Plano de Trabalho de Conclusão de Curso
Otimização de Motor de Jogos Através de Modelos Orientados a Dados

UDESC – Centro de Ciências Tecnológicas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação – Integral
Turma 2013/2 – Joinville/SC

Vinícius Bruch Zuchi – `vinicius.b.zuchi@gmail.com`
André Tavares da Silva – `andre.silva@udesc.br` *(orientador)*

17 de Março de 2017

Resumo

Devido à grande diferença de performance entre os microprocessadores e a velocidade de acesso à memória principal, muitas aplicações de computador modernas têm sofrido com um problema conhecido como gargalo do processador-memória, no qual o processador fica ocioso aguardando dados serem buscados na memória. Aplicações gráficas, principalmente jogos eletrônicos, são notórias por sofrerem deste problema pela grande quantidade de elementos diferentes que as constituem. Este trabalho propõe a implementação de um motor de jogo (uma engine), utilizando um conceito relativamente novo de programação, o design orientado a dados, com a finalidade de amenizar este problema de acesso constante à memória, escrita na linguagem Rust, linguagem a qual visa velocidade e uso eficiente e seguro da memória.

Palavras-chave: *Design Orientado a Dados (DOD). Engine. Rust*

1 Introdução e Justificativa

Nas últimas décadas, os microprocessadores tiveram um enorme avanço em termos de performance. A função deste componente essencial para os computadores modernos, que por vezes também é chamado de CPU (*Central Process Unit*), é a de processar as instruções e os dados de todos os programas presentes no computador (Clements, 2006).

Além de executar essas instruções, o microprocessador está constantemente lendo e escrevendo dados na memória principal, a memória RAM (*Random Access Memory*) (Clements, 2006). Sem essa possibilidade de leitura e escrita, os programas de computador por si só não teriam muitas utilidades, e sistemas complexos não poderiam existir.

Porém, há um problema na comunicação entre o processador e a memória: ela não é instantânea. Um sistema no qual a memória pode fornecer qualquer dado para o processador instantaneamente não é factível. Por este motivo, sistemas de memória são projetados utilizando uma hierarquia, na qual a memória é dividida em diferentes camadas, cada uma com uma capacidade de armazenamento maior, porém com um tempo de acesso maior do processador (Mahapatra and Venkatrao, 1999), a figura 1 demonstra essa hierarquia em ordem decrescente de velocidade.

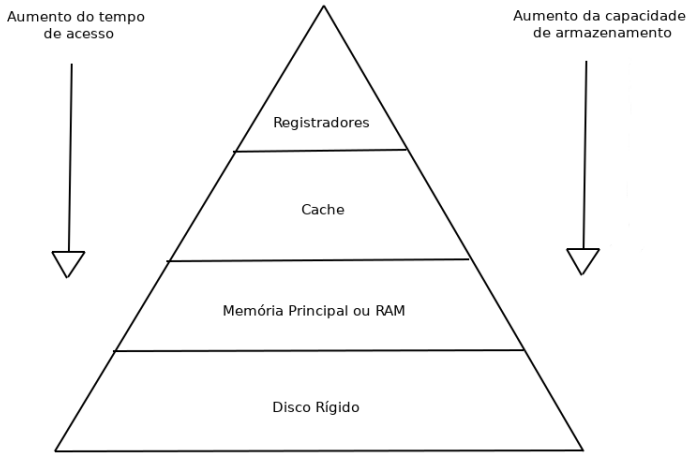

Figura 1: A hierarquia do sistema de memória.

As camadas do sistema de memória estão descritas a seguir:

- Registradores: são as porções da memória com o acesso mais rápido, porém em menor quantidade e com pouca capacidade de armazenamento. São localizadas dentro do processador, e os compiladores são os responsáveis pelo seu uso (Clements, 2006).
- Cache: localizado próximo ao processador, fornece velocidade de acesso rápida e armazena código e dados utilizados recentemente. Quando o processador tenta acessar algum dado presente no cache, ocorre um acerto do cache (*cache hit*), quando o dado não se encontra no cache, ocorre um erro do cache (*cache miss*), e o dado precisa ser buscado na memória principal (Mahapatra and Venkatrao, 1999).
- Memória Principal, *Dynamic Random Access Memory* (DRAM): em maior quantidade quando comparado ao cache, possui um custo menor, porém com um tempo de acesso maior, é responsável por lidar com as operações de entrada e saída. Apesar de ser muito mais lenta do que o cache, sua estrutura simplificada e relativamente baixo custo fez com que a DRAM se tornasse a memória principal dos sistemas modernos (Mahapatra and Venkatrao, 1999).

