. Para isso a classe `CUDAEngine` declara um objeto da classe `CUDARigidBodySpace`, um da classe `CUDAShapeSpace` e um da classe `CUDAJointSpace`, instanciados no construtor.

Um objeto da classe `CUDARigidBodySpace`, a qual deriva de `CUDARigidBodyStream`, representa um espaço de corpos rígidos na memória da GPU, ou seja, a lista de corpos rígidos. Objetos dessa classe são responsáveis por alocar/desalocar e transferir a lista de corpos rígidos para memória do dispositivo. O construtor da classe toma como argumento uma lista de corpos rígidos residente na memória do computador, a associa com o atributo `input`, e cria uma lista com a mesma dimensão na memória da GPU. Cabe ao método

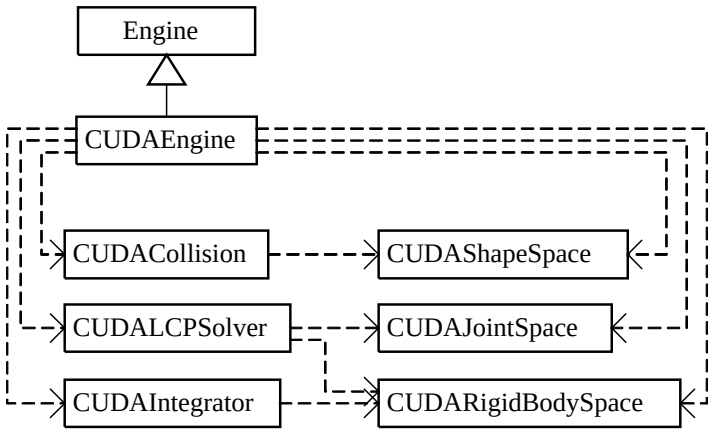


Figura 5.1: Diagrama UML do motor de física em CUDA.

`update()` invocar a função de cópia (transferência) da lista da CPU para GPU.

Como a lista de corpos rígidos possui várias listas, cada uma representando um atributo do corpo, várias transferências são realizadas. Porém para que a comunicação não se torne o gargalo do sistema, foram criadas marcas (ou *flags*) de modificação para cada lista de atributo. Dessa forma, o método `CUDARigidBodySpace::update()` verifica essas marcas e somente as listas com modificações são transferidas. Nessa versão, uma modificação é considerada quando qualquer mudança ocorrer na lista. Então, por exemplo, a situação de alteração da força e massa de alguns objetos, gera uma modificação nas listas de força e inversa da massa, portanto há a transferência de ambas. Se um corpo for criado/eliminado, então todas as listas são modificadas, portanto ocorre a transferência de todas as listas representantes dos corpos rígidos.

Os objetos da classe `CUDAShapeSpace` e `CUDAJointSpace` representam, respectivamente, um espaço de formas e um espaço de junções na memória do dispositivo. Essas classes possuem a mesma funcionalidade da classe `CUDARigidBodySpace`, diferenciando apenas do método `update()`.

O método `CUDAShapeSpace::update()` é responsável por enviar a lista de formas para a GPU. A lista de formas é dividida em duas listas, sendo uma com os dados principais da forma (`shapes`) e outra contendo as *poses* globais (`globalPoses`). Essa divisição ocorreu pois as *poses* das formas mudam constantemente, portanto devem ser atualizadas a cada frame; já os formas (`shapes`) são atualizadas somente quando há modificação, criação ou eliminação de formas. Nesse caso, a marca de modificação é ajustada e a transferência de toda a lista é realizada.

Na maioria das simulações não há constante alteração/criação/destruição de corpos (atores) e/ou formas, sendo o método adotado para atualização, descrito acima, suficiente. Porém, para simulações com constante modificação dos corpos, o ideal é a implementação de um gerenciador de memória no dispositivo.

O método `CUDAShapeSpace::update()` é responsável por enviar a lista de junções para a GPU. Essa lista é dividida em lista de restrições e índices de junções. A lista enviada ao dispositivo é composta apenas por restrições referentes às junções, com exceção do contato, pois essas serão criadas diretamente no dispositivo. Portanto, havendo alguma junção na cena, o método `CUDAJointSpace::update()` realiza a transferência de toda a lista para o dispositivo.

O método `CUDAEngine::run()` sobrecarrega o método virtual de mesmo nome da classe `Engine`. Este é responsável por atualizar as listas no dispositivo e invocar o detector de co-

lisões (`collision`), solucionador de PCL (`lcpSolver`) e solucionador de EDO (`Integrator`). Esses atributos são objetos das classes `CUDACollision`, `CUDALCPSolver` e `CUDAIntegrator`, respectivamente, instanciados pelo construtor. Um pseudocódigo do método `run()` é apresentado:

```
 void CUDAEngine::run()
{
  bodies.update();
  shapes.update();
  joints.update();

  collision.setInput(shapes,...);
  contacts = collision.run();

  lcpSolver.setInput(bodies, joints, contacts,...);
  lcpSolver.setProperties(...);
  constraintVel = lcpSolver.run();

  integrator.setInput(bodies);
  integrator.run(constraintVel,...);
}
```

Ao fim da execução desse método, os estados dos corpos rígidos estão atualizados e devem ser retornados à CPU para que seus respsctivos atores possam ser renderizados. As informações necessárias para a renderização são posição e orientação do corpo rígido, porém essas variáveis estão embutidas, juntamente com velocidade e aceleração, na lista que representa os estados de cada corpo. Portanto, a lista de estados inteira é retornada à CPU. O método `CUDARigidBodySpace::copyStateToInput()` é responsável por invocar a função que realiza a transferência.

5.2 Implementação do detector de colisões em CUDA

Esta seção apresenta alguns detalhes da implementação do detector de colisões em GPU utilizando CUDA. No Capítulo 3 são apresentados alguns conceitos utilizados na detecção de colisão e a implementação do detector de colisões em CPU. Portanto, é importante a compreensão daquele antes de iniciar a leitura dessa seção, pois a implementação descrita a frente é baseada na apresentada no Capítulo 3. Como visto, a implementação é dividida em duas fases: fase geral e fase.

O detector de colisões em CUDA é implementado através da classe `CUDACollision`, a qual deriva da classe `Collision` de acordo com o diagrama da Figura 5.2. O método `CUDACollision::run()` executa as chamadas aos kernels que implementam as etapas da detecção de colisão.

Na classe `Collision` existe um atributo `data` que contém os dados utilizados pelo detector de colisões em CUDA. Porém esses dados estão na memória da CPU. Para que a GPU tenha acesso a esses dados uma cópia é enviada à GPU, e o endereço desta é armazenado no atributo `CUDAdata` da classe `CUDACollision`. Isso é feito para evitar o envio dos dados via parâmetros dos kernels. Dessa forma somente o ponteiro `CUDAdata` é enviado, Figura 5.3.

5.2.1 Fase geral

A implementação da fase geral em paralelo é baseada em [Gra07] e utiliza o método da subdivisão espacial apresentado na Seção 3.2.

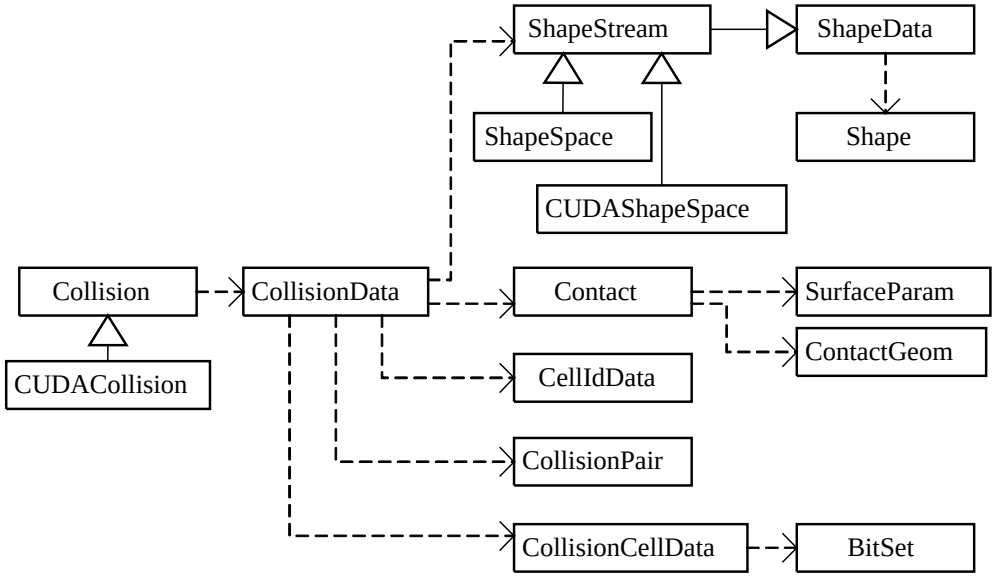


Figura 5.2: Diagrama UML do detector de colisão em CUDA.

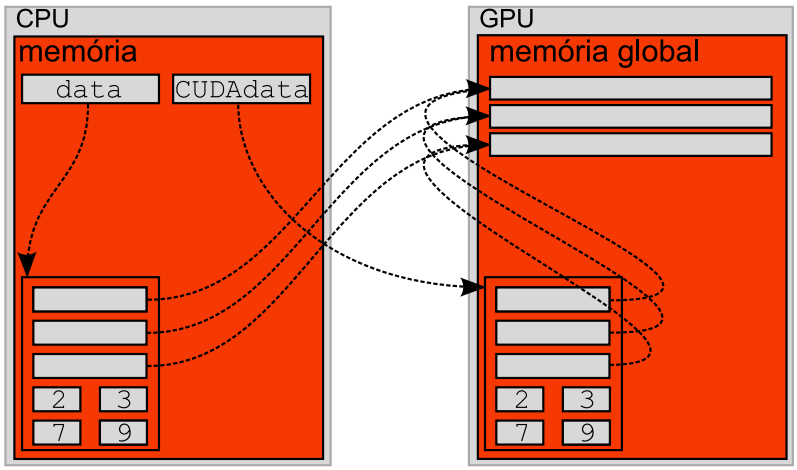


Figura 5.3: Esquema para organização dos dados em memória utilizados em CUDA.

Para a implementação do detector de colisões em CUDA, é utilizado o mesmo método implementado na forma seqüencial descrito na Seção 3.4, porém para usufruir da capacidade de processamento em paralelo das GPUs as etapas são implementadas para execução em paralelo. A implementação da fase geral seqüencial é dividida basicamente em três etapas: criação do vetor de identificadores da célula, ordenação do vetor e determinação das células de colisão e conseqüentemente criação dos pares de colisão.

A etapa de determinação das células de colisão e criação dos pares de colisão em seqüencial é feita de forma unificada, na qual durante a determinação das células de colisão já são criados os pares de colisão. Na implementação em paralelo essa etapa será particionada, pois após a finalização da determinação das células de colisão que é iniciado o processo de criação dos pares de colisão.

