UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MARCOS VINÍCIUS DE LIMA VENÂNCIO

DESEMPENHO DO DESIGN ORIENTADO A DADOS COM UNITY DOTS

QUIXADÁ

2023

MARCOS VINÍCIUS DE LIMA VENÂNCIO

DESEMPENHO DO DESIGN ORIENTADO A DADOS COM UNITY DOTS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientadora: Prof. Dra. Paulyne Matthews Jucá.

QUIXADÁ

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

V561d Venâncio, Marcos Vinícius de Lima.
Desempenho do Design Orientado a Dados com Unity DOTS / Marcos Vinícius de Lima Venâncio. – 2023.
56 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2023.
Orientação: Profa. Dra. Paulyne Matthews Jucá.

1. Desenvolvimento de Jogos. 2. Design Orientado a Dados. 3. Design Orientado a Objetos. 4. Unity DOTS. 5. Desempenho. I. Título.

CDD 004

MARCOS VINÍCIUS DE LIMA VENÂNCIO

DESEMPENHO DO DESIGN ORIENTADO A DADOS COM UNITY DOTS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em:

BANCA EXAMINADORA

Prof. Dra. Paulyne Matthews Jucá (Orientadora)

Prof. Dr. Arthur de Castro Callado
Universidade Federal do Ceará (UFC)

Prof. Dr. Wladimir Araujo Tavares
Universidade Federal do Ceará (UFC)

À minha família, por sua capacidade de acreditar em mim e investir em mim. Mãe, seu cuidado e dedicação foi que deram, em alguns momentos, a esperança para seguir. Pai, sua presença significou segurança e certeza de que não estou sozinho nessa caminhada.

RESUMO

A eficiência é uma busca eterna no mundo do desenvolvimento de *software*, principalmente em cenários extremos como o de jogos. O Design Orientado a Dados (DOD) surge como uma abordagem alternativa ao já consolidado Design Orientado a Objetos (DOO), provendo melhores caminhos para alcançar resultados de alto desempenho. Este trabalho busca comparar o desempenho dessas duas abordagens usando o *Unity DOTS*, que é uma série de pacotes adicionados à *engine Unity* que disponibilizam meios de programar jogos usando o Design Orientado a Dados. Um experimento-jogo foi criado tanto na versão DOD quanto na DOO e então submetido a vários testes em três *PCs* de configurações diferentes. Os resultados mostraram um ganho expressivo na abordagem DOD, entretanto em cenários onde o PC possui um gargalo na GPU, o ganho cobriu somente a eficiência de uso do hardware, não tendo impacto nos FPS (quadros por segundo), uma vez que o fluxo para a geração do *frame* é atrasado pela GPU, e não pela carga de trabalho na CPU.

Palavras-chave: Desenvolvimento de Jogos; Design Orientado a Dados; Design Orientado a Objetos; Unity DOTS; Desempenho.

ABSTRACT

The efficiency is an eternal quest in the world of software development, particularly in extreme scenarios such as games. Data-Oriented Design (DOD) has emerged as an alternative approach to the well-established Object-Oriented Design (OOD), providing better paths to achieve high-performance results. This work aims to compare the performance of these two approaches using Unity DOTS, which is a series of packages added to the Unity engine that enables game programming using Data-Oriented Design. Two versions of an experimental game were created, one using OOD and the other using DOD, and were subsequently subjected to tests on three PCs with varying hardware configurations. The results showed a significant performance gain with the DOD approach. However, in scenarios where the PC experiences a GPU bottleneck, the gain is limited to improved hardware utilization and does not impact the FPS (frames per second), as the frame generation is delayed by the GPU rather than CPU workload.

Keywords: Data-Oriented Design; Object-Oriented Design; Game Development; Unity DOTS; Performance;

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AoS	*Array of Structures*
CIL	*Common Intermediate Language*
CLR	*Common Language Runtime*
DOD	Design Orientado a Dados
DOO	Design Orientado a Objetos
DOTS	*Data-Oriented Technology Stack*
ECS	*Entity Component System*
FPS	Taxa de quadros por segundo
NPC	Personagem não jogável
SoA	*Structure of Arrays*

SUMÁRIO

1 INTRODUÇÃO

Desempenho é algo muito requisitado em jogos digitais, que precisam ser cada vez mais complexos, demandando assim custo em otimização pela equipe de desenvolvedores e hardware pela parte do cliente. O Design Orientado a Dados (DOD) surge como uma abordagem alternativa que consiste em se aproximar mais do hardware e dados em si, se distanciando da abstração provida pelo já consolidado Design Orientado a Objetos (DOO), prometendo assim ganhos de performance, um *multithreading* mais fácil de se implementar, além de melhorias na qualidade de código produzido (LLOPIS, 2009).

Algumas *engines* já disponibilizam formas para utilizar DOD, seja por meio de extensões ou nativamente, comumente utilizando o *pattern Entity Component System* (ECS). A *Unity* provê o *Data-Oriented Technology Stack* (DOTS) – uma *stack* orientada a dados, *multithread* e de alto desempenho –, tendo como premissa "*performance by default*" (BAYLISS, 2022).

Alguns jogos apresentam algumas partes que foram feitas usando a abordagem DOD, como *Stellaris* que o empregou na parte de IA (BARI, 2017), *Battlefield 3* em renderização (COLLIN, 2011), *Yahaha* para otimizar a conexão entre jogadores (ZHOU, 2023), e vários outros jogos. O DOD é versátil e pode ser usado em conjunto com o DOO numa abordagem híbrida, mas para este trabalho, o estudo foi feito de uma maneira mais isolada, sem interferência do DOO e vice-versa.

Portanto este trabalho busca comparar o desempenho dessas duas abordagens por meio de uma versão DOD e DOO de uma simulação de batalha entre tanques de dois lados. Inicialmente dois times de tanques são gerados de forma agrupada e então começam a atirarem no time inimigo até o último tanque ser destruído. Os cenários de teste levam o DOD a trabalhar tanto com um hardware atual e equilibrado quanto com o desiquilíbrio entre peças, o que pode trazer um resultado interessante na relação entre CPU e GPU.

