UNIVERSIDADE FEDERAL FLUMINENSE

MARCELO PANARO DE MORAES ZAMITH

Uma Arquitetura de Distribuição Dinâmica de Tarefas Entre CPU e GPU em Jogos Digitais

NITERÓI

2007

UNIVERSIDADE FEDERAL FLUMINENSE

MARCELO PANARO DE MORAES ZAMITH

Uma Arquitetura de Distribuição Dinâmica de Tarefas Entre CPU e GPU em Jogos Digitais

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 Distribuição Dinâmica de Tarefas Entre CPU e GPU em Jogos Digitais

Marcelo Panaro de Moraes Zamith

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. Aura Conci / IC-UFF - Universidade Federal Fluminense

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

Niterói, 31 de julho de 2007.

Dedico este trabalho aos meus pais que tanto me apoiaram nas minhas escolhas acadêmicas.

Agradecimentos

Ao meu orientador neste projeto, professor Esteban Walter Gonzalez Clua, pelo importante apoio nos momentos difíceis e por acreditar na realização desta idéia.

À professora Aura Conci pelo apoio e por tornar esta pesquisa possível, acreditando no meu trabalho.

Aos professores Paulo e Anselmo e ao amigo Luis Valente, pelo apoio e pelas contribuições que em muito ajudaram na produção desse texto.

À minha namorada, Isis, pois sem seu apoio, amor e paciência não teria chegado aonde cheguei.

Ao amigo Marcelo Cardoso que contribuiu para essa jornada.

Ao amigo Stênio Sã pela amizade, ajuda e apoio em todos os momentos difíceis.

À Angela e Maria da secretaria da pós, pela amizade e pelo trabalho.

Aos amigos Luciana, Renatha, Augusto, Edigar, Diego, Warley e todos os outros do instituto de computação, pela amizade e apoio.

Resumo

A tecnologia empregada nos processadores gráficos, impulsionada pela industria de entretenimento, veio permitir a utilização deste recurso (*Graphics Processing Unit* - GPU) para fins diferentes dos quais foi projetado (visualização). Desenvolve-se um novo campo de pesquisa na área de computação gráfica. *General Purpose on* GPU (GPGPU) vem tomando força devido à capacidade cada vez maior das GPU no processamento de uso geral.

Os jogos digitais têm evoluído constantemente, requerendo cada vez mais poder do *hardware* em geral, executando tarefas e estágios mais complexos, adotando módulos de inteligência artificial e física para reger o comportamento dos objetos em cena, construindo ambientes virtuais bem semelhantes ao mundo real. Simulações físicas têm a característica de serem tarefas que necessitam de constante processamento matemático, uma vez que as equações que regem o comportamento dos corpos são aplicadas, surgindo a questão de como usar a GPU para resolver tais equações.

Este dissertação, embora dê continuidade ao campo de pesquisa da UFF voltado ao desenvolvimento de jogos digitais e ao uso da GPU em simulações, inicia uma nova linha de pesquisa na área de computação visual e interfaces: o uso de técnicas de GPGPU, isto é, aplicação da GPU para solução de problemas gerais, tendo aplicação nas demais linha de pesquisa desta universidade. O trabalho tem por objetivo apresentar um novo modelo de arquitetura de jogo digital que contempla alocação dinâmica de tarefas entre os processadores (CPU e GPU). Em linhas gerais, é um modelo de arquitetura de jogo digital com capacidade de determinar qual processador deve processar uma certa tarefa, segundo um conjunto de regras e a capacidade do próprio *hardware*. Para suportar este modelo de arquitetura, é necessário que o *hardware* gráfico tenha capacidade de programação.

Abstract

Graphic Unit Processors (GPU) are evolving constantly. Recently, they are converging to technologiy that allow other tasks, different from typically computer graphic one, to be executed. This has created a new computer graphic research field: GPGPU (General purpose Computation in GPU) due to its increasing capacity in general purpose processing.

Digital games are requiring much than available hardware, since more complex tasks and stages are being executed, adopting more extensively AI and physics modules to define the behavior of objects in scene. Physics simulations have characteristics that need a high mathematic computation power, whereas the equations that define the body behavior are widely applied, arising the question of how to use GPU to solve physics problems

This work, in spite of ensuring continuity to UFF research field on the digital games development, starts a new research line in visual computation and interface field: the use of GPU to solve general problems, allowing their usage for applicatios of other research lines. This work aims to show a new architecture pattern of digital games with dynamic task allocation that consists in allocating a task in one of the processors (CPU or GPU), making available a set of toolkit allowing an casily implementation of any kind of application. This work also permits that programmers build applicatios using GPUs but without experience at shader programming.

Palavras-chave

1. General Poupose GPUs
2. Computar Graphics
3. Real Time Rendering
4. 3D Games
5. 3D Engines

Glossário

3DS	:	Estrutura de arquivo contendo informações sobre a malha de um modelo 3D
CG	:	C for Graphic
CPU	:	Unidade de processamento central
CUDA	:	Computer Unified Device Architecture
DevIL	:	Developer's Image Library
FBO	:	Frame Buffer Object
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
HDR	:	High Dynamic Range Image
HLSL	:	High Language Shader Language
ModIA	:	Módulo de Inteligência Artificial
pBuffer	:	Pixel Buffer
SDL	:	Simple DirectMedia Layer
Shader	:	Expressão adototada para definir um programa que funciona na GPU
SIMD	:	Single Instruction Multiple Data
Texel	:	Elemento de textura
TM	:	TaskManager
TSC	:	Task Script Configuration

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A indústria de entretenimento tem, cada vez mais, produzido jogos digitais sofisticados com cenas de realismo e personagens que usam técnicas de inteligência artificial sofisticadas e aplicações de física entre outras técnicas. Esta sofisticação dos jogos tem como objetivo passar para o usuário realismo, envolvendo-o no ambiente virtual inclusive sobre os aspectos emocionais. O jogo digital conhecido como *DOOM 3*[1] é um exemplo de como a indústria do entretenimento tem tratado o aspecto do realismo e as suas conseqüências sobre o usuário. Outro exemplo é a console *WII*[2], que tem o objetivo de inserir o usuário no ambiente virtual através do dispositivo de interação desenvolvido.

Entre os dispositivos que vem sofrendo avanços tecnológicos, as placas gráficas vem se destacando. Impulsionados pela indústria do entretenimento, os fabricantes têm tornado as placas gráficas mais presentes nos computadores pessoais, transformando estes em verdadeiras estações gráficas. A tecnologia aplicada nestes dispositivos permite, também, seu uso cada vez maior no processamento genérico.

General-purpose computation on GPUs (GPGPU) são técnicas que usam as *graphics processing unit* (GPU) para processamento genérico. Estas deram origem a uma linha de pesquisa dentro da área da computação gráfica e conduziram a um novo paradigma, onde a CPU (*central processing unit*) não necessita processar todos os dados, estes podem ser divididos para serem processados, também, pelas GPUs. Entretanto, as GPUs não permitem o processamento de qualquer tipo de algoritmo, executando algoritmos desenvolvidos para tratar os dados na forma de fluxo. Outra característica destes processadores é que são voltados para cálculos matemáticos, assim, problemas que envolvem cálculos matemáticos e que podem ser modelados na forma de fluxo são problemas que podem ser tratados pela GPU. Exemplo de problemas são as simulações físicas que podem ser modelados na forma de fluxo e necessitam de cálculos matemáticos, portanto, ideais para

serem processadas na GPU.

Essa dissertação tem como objetivo o desenvolvimento de uma arquitetura de ciclo de jogo utilizando técnicas de GPGPU com distribuição dinâmica de processamento entre CPU e GPU. Alguns trabalhos foram desenvolvidos usando a GPU. Dentre estes, há simulações de ondas oceânicas usando a GPU [3], simulações de nuvens na GPU [4], simulações de partículas [5] e há trabalhos que utilizam a GPU para tratar problemas de equações lineares [6], [7], [8] e [9]. A abordagem desta dissertação apresenta o uso da GPU no processamento genérico em jogos digitais com foco no processamento de física, maximizando seu uso e permitindo que o processador central (CPU) seja alocado para outras tarefas.

A dissertação apresenta um conjunto de funcionalidades desenvolvidas na forma de bibliotecas para serem incorporadas a outras aplicações, permitindo utilizar a GPU não apenas em trabalhos relacionados à computação visual, mas nas demais linhas de pesquisa.

A funcionalidade desta arquitetura de distribuição dinâmica de tarefas genéricas entre CPU e GPU em jogos digitais é apresentar um mecanismo de alocação e distribuição de tarefas, modelando-as de forma a serem tratadas por diferentes processadores e realizando a distribuição segundo um módulo de inteligência artificial, com um nível simples de programação, capaz de aprender sobre as tarefas e a arquitetura de hardware. Com o objetivo de tornar flexível este módulo, um pequeno interpretador de *script* foi desenvolvido na linguagem Lua [10], no qual o desenvolvedor é capaz de configurar esta inteligência artificial.

Durante o processo de desenvolvimento do trabalho, algumas necessidades tiveram que ser supridas, como organizar de forma eficiente as APIs de manipulação de shader e GPGPU. O primeiro conjunto de APIs a serem organizadas foi as de manipulação de shader, sendo desenvolvidas em uma estrutura de classes, onde estão implementados métodos de construção, compilação de detecção de erro e uma estrutura de dados eficiente para armazenagem dos parâmetros dos programas shaders. Para aplicação das técnicas de GPGPU, também foi necessário organizar as respectivas APIs na forma de classe, permitindo seu uso simples e eficiente à gerência e ao processamento *off-screen*. Foi construído um módulo de gerência de tarefas baseado em lógica *fuzzy* com o objetivo de alocar as tarefas e maximizar o uso dos processadores (CPU e GPU). Finalmente, desenvolveu-se recursos básicos e dois ciclos de jogos utilizando a GPU como co-processor matemático, viabilizando a arquitetura proposta.

As principais contribuições desta dissertação são: o uso da linguagem shader e técnicas

de GPGPU na forma de biblioteca como recurso, o uso desta biblioteca no *framework* GUFF, o desenvolvimento de uma arquitetura de jogos digitais com estágio de GPU para fins de processamento genérico e, finalmente, o desenvolvimento de um módulo inteligente com a função de otimizar o uso dos processadores através da distribuição dinâmica de tarefas.

Esta dissertação discute os fundamentos de jogos digitais, linguagem shader, GPGPU, aplicativos para edição de shaders e um *framework* para desenvolvimento de aplicações em tempo real (capítulo 2). Ainda é apresentada no capítulo 2 uma estrutura organizada em classes, na forma de recurso, para encapsular as APIs de manipulação de shader e GPGPU e integrar este recurso em um *framework* para jogos digitais.

O capítulo 3 discute os conceitos e a implementação de um ciclo de jogo baseado em uma arquitetura de distribuição de tarefas com o uso da GPU.

No capítulo 4 e 5, são debatidas evoluções sobre modelo de ciclo de jogo apresentado no capítulo 3, evoluções relacionadas a uma melhor utilização do modelo.

As conclusões deste trabalho são apresentadas no capítulo 6.

Capítulo 2

Conceitos de Arquitetura de Sistemas de Simulação e Visualização de Tempo Real

Neste capítulo, são apresentados conceitos sobre sistemas de simulação e visualização em tempo real. Entretanto, é importante ressaltar que não há um consenso sobre alguns dos conceitos apresentados. Aplicações em tempo real são uma classe especial de sistemas que têm uma forte restrição: o tempo de resposta, ou seja, se esta restrição for quebrada, então o sistema irá falhar [11]. Por exemplo, em aplicações multimídia, a aplicação constantemente recebe um fluxo de audio e vídeo. Se um ou ambos falham, a reprodução fica comprometida.

Jogos digitais são uma especialização de sistemas de visualização em tempo real, uma vez que o tempo de resposta é um requisito fundamental na dinâmica do próprio jogo.

A engenharia de software aplicada à jogos digitais define algumas soluções amplamente usadas e referenciadas, tanto em trabalhos acadêmicos como em ferramentas comerciais; tais como motores de jogos, *frameworks* e *toolkits*.

Neste capítulo, são discutidos conceitos sobre jogos digitais (seções 2.1 a 2.4), conceitos sobre GPU e linguagem shader (seções 2.5 e 2.6), processamento de propósito geral em GPU (seção 2.7) e a apresentação de um *framework* com a incorporação de um módulo de shader como recurso (seções 2.8 a 2.9). Na seção 2.10, é apresentada a conclusão do capítulo.

2.1 Motor de Jogo

Motor de jogo ou *game engine* é entendido como uma coleção de ferramentas e arquivos de dados que permite ao usuário implementar jogos digitais [12]. Para [13], é um núcleo mínimo de código capaz de prover algoritmos, funcionalidades e tarefas re-utilizáveis essenciais ao jogo. Já para [14], um *game engine* é composto por uma coleção de módulos de código de simulação que indiretamente determina o comportamento do jogo digital, isto é, a lógica ou o seu ambiente. Motores de jogos digitais comerciais, como os do *Unreal* [15] ou *Quake* [16], são compostos de diversas ferramentas para desenvolvimento de jogos digitais, tais como os editores de cenários, personagens ou de física.

Um motor de jogo é visto como um conjunto de bibliotecas necessárias à implementação de um jogo, as quais devem conter funcionalidades mínimas para seu propósito, isto é, construir um jogo. Como exemplo, há o tratamento de eventos de dispositivos de entrada e saída. As funcionalidades mínimas devem observar a plataforma para a qual são destinadas, ou seja, diferentes plataformas exigem requisitos mínimos de funcionalidades, por exemplo, PCs e consoles. Para PCs, o suporte mínimo à dispositivos de entrada deve implementar a captura de eventos do *mouse* e teclado, enquanto que,para consoles, deve haver suporte a captura de eventos de *joysticks*.

2.2 *Toolkit*

Toolkit, pela definição de [17], é o conjunto de classes que podem ser re-utilizadas, tais como as bibliotecas (STL [18] e Boost [19]). O objetivo da re-utilização é aplicar as mesmas bibliotecas em contextos de diferentes problemas. Como exemplo, a classe *map*, contida na biblioteca STL, permite a construção de um vetor dinamicamente, sendo que, tanto seu índice como o seu conteúdo podem ser uma classe, estrutura ou um dos tipos primitivos da linguagem. O *toolkit* pode ser visto não apenas como um conjunto de classes, mas como um conjunto de funcionalidades independente do modelo de programação (estruturada ou orientada a objetos), de caráter genérico, permitindo a mesma solução para aplicações com um grau de similaridade, ou seja, aplicações de tipos ou estruturas diferentes, porém, com problemas idênticos. Como exemplo, cita-se o armazenamento de dados em um vetor ou a sua ordenação.

2.3 *Framework*

Segundo [20], *framework* é um conjunto de bibliotecas que auxilia na construção e compilação de diferentes aplicativos em diferentes arquiteturas de sistemas operacionais ou hardware. Para [17], *framework* é composto por um conjunto de classes definidas para uma particular aplicação sobre as quais são criadas subclasses otimizadas para trabalhar com aplicações similares. Como exemplo há o XNA [21].

Em linhas gerais, motores de jogos e *toolkits* são partes de *frameworks*, disponibilizando uma variedade de recursos e funcionalidades na construção de jogos digitais e permitindo ao desenvolvedor focar seu trabalho em questões da própria aplicação, como a lógica do jogo digital.

2.4 Modelos de Ciclo de Jogo Digital

Jogos digitais são compostos por três módulos básicos. O primeiro é a aquisição da informação: o usuário (jogador) demanda de entrada de dados, em geral feita por teclado, *mouse*, *joysticks* ou qualquer outro dispositivo de interação [2], [22] e [23]. O segundo módulo é responsável pela atualização do estado do jogo digital que, em linhas gerais, analisa as entradas do usuário, processa e aplica mudanças no estado do jogo. Dentro deste módulo, também é aplicada a lógica, inteligência artificial, física e outros estágios responsáveis pela alteração de estado. O terceiro e último módulo é responsável pela visualização: consiste em retornar para o usuário o estado atualizado, estimulando alguns dos sentidos do próprio usuário por meio de imagem, som e, em alguns casos, pelo próprio dispositivo de entrada usado, como os *joysticks* com *feedback* [2].

Estágios e tarefas devem ser resolvidos dentro de um tempo determinado a fim de evitar problemas de interatividade. Qualquer alteração no tempo de processamento compromete a interação.

A medida usada para determinar a performance de aplicações de visualização em tempo real, em especial os jogos digitais, são quadros por segundos (FPS *frames per seconds*) gerados pela aplicação. Um quadro é basicamente um imagem digital construída e visualizada na tela. O limite mínimo aceitável de quadros por segundos está em torno de 25 FPS, o que garante uma animação de qualidade [13]. Há diferentes formas que uma aplicação de tempo real pode organizar seus módulos, tais como os discutidos em [13], [24] e [25].

Um jogo digital, em sua essência, é uma aplicação convencional, como um editor de texto. É orientado a eventos, característica de qualquer aplicação destinada a dispositivos micro-processados [26]. Os eventos de um jogo digital podem ser de tempo (o ciclo do jogo) ou de interação (mouse, teclado ou joystick). Portanto, assim como numa aplicação convencional, há três estágios bem definidos: captura de eventos de entrada, processamento e geração de uma saída. A diferença entre aplicações como um editor de texto e um jogo digital é a complexidade dos estágios.

Os ciclos de jogos digitais mais difundidos são: simplesmente acoplado, simplesmente acoplado sincronizado e multithread desacoplado, que serão mostrados e discutidos nos próximos tópicos desta dissertação.

2.4.1 Modelo Simplesmente Acoplado

O modelo simplesmente acoplado, utilizado por arquiteturas de hardware homogêneas, organiza os estágios de uma aplicação de tempo real de forma linear, iniciando pela entrada do usuário, seguido do modo atualização e, por último, executando o estágio de visualização. Este ciclo não aplica qualquer mecanismo de sincronismo [13] e [27], executando, o mais rápido possível, cada um dos estágios, conforme ilustrado na figura 2.1.

É uma arquitetura de ciclo de jogo não recomendada para ser aplicada em hardwares heterogêneos, como computadores em geral, devido à diversificação de hardware, em especial dos processadores, porém, amplamente adotada em consoles, pois estes possuem arquitetura de hardware homogêneo [28].