A primeira etapa consiste na criação do vetor de identificadores de célula. Essa criação pode ser feita de forma independente entre as formas, portanto cada thread da GPU é responsável pela criação das célula R e células I de uma forma. No caso em que há mais formas do que threads, cada thread deve manipular diversas formas. Mais precisamente a thread j

do bloco i trata os objetos $iB + j, iB + j + nT, iB + j + 2nT$, e assim por diante, onde B é o número de threads por bloco, T é o número total de threads e n é o número de objetos.

No armazenamento é reservado a quantidade máxima de células (uma célula R e $2^3 - 1$ células I) para cada forma da lista de formas. Se uma forma contém um número menor que $2^3 - 1$ de células I, então os identificadores da célula extras são marcados com `0xffffffff` para indicar que não são válidos. Cada thread armazena a quantidade de células válidas criadas em um vetor, para que posteriormente se saiba o total de células válidas criadas. O total de células válidas é computado somando os elementos do vetor de quantidades.

A melhor maneira de somar diversos valores de um vetor em paralelo é através de uma técnica chamada *redução*. Em CUDA, as threads de um bloco podem cooperar através da memória compartilhada, resultando em uma eficiente implementação da redução em paralelo. Resumidamente, é distribuído uma porção do vetor para cada bloco de threads. Primeiro é calculado as quantidades referentes a cada bloco. Esses valores são armazenados na memória global para cada bloco. Uma vez finalizado esse processo, um único bloco é invocado para, da mesma forma, somar os resultados de todos os blocos. A implementação da *redução* em CUDA é baseada em [MH07].

A Figura 5.4 ilustra o resultado da construção do vetor de identificadores de célula.

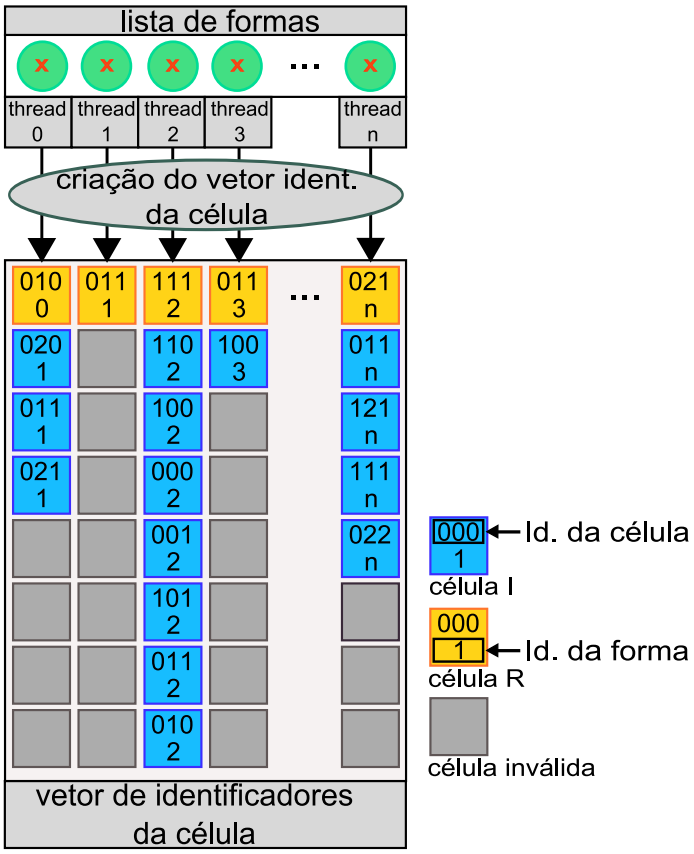


Figura 5.4: Conteúdo inicial de um vetor de identificadores de célula.

Em seguida, o vetor de células é ordenado pelo identificador da célula. A ordenação deve ocorrer da mesma forma da implementação seqüencial: células com o mesmo identificador devem ser ordenadas pelo seu tipo (células R antes de células I), portanto duas chamadas ao processo de ordenação.

Ao contrário da implementação seqüencial, onde alterou-se a forma de armazenamento para que o processo de ordenação fosse executado somente uma vez, nesse caso é utilizado

uma composição dos valores a considerar na ordenação em um só valor. Durante a criação das células é calculado um identificador `cellIdHP` somente para a ordenação da seguinte maneira:

`cellIdHP = (cellId << 1) + (hCell ? 0:1);` onde, `cellId` é o identificador da célula e `hCell` se verdadeiro, representa uma célula R, caso contrário, representa uma célula I. Dessa forma tem-se, por exemplo, identificador da célula 37 (100101), então as células R e células I terão, respectivamente, como identificador para ordenação 74 (1001010) e 75 (1001011).

O algoritmo de ordenação em paralelo utilizado para ordenar o vetor de células é o *radix sort* [Gra07]. O código utilizado para a implementação foi extraído dos códigos de demonstração desenvolvido pelo NVidia Corporation, podendo ser acessado em [Cor06].

A Figura 5.5 mostra o processo de ordenação aplicado ao vetor de células da Figura 5.4.

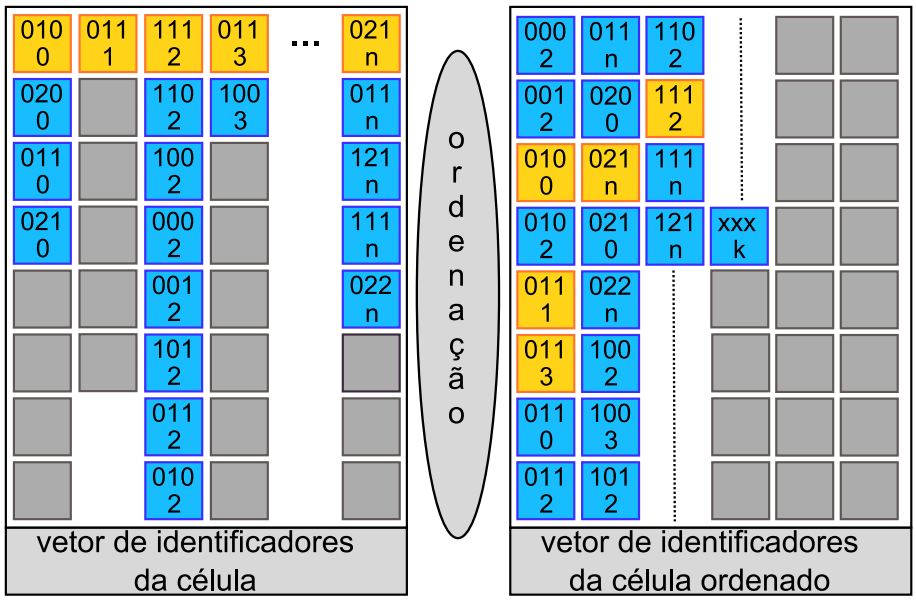


Figura 5.5: *Radix sort* aplicado ao vetor ID da célula da Figura 5.4.

Uma vez ordenado o vetor de células, é criada a lista de células de colisão. Em seguida, as células de colisão são executadas em paralelo, gerando os pares de colisão a serem analisados na fase exata. A lista de células de colisão é composta por elementos organizados através da seguinte estrutura:

```
struct CollisionCellData
{
   uint start;
   uint nHCell;
   uint nPCell;
   BitSet bPairs;
}
```

onde, `start`, é o índice para a célula (do vetor de células) inicial da célula de colisão, `nHRCell` e `nPCell` armazenam a quantidade de células R e a quantidade de células I na célula de colisão, respectivamente, e `bPairs` é um vetor de bits, cuja utilização é descrita mais adiante. `nHCell` + `nPCell` representam o total de células que compõem a célula de colisão.

A criação da lista de células de colisão é feita conforme descrito a seguir. O vetor de células ordenado é percorrido em busca de trocas de identificadores de célula, situação que marca o fim de uma célula de colisão e o início de outra. Para paralelizar essa tarefa são

distribuídos segmentos de tamanho igual, um para cada thread de cada bloco, do vetor de identificadores de células. Cada thread procura por transações, porém como a thread deve determinar o início e o fim de cada célula de colisão, a procura deve: extender-se para além do fim do segmento até um final de transação e ignorar a primeira transação (assumindo que esta foi avaliada pela thread precedente). A única exceção existente para o último caso é destinada à primeira thread do primeiro bloco, a qual deve tratar a primeira transação do início de seu segmento, já que não existe thread precedente a ela. Devido a soma das células válidas anteriormente, as células inválidas serão ignoradas.

De fato, o passo acima é executado duas vezes. Na primeira, é contado o número de objetos de cada célula de colisão e isso é convertido em deslocamentos no vetor de armazenamento através de uma soma prefixa em paralelo, cuja implementação foi baseada em [MH07]. Uma vez estabelecidas as posições de armazenamento, na segunda vez são criadas as entradas para cada célula de colisão no vetor de células de colisão, Figura 5.6. A quantidade de células de colisão criadas é obtida através da soma prefixa executada anteriormente.

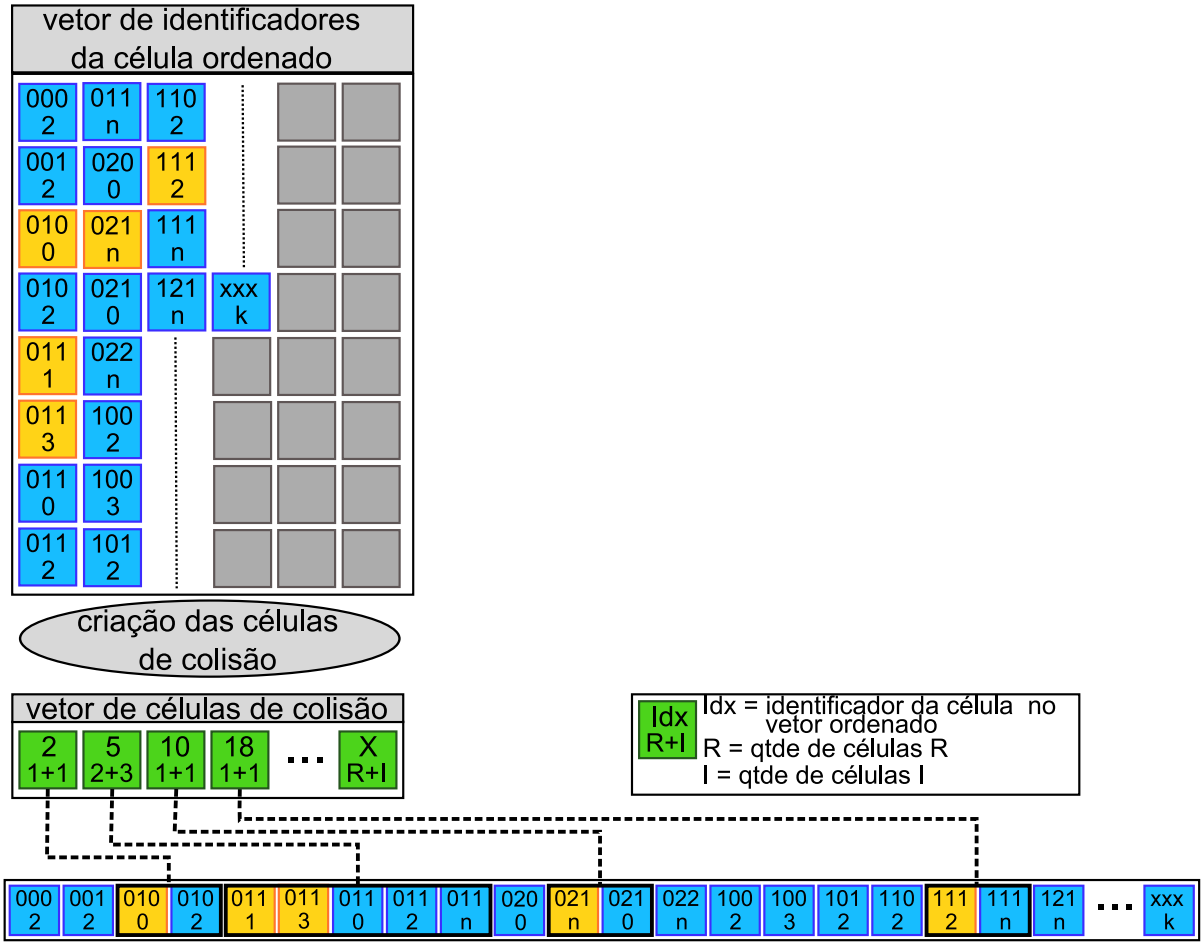


Figura 5.6: Criando as células de colisão.