Como citado anteriormente, os processadores obtiveram um enorme avanço nos últimos tempos. Porém, a memória DRAM não foi capaz de acompanhar esse crescimento dos processadores na

mesma velocidade. Desde a década de 1980 até a última década, os processadores têm aprimorado a uma taxa de 60 porcento ao ano, enquanto que o tempo de acesso à DRAM tem aprimorado a uma taxa de menos de 10 porcento ao ano (Patterson et al., 1997).

Essa disparidade de crescimento tem aumentado a lacuna de performance entre processador-memória, pois a latência de acesso a memória pelo processador está cada vez maior. Sempre que ocorre um *cache miss*, o processador precisa ficar alguns ciclos ocioso enquanto aguarda o dado necessário ser buscado na memória principal (Mahapatra and Venkatrao, 1999), como a velocidade de acesso à memória é relativamente muito menor do que a velocidade de processamento, um processador mais rápido significa apenas mais ciclos ociosos. Isso é conhecido como "Gargalo do Processador-Memória", e diante de tais limitações físicas, surge a necessidade de boas práticas de programação para que ocorra a menor quantidade de *cache misses* o possível.

Com esse problema em mente, ao analisar o design de programação orientado objetos, percebe-se que essa abordagem, enquanto que mais legível e com código mais reutilizável, sofre muitas vezes em termos de eficiência de memória, pois seu design é centrado em torno do problema e suas possíveis soluções, e não em torno dos dados, fornecendo abstração dos dados porém a custo de performance.

Quando uma classe de objetos possui dados, chamados de atributos, isto significa que essa classe está fornecendo um contexto aos dados, e este pode comprometer o uso desses dados, pois ao adicionar métodos ao contexto, pode-se criar a necessidade de adicionar mais dados à classe, o que pode rapidamente levar à classes que contém diversos fragmentos de dados que não estão relacionados entre si (Fabian, 2013).

Outra desvantagem do uso demasiado de programação orientada a objetos, está em sua própria definição, que opera sobre um único objeto. Essa organização de dados não é benéfica ao processador, ao buscar objetos na memória para se realizar operações sobre atributos específicos destes, todos os outros dados pertencentes à classe também precisam ser carregados, deixando os caches poluídos com dados desnecessários e aumentando a incoerência de cache.

Com a premissa de amenizar o máximo possível estes problemas, o design orientado a dados, como o nome sugere, encoraja a mudança da perspectiva da programação dos objetos para os dados, seus tipos, como está armazenado na memória e como será lido e processado (Fabian, 2013). Para alcançar esses objetivos, essa técnica sugere dividir cada objeto em diferentes componentes, e agrupar componentes do mesmo tipo na memória, sem se importar de qual objeto vieram. Isso resulta em largos blocos de dados homogêneos, permitindo o processamento sequencial dos dados, e garantindo um aprimoramento significativo na performance (Fabian, 2013).

Esta abordagem de design é mais condizente com a realidade de muitos programas complexos, raramente há apenas um ou uma pequena quantidade de objetos de um determinado tipo, sendo necessário o processamento de todos eles. Mas esta abordagem não traz a solução para todos os problemas, e também possui suas desvantagens, uma delas é a falta de intuitividade para codificação orientada a dados, uma das vantagens da orientação a objetos é a similaridade do pensamento com o mundo real e os seus problemas, deixando o código mais legível para os humanos. O design orientado a dados requer que o programador raciocine com uma mentalidade muito diferente da

qual a maioria está acostumada, e conceber código não intuitivo pode trazer complicações futuras para a adição de novas funcionalidades ou correções.

Para as aplicações na área de computação gráfica, esses problemas não seriam diferentes, principalmente levando em consideração que uma grande parte dos trabalhos nessa área utilizam a programação orientada a objetos. Essas aplicações, principalmente jogos eletrônicos modernos, utilizam um complexo sistema o qual possui muitos componentes diferentes que contém dados que constantemente transitam entre a memória, o processador e a GPU. Esse sistema é comumente conhecido como motor de jogo (do inglês *Game Engine*), ou simplesmente como engine.

Uma engine, apesar de não possuir uma definição absoluta, é geralmente entendida como o software responsável por lidar com todos os módulos que juntos compõem um jogo eletrônico, como por exemplo, bibliotecas matemáticas (contendo operações de vetores, matrizes, quaternions, métodos trigonométricos, etc.), gerência de memória, estruturas de dados personalizadas, o motor de renderização, gerenciador de recursos, ferramentas para depuração e análise de performance, sistemas de colisão e física, sistema de animações e partículas, sistema de detecção de inputs (mouse, teclado e outros controladores), sistema de áudio, sistema de rede para jogos online, entre outros (Gregory, 2009).

O motor de renderização, também conhecido como motor gráfico, é um dos maiores e mais complexos componentes de um motor de jogos. E apesar de não possuir apenas uma única maneira de projetá-los, a maioria dos motores de renderização modernos seguem filosofias de design em comum (Gregory, 2009). Esse componente será responsável por todos os métodos, estruturas e otimizações que serão responsáveis pela renderização dos gráficos e animações do jogo.

Um dos elementos presentes no motor gráfico é uma interface de dispositivo gráficos, que será responsável pela comunicação com a GPU e, consequentemente, com a renderização de baixo nível. Uma das bibliotecas capazes de realizar essa função é o OpenGL, uma API gráfica para acessar os recursos da GPU, contendo um rico conjunto de comandos (mais de 500) que são utilizados para especificar objetos, imagens e operações necessárias para a renderização de aplicações gráficas (Shreiner et al., 2013).

A programação em OpenGL moderno também requer o uso e entendimento de outro conceito igualmente importante, os shaders, que são pequenos programas que são especialmente compilados para a GPU e contém instruções que são executadas diretamente nesta (Shreiner et al., 2013). Shaders em OpenGL utilizam uma linguagem de programação própria, o GLSL (*OpenGL Shading Language*) (Shreiner et al., 2013).

Além dessa camada de renderização de baixo nível, o motor gráfico também conta com componentes essenciais de mais alto nível que são responsáveis pela gerência da geometria das malhas, como um grafo de cenas, que além de manipular as malhas estabelece a hierarquia entre elas e define subdivisões espaciais (Gregory, 2009). Técnicas de otimizações também são importantes, como por exemplo a remoção parcial de objetos que não são considerados para contribuir à imagem final, essa técnica é conhecida como *culling* (Akenine-Möller et al., 2008).

Todos esses elementos, técnicas e otimizações do motor gráfico no fim constroem o que é conhecido como pipeline de renderização. A principal função desse pipeline é renderizar uma imagem

bidimensional, dada uma câmera virtual, objetos tridimensionais, fontes de luz, equações de *shading*, texturas, entre outros (Akenine-Möller et al., 2008).

Considerando os problemas supracitados a respeito do gargalo de performance do processador-memória, e das desvantagens da programação orientada a objetos, este trabalho propõe a implementação de uma engine desde o princípio com o objetivo de explorar o potencial do design orientado a dados, um conceito pouco difundido entre a comunidade de programadores, e seus benefícios para aplicações gráficas, uma área na qual a orientação a objetos está fortemente impregnada. Uma vez que uma engine completa é uma aplicação consideravelmente grande e complexa, este trabalho irá se concentrar na implementação do motor gráfico e suas otimizações. Para a implementação da engine, foi escolhida a linguagem de programação Rust.

Rust é uma linguagem de programação de baixo nível, com tipagem estática e forte. A linguagem foi projetada com os objetivos de ser rápida, fácil de ser paralelizada e ter segurança de memória, prevenindo a ocorrência de condições de corrida, estouros de pilha, e acessos a posições de memória não inicializadas ou desalocadas (Matsakis and Klock, 2014). Além disso, como a linguagem é relativamente nova e grande parte das aplicações gráficas são escritas em C ou C++, esse trabalho pode ser considerado um bom meio para testar a performance da linguagem.

2 Objetivos

Objetivo geral: Implementar uma engine na linguagem de programação Rust e analisar sua eficiência e performance.

Objetivos específicos: Abaixo encontra-se uma lista dos principais objetivos a serem alcançados com este trabalho prático.

- Verificar o estado atual no que diz respeito às implementações das engines modernas;
- Explorar o potencial e limitações da linguagem de programação Rust para aplicações gráficas;
- Utilizar o conceito relativamente novo de design orientado a dados para proporcionar uma melhora na performance dos motores gráficos através da coerência de cache.
- Comparar a performance da engine implementada com outras que fornecem funcionalidades semelhantes mas que possuem um design orientado a objetos;
- Analisar a performance da engine, calculando uma média do total de *cache miss* ao longo de sua execução.

3 Metodologia

Este trabalho de conclusão de curso é essencialmente prático. Após a etapa de revisão bibliográfica sobre os conceitos necessários para a implementação da engine e o estudo aprofundado da linguagem Rust e suas características avançadas, a aplicação será desenvolvida em etapas e o método utilizado

será a implementação dos componentes da engine sequencialmente, no qual estes serão testados individualmente.

A próxima etapa será a integração dos componentes para a construção do pipeline gráfico e implementação das outras funcionalidades. Com a engine tendo todos os seus componentes e características funcionando devidamente, será por fim feito análises e comparações de sua performance e eficiência. Um *postmortem* será escrito no TCC, explicando o que foi e o que não foi bem sucedido.

Para a realização deste trabalho, as seguintes etapas deverão ser cumpridas:

(1) Revisão bibliográfica, sobre trabalhos relacionados ao desenvolvimento de motores gráficos e design orientado a dados.

(2) Estudo minucioso da linguagem Rust, suas características avançadas e sua filosofia sobre codificação e gestão de memória;

(3) Implementação e teste de uma biblioteca matemática minimalista, contendo operações para vetores, matrizes, quaternions e métodos trigonométricos.

(4) Revisão da biblioteca OpenGL, sua integração com a linguagem Rust, o fluxo base de uma engine e também da linguagem de alto nível para shaders GLSL;

(5) Implementação e teste de uma estrutura para manipulação da câmera e do pipeline gráfico;

(6) Implementação e teste do grafo de cena e do leitor de malhas externas.

(7) Projetar o sistema de iluminação;

(8) Integrar possíveis funcionalidades adicionais necessárias;

(9) Escrita da monografia da primeira parte;

(10) Desenvolvimento de uma aplicação através da engine para realização de *benchmarks*;

(11) Análise e comparação da performance da engine;

(12) Escrita da monografia da segunda parte, juntamente com o *postmortem* da engine.

4 Cronograma proposto

Etapas	2017											
	J	F	M	A	M	J	J	A	S	O	N	D
1			x									
2			x	x								
3			x	x								
4				x								
5				x	x							
6						x	x					
7							x					
8							x	x				
9			x	x	x	x						
10								x	x	x		
11										x	x	
12									x	x	x	

5 Linha e Grupo de Pesquisa

Este trabalho pertence a área de Computação Gráfica. O trabalho foca em rendering em tempo real para motores de jogos (game engine) e programação para GPU. Está associado ao grupo de pesquisa LARVA (*LAboratory for Research on Visual Applications*).

6 Forma de Acompanhamento/Orientação

Reuniões semanais serão feitas, salvo em situações nas quais ocorrerem imprevistos. Nesses casos, será avaliado a disponibilidade do orientador e do orientado para que um horário extra seja marcado, além de eventuais trocas de e-mails. Um repósitório no Github será criado para que o orientador possa acompanhar o processo de implementação do trabalho.

Referências

Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-Time Rendering, Third Edition*. CRC Press.

Clements, A. (2006). *Principles of Computer Hardware*. Oxford University Press, Inc., New York, NY, USA.

Fabian, R. (2013). *Data-Oriented Design*. DataOrientedDesign.com.

Gregory, J. (2009). *Game Engine Architecture*. Taylor & Francis.

Mahapatra, N. R. and Venkatrao, B. (1999). The processor-memory bottleneck: Problems and solutions. *Crossroads*, 5(3es).

Matsakis, N. D. and Klock, II, F. S. (2014). The rust language. *Ada Lett.*, 34(3):103–104.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. (1997). A case for intelligent ram. *IEEE Micro*, 17(2):34–44.

Shreiner, D., Sellers, G., Kessenich, J., and Licea-Kane, B. (2013). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. OpenGL. Pearson Education.

Andre Tavares da Silva ***Vinicius Bruch Zuchi***