1.1 Objetivos

Comparar a performance do DOD (Design Orientado a Dados), usando DOTS (*Unity Data-Oriented Design Stack*), com o DOO (Design Orientado a Objetos), em termos de renderização e uso de *hardware*, como CPU.

1.1.1 Objetivos específicos

- Conferir os ganhos de desempenho do Design Orientado a Dados;
- Verificar em quais situações um paradigma se sai melhor que o outro;
- Avaliar o impacto que diferentes configurações de hardware trazem;

2 FUNDAMENTAÇÃO TEÓRICA

Nesta sessão, é apresentada a base teórica sobre o Design Orientado a Objetos e o Design Orientado a Dados, assim como a *Unity*, *engine* usada no projeto, e alguns conceitos sobre jogos digitais.

2.1 Design Orientado a Objetos

O Design Orientado a Objetos busca o foco no objeto, definindo classes para dar significado a cada parte do software, e dando atributos e comportamentos por meio de propriedades e métodos dentro dessas classes e construindo relações entre elas (NAKOV; KOLEV, 2013). É uma abordagem já consolidada no mundo do software e bastante popular no desenvolvimento de jogos.

A *Godot* é um exemplo de *engine* que usa fortemente os princípios do Design Orientado a Objetos, onde cada objeto tem o seu tipo, que pode possuir derivados e herança, e um único *script* (SALMELA, 2022). Na Figura 1, é possível notar a ênfase nos tipos de objeto e na organização em hierarquia, onde conforme a hierarquia se aprofunda, os objetos vão ficando cada vez mais especializados.

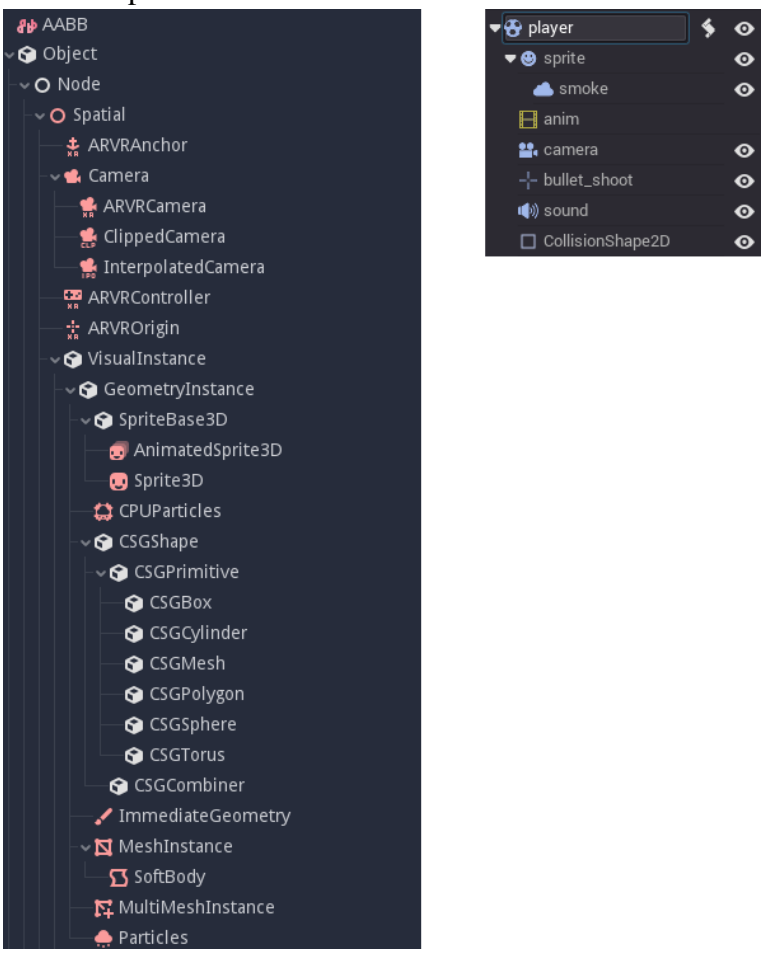
2.1.1 Princípios de Orientação a Objetos

"Os princípios da programação orientada a objetos se estruturam sobre abstração, herança, encapsulamento e polimorfismo" (NAKOV; KOLEV, 2013, p. 208).

A abstração esconde partes do código não necessárias para o seu uso como estrutura a se instanciar. Se temos um objeto que descreve uma porta a abrir, não precisamos saber em primeiro momento como ela é renderizada, ou se usa máquina de estados ou eventos. Seu funcionamento interno é abstraído, precisamos somente do que nos interessa de uma porta, como por exemplo os comportamentos de abrir, fechar, trancar e quaisquer outros que esta porta se propõe a fazer.

O encapsulamento está muito ligado a abstração a medida em que ele esconde o funcionamento interno do objeto. Muitas vezes é necessário usar uma variável de cache ou temporária, armazenar ponteiros ou estados de máquina, então o encapsulado esconde essas informações que só servem para o funcionamento do objeto. O encapsulamento pode variar entre esconder para todo objeto fora do escopo, ou permitir o acesso para objetos derivados ou que se

Figura 1 – Disposição dos objetos numa abordagem DOO, utilizada pela *Godot*



Fonte: Godot

relacionam de alguma forma.

A herança permite que objetos transmitam informação para as classes derivadas, reaproveitando código e dando uma especifidade maior àquilo proposto originalmente pelo objeto herdado (NAKOV; KOLEV, 2013). É uma característica extremamente poderosa, uma vez que possibilitar o reúso de muito código escrito onde os objetos contém dados dos objetos herdados. O polimorfismo é intrinsecamente interligado com a herança, ele promove um comportamento geral que muda em casos específicos. Várias vezes podemos generalizar objetos específicos numa classificação geral única.