Na próxima etapa as células de colisão são atravessadas, sendo cada uma processada por uma thread. Neste passo são realizados os testes de colisão e criados os pares de colisão entre formas cujas esferas envolventes se sobrepõem.

Replicando o que foi escrito na implementação seqüencial vista na Seção 3.4, cada célula de colisão é atravessada e os testes de colisão são efetuados somente entre células R × R e células R × I, ignorando os testes das células I × I. Antes de aplicar o teste de colisão entre as células R × I deve-se fazer a seguinte verificação: seja L o rótulo da célula R e L' o rótulo

da célula I; se $L' < L$ e o objeto da célula R intercepta L', então o teste é ignorado.

Resumindo, um teste de colisão é aplicado entre todas as células R × R, pois trata-se dos objetos cujos centróides pertencem à célula, e entre as células R × I, caso não seja verdadeiro o teste acima.

No caso de interseção entre as esferas envolventes das formas pertencentes ao teste de colisão, é criado um par de colisão entre as formas. Como não se sabe, a priori, a quantidade de pares que serão gerados por célula de colisão (e esse número é bastante variado), pode-se executar o algoritmo duas vezes, sendo a primeira responsável por contar o número de pares de colisão e a segunda encarregada de criar os pares de colisão nos locais determinados para cada thread. Esses locais são determinados após uma soma prefixa em paralelo (a mesma utilizada na etapa anterior) do vetor de quantidades de pares de colisão por célula de colisão. Feito isso cada thread conhece o local destinado à criação de seus pares de colisão.

A idéia de efetuar duas vezes os testes de colisão é pouco eficiente, portanto o processo foi otimizado com a criação de um vetor de bits (`bPairs`) de tamanho fixo que armazena, para cada par de formas, durante o primeiro passo na execução dos testes de colisão, 1 se o par deverá ser criado e 0 caso contrário, Figura 5.7. Em seguida, no segundo passo, o único teste a ser realizado é do bit referente ao par (e não mais o teste de colisão), diminuindo, dessa forma, a quantidade de processamento por thread nesse passo. Isso pode ser feito devido à seqüencia de testes, que são entre as células R × R e células R × I, ser a mesma em ambos os passos, Figura 5.7.

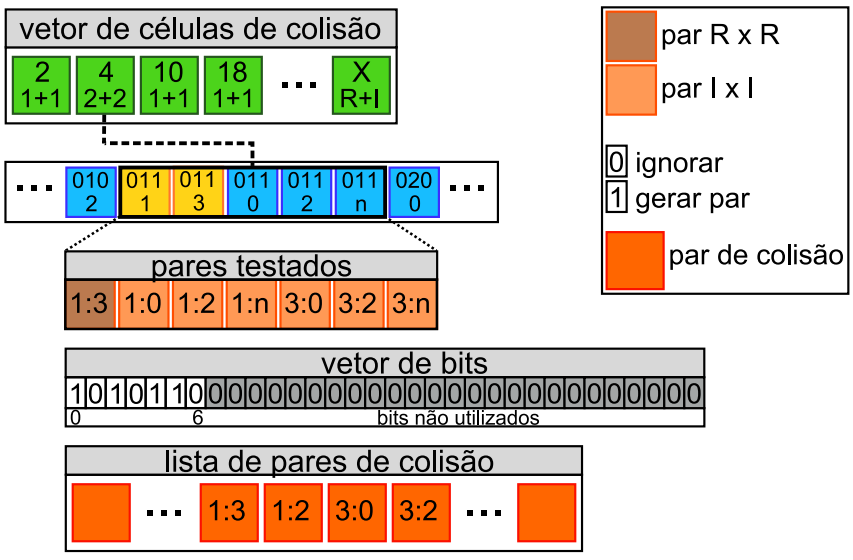


Figura 5.7: Atravessando a célula de colisão.

Executando e analisando essa etapa percebeu-se uma diferença na eficiência do algoritmo, principalmente quando as formas estavam muito próximas umas das outras. Isso ocorre devido à má distribuição dos testes de colisão entre as threads. Enquanto uma célula de colisão contém diversas formas para realizar os testes outras contêm somente algumas, causando assim esse desbalanceamento.

Uma forma não complexa encontrada para melhorar o balanceamento foi alocar mais threads para executar os testes de uma célula de colisão. O número de threads é relativo à quantidade de células pertencentes à célula de colisão, mais precisamente igual ao número de células R. Como visto anteriormente, os testes de colisão são efetuados entre uma célula R e uma célula R ou I, sendo assim cada thread é responsável pela realização dos testes entre uma célula R e as demais (algumas células R, para que não haja duplicação no teste, e todas

células I). Como cada forma gera uma e somente uma célula R, a quantidade total de threads alocadas é igual a quantidade de formas. A Figura 5.8, baseada na Figura 5.7, ilustra a divisão dos testes entre as threads.

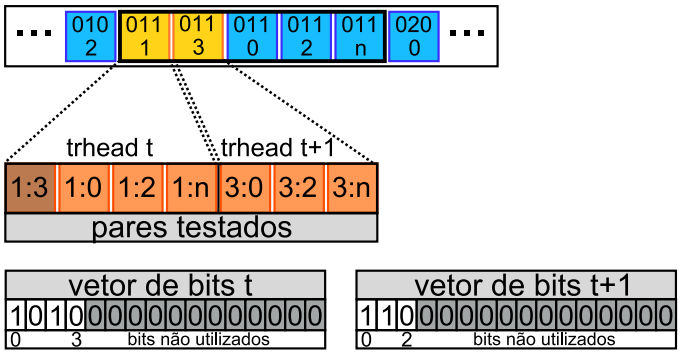


Figura 5.8: Atravessando a célula de colisão com mais de uma thread.

Com a divisão dos testes de colisão entre mais threads o vetor de bits continua sendo utilizado, porém cada thread possui seus bits e não mais um vetor de bits para a célula de colisão. A dimensão do vetor de bits, nesse caso, deve ser igual a quantidade máxima de células R e células I pertencentes à uma célula de colisão. Na implementação foi considerado um vetor de 256 bits.

A parte da fase geral em paralelo que considera as formas *unbounded* nas colisões consiste no seguinte processo: cada thread, associada a uma forma *bounded*, percorre a lista de formas *unbounded* efetuando os testes de colisão (no caso de não ser um plano) e criando os pares de colisão entre a forma *bounded* e as formas *unbounded*. Cada thread pode criar até u pares, onde u é o número de formas *unbounded* existentes na simulação. Portanto esse espaço é reservado a cada thread para armazenamento de seus pares. Porém, pode ocorrer das esferas envolventes não se interceptarem. Nesse caso o par de colisão é invalidado com o valor 0 em ambas as formas do par e é ignorado durante a análise na fase exata.

Nesse momento, a lista de possíveis de pares de colisão está preenchida e disponível para uma análise detalhada na fase seguinte. Dessa forma é finalizada a fase geral do detector de colisões. A análise dos resultados desta fase é apresentada no Capítulo 6.

5.2.2 Fase exata

Nesta seção é discutida a implementação em paralelo da fase exata do detector de colisões. A fase exata tem como objetivo realizar o cálculo de colisão precisa entre dois objetos, determinando propriedades geométricas do contato, tais como ponto de contato, normal no ponto, profundidade de penetração, que estão organizados na estrutura `Contact`. Esta etapa é executada logo após a fase geral do detector de colisões e, portanto, recebe a lista de possíveis pares de colisão provinda desta.

A fase exata em paralelo é implementada pelo método `CUDACollision::runNarrow()`. Dada a lista de possíveis colisões contendo pares de objetos, cada par é atribuído a uma thread, a qual executa o mesmo método da implementação seqüencial `Collision::collide(Contact* c, Shape*, Shape*)`. Esse método invoca o método adequado para a determinação dos pontos de contato da colisão de acordo com os tipos das formas do par de colisão analisado.

As funções de determinação dos contatos entre as formas são as mesmas das implementadas para o modo seqüencial, com exceção da função caixa-caixa. A ausência do método caixa-caixa ocorre pois a GPU não suporta a implementação encontrada ([Smi00]) para esse

método, causando erros de compilação no NVCC. Uma solução para esse caso é a implementação do algoritmo GJK (Gilbert-Johnson-Keerthi) apresentado em [vdB99], o qual provê detecção de colisão entre objetos convexos. Essa solução não foi adotada nesse trabalho e é sugerida como trabalho futuro.

Analisando a complexidade e o volume de cálculos executado por cada função, percebe-se uma diferença principalmente naquelas onde o primitivo caixa se envolve. Isso significa que algumas threads executarão mais código que outras, resultando que o ganho da GPU, com relação à CPU em uma implementação seqüencial, em certos casos, seja mínimo. Porém é levado em consideração que os pares de colisão estão armazenados na memória do dispositivo (determinados na etapa anterior) e que os pontos de contatos determinados nessa etapa já estarão também na memória do dispositivo prontos para serem utilizados na próxima etapa. Portanto, são eliminadas duas comunicações entre CPU e GPU, tornando-se assim um método viável.

As funções de determinação dos contatos podem gerar quantidades diferentes de contato por par de colisão. Como exemplo, na colisão entre duas esferas um contato é criado, e na colisão entre caixa e plano até três contatos podem ser criados. Decorrente disso, é disponibilizado para cada thread um espaço para a criação de, no máximo, três contatos. Porém, "buracos"ocorrem devido aos contatos não utilizados, ou inválidos. Os contatos inválidos são originados das colisões que geram um número de contatos menor que três, podendo ser zero no caso da não interseção das formas, e dos pares de colisão inválidos criados pela fase geral, gerando nesse caso, três contatos inválidos.

