UNIVERSIDADE FEDERAL FLUMINENSE

MARK EIRIK SCORTEGAGNA JOSELLI

Uma Arquitetura de Motor de Física para Games 3D com Processamento Híbrido entre CPU e GPU e Distribuição Dinâmica de Carga

NITERÓI

2007

UNIVERSIDADE FEDERAL FLUMINENSE

MARK EIRIK SCORTEGAGNA JOSELLI

Uma Arquitetura de Motor de Física para Games 3D com Processamento Híbrido entre CPU e GPU e Distribuição Dinâmica de Carga

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Computação Visual e Interfaces.

Orientador:
Esteban Walter Gonzalez Clua

NITERÓI

2007

Uma Arquitetura de Motor de Física para Games 3D com Processamento Híbrido entre CPU e GPU e Distribuição Dinâmica de Carga

MARK EIRIK SCORTEGAGNA JOSELLI

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre.

Aprovada por:

D.Sc. Esteban Walter Gonzalez Clua (Presidente) / IC-UFF - Universidade Federal Fluminense

---

D.Sc. Paulo Aristarco Pagliosa / DCT-UFMS - Universidade Federal de Mato Grosso do Sul

---

PhD Bruno Feijó / DI-PUC-RIO - Pontifícia Universidade Católica do Rio de Janeiro

---

D.Sc. Anselmo Antunes Montenegro / IC-UFF - Universidade Federal Fluminense

---

D.Sc. Mauricio Kischinhevsky / IC-UFF - Universidade Federal Fluminense

Niterói, 21 de dezembro de 2007.

Dedico este trabalho a minha familia que sempre me apoiou.

Agradecimentos

Primeiramente ao meu orientador, professor Esteban Walter Gonzalez Clua, que sempre me apoiou, mesmo nos momentos difíceis.

Ao professor Paulo Pagliosa por todo apoio e contribuições que me ajudaram a produzir este projeto.

Ao amigo Marcelo Zamith por todas as dicas e contribuições para este projeto.

À professora Aura Conci, que com suas aulas, redespertou em mim o gosto pela computação gráfica.

À minha namorada Maíra por todo o seu carinho e apoio.

À minha avó Oma que sempre me acolheu quando precisei ficar em Niterói.

E aos meus pais que sempre me ajudaram.

Resumo

O realismo em jogos digitais 3D, realidade virtual e simulações têm evoluído rapidamente, requerendo cada vez mais poder de processamento dos computadores. Os motores de física, necessários para tratar o realismo físico, requerem muito deste processamento por terem a característica de necessitarem muitos cálculos matemáticos, alguns com grande complexidade. Com o avanço das GPUs (*Graphics Processing Units*) programáveis este cálculo pode ser direcionado para ser processado nessas placas gráficas. Esta dissertação apresenta um novo motor de física que tem um subconjunto de seus cálculos implementados tanto na CPU como na GPU, utilizando uma arquitetura considerada inédita. Este motor foi batizado de GDE (*GPU Dynamics Engine*).

O processamento de cálculos matemáticos em GPU é altamente otimizado para um elevado número de cálculos devido à estrutura paralela da GPU, fazendo com que o GDE tenha um desempenho melhor na CPU quando há um pequeno número de corpos e um desempenho melhor na GPU para um alto número de corpos.

Esta dissertação apresenta também uma arquitetura de jogos digitais especifica, para ser usada com o GDE juntamente com o *framework* GUFF (Games UFF).

O GDE possui processamento tanto em CPU como em GPU. Como a GPU possui melhor processamento em alguns casos e, por outro lado, uma aplicação pode compartilhar os processadores com o sistema e outros aplicativos, apresentam-se heurísticas para realizar a distribuição automática de carga entre CPU e GPU, que também é uma contribuição inédita desta dissertação.

Abstract

The realism in 3D games, virtual reality and simulations are trespassing by a fast evolution, requiring each time more power from the computers processing. The physics engines, necessary for giving physics reality, require a lot of this processing because of their characteristic of having high complex mathematics' calculation. With the power increase of the programmable GPUs (Graphics Processing Unit) this calculation can be also processed in this graphics boards. This work presents a new physics engine that has some of its calculations' processed in both the CPU and GPU, using an architecture without precedent. This engine was called GDE (Gpu Dynamics Engine).

The processing of mathematical calculation on the GPU is highly optimized for a high number of calculation because of the parallel structure of the GPU, allowing a better performance of the GDE in the CPU when there are few number of bodies and a better performance in the GPU for high number of bodies.

This work also presents a specific architecture for games, to be used with the GDE together with the framework GUFF (Games UFF).

The GDE has implementations in both CPU and GPU. Because the GPU has a better processing in some case and the application can share the processors with the system and others applications, it has been developed heuristics to automatic distribution of computation between CPU and GPU. This research has no known precedence.

Palavras-chave

1. General Purpose GPUs
2. Simulação física de corpos rígidos
3. Motor de física
4. Arquitetura para jogos digitais
5. Computação distribuída
6. Computação gráfica
7. Jogos Digitais

Glossário

3DS	:	Estrutura de arquivo contendo informações sobre a malha de um modelo 3D
CPU	:	Unidade de processamento central
CUDA	:	Computer Unified Device Architecture
FPS	:	Frames por segundo
GLEW	:	OpenGL Extension Wrangler
GLSL	:	OpenGL Shader Language
GPGPU	:	Unidade de processamento gráfico para propósito geral
GPU	:	Unidade de processamento gráfico
Heightfield	:	Malhas de poligonos utilizadas para simulação de terrenos
ODE	:	Open Dynamics Engine (motor de física de código aberto)
PPU	:	Unidade de processamento de cálculos físicos
SDL	:	Simple DirectMedia Layer
Shader	:	Expressão adotada para definir um programa que executa na GPU
Texel	:	Elemento de textura

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O desenvolvimento da área de simulações em tempo real faz com que haja uma demanda para que estas simulações sejam mais realistas. Esse realismo pode ser aumentado de diversas maneiras: através de personagens mais inteligentes, modelos mais detalhados e realistas e um maior realismo na simulação física. A necessidade de um realismo físico pode tanto vir de situações reais, bem como simulações para entretenimento, como jogos eletrônicos.

Para implementar este comportamento físico dos personagens e objetos de uma aplicação normalmente utiliza-se motores de física. Os motores de física são tipicamente responsáveis pela forma como os corpos se comportam na simulação de acordo com a física newtoniana: colisão de corpos, objetos caírem e descerem morros, etc. Estes motores simulam como corpos rígidos interagem uns com os outros, podendo também ser utilizados para simulações de partículas, tecidos, fluídos entre outros.

Entre os mais conceituados motores comerciais temos o Havok [4] , utilizado no jogo Half-life 2 [5] e o PhysX [6], utilizado no jogo City of Villains [7]. Uma das características principais de ambos os motores é que estes utilizam processamento paralelo para otimização da simulação. O PhysX se utiliza de uma *Physics Processing Unit* (PPU) para otimizar seus cálculos, que é uma placa desenvolvida especificamente para este motor. Por outro lado, o Havok utiliza as *Graphics Processing Units* (GPU) programáveis, para otimizar seus cálculos.

Com o surgimento das GPUs programáveis, bem como o aumento do seu poder computacional, vem surgindo o conceito de *General-Purpose computation on* GPU (GPGPU), que é a utilização da GPU para processamento genérico, principalmente matemático. As GPUs programáveis possuem uma arquitetura altamente paralela e fortemente voltada

para cálculos matemáticos, o que possibilita sua utilização na solução de diversos problemas paralelizáveis como: problemas de quantum monte carlos [8], inteligência artificial [9] e de raycasting [10], dentre outros. Mais trabalhos nesta área podem ser vistos no portal de desenvolvedores para programação GPGPU [11].

Dentre os motores de código aberto e sem fins comerciais não há nenhum que atualmente utiliza estas abordagens de otimização, fato que faz com que o número de corpos rígidos em uma simulação fique bastante limitado. Uma das metas da presente dissertação consiste em portar para GPU alguns cálculos de um conceituado motor com estas características, mais especificamente o *Open Dynamics Engine* (ODE) [12], fazendo com que este motor fique otimizado para um grande número de corpos.

Pelo fato das GPUs serem altamente paralelas, só é possível otimizar em GPU simulações com grande número de corpos, enquanto que para simulações mais simples a CPU se comporta de forma melhor. Para que a simulação tire sempre o melhor proveito de ambos os processadores esta dissertação também propõe formas de distribuição entre CPU e GPU.

As principais contribuições desta dissertação são:

- Encapsulamento das principais funcionalidades do motor de física ODE para que o mesmo seja utilizado pelo framework GUFF;
- Desenvolvimento de um motor baseado no ODE que pode utilizar tanto a CPU como a GPU para processar alguns de seus cálculos de física, sendo o primeiro otimizado para pequeno número de corpos, enquanto que o segundo para um grande número de corpos;
- Discussão e desenvolvimento de uma arquitetura para simulações físicas e jogos digitais 3D;
- Desenvolvimento de heurísticas para a distribuição automática entre CPU-GPU.

O capítulo 2 desta dissertação apresenta os principais motores de física e a decisão de utilizar o motor ODE como base. Neste capítulo também é apresentado o *framework* GUFF e o encapsulamento do ODE para o Guff, apelidado de GuffODE.

O capítulo 3 apresenta o laço de física e os diversos métodos deste laço implementados em GPU, criando-se um novo motor de física apelidado de GDE (*GPU Dynamics Engine*).

No capítulo 4 são apresentadas e discutidas as diversas arquiteturas para a utilização com motores de física.

No capítulo 5 são discutidas várias formas de se distribuir tarefas entre CPU-GPU através dos conceitos de motor de física apresentado no capítulo 3.

O capítulo 6 apresenta os resultados adquiridos com a utilização deste motor de física juntamente com os métodos de distribuição entre a CPU e a GPU, bem como uma formalização de uma arquitetura específica.

Finalmente, o capítulo 7 apresentada as considerações finais e as conclusões deste trabalho.

Capítulo 2

GuffOde

Neste capítulo são apresentados diversos motores de física, a escolha pelo motor de física e também um encapsulamento do motor de física ODE para o uso com o *framework* Guff.

Um dos fatores preocupantes em simulações em tempo real, e não poderia ser diferente em simulações de corpos rígidos, é o tempo de resposta. Se este tempo de resposta for muito alto, a simulação tende a ficar lenta e, conseqüentemente, visualmente menos realística. Por esta razão, em jogos e simulações em tempo real, os cálculos de física devem priorizar a velocidade, mas ao mesmo tempo, manter a precisão de seus cálculos. Para que isto ocorra pode-se desenvolver um motor especificamente para estes cálculos ou utilizar um já existente. Esta dissertação utiliza como base um motor já existente.

Este capítulo apresenta de forma sucinta os *frameworks* para jogos bem como o GUFF, utilizado nesta dissertação (seção 2.1). Em seguida são apresentados os principais motores de física (seção 2.2) e o critério de escolha utilizado (seção 2.3). Na seção 2.4 presenta-se detalhes do motor de física ODE, bem como suas principais funcionalidades. Na seção 2.5 apresenta-se a implementação da GuffODE bem como os resultados iniciais que foram obtidos com ele. Finalmente, na seção 2.6, apresenta-se as considerações finais e as conclusões deste capítulo.

2.1 Framework para Jogos

Um *framework* consiste em uma arquitetura pré-construída, que serve como camada de abstração para a realização de tarefas repetitivas e comuns a todos os problemas de um determinado domínio [13].

No domínio de jogos *framework* pode ser definido como uma arquitetura, com um

conjunto de bibliotecas, que auxilia na implementação de um jogo digital. Os *frameworks* em geral possuem os seguintes componentes: Áudio, Dispositivos de entrada, Interpretador de Scripts, Visualização, módulo de IA (muitas vezes implementado através dos scripts), Redes, Simulação Física, como mostra a figura 2.1.

Figura 2.1: Dispositivos de um framework [1].

2.1.1 O Framework Guff

O *framework* Guff [1] foi implementado em C++ e foi projetado para atender os seguintes requisitos:

- Oferecer uma interface que modele uma possível arquitetura genérica de um jogo;
- Oferecer um conjunto de ferramentas para auxiliar no desenvolvimento dos programas;
- Funcionar em plataforma Windows e Linux;
- Gestão automática de todos os recursos;
- Facilidade de uso;
- Reuso de bibliotecas já existentes;
- Possibilitar que a configuração de parâmetros da aplicação seja determinada fora do código fonte.

O Guff se utiliza de bibliotecas auxiliares como:

- A biblioteca SDL [14] para criação de janelas;
- A API OpenGL [15] para visualização;
- A biblioteca Audiere [16] para o áudio;

- A biblioteca DevIL [17] para manipulação de imagens;
- A biblioteca Glew [18] para inicializações e uso de extensões do OpenGL;
- A Linguagem Lua [19] para script de configurações.

A figura 2.2 ilustra estas bibliotecas encapsuladas em espaços de nomes, alem das diversas outra implementações realizadas, tais como: biblioteca de matemática, biblioteca de tratamento de erros e carregamento de mapas BSP, formato utilizado no jogo Quake 3 [20].

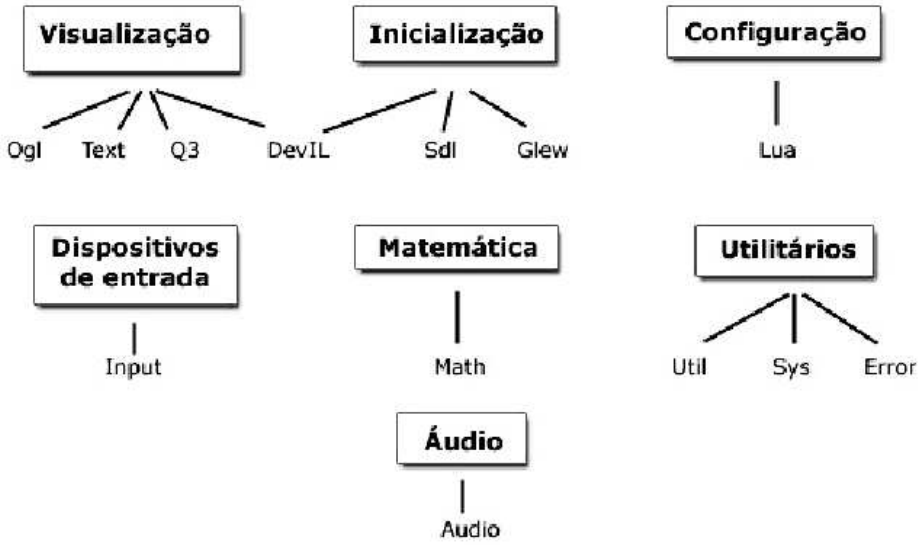

Figura 2.2: Módulos do framework GUFF [1].

Pode-se reparar na figura 2.2 que o framework Guff não possui um módulo responsável pela física. Este capítulo se concentra em descrever uma implementação deste módulo.

Todas estas características serão levadas em conta quando a escolha pela pelo motor física for feita, bem como na modelagem do encapsulamento do motor escolhido. Mais sobre este framework pode ser visto em [1].

2.2 Motores de Física

O principal requisito dos motores de física consiste em calcular como os corpos devem se comportar fisicamente de acordo com as leis da física newtoniana, ou modelos mais simplificados, em cada instante de tempo. Um motor de física para aplicações em tempo real, como é o caso deste trabalho, tem que calcular rapidamente e sem muitos erros para

que o tempo de resposta seja baixo e conseqüentemente possa haver interatividade na aplicação.

Para que a escolha pelo motor de física seja acertada pesquisou-se os principais motores de física disponíveis. Nesta seção apresenta-se os seguintes motores: Havok, PhysX, Newton, Tokamak, Bullet, ODE e OpenTissue.

2.2.1 Havok Physics

Este motor de física foi criada pela empresa Havok e foi primeiramente apresentada na Games Developer Conference de 2000. É um motor comercial escrito em C++ e possui implementações para Windows, Linux, Mac OS, Xbox [21], Xbox360 [21], GameCube [22], WII [22], Playstation 2 [23] e Playstation 3 [23]. Ele é usado em vários jogos comerciais.

Possui as seguintes características [4]: detecção de colisões, compactação para representação de grandes malhas de colisão, dinâmica de veículos, serialização de dados, suporte a arte e um depurador visual.

Uma das grandes inovações deste motor é o fato de ele se utilizar da GPU para otimizar alguns de seus cálculos da física.