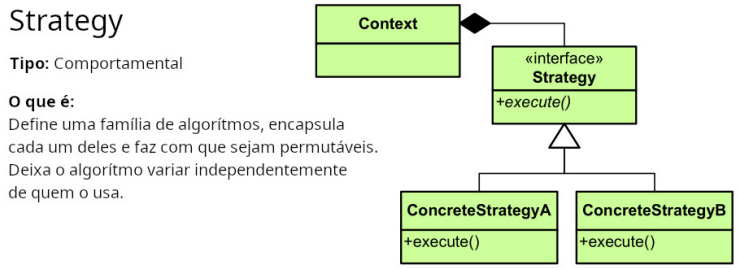
O DOO busca transferir a ideia que o ser humano tem de mundo como árvore com vários ramos, galhos e folhas para o computador e então construir sistemas complexos descrevendo os objetos nessa organização (FABIAN, 2018). A indústria abraça até hoje este paradigma e o usa em diversos ramos de negócio.

2.1.2 Padrões de Projeto comuns

(AMPATZOGLOU; CHATZIGEORGIOU, 2007) investigaram o código fonte de vários jogos existentes para saber os *patterns* mais utilizados, dentre eles o *Strategy* e o *Observer* foram os mais encontrados.

O *Strategy* define, geralmente com o uso de uma interface em comum, um conjunto de objetos encapsulados que implementam estes métodos em comum de forma própria e então disponibiliza acesso um deles (GAMMA *et al.*, 1995) . Este *pattern* faz uso principalmente das características de polimorfismo do DOO, já que o comportamento varia de acordo com a classe instanciada.

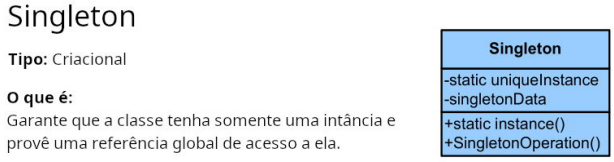
Figura 2 – Diagrama do *pattern* Strategy



Fonte: http://www.mcdonaldland.info/2007/11/28/40 (traduzido)

Outro *pattern* comum é o *Singleton*, que se comporta como um objeto estático em toda a aplicação, entretanto funcionando diferente de somente uma classe com campos estáticos, pois por se tratar de um objeto inicializável com construtor, permite configurações, métodos e comportamentos a mais e de forma mais organizada que somente uma classe estática (PARVIAINEN, 2017). Na *Unity*, geralmente é usado para preservar um objeto entre os carregamentos de cena.

Figura 3 – Diagrama do *pattern* Singleton



Fonte: http://www.mcdonaldland.info/2007/11/28/40 (traduzido)

O problema com o *Singleton* surge pois seu uso torna difícil fazer manutenções no código, além de não ser um objeto natural para existir no DOO. Para resolver esse problema,

existe uma alternativa que é usar Injeção de Dependência, um *pattern* bem popular em DOO que a indústria de *software* abraça há tempos (PARVIAINEN, 2017). De forma geral, injeção de dependência consiste em inicializar objetos e resolver suas "dependências"automaticamente. A configuração de inicialização é feita em alguma parte do programa, e então os objetos desejados têm suas instâncias inicializados conforme foi feita a configuração.

2.1.3 Problemas

O DOO não é uma “bala de prata” que resolve todos os problemas da melhor maneira possível. Para lidar com dados e operações – algo comum no desenvolvimento de jogos –, o DOO possui uma série de características que formam uma barreira e dificultam construir soluções de alto desempenho (NIKOLOV, 2018).

Como os objetos estão dispostos aleatoriamente na memória e na ordem de chamá-los, o uso de memória cache é mal aproveitado com a ocorrência de mais *cache miss*, é mais difícil paralelizar o código e as otimizações muitas vezes ficam presas ao escopo do objeto (LLOPIS, 2009). O fato de pensar as coisas como entidades únicas faz com que percamos a visão do todo e com isso o uso oportuno de tal visão.

É comum dos desafios enfrentados durante a programação em desenvolvimento de jogos incluírem grandes quantidades de um mesmo objetos, vários estados para uma entidade se comportar (LLOPIS, 2009). Problemas esses que não combinam bem visão de como o Design Orientado a Objetos se propõe a solucionar problemas. Então nessa mesma vertente surge o Design Orientado a Dados, que busca focar mais nos dados em si.

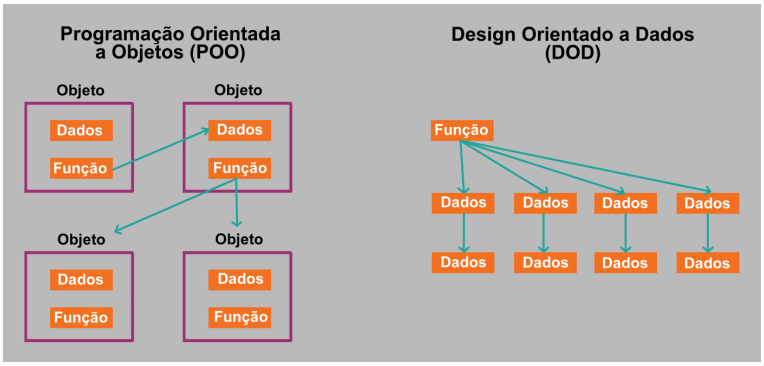
O DOD junto com o *pattern Entity-Component-System* (ECS) resolve por natureza esse problema de instância única e de dependências, que, como observado em (FEDOSEEV *et al.*, 2020), conseguiu substituir diversos *patterns* DOO.

2.2 Design Orientado a Dados

O Design Orientado a Dados (DOD) é uma abordagem de programação que foca nos próprios dados da aplicação, levando em conta os seus tipos, a forma como eles são armazenados na memória e o processamento deles (LLOPIS, 2009), contrastando com o Design Orientado a Objetos (DOO), que se concentra no próprio objeto e nas relações que ele tem com outros.

O pensamento é antigo e já se manifestava de várias formas. Após o lançamento do

Figura 4 – Disposição dos objetos no DOO e no DOD



Fonte: Unity Technologies (traduzido)

Playstation 3, onde a arquitetura de memória baseada em células induzia os desenvolvedores a pensar de maneira orientada a dados (BAYLISS, 2022). *Thief: The Dark Project* pode ser dito como um dos primeiros jogos ser construídos usando DOD.

A grande vantagem do DOD em relação do DOO está no pensamento em si como abordagem. Os problemas enfrentados no desenvolvimento de jogos sempre envolvem uma grande quantidade de instanciações de um mesmo objeto, isto é, de várias entidades, onde o DOO já não se sai tão bem no desempenho. Muitas vezes são vários Personagem não jogável (NPC)s para processar, vários objetos com a mesma mecânica, vários cálculos de uma mesma função de física e outros elementos comuns na maioria dos jogos (LLOPIS, 2009). Dessa forma, ter o controle dessas instâncias provê um controle conveniente para aproveitar o paralelismo e a memória cache, tudo isso ao preço de uma menor abstração (FABIAN, 2018).

Se comportando como um paradigma ou não, o DOD é capaz coexistir com outros paradigmas (FABIAN, 2018). Isso não é uma particularidade. Vários paradigmas de programação já coexistem naturalmente em linguagens multi-paradigmas, como *Java*, *C#*, *C++*, *JavaScript*, *Python* etc.

Por estar numa camada mais próxima dos dados em si e do hardware, a exigência de conhecimento dos programadores é maior, exigindo habilidades mais teóricas e de Ciência da Computação no geral (FEDOSEEV *et al.*, 2020).

2.2.1 Multitheading: Um recurso poderoso

O *multithreading* possibilita uma tarefa ser dividida em subtarefas para então se trabalhar em paralelo nelas ao mesmo tempo, assim obtendo ganhos de desempenho.

Antes de falar sobre *multitheading* é preciso definir o que é uma *thread*. Sem retornar totalmente à literatura clássica de Sistemas Operacionais, uma *thread* se assemelha a

um subprocesso, porém com a diferença que ela compartilha os dados de memória com outras *theads* que estão contidas neste processo. Dando assim várias possibilidades e principalmente o paralelismo, isto é, fazer várias tarefas ao mesmo tempo em paralelo (TANENBAUM, 2015). Portanto, o *multitheading* é uma técnica que consiste em usar várias *threads* para realizar um trabalho. Em desenvolvimento de jogos, o *multitheading* é essencial para alcançar resultados performáticos.

O Design Orientado a Dados consegue facilitar muito o emprego desta técnica, pois os dados ficam dispostos em coleções em vez de numa estrutura encadeada com comportamentos próprios, resultando em menos sincronizações e num código de melhor qualidade (LLOPIS, 2009). Dessa forma, o *multithreading* anda sempre junto com o DOD, sendo uma grande vantagem desta abordagem que explora o máximo do *multithreading*.

Para este projeto, otimização foi sempre algo buscado igualmente para os dois, e o *multitheading* é alcançado mais vezes na versão DOD dada a facilidade e conveniência que a abordagem proporciona, enquanto que a versão DOO ele é alcançado somente de forma implícita pelos mecanismos internos da *engine*, como na parte de física (UNITY, 2023a).

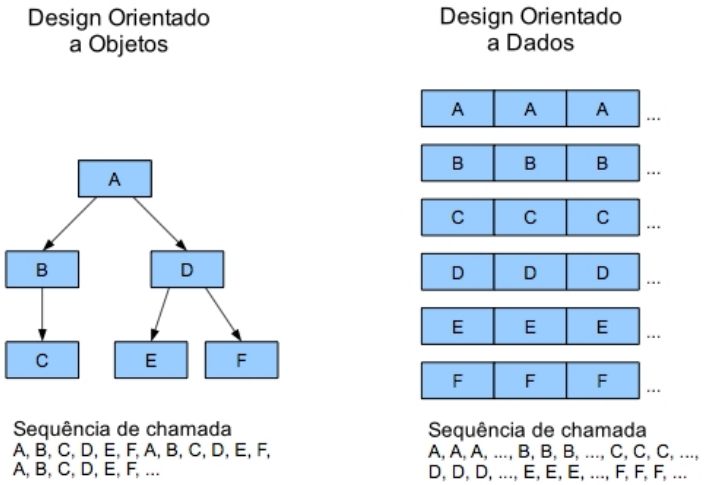
2.2.2 Comparação com o Design Orientado a Objetos

Uma forma de comparar o DOD com o DOO é pensarmos num sistema onde carros precisam ir de um ponto ao outro. No DOO, o sistema está dentro do carro e lá toda a lógica do que ele deve fazer para chegar até o destino. Se existir poucos ou muitos carros, não faz diferença. A visão primordial ao se implementar essa entidade é em como o carro irá se portar, também diante dos outros carros, para ir até o destino.

Já no DOD, todos os carros são controlados por um grande sistema formado de pequenas partes. Esse sistema "vê" todas as entidades e cada parte dele faz o trabalho que, quando unido, toma as decisões para que cada um chegue até o seu destino. Por ter uma visão mais perto dos dados e das estruturas de armazenamento, o paralelismo se torna algo mais simples, assim como as ordens de chamada, favorecendo a memória cache.

Até a forma como os dados são armazenados acaba por se diferir entre as duas abordagens. O Design Orientado a Objetos busca definir o todo de um objeto por vez, fazendo com que quando há muitas instanciações deste objeto, ele fique espalhado pela memória. Como todo o conteúdo está dentro do objeto, o acesso a cada um deles fica encadeado. Já no Design Orientado a Dados, os essas informações ficam armazenadas numa coleção, assim permitindo

Figura 5 – Comparação das ordens de chamada entre DOO e DOD



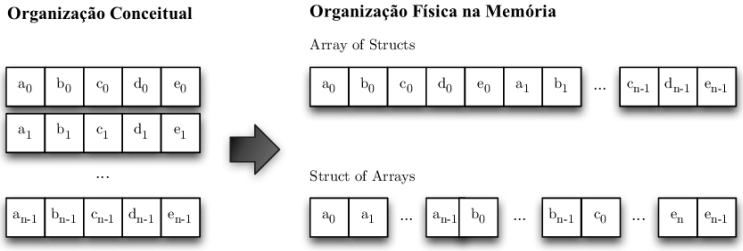
Fonte: (LLOPIS, 2009) (traduzido)

acesso direto aos dados, dando ganhos de desempenho. Essas duas formas divergentes de armazenamento é conhecido como *Array of Structures* e *Structure of Arrays*, que serão explicadas na seção a seguir.