Contatos inválidos geram processamento extra no motor de física, por isso devem ser eliminados. A eliminação dos contatos inválidos consiste em:

- Armazenar em um vetor a quantidade de contatos válidos criados durante a determinação dos contatos. Essa quantidade varia de 0 a 3.

- Efetuar uma soma prefixa no vetor de quantidades. Além da soma prefixa é determinado o total de contatos válidos.

- Copiar os contatos válidos para uma outra lista. Cada grupo de três contatos é atribuído a uma thread, e esta é responsável por copiar seus contatos válidos para o destino determinado pela vetor de quantidades após a soma prefixa.

A criação dos contatos e a eliminação dos contatos inválidos são ilustrados na Figura 5.9. A análise dos resultados é apresentada no Capítulo 6.

5.3 Implementação do motor de física em CUDA

A fim de determinar as forças de restrições decorrentes das junções e contatos em um dado instante de tempo para cada corpo rígido da cena, a tarefa do motor de física é dividida basicamente em três etapas, as quais consistem em: montagem do sistema correspondente ao problema de complementaridade linear (PCL) e resolução do PCL, aplicação de forças e integração da equação de movimento.

A implementação em paralelo foi baseada na implementação seqüencial desenvolvida no AS [dS08]. O algoritmo para resolução do PCL implementado em AS, baseado em [Erl04] e [Cat05], utiliza o método SOR com projeção.

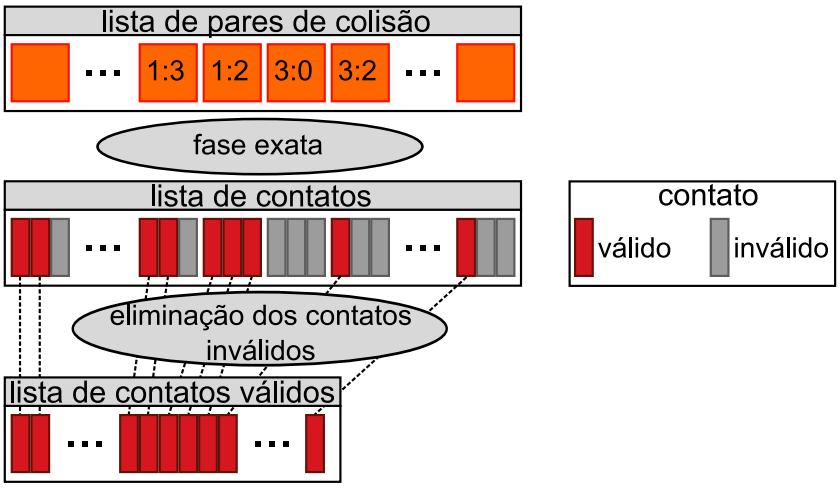


Figura 5.9: Criação dos contatos e eliminação dos contatos inválidos.

5.3.1 Solucionador de PCL

O diagrama de classes UML contendo as principais classes do solucionador de PCL em CUDA é ilustrado na Figura 5.10.

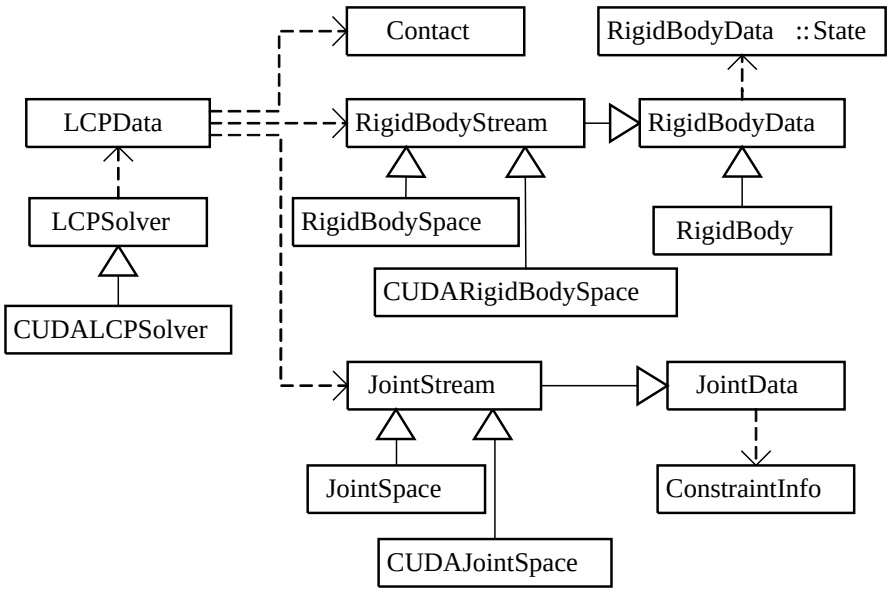


Figura 5.10: Diagrama UML do solucionador de PCL em CUDA.

A classe `LCPSolver`, da qual `CUDALCPSolver` deriva, declara um atributo `data` que contém os dados necessários referentes ao solucionador de PCL em CUDA, dados esses que estão na memória da CPU. Para que a GPU tenha acesso aos dados uma cópia é enviada ao dispositivo. O endereço dessa cópia é armazenado no atributo `CUDAdata` da classe `CUDALCPSolver`. Isso é feito para evitar o envio dos dados via parâmetros dos kernels. Dessa forma somente o ponteiro `CUDAdata` é enviado, Figura 5.3.

Da mesma forma que a implementação seqüencial, a implementação do solucionador de PCL em paralelo contém uma fase inicial na qual são criadas as restrições referentes à lista de contatos, e uma principal, na qual é montado e solucionado o problema do PCL.

A criação das restrições oriundas dos contatos é realizada de forma independente entre

os contatos, portanto cada thread é responsável pela criação das restrições (três) de um contato. A thread j do bloco i trata o contato $iB + j$, onde B é o número de threads por bloco. As restrições criadas por essa thread serão armazenadas na lista de restrições na posição $numberOfCJoints + (iB + j) * 3$, onde $numberOfCJoints$ é a quantidade de restrições referentes as junções com exceção do contato, criadas pela CPU.

Como visto na Seção 2.2, a criação das restrições referentes à um contato é feita através do método estático `LCPSolver::ComputeContactInfo(contact, data)`. Esse método será utilizado também em GPU, por isso possui os qualificadores `__host__` e `__device__`, o que permite a utilização em ambas arquiteturas. O pseudocódigo do kernel executado por cada thread é mostrado a seguir:

```
__global__ void assemblyContactsInfoKernel(CUDAdata)
{
  int contactIdx = blockIdx.x * blockDim.x + threadIdx.x;
  if (contactIdx >= numberOfContacts) return;

  LCPSolver::ComputeContactInfo(contacts[contactIdx], CUDAdata);
}
```

A invocação do kernel é feita através do comando
`assemblyContactsInfoKernel<<<nbBlocks, nbThreads>>>(CUDAdata);`
onde `nbBlocks` e `nbThreads` são a quantidade de blocos e threads por bloco, respectivamente, alocados para a tarefa. Esses valores devem ser calculados em função da quantidade de contatos. Através dessas quantidades não se define a quantidade exata de threads (que é igual ao número de contatos), mas sim uma quantidade multipla de `nbThreads`, que para processar todos os contatos deve ser maior que o número de contatos. Dessa forma, algumas threads estão livres de processamento, porém, nessas, o kernel é executado normalmente e devido a isso tem-se a verificação do índice da thread com a quantidade de contatos. Criada a lista de restrições, é realizado a montagem dos dados e resolução do PCL.

O pré-processamento é executado independentemente para cada corpo rígido. Portanto, para torná-lo paralelo basta atribuir cada corpo a uma thread. Novamente a thread j do bloco i é responsável pelo corpo rígido $iB + j$.

O método estático
`LCPSolver::ComputePreProcess(contact, data)` realiza o pré-processamento para um corpo rígido, e possui o qualificador necessário para ser utilizado em CUDA. O pseudocódigo do kernel do pré-processamento é:

```
__global__ void PreProcessKernel(CUDAdata)
{
  int rbIdx = blockIdx.x * blockDim.x + threadIdx.x;
  if (rbIdx >= numberOfRigidBodies) return;

  LCPSolver::ComputePreProcess(rb[rbIdx], CUDAdata);
}
```

A invocação do kernel é feita através do comando
`PreProcessKernel<<<nbBlocks, nbThreads>>>(CUDAdata);`

O processo de inicialização, onde são montados os vetores e matrizes que compões o PCL, é realizado de maneira independente entre as restrições. O paralelismo desse processo é feito com cada thread se responsabilizando por uma restrição, de tal forma que a thread j do bloco i é responsável pelo processamento da restrição $iB + j$. A inicialização de cada restrição é processada pelo método estático `LCPSolver::ComputeSORLCPInit(constraint, data)`,

que também possui o qualificador para a utilização em GPU. O pseudocódigo do kernel de inicialização é:

```
__global__ void SORLCPInitKernel(CUDAdata)
{
  int cIdx = blockIdx.x * blockDim.x + threadIdx.x;
  if (cIdx >= numberOfConstraint)
    return;

  LCPSolver::ComputeSORLCPInit(constraint[cIdx], CUDAdata);
}
```

A invocação do kernel é feita através do comando
`SORLCPSolverKernel<<<nbBlocks, nbThreads>>>(CUDAdata)`.

O último e mais importante passo é a execução do método SOR. O sistema a ser resolvido é formulado em função da lista de restrições. Cada restrição representa uma linha do sistema, portanto uma junção representa um bloco de linhas. A técnica de paralelismo utilizada é a resolução dos blocos, compostos por mais de uma junção, em paralelo e as linhas internas de cada bloco de forma seqüencial. Isso é semelhente a adotar o método de Jacobi ,Equação (2.58), entre os blocos e o método do SOR, ou Gauss-Seidel, Equação (2.59), internamente em cada bloco.

O método estático `LCPSolver::ComputeSORLCPSolver(joint, data)` processa uma iteração do método SOR para o conjunto (bloco) de restrições da junção. A fim de ilustrar o processamento assume-se inicialmente uma junção por thread, ou seja, o kernel executado em uma thread invoca o método para a respectiva junção `iteration` vezes, como ilustrado no pseudocódigo abaixo. A thread j do bloco i é responsável pelo processamento da junção $(iB + j)$.

```
__global__ void SORLCPSolverKernel(CUDAdata)
{
  int jIdx = blockIdx.x * blockDim.x + threadIdx.x;
  if (jIdx >= numberOfJoints)
    return;

  for (int i = 0; i < MAX_ITERATIONS; i++)
    LCPSolver::ComputeSORLCPSolver(joint[jIdx], CUDAdata);
}
```

Durante o processo iterativo otimizado do SOR visto na Seção 2.2.2, percebe-se em cada iteração a leitura e escrita no vetor V', o qual representa as velocidades (linear e angular), decorrentes das restrições, de cada corpo. Se cada corpo for associado a apenas uma junção, o processo ocorreria normalmente, pois as velocidades em ambos os corpos da junção seriam acessadas por somente uma thread. Porém, o que acontece na prática são corpos referenciados por várias junções, como, em um exemplo simples, vários contatos em um mesmo objeto. Dessa forma, podem ocorrer acessos simultâneos às velocidades do corpo, gerando leituras e escritas concorrentes.