2.2.2 PhysX

Este motor de física é o principal concorrente da Havok. É um motor também comercial apesar de ter licença gratuita em alguns casos, como para desenvolvimento para plataforma PCs. Foi criado pela empresa Ageia e anteriormente era chamado de Novodex. Foi escrito em C++ e possui implementação para Windows, Linux, Xbox [21], Xbox360 [21], Playstation 2 [23] e Playstation 3 [23]. Ele é usado em vários jogos comerciais.

Possui as seguintes características [6]: arquitetura focada para paralelismo de hardware, detecção de colisões, corpos deformáveis, sistema de partículas, criação e simulação de fluido volumétrico, simulações de roupa, dinâmica de veículos, suporte a arte e um depurador visual.

Uma das grandes inovações desta empresa foi o desenvolvimento de uma *physics processing unit* (PPU): um processador para aliviar a CPU de cálculos envolvendo física.

2.2.3 Newton

Motor de física gratuito, mas de código fechado. Foi criado por Julio Jerez disponível para plataforma Windows, Linux e MacOs. Foi escrito em C, aparentemente não é atualizado desde 01/2006 e não existe notícia de nenhum jogo comercial usando este motor.

Possui as seguintes características [24]: detecção de colisões, tratamento de colisões convexas, suporte a uma variedade de tipo de primitivas e junções, dinâmica de veículos e dinâmica de *ragdoll*.

2.2.4 Tokamak

Motor de física gratuito que acabou de ser lançada em código aberto. Criado por David Lam disponível para plataforma Windows. Escrito em C++ e não existe notícia de nenhum jogo comercial usando este motor.

Possui as seguintes características [25]: detecção de colisões, suporte a uma variedade de tipos de primitivas e junções, suporta um grande número de primitivas na mesma simulação e possui um sistema de corpos quebráveis.

2.2.5 Bullet

Motor de física gratuito e de código aberto. Escrito em C++ e foi criado por Erwin Coumans, que trabalhou anteriormente na Havok. Disponível para plataforma Windows, Linux, Mac OS e Playstation 3 [23], e é atualizado constantemente. Não existe notícia de nenhum jogo comercial usando este motor.

Possuí as seguintes características [26]: detecção de colisões utilizando a biblioteca Gimpact [27], suporte a uma variedade de tipos de primitivas e juntas, dinâmica de veículos, suporte a arte através do Blender[1] [28] e suporte ao COLLADA[2] [29].

2.2.6 Open Dynamics Engine (ODE)

Motor de física gratuito e de código aberta. Foi criado por Russel Smith e escrito em C++ com uma interface em C. Disponível para plataforma Windows, Linux, Mac OS, XBox [21] e Playstation 2 [23], é atualizado constantemente e é utilizado em alguns jogos

[1] Software livre de modelagem 3D.

[2] COLLADA (abreviação do inglês para COLLAborative Design Activity) é um padrão de exportação e importação de arquivos criado pela Sony e usado como padrão para o console Playstation 3.

comerciais como o bloodrayne 2[30], Taxi3:eXtreme Rush [31] e 18 Wheels of Steel: Pedal to the Metal [32], e motores de jogos digitais gratuitos como o OGRE3D [33] e o Crystal Space [34].

Possui as seguintes características [12]: detecção de colisões através das bibliotecas OPCODE [35] e Gimpact [27], suporte a uma variedade de tipos de primitivas e junções, oferece espaços de colisões, onde as colisões podem ser tratadas por hierarquias e suporte a *heightfields*[3].

2.2.7 OpenTissue

Motor de física gratuito e de código aberta. Foi criado pelo departamento de computação da universidade de Copenhagen e escrito em C++. Possui implementação para plataforma Windows, Linux e MacOS. É atualizado constantemente e não existe notícia de nenhum jogo comercial usando este motor.

Possui as seguintes características [36]: detecção de colisões, suporte a uma variedade de tipos de primitivas e junções, sistemas de partículas e corpos deformáveis.

2.3 Escolha do Motor de Física

As principais características que esta dissertação quer de um motor de física são: ser gratuito e de código aberto. Além disso, outras características desejáveis são:

- Ter sido usada por jogos comerciais (pois isto valida sua capacidade);
- Ser atualizada periodicamente;
- Possuir uma boa documentação e apresentar o maior número de recursos possível;
- Possuir uma grande comunidade de desenvolvedores.

Para melhor visualizar as características dos motores de física criou-se uma tabela comparativa 2.1, atribuindo-se notas, numa escala de 0 a 10, para os diversos requisitos estabelecidos.

Para facilitar a escolha criou-se o seguinte critério de pontuação baseado na importância de cada critério: ser gratuita - 30 pontos; código aberto - 30 pontos; possuir jogos

[3]Malhas para utilização em simulação de terrenos.

Tabela 2.1: Comparativo das características dos motores de física.

Motor de Física	Gratuita	Código Aberto	Possui Jogos Comerciais	Atualizada	Documentação	Recursos
Havok	Não	Não	Sim	Sim	10	10
PhisX	Sim	Não	Sim	Sim	10	10
Newton	Sim	Não	Não	Não	7	5
Tokamak	Sim	Sim	Não	Não	2	3
Bullet	Não	Não	Sim	Sim	7	8
ODE	Sim	Sim	Sim	Sim	8	8
OpenTissue	Sim	Sim	Não	Sim	1	8

comerciais - 10 pontos; possuir atualização constante - 10 pontos; boa documentação - 10 pontos; quantidade de recursos - 10 pontos. Assim, chegou-se à pontuação para os engines escolhidos apresentada na tabela 2.2.

Tabela 2.2: Notas dos motores de física.

Motor	Pontuação
ODE	96
Bullet	85
OpenTissue	79
PhysX	70
Tokamak	65
Newton	42
Havok	40

Pelo fato de ser um motor gratuito, com código aberto, consagrado no meio do entretenimento, possuir jogos comerciais, apresentar boa documentação e recursos adequados foi escolhida a ODE como o motor a ser implementado no GUFF.

2.4 ODE

Nesta seção é apresentada a arquitetura básica do ODE, descrevendo suas principais classes. São elas: *world*(mundo), *space* (espaço), *body*(corpo), *geom*(geometria) e *joints*(junções).

- **World(mundo)**:Simulação do mundo, é nele que se criam os *bodys*, os *spaces* e os *joints*. Pode haver mais de um *world*, mas objetos de diferentes *world*s não interagem entre si.
- **Space (espaço)**:É o espaço de colisões do mundo. Esta classe é otimizada utilizando um *culling* de colisões, fazendo com que somente as colisões em potencial sejam realmente testadas.

- **Body (corpo)**:Os corpos no ODE tem as seguintes características: tamanho, massa e posição. Podem-se aplicar neles torque, velocidade e força.

- **Geom (geometria)**:São os objetos fundamentais na detecção de colisão. O ODE possui 7 tipos de geometrias: *Sphere*(esfera), *box*(caixa), *capsule*(cápsula), *ray*(raio), *triangle mesh*, *transform* e *user define* (definido pelo usuario).

- **Joint(junção)**:Os *joints* consistem nos pontos de junção entre dois objetos. O ODE possui 10 tipos de junções, conforme pode ser visto na figura 2.3: *ball-and-socket joint* (junção bola e soquete) (a), *hinge joint* (junção hinge) (b), *slider joint* (junção slider) (c), *contact joint* (junção contato) (d), *universal joint* (junção universal) (e), *hinge-2 joint* (junção hinge-2) (f), *fixed joint* (junção fixa), *angular motor joint* (junção motor angular) (g), *plane 2D joint* (junção plano 2D) e a *Rotoide and Prismatic joint* (junção rotoide e prismatico) (h).

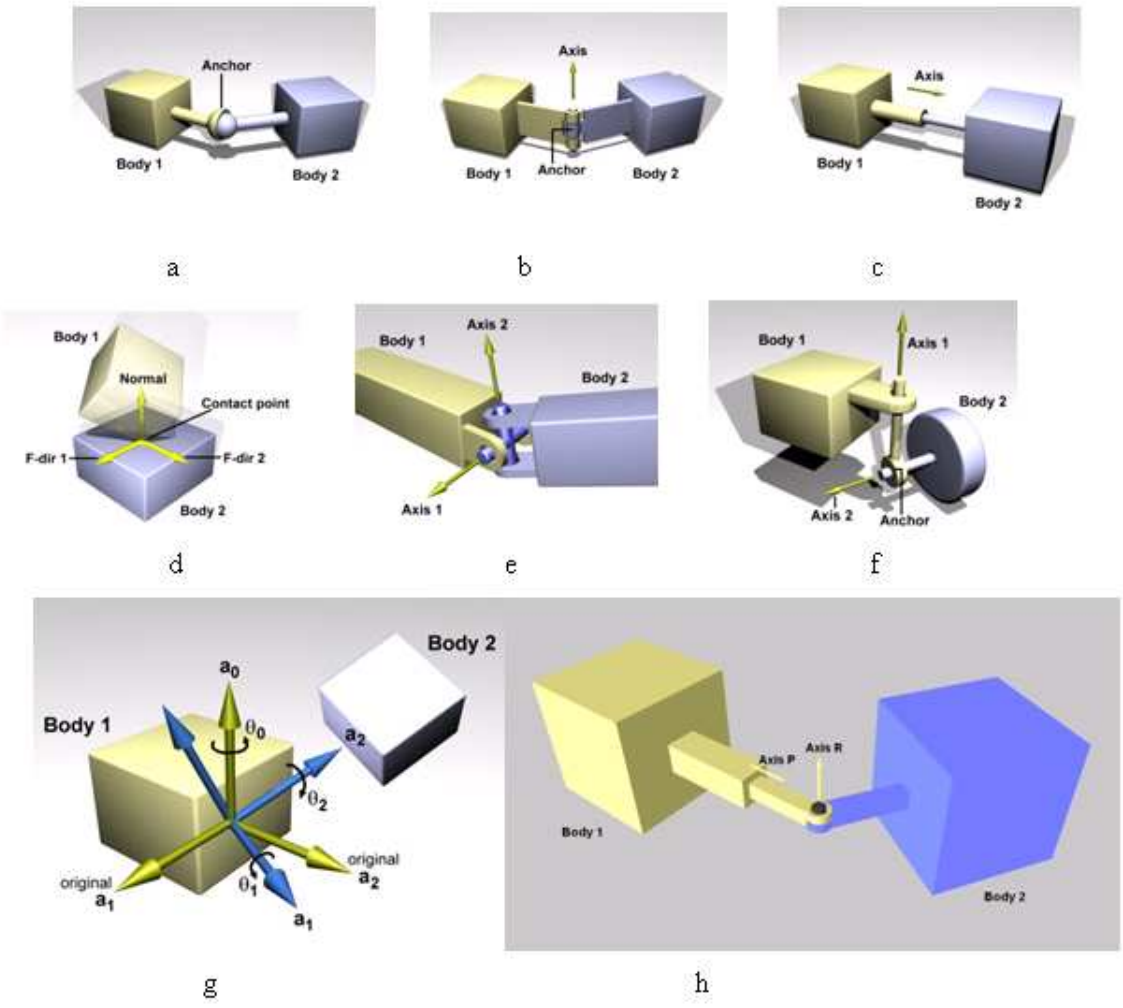

Figura 2.3: Tipos de junções suportadas pelo ODE [2].

2.5 GuffODE

O GuffODE foi implementado de forma a tirar o máximo proveito tanto da Guff quanto da ODE, bem como respeitar os requisitos da implementação do Guff, que foram mostrados na seção 2.1.1. Tendo isto em vista, foi implementado o GuffODE apartir do modelo ilustrado no diagrama de classes apresentado na figura 2.4.

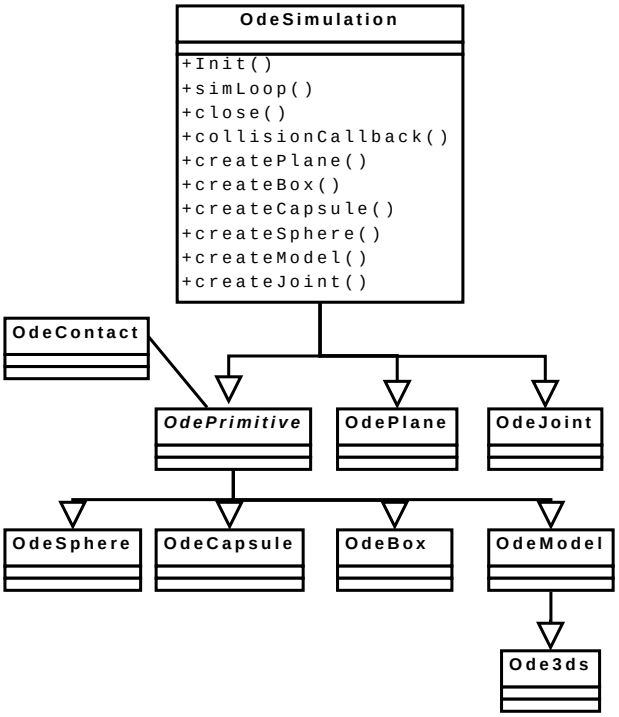

Figura 2.4: Diagrama de classes do GuffODE.

2.5.1 OdeContact

Classe que implementa como os corpos deverão se comportar após a colisão, sendo que cada corpo do GuffOde possui um OdeContact. Esta classe possui um ID que serve como identificação e também como identificador de prioridade, sendo que quanto maior o valor, maior a prioridade estipulada. Este prioridade serve para decidir de que corpo será usado o OdeContact quando dois corpos se colidirem.

2.5.2 OdePrimitive

Classe virtual que encapsula os corpos rígidos *(Body)* e as geometrias *(Geom)* do ODE. Possui um OdeContact que diz como o objeto irá se comportar quando ocorrer uma

colisão. Todas as primitivas do GuffODE são criadas pelo OdeSimulation. As classes que herdam desta classe são: OdeBox, OdeCapsule, OdeSphere e OdeModel.

2.5.3 OdePlane

Classe utilizada para encapsular os planos (*planes*) do ODE. Uma observação que se faz é que no ODE os planos são objetos estáticos que se estendem por todo o infinito. Deve ser criado pelo OdeSimulation, como exemplo o pseudo código 2.1.

```
1          OdeSimulation sim;
2          OdePlane Ground;
3          //criamos um plano no eixo XZ.
4          Ground.setGeom(sim.createPlane(0,1,0,0));
5
6          //no loop de renderização
7          Ground.render();
```

Código 2.1: Criação de um plano com o GuffODE.

2.5.4 OdeBox, OdeCapsule e OdeSphere

Classes que encapsulam primitivas simples do ODE: o OdeBox um cubo, o OdeCapsule uma cápsula e o OdeSphere uma esfera. Possui um ponteiro para uma textura para caso o objeto venha a ser texturizado. Devem ser criadas pelo OdeSimulation, como exemplo a criação de um OdeBox no pseudo código 2.2.

```
1          OdeSimulation Sim;
2          OdeBox Cubo;
3          //dimensões do cubo
4          dimensões[] = {1.0,1.0,1.0};
5          masss = 10;
6          Sim.createBox(&Cubo,massa,dimensões);
7          Cubo.setTexture("Imagem.bmp");
8          ...
9          //no loop de renderização
10         Cubo.render();
```

Código 2.2: Criação de um cubo com o GuffODE.

2.5.5 OdeModel e Ode3ds

A OdeModel é uma classe virtual que serve como classe base para os modelos 3D diferentes das primitivas simples (cubo, cápsula e esfera). Uma característica que possui é um modo de encapsulamento, isto é permite colocar o modelo 3D dentro de uma primitiva mais simples. Isto permite que a simulação fique mais leve, pois quando for realizados os testes de colisões, ao invés de realizar os testes com todos os polígonos, será realizados testes apenas com os do encapsulamento. Os modos de encapsulamento são: CBOX (que encapsula o modelo em um cubo), CSPHERE (que encapsula o modelo em uma esfera), CCAPSULE (que encapsula o modelo em uma cápsula) e CTRIMESH (que não encapsula o modelo). O modo padrão é o CBOX, sendo que o OdeSimulation calcula automaticamente as dimensões do encapsulamento durante a criação do modelo. Uma observação é que o método de encapsulamento deve ser chamado antes do OdeSimulation::createModel, já que após a chamada do OdeSimulation::createModel as primitivas mais simples já estão calculadas. Atualmente o único modelo implementado é o Ode3ds que encapsula os modelos 3ds, carregados pelo Guff. Um exemplo de criação de um modelo pode ser visto no pseudo código 2.3.

```
1          OdeSimulation Sim;
2          Ode3ds model;
3          model.setModel("model.3ds");
4          model.setCapsulateMode(CCAPSULE);
5          dReal mass = 10;
6          Sim.createModel (&model,mass);
7
8          //no loop de renderização
9          model.render();
```

Código 2.3: Criação de um Modelo 3DS com o GuffODE.