2.2.3 Array of Structures (AoS) e Structure of Arrays (SoA)

Array of Structures (AoS) e *Structure of Arrays* (SoA) são duas formas opostas de organizar dados. AoS acontece naturalmente no DOO, onde temos vários objetos espalhados na memória envolvendo cada um seus dados. Já o SoA há uma inversão, onde os dados, que se encontravam dentro de um objeto, estão agora envolvidos em *arrays*.

Figura 6 – Comparação entre AoS e SoA



Fonte: (PENNYCOOK *et al.*, 2013) (traduzido)

O DOD busca essa inversão, em comparação ao DOO, pois o controle desses dados são importantes demais para serem abstraídos. Organizando os dados em SoA, o DOD fornece melhores caminhos para tratar grandes quantidades de objetos, favorecendo o bom uso da memória cache (BAYLISS, 2022).

Assim cada grupo descritor de dados forma um componente, que será armazenado numa coleção e então iterado posteriormente. Esse passo a passo faz parte do *Entity Component System*.

2.2.4 Entity Component System

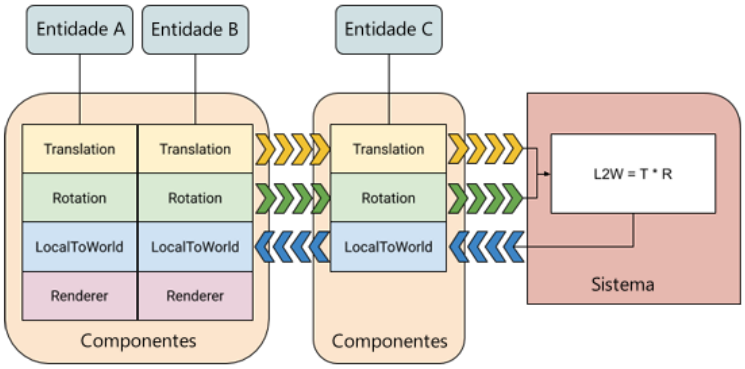
O *Entity Component System* (ECS) é um *pattern* que propõe uma organização em identidade *(Entity)*, dado *(Component)* e comportamento *(System)* (HäRKöNEN, 2019). O ECS é um passo essencial para usar o DOD, uma vez que padrões de arquitetura organizam o projeto, ajudando em diversos aspectos do código, como legibilidade, manutenibilidade, complexidade etc, impactando na performance.

A entidade é o objeto dentro do código com seus comportamentos e dados que o descreve fora dele. Os dados que cada entidade usa é chamado de componente, e a parte que descreve o comportamento é o sistema.

Ao poder organizar os dados e declarar certas propriedades como de somente leitura favorece muito o uso de *multithread* seguro, evitando condições de corrida e erros de sincronização (HäRKöNEN, 2019).

A Figura 8 expressa visualmente o conceito do ECS, onde há entidades com alguns componentes em comum e um sistema que usa os valores dos componentes de translação e rotação para obter a conversão de coordenadas locais para as globais.

Figura 7 – Diagrama da organização proposta pelo ECS



Fonte: Unity Technologies (traduzido)

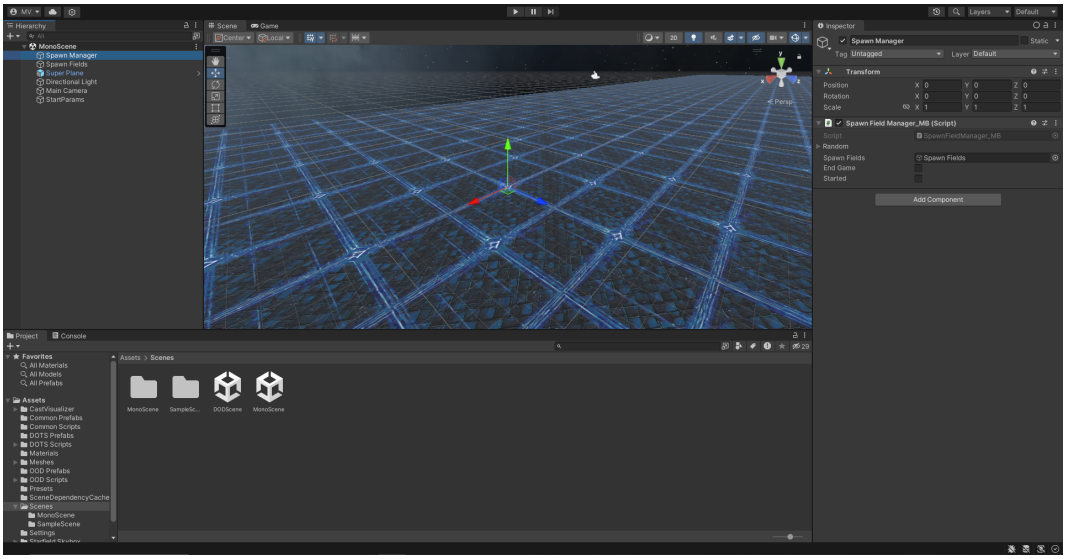
A seguir, a *engine Unity* é apresentada, assim como as principais características que se relacionam com este projeto.

2.3 Unity

A *Unity* é uma *engine* de jogos multiplataforma desenvolvida pela *Unity Technologies* e lançada em Junho de 2005 (HUSSAIN *et al.*, 2020). Por ser um ambiente 3D rico em recursos, como física e gráficos realistas, sua adoção vai além da indústria dos jogos, estendendo-se para arquitetura, engenharia, automação, animação, cinema e demais seguimentos onde um ambiente tridimensional com agentes inteligentes é necessário (JULIANI *et al.*, 2020).

Uma *engine* gráfica reúne vários recursos e os relaciona para criar um ambiente gráfico rico, por isso é um pilar para o desenvolvimento de jogos digitais. Coisas como modelos 3D, VFX (efeitos visuais), texturas, áudio, GUI (interface gráfica por meio do qual o usuário interage com o sistema), cinematografia etc (ACTON; GAMES, 2014). Por padrão, assim como muitas *engines* gráficas disponíveis na *Internet*, alguns componentes essenciais já vêm implementados na *Unity*, como alguns de física, iluminação, renderização, câmera, áudio, GUI etc.

Figura 8 – Interface gráfica padrão da *Unity Engine*



Fonte: Capturada pelo autor

2.3.1 Ambiente

O ambiente da Unity é baseado em componentes. Os objetos de uma cena, também chamados de *GameObjects*, são organizados em forma de árvore e permitem a anexação componentes, os quais descrevem as coisas do jogo. Atualmente os componentes são escritos em *C#*, uma linguagem com memória gerenciada e com uma comunidade ativa.

Um *GameObject* pode ser guardado num arquivo como um *Prefab*, que é um objeto pré-configurado que provê flexibilidade e eficiência no fluxo de trabalho (GOLDSTONE, 2011), além de funcionar como uma referência que, quando o conteúdo é alterado, se atualiza em todos os lugares em que o *Prefab* foi usado.

Por padrão, alguns componentes essenciais já vêm implementados, como alguns de física, iluminação, renderização, câmera, áudio, UI etc. O DOO é a abordagem padrão até o momento, como acontece na maioria das *engines* abertas ao público. É possível obter recursos para usar DOD por meio tanto de pacotes, que são *plugins* feitos tanto pela comunidade quanto pela *Unity*. Os componentes que o programador escreve são chamados de *scripts*. A anatomia de um *script* consiste em uma classe derivada de *MonoBehaviour* com métodos de *update*, isto é, métodos que são chamados várias vezes por segundo; e também métodos que são chamados somente quando algo acontece, como inicialização ou destruição de um *GameObject*.

Código-fonte 1 – *Script MonoBehaviour* que move o *GameObject* para a direita e para a esquerda num intervalo de tempo

```
1  public class MB_Move : MonoBehaviour
2  {
3      public float Duration, Speed;
4      private float timerCount;
5      void Start()
6      {
7          timerCount = Duration;
8      }
9      void Update()
10     {
11         if(TimerCount <= 0)
12         {
13             Speed *= -1f;
14             timerCount = Duration;
15         }
16
17         timerCount -= Time.deltaTime;
18         transform.Translate(Vector3.right * Speed * Time.deltaTime);
19     }
20 }
```

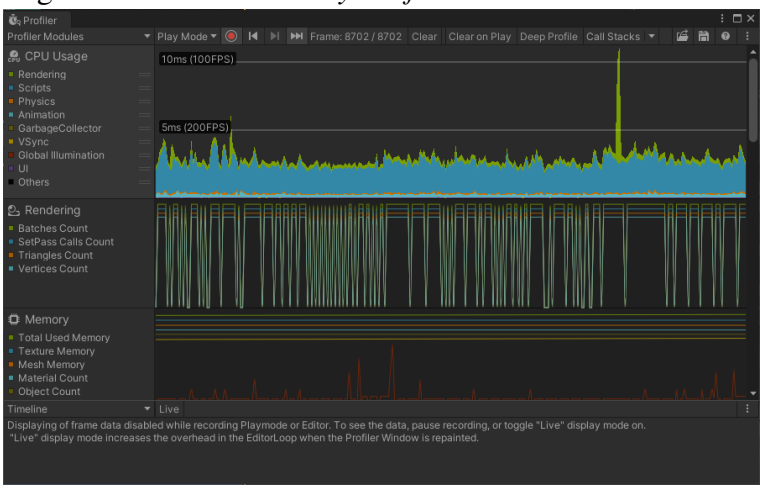
2.3.2 *Compilação*

Na compilação há *backends* disponíveis: Mono e IL2CPP (UNITY, 2022e). Todo *script* escrito em *C#* passa por uma compilação feita pelo *Roslyn* para *Common Intermediate Language* (CIL), que é um código intermediário baixo nível gerado a partir de qualquer coisa escrita na plataforma *.NET*, após isso o resultado é executado por uma máquina virtual – a *Common Language Runtime* (CLR) (MICROSOFT, 2022). A conversão para CIL é uma etapa comum de todos os *backends* e até mesmo para coisas feitas com *DOTS*, que será abordado mais a frente neste trabalho. Enquanto o *backend Mono* usa *JIT*, isto é, compilação que ocorre em tempo de execução, o *IL2CPP* faz uma conversão do CIL para *C++* e depois para código nativo de acordo com a plataforma alvo (UNITY, 2022e). Ainda antes do *IL2CPP* entrar em cena, ocorrem alguns procedimentos de otimização, como a remoção de partes inacessíveis ou não usadas do código, num processo chamado *bytecode stripping* (UNITY, 2022e).

2.3.3 *Profiler*

Em todo desenvolvimento de qualquer aplicação é necessário ter em mão ferramentas para depurar e ver por entre o visual gerado e o código escrito em si o que está acontecendo. *Scritps C#* podem depurados pelo *software* editor de código em conjunto com a *engine*, ademais existe o *Unity Profiler*, que entrega informações sobre o desempenho do jogo (UNITY, 2022d). Nele podemos detectar vazamentos de memória e supervisionar o uso do *hardware* (LINTRAMI, 2018), portanto é um recurso essencial para executar testes de performance e o uso de recursos do *hardware*.