Sobre escrita concorrente entre threads, a arquitetura CUDA garante somente que pelo menos uma thread irá escrever. Ou seja, dentre as threads que atualizam um mesmo corpo simultaneamente, pode ocorrer de apenas uma ter seu resultado computado.

A probabilidade de ocorrer escrita concorrente em um corpo está relacionada com a quantidade de junções às quais o corpo está associado e a quantidade de junções processadas por cada thread. Com o objetivo de diminuir essa probabilidade a idéia inicial foi aumentar o número de junções por thread, por exemplo três. Então, dado uma quantidade total de

junções, a quantidade de junções por thread e a quantidade de threads por bloco, determina-se a quantidade de blocos a serem executados na GPU. Como essa quantidade é variável, o processamento de alguns blocos pode ocorrer de forma seqüencial dependendo da capacidade do dispositivo. Com isso, é gerado um erro considerável na precisão dos resultados, pois uma parte do sistema está sendo solucionada sem a consideração integral deste. Portanto, a quantidade de blocos deve ser fixa, tal que utilize toda a capacidade da GPU.

A GPU disponibiliza uma quantidade de até 512 threads por bloco, que teoricamente são processadas simultaneamente, e até 65536 blocos. É assumido, para essa etapa, uma quantidade de 32 threads por bloco e 16 blocos, totalizando 512 threads[1] executadas em paralelo. Em uma simulação, tem-se facilmente uma quantidade de junções superior a essa. Como exemplo, é apresentada na Figura 5.11 um quadro de uma simulação contendo mais de nove mil junções (de contato).

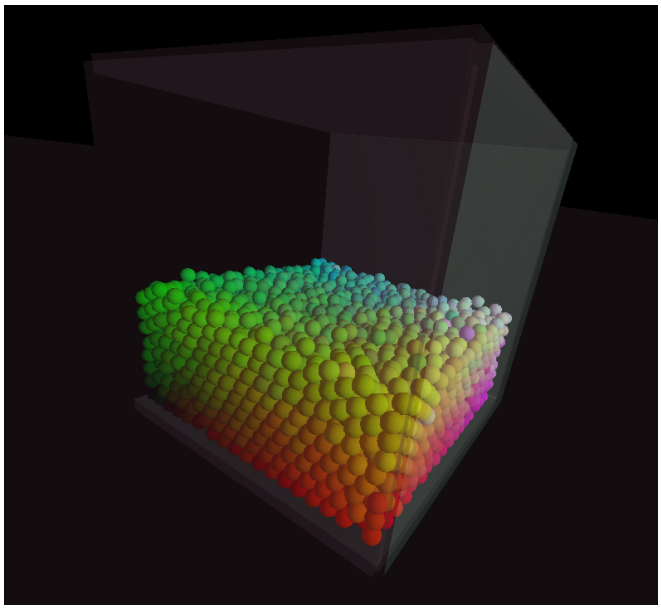


Figura 5.11: Cubo com 3375 esferas e 9376 contatos.

A sugestão é computar a quantidade de junções a serem processadas em uma thread como m/n, onde m é a quantidade de junções e n um número fixo (512) representando a quantidade de threads. Assim, no exemplo dado, esse número seria, na média, 18.31 junções por thread, ou seja, na melhor divisão, das 512 threads, 160 computariam 19 junções e 352 threads 18 junções, mantendo um balanceamento entre o processamento das threads. Quando o número de junções em uma simulação for menor que 512, volta-se a ter somente uma junção por thread e, então, problemas maiores de concorrência. Portanto, é estabelecido um número mínimo de três junções por thread. Com isso, simulações com um número razoável de junções (< 1536) podem ocasionar processadores livres no dispositivo, porém para essas simulações a diferença no desempenho não afeta a taxa de quadros por segundo, conforme verificado nos exemplos apresentados no Capítulo 6.

Além da organização do processamento entre as threads, o desempenho das aplicações executadas em CUDA é altamente dependente da organização dos acessos à memória. Assim como na CPU, as memórias com maior capacidade possuem maior custo computacional

[1] O número de blocos escolhido é tal que satisfaz a quantidade de blocos executados em paralelo na NVidia GeForce 8800GTX. O número de threads é o mínimo para manter a eficiência sugerido por [Cor06]. Isso mantém a GPU com a capacidade máxima de processamento.

para o acesso, e as mais rápidas são limitadas em dimensão. Estudou-se o uso da memória compartilhada, pois possui um custo de acesso muito inferior comparada à memória global. Porém, essa memória pertence a um bloco e só pode ser acessada pelas threads pertencentes a ele, e sua capacidade é 16 KB por bloco. Essa capacidade deve ser dividida entre as 32 threads do bloco, resultando em 512 B para cada thread. Assumindo uma thread com 30 junções, sendo cada junção com 6 restrições, resta menos de 3 B por restrição. Portanto, o uso da memória compartilhada tornou-se inviável para esse problema.

A fim de diminuir ainda mais a chance de atualizações concorrentes, a cada iteração a ordem de execução das junções em cada thread é alterada. A modificação é feita de forma aleatória gerando seqüências através de dois parâmetros: um indica a ordem comum ou inversa, e o outro a junção inicial da seqüência. Devido à inexistência de geração de números aleatórios em CUDA, são utilizados valores das iterações e índices das junções para o cálculo dos parâmetros.

Diminuir a concorrência durante as atualizações (vetor `V'`), tem como objetivo não só aumentar a precisão e convergência do sistema, mas também diminuir os conflitos de acesso à memória. A sugestão é utilizar a memória local para diminuir algumas leituras na memória global. Aplicando uma iteração do método SOR (sem projeção, para simplificar) para uma junção e assumindo as matrizes `J` e `J'` já reduzidas para doze elementos (Seção 2.2.2), apresenta-se o seguinte pseudocódigo:

```
1    for each(Constraints cIdx in joint)
2    {
3      deltaX = b(cIdx) - d(cIdx)*x(cIdx);
4      for (j=0; j<12; j++)
5        deltaX -= J(cIdx,j)*V'(j);
6      x(cIdx) = x(cIdx) + deltaX;
7      for (j=0; j<12; j++)
8        V'(j) += J'(cIdx,j)*deltaX;
9    }
```

Analisando o vetor[2] `V'` percebe-se leitura na linha 5 e leitura e escrita na linha 8. Uma maneira de utilizar a memória local é criar um `V'` local donde será efetuada a leitura (5). Porém, com isso, `V'` local deve ser atualizado sempre que ocorrer uma escrita em `V'`. Nesse caso, existe o problema de atualizações do `V'` pelas demais threads entre o momento da atualização do `V'` local e o uso do mesmo. Por isso, estudou-se mudanças no algoritmo para que o uso da memória local seja realizado logo após sua atualização.

A maneira encontrada para diminuir o intervalo mencionado acima de forma eficiente é através de uma reorganização do código. A memória local é utilizada de forma implícita no código, no qual a atualização de `V'` é feita na mesma instrução de sua leitura, portanto uma vez calculada a atualização, este valor, ainda em memória local, é utilizado, evitando um acesso a memória global. A nova organização é ilustrada no pseudocódigo a seguir.

```
deltaX = 0;
for each(Constraints cIdx in joint)
{
  acc = 0;
  for (j=0; j<12; j++)
    acc -= J(cIdx,j)* (V'(j) += J'(cIdx,j)*deltaX);
  deltaX = acc + b(cIdx) - d(cIdx)*x(cIdx);
  x(cIdx) += deltaX;
}
```

[2]Esse vetor contém doze elementos, representando as velocidades (linear e angular) de ambos os corpos associados à junção.

```
for (j=0; j<12; j++)
  V'(j) += J'(cIdx,j)*deltaX;
```

5.3.2 Solucionador de EDO

O diagrama de classes UML contendo as principais classes do solucionador de PCL em CUDA é ilustrado na Figura 5.1. A classe `CUDAIntegrator`, derivada de `Integrator` ([dS08] para maiores detalhes) tem como função atualizar o estado de cada corpo rígido. O método `CUDAIntegrator::setInput(rbStream)` fornece a lista de corpos rígidos ao solucionador, onde `rbStream` é um objeto da classe `RigidBodyStream`.

O método `CUDAIntegrator::run(constraintVel, dt)`, responsável pela invocação do kernel em GPU, toma como parâmetros as velocidades (linear e angular) oriundas das restrições de cada corpo rígido e o passo de tempo `dt`.

Tornar paralelo o processo de atualização do estado de cada corpo rígido não gera dificuldades, devido à independência das atualizações. O método `RigidBody::update()` realiza a integração numérica da equação de movimento de um corpo rígido. Então cada thread é responsável pela atualização de um corpo rígido, como mostrado no pseudocódigo a seguir:

```
__global__ void CUDAIntegrator::run(constraintVel, dt)
{
  int rbIdx = blockIdx.x * blockDim.x + threadIdx.x;
  if (rbIdx >= numberOfBodies)
    return;

  rbStream[rbIdx].update(constraintVel[rbIdx], dt);
}
```

5.4 Comentários finais

Neste capítulo apresentaram-se os algoritmos e as respectivas implementações para GPUs utilizando arquitetura CUDA. Na fase geral do detector de colisões do sistema de animação foi utilizado uma divisão espacial em células na qual uma forma é relacionada com uma célula, reduzindo assim as possíveis colisões para formas pertencentes à mesma célula ou células vizinhas imediatas. Tornar a detecção de colisão para execução em paralelo não é uma tarefa direta e algumas técnicas, descritas no texto foram utilizadas. Na fase exata foram utilizados em GPU os mesmos métodos da CPU, porém executados em forma paralela para cada um dos pares. O detector de colisões em GPU é um objeto da classe `CUDACollision`.

Além disso, foram apresentadas as implementações do solucionador de PCL em CUDA. O mesmo método, SOR com projeção, foi utilizado em CPU e GPU. A montagem do sistema e as pré-computações são implementadas em paralelo de forma direta, porém o método principal requer alguns cuidados para evitar a concorrência no acesso à memória entre as threads. Duas otimizações com relação à utilização da memória compartilhada foram feitas. O solucionador de PCL em GPU é um objeto da classe `CUDALCPSolver`.

Por fim, o solucionador de EDO em paralelo foi apresentado. A simplicidade do paralelismo aqui se torna clara, devido à independência dos corpos na atualização de seus estados. O solucionador de EDO em GPU é um objeto da classe `CUDAIntegrator`.

CAPÍTULO 6

Exemplos

6.1 Introdução

Neste capítulo são apresentados e discutidos resultados de simulações dinâmicas de corpos rígidos que ilustram o funcionamento do detector de colisões e motor de física propostos.

