RENDERIZAÇÃO E INTERAÇÃO OTIMIZADA DE MULTIDÕES EM JOGOS UTILIZANDO LINGUAGEM CUDA

OPTIMIZED RENDERING AND INTERACTION OF CROWDS AT GAMES USING LANGUAGE CUDA

Tiago Ungaro Bardella[1], Ismar Frango Silveira[2]

Abstract *— The history of digital games shows games using many types of enemy models and types of characters to control, like Real-Time Strategy games, for example. These huge amount of models into an important scene are called crowds. The crowds needs a high computer performance and specific algorithms in their interaction control to avoid immersion loss. With the popularization of graphic board languages like NVIDIA CUDA, new algorithms were created to easily increase the performance of crowds in digital games and their overwhelming superiority compared to the methods used in linear programming were proved in many researches. The goal of this work is to use these GPU techniques as base to implement a framework using CUDA language that will present better performance and simplicity compared to the others algorithms on the area of crowds in digital games.*

Index Terms — CUDA, Digital Games, GPU, Virtual Crowds.

INTRODUÇÃO

Multidões no conceito de jogos são modelos que são colocados em grande quantidade em uma respectiva cena do *game* para atrapalhar ou ajudar o jogador. Devido à complexidade de modelar, renderizar e interagir com grandes quantidades de indivíduos (conhecido como *Crowd Rendering*) em uma mesma cena que geralmente apresentam comportamentos distintos e únicos, foram desenvolvidos alguns algoritmos para tratar deste problema e evitar que o nível de imersão do jogador seja perdido por conta da baixa latência e serrilhamento (*Flickering*) que multidões costumam ocasionar com frequência quando um algoritmo pouco eficiente gera estas imagens. Alguns dos algoritmos clássicos para este tratamento de multidões são o LOD (*level-of-detail*) e a IBR (*Image-based Rendering*).

A simulação de uma multidão também é um grande desafio para os pesquisadores na área. Não só é necessário desenvolver um algoritmo com boa performance e em tempo real de multidões para não atrapalhar a imersão do jogador, como também é necessário tratar cada agente de uma multidão individualmente e coletivamente quando a multidão está em conflito entre si, com o cenário ou com outra multidão. A simulação de multidões somada com a modelagem e renderização da mesma tornam as multidões um dos maiores obstáculos para o desenvolvimento de um *game*.

Com o avanço da computação gráfica (entre elas a área de multidões de modelos), tornou-se necessário criar *hardwares* que pudessem conseguir acompanhar a evolução dos *softwares* gráficos. Este *hardware* é conhecido como *Graphics Processing Unit*, ou GPU.

As GPUs proporcionaram com seus múltiplos *cores* a capacidade de proporcionar um processamento gráfico em paralelo, o que agilizava a renderização de modelos durante um jogo, conseguindo acompanhar o avanço do *software*.
O conceito de GPGPU surgiu há poucos anos e trata-se de utilizar o poder computacional de uma GPU não somente para o processamento gráfico, mas para o processamento de física e matemática que um jogo necessita. Retirando o excesso de tarefas da CPU e transferindo-as para a GPU pode se alcançar um bom nível de otimização que antes não era possível.

Outras tarefas como criptografia, tratamento de banco de dados e processamento paralelo de computação em nuvem (*Cloud Computing*) foram propostas com o tempo utilizando-se de GPU, mas até então a GPGPU era complicada devido o conceitode linguagem específica de placas gráficas e inacessível para leigos em programação de *software*.

Todavia, com a criação da linguagem CUDA da NVIDIA para programação de placas gráficas (GPUs), novos algoritmos para multidões foram desenvolvidos com o intuito de melhorar ainda mais a performance que o tratamento de multidões tinha com uma programação linear.

Utilizando o grande poder computacional proporcionado pelas GPUs da atualidade, os algoritmos tradicionais de multidões como LOD e IBR foram adaptados e outros novos algoritmos foram implementados no conceito de programação paralela com resultados satisfatórios.

Entretanto, muitos jogos ainda apresentam problemas com estes novos algoritmos e a implementação de novos algoritmos demanda tempo e experiência para os desenvolvedores de jogos.