Figura 9 – Janela do *Unity Profiler*



Fonte: Capturada pelo autor

Vale destacar também o pacote *Performance Testing API*, disponibilizado pela *Unity*, que no momento se encontra ainda em versão *preview*. Tal pacote funciona em conjunto em o *Unity Profiler* e atua como uma API ligando o código escrito ao *profiler*, provendo atributos *C#*, funções e vários meios para ajudar a testar o desempenho (UNITY, 2022c).

2.3.4 Desafios técnicos

Fazer jogos é difícil, independentemente das tecnologias que se use. *Engines* fazem um ótimo trabalho abstraindo várias camadas da Computação Gráfica, de modo que os desenvolvedores podem focar mais em desenvolver suas ideias e resolver os problemas específicos de seus negócios. Ainda assim, as vezes é preciso olhar para as camadas mais próximas do *hardware*. Para certos requisitos ou objetivos serem cumpridos – principalmente os de desempenho – é necessário um olhar mais apurado até as entranhas dos mecanismos que fazem um jogo funcionar. Não obstante, as diversas configurações de PC existentes adicionam mais dificuldade em entregar um jogo robusto e suave, tudo isso respeitando prazos apertados comuns da indústria e orçamento do projeto.

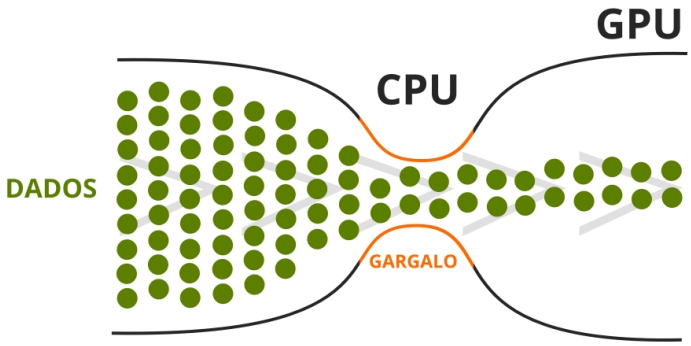
Um jogo suave deve manter uma taxa de quadros por segundo geralmente em 30 ou 60, para isso é necessário que o hardware consiga processar as informações a tempo para cada quadro ser formado. Apesar de games se remeterem muito a gráficos, geralmente toda a plataforma é usada, desde armazenamento até a GPU, principalmente em jogos complexos. Quando algum componente do hardware não consegue produzir na mesma intensidade que os demais, atrasando assim todo o trabalho, acontece o que é chamado de gargalo. Isso acontece geralmente na CPU ou GPU, que são os componentes usados de forma mais intensa na maioria dos jogos, fazendo com que algum deles fique ocioso esperando o outro componente terminar a tarefa necessária para gerar o quadro.

Há vários outros desafios, variando de acordo com cada característica única do jogo. Para este projeto em especial, houve um relacionamento com o problema do gargalo.

2.4 DOTS

O *Data-Oriented Technology Stack (DOTS)* da *Unity* é o um exemplo em larga escala do DOD (BAYLISS, 2022). Disponível desde a versão 2018.1 (FARYABI, 2018), esta *stack* é disponibilizada pela *Unity* por meio de pacotes a serem instalados pelo *Package Manager*.

Figura 10 – Ilustração de quando uma CPU menos potente atrasa o processamento dos dados



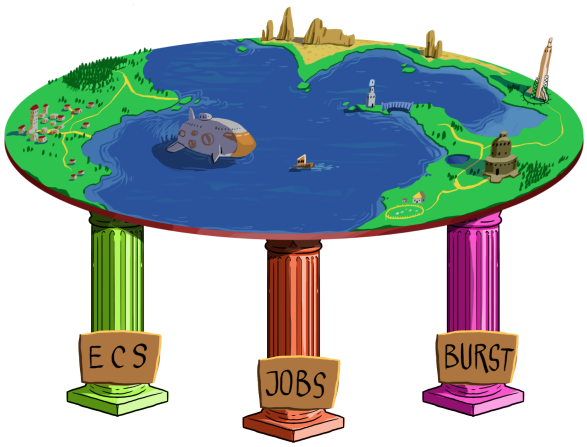
Fonte: Elaborada pelo autor

Portanto para se usar o DOTS é necessário ter instalado pacotes como o *Entities* e *Entities Graphics*, além de outros pacotes dependendo dos requisitos do desenvolvedor, como *Physics*, que provê recursos de física.

Passado por várias fases de testes, o DOTS hoje já se encontra usável para produção, apesar de ainda não abranger todos os elementos que compõem uma *engine*, como recursos de animação e áudio, providos respectivamente pelos pacotes *Animation* e *DSPGraph*, ambos ainda em fase *preview*.

Os pilares do DOTS são os pacotes *Entities, Jobs* e *Burst*, cada um com sua função especializada (UNITY, 2022f). Além disso a *Unity* entrega ferramentas de teste e de depuração.

Figura 11 – Esses três pacotes compõem os pilares do DOTS



Fonte: Unity Technologies (traduzido)

2.4.1 Entities: o Entity Component System

O pacote de entidades *(Entities)* é o responsável por prover o ECS. Dessa forma o *Entities* muda a forma de como se escreve os *scripts* dentro da *Unity*, em invés de usar somente *MonoBehaviours*, os *scripts* em ECS usualmente se dividem três partes: componente, sistema e entidade (UNITY, 2023b):

- **Component (Componente):** lugar onde definimos os tipos de dados que iremos usar;
- **System (Sistema):** onde definimos o comportamento dos componentes;
- **Entity (Entidade):** uma união de vários componentes, que descreverá o objeto no mundo;