A plataforma principal de teste utilizada contém um processador Intel Core 2 Duo E6300 com 2GB de memória RAM e uma GPU NVidia GeForce 8800GTX PCI-E 768MB com sistema operacional Microsoft Windows XP®Professional SP2.

Para obtenção dos tempos de execução dos algoritmos implementados neste trabalho, utilizou-se as funções do sistema operacional Windows `QueryPerformanceCounter()` e `QueryPerformanceFrequency()`, como mostrado no código abaixo:

```
float updateTime()
{
   static __int64 lastCount;
   __int64 count, freq;

    // count recebe o contador corrente
   QueryPerformanceCounter((LARGE_INTEGER*)&count);
    // freq recebe a freqüência do processador
   QueryPerformanceFrequency((LARGE_INTEGER*)&freq);

   float deltaTime = (float)(count - lastCount)*1000 / (float)freq;
   lastCount = count;
   return deltaTime;
}
```

A função `updateTime()` retorna o tempo, em milisegundos, transcorrido desde a última invocação desta. Primeiro, é capturada a contagem atual de um contador de alta resolução implementado em hardware. Este contador não pode ser zerado, por isso a necessidade de obter o valor antes e depois do trecho de código a ser medido. Em seguida, é recuperado o número de vezes, ou freqüência, que o contador é incrementado por segundo, sendo o tempo transcorrido a diferença da contagem entre a última medição e a medição atual dividida pela freqüência. Para medir o tempo gasto na execução de um trecho de código faz-se:

```
⋮
updateTime();
   // execuçao do trecho de código
   ⋮
float tempo = updateTime();
⋮
```

Foram desenvolvidas três aplicações, apresentadas nas Seções 6.2 a 6.4, cujas simulações são definidas de forma a explorar o processamento tanto em CPU quanto em GPU. Tais aplicações foram executadas, com 10 iterações do solucionadoor de PCL, para um número variado de corpos rígidos, junções, e colisões. Para cada execução, foram medidos os tempos de processamento em CPU e GPU para cada componente do sistema, usando a função `updateTime()` como exemplificado anteriormente. Os tempos medidos foram tabelados e discutidos a seguir.

O desempenho do detector de colisões é dado em função do número de formas que compõem os atores e a dispersão das mesmas na cena, sendo necessário maior processamento quando as formas estão próximas tanto em CPU quanto em GPU. Já a quantidade de restrições, sendo elas oriundas de contato e/ou junções, influenciam no desempenho da montagem e resolução do PCL, e além disso o desempenho da execução em GPU é afetado pela ocorrência de um mesmo ator em diversas restrições, ocasionando escrita concorrente na memória do dispositivo, como dito na Seção 5.3. O desempenho do solucionador de EDO considera o número de atores na cena.

Com relação ao tempo de comunicação entre CPU e GPU, são apresentados nas Tabelas 6.1 e 6.2 os tempos de transferência das listas de corpos rígidos, formas e restrições para a GPU e os tempos de retorno dos estados atualizados de cada corpo à CPU, respectivamente. Como descrito na Seção 5.1, as listas só são atualizadas em GPU caso haja alterações, com exceção da lista de restrições (oriundas de junções) criada em CPU e enviada a cada frame. Os tempos de transferência individuais das listas são apresentados na Tabela 6.1.

Corpos Rígidos		Formas		Restrições	
Qtde.	Tranf.(ms)	Qtde.	Tranf.(ms)	Qtde.	Tranf.(ms)
50	0,53	50	0,07	1000	0,08
1000	0,58	1000	0,16	1000	0,53
4000	0,97	4000	0,49	40000	1,95
16000	2,31	16000	1,53	100000	4,60

Tabela 6.1: Tempos de transferência das listas para a GPU.

Os dados que retornam à CPU contêm, para cada corpo rígido, além do novo estado, a matriz de orientação 3×3 obtida através do quaternion $\mathbf{q}$ do estado. Esses dados serão sempre retornados independente do tipo de aplicação. A Tabela 6.2 mostra os tempos de transferência.

6.2 Esferas

Nesta aplicação são simulados milhares de atores dentro de um cubo transparente. Cada ator tem um corpo rígido definido por uma esfera colorida de raio e densidade unitários. Considera-se o coeficiente de atrito igual a $0,5$ e o cubo com dimensão lateral igual a 40. A igualdade na dimensão das esferas faz com que as células do detector de colisões possuam

Corpos Rígidos	
Qtde.	Tranf.(ms)
50	0,45
1000	0,52
4000	0,72
16000	1,51

Tabela 6.2: Tempos de transferência dos novos estados para a CPU.

o tamanho ideal para o desempenho da simulação. Porém, com a proximidade das esferas, cada célula contém um número elevado de testes a realizar.

Na caixa há também uma esfera de cor branca na qual o usuário pode aplicar forças ao longo dos eixos principais do sistema global, fazendo com que esta se movimente e colida com as demais esferas da caixa.

A Figura 6.1 apresenta três frames da aplicação para ilustrar o comportamento das esferas.

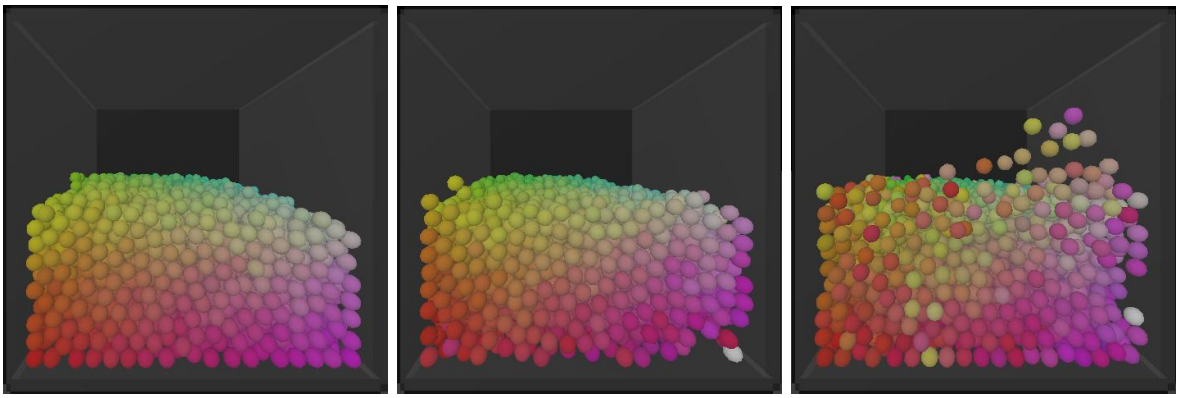


Figura 6.1: Imagens da simulação com 4000 esferas.

A Tabela 6.3 apresenta os tempos referentes à execução da simulação. Esta tabela contém em suas colunas os tempos de execução em CPU e GPU dos processos, sendo eles detector de colisões (Col.), montagem e resolução do PCL (PCL), e solucionador de EDO e atualização do novo estado dos corpos rígidos (EDO). As linhas das tabelas variam de acordo com a quantidade de atores utilizados na cena (Atores), e conseqüentemente a quantidade de restrições (Restr.). Além disso, é apresentado o *speedup* que é a relação entre o tempo gasto em CPU e o tempo gasto em GPU, representando maior eficiência da GPU, caso *speedup* maior que 1, ou maior eficiência da CPU, caso contrário. A Figura 6.2 apresenta o gráfico referente aos tempos de execução total em CPU e GPU mostrados na Tabela 6.3, além de uma linha horizontal indicando o tempo necessário de execução para alcançar 30 quadros por segundo. Simulações com o tempo total de execução abaixo da linha são consideradas simulações em tempo real.

O número de restrições da coluna Restr. é referente aos contatos entre as esferas sendo estas em repouso como na primeira imagem apresentada na Figura 6.1.

A análise dos dados confirma que o uso mais eficiente da GPU ocorre à medida que aumenta o número de atores a serem processados. De acordo com a Tabela 6.4, o detector de colisões em GPU começa a ser mais eficiente com 1000 esferas e atinge um *speedup* de $3,66$ vezes simulando 16000 esferas. O solucionador de EDO também atinge um *speedup* semelhante para a quantidade máxima testada, porém, conforme mostrado na Figura 6.3, o

		Tempo CPU(ms)				Tempo GPU(ms)			
Atores	Restr.	Col.	PCL	EDO	Total	Col.	PCL	EDO	Total
50	156	0,21	0,36	0,01	0,58	2,05	1,75	0,45	4,25
200	948	0,66	0,95	0,04	1,65	2,23	2,24	0,46	4,93
500	3819	2,02	5,95	0,11	8,08	2,70	3,30	0,46	6,46
1000	8151	4,55	11,93	0,24	16,72	3,46	6,65	0,49	10,60
2000	15276	9,86	25,12	0,47	35,45	4,82	13,09	0,52	18,43
3000	23805	15,68	40,23	0,75	56,66	6,01	20,42	0,56	26,99
4000	32370	19,50	52,63	0,96	73,09	7,19	26,45	0,69	34,33
8000	66507	44,08	114,79	1,96	160,83	12,71	75,18	0,81	88,70
12000	103023	68,55	178,45	3,09	250,09	19,92	127,37	0,91	148,20
16000	139092	92,11	236,33	4,38	332,82	25,16	187,12	1,07	213,35

Tabela 6.3: Tempos de execução da aplicação das esferas coloridas.

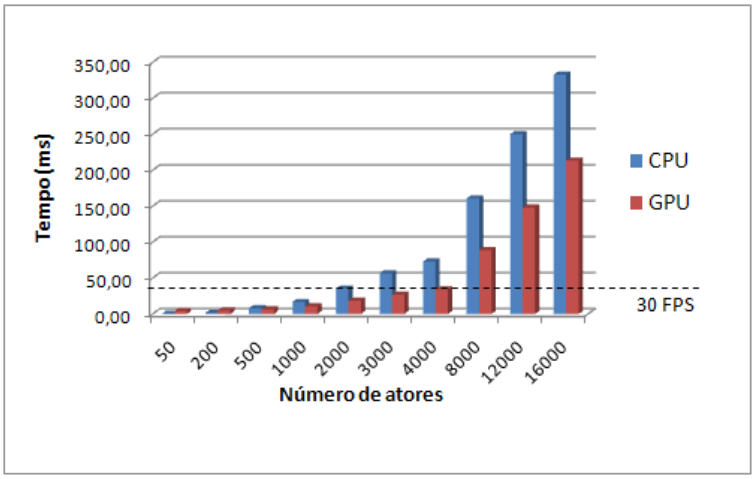


Figura 6.2: Tempo total de execução da aplicação em CPU e GPU.

tempo desse processo ocupa uma porção não significativa do tempo total, sendo a maior parte desse referente à montagem e resolução do PCL. O PCL em GPU alcança um desempenho de 2 vezes em relação à CPU, porém o desempenho diminui com o aumento da quantidade de atores. Esse comportamento ocorre pois quanto mais esferas no cubo, mais colisões, e conseqüentemente mais restrições e dependências entre elas, gerando concorrência ao acesso à memória cujo custo é alto. O *speedup* do tempo total foi de aproximadamente 2 vezes.