[1] Tiago Ungaro Bardella, aluno de mestrado de Engenharia Elétrica e Computação da PPGEE na Universidade Presbiteriana Mackenzie, breadtub2@hotmail.com
[2] Ismar Frango Silveira, professor doutor da PPGEE na Universidade Presbiteriana Mackenzie, ismarfrango@gmail.com
DOI 10.14684/WCCA.8.2015.336-340
© 2015 COPEC

O objetivo deste projeto é pesquisar, implementar e testar diversas técnicas de programação paralela para GPU utilizando linguagem CUDA para otimizar modelagem, renderização e simulação de multidões com foco em jogos digitais para proporcionar ao jogador uma experiência mais imersiva e maior nível de entretenimento evitando que ocorra a lentidão característica em jogos quando vários modelos são renderizados e interagem simultaneamente em uma mesma cena. Além disso, com a criação deste novo *framework*, desenvolvedores de jogos terão seu trabalho facilitado por utilizarem as técnicas prontas para trabalhar com multidões que este *framework* disponibilizará.

Este objetivo pode ser considerado partindo da seguinte hipótese: "É possível desenvolver uma nova técnica adaptada em um *framework* utilizando o poder computacional de uma placa gráfica que otimize o melhor processamento de multidões em jogos digitais existente na atualidade".

TRABALHOS RELACIONADOS

A renderização e simulação de multidões são fundamentais para a credibilidade de diversas áreas da Realidade Virtual como educação, treinamento e entretenimento [9]. Todavia, é necessário um poder computacional muito alto para renderizar um grande número de indivíduos o que torna o problema desafiador [11].

A simulação de multidões pode ser considerada um caso especial de ambiente virtual onde cada integrante da multidão conta como um agente inteligente ou um grupo de indivíduos controlados por um usuário. Cada agente possui seu próprio nível de inteligência, objetivos e comportamento, podendo ser semelhantes ou não dos demais integrantes da multidão.

Tanto a renderização de multidões como a modelagem de comportamento individual são problemas com um alto custo que cresce exponencialmente baseado na quantidade de agentes que fazem parte da multidão. Portanto, é necessário um *design* escalável que permita suportar grandes quantidades de agentes para manter o realismo e com um processamento que não seja pesado [9]. A figura 1 abaixo mostra um exemplo de jogo atual e escalável:

FIGURA. 1
GRANDE MULTIDÃO PROCESSADA COM QUALIDADE EM TEMPO REAL.
FONTE: HTTP://IMG.SARAIVACONTEUDO.COM.BR/CLIPART/IMAGES/DEAD-RISING-3_CRÉDITOS_CAPCOM3.JPG

O jogo Dead Rising 3 da Capcom exemplifica como é possível apresentar variedade, boa renderização e interação de uma quantidade absurda de agentes em uma multidão sem diminuir a imersão do jogador.

Segundo a referência [11], um impostor é uma técnica baseada em imagens que exibe um único polígono. As coordenadas de textura do impostor mudam de acordo com a posição da câmera e a orientação do objeto original. Objetos animados também mudam as coordenadas de textura para esse polígono baseado no *frame* de animação atual [9]. Um impostor pode ser definido como uma imagem 2D produzida por uma renderização de um objeto 3D complexo. Esta técnica é clássica no problema de simulação e renderização de multidões como uma forma de otimizar as multidões [8].

Outras soluções clássicas para a solução do problema são a LOD (*Level-of-detail*) e a IBR (*Image-based Rendering*) [11].

A LOD renderiza personagens distantes com menor detalhamento e personagens próximos com maior nível de detalhes. Já a técnica IBR reduz as faces enviadas para renderizar.

Outra técnica também utilizada é o *Occlusion Culling*, que deixa de renderizar as partes que estão encobertas por outros indivíduos ou por objetos, diminuindo assim o processamento que seria gasto indevidamente nestas partes ocultas [10].

O desenvolvimento de GPUs programáveis tornou possível alcançar um alto nível de realismo em simulações virtuais e jogos. As duas maiores empresas de GPU desenvolveram suas próprias linguagens com esse fim (NVIDIA com CUDA e AMD com CAL, ambas baseadas em linguagem C), tornando o uso da GPU mais flexível e sem algumas das limitações tradicionais de linguagens de *shaders* [4].