Processadores com um alto poder de processamento executam as tarefas mais rapidamente, o que implica que o próprio ciclo do jogo é executado com uma freqüência maior. O impacto que isto traz para dinâmica do jogo digital é a execução mais rápida de cada um dos estágios. Por outro lado, quando o ciclo trabalha em um hardware com processador de menor capacidade de processamento, ocorre exatamente o contrário: uma degradação da dinâmica do jogo digital, onde o ciclo de jogo é executado com menor freqüência, comprometendo as animações gráficas. Em ambos os casos, ocorre um efeito indesejável e que vai refletir na usabilidade do próprio jogo digital.

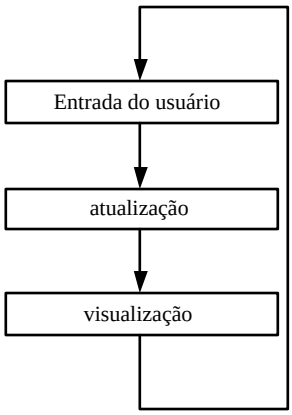

Figura 2.1: Modelo simplesmente acoplado

2.4.2 Modelo Simplesmente Acoplado Sincronizado

O modelo simplesmente acoplado sincronizado apresenta uma estrutura própria para tratar o problema apresentado na seção 2.4.1, que ocorre em arquiteturas heterogêneas de hardware. A solução apresentada é a execução do ciclo de jogo digital em uma freqüência fixa pré-definida [27] [29]. A freqüência fixa é aplicada como um mecanismo de sincronização. A adoção deste método no modelo fez com que um novo estágio fosse introduzido no ciclo de jogo chamado de sincronização, como ilustrado na figura 2.2.

O jogo digital apresenta o mesmo comportamento independentemente do processador, executando os estágios de forma seqüencial. O processamento do estágio de sincronização serve para garantir a uniformidade do ciclo em outras arquiteturas com capacidade computacional heterogênea, usando FPS fixo como parâmetro de sincronização, conforme ilustrado na figura 2.2.

A dinâmica do jogo digital não sofre qualquer tipo de melhora ou alteração em seu comportamento em relação ao processador, como animações mais suaves. Embora resolva parcialmente os problemas introduzidos pelo modelo simplesmente acoplado, não é o modelo ideal.

A solução ideal seria um modelo híbrido, capaz de ser processado por diferentes arquiteturas sem comprometimento da usabilidade, mostrando uma melhor qualidade na visualização quando executado por hardwares de maior poder computacional.

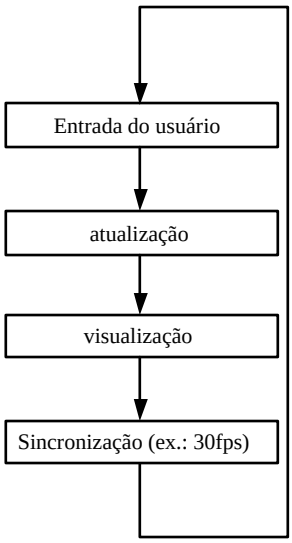

Figura 2.2: Modelo simplesmente acoplado sincronizado.

2.4.3 Modelo Multithread Desacoplado

O modelo multithread desacoplado introduz o conceito de programação concorrente, apresentando uma separação dos estágios em diferentes threads. Uma thread representa o ciclo principal, composto pela entrada do usuário e de atualização do estado jogo digital. A segunda thread executa apenas o estágio de visualização, conforme pode ser observado na figura 2.3.

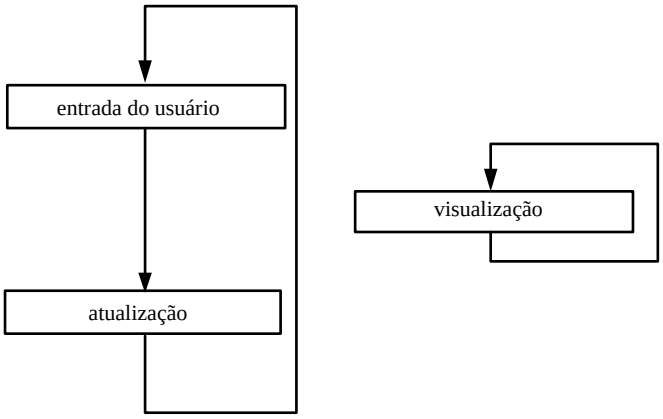

Figura 2.3: Modelo multithread desacoplado.

Embora o modelo apresente uma arquitetura diferenciada em relação às anteriores, o mesmo problema apresentado no modelo simplesmente acoplado ocorre: a interferência na dinâmica do jogo digital em função do poder computacional da arquitetura de hardware utilizada, atingindo diretamente as animações, seja por uma execução lenta e excessivamente suave ou por uma animação menos suave.

Dentro de um jogo digital, as animações, em geral, aplicam um deslocamento sobre um objeto em cena, isto é, o objeto possui uma velocidade, aceleração e posição; entretanto, as medidas utilizadas em um mundo virtual são relativas, não há como medir a velocidade em m/s ou km/h, ou a aceleração em m/s^2 ou ainda a posição em metros ou quilômetros. Em animações, as medidas absolutas são *texels* para deslocamentos no cenário, FPS para aceleração ou velocidade e o tempo é dado em milissegundos. Tomando como exemplo uma bola em queda, tanto no mundo real quando no mundo virtual, a nova posição da bola é dada pela equação 2.1. A diferença é apenas a medida utilizada. Em ambientes virtuais, a medida de tempo em geral tem o valor de $\frac{1}{30}$ milissegundos [29].

$$p_1 = p_0 + v \times t \tag{2.1}$$

A solução para evitar o comportamento indesejável do jogo digital é introduzir o conceito de tempo gasto entre o estágio de atualização e esse mesmo estágio na interação seguinte do ciclo. Esse tempo gasto é o parâmetro de entrada do próprio estágio de atualização, sendo aplicado em todos os cálculos desse estágio que utilizem a variável tempo em alguma equação 2.1. Aplicando esta técnica no modelo multithread desacoplado, introduzindo-se um estágio de sincronismo, conforme figura 2.4, permite-se que computadores com configurações diversificadas possam executar corretamente todas as tarefas contidas no estágio de atualização [28].

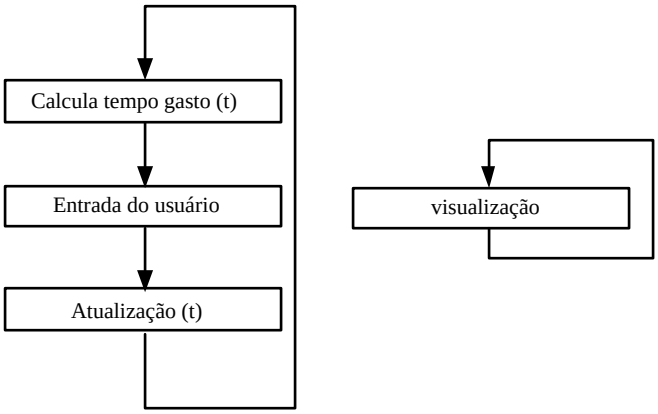

Figura 2.4: Modelo multithread desacoplado com sincronização.

2.5 Conceitos de GPU

Inicialmente o pipeline gráfico implementado no hardware gráfico era fixo, conforme ilustrado na figura 2.5. Embora esta arquitetura tivesse apresentado um avanço tecnológico

na área da computação gráfica na época do seu surgimento, os processadores gráficos não concediam liberdade aos desenvolvedores, principalmente os do mercado de entretenimento digital. Estes impulsionaram os fabricantes desse hardware a produzir versões programáveis do pipeline gráfico, o que tornou o uso da GPU flexível.

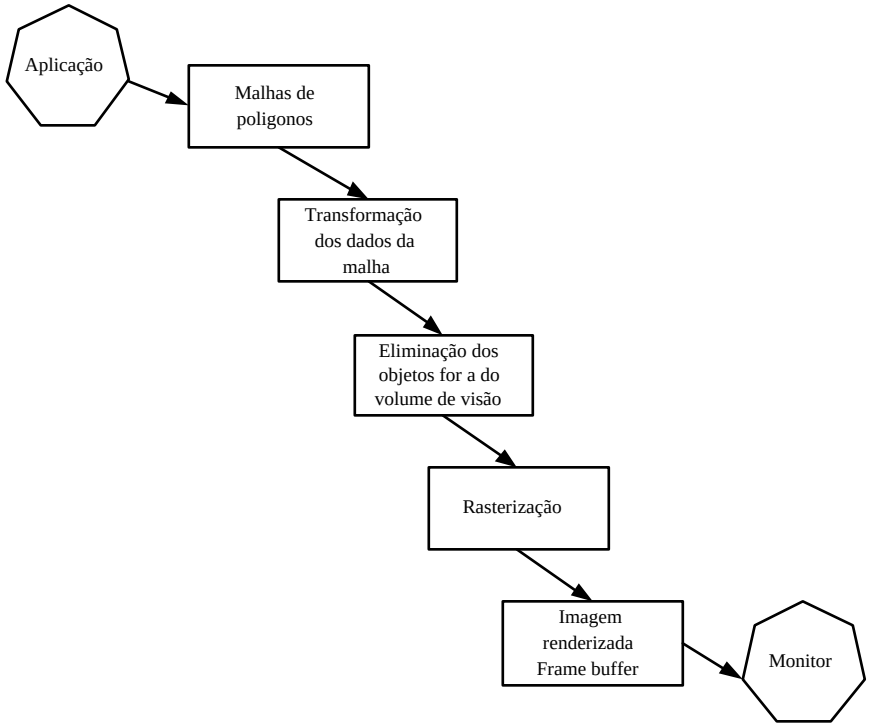

Figura 2.5: Pipeline gráfico fixo.

As modificações nos estágios do pipeline gráfico podem ser observadas na comparação das figuras 2.5 e 2.6. Na figura 2.6, pode-se observar onde se encontram os dois novos estágios introduzidos.

2.5.1 Programação em Shader

Shaders são programas inseridos no pipeline gráfico, executados na GPU. Cada um dos novos estágios exige uma programação específica, isto é, um programa de vértice e um programa de *pixel*. O primeiro, definido na literatura como *vertex* shader, trabalha questões da geometria, mapa de normais e outras operações relativas a transformações aplicadas sobre vértices. O segundo, definido como *pixel* shader, trabalha processamento de *pixel*. Neste estágio do pipeline gráfico programável, a informação é como uma imagem e não como uma malha 3D. No pixel shader, são implementadas as técnicas de processamento gráfico sobre as propriedades de cada texel da imagem, tais como cores, texturas 1D, 2D e 3D, texturas *cubemap* e *shadowmap*, entre outras.

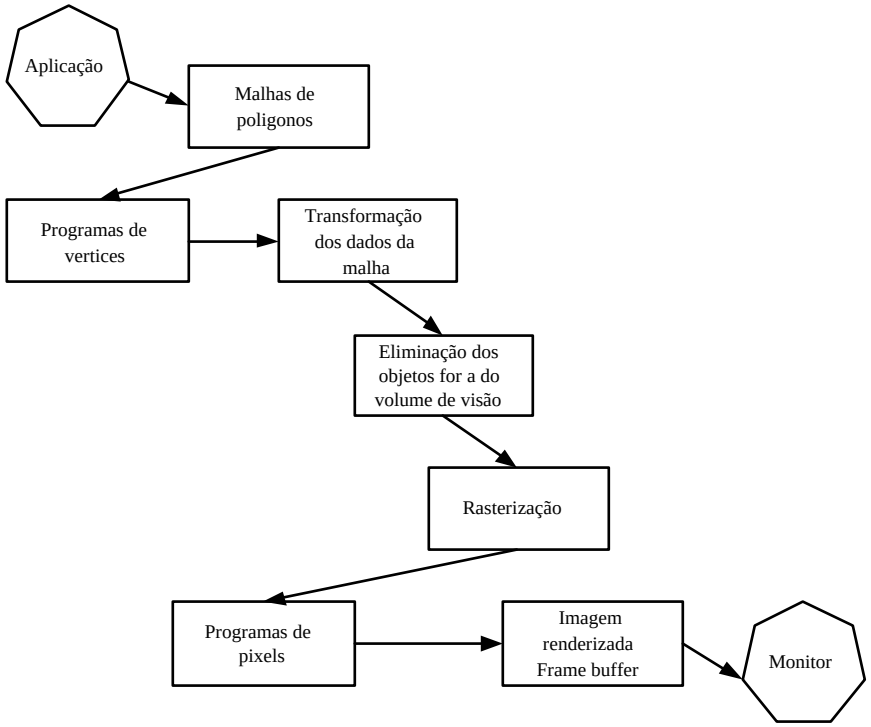

Figura 2.6: Pipeline gráfico programável.

Algumas técnicas exigem a combinação de programação em vértice e em *pixel*, como HDR, iluminação por *pixel*, *bump mapping*, entre outras.

As limitações impostas pelas linguagens shaders de alto nível não estão relacionadas a estas e sim com a versão do shader implementado pela placa gráfica. Como exemplos de limitações, cita-se: a precisão do número de ponto flutuante, quantidade de instruções do programa, tamanho e quantidade de texturas, quantidade de componentes de cada pixel e a velocidade do processamento.

2.5.2 Variáveis, Tipos e Qualificadores

Dentre as linguagens pesquisadas, todas implementam os tipos básicos da linguagem ANSI C e tipos próprios como matrizes, vetores e ponteiros para texturas. Além dos tipos variáveis, há também os qualificadores, que são responsáveis pela forma de armazenamento da variável [30], comuns entre as linguagens pesquisadas. Cinco são os qualificadores:

- *const* define uma variável como constante. Seu valor não pode ser alterado, colocando a variável apenas para leitura;
- *attribute* é utilizado apenas pelo programa de vértice. Variáveis com este qualificador são apenas de leitura dentro do programa de vértice e seu valor é escrito pela

API gráfica, permitindo valores diferentes para cada vértice da malha poligonal;

- *uniform* é um qualificador e também tem seu valor passado pela API gráfica. Tem a característica de ser apenas de leitura dentro do programa para vértice ou para pixel, funcionando como uma constante que é definida dentro a aplicação pela API gráfica;

- *varying* permite que as variáveis tenham características globais, possibilitando tanto a leitura quanto a escrita. Seu principal foco é permitir que valores sejam passados da programação em vértice para a programação em pixel e

- *default* não requer qualquer declaração. Pode ser acessada de qualquer parte do programa ou funções deste e está disponível tanto para leitura quanto para escrita.

As linguagens shader pesquisadas são baseadas na linguagem C/C++ implementando os tipos primitivos e também tipos específicos da linguagem, observando sempre a limitação imposta pelo hardware, como descrito abaixo:

- *void* - Semelhante ao ANSI C, na linguagem shader é usada apenas por funções para definir que não há valor a ser retornado;

- *boolean* - Expressa uma condição, semelhante ao C++, admite apenas dois valores (verdadeiro e falso). Os valores *true* e *false* são utilizados em estruturas condicionais (*if*, *while*, *for*, *do while*);

- *integer* - Limitado em 17bit's de precisão, incluindo um *bit* extra de sinal. Tanto para programação em vértice quanto para programação em *pixel*. Esta limitação ocorre devido ao suporte do hardware a tipos inteiros. Os valores literais aceitos são na base 10, 16 e 8 e são eficientemente aplicados em índices de arrays ou em estruturas de loops (*while*, *for*, *do while*);

- *float* - Amplamente utilizado em cálculos que envolvem operações com escalar, atualmente tem precisão simples IEEE para Shader 3.0;

- vetores - Utiliza o conceito de vetor do cálculo vetorial. Há vetores de 2, 3 e 4 componentes de qualquer dos tipos básicos e são amplamente utilizados em técnicas de processamento gráfico. São inicializados por meio do construtor;

- *matrizes* - As linguagens shader implementam o tipo matriz de ponto flutuante de tamanhos variados: 2x2, 3x3 e 4x4. São inicializadas por meio do construtor;

- *Sampler* - Por intermédio desta variável é que as texturas são acessadas. As texturas podem ser 1D, 2D, 3D, *shadowmap* e *cubemap*. São variáveis apenas de leitura e são inicializadas pela API gráfica. Usada apenas pelos programas de pixel;
- *Struct* - Tipos compostos de variáveis, semelhante ao C++, permite a criação de tipos compostos. Devem ser inicializados por meio de construtor; e
- *Array* - Representa um conjunto de variáveis de algum dos tipos básicos listados acima. A linguagem Shader suporta este tipo com algumas limitações como o tamanho e o suporte a arrays multidimensionais. Arrays são declarados estaticamente.

2.5.3 Linguagens Shaders: CG, HLSL, GLSL

Esta seção tem por objetivo mostrar as características de cada uma das linguagens shaders de maior destaque no mercado, bem como as ferramentas disponíveis para o desenvolvimento e edição. Será também discutido qual a linguagem utilizada nesta dissertação. As linguagens CG, HLSL e GLSL são bem difundidas e baseadas em C/C++ e cada qual com características particulares.

2.5.3.1 Linguagem CG

Foi a primeira a ser disponibilizada no mercado. Apesar de ser fortemente baseada na linguagem C, incorpora alguns recursos da linguagem C++, como o conceito de construtor. Está disponível para vários sistemas operacionais, tais como MAC OS, Linux e Windows. No sistema operacional Windows, foi desenvolvida para trabalhar com as duas APIs gráficas mais conhecidas do mercado (OpenGL e DirectX) [31]. Adota o conceito de *profile*, otimizando o código tanto para placas gráficas da NVIDIA quanto para as API gráficas. *Profile* é definido como um subconjunto de instruções disponíveis destinadas a uma API ou placa gráfica [32].

2.5.3.2 Linguagem GLSL - *OpenGL Shader Language*

É uma linguagem shader desenvolvida pela 3Dlabs [33] e incorporada a partir da versão 1.5 do OpenGL, integrada na própria API. O conceito de construtor também é adotado por esta linguagem, não só para inicializar novas variáveis ou arrays como para executar conversões de tipo.