Na Tabela 6.4 são apresentados os números de quadros por segundo (FPS) obtidos com a aplicação, considerando-se apenas colisão e a física. Observa-se que o detector de colisões e o motor de física são capazes de executar a aplicação em CPU a uma taxa de $28,21$ fps com 2000 atores, e, em GPU, a uma taxa de $29,13$ com 4000 atores. Portanto, não somente o desempenho da GPU cresce com o aumento do número de atores, como esperado, mas, além disso, até 4000 atores, o resultado desse desempenho se dá em tempo real, ou seja, a implementação em GPU pode ser usada em aplicações práticas que requeiram execução em tempo real.

		Speedup					FPS	
Atores	Restr.	Col.	PCL	EDO	Total		CPU	GPU
50	156	0,10	0,21	0,02	0,14		1724,14	235,29
200	948	0,30	0,42	0,09	0,33		606,06	202,84
500	3819	0,75	1,80	0,24	1,25		123,76	154,80
1000	8151	1,32	1,79	0,49	1,58		59,81	94,34
2000	15276	2,05	1,92	0,90	1,92		28,21	54,26
3000	23805	2,61	1,97	1,34	2,10		17,65	37,05
4000	32370	2,71	1,99	1,39	2,13		13,68	29,13
8000	66507	3,47	1,53	2,42	1,81		6,22	11,27
12000	103023	3,44	1,40	2,66	1,69		4,00	6,75
16000	139092	3,66	1,26	3,57	1,56		3,00	4,69

Tabela 6.4: *Speedup* dos tempos da aplicação das esferas coloridas.

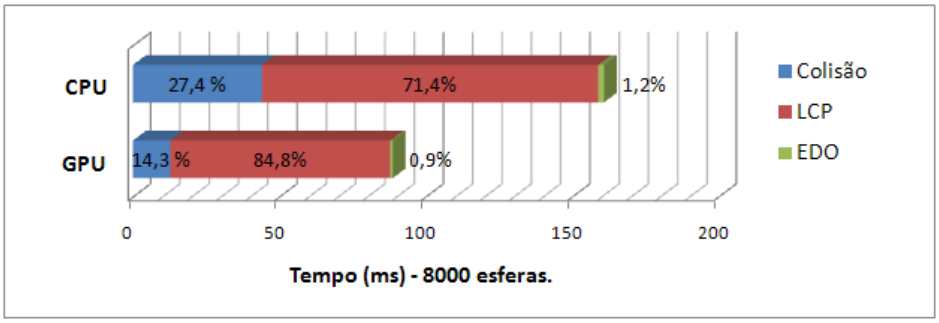


Figura 6.3: Tempos de execução da aplicação em CPU e GPU com 8000 esferas.

6.3 Cubo de moléculas

Nesta segunda aplicação um cubo de moléculas é simulado. O cubo é composto por $n \times n \times n$ atores com forma de esferas, representando as moléculas, e $3 \times n^2 \times (n-1)$ junções fixas, onde n é a quantidade de esferas na lateral do cubo. Cada junção fixa retira seis graus de liberdade de ambos os corpos pertencentes a junção, como visto na Seção 2.1, ou seja, adiciona seis restrições ao sistema. Um ator do cubo está conectado por:

- 3 junções, se posicionado em um canto do cubo;
- 4 junções, se posicionado em uma quina do cubo;
- 5 junções, se posicionado na superfície do cubo;
- 6 junções, se posicionado no interior do cubo.

Este exemplo foi elaborado para teste da estabilidade numérica do solucionador de PCL em GPU. Como visto na Seção 5.3, a dependência entre as restrições gera concorrência no acesso à memória, e quando há concorrência várias threads podem atualizar um mesmo resultado, o que poderia desestabilizar o sistema. Para um cubo de lado contendo 10 esferas, tem-se 1000 esferas e 2700 junções totalizando 16200 restrições; contudo, com 10 iterações, as junções são mantidas estavelmente em tempo real. As esferas possuem raio e densidade unitários e o espaçamento de $0,5$ em cada coordenada. Para testar a estabilidade aplica-se uma força nas esferas de dois cantos extremos do cubo tal que o mesmo gire no espaço, uma vez que a gravidade está sendo desconsiderada ou ajustada para zero. Após a aplicação das

forças, todas as esferas devem girar conjuntamente, pois estão "fixadas"umas nas outras. A Figura 6.4 apresenta frames da aplicação em execução.

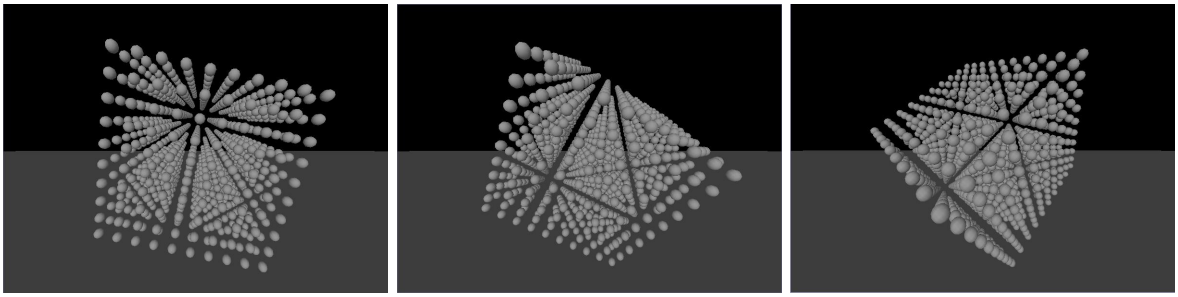


Figura 6.4: Imagens da simulação do cubo de moléculas com 1000 esferas e 2700 junções.

A Tabela 6.5 mostra os tempos referentes somente à montagem e execução do PCL nas simulações, pois não há colisões na cena.

Atores(n)	Restr.	Tempo CPU(ms)	Tempo GPU(ms)	Speedup
64 (4)	864	1,11	3,81	0,29
216 (6)	3240	4,15	4,27	0,97
512 (8)	8064	10,54	6,49	1,62
1000 (10)	16200	21,52	12,98	1,66
1728 (12)	28512	37,82	22,95	1,65
2744 (14)	45864	62,26	40,38	1,54
4096 (16)	69120	91,58	62,31	1,47
5832 (18)	99144	132,15	101,72	1,30

Tabela 6.5: Resultados do PCL no cubo de moléculas.

Nota-se o desempenho maior do PCL para um número de restrições semelhante em ambas arquiteturas, se comparado com a aplicação anterior. Isso acontece pois a montagem das restrições referentes aos contatos é computada como parte do PCL, porém a montagem das restrições referentes às junções não. Portanto, como nesta aplicação não há contatos, não há montagem das restrições e o tempo do PCL é reduzido.

O desempenho do solucionador de PCL para essa aplicação é afetado pela grande dependência gerada através das junções que conectam os atores da cena. Contudo, o sistema manteve-se estável e a GPU teve seu desempenho superior nessa aplicação para simulações com o número de atores a partir de 512.

6.4 Funil

A aplicação considera atores cuja forma é uma composição de três cápsulas posicionadas em cruz, como ilustrado na Figura 6.5. Esses atores caem em um objeto semelhante a um funil. Na simulação é considerado o coeficiente de atrito igual a $0,5$. As cápsulas de cada ator possuem raio $0,1$, comprimento $0,7$ e densidade 2.

O propósito dessa aplicação é medir o desempenho do detector de colisões. A fim de aumentar o processamento durante a execução do componente, escolheu-se atores multiformas para aumentar os testes de colisão em cada célula. Cápsulas foram utilizadas pois seu

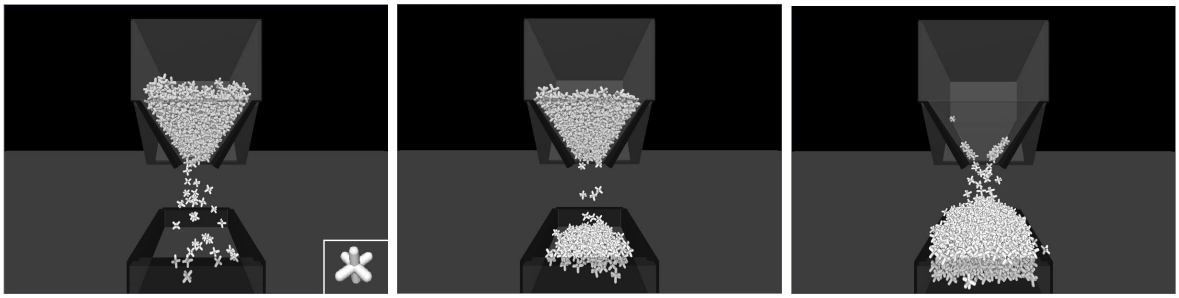


Figura 6.5: Imagens da simulação do funil com 800 atores.

cálculo de interseção possui maior processamento comparado ao cálculo de interseção entre esferas.

A Tabela 6.6 apresenta os tempos referentes somente à execução do detector de colisões, incluindo fase geral e fase exata, da aplicação. Além das implementações do detector de colisões com algoritmo da divisão espacial da fase geral em CPU e GPU, são apresentados também os tempos do detector de colisões em CPU com a implementação por força bruta (FB) da fase geral. Para a detecção de colisões é considerado o número de formas na cena, nesse caso, três vezes o número de atores.

	Colisão(ms)			
Atores	CPU(FB)	CPU	GPU	Speedup
50	2,93	1,36	2,77	0,49
100	7,56	3,72	4,18	0,89
200	19,95	9,54	5,66	1,69
400	58,83	26,72	8,25	3,24
800	172,56	62,78	11,43	5,49
1600	574,12	133,16	18,15	7,34

Tabela 6.6: Resultados da terceira aplicação.

A detecção de colisão em GPU atingiu um *speedup* de $7,34$ vezes em relação à mesma implementação em CPU. O tempo alcançado é mais de 30 vezes mais rápido que a implementação da força bruta em CPU, Figura 6.6. Atores com múltiplas formas acarretam o aumento da quantidade de testes de colisão, porém em GPU isso é realizado por diversas threads, o que aumenta a eficiência do dispositivo. Isso comprova que esta tarefa pode ser eficientemente implementada em GPU.