2.5.6 OdeJoint

Classe que representa uma junção genérica do ODE. Deve ser criada pela OdeSimulation. Para dizer o tipo de junção necessário, sendo que usa o método setJointType(int num). Os tipo possíveis são: JBALL, JHINGE, JSLIDER, JUNIVERSAL, JHINGE2, JFIXED, JAMOTOR e JPR.

2.5.7 OdeSimulation

Classe principal desta implementação. Nela ocorre o encapsulamento dos objetos *world* e *space* do ODE. É nela também que se criam os objetos que pertencem ao mundo a ser simulado. Aqui são implementados diversos métodos, sendo os mais relevantes:

- **Init()**:Método que faz a inicialização do ODE e do mundo a ser simulado. Pode ser invocado de duas maneiras: parametrizado com os valores padrões (que depois podem ser alterados através de funções do tipo "set") ou através de um script Lua [19],como mostra a tabela 2.3.

Tabela 2.3: Parametros do GuffODE definidos em script Lua.

Parâmetro	**Significado**	**Valor Padrão**
CFM	Constante de mistura de forças (serve para dizer se a colisão entre os objetos serão elásticas ou inelásticas)	1e-5
ERP	Parâmetro de redução do erro	0,5
NumIter	Número de iterações que o ODE fará a cada passo da simulação	20
TimeStep	Quanto o tempo deverá aumentar a cada passo	10 ms
GravityX	Força da gravidade no eixo X	0,0
GravityY	Força da gravidade no eixo Y	0,0
GravityZ	Força da gravidade no eixo Z	0,0

- **simLoop ()**: Neste método é feita a atualização da simulação. Pode ser chamado de duas maneiras: simLoop() ou simLoop(dReal timeStep). Na primeira maneira chama-se essa função no laço principal sem ter que se preocupar com o tempo da atualização, já que o OdeSimulation cuida disto. Na segunda forma deve-se preocupar com o tempo da atualização. Uma observação que se faz aqui é que na segunda forma pode-se ter passos de tempo variáveis, algo não recomendado pelo criador da ODE, pois pode conduzir a muitos erros dependendo da variação desse tempo de passo.

- **close ()**: Método que deverá ser chamado após o fim da aplicação, ou no caso de se querer criar uma outra OdeSimulation. Serve para fechar o ODE.

- **CollisionCallback ()**: Método a ser chamado pelo ODE toda vez que ocorrer uma colisão. Pode ser reimplementado, em uma classe derivada. Também pode-se utilizar a versão já implementada, que verifica quais dos dois objetos que estão colidindo possui a maior prioridade do OdeContact, e o que for maior será usado como sendo como os corpos vão se comportar após o contato.

2.5.8 Resultados

Os resultados da implementação do ODE sobre o GUFF foram satisfatórios, pois as principais funcionalidades do ODE foram mantidas assim como os requisitos do Guff, validando o acerto da escolha do motor, bem como da sua modularização. Foram criados diversos templates para auxiliar um usuário a usar o framework. Um frame de um dos templates pode ser visto na figura 2.5.

Figura 2.5: Um frame de exemplo de um dos testes com o GuffODE.

2.6 Conclusão

Neste capítulo apresentou-se inicialmente um levantamento e pesquisa sobre os principais motores de física disponíveis. Posteriormente escolheu-se a solução mais adequada para a proposta de pesquisa e para a arquitetura do GUFF, para finalmente implementar com sucesso o GuffODE.

Capítulo 3

O Motor de Fisica GDE

Novas GPUs programáveis estão trazendo novos paradigmas em suas arquiteturas, tais como CUDA (*Compute Unified Device Architecture*) [37] nas series G8X da nVidia. Estas tecnologias aumentam o poder de computação genérica que pode ser efetuada na GPU trazendo novas possibilidades para o uso de GPGPU. Este capítulo apresenta um novo motor de física, apelidado de GDE (*GPU Dynamics Engine*), que pode se utilizar tanto da CPU como da GPU para realizar parte de seus cálculos.

Há várias razões que motivam o uso da GPU para cálculos não apenas dedicados à computação gráfica: desafogar a CPU, tratar problemas de alto grau de paralelismo com maior eficiência e utilizar cálculos matemáticos implementados em hardware na GPU, quando estes não existem na CPU. Como conseqüência de implementar parte da física na GPU, tem-se a possibilidade de processar um número maior de corpos rígidos em um menor tempo, em função do paralelismo das GPUs.

Pelo fato deste capítulo apresentar funcionalidades do motor de física implementadas em GPU, na seção 3.1 apresenta-se a forma de programação GPGPU, bem como vantagens, desvantagens e trabalhos relacionados.

O restante do capítulo esta organizado da seguinte maneira: na seção 3.2 os principais conceitos de dinâmica de corpos rígidos necessários à compreensão do laço de física são introduzidos. Na seção 3.3 é descrito o laço do motor de física proposto neste trabalho. A partir de então são apresentadas as etapas deste laço que foram implementadas em GPU. Na seção 3.4 apresenta-se a detecção de colisões e na seção 3.5 o cálculo de inércia e aplicação da gravidade. Na seção 3.6 apresenta-se o integrador da equação de movimento dos corpos ou passo. Finalmente, na seção 3.7, são apresentadas as conclusões deste capítulo.

3.1 GPU

As GPUs são processadores dedicados a operações de computação gráfica. O surgimento das GPUs programáveis permitiu utilizá-las também para processar dados genéricos. Entretanto, devem ser obedecidas diversas restrições (sem o uso de CUDA), como:

- ineficiente operação de memória "scatter"(isto é, a operação de escrita de vetores não pode ser indexada);
- a precisão de ponto flutuante é menor que a da CPU (a precisão da GPU é de 32 bits enquanto que a precisão da CPU é de 64 bits);
- a programação de GPU somente pode ser realizada através de uma API gráfica [38];
- falta de operadores de inteiros como os operadores lógicos OU, E, XOU e NÃO.

Por outro lado, o desempenho da GPU é muito superior que à CPU quando considerados todos os seus processadores paralelos. Uma ATI X1900 XTX, por exemplo, pode processar em 240 GFLOPS contra 25.6 GFLOPS dos processadores Intel Pentium Extreme Edition SSE de dual-core 3.7 [39]. Pode-se comparar na figura 3.1 a evolução e desempenho das GPUs em relação às CPUs.

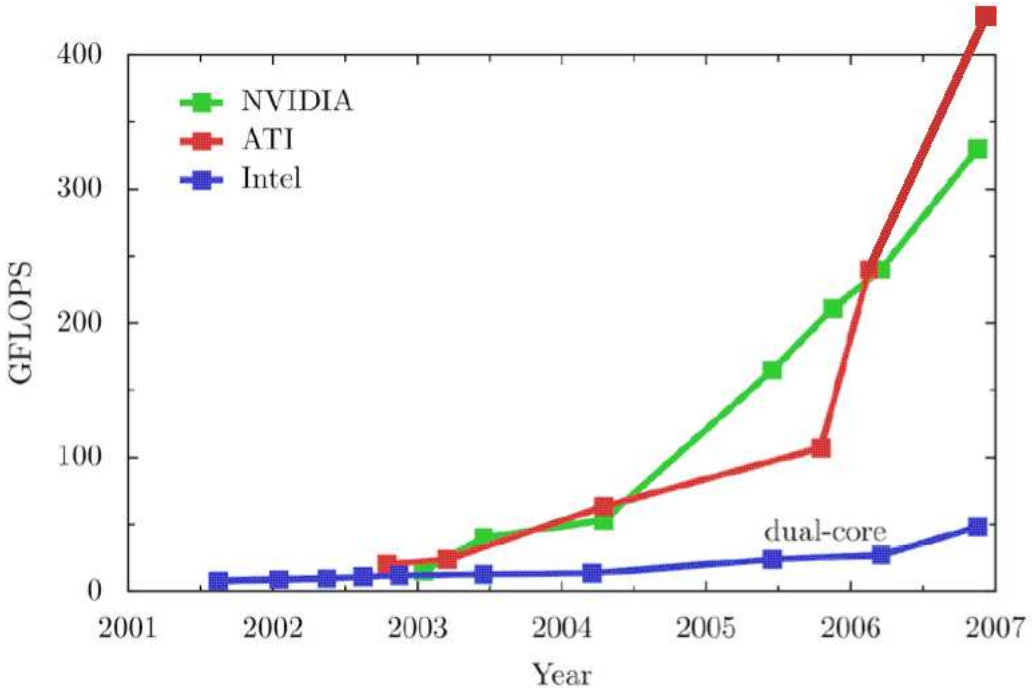

Figura 3.1: Evolução da capacidade de processamento da GPU comparada com a CPU [3].

Existem dois tipos de processadores programáveis nas GPUs (sem se utilizar da tecnologia CUDA): processadores de vértices e fragmentos. Programas executados no processador de vértices são chamados de *shaders* de vértices e os programas processados no processador de fragmento são chamados de *shaders* de fragmentos.

Neste trabalho, os métodos implementados em GPU vão se utilizar do processador de fragmentos, que é mais comum para o uso para GPGPU por duas razões [40]:

1. Uma GPU tipicamente possui mais processadores de fragmentos do que de vértices, como é o caso das GPUs das duas principais fabricantes, ATI e nVidia;

2. O resultado do processador de fragmentos vai diretamente para memória, e pode ser reutilizado como entrada por outro shader. No caso dos processadores de vértices, o resultado deve passar antes pelo rasterizador e pelo processador de fragmento para apenas depois chegar à memória.

GPUs são muito boas para processar aplicações que requerem alto grau de operações matemáticas com um grande volume de dados. Por causa de sua arquitetura paralela (por exemplo, a placa nVidia GeForce 8800 tem 128 processadores de fragmentos [41]), o desenvolvimento deste tipo de aplicação requer um paradigma diferente do tradicional modelo seqüencial usado na CPU.

3.1.1 Trabalhos Relacionados com o Uso da GPU para Propósitos Genéricos

Green [42] apresenta o motor de física Havok FX, bem como diversos de seus métodos implementados na GPU, tendo resultados de até 8 vezes mais rápidos com o uso de uma placa de video nVidia GeForce 8800GTX com um processador Intel Core 2 Duo Extreme 2.93GHz que a versão em CPU. Justificando-se a implementação de algumas funcionalidades de física na GPU ao invés da CPU devido ao alto desempenho dos processadores de fragmento, o que possibilita uma alta paralelização dos problemas que podem ser resolvidos nesta estrutura.

Além da Havok FX existem diversos trabalhos abordando a implementação de processos da simulação física na GPU, tais como: sistemas de partículas [43] , sistemas de corpos deformáveis [44], simulação de fluidos [45], simulações de ondas [46] e detecção de colisões [47].

3.2 Dinâmica de Corpos Rígidos

Seja uma cena constituída de n corpos rígidos. O *estado* $S_i(t)$ do i-ésimo corpo da cena no tempo t é caracterizado por quatro variáveis que representam as *coordenadas* e *velocidades generalizadas* do corpo [48]:

$$S_i(t) = \begin{bmatrix} x_i(t) \\ mq_i(t) \\ P_i(t) \\ L_i(t) \end{bmatrix}, \tag{3.1}$$

onde x_i é a *posição* em coordenadas globais do *centro de massa* C_i do corpo, q_i é um quatérnion que representa a *orientação* de um sistema de referência local do corpo (com origem em C_i) em relação ao sistema de coordenadas globais, e P_i e L_i são o *momento linear* e *momento angular* do corpo, respectivamente, ambos em relação ao sistema de coordenadas globais. A principal funcionalidade do motor de física é, dados o estado $S_i(t)$ e a *força* externa F_i (a força devida à gravidade é calculada pelo motor) e o *torque* externo t_i atuantes em cada corpo i em um instante t, determinar o novo estado $S_i(t + \Delta t)$ de todos os n corpos rígidos da cena no instante $t + \Delta t$, onde Δt é um *passo de tempo* da simulação. Essa determinação requer a integração numérica da *equação de movimento* de cada corpo rígido i [48]:

$$\frac{d}{dt}S_i(t) = \begin{bmatrix} v_i(t) \\ \frac{1}{2}w_i(t)q_i(t) \\ F_i(t) \\ t_i(t) \end{bmatrix}, \tag{3.2}$$

onde $v_i = m_i^{-1}P_i$ é a *velocidade linear* em coordenadas globais do centro de massa C_i do corpo, m_i é a *massa* do corpo, w_i é o quatérnion $[0, w_i]$, $w_i = I_i^{-1}L_i$ é a *velocidade angular* do corpo e a matriz 3×3 I_i é o *tensor de inércia* do corpo no instante t. Este último é calculado em t como

$$I_i(t) = R_i(t)\, I_{0i}\, R_i(t)^T, \tag{3.3}$$

onde R_i é a *matriz de rotação* 3×3 correspondente ao quatérnion q_i e I_{0i} é o tensor de inércia computado em relação ao sistema local do corpo rígido i no momento de sua criação. Se a densidade (e em conseqüência a massa) e a geometria do corpo for constante ao longo da simulação, I_{0i} também será (geralmente é mais eficiente calcular $I_i(t)$ através da Equação (3.3) do que com as equações integrais de definição de momento de inércia [49]).

A Equação (3.2) é uma equação diferencial ordinária (EDO) de primeira ordem. O componente do motor de física responsável pela sua integração é denominado *solucionador*

de EDO. Este método é utilizado no passo da simulação.

Geralmente, o movimento de um corpo rígido não é livre, mas sujeito a *restrições* que eliminam um ou mais *graus de liberdade* do corpo ou impedem que dois corpos se interpenetrem. (Um corpo rígido sem restrições tem 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). O tratamento de restrições a ser implementado no motor proposto introduz, para cada restrição aplicada a um corpo, uma *força de restrição* incógnita que deve ser determinada e aplicado ao corpo Dois tipos de restrições serão considerados no motor: (1) as impostas por *junções* entre (normalmente dois) corpos, e (2) resultantes do *contato* (de colisão ou repouso) entre dois ou mais corpos [48].

A fim de computar as forças de restrições que previnem a interpenetração dos corpos rígidos em contato, o motor deve conhecer no tempo t o conjunto de *pontos de contato* entre cada par de corpos da cena. (A *detecção de colisões* não é propriamente função do motor de física, mas de um componente a ele integrado responsável por esta tarefa.) Uma vez detectados os pontos de contato em t, o motor deve determinar simultaneamente (pois uma pode influenciar no efeito de outra) as forças de restrições devidas a junções e contatos e então aplicá-las aos respectivos corpos para proceder à integração da equação de movimento. Matematicamente, esta condição pode ser formulada como um *problema de complementaridade linear* (PCL) misto, o qual pode ser eficientemente resolvido, por exemplo, pelo algoritmo de Dantzig [50]. O componente do motor de física responsável por esta tarefa é denominado *solucionador de PLC*.

3.3 O Laço de Física

O GDE usa uma versão modificada da versão 0.8 do motor ODE, descrita no capítulo anterior.

Sumarizando o laço do motor de física, discutido na seção anterior, é responsável por:

- Detectar colisões entre os corpos;
- Transformar a matriz de tensão de inércia de coordenadas locais para coordenadas globais;
- Aplicar a força da gravidade nos corpos;

- Colocar todas as junções e contatos como um problema de complementaridade linear (PCL) e resolvê-lo;
- Atualizar as forças que atuam nos corpos de acordo com o resultado do PCL;
- Calcular um passo na simulação para cada corpo, calculando a nova posição e velocidades de acordo com as forças e o tempo.

Estes passos podem ser vistos na figura 3.2

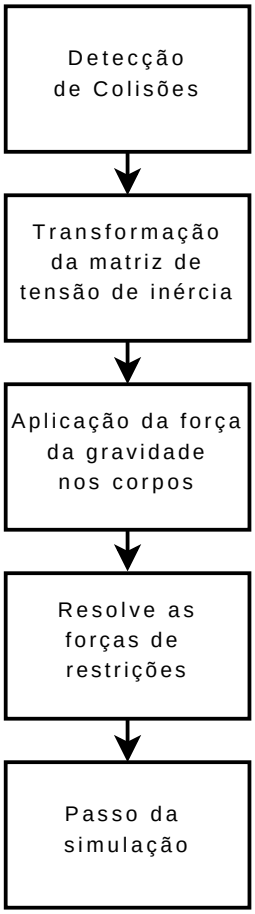

Figura 3.2: O Loop do motor de física GDE.

3.4 Detecção de Colisões

Realizar a detecção de colisões é uma tarefa árdua. Normalmente a tarefa de detectar as colisões é feita em duas etapas: a primeira chamada de *broad phase* e a segunda *narrow phase*. Na *broad phase* o motor detecta quais corpos têm probabilidade de colidirem entre si, descartando os que não têm chance. E na *narrow phase* são realizados testes de colisões entre os pares que passaram pela *broad phase*.

É nesta segunda fase onde realmente se realizam as colisões dos corpos. Ela calcula o ponto onde os corpos se interceptam, a profundidade desta interceptação e a normal da colisão. Esta fase tem cálculos muito intensos então não pode ser feita em tempo real para todos os corpos se a simulação tiver um grande número de corpos. Ela precisa de uma *broad phase* para retirar os corpos que não possuem chance de colidir.

A *broad phase* pode ser feita de muitas maneiras: usando um algoritmo de grades e *axis aligned bounding box*, algoritmo de uma *axis aligned bounding box*, algoritmo de uma *sphere bounding*, etc.

Nesta dissertação serão usados os *sphere bounding*. Esta abordagem possui algumas vantagens em relação as *axis aligned bounding box*, pois ela não necessita ser modificada conforme o corpo sofre transformação de posição e rotação, já que uma esfera não muda de raio quando ocorre rotação. O algoritmo para realizar este teste é muito simples e consiste em verificar se os corpos estão a uma distancia d (calculado pela equação 3.4) menor que a soma dos raios ($d < r_1 + r_2$), se isso ocorrer houve colisão entre ambas as esferas, conforme ilustrado na figura 3.3.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{3.4}$$

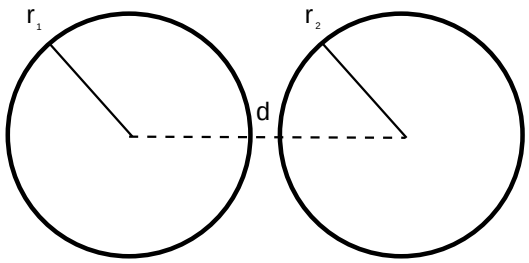

Figura 3.3: duas *spheres boundings* próximas.

3.4.1 Colisão na GPU

A *broad phase* está presente na GDE e foi implementada tanto na CPU quanto na GPU. Para a implementação na GPU é necessário modelar o problema adequadamente, utilizando a memória de texturas. Para isto utiliza-se uma estrutura onde cada linha da textura possui a representação de 4 corpos, como mostra a figura 3.4.

Para realizar os cálculos e para melhor aproveitar o paralelismo da GPU, faz-se a detecção da seguinte maneira: realiza-se os testes da primeira linha contra todas as outras linhas num passo de GPU. A seguir, realiza-se a segunda linha contra todos os corpos,

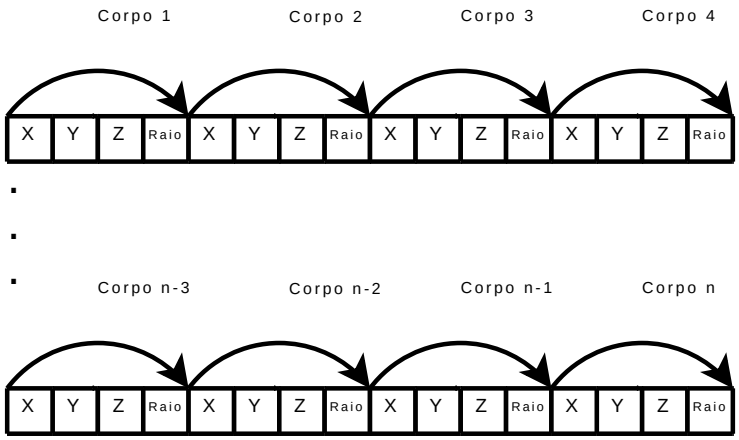

Figura 3.4: Estrutura dos corpos escritas num espaço de textura.

menos os da primeira linha, em outro passo da GPU e assim por diante, até serem realizados todos os testes. O resultado de cada teste de linha é guardado numa textura para posteriormente, na *narrow phase*, serem verificados em CPU. Os resultados deste teste são guardados em uma textura de tamanho n, sendo colocado 1, caso o par de corpos terem chance de colidir, ou 0, caso os corpos não terem chance de colidir. Pode-se ver como exemplo uma ilustração do teste do primeiro corpo contra os quatro últimos, com o resultado sendo posto em um texel, na figura 3.5.

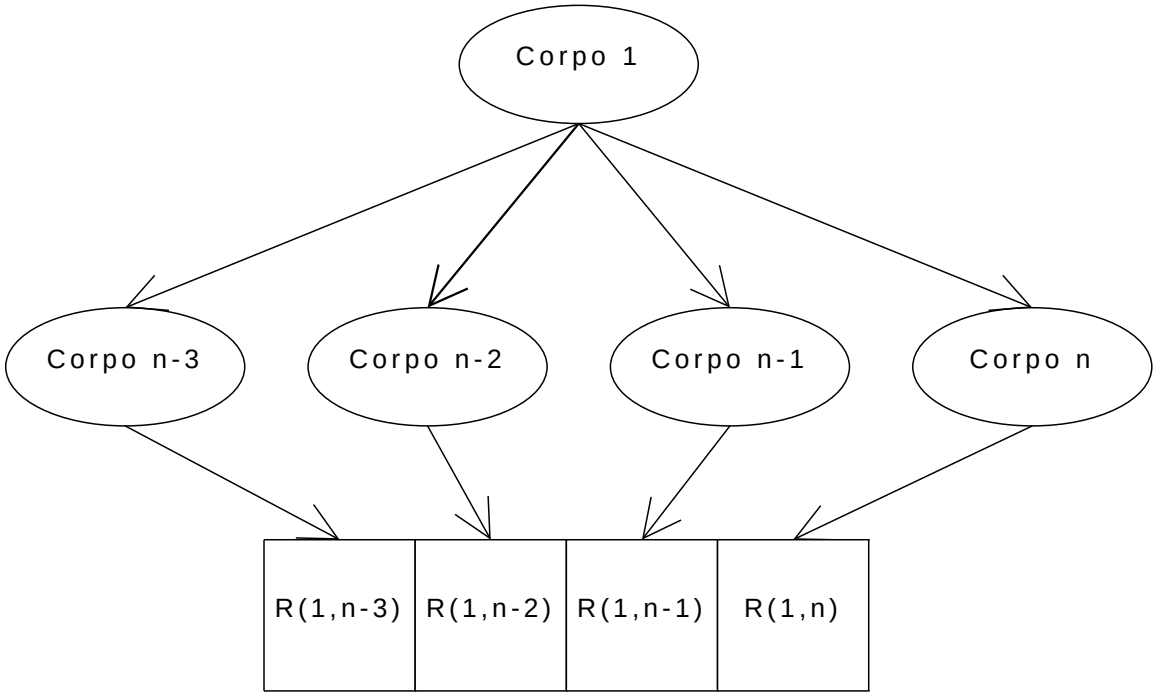

Figura 3.5: Exemplo de um teste de colisão na GPU.

O número máximo de corpos que este método, e por conseqüência a GDE, pode processar na GPU é 16384 corpos.

3.5 Cálculo do Tensor de Inércia e Aplicação da Gravidade nos Corpos

O cálculo de inércia e a aplicação da gravidade nos corpos são pré-cálculos que vão ser utilizados pelos métodos posteriores do laço, o PCL e o passo. Estes métodos estão no GDE implementados na GPU como um único shader.

Esta seção esta dividida em 3 subseções: transformação da matriz de inércia, aplicação da gravidade nos corpos e a implementação em GPU.

3.5.1 Transformação da Matriz de Inércia

A importância do tensor de inércia consiste em aproximar a forma de como a massa do corpo rígido é distribuída em relação ao centro de massa do corpo, conforme descrito na seção 3.2.

O cálculo de transformação do tensor de inércia de coordenadas locais para coordenadas globais é simples: o método recebe como entrada uma matriz I_0 de dimensão 3x3 com a tensão de inércia do corpo em coordenadas locais, pré-calculada quando o corpo é criado, e um quatérnion q que representa a orientação do sistema local do corpo em relação ao sistema de coordenadas globais. Então o método transforma o tensor de inércia em coordenadas locais I_0 para coordenadas globais I de acordo com a equação 3.5.

$$I = RI_0R^T, \tag{3.5}$$

onde R é a matriz de rotação $3x3$ correspondente ao quatérnion q.

O método então retorna a nova matriz de inércia I em coordenadas globais. Este método pode ser melhor visualizado pelo código 3.1.

3.5.2 Aplicação da Força da Gravidade nos Corpos

A força da gravidade é a força que atrai os corpos para a terra. O cálculo da aplicação desta força é simples: o método recebe como entrada a gravidade g, a força aplicada F no corpo e a massa m do corpo. A força resultante F_r é calculada de acordo com a equação 3.7, adicionando a F o componente F_g, computado de acordo com a segunda lei de Newton na equação 3.6.

```
1  Transformar_Inercia(I0,q)
2  {
3          //transforma o quaternio em uma matriz
4          R = quatParaMatriz(q);
5          //transpõe a matriz
6          Rt = transpMatriz(R);
7          //transforma a matriz de inércia
8          I = transLocalParaGlobal(I0, R, Rt);
9
10         Retorna I;
11 }
```

Código 3.1: Método de transformação da matriz de inércia de coordenadas locais para coordenadas globais.

$$F_g = mg. \tag{3.6}$$

$$F_r = F + F_g. \tag{3.7}$$

O método então retorna a força resultante F_r. Este método pode ser melhor visualizado no código 3.2.

```
1  AplicaçãoGravidade(F,m,g)
2  {
3          //calcula a força da gravidade
4          Fg = m * g;
5
6          //calcula a força resultante
7          Fr = F + Fg;
8
9          Retorna Fr;
10 }
```

Código 3.2: Método de aplicação da força da gravidade nos corpos.

3.5.3 Implementação na GPU

Os dois métodos descritos nesta seção foram implementados em GPU. Para que isto ocorra foi necessário modelar os dados adaptando-os a estrutura de memória de textura e *texels*, solicitada pela GPU. A matriz de inércia foi enviada como sendo três texturas, uma para cada linha. O quatérnion como outra textura e a força aplicada sobre o corpo como ainda

outra textura. Em cada uma destas texturas, cada *texel* contém os dados de um corpo. A gravidade foi enviada como uma variável uniforme. Os dados de saída assemelham-se aos de entrada: a matriz de inércia está em coordenadas globais como três texturas, uma para cada linha e a força resultante como outra textura.

Em resumo, este shader envia cinco texturas para a GPU e recebe quatro texturas dela. Esse shader será referenciado pela presente dissertação como shader de inércia daqui em diante.

3.6 Passo da simulação

O passo da simulação é responsável por integrar as equações de movimento dos corpos. Esse método recebe as informações sobre o estado corrente de um corpo no tempo t (i.e., sua posição p, quatérnion q, velocidade linear v, e velocidade angular ω), a sua massa invertida m^{-1}, a o tensor de inércia em coordenada globais I, as forças aplicadas F, e o torque τ. O método também recebe o tempo do passo Δt. O método calcula o novo estado, i.e., a nova posição, quatérnion, velocidade linear e a velocidade angular do corpo rígido no tempo $t + \Delta t$, aplicando um esquema de integração numérica. O método de cálculo da rotação do corpo, i.e., o método de cálculo do novo quatérnion pode ser feito de três diferentes maneiras, para o caso de se querer restringir a rotação de um corpo como é o caso de algumas junções, que são: uma rotação infinita (isto é, uma rotação normal e sem restrições), uma rotação finita (isto é, uma rotação com limite de ângulo) e uma rotação finita ao redor de um determinado eixo E que é também passado ao método (isto é, uma rotação com limite de ângulo e ao redor de um determinado eixo). No código 3.3 este método pode ser visto melhor.

3.6.1 Passo na GPU

O presente trabalho implementou este método com sucesso na GPU. Para isto foi necessário modelar os dados de entrada como textura e *texels*. Cada *texel* de uma textura representa os dados de um corpo. Para texturas utiliza-se:

- Uma textura para a posição dos corpos;
- Uma textura para a velocidade linear junto com um sinalizador indicando se há rotação finita;

```
1  Passo(p, v, F, w, T, I, q, E, t )
2  {
3          //calcula a nova velocidade linear (v)
4          calcVel(v, F);
5
6          //calculanova posição(p)
7          calcPos(p, v, t);
8
9          //calcula nova velocidade angular(w)
10         calcVelA(w, T, I, t);
11
12         //se deve ser feita uma rotação finita
13         Se(rotação_finita)
14         {
15                 //se deve ser feita ao redor de um eixo
16                 Se(rotação_ao_redor_de_eixo)
17                 //calcula uma rotação finita ao redor de um eixo (E)
18                 calcRotEixo(q, E, w);
19                 Senão
20                 //se somente rotação finita
21                 calcRotFinita(q, w);
22         }
23         else
24                 //Se não tem restrições calcula uma rotação infinita
25                 calcRotInfinite(q, w);
26
27         //returna o novo estado
28         returna p, q, v, w;
29 }
```

Código 3.3: Método de aplicação de um passo nos corpos.

- Uma textura para a velocidade angular junto com a massa invertida;
- Uma textura para o quatérnion;
- Uma textura para o torque;
- Três texturas para a matriz de inércia em coordenadas globais;
- Uma variavel uniforme para o passo;
- Uma textura para o eixo com um sinalizador indicando se deve realizar uma rotação ao redor do eixo.

Em resumo, esse método envia dez texturas para a GPU e recebe quatro.

3.7 Otimização da GPU

Nesta seção serão apresentados alguns mecanismos implementados para otimizar o processo da GPU.

3.7.1 Colocando os *Shaders* para Trabalharem Seqüencialmente

O gargalo da GPU,geralmente, são as transferências de texturas. Usando o *shader* de passo com os dados do *shader* de inércia e do *shader* de colisão é possível diminuir este tráfego, pois o *shader* de passo utiliza-se de dados calculados e recebidos pelo *shader* de inércia e também da mesma posição passada para o *shader* de colisão.

O *shader* de passo usa um quatérnion, que não é alterado entre o *shader* de inércia e o de passo, podendo ser enviado para a GPU uma única vez. Os resultados do *shader* de inércia, o tensor de inércia em coordenadas globais e as forças calculadas são as entradas do *shader* de passo, podendo ser utilizadas sem ter que ser reenviadas. Com isso, o *shader* de passo somente terá que receber 4 texturas ao invés de 10, quando não se efetua o reuso de texturas.

Se a simulação não possuir o método PCL, i.e. sem junções e contatos entre os corpos, ele não tem que receber os dados do shader de inércia para a CPU, i.e. a matriz de inércia em coordenadas globais e as forças calculadas. Com isso os métodos reutilizando os dados e sem o PCL tem juntos dez envios de texturas e quatro recebimento de texturas contra quinze envios e oito recebimentos quando os *shaders* não reaproveitam as texturas.

Por outro lado, se a simulação possui contatos ou junções, será necessário ajustar e resolver o PCL. Para isso, os resultados do shader de inércia têm de ser passados para CPU. Quando isso ocorre, os métodos têm dez envios de texturas e oito recebimentos de texturas.

3.8 Reutilização de Dados

Uma outra forma implementada para otimizar o desempenho da GPU foi à reutilização de dados. Quando se utiliza de um motor de física para aplicações tradicionais como jogos digitais, normalmente, não se modificam as tensões de inércia de um corpo. Este seria um exemplo onde só é necessário enviar a textura para a GPU quando este dado for modificado fora da GPU. O mesmo ocorre com a posição, quatérnion, velocidade linear,

força, velocidade angular e o eixo de rotação.

Para que isto ocorra foi implementado, dentro da estrutura interna do motor, sinalizadores. Quando os dados forem modificados fora da GPU estes são ativados e os dados são reenviados para a GPU.

3.9 Conclusão

Neste capítulo apresentou-se o motor de física GDE bem como suas principais funcionalidades e as suas respectivas funcionalidades na GPU. Os testes com este motor, bem como a discusão dos resultados, se encontram no capítulo 6.

Capítulo 4

Arquiteturas para Games e Simulações Físicas

As arquiteturas sistemas de simulações interativas em tempo real são tradicionalmente divididas em três módulos principais [51]:

- Aquisição de dados de entrada;
- Processamento dos dados;
- Visualização dos resultados.

A aquisição de dados de entrada em geral é feita por teclado e mouse, mas pode ser também feita por joysticks.

O módulo de processamento é onde o motor de física se enquadra e onde são realizados os cálculos referentes à atualização dos corpos no mundo. Nesta etapa, para se aumentar o realismo em relação ao comportamento inteligente de alguns elementos da cena, também podem ser colocados os processamentos de Inteligência Artificial, quando se trata de jogos digitais.

Na visualização dos resultados ocorre a renderização da cena que o usuário verá o resultado na tela do monitor.

Uma das restrições das simulações em tempo real é o tempo de resposta. Ele não pode ser muito alto ou o principal requisito da aplicação falha. Uma das métricas mais comuns para se medir este aspecto consiste na contagem de quadros por segundo ou FPS. Como mínimo de FPS em uma aplicação 3D em tempo real tem-se comumente de 16 [52]. Qualquer valor acima de 30 já é considerado um valor ótimo, uma vez que o olho humano é capaz de capturar imagens a 25 FPS, aproximadamente. Quando a taxa é maior

que 60 FPS muitos quadros calculados e desenhados são descartados, já que o monitor normalmente tem uma faixa de atualização de 60 Hz, ou seja 60 atualizações de tela por segundo. Desta forma, uma aplicação ótima deve possuir taxas entre 30 e 60 FPSs para não ser perceptível pelo olho humano mudanças de quadro e para que os cálculos e desenho realizados pela aplicação não sejam descartados.

Este capítulo apresenta diversos modelos de arquitetura para simulações em tempo real pensando principalmente aquelas que utilizam motores de física, mais especificamente o GDE. Na seção 4.1 são apresentadas as formas de atualização nas arquiteturas bem como discussões sobre as melhores formas de utilizá-las. Na seção 4.2 apresenta-se diversos modelos de arquiteturas e na seção 4.3 é apresentada a arquitetura que será utilizada nos testes. A seção 4.4 apresenta as conclusões deste capítulo.

4.1 Formas de Atualização

Para facilitar as discussões da próxima seção serão apresentadas aqui as formas de atualização de uma cena. Os módulos que precisam realizar a atualização são: o processamento da cena e renderização da cena.

Para isto, existem diversas estratégias de se atualizar uma cena:

- Freqüência livre;
- Freqüência por tempo entre os quadros;
- Freqüência determinada.

Para facilitar a apresentação separaram-se estas maneiras em diferentes subseções.

4.1.1 Atualização por Freqüência Livre

A atualização por freqüência livre pode ser descrita como a estratégia que procura atualizar a cena o mais rápido que for possível. Como para as funções de processamento da cena pelo motor de física devem usar como parâmetro de entrada o tempo de passo, este tempo, neste caso, é dado como um valor constante, como por exemplo, um milisegundo.

Este tipo de atualização é muito comum de se ver implementado em simulações de tempo real, pelo fato de ser o mais simples de se implementar.

Um dos inconvenientes deste método consiste em que este tipo de atualização da renderização, feito em computadores rápidos, pode rodar a mais de 60 FPS fazendo com que diversos quadros calculados e renderizados sejam descartados. Em contrapartida existe a vantagem que para a renderização neste modo se utiliza de todo o potencial da placa gráfica.

Um inconveniente deste tipo de atualização para motores de física consiste que para diferentes computadores a aplicação irá funcionar de maneira diferente (com taxas de atualização diferentes). Para se contornar isto se pode colocar o tempo transcorrido entre os quadros como um parâmetro de entrada do tempo de passo do motor de física.

4.1.2 Atualização por Freqüência entre os Quadros

A atualização por freqüência entre quadros pode também ser descrita por atualizar a cena o mais rápido que puder. Para a renderização este modo se comporta exatamente igual a atualização por freqüência livre, com as suas vantagens e desvantagens. Entretanto, para a atualização do motor de física, o parâmetro de entrada do tempo de passo passa a ser o tempo transcorrido entre os passos da simulação.

Uma das vantagens sobre a atualização por tempo livre é que a aplicação funcionará de maneira semelhante em diferentes computadores desde que eles atendam os requisitos mínimos para a execução da aplicação, pois a taxa de atualização da cena permanece constante.

Um inconveniente é que para os motores de física, pelo fato de a freqüência ser variada, pode surgir erros no motor de física. Assim, considere-se como exemplo o teste de colisão: pelo fato da maioria dos motores de física utilizarem testes de colisão simples, se por acaso entre, um quadro e outro o tempo de passo for alto, pode ocorrer de um corpo passar pelo o outro sem colidir, pelo fato de se ter dado um salto entre um ponto e outro. Outra situação que pode gerar problemas está no integrador ou passo, que devido a modificação dos tempos de entrada pode ocasionar erros em seus cálculos.

4.1.3 Atualização por Freqüência Determinada

Neste tipo de atualização a freqüência em que se atualiza a renderização e a cena são controladas por uma freqüência determinada. Esta freqüência pode ser a mesma da renderização, bem como pode ser diferente. Normalmente o tempo de atualização da cena pelo motor de física é menor que o da atualização da renderização, em situações

onde os cálculos podem ser muito intensos. Mesmo com esta taxa de atualização de física, esta arquitetura permite que o FPSs de atualização da renderização seja de 30 à 60 FPSs, como discutido anteriormente.

Neste tipo de abordagem a simulação se livra das desvantagens de ter quadros descartados e de ocorrerem erros na simulação física. Em contrapartida utilizando-se de freqüência determinada para renderização faz com que a simulação não rode com todo o potencial que ele poderia.

4.2 Modelos de Arquitetura

Esta seção apresenta diversos modelos de arquitetura, tendo como principal foco games ou aplicações em tempo real com uso de motores de física. Um trabalho similar a este foi desenvolvido em [52] com abordagem em games, sem levar em consideração processamento GPGPU.

4.2.1 Laço Acoplado Simples

A arquitetura mais simples de se processar um laço de jogos digitais é de maneira seqüencial [51] como mostra a figura 4.1.

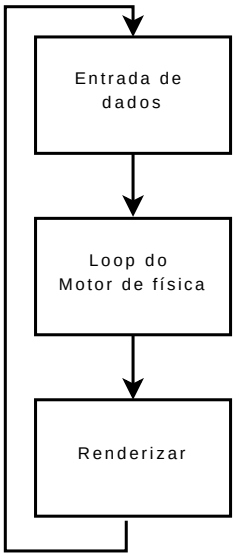

Figura 4.1: Laço acoplado simples.

A maior vantagem deste tipo de arquitetura é a sua simplicidade de implementar e especialmente de executar. Entre as desvantagens está o fato de que quando a etapa do motor de física estiver muito sobrecarregada (caso que poderá ocorrer quando há muitos

corpos rígidos numa cena, por exemplo), a renderização será prejudicada, pois o *pipeline* irá esperar pelo final da atualização dos corpos para depois proceder com a renderização. O mesmo ocorre para a entrada do usuário. Pode também ocorrer que a renderização seja o gargalo do laço, o que poderá ocorrer quando a cena possui muitos polígonos para desenhar na tela, por exemplo. Neste caso, a atualização dos corpos rígidos pelo motor de física será atrasada, bem como a entrada do usuário. Para contornar isto serão apresentadas duas arquiteturas diferentes, com etapas desacopladas entre si.

4.2.2 Laço com Renderização Desacoplada

Para se evitar as desvantagens do modelo simples acoplado, pode-se criar uma thread separada para ficar responsável pela renderização [52], como mostra a figura 4.2.

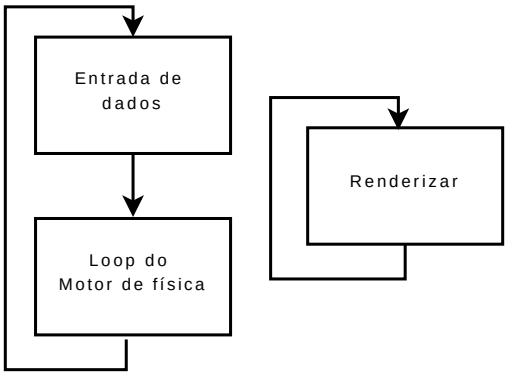

Figura 4.2: Laço com a renderização desacoplada.

Este tipo de atualização evita as desvantagens da simples acoplada. Por outro lado, traz uma desvantagem, que consiste numa implementação mais complicada do que no simples acoplado.

Este tipo de arquitetura apresenta maior eficiência em cenas complexas para renderização (como, por exemplo, cenas com elevado número de polígonos) e elementos simples para o motor de física (como, por exemplo, um baixo número de corpos rígidos), pois caso contrário a entrada do usuário ainda pode ficar comprometida.

4.2.3 Laço com o Motor de Física Desacoplada

Outro modo de se evitar as desvantagens da arquitetura simples acoplada é desacoplando o módulo do motor de física como pode ser visto na figura 4.3.

Este tipo de atualização, como a com renderização desacoplada, evita as desvantagens

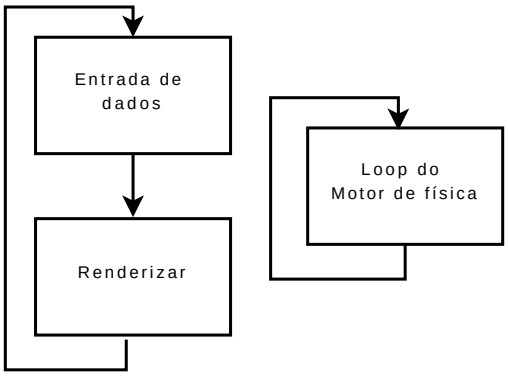

Figura 4.3: Laço com o motor de física desacoplado.

da simples acoplada. Entretanto, uma desvantagem desta arquitetura é que a implementação é mais complicada do que no simples acoplado.

Este tipo de arquitetura terá maior eficiência em cenas complexas para o motor de física, mas simples para a renderização, pois caso contrário à entrada do usuário pode ficar comprometida.

4.3 Arquitetura Escolhida

Para utilização nos testes em tempo real do motor de física GDE foi a escolhida a arquitetura com motor de física desacoplado com freqüência determinada, com o motor de física atualizado a uma freqüência de 20 FPS e com a renderização atualizada sempre que possível.

Esta escolha se deu pelas seguintes razões:

- Evitar erros no motor de física;
- Utilizar todo o potencial da placa gráfica;
- Utilizar uma arquitetura de forma que a simulação se comporte de maneira similar em diversos tipos de computadores;
- Concentrar em simulações com baixo número de polígonos e alto número de corpos rígidos, focando a aplicação no realismo físico.

Os resultados desta arquitetura podem ser vistos no capítulo 6.

4.4 Conclusão

Neste capítulo foram apresentadas diversas arquiteturas para um jogo digital e aplicações de simulação e visualização em tempo real, bem como vantagens e desvantagens delas. Apresentou-se também a arquitetura que será utilizada com a GDE e com o GUFF.

Capítulo 5

Distribuição entre CPU e GPU

Com o uso de métodos implementados, tanto em CPU como em GPU, é necessário decidir como distribuir as tarefas entre CPU e GPU de forma a maximizar o uso dos recursos disponíveis. Esta distribuição pode ser feitas de diversas formas: definida pelo desenvolvedor ou usuário (seção 5.1), definida automaticamente através de módulos de inteligência artificial e lógica fuzzy, conforme apresentado por Zamith [53] (seção 5.2) ou definida por heurísticas simples para fazer uma distribuição automática (seção 5.3).

5.1 Modo de Decisão do Usuário

Uma das formas de se decidir entre a utilização da CPU ou da GPU é através de funções chamadas diretamente pelo usuário. Para isto, deve-se implementar uma função de forma que o desenvolvedor escolha qual processador irá processar as tarefas através de um parâmetro. Esta função está implementada no GDE e apresenta os seguintes parâmetros: *world* que é o mundo da simulação utilizado e *mode* que é o modo de processamento (sendo passado verdadeiro para realizar o processamento na GPU e falso para realizar o processamento na CPU). Esta função pode ser vista abaixo:

dWorldSetUseGpu(world,mode);

Outra maneira do desenvolvedor escolher é através do encapsulamento, conforme apresentado no capítulo 2, porém ao invés de chamar o método GuffODE utiliza-se o GuffGDE, que apresenta as mesmas funcionalidades do GuffODE mas com o motor de física GDE ao invés do ODE. O exemplo abaixo ilustra esta situação:

GDESimulation::setUseGpu(bool mode);

Finalmente, também se pode permitir que o usuário final decida qual o processador a ser utilizado, através de um script Lua, baseada na idéia desenvolvida em [53]. Neste script define-se como exemplo:

$useGPU = true$

5.2 Decisão através de uma Abordagem de Inteligência Artificial

Esta solução tem como entrada o tempo gasto e o FPS dos processos realizados tanto em GPU como em CPU. Mediante estes dados, através de um script Lua, o módulo decide qual processador vai executar o restante das tarefas. Este módulo é separado em três etapas: conversão das variáveis de entrada; máquina de inferência de acordo com um banco de regras; e a conversão do conjunto fuzzy numa decisão CPU ou GPU. Estas etapas podem ser vistas na figura 5.1.

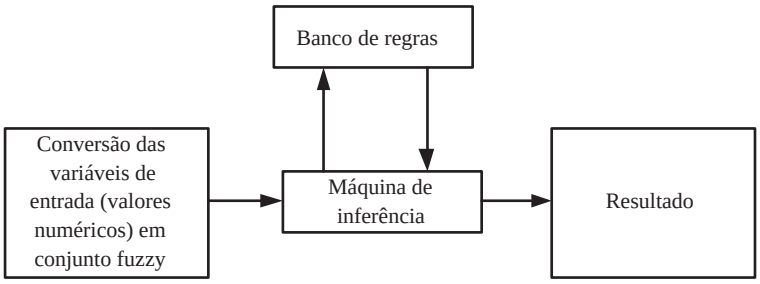

Figura 5.1: Modelo de diagrama de inferência.

Na conversão das variáveisé onde os valores númericos de entrada são transformados em valores qualitativos. Neste caso, o FPS e o tempo gasto tem um grau de pertinência em um dos cinco conjuntos ("muito baixo", "baixo", "normal", "rápido" e "muito rápido"). De acordo com estes dois graus de pertinência cria-se a base de conhecimento para cada processador. Esta base é usada pela maquina de inferência para realizar a decisão. Um exemplo de decisão pode ser visto abaixo:

Se $<CPU == muito\ fácil>$ **e** $<GPU == normal>$ **Então** $<Processador = GPU>$.

Para testar o GDE com este módulo utilizou-se o modelo de arquitetura apresentada na figura 5.2. No caso as duas threads possuem um ponteiro para o mesmo objeto que possui a simulação (GDESimulation), sendo que na thread da CPU é invocada uma função para processar na CPU enquanto que na thread da GPU é invocada uma função para processar na GPU.

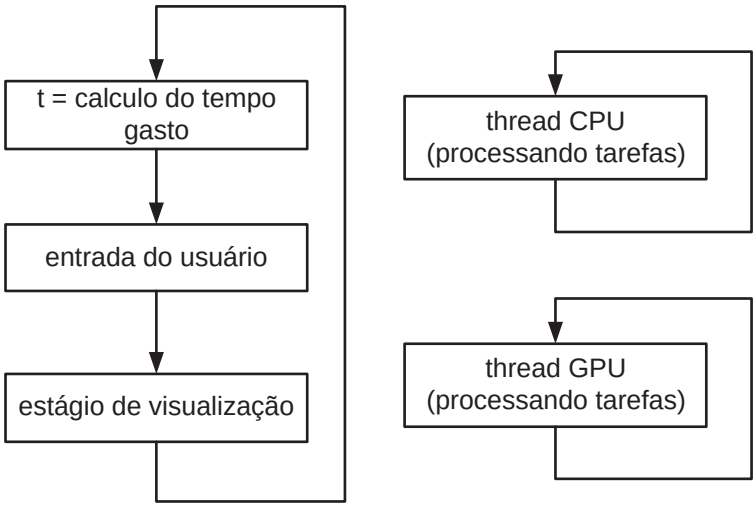

Figura 5.2: Modelo *multithread* desacoplado com distribuição dinamica de tarefas.

Mais informações sobre esta abordagem podem ser encontradas em [53].

5.3 Heurísticas de Decisão

Para o uso de heurísticas de decisão com o GDE criou-se uma classe base virtual de onde todas as heurísticas são herdadas chamada de *Distributor*. Para fazer uso do *Distributor* todos os métodos que são implementados em CPU-GPU tem o seu tempo contado por ele e a cada frame é pedida uma decisão sobre como processar o próximo frame, conforme mostrado no fluxograma da figura 5.3.

Para fazer uso desta estratégia de distribuição, as classes herdadas do *Distributor* devem implementar o método abstrato denominado de *decideMode*. Desenvolveu-se então diversos modos de realizar esta distribuição: modo desenvolvedor, modo início, modo tempo real, modo recurso e modo automático. Todas estas classes podem ser vistas no diagrama UML da figura 5.4.

5.3.1 Modo Desenvolvedor

Neste modo a decisão de como se distribuir fica a cargo do desenvolvedor que vai utilizar o GDE. Para isto o desenvolvedor passa para o GDE como parâmetro um ponteiro de função com o modo que a decisão deverá ser realizada. Para realizar esta decisão o desenvolvedor tem acesso aos dados dessa função parametrizada, que são: tempo gasto na CPU e o tempo gasto na GPU ambos calculados pela classe mãe *Distributor*.

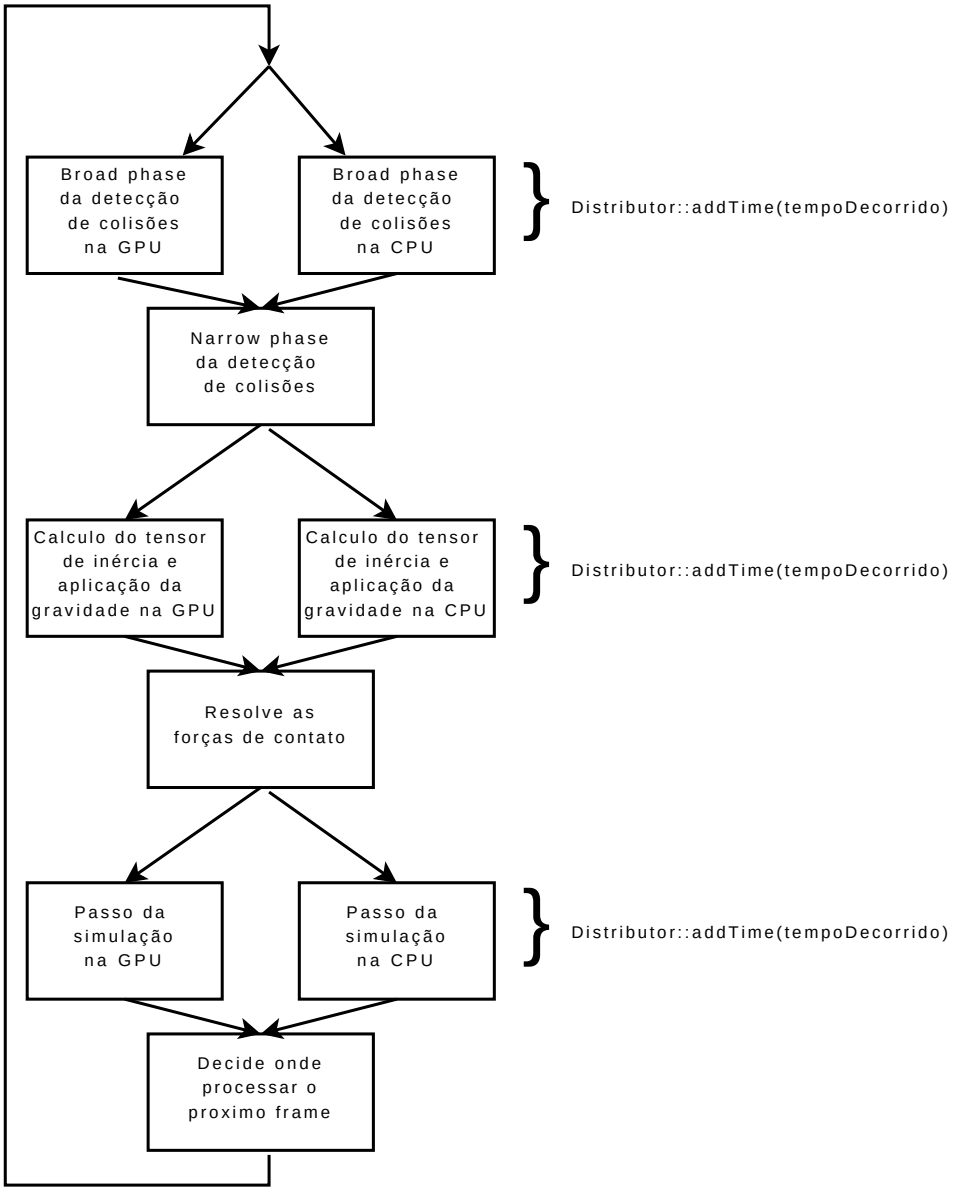

Figura 5.3: Fluxograma do loop de física com o distributor.

5.3.2 Modo Início

Neste modo a decisão fica a cargo do GDE. Este módulo utiliza a seguinte estratégia: calcula 10 frames na GPU e em seguida 10 frames da CPU. A partir dai os próximos frames serão calculados no processador que apresentou o resultado mais rápido dos dois métodos, como mostra o código 5.1.

Pode-se ver que este método, em condições normais, sempre escolherá o processador mais rápido para processar as tarefas, sem necessitar que um usuário ou o desenvolvedor o selecione.

Uma desvantagem deste modo, assim como o módulo de decisão com Inteligência

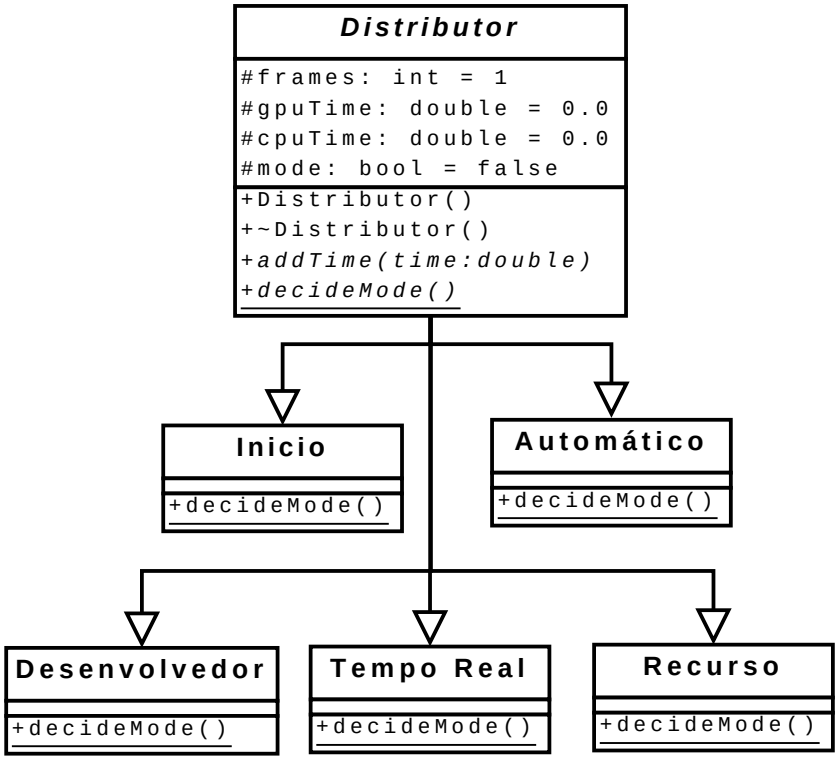

Figura 5.4: Hierarquia de classes derivadas de Distributor.

```
1  Modo_Inicio()
2  {
3          If(frames == 10)
4          {
5                  calculaTempoGastoGPU();
6                  modo = CPU;
7          }
8          If(frames == 20)
9          {
10                 calculaTempoGastoCPU();
11                 If( TempoGPU < TempoCPU)
12                         modo = GPU;
13         }
14
15         Retorne modo;
16 }
```

Código 5.1: Método de distribuição do modo início.

Artificial e lógica fuzzy, é que não é capaz de prever se a aplicação poderá vir a ser retardada por outra thread disparada pelo sistema, tal como um anti vírus ou agentes de atualização automática do Sistema Operacional. Além disso, a cena simulada pode variar de acordo do tempo. Para solucionar isto serão apresentados os módulos descritos

a seguir.

5.4 Modo Tempo Real

Nesta solução a decisão também fica a cargo do GDE. Este modo utiliza-se da seguinte estratégia: calcula 5 frames na GPU e em seguida 5 frames na CPU, a partir de então calcula 90 frames no processador que teve o melhor desempenho. Depois de calculados os 90 frames, repete-se a estratégia, como pode ser visto no código 5.2.

```
1  Distribuição_Tempo_Real()
2  {
3          If (frames == 5)
4          {
5                  calculaTempoGPU();
6                  modo = CPU;
7          }
8
9          If (frames == 10)
10         {
11                 calculaTempoCPU();
12
13                 If (tempoGPU < tempoCPU)
14                 modo = GPU;
15         }
16
17         If (frames == 100)
18         {
19                 frames = 0;
20                 modo = GPU;
21         }
22
23         Retorna modo;
24 }
```

Código 5.2: Método de distribuição do modo tempo real.

A partir deste pseudocódigo pode-se ver que com o uso deste modo, somente 5% dos frames serão calculados usando o processador mais lento e 95% usando o processador mais rápido.

5.5 Modo Recurso

Este modo também deixa a decisão a cargo da GDE. A idéia desta estratégia consiste em utilizar bibliotecas para verificar quanto de recurso está sendo usado da GPU e quanto da CPU. Para a GPU utilizou-se o nVperfKit [54] e para a CPU foi utilizada a API do Windows.

Para justificar o uso de verificação de recurso foi realizado um teste inicial, utilizando-se somente o processo de checagem de *sphere bounding*. Os resultados podem ser vistos na tabela 5.1.

Tabela 5.1: Resultados do % de uso da CPU e da GPU pelo método de sphere bound.

Corpos	CPU FPS	GPU FPS
64	0	82,2
128	1	85,0
256	5	86,8
512	25	90,0
1024	59	90,2
2048	85	90,3

Através dos resultados apresentados destaca-se que o nvPerfKit não é muito indicado para ser utilizado, pois mesmo com diferenças grandes do número de corpos a porcentagem de uso da GPU dado pelo nvPerfKit é muito pequena (0,1%), além disso em outros testes vê-se que a variação entre os frames processados é grande. Já no caso da CPU, a API do Windows mostrou-se bastante adequada.

Analisando a carga de processamento da CPU, desenvolveu-se a seguinte estratégia: realiza-se o calculo de 10 frames na GPU e em seguida 10 frames na CPU. Com os resultados, escolhe-se o processador mais rápido. Em caso do processador escolhido ser a CPU, ativa-se uma variável "usoCPU", e após 10 frames passa-se a verificar a carga de uso da CPU, sempre que sua carga subir a mais de 80%, envia-se 10 frames para serem calculados na GPU, para desafogar a CPU. Pode-se melhor visualiza-lo no código 5.3.

Este modo funciona melhor para simulações onde a CPU é mais rápida que a GPU. Nesta situação, quando a CPU estiver sobrecarregada, pode realizar uma distribuição de parte do trabalho para a GPU, desafogando assim a CPU. Quando a carga de trabalho da CPU voltar ao normal, uma nova redistribuição pode ser feita.

Nos casos onde a GPU for mais rápida que a CPU, este modo irá funcionar como o modo "início" sem realizar nenhuma redistribuição.

```
1  Modo_Recurso ()
2  {
3          If(frames == 10)
4          {
5                  calculaTempoGPU();
6                  modo = CPU;
7          }
8          If(frames == 20)
9          {
10                 calculaTempoCPU();
11                 If( TempoGPU < TempoCPU)
12                         modo = GPU;
13                 Else
14                         usoCPU = verdadeiro;
15         }
16         If(usoCPU && frames > 30)
17         {
18                 Porcentagem = percCPU();
19
20                 If( Porcentagem > 80)
21                 {
22                         modo = GPU;
23                         frames = 21;
24                 }
25         }
26
27         Retorne modo;
28 }
```

Código 5.3: Método de distribuição do modo recurso.

5.6 Modo Automático

Neste modo a decisão fica a cargo da GDE. A estratégia é simples: calculam-se 10 frames na GPU e 10 na CPU, guardando o melhor tempo de cada processador. Ao término do processamento destes 20 frames decide-se pelo processador com o tempo mais rápido, e calcula-se nele os próximos 10 frames, se o tempo deste processador a partir destes 10 frames for menor que o melhor tempo do outro processador, calcula-se 10 frames no outro processador e assim por diante, conforme o código 5.4.

Este modo se utiliza do processador mais rápido na maioria dos casos, utilizando-se do processador mais lento, como um processador auxiliar para aliviar o mais rápido, sempre que o mais rápido ficar lento, o que deverá acontecer quando estiver sobrecarregado.

```
1  Distribuição_Auto()
2  {
3          If(modo == GPU)
4                  melhorTempoGPU = pegarMenor(TempoGPU,melhorTempoGPU);
5          If(modo == CPU)
6                  melhorTempoCPU = pegarMenor(TempoCPU,melhorTempoCPU);
7          If (frames == 10)
8          {
9          modo = CPU;
10         }
11         If (frames == 20)
12         {
13                 If (melhorTempoGPU < melhorTempoCPU)
14                         modo = GPU;
15         }
16         If (frames > 30)
17         {
18                 If(modo == GPU)
19                         If(TempoGPU > melhorTempoCPU)
20                         {
21                                 modo = CPU;
22                                 frames = 21;
23                         }
24                 If(modo == CPU)
25                         If(TempoCPU > melhorTempoGPU)
26                         {
27                                 modo = GPU;
28                                 Frames = 21;
29                         }
30         }
31         Retorne modo;
32 }
```

Código 5.4: Método de distribuição do modo automático.

5.7 Conclusão

Este capítulo propôs diversas maneiras de se distribuir tarefas entre CPU e GPU. Apresenta-se dois modos onde testes para a decisão são realizados no começo da simulação. Estes modos são importantes nos casos em que não se tem conhecimento de qual o processador mais rápido e se deseja uma seleção de recurso automática. Uma desvantagem destes modos é que eles não prevêem o caso da aplicação ser retardada por outra aplicação do sistema, no caso da CPU ou GPU já estarem sobrecarregados com tarefas. Para contornar este problema, o presente trabalho propõem 3 heurísticas de decisão em tempo real .

Verificou-se, por testes apresentados no capítulo 6, que o modo automático se comporta melhor na maioria dos casos e pode realizar mudanças de processador em tempo real, contornando problemas de algum processador estar sobrecarregado.

Além disto, o uso da GDE, prevê também que o desenvolvedor pode querer programar uma heurística própria para a decisão. Para isto ele pode se utilizar do modo de desenvolvedor, tendo apenas que implementar uma função.

Capítulo 6

Resultados

Neste capítulo serão apresentados os resultados da GDE. Na seção 6.1 serão apresentados os resultados da implementação do motor tanto em CPU como em GPU descritos no capítulo 3. Na seção 6.2 apresentam-se os resultados do motor de física com a arquitetura descrita no capítulo 4. Na seção 6.3 apresentam-se os resultados dos modos de distribuição automáticos, propostos no capitulo 5. Finalmente, na seção 6.4, são apresentadas as conclusões do capítulo.

Todos os testes desta dissertação foram feitos em um PC AMD Athlon64 2800+ com uma placa gráfica nVidia 6200 com barramento AGP 8x. Todos os tempos foram tomados em milisegundos utilizando-se da API do Windows.

6.1 Resultados do Motor de Física GDE

Nesta seção, apresentam-se os resultados obtidos pelo motor de física GDE com métodos implementados em CPU e GPU, conforme apresentado no capítulo 3.

Na tabela 6.1 e no gráfico das figura 6.1 pode-se ver os resultados do método de *broad phase*, discutido na seção 3.4, implementado tanto em GPU como em CPU.

Observa-se, através da tabela 6.1 e da figura 6.1, que quanto maior o número de corpos, melhor o desempenho da GPU. E sendo que a partir de 2048 corpos a GPU ganha em performance da CPU.

Na tabela 6.2 e nos gráficos das figuras 6.2 e 6.3 pode-se ver os resultados dos métodos de transformação da matriz de inércia de coordenadas locais para coordenadas globais juntamente com a aplicação da gravidade nos corpos implementados tanto em CPU como em GPU, ambos descritos na seção 3.5.

Tabela 6.1: Resultados númericos (em milisegundos) do shader de colisão.

Corpos	Tempo de CPU	Tempo de GPU
16	0,01	0,29
64	0,11	0,94
256	1,79	6,09
1024	30,70	33,73
2048	126,80	112,48
4096	535,66	406,33
8192	3607,27	1502,30
16380	16451,95	5816,5

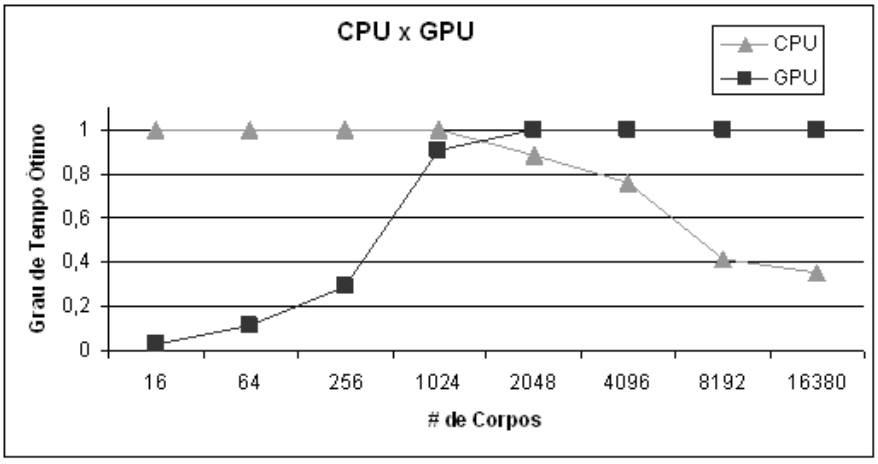

Figura 6.1: Evolução da CPU e GPU no método de sphere bound comparadas através do grau de tempo ótimo.

Tabela 6.2: Resultados númericos (em milisegundos) com o shader de transformação de inércia e aplicação da força da gravidade.

Corpos	Tempo de CPU	Tempo de GPU		
		Processo	Transferência	Total
16	0,03	0,07	0,28	0,35
64	0,07	0,07	0,32	0,39
256	0,25	0,08	0,41	0,49
1024	1,01	0,19	0,83	1,02
2048	2,13	0,34	1,53	1,87
4096	4,050	0,51	2,35	2,86
8192	8,29	1,00	5,52	6,52
16384	16,23	1,60	8,99	10,49

Estes resultados númericos da tabela 6.2 mostram que o processamento da GPU começa a ficar mais rápido que o da CPU quando mais de 64 corpos estão presentes na cena. Entretanto, o gargalo está na transferência de texturas, algo que pode gastar mais de 80% do tempo, como pode ser visto na figura 6.3. De qualquer maneira, para 2048

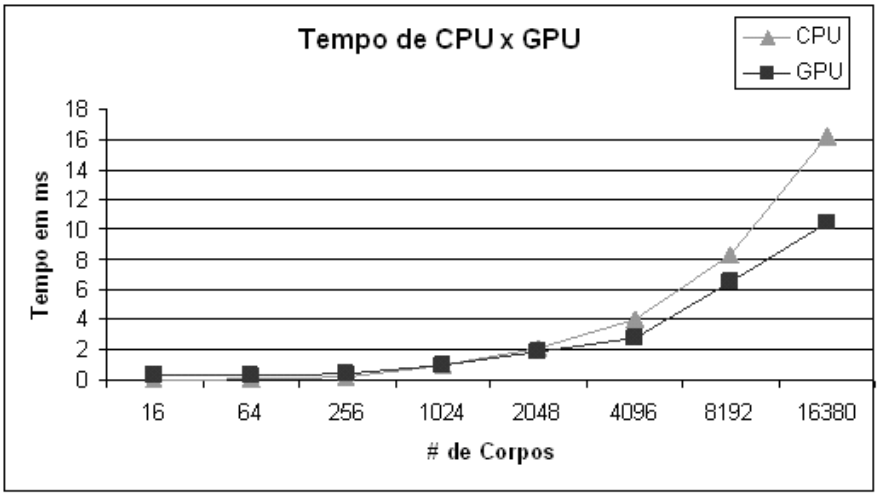

Figura 6.2: Tempo de CPU x tempo de GPU para o shader de inércia.

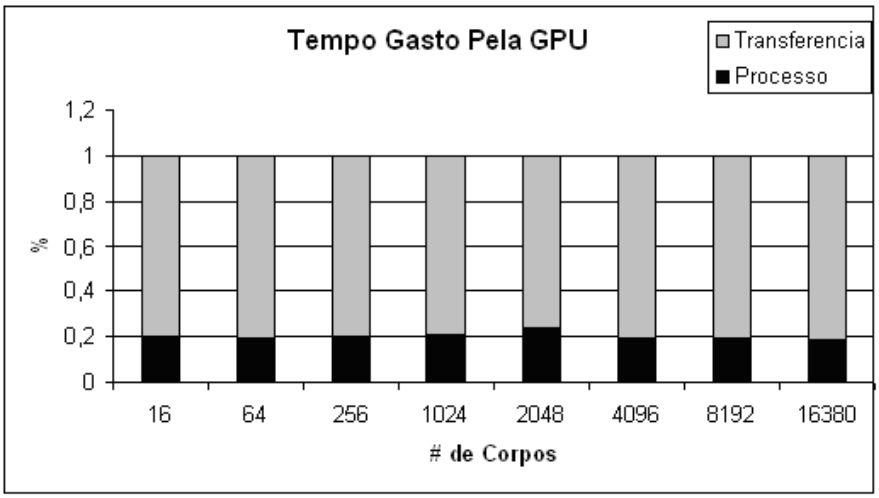

Figura 6.3: Tempo gasto pela GPU discriminado para o shader de inércia.

corpos a GPU tem um tempo mais eficiente que a CPU, mesmo considerando o tempo de transferência, como pode ser melhor visualizado na figura 6.2.

Na tabela 6.3 e nos gráficos das figuras 6.4 e 6.5 pode-se ver os resultados obtidos pelo método de passo do GDE descrito na seção 3.6.

Pode-se ver pela gráfico da figura 6.4 que esse shader se apresenta mais rápido que a implementação em CPU a partir de 2048 corpos. No gráfico da figura 6.5 vê-se que também o tráfego da GPU esta diminuindo o desempenho geral da GPU.

Na tabela 6.4 podem ser vistos os resultados da otimização para os métodos de GPU do loop do GDE sem o PCL e na tabela 6.5 o loop com o PCL ambos descritos na seção 3.7.1.

Tabela 6.3: Resultados númericos (em milisegundos) do shader de passo.

Corpos	Tempo de CPU	Tempo de GPU		
		Processo	Transferência	Total
16	0,05	0,08	0,32	0,40
64	0,09	0,08	0,33	0,41
256	0,34	0,13	0,51	0,64
1024	1,18	0,25	0,96	1,21
2048	2,40	0,58	1,89	2,47
4096	4,61	0,83	3,45	4,28
8192	12,10	1,78	7,25	9,03
16384	18,21	2,83	12,27	15,10

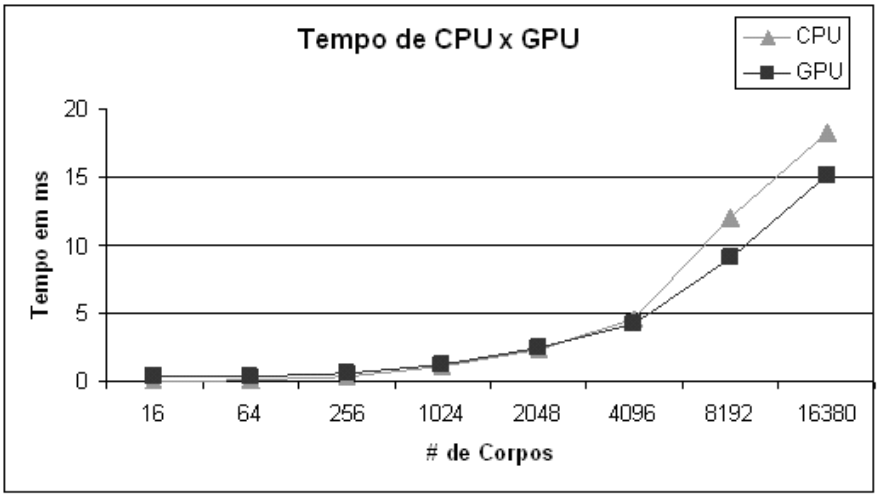

Figura 6.4: Tempo de CPU x tempo de GPU para o shader de passo.

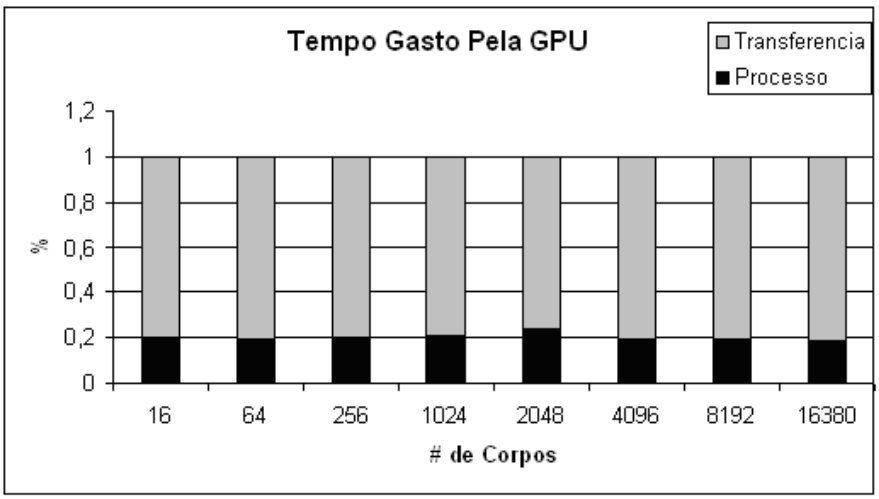

Figura 6.5: Tempo gasto pela GPU discriminado para o shader de passo.

Estes testes preliminares podem ver uma considerável diminuição de transferências da GPU, fazendo com que ela se comporte de forma muito melhor.

Tabela 6.4: Resultados númericos (em milisegundos) dos 3 shaders juntos sem o PCL.

Corpos	Tempo de CPU	Tempo de GPU	Ganho
16	0,04	0,63	0,06
64	0,24	1,51	0,16
256	2,33	5,15	0,45
1024	34,90	34,92	1,00
2048	131,55	115,04	1,14
4096	567,57	407,97	1,39
8192	4235,43	1542,32	2,75
16384	16855,02	5760,42	2,93

Tabela 6.5: Resultados númericos (em milisegundos) dos 3 shaders juntos com o PCL.

Corpos	Tempo de CPU	Tempo de GPU	Ganho
16	0,04	0,75	0,05
64	0,24	1,71	0,14
256	2,33	5,43	0,43
1024	34,90	36,45	0,96
2048	131,55	118,98	1,11
4096	567,57	418,25	1,36
8192	4235,43	1564,02	2,71
16384	16855,02	5909,24	2,85

A tabela 6.6 mostra os resultados da otimização para os métodos em GPU descrita em 3.8 de reutilização de dados em dois casos extremos, reenviando todos os dados e sem o reenvio dos dados.

Tabela 6.6: Resultados numéricos (em milisegundos) do reenvio de dados.

Corpos	Tempo de CPU	Tempo de GPU com reenvio	Tempo de GPU sem reenvio
16	0.04	0.75	0.62
64	0.24	1.71	1.52
256	2.33	5.43	4.92
1024	34.90	36.45	35.82
2048	131.55	118.98	110.76
4096	567.57	418.25	401.06
8192	4235.43	1542.32	1493.10
16384	16855.02	5909.24	5833.76

Pode-se ver os resultados do ganho da GPU com todas as otimizações comparado com a CPU pelo gráfico da figura 6.6. Deste resultados pode-se verificar que a partir de 1024 corpos a GPU se comporta mais rápida.

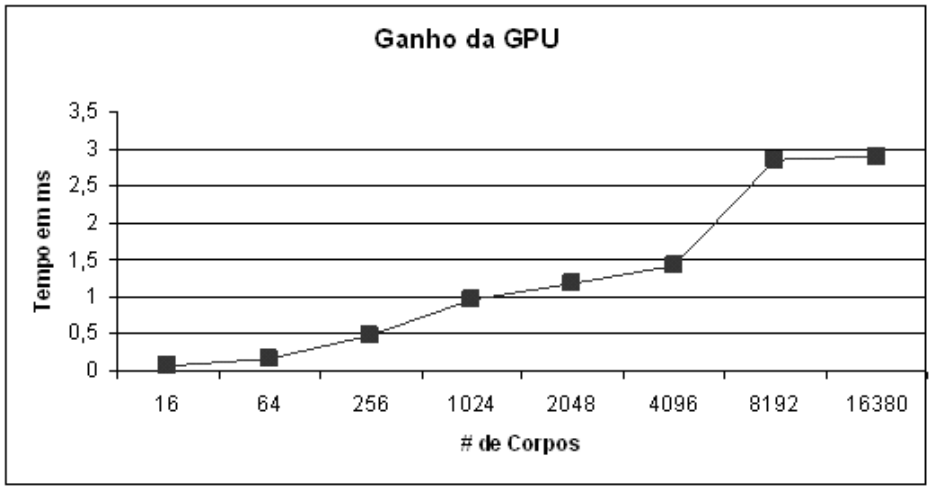

Figura 6.6: Ganho da CPU em relação a CPU.

6.2 Resultados com a Arquitetura

Nesta seção apresentam-se os resultados do motor de física desenvolvido GDE juntamente com o *framework* GUFF e a arquitetura (escolhida no capítulo 4) com motor de física desacoplado com freqüência determinada em 20 FPS e com a renderização atualizada sempre que possível.

Na tabela 6.7 temos os resultados da arquitetura com testes se utilizando da GPU e da CPU.

Tabela 6.7: Resultados da arquitetura em FPS.

Corpos	CPU FPS	GPU FPS
16	880	835
64	819	807
256	710	702
1024	428	432
2048	48	60
4096	38	52

Como resultado visual pode-se ver um frame da aplicação na figura 6.7.

6.3 Resultados da Distribuição entre CPU e GPU

Nesta seção se apresentam os resultados do capítulo 5 desta dissertação, mais especificamente os resultados com os modos de distribuição entre CPU e GPU.

Na tabela 6.8 pode-se ver os resultados, medidos em frames por segundo, do modo

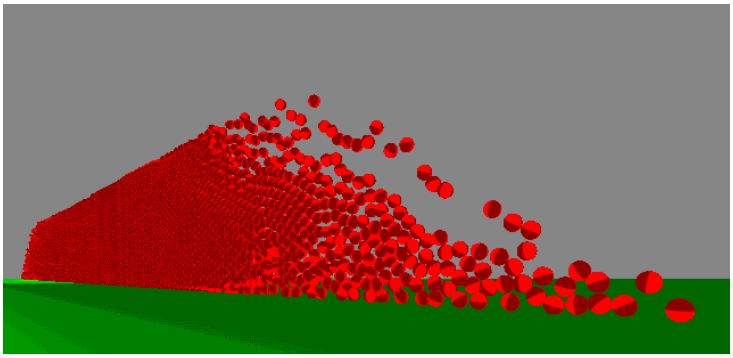

Figura 6.7: Um frame da aplicação.

de decisão automática utilizando-se de lógica fuzzy e da arquitetura feita para ela, ambos descritos na seção 5.2.

Tabela 6.8: Resultados do modo de decisão automático com lógica fuzzy.

Corpos	FPS
16	78
64	72
256	66
1024	62
2048	54
4096	38

Na tabela 6.9 pode-se ver os resultados numéricos do modo inicio descrito na seção 5.3.2 em 100 frames da aplicação após os 20 frames de testes iniciais.

Tabela 6.9: Resultados numéricos do modo de decisão inicio em 100 frames da aplicação.

Corpos	Tempo de CPU	Tempo de GPU	Tempo Modo Inicio
16	4,97	51,15	4,97
64	26,49	125,34	26,49
256	250,18	512,22	250,18
1024	3454,72	3567,00	3454,72
2048	13284,79	11736,40	11736,40
4096	55266,04	40718,73	40718,73
8192	395875,92	155031,21	155031,21

Dos resultados da tabela 6.9 pode-se ver que este modo sempre seleciona o processador mais rápido para processar as tarefas.

Na tabela 6.10 pode-se ver os resultados numéricos do modo tempo real descrito na seção 5.4 em 100 frames da aplicação.

Estes resultados (tabela 6.10) evidenciam que, mesmo com algum tempo gasto no processador mais lento, não ocorrerá um prejuízo no desempenho da aplicação. Para um

Tabela 6.10: Resultados numéricos do modo de decisão tempo real em 100 frames da aplicação.

Corpos	Tempo de CPU	Tempo de GPU	Tempo Modo Tempo Real
16	4,97	51,15	8,56
64	26,49	125,34	34,02
256	250,18	512,22	270,24
1024	3454,72	3567,00	3502,22
2048	13284,79	11736,40	11888,45
4096	55266,04	40718,73	41957,80
8192	395875,92	155031,21	166708,33

número pequeno de corpos, mesmo que o tempo pareça alto em relação ao processamento na CPU, não ocorre o retardamento da aplicação, já que o tempo é pequeno. Para um grande número de corpos, a diferença entre o processamento na GPU, usando o modo tempo real, é pequena. Enfim, entre 1024 e 4096 corpos este modo se comporta melhor, já que a diferença de tempo entre CPU e GPU não é substancial.

Na tabela 6.11 pode-se ver os resultados numéricos do modo recurso descrito na seção 5.5 em 100 frames da aplicação.

Tabela 6.11: Resultados numéricos do modo de decisão recurso em 100 frames da aplicação.

Corpos	Tempo de CPU	Tempo de GPU	Tempo Modo Auto
16	4,97	51,15	5,31
64	26,49	125,34	27,09
256	250,18	512,22	253,30
1024	3454,72	3567,00	3504,02
2048	13284,79	11736,40	11736,40
4096	55266,04	40718,73	40718,73
8192	395875,92	155031,21	155031,21

Dos resultados da tabela 6.11 pode-se ver que quando há 1024 ou menos corpos, onde a CPU é mais rápida que a GPU, este modo se comporta praticamente igual ao melhor modo, mas podendo realizar distribuir tarefas para GPU caso a CPU esteja sobrecarregada. Com mais de 1024 corpos este modo seleciona a GPU como processador e não realiza mais distribuições entre os processadores.

Na tabela 6.12 pode-se ver os resultados numéricos do modo auto descrito na seção 5.6 em 100 frames da aplicação.

Destes resultados (tabela 6.12 pode-se ver que este modo se comporta praticamente como o melhor caso, mas podendo distribuir tarefas entre os processadores caso o utilizado

Tabela 6.12: Resultados numéricos do modo de decisão automático em 100 frames da aplicação.

Corpos	Tempo de CPU	Tempo de GPU	Tempo Modo Auto
16	4,97	51,15	5,11
64	26,49	125,34	26,99
256	250,18	512,22	252,22
1024	3454,72	3567,00	3503,12
2048	13284,79	11736,40	11772,38
4096	55266,04	40718,73	40732,22
8192	395875,92	155031,21	155092,74

esteja sobrecarregado.

No gráfico da figura 6.8 pode-se ver um comparativo dos modos de distribuição, numa escala de 0 á 1 sendo 1 tempo ótimo.

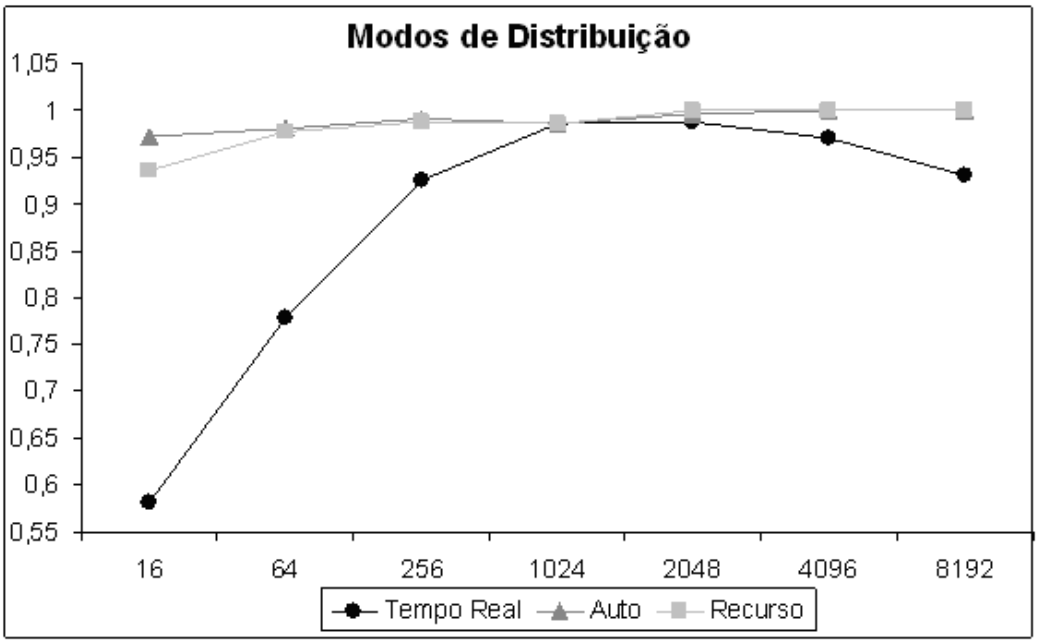

Figura 6.8: Gráfico comparativo dos modos de distribuição em escala de 0 a 1.

6.4 Conclusão

Neste capítulo apresentaram-se os resultados obtidos pelas propostas da dissertação. No próximo capítulo serão apresentadas às conclusões finais e sugestões de trabalhos futuros.

Capítulo 7

Conclusão e Trabalhos Futuros

Os motores de física são componentes fundamentais para aumentar o realismo em simulações e jogos digitais e realizam cálculos pesados e complexos. Devido a esta complexidade, soluções bem sucedidas de motores de física utilizam-se de processamento paralelo. Uma das formas de se utilizar processamento paralelo é com as GPUs programáveis, onde partes desses cálculos podem ser processadas nas GPUs, através de conceitos de GPGPU (*general-purpose computation on GPU*).

A dissertação apresentou um novo motor de física híbrido para jogos digitais e simulações em tempo real, ao qual se chamou de GDE, que pode utilizar tanto a CPU como a GPU para realizar alguns de seus cálculos. Este motor foi baseado no motor ODE, que é gratuito e de código aberto. Pelos testes realizados e apresentados pode-se verificar que este motor tem uma otimização em GPU, comparado com a implementação em CPU, para grande número de corpos rígidos. Estes resultados comprovam que com o uso da GPU, a simulação pode ter um maior número de corpos com um tempo de resposta mais baixo, viabilizando a presença de um grande número de elementos na simulação.

Este trabalho também apresentou um encapsulamento das principais funções deste motor para facilitar o uso com o *framework* GUFF e respeitar os seus requisitos de implementação.

Nesta dissertação, foram apresentadas e discutidas diversas arquiteturas para jogos digitais e simulações físicas escolhendo uma para ser usada em testes com o GDE e o GUFF.

Apresentaram-se também diversas formas de se distribuir carga entre a CPU e a GPU, bem como testes apropriados. Destes diversos modos de distribuição apresentadas, a que se comportou melhor nos testes foi o modo automático de distribuição.

A importância da distribuição automática de tarefas entre CPU e GPU deve-se especialmente às seguintes razões:

- Tirar o melhor proveito de ambos os processadores;
- Evitar que a decisão de escolha do processamento seja feito pelo desenvolvedor;
- Realizar uma distribuição de tarefas para outros processadores quando o sistema verificar, de forma automática, que um dos recursos está sobrecarregado.

É importante ressaltar que apesar de existirem diversos trabalhos na área de GPGPU, não se tem conhecimento de nenhum trabalho de distribuição de carga entre CPU e GPU além desta dissertação e a de Zamith [53].

7.1 Dificuldades

As linguagens de *shaders* para GPU, apesar de serem baseadas na linguagem de programação C, são mais difíceis de serem programadas, pois possuem diversas limitações e um paradigma diferente do tradicional método seqüencial. Outra dificuldade de se programar GPUs é o fato de não haver depurador, sendo às vezes necessário realizar diversos testes para se encontrar erros.

Há muito pouca documentação sobre as técnicas de se desenvolver aplicações em GPGPU, apesar de haverem diversos trabalhos recentemente publicados na área. Uma das maiores dificuldades neste sentido foi a de adaptar a forma seqüencial de se realizar os cálculos de física na forma paralela utilizada pela GPU.

7.2 Trabalhos futuros

Uma proposta de trabalho futuro é desenvolver a *narrow phase* e o PCL do GDE em GPU. Outra proposta é otimizar ainda mais o GDE, utilizando-se da nova tecnologia CUDA.

Outro trabalho que ainda pode ser feito com o GDE é um grau mais elevado de paralelismo do laço. Um exemplo poderia ser enquanto o GPU estiver realizando cálculos da *broad phase* da detecção de colisões, a CPU já estar realizando cálculos da *narrow phase* da detecção de colisão em paralelo. A mesma coisa pode ser feita com a preparação do LCP juntamente com o *shader* de inércia e aplicação da gravidade.

No caso de arquiteturas, propõe-se como trabalho futuro desenvolver uma arquitetura hibrida automática, onde a aplicação possui 2 *threads*: uma para a renderização e outra para o motor de física. A entrada do usuário ficaria na *thread* que possui menor carga de trabalho, sendo essa escolha feita por um modo automático.

Referências

[1] VALENTE, L. *Guff: um framework para desenvolvimento de jogos*. Dissertação (Mestrado) — Universidade Federal Fluminense, 2005.

[2] SMITH, R. *Manual do Open Dynamics Engine*. Disponível em: `http://opende.sourceforge.net/wiki/index.php/Manual`. 20/12/2007.

[3] HOUSTON, M. *Introduction*. 2007. Siggraph07 GPGPU Tutorial. Disponível em: `http://www.gpgpu.org/s2007/`. 20/12/2008.

[4] HAVOK. *Havok Physics*. Disponível em: `http://www.havok.com/`. 20/12/2007.

[5] VALVE. *Software Corporation. Half-life 2*. Disponível em: `http://orange.half-life2.com/`. 20/12/2007.

[6] AGEIA. *PhysX*. Disponível em: `http://www.ageia.com`. 20/12/2007.

[7] CRYPTIC. *Studios. City of Villains*. Disponível em: `http://www.CityofVillains.com/`. 20/12/2007.

[8] ANDERSON, A.; III, W. G.; SCHRODER, P. Quantum monte carlo on graphical processing units. *Computer Physics Communications 177(3)*, p. 298–306, 2007.

[9] RUDOMYN, T.; MILLAN, E.; HERNANDEZ, B. Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory 13(8)*, p. 741–751, 2005.

[10] MULLER, C.; STRENGERT, M.; ERTL, T. Adaptive load balancing for raycasting of non-uniformly bricked volumes. *Parallel Computing 33(6)*, p. 406–419, 2007.

[11] GPGPU.ORG. *Portal de desenvovedores*. Disponível em: `http://www.gpgpu.org`. 20/12/2007.

[12] SMITH, R. *Open Dynamics Engine*. Disponível em: `http://www.ODE.org/`. 20/12/2007.

[13] OLIVEIRA, B. de; MATTOS, E. *Construindo um Framework para Jogos FPS com XNA*. 2007. Disertação (Graduação) - Universicade Federal Fluminense.

[14] SDL. *SDL - Simple DirectMedia Layer*. Disponível em: `http://www.libsdl.org/intro.br/toc.html`. 20/12/2007.

[15] OPENGL. *OpenGL The Industry's Foundation for High Performance Graphics*. Disponível em: `http://www.opengl.org/`. 20/12/2007.

[16] AUDIERE. *Audiere*. Disponível em: `http://audiere.sourceforge.net`. 20/12/2007.

[17] DEVIL. *DevIL - A full featured cross-platform image libary.* Disponível em: http://www.imagelib.org. 20/12/2007.

[18] GLEW. *GLEW - The OpenGL Extension Wrangler Library.* Disponível em: http://glew.sourceforge.net/. 20/12/2007.

[19] LUA. *The programming language Lua.* Disponível em: http://www.lua.org/manual/. 20/12/2007.

[20] QUAKE. *id Software. Quake III Arena.* Disponível em: http://www.idsoftware.com/ga . 30/06/2007.

[21] MICROSOFT. *XBox e Xbox360 consoles.* Disponível em: http://www.xbox.com/pt-BR. 20/12/2007.

[22] NINTENDO. *GameCube e Wii consoles.* Disponível em: http://www.nintendo.com. 20/12/2007.

[23] SONY. *Playstation 1,2 e 3 consoles.* Disponível em: http://www.playstation.com. 20/12/2007.

[24] JEREZ, J. *Newton Game Dynamics.* Disponível em: http://www.physicsengine.com/. 20/12/2007.

[25] LAM, D. *Tokamak Game Physics.* Disponível em: http://www.tokamakphysics.com/. 20/12/2007.

[26] COUMANS, E. *Bullet Physics Library.* Disponível em: http://www.bulletphysics.com/Bullet/. 20/12/2007.

[27] GIMPACT. *Gimpact - Colllision Detection Library.* Disponível em: http://gimpact.sourceforge.net/. 20/12/2007.

[28] BLENDER. *Blender.* Disponível em: http://www.blender.org/. 20/12/2007.

[29] COLLADA. *COLLAborative Design Activity.* Disponível em: http://www.collada.org. 20/12/2007.

[30] TERMINAL. *Reality. BloodRayne 2.* Disponível em: http://www.Bloodrayne2.com/. 20/12/2007.

[31] TEAM6. *Taxi3: eXtreme Rush.* Disponível em: http://www.team6-games.com/. 20/12/2007.

[32] SCS. *Software. 18 Wheels of Steel: Pedal to the Metal.* Disponível em: http://www.scssoft.com/pttm.php. 20/12/2007.

[33] OGRE3D. *Ogre3d.* Disponível em: http://www.ogre3d.org/. 20/12/2007.

[34] 3D, C. S. *Crystal Space 3D.* Disponível em: http://www.crystalspace3d.org/. 20/12/2007.

[35] OPCODE. *Optimized Collision Detection.* Disponível em: http://www.codercorner.com/Opcode.htm. 20/12/2007.

[36] OPENTISSUE. *OpenTissue*. Disponível em: `http://www.opentissue.org/`. 20/12/2007.

[37] NVIDIA. *CUDA - Compute Unified Device Architecture*. Disponível em: `http://developer.nvidia.com/object/cuda.html`. 20/12/2007.

[38] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture documentation version 1.1*. Disponível em: `http://developer.nvidia.com/object/cuda.html`. 20/12/2007.

[39] OWENS, J. D. et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, v. 26(1), p. 80–113, 2007.

[40] HARRIS, M. Mapping computational concepts to gpus. *M. Pharr(Ed.), GPU Gems (2), Addison-Wesley, Boston, USA*, p. 493–508, 2005.

[41] NVIDIA. *GeForce 8800 GPU architecture overview. TB-02787-001_ v0.9*. 2006. Technical report.

[42] GREEN, S. *GPGPU Physics*. 2007. Siggraph07 GPGPU Tutorial. Disponível em: `http://www.gpgpu.org/s2007/`. 20/12/2008.

[43] KIPFER, P.; SEGAL, M.; WESTERMANN, R. Uberflow: a gpu-based particle engine. *Graphics Hardware 2004*, p. 115–122, 2004.

[44] GEORGII, J.; ECHTLER, F.; WESTERMANN, R. Interactive simulation of deformable bodies on gpu. *Proceedings of Simulation and Visualization 2005*, p. 247–258, 2005.

[45] NGUYEN, H. *GPU Gems 3 - Programming Techniques for High-performance Graphics and General-Purpose Computation*. Boston, EUA: Addison-Wesley, 2007.

[46] SALGADO, A. V. *Simulação visual em tempo real de ondas oceânicas utilizando a GPU*. Dissertação (Mestrado) — Universidade Federal Fluminense, 2005.

[47] GOVINDARAJU, K. N. et al. CULLIDE: interactive collision detection between complex models in large environments using graphics hardware. *Graphics Hardware 2003*, p. 25–32, 2003.

[48] FEIJO, B.; PAGLIOSA, P. A.; CLUA, E. W. G. Visualização, simulação e games. *Breitman, K. and Anido, R. (Ed.), Atualizações em Informática, Editora PUC-Rio, Rio de Janeiro, Brasil*, 2006.

[49] BARAFF, D. *Physically based modeling: rigid body simulation*. 2001. Disponível em: `http://www.pixar.com/companyinfo/research/pbm2001/notes.pdf`. 20/12/2007.

[50] BARAFF, D. Fast contact force computation for nonpenetrating rigid bodies. *SIGGRAPH 1994*, 1994.

[51] ROLLINGS, A.; MORRIS, D. *Game Architecture and Design: A New Edition*. [S.l.]: New Riders Publishing, 2003.

[52] VALENTE, L.; CONCI, A.; FEIJO, B. Real time game loop models for single-player computer games. *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, p. 89–99, 2005.

[53] ZAMITH, M. *Uma Arquitetura de Distribuição Dinâmica de Tarefas entre CPU e GPU em Jogos Digitais*. Dissertação (Mestrado) — Universidade Federal Fluminense, 2007.

[54] KIEL, J.; DIETRICH, S. *GPU Performance Tuning with NVIDIA Performance Tools*. 2006. Game Developers Conference.