A principal característica é o acesso ao uso de variáveis de ambiente do próprio OpenGL. Variáveis contendo informações definidas na aplicação, como iluminação, características do material, matriz de visualização e outras, podem ser acessadas diretamente pelo programa shader.

2.5.3.3 Linguagem HLSL - *High Level Shader Language*

Desenvolvida inicialmente pela Microsoft e NVIDIA, foi incorporada no SDK do DirectX a partir da versão 8, atualmente sendo mantida apenas pela Microsoft. Assim como na linguagem GLSL, traz suporte a variáveis de ambiente da API gráfica (Direct X). Esta linguagem esta integrada com o DirectX e, portanto, permite buscar informações de variáveis dentro do ambiente DirectX, diferentemente da linguagem CG, onde é necessário passar os valores destas variáveis como parâmetro de alguma função [34].

2.5.4 Utilizando GLSL como Linguagem Shader

As três linguagens apresentadas são bem conhecidas, cada qual com recursos e limitações semelhantes, entretanto, a escolha por uma destas linguagem deve considerar questões como facilidade de portabilidade e manutenção e licença da API gráfica utilizada.

Para a presente pesquisa, foi adotada a linguagem OpenGL Shader Language (GLSL) como linguagem shader padrão. O principal fundamento na escolha desta linguagem é a sua integração com o *framework* utilizado na pesquisa. O *framework* GUFF [35], melhor discutido na seção 2.8, utiliza o OpenGL como API gráfica. Outras características da linguagem GLSL tiveram importância na decisão, tais como:

- compatibilidade com diversos sistemas operacionais tais como Linux, Mac OS e Windows;
- os programas shaders podem ser usados em diferentes placas gráficas que oferecem suporte a esta linguagem; e
- o código é compilado pelo *driver* da placa gráfica, garantindo a criação de código otimizado.

É importante ressaltar que uso da GPU no processamento genérico não depende da linguagem shader, mas sim do suporte que a API gráfica dá ao recurso de processamento de propósito geral.

2.6 Plataformas de Desenvolvimento de Shaders: *RenderMonkey*

Objetivando uma melhor produtividade no desenvolvimento de shaders, algumas ferramentas foram produzidas, tais como RenderMonkey [36] ou FXComposer [37]. Cada ferramenta trabalha com diferentes linguagens shaders. O RenderMonkey traz suporte para edição de shaders em GLSL e HLSL. O FXComposer traz suporte à linguagem CG. Levando em conta a linguagem shader adotada, a ferramenta de desenvolvimento definida para o presente trabalho foi o RenderMonkey. A decisão pelo uso desta ferramenta é justificada pelo fato de ter suporte do desenvolvimento da linguagem GLSL, de trabalhar com modelos 3ds[1] e de ser uma ferramenta livre.

2.7 Processamento de Propóstito Geral em GPU

Processamento de uso geral em GPU, também definido como GPGPU (*general-purpose computation on* GPUs), significa utilizar o processador gráfico para tratar problemas genéricos. Este recurso é proporcionado devido ao crescente avanço da capacidade das GPUs, que vêm permitindo o desenvolvimento de pesquisas na utilização dessa arquitetura para processar dados em paralelo e na visualização de simulações científicas [38].

Tecnologias como *pixel buffer* (pBuffer) e *frame buffer object* (FBO) vêm permitindo o uso desta arquitetura específica no processamento de uso geral. Problemas que antes eram tratados apenas pela CPU, agora podem ser trabalhados pela GPU. O desenvolvimento da série 8 da NVIDIA traz uma nova tecnologia para o processamento de propósito geral pelas GPUs: *Compute Unified Device Architecture* (CUDA) [39], que permite o uso maior da GPU no processamento genérico, trazendo um novo paradigma de programação, onde um processamento pode ser realizado tanto pela CPU quanto pela GPU.

Aplicações matemáticas e simulações físicas são problemas que podem ser modelados para serem processados pela GPU, podendo ser bem aplicadas para resolver problemas de tempo real. Além da complexidade da programação em GPU, uma outra questão fundamental que ainda impede o uso mais freqüente da GPU é o tempo de latência que existe na transferência dos dados para a memória de vídeo, o que pode degradar a performance.

A programação de GPU requer o conhecimento de alguns conceitos próprios que

[1]Estrutura de arquivo suportado pelo *framework* GUFF.

possuem equivalência no modelo de programação convencional, isto é, para a CPU. A seguir são apresentados alguns destes conceitos.

2.7.1 Programação em Fluxo

O modelo de programação para GPU utiliza um paradigma diferente do modelo tradicional, baseado em fluxo de dados, onde cada elemento do fluxo é processado de forma independente em relação ao elemento vizinho, é arquitetura SIMD (*single instruction, multiple data*), um fluxo de instruções para múltiplos fluxos de dado. Os elementos do fluxo são um conjunto ordenado de um mesmo tipo, que são processados por um *kernel*[2]. O *kernel* é o programa que é executado na GPU, é o fluxo de instruções, portanto, há uma cópia do *kernel* em cada processador de fluxo. Cada cópia processa um elemento do fluxo e escreve o resultado do processamento gerando um fluxo de saída. Este é enviado para um *buffer* de saída.

2.7.2 Texturas, Matrizes e Vetores

Estruturas como vetores (*arrays*) podem ser multidimensionais, assim como as matrizes. No caso de vetores, na aquitetura de GPUs, as estruturas podem ser representadas através de texturas, isto é, as informações para serem processadas em GPUs devem ser organizadas em uma destas estruturas de dados (vetores ou matrizes) e transferidas para a memória da placa gráfica como simples texturas, onde serão processadas como tal. Um textura de saída é produzida e colocada no *buffer* de saída da placa gráfica a fim de ser visualizada no monitor. No processamento genérico realizado pela GPU, este *buffer* de saída não é visualizado no monitor, mas sim lido e armazenado em uma estrutura de vetor ou matriz dentro do modelo tradicional de programação [40], a qual pode ser devolvida à CPU ou utilizada em um novo passo de processamento da GPU.

Texturas são a única estrutura de dados acessada por um *kernel* e cada elemento é acessado como um elemento do fluxo. No modelo de programação tradicional, esta estrutura é vista na forma de um vetor ou matriz, portanto, sendo acessada por uma das estruturas de repetição. Logo, o acesso a cada elemento da textura por um *kernel* pode ser equiparado com o acesso a um elemento do vetor no modelo de programação tradicional.

[2]Nome dado a um programa shader.

2.7.3 *Pixel Buffer* x *Frame Buffer Object*

O OpenGL possui duas APIs para trabalhar com visualização *off-screen*: *pixel buffer* (pbuffer) e *frame buffer object* (FBO). O foco principal destas APIs é gerar, dinamicamente, texturas ou aplicação de sombras. Ambas são extensões do OpenGL. A combinação dessas APIs com a programação em shader permite a utilização da GPU para processamento genérico. Embora seja possível utilizar tanto programas de vértice quanto programas de *pixel*, a programação em *pixel* é amplamente adotada no uso geral da GPU, devido ao espaço de memória, uma vez que os dados armazenados na forma de textura ocupam menos espaço que uma representação por vértice, que precisa de uma estrutura grande e complexa para armazenar cada vértice [40].

2.8 *Framework* GUFF

GUFF (*games* UFF) é um *framework*, fruto de um projeto de pesquisa anterior [35]. Foi adotado como framework para desenvolver a presente pesquisa por dois aspectos:

- dar continuidade ao projeto anteriormente iniciado nesta instituição, agregando funcionalidades e
- este *framework* possui as bibliotecas necessárias para o desenvolvimento deste projeto de pesquisa.

O GUFF foi desenvolvido utilizando a linguagem C++ e incorporando algumas bibliotecas necessárias a manipulação de recursos, tais como: texturas, janelas de visualização, modelos 3D, fontes e outros recursos. Estas foram utilizadas com o objetivo de abstrair aspectos específicos dos sistemas operacionais. É composto por um conjunto de classes que objetivam auxiliar o desenvolvimento de aplicações, tais como jogos digitais e simuladores de tempo real. Adota os seguintes paradigmas:

- suporte para funcionar nos sistemas operacionais: Windows e Linux;
- gestão automática de recursos e
- re-utilização das bibliotecas

2.8.1 Bibliotecas do GUFF

As bibliotecas utilizadas são livres e, em linhas gerais, foram utilizadas para abstrair os detalhes específicos dos sistemas operacionais e de implementação. Segue uma pequena descrição:

- SDL (*Simple DirectMedia Layer*) [41] - Para a criação e manipulação de janela e dispositivos de entrada e saída, está disponível para diversos sistemas operacionais;
- OpenGL [30] - É a biblioteca gráfica voltada para construção de imagens 3D em tempo real. É uma das mais utilizadas na área de visualização e jogos digitais. Foi desenvolvida pela Silicon Graphics em 1992;
- GLEW (*OpenGL Extension Wrangler*) [33] - É a biblioteca de extensão do OpenGL. Apresenta recursos que não estão ainda incorporados na biblioteca principal do OpenGL;
- boost [19] - É um conjunto de bibliotecas voltada para soluções de gestão de memória, programação concorrente, entrada e saída, entre outras;
- LIB3DS [42] - É uma biblioteca voltada par manipulação de arquivos com estrutura 3DS, que é a estrutura utilizada pelo software 3d Studio Max, usada na modelagem de objetos tridimensionais;
- FTGL[43] - Esta biblioteca traz suporte ao uso de tipos de fontes de texto em aplicações que usam a API gráfica do OpenGL;
- audiere [44] - Biblioteca para decodificação e gerência de áudio, com suporte a diversos formatos de arquivo;
- LUA [10] - É uma linguagem de *script*, possui um interpretador que permite seu uso como qualquer outro tipo de linguagem. Com o auxílio de sua biblioteca, é possível estender a aplicação, embutindo o interpretador, fazendo com que a mesma possa executar e trocar informações com o *script* e
- DevIL (*Developer's Image Library*) [45] - Esta biblioteca traz suporte para manipulação de diversos formatos de arquivos de imagem.

2.8.2 A Gestão Automatizada de Recurso do GUFF

Valente [35] define como recurso algo do sistema operacional, em quantidade limitada, que para ser utilizado necessita de duas comunicações com o sistema operacional. A primeira para solicitar o recurso desejado e a segunda para informar do seu término. Já Tanenbaum [46] define o recurso como qualquer coisa que pode ser utilizada somente por um único processo em qualquer instante. O *framework* GUFF, visando diminuir a carga de responsabilidade do desenvolvedor, implementa a gestão automática de recurso utilizando propriedades da linguagem C++, tornando a aplicação mais segura e de fácil manutenção. Através dos objetos chamados de automáticos, aqueles que possuem a capacidade de gerir seus próprios recursos, este paradigma é alcançado.

2.8.3 A Camada de Aplicação do GUFF

O *framework* GUFF possibilita a modelagem da camada de aplicação como uma máquina de estados, devido ao fato de que os jogos digitais são decompostos em vários estados. Os estados são organizados de uma forma hierárquica, onde cada filho tem que ter um pai. Há um estado mestre sobre o qual os demais são subordinados, conforme pode ser observado na figura 2.7.

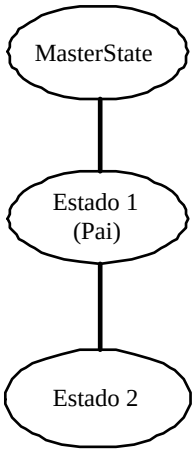

Figura 2.7: Hierarquia de estados.

2.8.4 O *Toolkit* do GUFF

O *toolkit* do *framework* GUFF disponibiliza funcionalidades para solucionar diversos problemas de visualização, matemática, áudio, tratamento de dispositivos de entrada, entre outros. O *toolkit* é classificado em cinco módulos divididos em espaços de nomes (*names-*

paces), usados para tratar soluções relacionadas. A figura 2.8 ilustra, de forma esquemática, como os módulos estão estruturados.

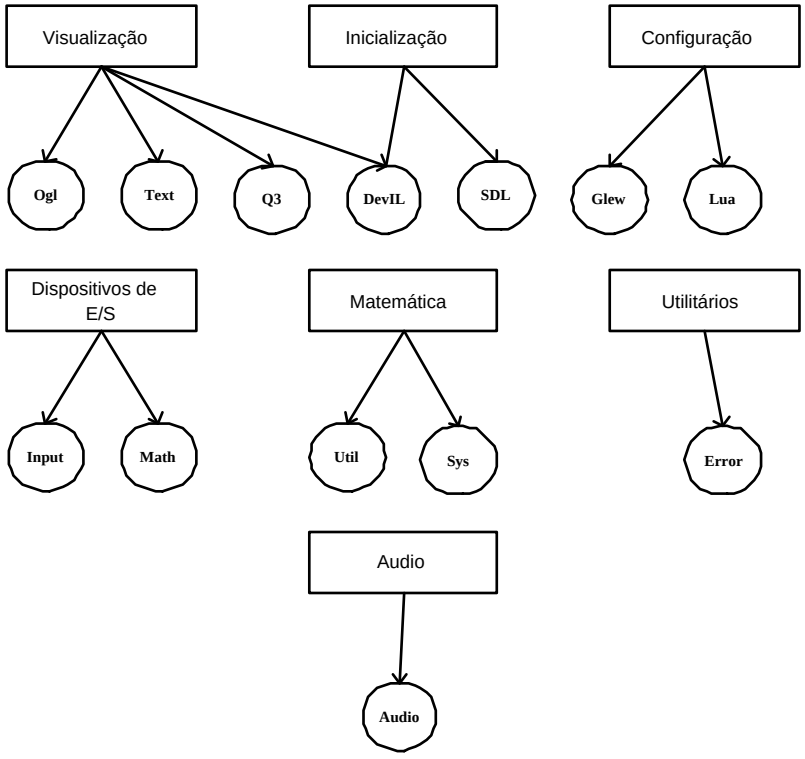

Figura 2.8: Relacionamento entre os módulos e *namespaces* do *framework* GUFF.

2.9 Habilitando Módulo de Shader no Framework GUFF

O *framework* GUFF traz suporte a diversos recursos, entretanto, não permite programação em Shader. Uma das contribuições dessa dissertação é habilitar um módulo de Shader, estendendo o projeto e viabilizando para implementação dos estudos de caso. A incorporação do shader como recurso manteve o padrão de gestão automática de recurso [35]. O suporte a shader foi incorporado como um recurso dentro do GUFF, permitindo que a classe abstrata shader possa ser estendida para implementar, futuramente, outras linguagens, conforme diagrama de classes (figura 2.9).

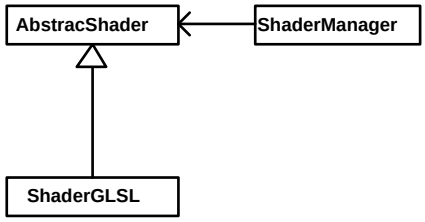

Figura 2.9: Diagrama de classes da classe shader e do gerenciador de shader.

2.9.1 Gerenciamento de Recurso - Shader

Baseado no conceito de automática gestão de recursos, descrito na seção 2.8.2, a gerência de shaders é realizada de forma automática, independentemente do desenvolvedor. A criação e o gerenciamento são feitos por meio de uma classe gerenciadora de shaders, conforme diagrama de classes (figura 2.9). Este gerenciador tem a função de armarzenar os shaders compilados, bem como todos os seus parâmetros, tenham eles qualificadores *uniform* ou *attribute*. Um shader é composto por dois códigos diferentes: um escrito para ser processado pelos processadores de vértice e outro escrito para os processadores de *pixel*. Quando estes shaders são invocados pela primeira vez, são compilados e armazenados pelo gerenciador. O gerenciador armazena os shaders compilados e constrói um *buffer* de programas. Novas instâncias de um programa shader armazenado no *buffer* são apenas ativadas, suprimindo o tempo gasto com a leitura dos arquivos fontes e o tempo da compilação.

2.9.2 Como Escrever Shaders Para o GUFF

O processo de desenvolvimento do shader tem seu inicio na ferramenta RenderMonkey. Através desta, o código, de vértice e de *pixel*, é construído, compilado e depurado, inclusive aplicando sobre malhas de modelos 3DS. Ao final do desenvolvimento do shader, cada programa (vértice e *pixel*) é salvo em arquivos próprios.

Os projetos que utilizam o *framework* GUFF com shaders precisam do nome dos arquivos salvos pela ferramenta RenderMonkey. Estes arquivos são passados para o gerenciador que carrega, compila e passa para o objeto shader o código compilado, o objeto shader que o gerenciador previamente instanciou. O objeto shader para aplicar o efeito necessita ser ativado durante o estagio de visualização. O diagrama de seqüência, figura 2.9, mostra a troca de mensagem durante a execução da aplicação. A aplicação é definida pela classe SimpleApp, o shader pela classe ShaderGLSL e o gerenciador pela classe ShaderManager, conforme figura 2.10.

2.10 Conclusão do Capítulo

Os conceitos de jogos digitais, aplicações em tempo real, GPU, linguagem shader e uso da GPU para propósito genérico, discutidos neste capítulo, são necessários para compreensão desta dissertação. Também é apresentada uma solução para manipulação e gerência de

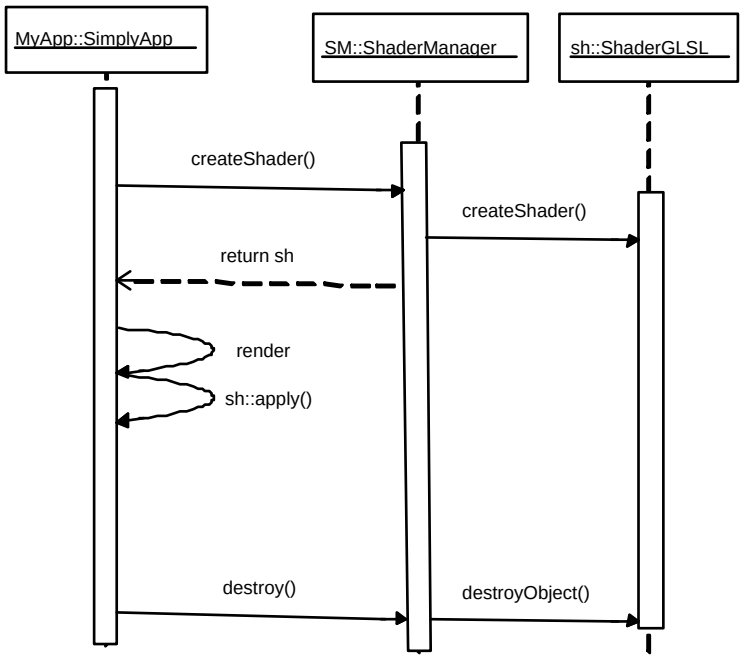

Figura 2.10: Seqüência da aplicação invocando o shader.

shader em aplicações de tempo real, em especial jogos digitais, o módulo de shader. Este módulo descrito no capítulo foi uma solução adotada para tratar o uso de shader como um recurso através de uma classe shader, conforme ilustrado no diagrama de classes 2.9, as APIs necessárias ao uso e gerência são encapsulados pela classe que utiliza o paradigma de gestão automática de recursos, retirando do desenvolvedor a responsabilidade de gerenciar a memória com o uso do shader.

O módulo de shader mostra uma solução interessante com algumas vantagens. A primeira é que, por meio deste módulo, é possível haver diferentes shaders dentro da mesma aplicação, bastando apenas ativar o desejado. A segunda vantagem é sua capacidade de autogestão e a última é a própria estrutura baseada na programação orientada a objeto. Está estrutura hierárquica de classes permite facilmente a implementação de outras linguagens shaders como CG ou HLSL. Assim este módulo é a primeira das contribuições que esta dissertação apresenta.

O presente capítulo atinge seus objetivos,conceitos sobre aplicações de tempo real, linguagem shader e técnica de GPGPU são apresentados e o desenvolvimento de uma biblioteca shader.

Capítulo 3

Modelo Multithread Desacoplado com Estágio de GPGPU

Baseado no modelo *multithread* desacoplado apresentado em [27], figura 3.1, o modelo proposto neste capítulo, denominado de modelo *multithread* desacoplado com estágio de GPGPU, introduz um novo estágio, responsável pelo processamento genérico em GPU. O objetivo desse novo estágio é utilizar a GPU como co-processador matemático e físico, executando em paralelo alguns estágios da aplicação.

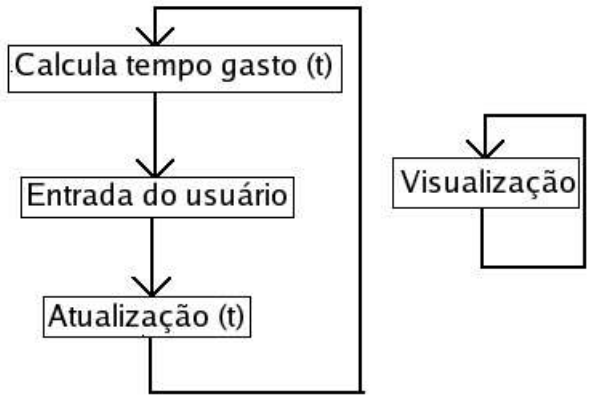

Figura 3.1: Modelo *multithread* desacoplado com sincronização.

O modelo *multithread* desacoplado com estágio de GPGPU é composto por três *threads* conforme ilustra a figura 3.2. A primeira *thread* trata do estágio de visualização, a segunda trata do estágio de atualização e terceira *thread* trata das tarefas que são processadas pela GPU.

Uma questão que surge em ambientes multitarefas é o compartilhamento de informações. Na arquitetura aqui usada, há o compartilhamento dos atributos dos objetos em

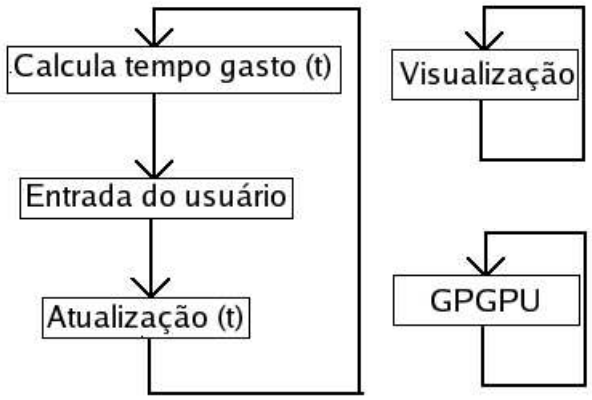

Figura 3.2: Modelo *multithread* desacoplado com GPGPU.

cena pelas *threads*. Assim como em qualquer sistema *multithread*, com compartilhamento de informação, é necessário adotar uma forma de sincronismo cujo objetivo é evitar inconsistências. A forma e o objeto de sincronismo, desenvolvido na presente arquitetura, são ainda discutidos neste capítulo.

O presente capítulo apresenta uma arquitetura de jogo digital com um estágio de GPGPU e está organizado em seções. Na seção 3.1, é debatido o estágio de GPU no ciclo de jogo, o uso de *threads* e o modelo *multithread* proposto são discutidos na seção 3.2, nas seções 3.3 a 3.7 é apresentada a implementação do modelo e o estudo de caso usado para validar o mesmo. Finalmente, na seção 3.8 é apresentada a conclusão.

3.1 O Estágio de GPU

O modelo proposto neste capítulo apresenta uma arquitetura inovadora, utilizando a GPU como co-processador matemático e diminuindo a carga de processamento da CPU, conseqüentemente, aumentanto o desempenho do processamento de outros estágios da aplicação, tais como IA, lógica, comunicação, etc.

Como discutido na seção 2.5, as GPUs usam um paradigma de programação diferente do tradicional, exigindo uma modelagem diferenciada dos problemas tratados na aplicação, motivo pelo qual houve a necessidade da introdução de um estágio novo [40].

O estágio de GPU pode ser considerado como uma interface entre a aplicação e a GPU. Sua criação permite que sua execução seja feita em uma *thread* própria, proporcionando um grau de concorrência entre os estágios do ciclo de jogo.

3.2 As *Threads* do Modelo *Multithread* Desacoplado com Estágio de GPGPU

Antes de iniciar a discussão sobre o uso de *threads*, no modelo ora apresentado, é necessário um entendimento de como este recurso poderoso, disponibilizado pelos sistemas operacionais, é implementado.

A criação de *threads*, em sistemas operações como o Windows e os baseados no Unix, gera uma relação entre as *user-threads* (*threads* do usuário ou aplicação) e *kernel-threads* (*threads* em modo *kernel* que fazem chamadas ao sistema - *system call*), três tipos de relação existem entre as *user-threads* e *kernel-threads* [11]:

- Relacionamento n *user-threads* para 1 *kernel-thread*: Este modelo mapeia uma relação de muitas *user-threads* para um único *kernel-thread*, isto é, há apenas um modo *kernel* para atender todas as *threads*. Portanto, quando há chamadas ao sistema operacional de mais de uma *thread*, concorrentemente, no modo usuário, apenas uma é atendida. As demais *thread*, que também fizeram chamadas ao sistema operacional, ficam aguardando para serem atendidas.

- Relacionamento 1 *user-thread* para 1 *kernel-thread*: Mapeamento caracterizado por haver um único *kernel-thread* para cada *user-thread*, isto é, para cada *thread* criada, há um modo *kernel* relacionado, permitindo chamadas simultâneas ao sistema pelas diversas *threads*, inclusive alocando múltiplas *threads* em múltiplos processadores. Este modelo cria um *overhead* no contexto das *threads*, causando um efeito contrário ao desejado, isto é, a degradação do sistema;

- Relacionamento n *user-threads* para n *kernel-threads*: Este modelo é caracterizado por alocar vários *kernel-threads* em menor quantidade que os *user-kernel*. Tem o objetivo de resolver os problemas apresentados pelos dois modelos anteriores, alocando quantas *user-threads* necessárias e compartilhando estas entre as *kernel-threads* e proporcionando um ambiente *multithread* sem degradação do sistema.

A viabilidade do atual modelo necessita do uso de *threads*. Para facilitar a portabilidade e baseado no padrão de desenvolvimento do GUFF, foi construída uma abstração para manipulação e gerenciamento de *threads*, encapsulando as APIs necessárias, conforme ilustra o diagrama de classes (figura 3.3). A figura 3.2 mostra uma arquitetura composta do ciclo principal de duas *threads*. Cada um destes componentes foi criado

através de uma definição de classes apropriadas. O ciclo principal deriva de uma classe abstrata que implementa funcionalidades de gerência e execução do ciclo principal do jogo. As duas *threads*, a de processamento em CPU e de processamento em GPU, derivam de uma classe abstrata que trata da parte de gerência, criação e destruição de *threads*. A primeira implementa a codificação necessária para executar o estágio de atualização e a segunda implementa a codificação para realizar chamadas ao estágio de GPGPU. Os estágios de entrada de usuário e visualização ficam na *thread* principal.

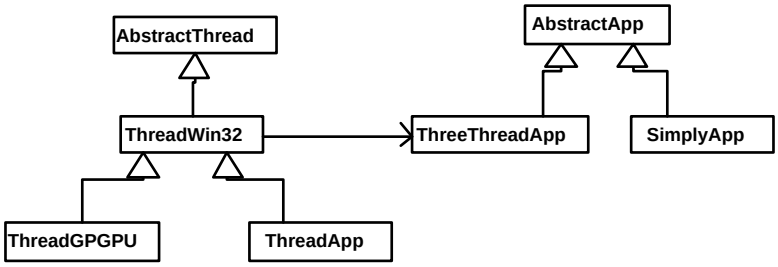

Figura 3.3: Diagrama de classe da classe *thread*.

A abstração garante uma camada de interface entre a aplicação de visualização em tempo real e a biblioteca utilizada. Entre as bibliotecas disponíveis, a adotada foi a Win32. Desta forma, a classe abstrata usa as APIs da biblioteca Win32 para criação e gerenciamento de *threads*.

A biblioteca Win32 foi escolhida por ser uma biblioteca gratuita disponível para o sistema operacional Windows e por ter suporte ao compartilhamento de *threads* em modo *kernel*, modelo de relacionamento n *user-threads* para n *kernel-threads* [47], apresentado como o modelo mais eficiente de relacionamento entre os modos de thread [11], embora o uso específico desta biblioteca comprometa sua portabilidade.

No modelo proposto, o sincronismo é realizado entre duas das três *threads*, isto é, a sincronização é aplicada entre as *threads* que alteram o estado de objetos dentro do cenário. Os estágios de atualização e GPGPU modificam o mesmo conjunto de informações, portanto, o sincronismo é realizado entre as *threads* que tratam desses estágios, garantindo a mútua exclusão sobre a seção crítica. Cada *thread* aguarda a sinalização da outra para entrar na seção crítica e, ao final, envia uma mensagem sinalizando que saiu da seção crítica. Neste momento, a *thread* que estava aguardando a sinalização recebe a mensagem e entra na seção crítica. As *threads* ficam alternando o acesso à seção crítica [48].

3.2.1 Aspectos da Implementação de Threads

Alguns aspectos existentes em sistemas *multithreads* foram tratados pela abstração em classe. O primeiro é a suspensão da *thread* que consiste na paralisação da mesma, entretanto, não há como determinar sobre qual ponto do conjunto de instruções, que a *thread* está executando, a suspensão irá ocorrer. Neste caso, a falta de garantia do ponto de retorno de uma *thread* suspensa é um problema, eventualmente ocasionando inconsistência na informação processada pela *thread*. A abstração, na forma de classe, construída trata a suspensão de forma diferente: antes de suspender a *thread*, é feito um sincronismo interno, garantindo sempre o mesmo ponto de retorno. É importante ressaltar que a suspensão não deve ser usada como mecanismo de sincronização [11], uma vez que isto implicaria em um custo computacionalmente maior, isto é, a suspensão tem o custo do uso de um objeto de sincronismo mais o custo da própria instrução de suspensão, ao passo que o sincronismo tem somente o custo o objeto de sincronismo. A segunda questão a ser tratada é a alocação e desalocação do contexto. A alocação de *thread* gera um contexto alocado pelo sistema operacional, contexto composto de espaço de memória, contador de programa e espaço na pilha. Todo o contexto alocado tem que ser liberado ao fim da execução da aplicação. A abstração de *thread*, ao receber uma mensagem de finalização, processa todas as instruções necessárias para liberação dos recursos alocados no sistema operacional de forma automática, garantindo uma gestão automática de recurso [35].

3.3 Modelo *Multithread* Desacoplado com Estágio de GPGPU - Estudo de Caso

Para validar o modelo proposto, foi implementado o tratamento de colisão entre corpos sólidos móveis. A colisão entre estes é uma tarefa composta por duas etapas: a primeira é a identificação da colisão e a segunda identifica o que fazer quando uma colisão ocorreu. Para tratar a colisão, o algoritmo foi baseado em esfera, embora outros algoritmos possam ser facilmente implementados. A modelagem do estudo de caso aplica apenas a troca de velocidade como tratamento de colisão. No estágio de GPGPU, as informações que devem ser processadas na GPU são representas na forma de texturas. Portanto, no estudo de caso, os atributos referentes aos copos sólidos móveis, tais como posição, velocidade e aceleração, são colocados em vetores diferentes, um para cada atributo. Em seguida, estes vetores são transferidos para a placa gráfica. No processamento, o programa de *pixel* trata as colisões e escreve, em um *buffer* de saída, os resultados. Estando o resultado de

todas as colisões em um *buffer* de saída, este é lido e o estado de cada corpo sólido móvel do cenário é atualizado.

3.3.1 Modelagem do Problema no Estágio de GPGPU

A primeira etapa do estágio de GPGPU é a representação das informações em textura. Tratando-se de colisão, os atributos dos corpos sólidos móveis mapeados são: posição, velocidade e aceleração. A aplicação define um vetor de ponteiros de corpos sólidos móveis de tamanho n, onde n representa o número de corpos sólidos móveis no cenário. Cada corpo sólido móvel tem seu endereço incluído no vetor.

O estágio de GPGPU, ao ser invocado, copia os atributos dos sólidos móveis para três outros vetores: vetor de posição, vetor de velocidade e vetor de aceleração. Este conjunto de vetores compõe uma matriz de tamanho $n \times 4$, onde n é o numero de sólidos móveis no cenário e a constante 4 define que cada sólido móvel terá 4 posições, ou seja, está ocupando um texel com 4 canais de cores (RGBA - *Red*, *Green*, *Blue* e *Alpha*) [40].

Atualizados os vetores de atributos, estes são enviados à placa gráfica por intermédio da classe GPGPU, que, após enviar todas as texturas, ativa cada uma delas.

Ativadas as texturas, é executado o shader de GPGPU, ou seja, a colisão entre os sólidos móveis do cenário é processada. A execução de um programa shader gera um resultado que está associado a uma janela de visualização [40], mesmo que o processamento seja feito usando técnicas de GPGPU.

Neste caso, um problema surge: executar o estágio de GPGPU sem comprometimento da visualização da aplicação de tempo real. A biblioteca SDL, utilizada como gerenciador de janelas do GUFF, não tem suporte a múltiplas janelas, o que impede a criação de uma segunda janela para ser associada ao estágio de GPGPU. Portanto, para solucionar tal problema, foram utilizadas APIs da biblioteca Win32, no que trata da parte de gerência de janelas [47]. Os recursos disponíveis por esta biblioteca permitem que seja criada um segunda janela, associada ao estagio de GPGPU, a qual não é exibida, apenas criada durante o processo de inicialização da aplicação e devidamente destruída quando a aplicação é finalizada.

O algoritmo utilizado é baseado em esfera, sendo implementado na linguagem shader, processado apenas pelos processadores de *pixel*. Outros algoritmos de colisão baseados em *boundbox*, tais como os baseados em cubo ou cilindro, podem ser adaptados.

O programa shader tem dois ponteiros para cada textura de atributo. Estes atributos são dos corpos sólidos e referem-se à posição, velocidade e aceleração. Sobre o segundo ponteiro de cada atributo, aplica-se um deslocamento. Este deslocamento garante a colisão entre dois corpos diferentes, isto é, a comparação é realizada entre *pixels* diferentes. O descolamento é incremental e trabalha em conjunto com a aplicação que invoca o shader de colisão, invocando a quantidade de corpos sólidos menos um. A figura 3.4 mostra uma ilustração de como o programa shader interpreta os *pixels*.

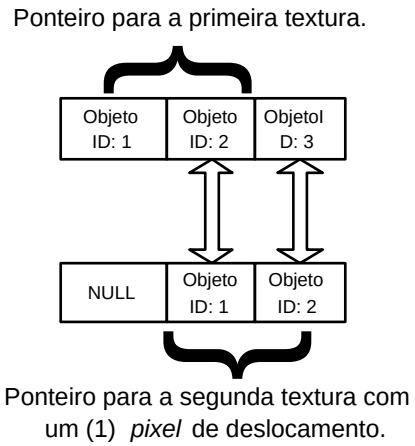

Figura 3.4: Representação do objetos na textura.

Ao final da execução do programa de *pixel*, um *buffer* de saída com o resultado das colisões é produzido. Este *buffer* é lido pela aplicação e o resultado é transferido para um vetor, que está no estágio de GPGPU, de tamanho igual aos vetores dos atributos dos sólidos móveis. Este vetor com a textura de saída é então varrido pela CPU. A posição referente ao canal alpha do *pixel* é conferida. Quando há a indicação de colisão, este canal tem seu valor igual a 1 e o estado dos copos sóldios móveis são atualizados.

3.3.2 Tratando Colisões - Detalhes da Implementação

A tarefa de colisão, anteriormente tratada pela CPU, passa a ser resolvida pela GPU por meio de técnicas de processamento de GPGPU. A colisão entre todos os corpos sólidos móveis do cenário gera uma combinação:

$$C_n^2 = \binom{n}{2} \tag{3.1}$$

onde n representa o número de corpos sólidos móveis. Para garantir a quantidade necessária de combinaçõe entre a colisão dos corpos sólidos móveis, o programa shader é executado $(n-1)$ vezes. A informação do deslocamento aplicado sobre os ponteiros da

segundas texturas, dos atributos posição, velocidade e aceleração, é passado como parâmetro ao programa shader.

3.3.3 Testes do Modelo - *Multithread* Desacoplado com Estágio de GPGPU

O principal aspecto em um modelo *multithread* é a identificação de quais são as informações que são compartilhadas por mais de uma *thread*. Para evitar inconsistência da informação, é necessário garantir a exclusão mútua [48] por meio da seção crítica.

A figura 3.5 mostra quais estágios estão sendo tratados concorrentemente e quais estão sendo executados alternadamente. Os estágios de atualização da câmera e visualização sempre ocorrem. Porém, o estágio de atualização e colisão dos corpos sólidos móveis não podem ocorrer juntos. A restrição é garantida pela sinalização entre as *threads* através do uso de um objeto de sincronismo. O sincronismo é realizando entre as *threads* de CPU e GPU, as quais executam os estágios de atualização e de GPGPU (tarefa de colisão). Cada *thread* tem duas variáveis (*RedSync* e *GreenSync*) que são responsáveis por garantir a sincronização. Seus valores são compartilhados, ou seja, a *RedSync* de uma *thread* compartilha o valor da *GreenSync* da outra *thread*, assim, executando exclusivamente o acesso à seção crítica. A figura 3.5 também mostra como os estágios estão distribuídos entre a CPU e GPU: o ciclo principal é processado pela CPU e é responsável pela câmera e atualização dos sólidos; visualização e a *thread* de GPGPU são processados na GPU e responsáveis pela visualização da cena e a detecção da colisão em GPGPU. Embora as *threads* garantam a concorrência no processamento, a visualização da cena ocorre apenas quando todos os objetos já foram atualizados, tanto no estágio de GPGPU quando no estágio de atualização.

Para avaliar o modelo proposto, testes de performance foram realizados e comparados com o modelo de ciclo de jogo original do GUFF. O hardware usado foi um Pentium D 3.4Ghz, 1GB de memória *dual channel* e placa gráfica NVIDIA geForce 6200 AGP 8x.

Dois grupos de testes foram aplicados. O primeiro usou o modelo *single thread* acoplado sincronizado e o outro usou o modelo *multithread* desacoplado com estágio de GPGPU. Para cada grupo, dez testes foram executados medindo o tempo de processamento da CPU e GPU. Cada teste consistiu em 500 colisões entre 16 sólidos móveis iniciados a partir de uma trajetória, velocidade e aceleração aleatórios, sem nenhuma interação por parte do usuário.

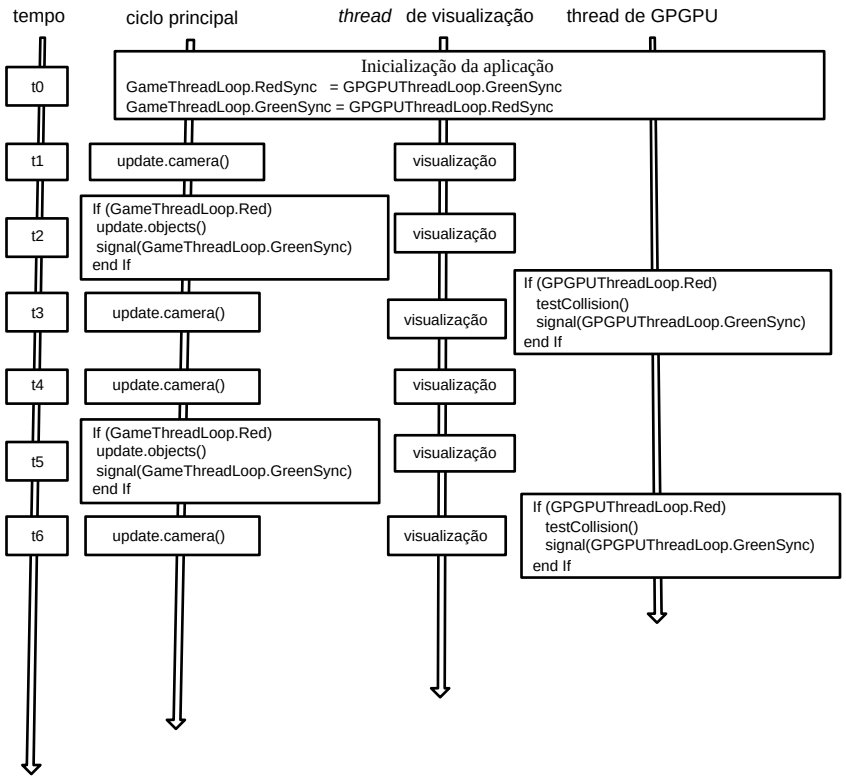

Figura 3.5: Representação dos estágios em paralelo.

As tabelas 3.1 e 3.2 mostram os resultados correspondentes aos modelos *multithread* com GPGPU e *single thread* sincronizado acoplado, respectivamente. A coluna **tempo** representa o tempo gasto no processamento das 500 colisões, **tempo máximo** representa o tempo total do teste, **processamento** é o tempo efetivo de processamento da GPU ou CPU (todos os tempos estão marcados em segundos) e **FPS** é o número de quadros por segundos.

Tabela 3.1: Resultados: Modelo *multithread* desacoplado com GPGPU.

tempo	**tempo máximo**	**processamento**	**FPS**
1,578	0,015	0,030000014	77,3131
1,938	0,016	0,047000030	79,9794
1,703	0,016	0,062999996	76,9231
2,219	0,016	0,032000001	84,2722
1,719	0,016	0,032000004	77,9523
2,093	0,016	0,032000085	84,5676
2,234	0,016	0,016000003	83,2587
1,968	0,016	0,045999954	82,8252
1,312	0,016	0,030999969	70,8841
2,203	0,016	0,015999996	83,9764

A tabela 3.3 representa uma comparação dos tempos médios das 10 instâncias de teste entre os processadores. A linha de titulo **FPS** representa o número de quadros por

Tabela 3.2: Resultado: Modelo *singlethread* sincronizado acoplado.

tempo	**tempo máximo**	**processamento**	**FPS**
22,625	0,016	0,063000096	240,442
83,594	0,016	0,330000447	242,290
44,234	0,016	0,232999674	242,302
27,140	0,016	0,329000170	244,068
42,828	0,016	0,266999744	240,870
50,063	0,016	0,234000313	242,714
14,328	0,016	0,063999999	238,554
15,328	0,016	0,126000026	238,192
30,188	0,016	0,109999970	241,420
22,110	0,016	0,172999781	240,299

segundo, **processamento** o tempo efetivo de processamento, as linhas **tempo**, **minimo** e **máximo** correspondem aos tempos mínimos e máximos, respectivamente, de processamento. Todos tempos são dados em segundos.

Tabela 3.3: Comparativo ente GPU e CPU

	GPU	**C**PU
FPS	80,1952100	241,1151000
processamento	0,0345000	0,1929000
tempo	1,8967000	35,2438000
minimo	0,0160000	0,0630001
máximo	0,0630000	0,3300004

Os resultados mostrados pelas tabelas 3.1, 3.2 e 3.3 mostram um ganho de performance obtido pelo uso do modelo *multhread* desacoplado com estágio de GPGPU. O aumento de performance é devido a execução concorrente dos estágios e ao uso da GPU.

A introdução do estágio de GPGPU no tratamento de colisão entre os sólidos móveis reduziu significativamente o tempo de processamento da própria colisão. Por outro lado, os testes mostram uma perda na taxa de quadros por segundos, ocasionada pelo próprio uso da GPU. Entretanto, esta perda não afeta a qualidade da animação, que continuou preservada, uma vez que a taxa ficou em torno de 80 quadros por segundo, uma taxa ainda maior que a taxa de atualização do próprio vídeo.

3.4 Conclusão do Capítulo

Este modelo demonstra a introdução de um novo estágio em arquiteturas *multithread* desacoplada em motores de jogos digitais, responsável pelo processamento genérico na

GPU, exemplificado pela detecção de colisão. Outros estágios, como IA e física podem ser modelados para serem tratados em GPU, bastando modelar de forma correta os dados em texturas e desenvolver um algoritmos em shader correspondentes.

A introdução de um novo estágio aplicando um mecanismo de balanceamento de carga entre os processadores é uma abordagem interessante, no sentido de obter um uso mais eficiente dos recursos computacionais em jogos digitais. Fazendo isto, é possível utilizar o poder computacional em outros estágios, tais como IA e física.

O estudo de caso e os resultados obtidos demonstram que o uso da GPU no processamento de propósito geral é um linha de pesquisa promissora, aumentando o desempenho em motores de jogos digitais de sistemas de simulação de tempo real.

As placas gráficas mais modernas do mercado, como a série 8 da geForce [49], apresentam muito mais recursos no processamento de propósito geral nas GPUs, permitindo, inclusive, a alocação de uma de suas GPUs apenas para gerenciar o balanceamento de carga entre os processadores disponíveis, criando novos campos de pesquisa.

O presente capítulo atinge seus objetivos, apresenta uma arquitetura de ciclo de jogo com um estágio de GPGPU, no qual é viabilizado o processamento na GPU.

Capítulo 4

Modelo Multithread Desacoplado com Distribuição Dinâmica de Tarefas

O modelo *multithread* desacoplado com distribuição dinâmica de tarefas apresenta um método eficiente de processamento híbrido usando *threads*, construído com base em uma evolução do modelo *multithread* desacoplado com estágio de GPGPU. Este modelo apresenta uma forma mais eficiente de uso da GPU como co-processador matemático e no processamento de diversos estágios, baseado em um *script* de configuração, pois, através deste, determina-se onde cada estágio é processado, permitindo otimizar o uso dos processadores conforme a arquitetura de hardware utilizada.

As principais diferenças em relação ao modelo anterior são a presença de um gerenciador de tarefas e a distribuição dos estágios entre as *threads*, que é feito dinamicamente. O modelo é composto por três *threads*, semelhante ao modelo anterior, com exceção dos estágios de entrada de dados do usuário e o de visualização, que são estáticos. Os demais são tratados como tarefas e são vistos como uma abstração de processos [46]. O gerenciador de tarefas é visto como uma abstração de servidor de tarefas e as *threads* são vistas como clientes de tarefas.

O presente capítulo apresenta uma evolução do modelo de ciclo de jogo apresentado no capítulo anterior. No modelo apresentado neste capítulo, é permitido ao desenvolvedor determinar onde cada tarefa será processada, na CPU ou GPU, conforme *script* de configuração em linguagem Lua. Assim, na seção 4.,1 é apresentado o novo modelo baseado no modelo debatido no capítulo anterior. O módulo de gerência de tarefas e sua integração com o *script* de configuração são apresentados nas seções 4.2 e 4.3 e, na seção 4.4, é apresentada a conclusão do capítulo.

A figura 4.1 mostra a arquitetura do modelo proposto, o ciclo principal, responsável

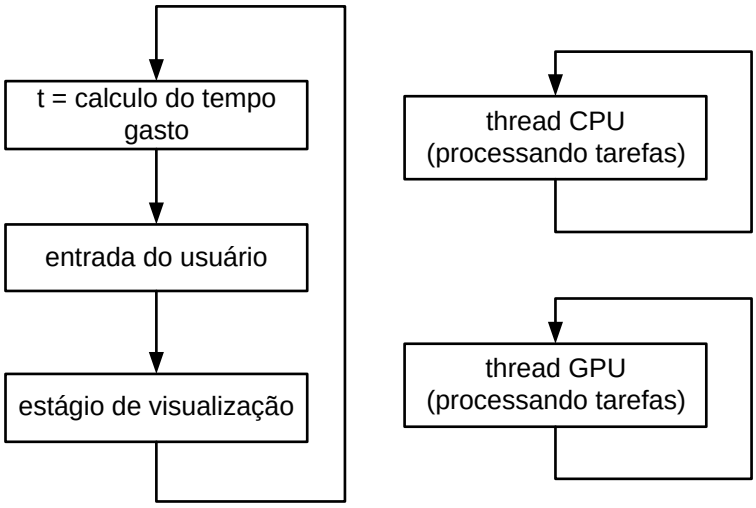

Figura 4.1: Modelo *multithread* desacoplado com distribuição dinamica de tarefas.

pela execução dos estágios estáticos e duas *threads* como função o processamento na CPU e de GPU. Cada uma delas carrega e processa uma tarefa por vez, conforme decisão do gerenciador de tarefas que, por sua vez, tem como fonte de sua decisão um *script* de configuração baseado na linguagem Lua [10].

4.1 Tarefas no Modelo *Multithread* Desacoplado com Distribuição Dinâmica de Tarefas

O modelo proposto é baseado no conceito de tarefas, funcionando como uma camada intermediária entre o estágio da aplicação e os processadores. Cada estágio corresponde a uma ou a um conjunto de tarefas, como a lógica, física, IA e colisão de um jogo digital.

Uma tarefa é modelada como uma especialização da classe abstrata. A adoção de uma classe abstrata garante que o carregamento seja realizado de forma semelhante por ambas as *threads*. A classe abstrata possui diversos métodos, dentre eles: *execCPU* e *execGPU*. Em cada um desses métodos, é feita a codificação necessária para permitir a execução da tarefa por ambos os processadores, CPU ou GPU.

O processo de codificação para GPU pode ser complexo, uma vez que usa um paradigma diferente de programação[40]. A modelagem de um estágio da aplicação para ser processado em GPU envolve três etapas. A primeira trata do mapeamento da descrição do problema para um conjunto de texturas, a segunda é a própria codificação de um programa de *pixel* e a terceira e última etapa é a codificação da solução produzida pela GPU a partir das texturas que representam o resultado.

Além da complexidade do modelo de programação em GPU, outros problemas podem surgir no presente modelo, como uma tarefa a ser executada por ambos os processadores. Para tratar esse problema, cada tarefa tem um sinalizador que indica por qual processador está sendo tratada ou indica que não está sendo processada. Na programação *multithread*, surgem problemas de acesso à informações compartilhadas. As soluções aplicadas aos problemas de *thread* são as mesmas do modelo anterior e tratadas da mesma forma.

4.1.1 Tarefas Que Podem Ser Tratadas em GPU

As tarefas que podem ser tratadas em GPU são aquelas que envolvem cálculos matemáticos complexos e que podem ser tratados como um fluxo de informação, em especial, simulações físicas.

A aplicação de física para jogos digitais são implementadas em GPU, maximizando o poder de processamento da GPU e liberando a CPU para trabalhar outras tarefas. Exemplos destas soluções são as representações de matrizes e vetores [6], onde se implementam a solução para o gradiente do conjugado de matriz esparsa e solução para problemas de *multigrid* de *grids* regulares. Outros exemplos são apresentados por: [9] e [50] para tratar problemas de álgebra linear e solução para sistemas lineares densos apresentados por [7].

A NVIDIA fornece um conjunto de APIs para processamento de propósito geral em GPU, como o tratamento de colisão entre corpos sólidos móveis, embora a escolha sobre qual processador uma tarefa será processada não depende unicamente do poder computacional das GPU. Outro fator fundamental na eficiência do processamento de uso geral na GPU é o barramento da placa gráfica e da memória principal.

4.1.2 Codificação para GPU e Programa de *Pixel*

Os três estágios básicos de codificação para o uso geral da GPU são desmembradas em cinco estágios bem definidos.

- Mapeamento do problema em texturas.
- Transferência para memória da placa gráfica.
- Codificação da GPU através de um programa de *pixel*.
- Execução do shader.

- Leitura da memória da placa gráfica.
- Mapeamento da textura de saída na solução do problema.

Algumas destas etapas eram realizadas dentro do estágio de GPGPU do modelo anterior, lançando mão do conceito de tarefa. Este mapeamento passou a ser realizado pela classe tarefa, no seu método *execGPU*. Em linhas gerais, o problema é mapeado em um ou mais vetores. Estes vetores, em seguida, são transferidos para a memória da placa gráfica, sendo esta transferência realizada por intermédio da classe GPGPU, conforme ilustado no diagrama de classes 4.2. Dentro da memória da placa gráfica, estes vetores passam a ser vistos como texturas.

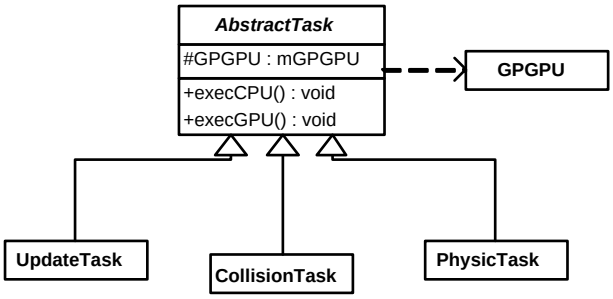

Figura 4.2: Diagrama de classes de tarefas.

A leitura da memória da placa gráfica e o mapeamento da textura de saída na solução do problema são também implementados dentro do método *execGPU*. A textura de saída é copiada para um vetor (de mesmo tamanho dos vetores de entrada), onde então as informações resultantes do processamento das GPUs.

As restrições da solução apresentada são os limites da placa gráfica e a complexidade da modelagem do problema para o paradigma da arquitetura SIMD. Os parâmetros que algumas tarefas podem ter são passados pelo *script* de configuração, observando a restrição de que todos os parâmetros devem ter valores padrões.

4.2 O Gerenciador de Tarefas

O gerenciador de tarefas é o núcleo do modelo proposto, sendo responsável por distribuir e gerenciar as tarefas para ambos os processadores CPU e GPU. A aplicação que usar esta arquitetura considera ambos os processadores em um mesmo nível de abstração, bastando apenas que a tarefa seja executada, não importando qual processador vai trabalhar. O gerenciador de tarefas irá prover as tarefas conforme a necessidade da aplicação. Para

este, é importante saber qual processador vai executar uma determinada tarefa. As regras obedecidas pelo gerenciador de tarefas são determinadas num *script* de configuração, que dirá qual processador deve ser alocado para cada tarefa e a ordem de execução de todas as tarefas. O gerenciador é visto pela aplicação como um servidor dedicado de tarefas, respondendo aos seus clientes, as *threads* de CPU e GPU.

Durante o ciclo de vida da aplicação, é feita uma constante comunicação entre as *threads* e o gerenciador de tarefas, as primeiras requisitando tarefas e o último enviando as mesmas, ou seja, as tarefas. A comunicação é baseada em um protocolo próprio.

O gerenciador de tarefas é um módulo dentro da aplicação, estruturado na forma de classe e construído como um modelo *singleton*[1] de classe. O conceito de tarefa e a estrutura de gerência apresentam um novo paradigma de modelo de ciclo de jogo, onde um conjunto de tarefas pode ser executado por diferentes arquiteturas de processador. Como um servidor, o gerenciador de tarefas deve atender, eventualmente, requisições simultâneas para evitar problemas de acesso concomitante por parte do próprio gerenciador à sua tabela de tarefas, simplificando a arquitetura a fim de não utilizar objetos de sincronismo. A classe gerenciadora é dotada de duas tabelas de tarefas, cada uma atende a requisição de um processador.

A figura 4.3 mostra um diagrama com a classe gerenciador de tarefas, chamada de *TaskManager* ou pela sigla TM, bem como as *threads* de CPU e GPU e os métodos responsáveis pela troca de mensagens.

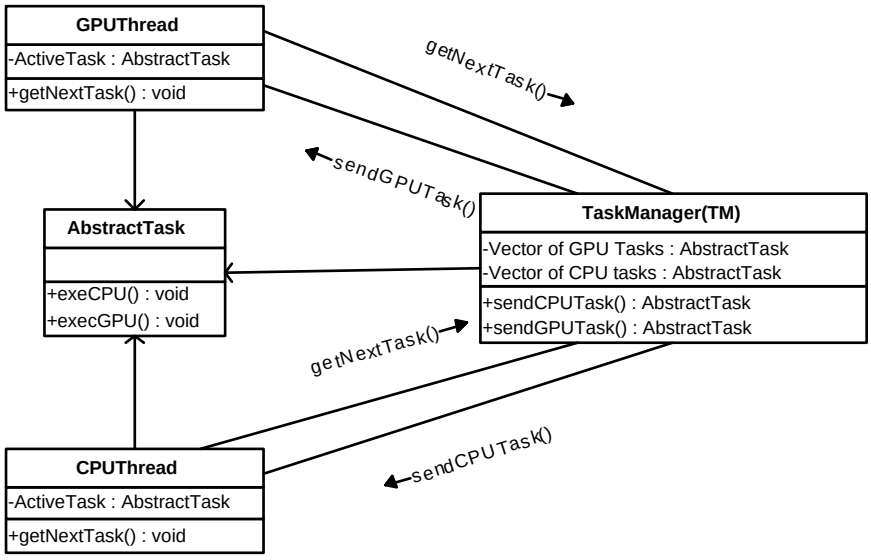

Figura 4.3: Troca de mensagens entre *threads* e o gerenciador de tarefas.

[1] A aplicação contém apenas uma instância da classe.

A comunicação durante o ciclo de vida da aplicação é ilustrada na figura 4.4, onde, no tempo $t0$, a aplicação inicializa as *threads* do TM e a partir do tempo $t1$ cada *thread* funciona de forma assíncrona, requisitando uma tarefa e o TM retornando a própria tarefa.

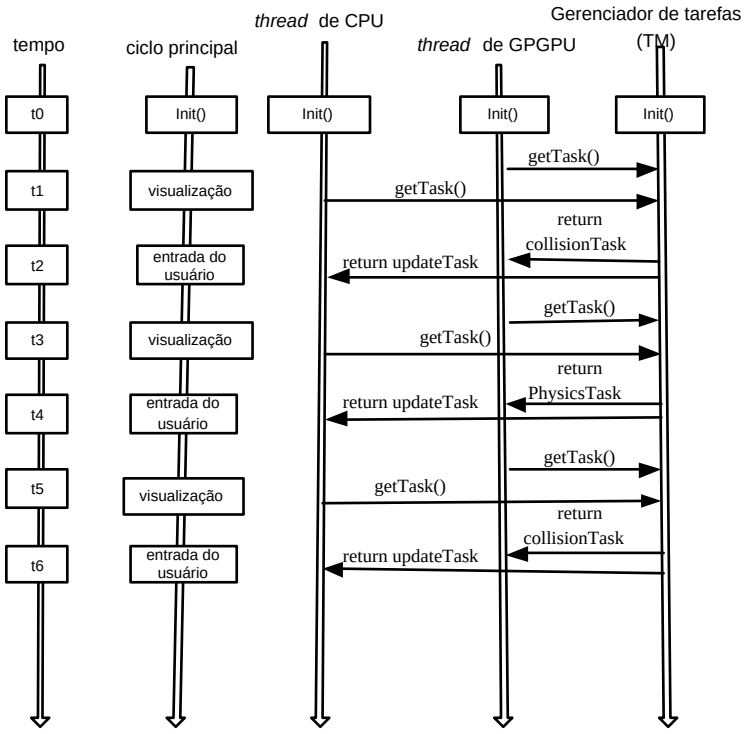

Figura 4.4: Ciclo de vida da aplicação e troca de mensagens entre *threads* e o gerenciador de tarefas.

4.3 Configuração do Gerenciador de Tarefas

O arquivo de configuração do gerenciador é chamado de *Task Configuration Script* (TCS), escrito em linguagem Lua [10] e composto por três partes. A primeira é responsável apenas pelas primeiras tarefas de cada processador, a segunda parte é composta por uma lista de tarefas, onde cada uma é associada a um dos processadores objetivando definir onde serão processadas e a terceira e última parte apresenta uma lista de tarefas contendo a configuração da tarefa subseqüente, definindo a ordem de processamento. A ordem de processamento contempla três distintos casos:

- A próxima tarefa é diferente da atual.
- A próxima tarefa é a mesma que a atual.
- Não há mais tarefas.

No primeiro caso, o TM retorna a tarefa subseqüente. No segundo caso, há apenas uma *tarefa* para o processador. No último caso, o TM retorna *NULL* para a *thread* que realizou a requisição, informando que não há mais tarefas.

A listagem ilustrada na figura 4.5(a), pode ser observada a ilustração da estrutura do *script* de configuração do TM, onde são definidas três diferentes tarefas chamadas de: *UPDATETASK*, *COLLISIONTASK* e *PHYSICSTASK*. As linhas 1 e 2 representam a primeira parte da configuração, indicando quais são a primeiras tarefas de CPU e da GPU, a segunda e terceira parte da configuração estão intercaladas nas linhas seguintes. Nas linhas 3, 5 e 7, são definidos os processadores responsáveis pela execução das tarefas e, nas linhas 4, 6 e 8, são definidas as tarefas subseqüentes.

```
01:CPU = " UPDATETASK "
02:GPU = " COLLISIONTASK "
03:UPDATETASK = " CPU"
04:UPDATETASKNEXT = " UPDATETASK "
05:COLLISIONTASK = " GPU"
06:COLLISIONTASKNEXT = " PHYSICSTASK "
07:PHYSICSTASK = " GPU"
08:PHYSICSTASKNEXT = " COLLISIONTASK "
```

(a)

```
01:CPU = " UPDATETASK "
02:UPDATETASK = " CPU"
03:UPDATETASKNEXT = " UPDATETASK "
04:COLLISIONTASK = " CPU"
05:COLLISIONTASKNEXT = " PHYSICSTASK "
06:PHYSICSTASK = " CPU"
07:PHYSICSTASKNEXT = " COLLISIONTASK "
```

(b)

Figura 4.5: *Script* Lua: (a) configuração correta (b) configuração com erro.

O TM não tem a função de alocar dinamicamente uma tarefa dentro da aplicação, mas, sim, de gerenciar as tarefas já existentes, alocando o processador que vai trabalhar a tarefa.

Durante o processo de configuração das tarefas, deve-se observar as seqüências. Uma configuração equivocada pode colocar as tarefas em uma estado de *deadlock* ou *starvation*. O problema surge quando há mais de uma tarefa para um mesmo processador e uma delas aponta para si como a próxima tarefa, produzindo um estado onde apenas uma tarefa fica sendo processada enquanto as demais ficam em espera.

É ilustrado na figura 4.5(b) um *script* com erro. A linha 1 define a primeira tarefa da CPU, as linhas 2, 4 e 6 definem que as respectivas tarefas serão processadas em CPU e as linhas 3, 5 e 7 definem as tarefas subseqüentes. O erro ocorre porque a linha 3 define como tarefa subseqüente a tarefa UPDATETASK como sendo ela própria e fazendo com que as demais nunca sejam processadas.

4.3.1 Processando Um Simulador de Particulas - Detalhe da Implementação

Para validar o modelo proposto, um estudo de caso foi implementado: simulação dinâmica de corpos rígidos sem restrições, onde todo o algoritmo foi implementado para ambos os processadores (GPU e CPU) modelado na forma de tarefa, permitindo a execução por ambos os processadores.

Antes da descrição do estudo de caso, é necessário compreender o que é um sistema de partículas. Sistema de partículas é composto por um conjunto de pontos que possuem atributos que descrevem sua aparência e comportamento no cenário virtual e no tempo, definido como tempo de vida. No estudo de caso, as partículas são um conjunto de pontos onde cada qual possui quatro atributos. Estes atributos descrevem o comportamento das partículas no cenário durante o tempo de vida. Assim, a cada instante t da animação, as partículas são atualizadas conforme seus atributos, que são:

- Posição da partícula no mundo virtual baseado em seu centro de massa;
- Um quatérnio que representa uma rotação aplicada sobre a partícula;
- O momento linear;
- E o momento angular.

É necessário atualizar estes atributos para o instante $t + 1$. A simulação consiste sempre em calcular e atualizar os atributos das partículas no instante $t + 1$ baseado nos valores destes mesmos atributos no instante t, portanto, a cada interação da simulação, estes atributos são mapeados e calculados para atualizar os atributos, modificando o estado das partículas na simulação. O processo de calculo é feito tanto na GPU quanto na CPU, conforme configuração realizada no *script*.

O estudo de caso consiste basicamente no movimento de partículas criadas por um emissor de partículas. Cada partícula tem a forma de uma esfera como representação da partícula, iniciando com raio 1 e densidade 10, não importando a medida, uma vez que no mundo virtual não há uma forma de medição. As partículas são geradas a partir de um ponto de origem e um intervalo de tempo fixo. A velocidade linear é aleatoriamente definida. A cada passo de tempo, uma força termodinâmica vertical é aplicada, fazendo com que as partículas subam. O raio da partícula inicia com valor um e cresce até o tamanho máximo três durante o ciclo de vida da partícula. Quando o ciclo de vida das

partículas é atingido, estas são destruídas e re-criadas. O critério que determina o tempo de vida é a um certo número de partículas simultâneas no cenário, no estudo de caso é um parâmetro, este tem seu valor definido dentro do *script* configuração Lua.

As partículas são modeladas em *pixels* de uma só textura e enviadas a GPU, onde são produzidos quatro *buffers* de saída, representando cada uma das variáveis que regem o comportamento da partícula em cena. Após a leitura dos *buffers*, o estado das partículas é atualizado. A figura 4.6 mostra um quadro da simulação.

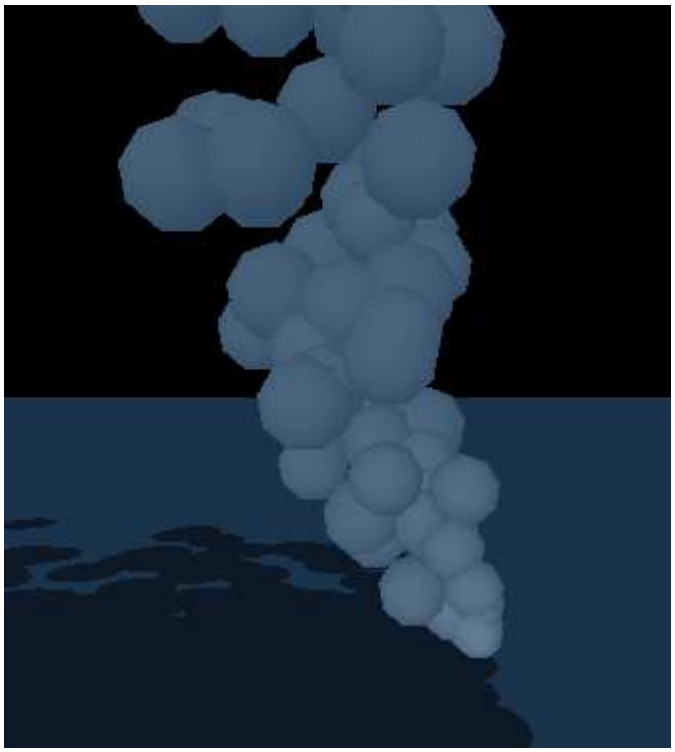

Figura 4.6: Um quadro da simulação.

4.3.2 Testes do Modelo - Modelo *Multithread* Desacoplado com Distribuição Dinâmica de Tarefas

Baseado no estudo de caso e no modelo proposto, dois grupos de testes foram realizados (simulação em CPU e GPU), onde os corpos rígidos variaram de 1 a 4000 e a cada 500 partículas foram feitas análises sobre quantidade de quadros por segundos e tempo de processamento em ambos os processadores, bem como a latência da memória da placa gráfica. Todos os tempos foram dados em segundos. Um campo definido como **ganho** foi incluído e consiste na divisão do tempo de CPU pelo tempo em GPU.

Tabela 4.1: Resultados experimentais da simulação.

	FPS			tempo de GPU			
corpos	CPU	GPU	tempo de CPU	memória	processamento	total	ganho
500	135,00	107,56	0,00541	0,00258	0,01182	0,01441	0,37531
1000	78,59	67,54	0,01347	0,00328	0,01295	0,01623	0,82990
1500	45,79	43,11	0,02225	0,01036	0,01319	0,02355	0,94505
2000	32,77	31,23	0,03047	0,02039	0,01214	0,03253	0,93661
2500	25,75	24,73	0,03995	0,02961	0,01175	0,04136	0,96581
3000	20,90	20,25	0,04760	0,03393	0,01219	0,04613	1,03182
3500	18,24	17,47	0,05652	0,04731	0,01153	0,05884	0,96067
4000	15,66	15,37	0,06522	0,05483	0,01174	0,06657	0,97978

4.4 Conclusão do Capítulo

A arquitetura apresenta um novo conceito de ciclo de jogo, onde a GPU é utilizada como co-processador matemático auxiliando a CPU. A construção dessa arquitetura envolve o desenvolvimento de classes para manipulação de shader e da GPU no processamento *off-screen* e um núcleo de gerência. Para validar a arquitetura, uma simulação de partículas foi construída e, através do *script*, foi possível definir o processador, conforme visto no exemplo.

O presente trabalho apresenta resultados de que modelar tarefas que necessitam de uma constante atualização não são ideais para serem tratadas em GPU, pois a latência no acesso a memória da placa gráfica ainda é um problema que deve ser tratado, ou por novas arquiteturas de hardware ou por otimização da codificação. Portanto, a arquitetura proposta pode ser vista como um novo padrão de modelo no desenvolvimento de jogos e simulações de tempo real.

O objetivo do capítulo foi atingido, o módulo de ciclo de jogo com estágio de GPGPU baseado no *script* de configuração foi desenvolvido, permitindo a a configuração de cada tarefa.

Capítulo 5

Alocação Dinâmica de Tarefas entre CPU e GPU

A alocação dinâmica de tarefas, utilizando técnicas de GPGPU, é um tema de grande relevância. Embora muitos trabalhos tenham sido desenvolvidos aplicando tais técnicas [40] e [3], poucos estão neste campo, sendo esta dissertação um dos primeiros.

A arquitetura discutida no capítulo anterior apresenta uma estrutura transparente na distribuição de carga de processamento entre CPU e GPU. A distribuição realizada por esta arquitetura é manual, implicando na configuração através de um arquivo. Com o objetivo de tornar esta arquitetura mais flexível e automatizar o processo decisório de distribuição de carga, este capítulo vem discutir o uso de um módulo inteligente na arquitetura de ciclo de jogo debatida anteriormente.

O funcionamento de um módulo inteligente no processo decisório de distribuição de carga esta dividido em três momentos:

- Identificação das variáveis utilizadas no processo decisório;
- Definição da técnica de inteligência artificial pode ser aplicada; e
- Mapeamento do resultado da aplicação da técnica em uma decisão.

O objetivo deste capítulo é acoplar, no modelo de ciclo de jogo apresentado no capítulo anterior, um módulo inteligente com a função de otimizar o uso da GPU no processamento das tarefas, automatizando o processo desenvolvido, de forma manual, no modelo do capítulo anterior. O capítulo está organizado da seguinte forma: a seção 5.1 discute sobre as variáveis envolvidas no processo decisório de uma tarefa. Aspectos sobre o módulo de inteligência artificial e a integração com a linguagem Lua são abordados nas seções 5.2,

5.3 e 5.4. O processo classificatório das tarefas relacionado ao processador é debatido na seção 5.5 e, na seção 5.6, é apresentada a conclusão do capítulo.

5.1 Variáveis Aplicadas ao Processo Decisório

As variáveis necessárias ao processo decisório estão relacionadas a um conjunto de fatores do hardware tais como: capacidade e taxa de utilização dos processadores (CPU e GPU), tempo de processamento, *frames* por segundos (FPS), barramento da placa gráfica e da memória principal, taxa de transmissão do disco rígido, tipo da tarefa e quantidade de processos. Cada um destes itens é uma variável que influencia no desempenho de uma aplicação em tempo real e que pode ser usada como dado de entrada no módulo de inteligência.

Dentre variáveis apresentadas, as utilizadas nesta pesquisa foram os tempos de processamento de CPU e GPU, isto é, o tempo gasto por uma tarefa para ser executada pelo processador, e os quadros por segundo (FPS) demandados pela GPU. A escolha destas medidas de desempenho está baseada nos seguintes aspectos: custo computacional em obter o resultado da medida e complexidade para desenvolver o algoritmo de obtenção da mesma.

O custo computacional na obtenção do resultado está relacionado com a execução do próprio algoritmo, enquanto a complexidade para desenvolvê-lo está relacionada com a construção do algoritmo e a utilização de APIs para acessar a informação desejada. Eventualmente, tais informações deverão ser obtidas por meio de linguagem de máquina.

5.2 O Gerenciador de Tarefas e o Módulo de Inteligência Artificial

Esta arquitetura adota um módulo inteligente que trabalha em conjunto com o gerenciador de tarefas (*TaskManager* - TM), discutido na seção 4.2. Para garantir a flexibilidade do módulo de inteligência (ModIA), este foi desenvolvido como classe abstrata e o TM possui um ponteiro para esta classe, passando a consultar o ModIA em vez do *Task Configuration Script*(TCS), conforme pode ser observado no diagrama de classes 5.1.

O desenvolvimento da classe abstrata tem por objetivo permitir a implementação de outras técnicas e outras medidas de desempenho. Está arquitetura adotada pelo módulo de inteligência artificial é composta por duas camadas: a primeira no nível de aplicação e

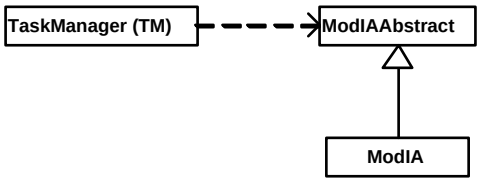

Figura 5.1: Diagrama da classe do ModIA e *TaskManager*.

a segunda camada é em nível de *script*. Na primeira, se encontra o ModIA e, na segunda, está o *script* em linguagem Lua, conforme é discutido neste capítulo. A figura 5.2 ilustra esta arquitetura.

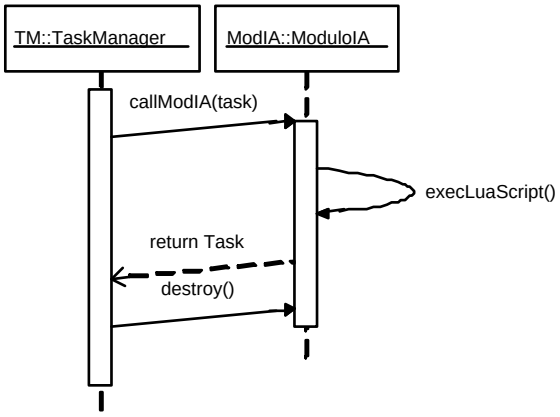

Figura 5.2: Digrama de seqüência do gerenciador de tarefas e o ModIA.

Com a inicialização da aplicação, o TM executa cada tarefa uma única vez na CPU e em seguida na GPU, gravando dentro da própria tarefa as informações necessárias para o ModIA. Tem o objetivo de alimentar uma base de dados de conhecimento.

Após a construção da base de conhecimento, o TM realiza uma comunicação com o ModIA através de um protocolo próprio. A comunicação ocorre em duas etapas bem definidas: na primeira o TM envia para ModIA cada tarefa e, na segunda etapa, a tarefa é devolvida pelo ModIA ao TM, com a informação para qual processador deve ser enviada, permanecendo esta informação gravada na própria tarefa.

As tarefas exclusivas da CPU contem uma informação indicando este *status* para evitar que o gerenciador de tarefas as envie para serem processadas na GPU, pois há tarefas que não são tratadas pelo modelo de programação da GPU. Nestes casos, são executadas apenas pela CPU. Como exemplo, há a tarefa de leitura de uma textura em disco.

5.3 Módulo de Inteligência Artificial Associado à Linguagem Lua

ModIA trabalha integrado com a linguagem Lua e todo o processamento de inteligência artificial do módulo é executado por um *script* em linguagem Lua. Durante o processo de inicialização do TM, este informa ao ModIA qual é o arquivo com o *script* Lua. Ao receber uma tarefa, o ModIA, executa o *script* Lua, passando-lhe as variáveis adotadas como base de conhecimento. Ao final, obtém-se do *script* Lua a informação de qual processador deve executar a tarefa.

A escolha por desenvolver todo o procedimento decisório dentro de um *script* tem como principal objetivo a maior flexibilidade, permitindo aplicar outras técnicas de forma simples sobre a ótica de engenharia de software, como o uso de heurísticas ou técnicas de inteligência artificial.

A chamada feita pelo ModIA ao *script* Lua é fixa, portanto, cada classe que estende a ModIA deve conter o nome da função associada na linguagem Lua e esta função deve esperar as variáveis adotadas como base de conhecimento. Como exemplo, temos o FPS e o tempo gasto, retornando o processador (CPU e GPU). A adoção de qualquer uma das medidas, como discutido na seção 5.1, requer o desenvolvimento de módulos de inteligência artificial e sua associação com uma função dentro de um *script* em Lua. Esta é uma limitação da arquitetura.

O uso de quaisquer heurísticas ou técnicas de inteligência artificial implica em um custo computacional que influencia diretamente o comportamento do ciclo de vida da aplicação em tempo real. Por outro lado, é necessário que a aplicação adquira informações para garantir um processo decisório eficiente.

A execução do *script* Lua possui um custo computacional. Este custo deve ser considerado no desenvolvimento da técnica de inteligência artificial, portanto, não é interessante utilizar técnicas complexas ou que levem um alto tempo de processamento, pois tais técnicas acabariam por comprometer a dinâmica do jogo digital ou da aplicação de simulação em tempo real.

5.4 O Módulo de IA

Como estudo de caso e para mostrar o desempenho da arquitetura do módulo de inteligência, foi implementado um módulo utilizando a técnica de lógica *fuzzy*. A técnica foi

aplicada sob um aspecto simples em relação à própria lógica *fuzzy*, com o objetivo de não comprometer o desempenho da aplicação de tempo real.

5.4.1 Aspectos da Lógica *Fuzzy*

A lógica *fuzzy* é a ciência que trata os princípios formais do raciocínio aproximado e impreciso. Modela conceitos imprecisos do raciocínio humano, usados na tomada de decisão, em um formato de lógica formal capaz de ser aplicada por computadores.

A decisão equivocada na alocação de uma tarefa pode ocasionar um efeito indesejado no comportamento da aplicação de tempo real e também pode gerar uma subutilização um hardware disponível, como no caso das placas gráficas atuais.

Fazendo um paralelo com o mundo de negócios, decisões são situações onde profissionais devem escolher um caminho a fim de solucionar um problema. Na aplicação em tempo real, é possível afirmar que as tarefas, o gerenciador de tarefas e o seu processamento são comparados ao funcionamento de uma empresa, onde a própria empresa é a aplicação em tempo real, o trabalho realizado é a execução da tarefa, os clientes são os usuários e os funcionários são os processadores. Assim, o gerente (gerenciador de tarefas), conhecendo a capacidade de cada funcionário (CPU ou GPU), distribui, da forma mais eficiente, trabalhos para cada funcionário, maximizando todo o processo e melhor atendendo o cliente (usuário). Portanto, o objetivo de inserir um módulo de lógica *fuzzy* dentro do gerenciador de tarefas é permitir que a aplicação maximize o uso do hardware para um melhor desempenho.

5.4.2 Módulo de IA com Lógica *Fuzzy*

A adoção de um módulo de lógica *fuzzy* dentro da arquitetura discutida no capítulo anterior é interessante, pois permite ao modelo aprender com a execução da própria aplicação em tempo real, adotando conceitos imprecisos ou nebulosos, tanto para quadros por segundos, como para tempo de processamento. Isto permite que o gerenciador adote, também, conceitos imprecisos, incertos ou nebulosos em vez de conceitos binários ou ainda por configuração do usuário.

O módulo *fuzzy* tem como objetivo receber variáveis de entrada usadas como medidas na aplicação de tempo real, quadros por segundos (FPS) e tempo de processamento (tempo gasto). Após a entrada destas variáveis, é realizado o processo de inferência e produzido um conjunto de resultados que vão servir de base no processo decisório.

No módulo do gerenciador, as variáveis FPS e tempo gasto são submetidas a funções de pertinência, uma para cada variável, onde são classificadas em cinco faixas ("muito baixo", "baixo, "normal", "alto"e "muito alto"). Cada qual é chamada de variável lingüística e é definida por uma quádrupla $(X, \Omega, T(X), M)$, onde X é a variável, Ω o universo do discurso, $T(X)$ é o conjunto de nomes que classificam a variável X e M são funções que associam uma função de pertinência a cada elemento de $T(X)$.

As figuras 5.3 e 5.4 ilustram as duas variáveis lingüísticas (FPS e tempo gasto) nos termos nebulosos.

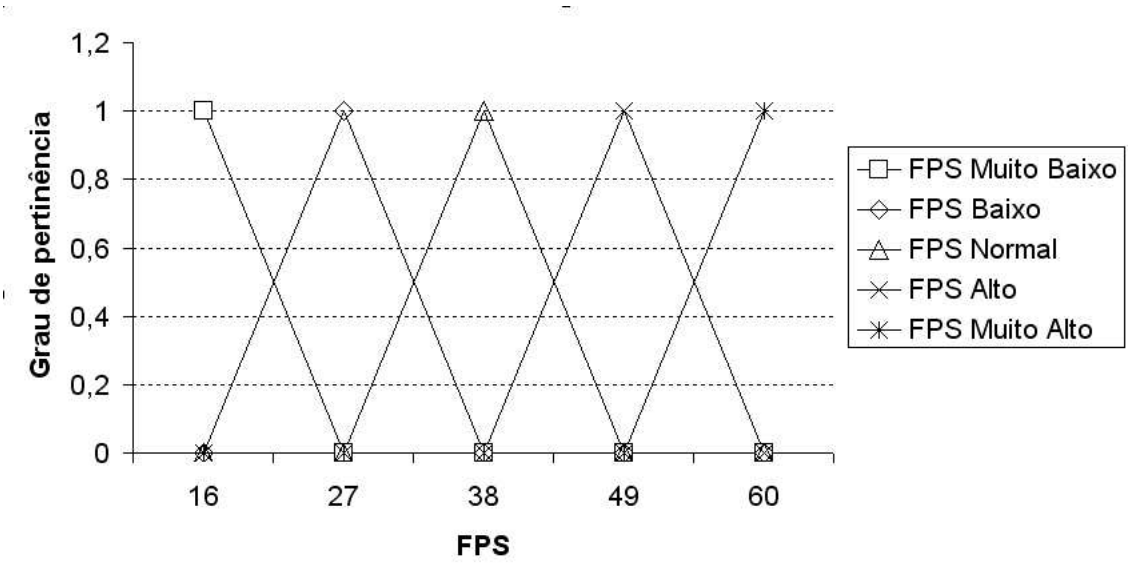

Figura 5.3: Termos lingüísticos que mapeiam a variável FPS.

O grau de pertinência de um dado elemento X_i do universo do discurso é dado por $\mu_\alpha(X_i)$ e representa o quanto este elemento está inserido no conjunto *fuzzy* α.

Os controladores nebulosos são robustos, sendo facilmente adaptáveis em sistemas heterogêneos, são adaptáveis em sistemas onde a modelagem é complexa e de difícil representação matemática e possuem grande utilidade em sistemas não-lineares, podendo inclusive ser aplicados em sistemas onde a incerteza se apresenta de forma intrínseca.

O grau com que um valor X qualquer, que representa uma das variáveis de entrada, em um conjunto *fuzzy* α, satisfaz o termo lingüístico A é chamado de pertinência de X^* em A, dada pelo tempo gasto e é sempre o inverso do FPS, portanto, é muito interessante para o gerenciador uma combinação mais próxima de FPS alto e tempo gasto baixo. No módulo *fuzzy* do gerenciador, a quantidade de variáveis linguisticas é estático, apenas sendo permitido definir, em um *script* de configuração, os limites de cada uma das variáveis relacionadas ao FPS da aplicação e do tempo gasto para cada tarefa.

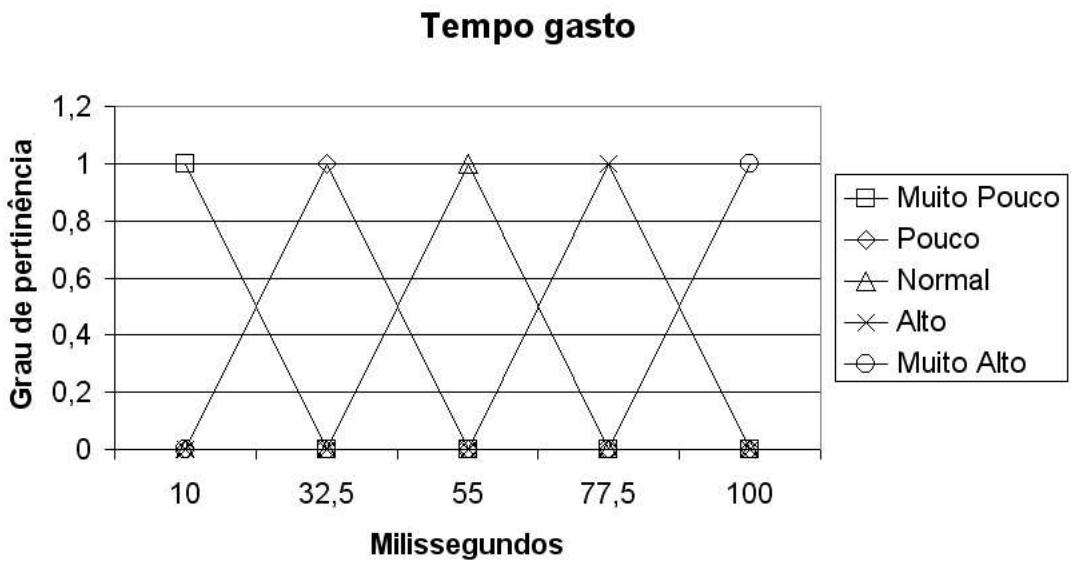

Figura 5.4: Termos lingüísticos que mapeiam a variável tempo gasto.

5.4.3 A Tarefa e o Módulo de Lógica *Fuzzy*

O Módulo de lógica *fuzzy* desenvolvido na presente dissertação, além de implementar a execução da função associada ao módulo de inteligência artificial, discutido na sessão 5.3, também exige parâmetros de configuração.*Script* de configuração do módulo *fuzzy* é similar a versão anterior apresentada no capítulo anterior, porém este possui uma função associada ao módulo de IA da aplicação e todas as configurações e algoritmos são chamados de dentro desta função.

O *script*, baseado na linguagem Lua, é composto por três partes distintas. A primeira trata a configuração dos limites globais, isto é, os valores mínimo e máximo do FPS. A segunda parte é composta pela definição dos limites de cada tarefa. A falta de qualquer um dos parâmetros implica em um valor padrão atribuído pelo próprio *script*. A terceira parte são as funções que implementam toda a logica *fuzzy* utilizada pelo módulo e chamadas pela função associada ao módulo de IA.

O FPS pode ter seus limites atribuídos entre 16 FPS como mínimo e, como limite máximo, a freqüência do próprio monitor. Não é recomendável configurar valores acima da freqüência do monitor, pois, nesses casos, os quadros produzidos repetidos são descartados, desperdiçando o poder computacional da placa gráfica dentro do ciclo de jogo.

É observada, na figura 5.5, a listagem que exemplifica as configurações de duas tarefas chamadas de *SMOKETASK* e *UPDATETASK*. Nas linhas 1 e 2, são definidos os limites máximo e mínimo do FPS. As linhas 3 e 4 definem os limites máximo e mínimo da primeira

```
01:FPSMIN =  16
02:FPSMAX =  60
03:SOMKETASKMIN =  0.1000
04:SOMKETASKMAX =  0.3000
05:COLLISIONMIN =  0.1000
06:COLLISIONMAX =  0.2000
```

Figura 5.5: *Script* de configuração do módulo *fuzzy*.

tarefa. As linhas seguintes definem os limites das demais tarefas.

A presente versão do TM com módulo de lógica *fuzzy* não configura que tarefa vai ser alocada em um processador, mas passa para o ModIA a função de definir qual processador vai tratar a tarefa através da execução da função associada no *script* Lua.

Uma configuração equivocada ocasiona um comprometimento do processo decisório e gera execuções não tão otimizadas. Entretanto, o funcionamento da aplicação não é comprometido por problemas como *deadlock* ou *starvation*.

5.4.4 O Modelo Adotado

O modelo de inferência *fuzzy* adotado foi baseado no Modelo de Mamdami [51]. A fundamentação da escolha está baseada no fato de que este modelo possui uma relação *fuzzy* tanto no antecedente quanto no conseqüente, modelando perfeitamente o problema de decisão baseado nas variáveis FPS e tempo gasto, que são os antecedentes. Estas variáveis podem assumir um dos cinco valores: "muito baixo", "baixo", "normal", "rápido"e "muito rápido"e o conseqüente pode assumir valores imprecisos, podendo ter a saída classificada em cinco valores: "muito fácil", "fácil", "normal", "difícil"e "muito difícil".

Este modelo é composto por três fases: a primeira trata da conversão das variáveis de entrada, que são valores numéricos, num conjunto *fuzzy*, a segunda fase é aplicada à máquina de inferência e a terceira fase trata da conversão do conjunto *fuzzy* de saída em um resultado objetivo. A figura 5.6 ilustra o modelo de inferência adotado. A característica da regra semântica utilizada no processo de inferência é conhecida como máximo e mínimo, onde máximo são as operações de união e mínimo são as operações de interseção.

5.4.4.1 Processo de Generalização - *Fuzzification*

O processo de generalização, também chamado por *fuzzification*, é o processo onde as variáveis numéricas de entrada do processo decisório são transformadas em variáveis qualitativas. Cada variável numérica de entrada (FPS e tempo gasto) tem um grau de perti-

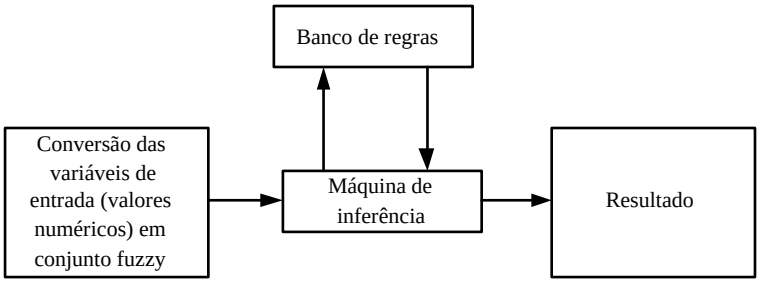

Figura 5.6: Modelo de diagrama de inferência.

nência em um dos cinco conjuntos ("muito baixo", "baixo", "normal", "rápido"e "muito rápido"). Cada variável é classificada no conjunto em que obteve o maior grau de pertinência, obtendo o maior grau de pertinência de FPS($\mu(FPS)$) e tempo gasto($\mu(TG)$). Em seguida, é processada a interseção ou operação de mínimo $min[\mu(FPS), \mu(TG)]$, obtendo-se o resultado a partir da consulta na tabela 5.1 e finalizando esta etapa.

Os cinco conjuntos, "muito baixo", "baixo", "normal", "rápido"e "muito rápido", são construídos para cada tarefa, onde são informados apenas os valores mínimos e máximos de FPS e tempo gasto. Tais valores são referentes aos conjuntos "muito baixo"e "muito rápido". Uma vez definidos os limites máximos e mínimos das variáveis, os valores intermediários são calculados pela execução da função associada ao módulo de IA a partir do ponto médio. Por exemplo, para o FPS, definindo seus limites entre 16 e 60, os valores intermediários são calculados em 27 para FPS "baixo"; 38 para FPS "normal"e 49 para FPS "alto". A tabela 5.1 é construída baseado no conjunto de variáveis lingüísticas.

Tabela 5.1: Tabela de variáveis lingüísticas.

FPS/tempo gasto	muito pouco	pouco	normal	alto	muito alto
muito baixo	fácil	normal	difícil	muito difícil	muito difícil
baixo	fácil	normal	difícil	muito difícil	muito difícil
normal	normal	fácil	normal	difícil	muito difícil
alto	muito fácil	fácil	normal	difícil	difícil
muito alto	muito fácil	fácil	normal	difícil	difícil

5.4.4.2 Base de Conhecimento.

O banco de conhecimento é composto por um conjunto de regras armazenadas na forma **Se** <*antecedente*> **Então** <*conseqüente*>. Nestas regras, o conseqüente também é uma variável qualitativa. Diferentemente da fase anterior, esta é comum a todas as tarefas.

O conjunto de regras deve contemplar todas premissas geradas na fase anterior (processo de generalização). As premissas são consultadas na base de conhecimento com o

objetivo de produzir um conseqüente, onde todas as regras serão aplicadas. Não são admitidos valores de entrada que não sejam tratados por uma regra. O controlador nebuloso possui tantas regras quantas forem necessárias para mapear totalmente as combinações dos termos das variáveis, ou seja, a base está completa quando todas as opções são contempladas, garantindo que sempre haja ao menos uma regra para qualquer entrada.

5.4.4.3 *Desfuzzification*

Esta fase é responsável pela conversão da variável qualitativa em quantitativa dentro do modelo de Mamdami. A partir de um conjunto *fuzzy* de saída obtido pelo processo de inferência, chega-se um valor numérico. Há diversas técnicas utilizadas neste ponto, tal como a técnica do centro de massa. O modelo adotado no presente projeto aplicou uma pequena modificação neste estágio, uma vez que o resultado final consiste em apenas dois valores: CPU ou GPU. Como esta informação é obtida na fase anterior, esta fase foi suprimida do modelo, simplificando todo o processo.

5.5 GPU, CPU e o Processo Classificatório

Em linhas gerais, a execução do processo utilizando o ModIA com linquagem Lua no processo de distribuição de carga ocorre conforme as seguintes etapas. As informações sobre uma tarefa durante o processamento da GPU e CPU são submetidos ao módulo *fuzzy* do gerenciador. Ao final, cada tarefa recebe uma classificação de saída para cada processador. Após a classificação de saída baseada na tabela 5.1, são produzidos dois conjuntos de regras, uma para cada processador. Em seguida, a tabela 5.2 é aplicada e o processador é definido. As regras possuem dois antecedentes, um para cada processador, e um conseqüente, como exemplificado:

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

As tarefas que são executadas exclusivamente pela CPU possuem um atributo. Este é verificado pelo TM, que impede que a tarefa não seja submetida do processo decisório e não seja executada pela GPU.

Tabela 5.2: Base de conhecimento.

Classificação de saída CPU	Classificação de saída GPU	Processador
muito fácil	muito fácil	GPU
muito fácil	fácil	GPU
muito fácil	normal	GPU
muito fácil	difícil	CPU
muito fácil	muito difícil	CPU
fácil	muito fácil	GPU
fácil	fácil	GPU
fácil	normal	GPU
fácil	difícil	CPU
fácil	muito difícil	CPU
normal	muito fácil	GPU
normal	fácil	GPU
normal	normal	GPU
normal	difícil	CPU
normal	muito difícil	CPU
difícil	muito fácil	GPU
difícil	fácil	GPU
difícil	normal	GPU
difícil	difícil	CPU
difícil	muito difícil	CPU
muito difícil	muito fácil	GPU
muito difícil	fácil	GPU
muito difícil	normal	GPU
muito difícil	difícil	CPU
muito difícil	muito difícil	CPU

5.6 Conclusão do Capítulo

O capítulo apresenta uma solução de IA no módulo de gerência, proporcionando a este uma autonomia. Certamente outras evoluções devem ser observadas na arquitetura do ciclo de um jogo, em especial na parte de IA do módulo de gerência. O uso de lógica *fuzzy* é uma decisão interessante, trazendo para dentro da aplicação a incerteza e a nebulosidade existente no mundo real, o que se mostra bastante aplicável em razão do modelo heterogêneo de arquitetura implementada em PCs, dados os diferentes modelos de placas, memórias, discos rígidos e placas gráficas disponíveis. Analisar apenas que uma tarefa foi lenta quando processada pela GPU ou pela CPU não é informação suficiente para o processo decisório, sendo necessário a análise do comportamento da tarefa por ambos os processadores. Apesar desta ser uma estrutura automatizada de decisão, ainda há a possibilidade de uma interferência do desenvolvedor da aplicação no processo decisório. O uso do *script* permite que haja interferência do usuário no processo decisório. O modelo de

lógica *fuzzy* aplicado foi simples, em razão da própria complexidade da técnica de IA, não sendo recomendável o uso de técnicas que demandam um grande esforço computacional, como algoritmos genéticos, busca tabu, *grasp* ou mesmo redes neurais que requerem um alto custo computacional no seu treinamento. A arquitetura apresentada é eficiente, pois viabiliza o uso de técnicas de IA ou de heurísticas para otimizar a distribuição de tarefas separada da aplicação através do *script* Lua.

O objetivo deste capítulo foi atingindo, um módulo inteligente foi desenvolvido e acoplado no modelo de ciclo de jogo digital. Através deste módulo, pode-se aplicar técnicas de inteligência artificial e de heurísticas, bastando apenas adapta-las à linguagem Lua.

Capítulo 6

Conclusão

A dissertação apresenta uma arquitetura de ciclo de jogo utilizando técnicas de GPGPU com alocação de tarefas. O uso destas técnicas demandam de um conhecimento do paradigma de programação em fluxo.

Este trabalho apresenta fundamentos da programação em GPU, cada vez mais presentes na área de pesquisa de computação visual, e seu relacionamento com o modelo tradicional, o modelo de programação utilizado por CPUs, apresentando em detalhes a modelagem necessária para uma arquitetura SIMD.

Há trabalhos que utilizam técnicas de GPGPU para problemas matemáticos e simulações físicas, já citados nesta dissertação, entretando, poucos trabalhos mostram técnicas que permitam, de forma uniforme, que arquiteturas distintas executem a mesma tarefa e ainda que o comportamento da aplicação seja diferente para diferentes conjuntos de arquitetura de hardware. Em outras palavras, um modelo capaz de adaptar-se a qualquer tipo de hardware, maximizando o uso da CPU e GPU.

Baseado neste conceito, foi construído um modelo de arquitetura inteligente capaz de alocar dinamicamente uma tarefa em um dos processadores disponíveis. A arquitetura foi viabilizada aplicando o conceito de tarefa "*processador processa tarefa*".

Em testes desenvolvidos, foi concluído que não é qualquer tarefa de física ou matemática que pode ser tratada pela GPU. Em alguns casos, a GPU apresenta uma performance semelhante a da CPU, devido a latência no acesso à memória da placa gráfica, fato que deve ser minimizado por novas séries de placas gráficas e pela evolução do próprio barramento. Em linhas gerais, o uso da GPU é sempre útil, pois, mesmo com a latência, permite que outras tarefas sejam processadas pela CPU.

O trabalho mostra também o compartilhamento deste recurso entre a tarefa de visu-

alização e propósito geral sem degradação ou qualquer outro problema. A desvantagem é a falta de uma ferramenta com a função de construir tarefas com as respectivas codificações (CPU e GPU), de forma automática, a partir da identificação de códigos escritos apenas para CPU, como códigos escritos na linguagem C/C++, isto é, a falta de um "pré-compilador"de tarefas.

O presente trabalho apresenta uma arquitetura de ciclo de jogo digital com distribuição dinâmica de tarefas usando a GPU. Esta arquitetura é composta de dois níveis de desenvolvimento. No primeiro nível, o desenvolvedor não necessita ter qualquer conhecimento de programação em linguagem shader, é necessário apenas ter conhecimento das tarefas, disponibilizadas na forma de bibliotecas, para configurar as mesmas no *script* de configuração. O segundo nível exige um conhecimento da linguagem shader e da estrutura da tarefa e, neste nível, as tarefas são construídas pelo desenvolvedor.

O objetivo deste trabalho é mostrar uma arquitetura de ciclo de jogo com uso de técnicas de GPGPU com distribuição de tarefas e este objetivo foi alcançado, disponibilizando bibliotecas de tarefas e estrutura para desenvolvimento de outras tarefas. Para alcançar este objetivo, foi necessário construir um *toolkit* de *shader* para utilizá-lo na forma de recurso, o que foi atingido, sendo também uma das contribuições desta dissertação.

6.1 Dificuldades

A arquitetura de ciclo de jogo com distribuição dinâmica de tarefas entre CPU e GPU é uma estrutura complexa que envolve outras áreas da ciência da computação e que apresenta alguns pontos de dificuldades. A primeira barreira transposta foi a falta de documentação e poucos trabalhos sobre o tema, sendo necessário buscar pesquisas desenvolvidas em outras linhas para aprender sobre como manipular a GPU na resolução de problemas genéricos.

Desenvolver uma estrutura eficiente de trabalho utilizando múltiplos programas shaders e usando concomitantemente tarefas de visualização e técnicas de processamento *off-screen* utilizando a GPU foi um desafio, onde foram desenvolvidas técnicas para permitir o funcionamento simultâneo do processamento genérico da GPU com o uso deste recurso nas tarefas de visualização.

Outra dificuldade foi permitir que uma mesma tarefa pudesse ser processada por ambos os processadores, buscando auxílio de outras áreas, como engenharia de software, para adotar uma solução que permitisse tratar a mesma tarefa por duas formas de processa-

mento diferentes.

O aspecto mais difícil do trabalho foi desenvolver um mecanismo inteligente capaz de realizar a alocação das tarefas sem a interferência do usuário, uma vez que a análise dos dados necessários à decisão pode, eventualmente, comprometer a execução da aplicação.

6.2 Trabalhos futuros

Uma proposta de trabalho futuro é desenvolver um mecanismo de decisão mais eficiente, capaz de gerar decisões sem comprometimento da aplicação.

Outros temas podem ser abordados como o desenvolvimento de *frameworks* capazes de identificar pontos em códigos e gerar automaticamente códigos similares em linguagem shader, transformando tais pontos em tarefas e liberando o desenvolvedor de conhecer a programação em GPU, um "pré-compilador"de tarefas.

Por último, propõem-se levar à GPU a execução do gerenciador de tarefas, propiciando um paralelismo no processamento.

Referências

[1] DOOM3. *DOOM 3*. Disponível em: `http://www.doom3.com/`. 30/06/2007.

[2] NINTENDO. *Wii Nintendo console*. Disponível em: `http://wii.nintendo.com`. 30/06/2007.

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

[4] ZELLER, C. Cloth simulation on the GPU. In: *ACM SIGGRAPH 05: ACM SIGGRAPH 2005 Sketches*. [S.l.: s.n.], 2005.

[5] MELO, R. H. C. de; VIEIRA, E. de A.; CONCI, A. Using particle systems to modulate celebrations with fireworks. *12th International Conference on Computer Graphics and Geometry*, 2006.

[6] BOLZ, J. et al. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics*, v. 22, n. 3, p. 917–924, 2003.

[7] GALOPPO, N. et al. Lu-gpu: efficient algorithms for solving dense linear systems on graphics hardware. In: *Proceedings of the ACM/IEEE Conference on Supercomputing*. [S.l.: s.n.], 2005. p. 3–14.

[8] OWENS, J. D. et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, v. 26, 2007. To appear.

[9] KRüGER, J.; WESTERMANN, R. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, v. 22, n. 3, p. 908–916, 2003.

[10] LUA. *The programming language Lua*. Disponível em: `http://www.lua.org/manual/`. 30/06/2007.

[11] SILBERSHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating System Concepts*. [S.l.]: John Wiley & Sons. Inc, 2005.

[12] GERALD, A. S. The story engine concept in cs education. In: *CCSC: Rocky Mountain Conference 2004*. [S.l.: s.n.], 2004.

[13] SáNCHEZ, D.; DALMAU, C. *Core Techniques and Algorithms in Game Programming*. [S.l.]: New Riders Publishing, 2006.

[14] LEWIS, M.; JACOBSON, J. Game engine in scientific research. *Communications of the ACM*, v. 45, n. 1, p. 27–31, 2003.

[15] UNREAL. *Epic Games, Inc. Unreal Technology*. Disponível em: `http://www.unrealtechnology.com`. 30/06/2007.

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

[17] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object Oriented Software.* [S.l.]: Addison-Wesley, 1995.

[18] SGI. *Standard Template Library Programmer's Guide.* Disponível em: http://www.sgi.com/tech/stl/stl_introduction.html. 30/06/2007.

[19] BOOST. *Boost documentation.* Disponível em: http://www.boost.org/libs/libraries.htm. 30/06/2007.

[20] JOHNSON, E. R.; MCCONNELL, C.; LAKE, M. J. *The RTL system: A framework for code optimization. In Robert Giegerich and Susan L. Graham, editors.* 255-274 p. 30/06/2007.

[21] MICROSOFT. *Windows documentation.* Disponível em: http://msdn2.microsoft.com/en-us/xna/default.aspx. 30/06/2007.

[22] SONY. *Playstation console.* Disponível em: http://www.playstation.com. 30/06/2007.

[23] MICROSOFT. *XBox 360 console.* Disponível em: http://www.xbox.com/pt-BR. 30/06/2007.

[24] DICKINSON, P. *Instant Replay: Building a Game Engine with Reproducible Behavior.* Disponível em: http://www.gamasutra.com/features/20010713/dickinson_01.htm. 30/06/2007.

[25] WATTE, J. *Canonical Game Loop.* Disponível em: http://www.mindcontrol.org/ hplus/graphics/game_loop.html. 30/06/2007.

[26] HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach.* Third. [S.l.]: Morgan Kaufmann, 2003.

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

[28] VALENTE, L.; CONCI, A.; FEIJó, B. Real time game loop models for single-player computer games. In: *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment.* [S.l.: s.n.], 2005. p. 89–99.

[29] LAMOTHE, A. *Tricks of the Windows Game Programming Gurus Fundamentals of 2D and 3D Game Programming.* [S.l.]: Sams, 1999.

[30] OPENGL. *OpenGL The Industry's Foundation for High Performance Graphics.* Disponível em: http://www.opengl.org/. 30/06/2007.

[31] NVIDIA. *NVIDIA.* Disponível em: http://developer.nvidia.com/object/sdk_home.html/. 30/06/2007.

[32] NVIDIA. *User's Manual - A Developer's Guide to Programmable Graphics.* [S.l.]: NVIDIA Corporation, 2006.

[33] 3DLABS. *3Dlabs Developer Relations*. Disponível em: http://developer.3dlabs.com. 30/06/2007.

[34] MICROSOFT. *Windows documentation*. Disponível em: http://msdn2.microsoft.com/pt-br/xna/aa937788.aspx. 30/06/2007.

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

[36] ATI. *RenderMonkey Toolsuite*. Disponível em: http://ati.amd.com/developer/rendermonkey/index.html. 20/05/2007.

[37] NVIDIA. *NVIDIA*. Disponível em: http://developer.nvidia.com/page/opengl.html/. 30/06/2007.

[38] KOLB, A.; CUNTZ, N. *Dynamic Particle Coupling for GPU-based Fluid Simulation*. Disponível em: http://www.gpgpu.org/cgi-bin/blosxom.cgi/Scientific%20Computing/Dynamics%20Simulation/index.html. 30/06/2007.

[39] NVIDIA. *GeForce 8800 GPU architecture overview. TB-02787-001_v0.9*. [S.l.], 2006.

[40] FERNANDO, R. *GPU Gems 2 - Programming Techniques for High-performance Graphics and General-Purpose Computation*. [S.l.]: Addison-Wesley, 2005.

[41] SDL. *SDL - Simple Directmedia Layer*. Disponível em: http://www.libsdl.org/intro.br/toc.html. 30/06/2007.

[42] LIB3DS. *lib3ds Homepage*. Disponível em: http://lib3ds.sourceforge.net. 30/06/2007.

[43] FTGL. *FTGL Homepage*. Disponível em: http://homepages.paradise.net.nz/henryj/code/. 30/06/2007.

[44] AUDIERE. *Audiere*. Disponível em: http://audiere.sourceforge.net. 30/06/2007.

[45] DEVIL. *DevIL - A full featured cross-platform image libary*. Disponível em: http://www.imagelib.org. 30/06/2007.

[46] TANENBAUM, A. S.; WOODHULL, A. S. *Sistemas Operacionais - Projeto e Implementação*. [S.l.]: Bookman, 2002.

[47] MICROSOFT. *Windows documentation*. Disponível em: http://msdn2.microsoft.com/en-us/library/ms682402.aspx. 30/06/2007.

[48] HAUSER, C. et al. Using threads in interactive system: A case study. *Proc. Of the Fourtennth Symp. On Operation System Principles ACM*, p. 94–103, 1993.

[49] GREEN, S. *NVIDIA cloth sample*. Disponível em: http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html#glsl_physics. 30/06/2007.

[50] FATAHALIAN, K.; SUGERMAN, J.; HANRAHAN, P. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In: *Graphics Hardware 2004*. [S.l.: s.n.], 2004. p. 133–138.

[51] REZENDE, S. O. *Sistemas Inteligentes - Fundamentos e Aplicações*. [S.l.]: Manole, 2003.