6.5 Comentários finais

Neste capítulo foram apresentados os resultados de desempenho em três aplicações executadas em CPU e GPU. Através dos tempo apresentados pode-se concluir que o solucionador de EDO e o detector de colisões são tarefas que adequam-se à arquitetura paralela atingindo um desempenho superior à execução seqüencial. As implementações desses são escaláveis, portanto, se executadas em uma GPU composta por um número maior de processadores, o ganho pode ser ainda maior. Por outro lado, o solucionador de PCL não é um problema adequado para implementação em GPU, pois, como comentado na Seção 5.3, além do pro-

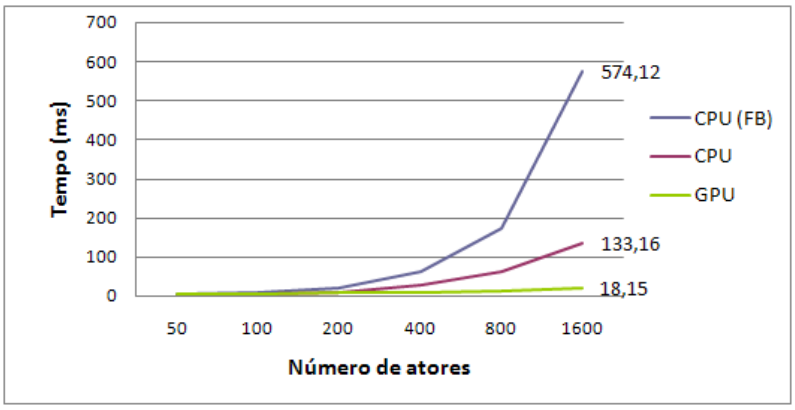


Figura 6.6: Tempos de execução do detector de colisões no exemplo do funil.

blema da precisão numérica, há a concorrência da escrita em memória, o que consome um maior processamento. Apesar disso, o resultado obtido, em torno de 2 vezes mais rápido que a CPU nos casos de teste realizados, é satisfatório e pode ser comparado com resultados de [Vas08], o qual trata da implementação em GPU de métodos iterativos para a solução de sistema lineares, onde os *speedups* alcançados foram entre $1,5$ e 3 vezes.

Concluindo, com o detector de colisões e o motor de física implementados em GPU, o passo de simulação teve seu desempenho melhorado, comprovando a eficiência do uso da GPU como co-processador do sistema de animação.

CAPÍTULO 7

Conclusão

7.1 Discussão dos resultados obtidos

O objetivo geral deste trabalho foi o desenvolvimento de um detector de colisões e um motor de física para GPU com arquitetura CUDA. Ambos os componentes são extensões de um framework para simulação e animação baseada em física chamado AS. Assim, o sistema de animação é capaz de executar em ambos os processadores, CPU e GPU. CUDA é uma nova arquitetura de hardware e software que oferece às aplicações intensivas computacionalmente acesso ao poder de processamento paralelo das GPUs através de uma nova interface de programação, utilizando a linguagem C.

O detector de colisões foi dividido em duas fases. Na fase geral, os testes de colisão são minimizados através de cálculos simplificados, determinando os possíveis pares de colisões. Cabe à fase exata então avaliar esses pares e determinar com precisão os contatos e suas informações. Na implementação da fase geral utilizou-se um método de divisão espacial em ambas arquiteturas. Em CPU o gargalo está na criação das células, pois isso é feito de forma seqüencial para cada objeto. Em GPU, isso é executado em paralelo, e o gargalo está na análise das células de colisão. Nessa etapa, em uma primeira implementação cada thread era responsável por uma célula de colisão, porém essas podem conter números variados de formas a serem testadas, resultando em um balanceamento de carga ineficiente. Por isso, otimizou-se a implementação onde cada célula de colisão é avaliada por diversas threads. O problema não foi inteiramente resolvido, porém o desempenho foi melhorado mantendo a mesma precisão. Em um dos exemplos, o detector de colisões em GPU foi sete vezes mais rápido comparado à CPU.

O solucionador de PCL foi implementado através do método SOR. Nesse método, algumas otimizações e pré-computações são consideradas [Erl04]. Durante a implementação em CUDA encontra-se alguns problemas devido às escritas concorrentes entre as threads. Cada thread é responsável por calcular as interações referentes a um conjunto de junções de forma seqüencial, porém, pode ocorrer que duas threads tentem modificar a força resultante das restrições de um mesmo corpo. Isso, além de problemas de precisão, afeta o desempenho. Portanto, otimizações foram propostas com o objetivo de reduzir as chances de concorrência. A primeira foi aumentar a quantidade de junções por thread e a segunda reorganizar o laço principal do kernell reduzindo o número de leituras na memória global. Assim como em muitos problemas da computação, tem-se aqui precisão versus desempenho e isso é determinado através do número de iterações executadas pelo método. A precisão em GPU foi satisfatória para o

objetivo do motor e o *speedup* alcançado foi maior que dois.

No solucionador de EDO foi implementado o método de integração numérica de Euler. Devido à integração da equação de movimento de um corpo não influenciar em outro, o algoritmo paralelo torna-se simplificado e cada thread fica responsável por determinar o estado de um corpo. O desempenho desse componente em GPU, apesar de não influenciar muito no processo total, foi também superior à CPU.

O desempenho das aplicações implementadas em GPU utilizando CUDA está altamente relacionado com a organização e utilização dos dados na memória no dispositivo. Uma implementação eficiente deve utilizar memória de baixa latência e alta largura de banda, porém essa contém menor capacidade de armazenamento. Portanto, aí está a complexidade de desenvolver um método utilizando os melhores recursos, atingindo assim um resultado eficiente. O uso dos diversos tipos de memória foram estudados, porém barreiras de uso foram encontradas devido aos métodos utilizados. Além disso, a implementação dessa primeira versão do motor de física foi bastante trabalhosa, sendo assim impossível desenvolver e testar outras técnicas mais eficientes para cada uma das etapas. Essa primeira versão serve como base para o desenvolvimento de novos estudos e pesquisas nessa área. Afinal, as GPUs estão no começo de uma grande evolução.

Em virtude dos resultados obtidos, pode-se afirmar que todos os objetivos propostos no trabalho foram plenamente atingidos.

7.2 Trabalhos futuros

Algumas possibilidades de extensão do trabalho são relacionadas a seguir:

- Implementação de hierarquia de células na fase geral do detector de colisões, permitindo simulações com objetos de dimensões variadas com a mesma eficiência. Na fase exata, implementação do algoritmo GJK [vdB99] para objetos convexos.
- Estudar o grau de convergência do método utilizado no solucionador de PCL assim como outras maneiras de execução desse método em GPU.
- Otimizar o solucionador de PCL em CUDA através do uso de memória compartilhada.
- Considerar, na colisão, objetos com formas definidas através de malhas de triângulos.
- Estender o motor de física para corpos flexíveis.

Referências Bibliográficas

[AGE05] AGEIA. A white paper: Physics, gameplay and the physics processing unit. Relatório técnico, 2005. *http://www.ageia.com/pdf/wp_2005_3_physics_gameplay.pdf*. Acessado em 28 de março de 2006.

[AS01] Ulf Assarsson e Per Stenström. A case study of load distribution in parallel view frustum culling and collision detection. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, páginas 663–673. Springer-Verlag, London, UK, 2001. ISBN 3-540-42495-4.

[Bar91] David Baraff. Coping with friction for non-penetrating rigid body simulation. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, páginas 31–41. ACM, New York, NY, USA, 1991.

[Bar92] David Baraff. *Dynamic simulation of nonpenetrating rigid bodies*. Tese de Doutorado, Ithaca, NY, USA, 1992.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, páginas 23–34. ACM, New York, NY, USA, 1994.

[BF97] Richard L. Burden e J. Douglas Faires. *Numerical Analysis*. Brooks/Cole Publishing Company, 1997. 6th edition.

[BW97] David Baraff e Andrew Witkin. Physically based modeling: Principles and practice. ACM Siggraph Course Notes, ACM Press, 1997.

[Cat05] Erin Catto. Iterative dynamics with temporal coherence. Game Developer Conference, 2005.

[Cor06] NVIDIA Corporation. Cuda programming documentation, 2006. *http://www.nvidia.com/object/cuda_home.html*. Acessado em 03 de março de 2008.

[Cor08] AGEIA Corporation. The ageia physx sdk. Disponível em http://developer.download.nvidia.com/PhysX/2.8.1/PhysX_2.8.1_SDK_Core.msi, último acesso em 15/07/2008, 2008.

[dS08] Alexandre S. da Silva. *Um Motor 3D para Simulação Dinâmica de Corpos Rígidos*. Dissertação de Mestrado, Universidade Federal de Mato Grosso do Sul, Campo Grande - MS, 2008.

[Ebe04] David H. Eberly. *Game Physics (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. First Edition.

[Eri04] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[Erl04] Kenny Erleben. *Stable, Robust, and Versatile Multibody Dynamics Animation*. Tese de Doutorado, The Department of Computer Science, University of Copenhagen, Denmark, Nov 2004.

[Gra07] Scott Le Grand. Broad-phase collision detection with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, capítulo 32, páginas 697–721. Addison Wesley, 2007.

[GW07] I. Grinberg e Yair Wiseman. Scalable parallel collision detection simulation. In R. J. P. de Figueiredo, editor, *SIP*, páginas 380–385. IASTED/ACTA Press, 2007.

[KLRS04] A. Kolb, L. Latta, e C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, páginas 123–131. ACM, New York, NY, USA, 2004.

[KP03] D. Knott e D. Pai. Cinder: Collision and interference detection in real–time using graphics hardware. In *Proceedings of Graphics Interface*, páginas 73–80. 2003.

[KSW04] Peter Kipfer, Mark Segal, e Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, páginas 115–122. ACM, New York, NY, USA, 2004.

[LK02] Orion Sky Lawlor e Laxmikant V. Kalée. A voxel-based parallel collision detection algorithm. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, páginas 285–293. ACM, New York, NY, USA, 2002.

[MH07] John D. Owens Mark Harris, Shubhabrata Sengupta. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, capítulo 39, páginas 851–876. Addison Wesley, 2007.

[Mir96] Brian Vincent Mirtich. *Impulse-based dynamic simulation of rigid body systems*. Tese de Doutorado, University of California, Berkeley, 1996.

[Mur88] Katta G. Murty. *Linear complementarity, linear and nonlinear programming*, volume 3 de *Sigma Series in Applied Mathematics*. Heldermann Verlag, 1988.

[Ngu07] Hubert Nguyen. *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2007.

[Oli06] L. L. Oliveira. *Um sistema de animação baseado em dinâmica de corpos rígidos articulados*. Dissertação de Mestrado, UFMS, Campo Grande - MS, 2006. In Portuguese.

[PF05] Matt Pharr e Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[RWC92] Richard E. Stone Richard W. Cottle, Jong-Shi Pang. *The Linear Complementarity Problem*. Academic Press, 1992.

[SL06] Rahul Sathe e Adam Lake. Rigid body collision detection on the gpu. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, página 49. ACM, New York, NY, USA, 2006.

[Smi00] Russell Smith. Open dynamics engine, 2000. *http://www.ode.org*. Acessado em 20 de novembro de 2007.

[Vas08] Filip Vasely. *Iterative GPGPU Linear Solvers for Sparse Matrices*. Dissertação de Mestrado, Czech technical University – Faculty of Electrical Engineering, 2008.

[vdB99] Gino van den Bergen. A fast and robust GJK implementation for collision detection of convex objects. *Journal of Graphics Tools: JGT*, 4(2):7–25, 1999.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo