Faculdade projeção

DOD para otimização server-side em jogos

Documento de Visão

Versão 1.6.1

Histórico de Revisão

Data	Versão	Descrição	Autor
06/09/2020	1.0.0	Criação do Documento de Visão para descrição geral através do tema “DOD para otimização server-side em jogos”.	Nádio Dib
09/09/2020	1.1.0	Adição de conteúdo no capítulo 1 e revisão da formatação dos elementos textuais e contextuais deste item mediante adaptação para pesquisa científica.	Nádio Dib
09/09/2020	1.1.1	Adição de itens, definições e referências bibliográficas ao glossário deste documento. Além de correção de espaçamento de informações fornecidas no conteúdo do capítulo 1.	Nádio Dib
12/09/2020	1.2.0	Adição de conteúdo no capítulo 2.	Nádio Dib
12/09/2020	1.3.0	Adição de conteúdo no capítulo 3 e revisão da formatação dos elementos textuais do capítulo 2. Além de adição de itens pendentes no glossário.	Nádio Dib
12/09/2020	1.3.1	Correção ortográfica no título do capítulo 4.	Nádio Dib
12/09/2020	1.4.0	Adição de conteúdo no capítulo 4. Revisão da formatação dos elementos textuais dos capítulos 2 e 3. Correção de formatação de elementos das referências bibliográficas.	Nádio Dib
14/09/2020	1.5.0	Adição de conteúdo nos capítulos 5 e 6. Revisão da formatação do documento.	Nádio Dib
14/09/2020	1.5.1	Adição de conteúdo extra no capítulo 6.	Nádio Dib
26/10/2020	1.6.0	Adição de conteúdo no capítulo 7 e 8. Finalização da Conclusão e atribuição de Anexos e Apêndices referentes a pesquisa realizada. Revisão da formatação do documento. Correção de formatação de elementos visuais (figuras e espaçamentos) nos capítulos do documento.	Nádio Dib
24/11/2020	1.6.1	Formatações finais em todo o Documento de Visão, segundo orientações e recomendações.	Nádio Dib

Faculdade projeção

Sumário

Introdução

Ao decorrer de vários anos de trabalho informal na área de desenvolvimento de jogos para criação, manutenção e otimização de aplicações server-side ou *back-end* para servidores através da utilização do *framework* .NET, percebi que sobrecargas de execução nas operações de rotina repetitiva ou *loop* eram de intensa interação entre instâncias de objetos quando se tratava de comunicação por meio de encapsulamento de mensagens com a abordagem do paradigma *Object-Oriented Programming* (OOP) eram muito constantes e necessitavam e atenção.

Logo, a proposta deste documento é coletar, analisar e definir as necessidades e funcionalidades gerais do projeto "DOD para otimização server-side em jogos" para fins acadêmicos e de estudo prático.

Seu foco está nas necessidades da devida compreensão de conceitos e fundamentos computacionais para uma melhor adequação de operações exaustivas exigidas por aplicações de servidor de jogos através de uma nova abordagem feita pelos materiais levantados de DOD e os motivos da existência destas necessidades para mitigação dos problemas contextuais oriundos deste mesmo tema.

Termos e abreviaturas específicos podem ser encontrados no Glossário do respectivo projeto.

Contextualização

Dentro desta perspectiva, rotinas repetitivas em servidores de jogos tornaram-se regiões críticas e sensíveis durante suas operações quando em funcionamento no modo de produção ou *release*. Estas rotinas se caracterizaram como regiões críticas devido a implementação de conceitos de *multithreading* para utilização de recursos de programação paralela concorrente, nos quais, muitas das vezes eram caracterizados por recursos compartilhados entre processos adjuntos das rotinas para fins de otimização.

A utilização de conceitos e fundamentos dos assuntos de *concurrent programming* e *parallel programming* auxiliaram até de certa forma na confecção de uma solução para mitigação de sobrecargas de trabalho nestas operações exigidas por essas rotinas, que se caracterizavam por requisitar alto uso de recursos da CPU, no qual a aplicação estava em funcionamento, naquele ambiente computacional.

Entretanto, após investigar com mais detalhe sobre como possibilitar uma melhor adequação para estas rotinas, a simples utilização dos fundamentos de *Data-Oriented Design* (DOD) juntamente com o devido discernimento do funcionamento do sistema operacional, possibilitavam menor utilização dos recursos computacionais, pelo qual a aplicação está em funcionamento naquele ambiente operacional, e proporcionar confidencialidade no tratamento de dados para procedimentos de concorrência dos recursos, pois estes problemas recorrentes da utilização de conceitos e implementações de funcionalidades *multithreading* seriam ausentes, nos quais procedimentos assíncronos passariam a ser síncronos e pertencentes da mesma *thread* que caracteriza rotina.

De acordo com os assuntos abordados previamente na Introdução deste Documento de Visão e Escopo (DVE) de forma superficial, há a necessidade de apresentar esta abordagem oriunda de conceitos existentes das áreas de desenvolvimento de software como Sistemas Operacionais, Arquitetura de Software, Estrutura de Dados e Algoritmos, Modelagem de Software Orientado a Objeto e Lógica de Programação, nos quais fazem parte da formação acadêmica do curso de Engenharia de Software.

Problemática

Operações em rotinas que desempenham processos de longa duração e iterações entre inúmeras instâncias de objetos através do paradigma OOP, não são eficientes quando suscetível a uma grande coleção de instâncias de objetos para iteração resultando no aumento do tempo de resposta da rotina que desempenha um papel crítico para toda a aplicação.

Este problema afeta a integridade e confiabilidade de aplicações de jogos para servidores, comprometendo o seu devido funcionamento sem a necessidade de alocação de melhores recursos computacionais, apenas trabalhando com os já existentes.

A necessidade de alocação e investimento de máquinas mais robustas para o devido funcionamento da aplicação de jogos em servidores, bem como alto custo para manutenção e mitigação deste problema abordando apenas o paradigma OOP, onde é necessário o contrato de profissionais mais experientes para apoio técnico de *tunning* ou utilização de outras tecnologias para serem acopladas na aplicação e assim possibilitar uma falsa sensação de solução para este problema.

Solução Proposta

Utilizar os conceitos e fundamentos multidisciplinares do curso de graduação de Engenharia de Software com o padrão de desenvolvimento DOD, para auxiliar em uma melhor compreensão deste problema sem a necessidade de alto investimento financeiro para migração de outras tecnologias, pois este estudo tem como base providenciar uma sugestão de solução para este problema específico nos quais grandes corporações recentemente utilizaram para outros fins, porém de mesma complexidade e necessidade.

Sendo assim, proponho a devida interação multidisciplinar para fundamentar os conceitos que serão relacionados no decorrer deste projeto.

O que se esperar

Ao final deste projeto, é esperado alcançar um resultado satisfatório provenientes de testes, citações científicas, referências bibliográficas acadêmicas, bem como aplicação dos fundamentos multidisciplinares citados na Contextualização deste DVE em prática.

Objetivo Geral

O objetivo deste documento é contribuir para o aprimoramento e aperfeiçoar a qualidade das informações ofertadas, provenientes de pesquisas e estudos científicos. Sendo assim, este projeto tem como objetivo geral englobar várias áreas de conhecimento que estão interligadas de modo a atender as necessidades essenciais para o devido atendimento destes requisitos propostos na Solução Proposta deste DVE.

Objetivos Específicos

Utilizando como referência principal, mas não somente analisando de forma genérica certos fundamentos e conceitos teóricos, este projeto tem como ênfase específica nos seguintes objetivos:

I. Compreender o devido funcionamento de procedimentos de encapsulamento da mensagem através do paradigma OOP dentro do ambiente computacional;
II. Analisar de forma conceitual partições da memória gerenciadas pelo sistema operacional de modo a atender as necessidades de compreensão para o padrão de desenvolvimento DOD;
III. Interpretar as vantagens e desvantagens entre o paradigma OOP e o padrão de desenvolvimento DOD;
IV. Proporcionar através de uma nova perspectiva a devida relação entre o padrão de desenvolvimento DOD com recursos computacionais, sendo estes o processador e procedimentos de alocação de memória;
V. Capacitar possibilidade de generalização do padrão de desenvolvimento DOD para outros fins que atendem as mesmas características publicadas através dos relatórios de teste e demandam mesma aplicabilidade.

Estrutura do Projeto

Este projeto foi alocado nos seguintes capítulos de modo a auxiliar o seu devido entendimento e compreensão com a linha de pesquisa adotada, sendo estes:

- Capítulo 1 – alocação de memória durante procedimentos instanciação de objetos;
- Capítulo 2 – quando o paradigma OOP se torna um problema;
- Capítulo 3 – funcionamento da memória dentro do ambiente computacional;
- Capítulo 4 – operações assíncronas para otimização de processos através dos fundamentos *multithreading*;
- Capítulo 5 – OOP *versus* DOD;
- Capítulo 6 – os motivos por trás do DOD;
- Capítulo 7 – a devida utilização DOD em operações de alta performance e baixa latência de resposta em aplicações *server-side* de jogos;
- Capítulo 8 – as aplicações do padrão de desenvolvimento DOD pela empresa Unity em sua nova tecnologia, o Unity DOTS;
- Conclusão;
- Referências Bibliográficas;
- Glossário;
- Anexos;
- Apêndices.

Capítulo 1 – alocação de memória durante procedimento de instanciação de objetos

Quando trabalhamos com procedimentos do paradigma de orientação à objetos ou OOP, devemos ter em mente como de fato as ações de instanciação são organizadas na memória. Para isso é necessário compreender que existem duas maneiras[1] de alocar dados na memória, como:

- **Alocação de memória através da compilação:** caracterizado ou não por alocação estática, pois a memória requerida que deverá ser alocada por variáveis, tamanhos fixos, tipos de dados a serem trabalhados e conjunto padrão de coleções e definições devem ser conhecidos e alocados neste procedimento durante a compilação do programa; e
- **Alocação de memória dinâmica:** este procedimento é caracterizado por uma alocação dinâmica da memória, ou seja, no decorrer do uso da aplicação a memória exigida é devidamente realocada de acordo com os segmentos programados e definições de tamanho ou quantidade de espaço exigido que não precisam ser conhecidas pelo compilador. Entretanto, apesar de criar vários itens ou instâncias de objetos que serão alocados na memória, a alocação dinâmica de memória não fornece suporte para criar novos nomes de itens, ou seja, através de operações programadas utilizam-se recursos já existentes naquelas operações para assim criar através de forma dinâmica um espaço na memória para aquele novo item que baseia-se nas definições pré-definidas do algoritmo.

Todavia, durante a instanciação de objetos na memória deve ser realizado com cautela. Tanto em ações de criação quanto em destruição, é necessário ter essa preocupação do programador, pois a alocação exacerbada de novas instâncias de objetos na memória sem o devido tratamento de procedimentos de remoção dos espaços alocados, caracteriza uma situação de **vazamento de memória** ou *memory leak*[2]. Entretanto, é importante ressaltar que somente classes de objetos que serão instanciadas na memória de forma dinâmica, ou *constructor*, necessitarão de procedimentos de liberação dos espaços alocados, conhecido pela terminologia *deconstructor*.

Existem linguagens e *frameworks*, que provém a liberação de espaços alocados da memória de forma a diminuir essa preocupação do programador, como por exemplo: nas linguagens Java e C# (.NET) a liberação de instâncias alocadas de objetos sem a devida utilização são constantemente descartadas através do ***garbage collector***[2] e suas gerações que operam como auxiliadores na coleta de lixo para evitar estouros de memória durante a operação da aplicação no sistema operacional.

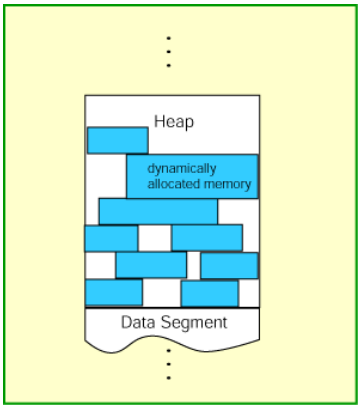

Figura 1.1 – caracterização de buracos ou espaços na memória através do problema de *memory fragmentation*. Fonte: Central Connecticut State University.

Basicamente, a utilização da alocação de memória dinâmica parte do princípio na OOP de que a memória é fornecida como um facilitador para auxiliar as requisições de retorno de endereços onde os dados podem ser pesquisados, verificados e alterados de forma dinâmica, assim facilitando os procedimentos para essas operações complexas. Porém, situações como criação de vários buracos ou *gaps* dentro da memória através de ações de criação ou destruição, podem desenvolver problemas que fazem deste conceito tornar-se ineficiente, pois essa situação é conhecida como **fragmentação da memória** ou *memory fragmentation*[3], como pode ser analisado na **figura 1.1** os espaços criados. Felizmente, essa situação pode ser mitigada em sistemas operacionais modernos.

Capítulo 2 – quando o paradigma OOP se torna um problema

O paradigma de orientação à objetos é ainda muito utilizado no desenvolvimento de software atualmente e consigo trás várias abordagens, nos quais auxiliam o programador a efetuar operações de diferente nível de complexidade com a devida compreensão deste paradigma. A implementação deste tipo de paradigma fornece a capacidade ao programador em compreender procedimentos de engenharia de uma perspectiva mais simples e fácil.

Algumas das principais vantagens quanto a utilização do paradigma OOP, podem ser listadas abaixo:

- **Herança e virtualização de métodos[6][7]:** permite que objetos sejam definidos de modo a permitir que todos seus dados e métodos existentes ao seu antecessor sejam herdados. Entretanto, alguns métodos oriundos da definição de métodos antecessores possuem a capacidade de virtualização, ou seja, apesar de herdarem os comportamentos pré-definidos, estes são capazes de comportarem-se de forma diferente e única;
- **Encapsulamento[8]:** aplicação estrita da ocultação da informação. Esta técnica tem como minimizar a dependência através da separação de objetos definidos através de interfaces externas, ou seja, auxilia a reduzir o acoplamento e a implementação de abstração de tipos de dados, de modo a promover e melhorar a manipulação destes parâmetros pertencentes ao objeto. Esta vantagem fornece suporte ao programador de proteger os dados internos daquele objeto de modo a promover a devida utilização, e funcionalidade correta, das propriedades, de modo a evitar retrabalho e possíveis ocorrências indesejadas através de atribuições não validadas para aquele objeto;
- **Reutilização[8]:** habilidade de reutilizar o sistema, ou partes dele, na construção de novos sistemas. Apesar de não ser algo pertencente aos componentes do paradigma OOP, esta característica pode ser encontrada em procedimentos onde estes componentes específicos, como classes, possam ser combinados e modificados para atender as devidas demandas de uma nova aplicação e para isso este paradigma pode ser essencial quando combinado a elementos de polimorfismo e herança; e
- **Extensibilidade[8]:** facilidade no qual o software do sistema tem de ser alterado ou modificado. Bem como a reutilização do software, ou de partes dele, em aplicações de outros sistemas corroboram em outra característica oriunda deste paradigma.

Logo, é notável as vantagens quanto o uso do paradigma de orientação à objetos e os futuros benefícios quanto utilização e aplicação no desenvolvimento de softwares. Todavia, nesta pesquisa venho trazer uma outra concepção e abordagem em que, estas características do OOP podem ser prejudiciais para manipulação de dados em procedimentos que exigem processamento com tempo de latência baixo devido aos conceitos de encapsulamento e organização dos dados na memória.

Capítulo 3 – funcionamento da memória dentro do ambiente computacional

Para abordar sobre o funcionamento da memória no ambiente computacional é necessário compreender alguns conceitos, como por exemplo *cache*. Desde os anos de 1980, o desenvolvimento dos microprocessadores recebeu inúmeras melhorias para melhor desempenho dentro do ambiente computacional. Entretanto, a evolução dos componentes para aumento de performance no acesso e uso da memória não cresceu na mesma proporção e isto proporcionou um buraco entre essas tecnologias, como pode ser observado na **figura 3.1**[9].

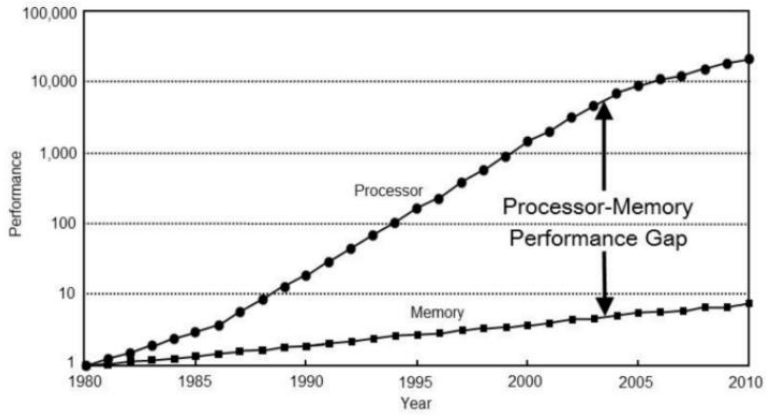

Figura 3.1 – comparativo de performance entre as tecnologias utilizadas nos processadores e memórias para ambientes computacionais. Fonte: BitSquid.

Este cenário motivou que grupos de pesquisas proporcionassem uma solução para este problema. Sendo assim, o conceito de *cache* é implementado no microprocessador, onde pequenas porções de memória são utilizadas para armazenar informações geridas pela CPU e que serão constantemente requisitadas para agilizar procedimentos de consulta a estes dados temporariamente salvos[10].

Agora, com essa nova abordagem, microprocessadores poderiam proporcionar um suprimento direto para acesso rápido da memória evitando acessar outros níveis, no qual proporciona maior tempo de busca e obtenção das informações contidas na memória. O funcionamento da memória é subdividido em níveis ou *layers*, nos quais cada nível é responsável em guardar dados que serão utilizados durante execução de softwares. Quanto maior o nível, mais tempo de latência ou maior o número de *clocks* serão exigidos para o processador para obter ou alterar aquele dado e esta característica de não encontrar uma informação na memória é conhecida como ***cache miss***[11] que pode comprometer a performance da aplicação. Um exemplo de uma operação de *cache miss*, pode ser demonstrada através da **figura 3.2** onde uma aplicação não encontra um dado dentro do *cache* e requisita uma consulta à uma base de dados.

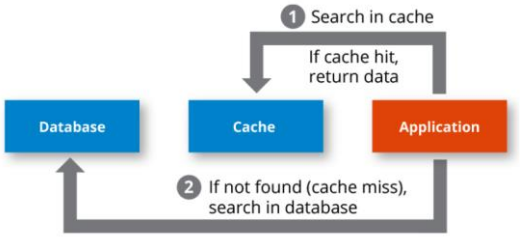

Figura 3.2 – demonstração de uma operação *cache miss* em um ambiente computacional por uma aplicação através de verificação *cache* e requisição à uma base de dados. Fonte: Hazelcast.

O processo unitário para adicionar mais *caches* dentro de um microprocessador pode ter consequências ou penalidades de tempo de resposta, com citado anteriormente através de vários ciclos ou *clocks*. Como mencionado pela patente publicada pela IBM em 2000[12], o *cache* L2 atua como intermediador entre o sistema de memória primário e os componentes à bordo de *caches*, de modo a proporcionar um espaço maior no armazenamento de dados ou instruções que os *caches* padrões, no qual pode ser observada na **figura 3.3**.

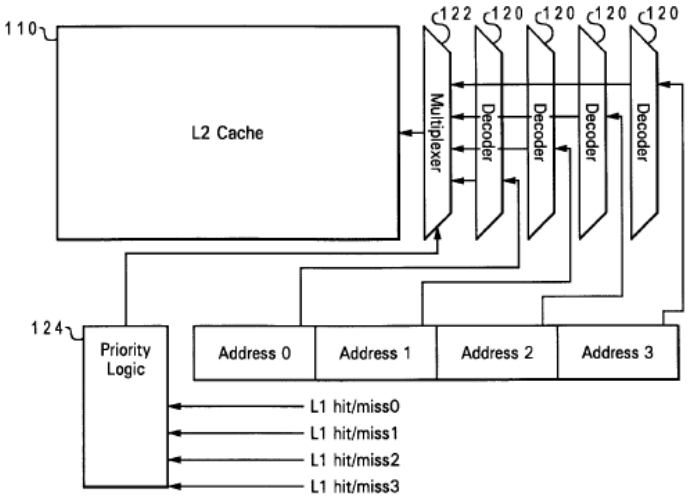

Figura 3.3 – diagrama representativo do *caches* L1 e L2 para acesso de memória sobreposta a múltiplos níveis dentro de uma arquitetura de microprocessador. Fonte: IBM Corporation.

Capítulo 4 – operações assíncronas para otimização de processos através dos fundamentos *multithreading*

Com a evolução dos sistemas operacionais foram possibilitadas operações nas quais auxiliavam a solucionar de produtividade nestes ambientes, como para uso científico ou comercial quanto ao uso adequado do processador[5]. Alguns problemas que eram evidenciados naquela época foram *CPU Bounded* e *I/O Bounded*, onde estavam relacionados a taxa de processamento e cálculo para instruções e tráfego de entrada-saída de dados através de periféricos pertencentes ao computador.

Contudo, os sistemas operacionais possibilitaram que mecanismos de sistemas multiprogramação junto do conceito de pseudoparalelismo permitissem que procedimentos de múltiplos processos fossem compartilhados simultaneamente. Essa concepção possibilitou o aumento da produtividade de acordo com o crescimento de quantidade de tarefas completas por unidade de tempo, também conhecida como a característica ***throughput***[5].

Para ser abordado procedimentos de *multithreading*, é necessário compreender que esta ferramenta está comumente presente em diversos *frameworks*, como por exemplo .NET Framework 4[13]. Como é citado pela API de referência da Microsoft sobre *multithreading*, a utilização da programação *multithread* permite que a aplicação promova uma interação de melhor qualidade ao usuário que a utiliza.

Alguns procedimentos de multitarefas ou *multithreading*, podem ser caracterizados em operações paralelas, nos quais dividem-se em processos paralelos[5]:

- **Independentes:** são processos paralelos que quando utilizam recursos completamente distintos, não se relacionam em disputas com outros processos;
- **Concorrentes:** são processos paralelos que quando pretendem utilizar um mesmo recursos, dependem de uma ação do sistema operacional para definir a ordem de utilização; e
- **Cooperantes:** é quando dois ou mais processos utilizam em conjunto, um mesmo recursos para completarem uma dada tarefa.

Logo, esta atribuição fornece suporte que processos hajam de forma independente com as características especificadas anteriormente, onde pode ser observado na **figura 4.1**[4]. Todavia, a utilização de procedimentos de multitarefas não é necessariamente um fator de aumento de performance, pois com o uso indevido ou em excesso pode proporcionar uma queda de qualidade no tempo de resposta de uma aplicação[4].

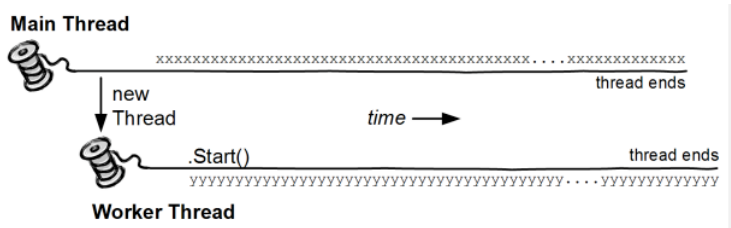

Figura 4.1 – representação de um *thread* em execução independente. Fonte: O'Reilly Media, Inc.

Sendo assim, em meio ao ambiente computacional para desenvolvimento de aplicações *server-side* de jogos, foi crucial a implementação destes fundamentos para proporcionar aumento de performance em procedimentos, processos ou tarefas que se caracterizavam como exaustivos para a plataforma computacional de modo a comprometer a integridade do ambiente ou aplicação.

Capítulo 5 – OOP *versus* DOD

Como foi abordado nos capítulos anteriores deste DVE, alguns fundamentos do paradigma de orientação à objetos fornece ao desenvolvedor de software em abstrair com melhor adequação de elementos pertencentes aquela situação, ou proposta estipulada, no mundo real, ou seja, os requisitos pertencentes a um objeto real tornam-se mais simples de implementar quando utilizando OOP.

Todavia, para fins de performance e maior velocidade na entrega de uma resposta pelo sistema através de consultas na memória, o uso do paradigma OOP não é o mais adequado quando situado em certas operações que exigem extremo processamento com o mínimo de tempo de espera possível.

Um dos motivos do paradigma OOP ser considerado lento, quanto a consulta de instruções ou informações na memória, é a sua estruturação, como pode ser observado na **figura 5.1**[14].

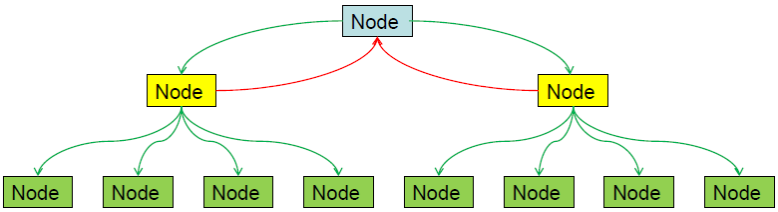

Figura 5.1 – representação hierárquica do paradigma OOP. Fonte: Sony Computer Entertainment Europe Research & Development Division.

Este paradigma não tem como ênfase organização dos dados na memória e isso prejudica operações que são consideradas críticas no meio do desenvolvimento de jogos. Entretanto, existem situações na quais o padrão de desenvolvimento DOD contribui para essa demanda de otimização em cenários bem específicos nos quais serão abordados nos próximos capítulos deste DVE.

Quando dentro do ambiente de desenvolvimento é importante notar que algumas premissas estão presentes e compactuam para decisões de utilizar ou não o paradigma OOP e começar a ter preocupação da estruturação dos dados pela aplicação no ambiente computacional, como: controle dos dados de entrada, variações de dados bruscas, padrões de dados, comportamento do hardware e compilador, comportamento dos dados na memória etc. Sendo assim, o padrão de desenvolvimento DOD tem como papel essencial introduzir alguns conceitos nos quais pequenos ajustes dentro as estruturas de dados e de código resultam em aumento considerável para melhor funcionamento da aplicação.

Capítulo 6 – os motivos por trás do DOD

Como citado no capítulo 5, OOP não leva em consideração a organização dos dados na memória e isso pode ser prejudicial para o devido funcionamento da aplicação, esse comportamento pode ser observado na **figura 6.1**[15].

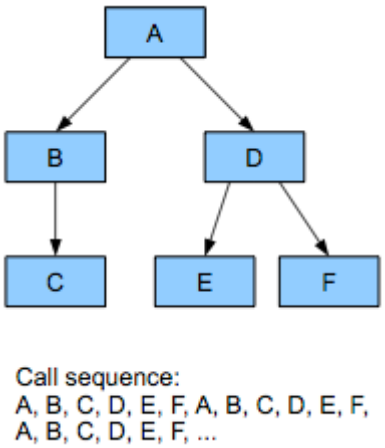

Figura 6.1 – representação dos dados através do paradigma OOP. Fonte: Medium.

Contudo, com a devida organização da estruturação dos dados, é possível otimizar as operações de consulta das informações na memória, como pode ser observado na **figura 6.2**[15].

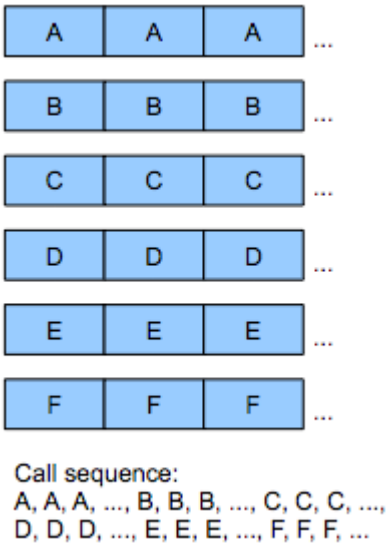

Figura 6.2 – representação dos dados através do padrão de desenvolvimento DOD. Fonte: Medium.

Um dos motivos da chamada da sequência estar organizada é que através do padrão de desenvolvimento DOD, a perspectiva de programação passa de objeto a dados, ou seja, os dados são organizados na memória e os procedimentos de leitura e processamento atuam de maneira conjunta com o processador evitando esforço para acessar essas informações. Outra vantagem de adotar esse padrão é a facilidade de implementações de paralelismo que estão presentes nos frameworks mais atuais de modo a obter vários núcleos de processamento simultâneo para agilizar as tarefas, veja **figura 6.3**[9].

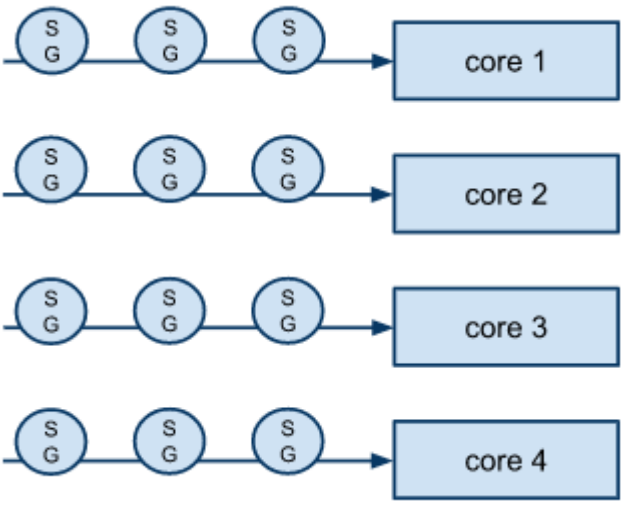

Figura 6.3 – representação do processamento de dados através do padrão de desenvolvimento DOD. Fonte: BitSquid.

Esta nova abordagem de levar em consideração a organização dos dados caracteriza o padrão de desenvolvimento DOD com os seguintes atributos:

- **Menor complexidade na implementação de operações de paralelização[16]:** devido a homogeneidade da organização dos dados na memória, procedimentos sequenciais tornam-se mais simples de executar e com isso, junto com a evolução do processamento multicores (vários núcleos de processamento), estas operações são mais simples;
- **Utilização de *cache*[14][16]:** com operações de *prefetching*[1] ou busca antecipada, é possível trabalhar com dados de modo a antecipar futuras operações de alteração e predição das informações, com isso ações de instruções que demandam mais tempo são evitadas de modo a otimizar o uso adequado da memória através dos níveis mais baixos, como *caches* L1 e L2[12];
- **Modularidade no código[16]:** após a organização dos dados, a compreensão do código torna-se mais simples devido ao baixo nível de dependência entre as partes do funcionamento daquele código, ou seja, operações futuras de manutenção são mais simples de executar; e
- **Facilidade para realização de testes[16]:** devido a modularidade no código, operações de teste unitário tornam-se mais simples de serem realizadas, auxiliando o processo de desenvolvimento daquela funcionalidade para o software.

Capítulo 7 – a devida utilização DOD em operações de alta performance e baixa latência de resposta em aplicações *server-side* de jogos

Atualmente, servidores de jogos multiplayer, também conhecidos pela atribuição *Massively Multiplayer Online* (MMO), são geridos por aplicações nas quais realizam simulações de ambientes virtuais experienciados dentro do jogo pelos jogadores. Todavia, certos processos internos ou paralelizados por um cluster de processos semelhante a um gateway IPC, realizam operações de alto processamento como cálculos exaustivos ou mesmo atualização dinâmica de dados na memória da aplicação, como demonstrado no **Apêndice A**.

Em certos cenários, quando essas rotinas internas não são otimizadas, podem resultar em falhas catastróficas em cascata devido sua complexidade e dependência de inúmeros recursos compartilhados na memória. Entretanto, devido a criticidade que desempenham, estas rotinas estão interligadas no devido funcionamento de uma ou várias outras rotinas dentro de uma aplicação para jogos MMO, justificando o possível efeito sequencial em caso de falta de otimização destas funcionalidades.

Como mencionado no capítulo 5, nem sempre a utilização do paradigma OOP torna-se a melhor opção para o programador quando tratando-se destas situações específicas citadas anteriormente. Sendo assim, quando aplicado o padrão de desenvolvimento DOD nestas seguintes condições como: previsibilidade dos dados de entrada nos processos de ciclo infinito dentro das rotinas; possibilidade de implementar tecnologias para processamento multitarefa pelo framework ou linguagem utilizado; regularidade dos dados dentro das rotinas, onde estes não são referentes de outras tarefas paralelas

[1] Algumas linguagens de programação possuem ferramentas nativas para operações de *prefetching*, por exemplo na linguagem C++, existe uma instrução chamada **dcbt** ou *Data Cache Block Touch*[17], no qual tem como papel habilitar uma requisição de busca em bloco no *cache* de modo a antecipar o carregamento daquela informação pelo seu endereço de memória.

dentro do mesmo ambiente; devido conhecimento das operações que serão desempenhadas durante os processos de ciclo contínuo dentro das rotinas.

O padrão de desenvolvimento DOD, tem como vantagens principais nessa situação:

- **Identificação de padrões:** caso exista um padrão de dado a ser seguido dentro da rotina, este pode ser aplicado em uma classe Manager[9] para implicar numa previsibilidade de predição daquele dado durante os ciclos de processamento da rotina;
- **Sequência homogênea de dados:** com a utilização de estruturas de sequência de dados como vetores, listas ou coleções (preferencialmente nativos do framework ou linguagem utilizados), a ordenação dos dados em endereços na memória torna-se mais efetivo e otimizado, facilitando as operações de iteração entre esses data sets;
- **Processamento ordenado dos data sets:** proveniente do item anterior, a utilização da sequência homogênea de dados assiste nas operações de iteração entre as sequências, melhorando a iteração para cada registro de dado referenciado na memória, nos quais podem ser previstos através de uma classe Manager, evitando exceções de *deadlock* ou *thread-unsafe*, pois os dados estão pertencentes ao mesmo domínio e contexto da thread principal da rotina;
- **Remoção de chamadas ou invocações virtualizadas:** evitar chamada de métodos virtuais, auxilia nas operações de consulta daquele tipo de dado na memória. Entretanto, apesar de violar os conceitos de isolamento dos dados provenientes do paradigma OOP, esta operação é primordial para que atualizações nos endereços de memória de cada referência seja alterado diretamente e não utilize mediadores para assistir essa operação, pois é custosa dentro de rotinas críticas como estas vistas em aplicações de servidores de jogos MMO;
- **Conversão linear e processamento multicores:** unindo as definições da sequência homogênea de dados e o processamento ordenado dos data sets, resultam em uma conversão linear de processamento dos dados dentro da rotina de modo a otimizar a performance durante a execução naquele ciclo. Outro fator que interessante em aplicar neste item é o processamento paralelo para agilizar a iteração dos objetos, esta operação é mais simplificada por não utilizar o paradigma OOP e os dados são diretamente acessados e referenciados para cada iteração paralela, sendo facilmente previsíveis; e
- **Uso de layers de memória do processador:** algumas operações específicas podem ser utilizadas para "aquecer" os níveis de acesso de memória do processador, que por desconhecimento do programador não são devidamente utilizados para auxiliar em consultas via cache para aquele dado referenciado na memória que será consultado e possivelmente atualizado com um novo valor. Quando uma operação de *prefetching* é aplicada, esta pode ser combinada em operações de conversão linear de consulta de cada dado na memória, com isto é possível validar endereços da memória inválidos e evitar verificações desnecessárias durante o ciclo de processamento da rotina.

Contudo, o programador deve ser moderado na utilização de certos recursos oriundos do padrão de desenvolvimento DOD, como: operações de processamento multicores e *prefetching*. Em operações de processamento multicores, a utilização em demasia dessas funcionalidades pode danificar o funcionamento operacional da rotina e perda no controle dos dados durante os ciclos sequenciais da própria rotina que desempenha uma operação de simulação dentro do jogo. Em operações de *prefetching*, o uso deste recurso pode denegrir o devido funcionamento do processador quanto utilizado em operações *multithreading*.

Enfim, o paradigma OOP ainda pode fazer parte de algumas consultas não relacionados internamente na rotina, de modo a evitar o contato direto com os ciclos de processamento crítico. Sendo assim, o padrão de desenvolvimento DOD atende os requisitos básicos para que operações exaustivas, mas com a existência de certas condições, possam ser processadas mais rapidamente e evitar o uso de inúmeros recursos provenientes do hardware ou de serviços e módulos assistivos do ambiente computacional aplicado.

Capítulo 8 – as aplicações do padrão de desenvolvimento DOD pela empresa Unity em sua nova tecnologia, o Unity DOTS

Em 2019, a empresa Unity lançou sua nova tecnologia chamada Unity DOTS (Data-Oriented Technology Stack)[18] com o intuito de otimizar a performance de jogos através de processamento *multithreading* e obter ampla utilização de recursos dos processadores como multicores utilizando códigos escritos na linguagem C# nos quais são referenciados durante a compilação para funções nativas em C++[19].

Figura 8.1 – os pilares do Unity DOTS: Entity Component System (ECS)[21], C# Job System[22] e Burst Compiler[23]. Fonte: Unity Learn.

A utilização desta tecnologia no desenvolvimento de jogos permite incluir devidamente o processamento paralelo de dados de modo a otimizar o funcionamento de projetos Unity. Esse aumento substancial de performance em aplicações de jogos é devido a implementação de fundamentos e abordagens oriundos do padrão de desenvolvimento DOD[20]. O Unity DOTS é uma combinação de novas tecnologias para o Unity, que utiliza um conceito diferente da tradicional de analisar certas operações contrários ao paradigma OOP.

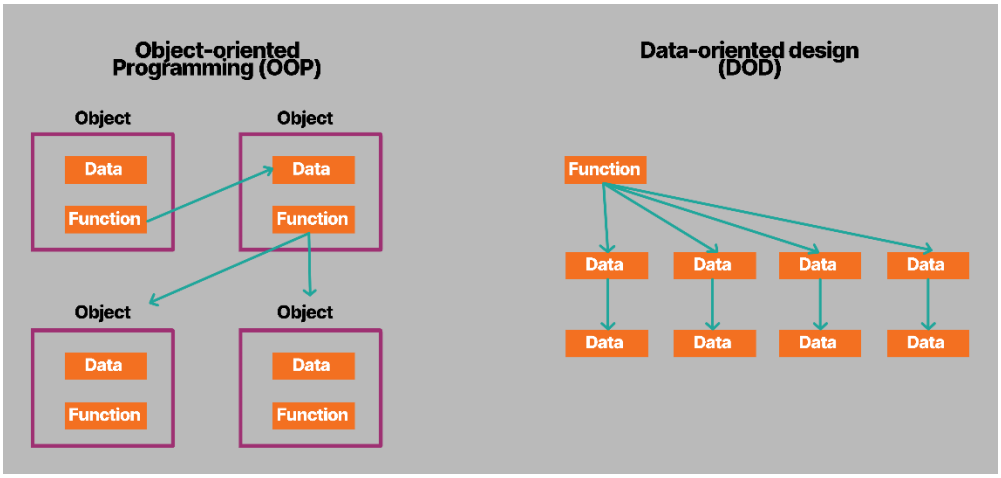

Figura 8.2 – assimilação entre o paradigma OOP e o padrão de desenvolvimento DOD. Fonte: Unity Learn.

Um dos principais motivos em aplicar o padrão de desenvolvimento DOD em projetos Unity é de utilizar seus fundamentos de modo a providenciar um ganho considerável em performance nas aplicações evitando os contextos caracterizantes do paradigma OOP, chamados de *cache missing*[24].

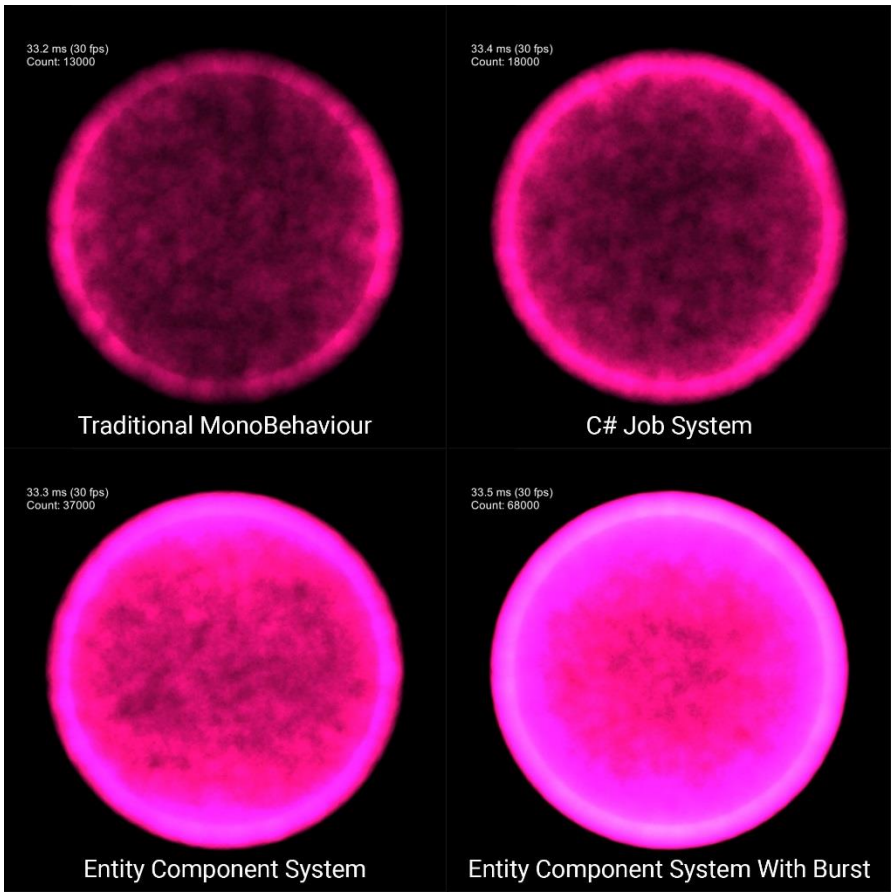

Figura 8.3 – comparativo de performance ao utilizar o Unity DOTS em um projeto Unity. Fonte: Mike Geig.

DOD foi embutido dentro do sistema ECS[19][24] no Unity DOTS, onde:

- **Component:** é composto por dados referenciados em endereços na memória para acesso da informação;
- **Entity:** é representado por uma "coleção" de componentes ou conjunto agregado de dados referenciados para cada informação;
- **System:** é o ambiente que executa as funções que serão iteradas para o agregado de entidades compostas pelos componentes formados por dados referenciados.

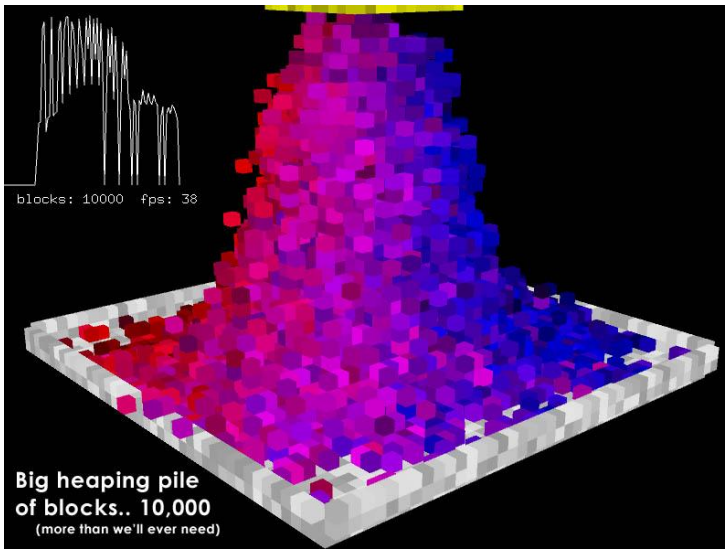

Figura 8.4 – gráfico demonstrativo de ganho de performance ao utilizar o sistema ECS em um projeto Unity. Fonte: Mark Zifchock.

Com isso, a empresa continua fornecendo suporte a esta tecnologia e aposta que a arquitetura adotada para essa abordagem é a mais adequada para atender aos recursos computacionais existentes em nossa geração. Todavia, essa tecnologia ainda continua em desenvolvimento e a empresa Unity vagarosamente está adaptando seus sistemas para aplicar definitivamente nas plataformas para desenvolvimento de jogos.

Ao mesmo tempo, a empresa acredita que este padrão de desenvolvimento, cuja abordagem principal é o devido controle dos dados dinamicamente através dos recursos computacionais existentes, será o futuro da Indústria de jogos para simulações em tempo real 3D[19], como referenciado em jogos populares mundialmente de outros estúdios: *Overwatch*[25] (Blizzard) e *The Witcher 3*[26] (CD PROJEKT RED).

Conclusão

Após esta análise detalhada sobre quais condições são mais adequadas em aplicar certos tipos de padrões durante o desenvolvimento de algoritmos para jogos, pode ser crucial no desempenho a longo prazo, como minimização de gastos com recursos computacionais extras, por exemplo. Entretanto, o programador deve ter a consciência de que o paradigma OOP pode trazer malefícios durante o desenvolvimento, não no quesito de coesão para compreensão do código, mas a respeito da estruturação dos dados que serão constantemente referenciados por instâncias de objetos.

Certos procedimentos no início do desenvolvimento de um projeto de uma rotina crítica para um ambiente de servidor de jogos MMO, deverão exigir alto desempenho durante seu processamento para o funcionamento adequado.

Como foi demonstrado durante alguns testes nos apêndices, **Anexo A** e **Anexo B**, é evidente os benefícios ao utilizar o padrão de desenvolvimento DOD devido: a agilidade de processamento que estas rotinas exigem; utilização de níveis de memória do processador referenciados nas operações de *cache*; possibilidade de processamento *multithreading* e multicores; processamento modular de sequência de dados através de coleções de data sets por endereçamentos na memória dos objetos; e a possibilidade de aplicação em outras áreas no desenvolvimento de jogos como mencionado na tecnologia desenvolvida pela empresa Unity, o Unity DOTS.

Referências Bibliográficas

[1]. "Dynamic Memory Allocation", Department of Computer Science, Florida State University, disponível em https://www.cs.fsu.edu/~myers/c++/notes/dma.html ;
[2]. "Memory allocation and deallocation (in relation to OOP)", University of Washington, disponível em http://courses.washington.edu/css342/zander/css332/memoryoop.html ;
[3]. "Dynamic Memory Allocation", Central Connecticut State University, disponível em https://chortle.ccsu.edu/assemblytutorial/chapter-33/ass33_3.html#:~:text=The%20program%20then%20uses%20this,part%20of%20the%20data%20segment. ;
[4]. "Threading in C#", Albahari, J., O'Reilly Media, Inc., disponível em http://www.albahari.com/threading/ ;
[5]. "Notas sobre Sistemas Operacionais – Sistemas operacionais: processamento de dados". Jandl, P. Jr., fev. 2004, versão 1.1;
[6]. "Object Oriented Programming of the finite element method", R.I. Mackie, 1992, Elsevier Science Publishers Ltd. ;
[7]. "Object Oriented Programming and Numerical Methods", R.I. Mackie, 2008, John Wiley & Sons, Ltd.;
[8]. "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", M. Josephine, ago. 1987, Department of Computer Science, Columbia University ;
[9]. "Practical Examples in Data Oriented Design", Frykholm, N., BitSquid, disponível em https://docs.google.com/presentation/d/17Bzle0w6jz-1ndabrvC5MXUIQ5jme0M8xBF71oz-0Js/present?slide=id.i0 ;
[10]. "How L1 and L2 CPU Caches Work, and Why They're an Essential Part of Modern Chips", Hruska, J., Extreme Tech, disponível em https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips ;
[11]. "What Is a Cache Miss?", Hazelcast, Inc. disponível em https://hazelcast.com/glossary/cache-miss/ ;
[12]. "Multiple level cache memory with overlapped L1 and L2 memory access", Dhong, S. H.; Hofstee, H. P.; Meltzer, D.; Falls, W.; Sillberman, J. A., out. 2000, IBM Corporation ;
[13]. "Managed Threading", Microsoft, disponível em https://docs.microsoft.com/en-us/dotnet/standard/threading/ ;
[14]. "Pitfalls of Object Oriented Programming", Albrecht, T., Research & Development Division, Sony Computer Entertainment Europe ;
[15]. "Data-Oriented vs Object-Oriented Design", Mines, J., Medium, disponível em https://medium.com/@jonathanmines/data-oriented-vs-object-oriented-design-50ef35a99056 ;
[16]. "Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)", Llopis,. N., Games from Within, disponível em http://gamesfromwithin.com/data-oriented-design ;
[17]. "dcbt (Data Cache Block Touch) instruction", IBM Knwledge Center, disponível em https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/assembler/idalangref_dcbt_instrs.html ;
[18]. "DOTS – Unity's new multithreaded Data-Oriented Technology Stack", disponível em https://unity.com/dots ;
[19]. "What is Unity's new Data-Oriented Technology Stack (DOTS)", disponível em https://hub.packtpub.com/what-is-unitys-new-data-oriented-technology-stack-dots/ ;
[20]. "What is DOTS and why is it important?", disponível em https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important ;
[21]. "Entity Component System", disponível em https://learn.unity.com/tutorial/entity-component-system ;
[22]. "C# Job System", disponível em https://docs.unity3d.com/Manual/JobSystem.html ;
[23]. "Burst User Guide", disponível em https://docs.unity3d.com/Packages/com.unity.burst@1.3/manual/index.html ;
[24]. "On DOTS: Entity Component System", disponível em https://blogs.unity3d.com/2019/03/08/on-dots-entity-component-system/ ;
[25]. "Overwatch 2", disponível em https://overwatch2.playoverwatch.com/en-us/trailer ;
[26]. "The Witcher 3 Archives – CD PROJEKT RED", disponível em https://en.cdprojektred.com/category/the-witcher-3/ .

Glossário

Cache – é uma pequena porção da memória que pode armazenar informações ou dados ou instruções que serão utilizadas pela CPU[10].

Cache miss – é uma operação no qual uma informação não é encontrada naquele contexto e assim recorre aos próximos níveis de memória, proporcionando perda de performance na aplicação através de ciclos de processamento extra realizados pelo processador[10].

DOD – *Data-Oriented Design*, é um padrão de desenvolvimento para uma abordagem de otimização mais eficiente do uso de CPU mediante o desenvolvimento de jogos. Fonte: "Data-oriented design", Wikipedia.

Multithreading – são ferramentas disponibilizadas através de *frameworks* que possibilitam a utilização dos recursos computacionais do ambiente, de modo a possibilitar processos de paralelismo para assim possibilitar aumento de performance na execução de aplicações[4].

OOP – *Object-Oriented Programming*, é um paradigma de orientação à objeto utilizado na programação baseado nos conceitos de objeto, nos quais estes podem ser acessados e conter dados ou informações e assim serem facilmente manipulados através de procedimentos, por exemplo polimorfismo, encapsulamento etc. Fonte: "Object-oriented programming", Wikipedia.

Processo computacional – é uma atividade que ocorre no meio computacional, usualmente possuindo um objetivo definido, tendo duração finita e utilizando uma quantidade limitada de recursos computacionais, onde é caracterizado pelo termo tarefa ou *task*[5].

Processos paralelos – são executados ao mesmo tempo, sendo caracterizados como independentes, concorrentes e cooperantes[5].

Processos paralelos concorrentes – são processo que desejam utilizar uma região crítica e assim dependem de uma ação do sistema operacional para definir a ordem de utilização[5].

Região crítica – *critical section*, ocorre quando um dado recurso computacional só pode ser utilizado por um único recurso simultaneamente[5].

Threads – são fluxos independentes de execução, onde pertencem a um mesmo processo, ou seja, requerem menos recursos de controle por parte do sistema operacional[5].

Anexos

Anexo A – DOD e OOP Benchmarking

Autor: Stanislav Denisov (Budapest, Hungary).
Descrição: este algoritmo representa um comparativo entre o paradigma OOP e o padrão de desenvolvimento DOD, foram utilizados três processadores durante os testes de performance em milissegundos (ms) para 10 milhões de iterações, segundo a **tabela A.1**.

Processador	DOD	OOP
AMD FX-4300	107.224ms	1850.76ms
AMD Ryzen 5 1400	81.57ms	1378.33ms
Intel Core i7-4700MQ	151.179ms	3964.19ms

Tabela A.1 – comparativo realizado por Stanislav Denisov.

```
#include <iostream>
#include <Windows.h>

using namespace std;

#define ITERATION_COUNT 10000000

double PerformanceFrequency = 0.0;
__int64 CounterStart = 0;

void StartCounter() {
    LARGE_INTEGER performanceCount;

    if (!QueryPerformanceFrequency(&performanceCount))
        cout << "QueryPerformanceFrequency failed!" << endl;

    PerformanceFrequency = double(performanceCount.QuadPart) / 1000.0;

    QueryPerformanceCounter(&performanceCount);

    CounterStart = performanceCount.QuadPart;
}

double GetCounter() {
    LARGE_INTEGER performanceCount;

    QueryPerformanceCounter(&performanceCount);

    return double(performanceCount.QuadPart - CounterStart) / PerformanceFrequency;
}
```

```
struct Object {
    int a;
    int b;
    int c;

    int Foo() {
        return a + b + c;
    }
};

struct FragmentedObject {
    int a;
    char f1[64];
    int b;
    char f2[128];
    int c;
    char f3[256];
    int d;

    int Foo() {
        d = a + b + c;

        return d;
    }
};

struct DerivedObject : Object {
    char o1[64];
    char o2[128];
    char o3[256];
};

int Foo(int a, int b, int c) {
    return a + b + c;
}

int main() {
    auto a = new int[ITERATION_COUNT];
    auto b = new int[ITERATION_COUNT];
    auto c = new int[ITERATION_COUNT];
    auto o = new struct Object[ITERATION_COUNT];
    auto f = new struct FragmentedObject[ITERATION_COUNT];
    auto d = new struct DerivedObject[ITERATION_COUNT];

    // DOD
    StartCounter();
```

```
    for (int i = 0; i < ITERATION_COUNT; ++i) {
        Foo(a[i], b[i], c[i]);
    }

    cout << GetCounter() << " (DOD, per array of data iterations, perfect utiliza-
tion)" << endl;

    // OOD
    StartCounter();

    for (int i = 0; i < ITERATION_COUNT; ++i) {
        o[i].Foo();
    }

    cout << GetCounter() << " (OOD, per structure of data iterations, perfect utiliza-
tion)" << endl;

    // OOD (Fragmented)
    StartCounter();

    for (int i = 0; i < ITERATION_COUNT; ++i) {
        f[i].Foo();
    }

    cout << GetCounter() << " (OOD, per structure of data iterations, fragmented utiliza-
tion)" << endl;

    // OOP (Inheritance)
    StartCounter();

    for (int i = 0; i < ITERATION_COUNT; ++i) {
        d[i].Foo();
    }

    cout << GetCounter() << " (OOP, per structure of data iterations, derived utiliza-
tion)" << endl;

    delete[] a;
    delete[] b;
    delete[] c;
    delete[] o;
    delete[] f;
    delete[] d;
}
```

Anexo B – análise de cache missing entre OOP e DOD

Autor: desconhecido.
Descrição: este algoritmo representa um comparativo entre o paradigma OOP e o padrão de desenvolvimento DOD no quesito de análise entre a quantidade de *cache missing* durante as iterações entre as instâncias de objetos alocadas no nível de memória L1 processador para 10 iterações, o resultado do teste foi fornecido na **tabela A.2**.

DOD	OOP
9	16

Tabela A.2 – comparativo entre OOP e DOD quanto ao número de *cache missing* no nível de memória L1 do processador.

```
static string buffer = "";
static int cacheMisses = 0;

const int numberOfValuesCached = 4;
const int maxNumberOf32BitsElements = 16; // 64B L1 cache

class MemoryThing { }

class Float : MemoryThing {
    private readonly float value;

    public Float (float value) {
        this.value = value;
    }

    public static implicit operator Float (float value) {
        return new Float (value);
    }

    public static Float operator + (Float first, Float second) {
        return new Float (first.value + second.value);
    }

    public static Float operator - (Float first, Float second) {
        return new Float (first.value - second.value);
    }

    public override string ToString () {
        return "{" + value + "}";
    }
}

class Int : MemoryThing {
```

```
    private readonly int value;

    public Int (int value) {
        this.value = value;
    }

    public static implicit operator Int (int value) {
        return new Int (value);
    }

    public static Int operator + (Int first, Int second) {
        return new Int (first.value + second.value);
    }

    public static Int operator - (Int first, Int second) {
        return new Int (first.value - second.value);
    }

    public override string ToString () {
        return "{" + value + "}";
    }
}

class Entity : MemoryThing {
    public Float x, y;
    public Int sprite;

    public Entity (float x, float y, int sprite) {
        this.x = x;
        this.y = y;
        this.sprite = sprite;
    }

    public override string ToString () {
        return x.ToString () + y.ToString () + sprite.ToString ();
    }
}

class Entities {
    public List<MemoryThing> xs = new List<MemoryThing> ();
    public List<MemoryThing> ys = new List<MemoryThing> ();
    public List<MemoryThing> sprites = new List<MemoryThing> ();

    public void NewEntity (float x, float y, int sprite) {
        xs.Add (new Float (x));
        ys.Add (new Float (y));
```

```
            sprites.Add (new Int (sprite));
        }
    }

    static void StoreValueInMemory (List<MemoryThing> objects, int pos) {
        // Add new values
        int cachedValues = 0;
        for (int i = pos; i < objects.Count; ++i) {
            string values = objects[i].ToString ();

            int numValues = values.Count (f => f == '{');
            while (cachedValues < numberOfValuesCached) {
                int indexToCopyTo = values.IndexOf ('}');
                if (indexToCopyTo >= 0) {
                    string newValue = values.Substring (0, indexToCopyTo + 1);
                    if (buffer.IndexOf (newValue) < 0) {
                        buffer += newValue;
                        cachedValues++;
                    }
                    values = values.Substring (indexToCopyTo + 1);
                } else {
                    break;
                }
            }

            if (cachedValues >= numberOfValuesCached) {
                break;
            }
        }

        int count = buffer.Count (f => f == '{');

        while (count > maxNumberOf32BitsElements) {
            // Remove oldest values
            int indexToDeleteTo = buffer.IndexOf ('}');
            if (indexToDeleteTo >= 0) {
                buffer = buffer.Substring (indexToDeleteTo + 1);
                count--;
            } else {
                break;
            }
        }

        Console.WriteLine (string.Format ("Stored in memory   : {0}", buffer));
    }
```

```
static void GetValueInMemory (List<MemoryThing> objects, int pos) {
    MemoryThing yourObject = objects[pos];
    string values = yourObject.ToString ();
    int numValues = values.Count (f => f == '{');

    for (int i = 0; i < numValues; ++i) {
        int indexToCopyTo = values.IndexOf ('}');
        if (indexToCopyTo >= 0) {
            string newValue = values.Substring (0, indexToCopyTo + 1);
            if (buffer.IndexOf (newValue) != -1) {
                // Keep less recently used at the beginning and the most re-
cently used at the end of the list
                //buffer = buffer.Replace(newValue,"");
                //buffer += newValue;

                values = values.Substring (indexToCopyTo + 1);

                //Console.WriteLine(string.Format("Found in memory    : {0}", newValue));
            } else {
                //Console.WriteLine(string.Format("Not found in memory: {0}", newValue));
                cacheMisses++;

                StoreValueInMemory (objects, pos);
            }
        } else {
            break;
        }
    }
}

public static void Main (string[] args) {

    int numberOfEntities = 10;
    List<MemoryThing> entities = new List<MemoryThing> ();
    for (int i = 0; i < numberOfEntities; ++i) {
        entities.Add (new Entity ((float) i + 10000.0f, (float) i + 200000.0f, i));
    }

    // Lets say we update the positions...
    for (int i = 0; i < numberOfEntities; ++i) {
        GetValueInMemory (entities, i);
    }

    // ... and somewhere else in the code we increment the frame for their animation
    for (int i = 0; i < numberOfEntities; ++i) {
        GetValueInMemory (entities, i);
```

```
    }

    Console.WriteLine (string.Format ("Number of cache misses for OOP: {0}", cacheMisses));

    cacheMisses = 0;
    buffer = "";
    Entities DoDEntities = new Entities ();
    for (int i = 0; i < numberOfEntities; ++i) {
        DoDEntities.NewEntity ((float) i + 10000.0f, (float) i + 200000.0f, i);
    }

    // Lets say we update the positions...
    for (int i = 0; i < numberOfEntities; ++i) {
        GetValueInMemory (DoDEntities.xs, i);
        GetValueInMemory (DoDEntities.ys, i);
    }

    // ... and somewhere else in the code we increment the frame for their animation
    for (int i = 0; i < numberOfEntities; ++i) {
        GetValueInMemory (DoDEntities.sprites, i);
    }

    Console.WriteLine (string.Format ("Number of cache misses for DOD: {0}", cacheMisses));
}
```

Apêndices

Apêndice A – código demonstrativo de uma rotina crítica de uma aplicação de um servidor de jogos MMO

Autores: Jay Russell (Department of Computing – Imperial College London, Inglaterra) e Nádio Dib (Departamento de TI – UniProjeção, Brasil).
Descrição: este algoritmo representa uma rotina característica de operações cíclicas e sequenciais em uma aplicação de um servidor de jogos MMO, onde comumente referências de objetos através de propriedades e métodos oriundos do paradigma OOP são evidenciados de modo a atualizar dinamicamente novas e novas instâncias de objetos durante o *runtime* da aplicação.

```
public bool Update (double dt) {
    UpTime += dt;
    UntilNewTick += dt;

    while (!PendingIncomingMessages.IsEmpty)
        if (PendingIncomingMessages.TryDequeue (out var multiplexerMessage))
            HandleMessage (multiplexerMessage);

    // do player update every chance we can
    foreach (var clientConnection in ClientConnections.Values) {
        if (clientConnection.NoUpdateAckReceived)
            continue;

        var tiles = new List<GroundTileData> ();
        var newObjs = new List<ObjectData> ();
        var drops = new List<int> ();

        if (!clientConnection.addSelf) {
            newObjs.Add (new ObjectData (782, new ObjectStatusData (clientConnection.Ob-
jectId, clientConnection.X, clientConnection.Y, new List<StatData> () {
                new StatData (StatData.ACCOUNT_ID_STAT, "0"),
                    new StatData (StatData.NAME_STAT, "Slendergo"),
                    new StatData (StatData.SPEED_STAT, (int) (87 * 1.5))
            })));
            clientConnection.addSelf = true;
        }

        if (!clientConnection.IgnoreTileUpdate) {
            if (!clientConnection.FirstAckReceived || Math.Abs (clientConnection.X 0 cli-
entConnection.PrevX) > 0 || Math.Abs (clientConnection.Y - clientConnection.PrevY) > 0) {
                var s2w = new Stopwatch.StartNew ();

                foreach ((int x, int y) in clientConnection.SightPoints) {
```

```
                    var dx = (int) (x + clientConnection.X);
                    var dy = (int) (y + clientConnection.Y);
                    if (db < 0 || dy < 0 || dx >= clientConnection.Game.Width || dy >= cli-
entConnection.game.Height)
                        continue;

                    var tile = clientConnection.Game.Tiles[dx, dy];
                    if (clientConnection.SeenTiles[dx, dy] >= tile.UpdateCount)
                        continue;

                    SeenTiles[dx, dy] = tile.UpdateCount;

                    tiles.Add (new GroundTileData (dx, dy, tile.GroundType));
                }

                Console.WriteLine ($"{tiles.Count} tiles took: {s2w.Elapsed} {s2w.ElapsedMil-
liseconds} ms");
            }

            clientConnection.IgnoreTileUpdate = true;
        }

        if (tiles.Count > 0 || newObjs.Count > 0 || drops.Count > 0) {
            NetworkMultiplexer.Instance.PrepareMessage (new UpdateMes-
sage (tiles, newObjs, drops), clientConnection.ConnectionId);
            clientConnection.NoUpdateAckReceived = true;
        }
    }

    // game logic first
    if (UntilNewTick >= 0.2) {
        // do logic before newtick is sent to update everything first
        var sw = Stopwatch.StartNew ();

        var cleanup = new List<GameObject> ();
        foreach (var go in TempTestGameObjects.Values)
            if (!go.Update (dt))
                cleanup.Add (go);

        // do player newtick after all logic is done
        foreach (var clientConnection in ClientConnections.Values)
            clientConnection.HandleNewTick (dt);

        foreach (var go in cleanup)
            TempTestGameObjects.TryRemove (go.ObjectId, out _);
```

```
        Console.WriteLine ($"Update handle took: {sw.ElapsedMillisec-
onds} | {sw.Elapsed} with : {ClientConnections.Count} connections");

        UntilNewTick = 0.0;
    }

    while (!PendingIncomingMessages.IsEmpty)
        if (PendingIncomingMessages.TryDequeue (out var bytes))
            NetworkMultiplexer.PublishMessage (bytes);

    // todo make a type variable instead
    if (IdName != "Nexus")
        if (UpTime >= 300 && CanShutdown ())
            return false;

    return true;
}
```