Em CUDA, cada *thread* na GPU executa a mesma função do *kernel* e tem um identificador e uma memória local. São organizadas em blocos e podem sincronizar sua execução e compartilhar o mesmo espaço de memória. Um conjunto de blocos representam um *grid*, que pode ser unidimensional ou bidimensional. O *grid* em turnos pega

para si a memória global. Um *kernel* consiste do código que será executado pela GPU. Para cada chamada ao *kernel*, uma configuração contendo o número dos blocos em cada *grid*, o número de *threads* por bloco e, opcionalmente, a quantidade de memória compartilhada para ser alocada no fluxo associado ao *kernel* são necessárias [1].

Uma técnica de interações para evitar colisões entre multidões, mais recente, envolve planejamento de caminhos e algoritmo A* (conhecido como "A estrela"), ou seja, encontrar uma das soluções mais básicas em jogos: O melhor caminho dentre duas possibilidades.

No final da execução de cada *thread* de *pathfinding*, o algoritmo devolve os caminhos encontrados para cada agente junto com o custo cumulativo do caminho calculado por A*. Durante a implementação e desenvolvimento do algoritmo de navegação na GPU, foi possível identificar alguns gargalos e otimizações que aumentarão o ganho alcançado com implementação nas placas gráficas.

Diversos tipos de jogos, especificamente os *First Person Shooters* (FPS) e *Real Time Strategy* (RTS) são populados por atores sintéticos que devem agir como agentes autônomos. Estes agentes autônomos são chamados de *Non-player Characters* (NPCs) e são personagens com a habilidade de assumir um papel no ambiente com comportamento de improviso. Para alcançar este objetivo, os agentes devem agir no mundo virtual, perceber, reagir e lembrar suas percepções sobre este mundo, ter a inteligência de pensar sobre os efeitos que suas ações causarão e, finalmente, aprender com sua experiência.

Para ilustrar como a performance é um problema crucial na simulação de multidões, é conhecido que para um *game* ser jogável, ele precisa ser executado em um *rate* entre 30 e 100 *frames* por segundo, o que implica em 0,02 segundos por *frame*. Cada *frame* (passo de uma animação) inclui a atualização do *status* do *game*, controle de entradas do usuário, processamento gráfico, computação de física, estratégia da Inteligência Artificial, planejamento de caminhos (*pathplanning*), entre outros. Depois é possível considerar algo em torno de um milissegundo por etapa de *pathplanning* (com arquitetura *multicore* essa restrição é relaxada).

É importante em uma simulação de *pathplanning* que as células de obstáculos sejam atualizadas antes das células de objetivo. Caso elas não sejam atualizadas nesta forma sequencial, caso um agente tenha um objetivo muito próximo de um obstáculo, tanto o objetivo como o obstáculo serão mapeados para a mesma célula e o agente poderá ficar perdido sem objetivo para alcançar [2].

Porém esta técnica garante 25% de eficiência que processa toda a lógica do jogo dentro da GPU. Diversos jogos usam um processo de GPGPU que processa algumas tarefas na GPU e outras na CPU. Isso causa limitações na simulação pois requer grande quantidade de transferência de dados entre a CPU e a GPU, o que pode "gargalhar" a simulação, deixando-a com performance indesejável. Utilizando a CUDA para esse processo é possível minimizar a comunicação entre GPU e a CPU [3].

METODOLOGIA

Inicialmente quatro fases diferentes são propostas para o projeto e que estão divididas como mostra os itens a seguir:

- Modelagem do Problema
- Implementação da DLL de Otimização
- Implementação do Ambiente de Testes
- Testes da Aplicação

Modelagem do Problema

Durante a primeira nova etapa que já está em progresso, chamada de "Modelagem do Problema", serão levantados os requisitos necessários para uma boa implementação, assim como fazer modelos de caso de uso, diagramas de classe e diagramas de sequência. Estes modelos serão fundamentais para arquitetar formalmente o problema e facilitar a etapa de implementação do sistema que será desenvolvido para otimizar multidões.

A previsão é que esta etapa não consuma mais do que um mês para arquitetar todo o problema de implementação da DLL e do ambiente de testes. O objetivo principal desta fase é guiar o desenvolvimento e evolução da solução.

Implementação da DLL de Otimização

Com o auxílio da linguagem CUDA em C para programar GPUs previamente estudada e da interação, modelagem, simulação e renderização de multidões em jogos, o objetivo da fase de "Implementação da DLL de Otimização" é criar um *framework* em formato de DLL que poderá ser utilizado por *game engines* para otimizar problemas relacionados com uma grande quantidade de modelos simultaneamente na tela, sejam estes problemas relacionados com a criação, a renderização ou da interação entre multidões.

Todavia, é conhecido na área de otimização de processamento gráfico que é impossível com a tecnologia computacional atual otimizar todos os quesitos ao mesmo tempo. Portanto, caso a renderização de modelos seja alavancada pelo *framework*, a interação de multidões será prejudicada e vice-versa. Caberá ao desenvolvedor do jogo em questão optar qual otimização ele irá preferir durante a execução do *game* e indicar para a DLL qual etapa do tratamento de multidões necessitará de maior processamento paralelo via placas gráficas.

O desenvolvimento do *framework* utilizará as técnicas abordadas e estudadas anteriormente para otimizar da melhor maneira possível todos os tipos de otimização de multidões (criação, renderização e interação), porém ele não suportará, a princípio, uma otimização generalizada de todos os tipos devido a questões de tempo e recursos de *hardware* que serão utilizados no projeto.

Para todos os tipos de otimização envolvidos no projeto e citados no artigo, a DLL sempre focará na otimização em

termos de velocidade e agilidade de processamento. Existem outros tipos de otimização como performance e qualidade em jogos, mas estes tipos de otimização fogem do objetivo do projeto e das técnicas previamente estudadas nas referências da dissertação.

A implementação da DLL será voltada para a otimização de multidões em jogos digitais, independentemente se forem multidões de personagens humanos, animais, monstros ou qualquer outro tipo de multidões de NPCs. A DLL deverá ser informada por funções compartilhadas com o jogo as seguintes indicações:

- Tipo de otimização que será realizada (Criação, Renderização ou Interação entre os personagens).
- Indicador representativo de cada agente da multidão.
- Posição inicial de cada indivíduo de uma multidão.
- Dados de modelagem (para o caso de uma otimização de criação).
- Malha de textura (para o caso de uma otimização de renderização).
- Posição inicial e objetivos (para o caso de uma otimização de interação).

As funções disponibilizadas pelo *framework* deverão controlar todos os aspectos de tratamento de programação paralela de GPU em multidões para o desenvolvedor do jogo.

Com os dados mencionados acima, será possível abstrair o tratamento de multidões e agilizar o desenvolvimento dos outros componentes do jogo que não dependam de multidões.

A partir deste ponto, o *framework* assumirá o controle do processamento paralelo para a multidão e os tipos de otimização selecionados pelo desenvolvedor através da *game engine* e terá tanto os recursos de programação em GPU quanto de uma *game engine* como idealizado para a implementação de um jogo [6].

A dificuldade de interação e queda de processamento que diminuem drasticamente a imersão que os modelos de multidão idealmente deveriam proporcionar para o jogador inspiraram a criação desta fase de implementação de uma DLL, para que desenvolvedores de jogos possam usufruir de funções de otimização desenvolvidas em linguagem CUDA em C.

Este framework em questão disponibilizará funções em linguagem C que serão utilizadas para modelagem, renderização e interação com multidões, mas de forma paralela e sincronizada, poupando tempo do desenvolvedor que desejar trabalhar com tais recursos sem precisar administrar as *threads* de placa gráfica e, mesmo assim, se aproveitar do poder computacional que elas oferecem através da DLL.

Implementação do Ambiente de Testes

Para auxiliar na fase de testes, serão desenvolvidos dois jogos utilizando a *game engine* Unity 3D para comparar a eficiência da DLL implementada. A *game engine* Unity 3D foi previamente estudada para ser possível arquitetar e desenvolver um ambiente de testes em formato de dois jogos controlados, sendo que este ambiente deve ser flexível para testes com e sem a DLL de otimização para comprovar a maior eficiência e eficácia que a DLL desenvolvida proporciona em comparação com o ambiente sem otimização de GPU.

O primeiro jogo será apenas um teste de exaustão da CPU e da GPU com multidões. Nele serão criados vários modelos aleatórios que interagirão randomicamente sem preocupação com detecção de colisão ou renderização de modelos. O objetivo deste ambiente em específico é apenas sobrecarregar o processamento para comparar como a máquina de testes reage em exaustão utilizando a DLL em comparação quando ela não utiliza a DLL para este ambiente de exaustão em questão.

Já o segundo jogo será voltado para um caso mais real de multidões em um FPS (*First Person Shooter*) desenvolvido especialmente para estes testes chamado "*Apple Shooter*".

Neste jogo diversas maçãs representam as multidões de inimigos que aparecem desenfreadamente para serem eliminadas, como demonstra a figura 2 a seguir:

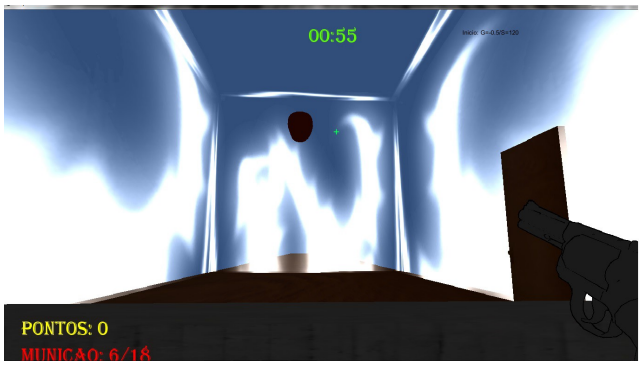

FIGURA. 2
SCREENSHOT DO JOGO DE TESTES APPLESHOOTER

Com o ambiente de testes do *AppleShooter* será possível comprovar a qualidade da DLL desenvolvida para um caso real de *game*, direcionar e motivar sua utilização por desenvolvedores com problemas de multidões no mercado de jogos.

Testes da Aplicação

A fase de testes da aplicação é fundamental não apenas para avaliar a qualidade e funcionalidade do *framework* desenvolvido, mas, além disso, nesta etapa será possível apontar e corrigir possíveis erros de implementação que

podem ter passado desapercebidos na fase anterior, evitando que o *framework* apresente resultados indesejáveis.

Também será possível ajustar o ambiente de testes para se adequar melhor com a DLL proposta e obter resultados mais satisfatórios que não eram possíveis anteriormente.

Portanto, as etapas de modelagem de problemas, implementação do *framework* e implementação do ambiente de testes voltarão ao final da fase de testes da aplicação, para finalizar detalhes de arquitetura e implementação encontrados durante estas fases de erros nos testes.

REFERÊNCIAS

[1] DA SILVA, A. R.; LAGES, W. S.; CHAIMOWICZ, L. Boids that see. ACM Association For Computing Machinery, Minas Gerais, v. 7, n. 4, 2009.

[2] FISCHER, L.; SILVEIRA, R.; NEDEL, L. Gpu accelerated path-planning for multi-agents in virtual environments. SBGames, IEEE Computer Society, Rio Grande do Sul, 2009.

[3] JOSELLI, M. et al. Techniques for designing gpgpu games. International Games Innovation Conference, IEEE Computer Society, Rio de Janeiro, 2008.

[4] JOSELLI, M. et al. A new physics engine with automatic process distribution between cpu-gpu. ACM Association For Computing Machinery, Rio de Janeiro, 2008.

[5] JOSELLI, M. et al. A neighborhood grid data structure for massive 3d crowd simulation on gpu. Brazilian Symposium on Digital Games and Entertainment, IEEE Computer Society, Rio de Janeiro, 2009.

[6] NAVARRO, A. et al. Using 3d game engines and gpu for ray launching based channel modeling in indoor. IEEE Computer Society, Cali and Cucuta and Valencia, 2014.

[7] SILVA, A. et al. Gpu path_nding optimization. SBGames, ACM Association For Computing Machinery, Pernambuco, 2011.

[8] TANG, B.; MIAO, L. Real-time rendering for 3d game terrain with gpu optimization. Second International Conference on Computer Modeling and Simulation, ACM Association For Computing Machinery, Jinhua, 2010.

[9] VIGUERAS, G.; ORDUÑA, J.; LOZANO, M. Performance improvements of real-time crowd simulations. IEEE Computer Society, Valencia, 2010.

[10] YILMAZ, E.; ISLER, V.; ÇETIN, Y. Y. The virtual marathon: Parallel computing supports crowd simulations. IEEE Computer Society, p. 26{33, 2009. 11

[11] YONG, Z. et al. Real-time crowd rendering and interactions on gpu. International Conference on Computer Science and Software Engineering, IEEE Computer Society, Pequim, 2008.