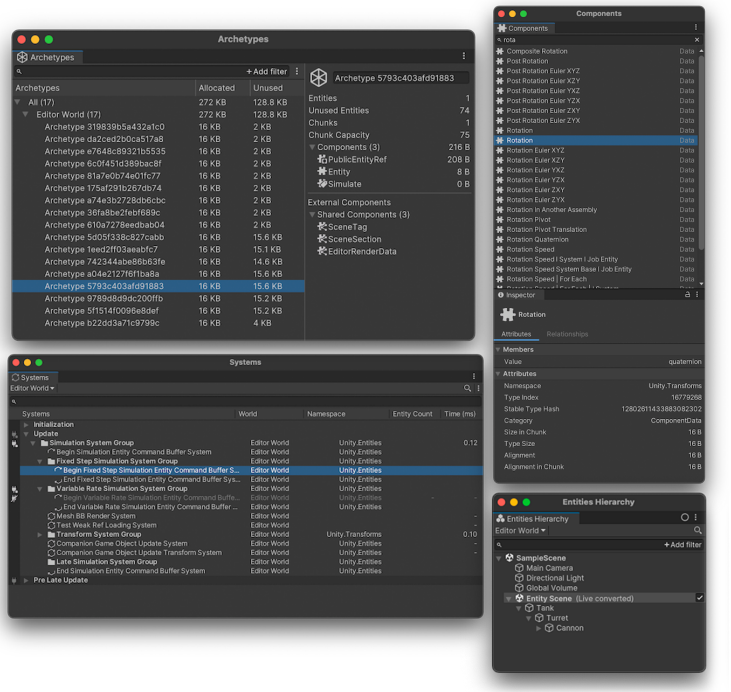
Há outras estruturas que ajudam o *pattern* ECS funcionar, como *Aspects*, que unem vários componentes e seus estados numa única estrutura, assim ajudando a criar *queries* e grupos de componentes com algum significado para o desenvolvedor; *Bakers*, que transformam *GameObjects* em entidades no mundo DOTS; além de variações de cada uma dessas partes que compõem o ECS (UNITY, 2023b). Na prática, isso se traduz em vários objetos – classes e principalmente *structs* em *C#* – que herdam classes e, mais habitualmente, *interfaces* para cada uma de suas finalidades, que por vezes seriam algo que só um único *MonoBehaviour* faria.

Por ser um paradigma novo, isso implica no surgimento de vários conceitos, que ajudam a organizar a área de trabalho e os caminhos até a implementação final de qualquer coisa DOD em DOTS. Uma combinação única de componentes é chamada de *archtype*, que serve para identificar eficientemente as entidades da cena, sendo assim a principal forma de acesso. Cada *archtype* é único e compartilhado entre as entidades (UNITY, 2023b). Todas as entidades estão contidas numa estrutura chamada *world*, que tem o papel de gerenciar as entidades, seja para criar, destruir ou modificar (UNITY, 2023b).

No editor, após a instalação do pacote, várias janelas de supervisão podem ser habilitadas. Elas são muito úteis para observar os objetos em funcionamento e saber de forma mais profunda o que está acontecendo, como as estruturas estão interligadas e dispostas da memória.

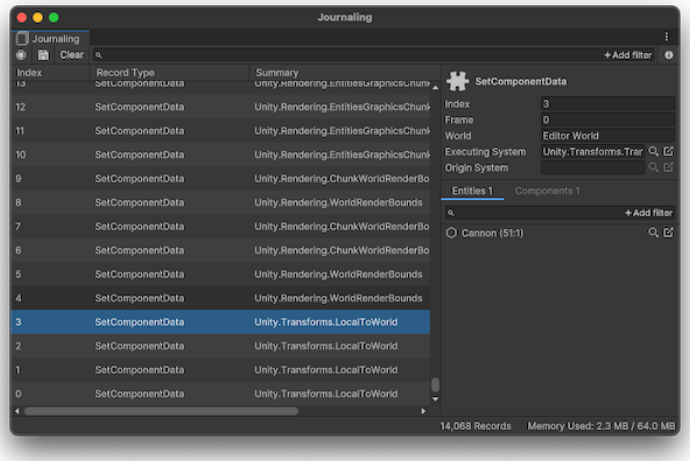
Vale a pena destacar o recurso *Journaling*, que registra, pelo menos, a maior parte das escritas nas estruturas (UNITY, 2023b), e, além de possuir também uma janela para o editor, provê uma integração com a IDE, tornando-se assim uma importante aliada na hora de depurar a aplicação.

Figura 12 – Janelas de *archtypes*, sistemas, componentes e hierarquia adicionadas pelo pacote *Entities*



Fonte: Unity Technologies

Figura 13 – Janela de Journaling provida pelo *Entities*



Fonte: Unity Technologies

2.4.2 Burst

O *Burst* é um compilador que converte CIL em código nativo de forma extremamente otimizada (UNITY, 2022a). Diferentemente do *IL2CPP*, o *Burst* transforma o *script* escrito pelo programador, já compilado em CIL pelo compilador *Roslyn*, para um código fonte *C++* otimizado usando o *LLVM*, que é uma estrutura consolidada e madura para compilação de

programas escritos em várias linguagens (UNITY, 2021).

O *Burst* já se encontra em versão *release* e pode ser baixado normalmente pelo *Package Manager*. Para utiliza-lo, além de estar ativado no *editor*, basta adicionar o atributo "*[BurstCompile]*" sobre o *struct* de um *Job*.

O *Burst* é um dos pilares do DOTS e usado em quase todos os *scripts* ECS.

2.4.3 Jobs

O *Jobs* entrega uma forma segura de escrever códigos *multithreading* (UNITY, 2022b). Seria um desperdício usar DOD e não aplicar paralelização. O sistema de *Jobs* da *Unity* atua especificamente nessa questão de auxiliar o programador e, no momento, é um recurso que já é recomendado em produção, mesmo sem ser em um projeto DOTS.

Um *job* é uma unidade de trabalho que faz alguma tarefa no programa. Portanto, um sistema de *jobs* é um gerenciador de vários *jobs*, que acontecem sob *threads*, assim abstraindo parte da complexidade da programação paralela (UNITY, 2022b).

Um dos principais perigos de escrever códigos *multithreading* são as condições de corrida. Uma condição de corrida ocorre quando duas operações simultâneas são executadas numa mesma variável, podendo entregar a leitura de um valor que fora sobrescrito em questão de milisegundos, logo esta leitura é de um valor que não existe mais. Um código *multithread* errado pode não causar um erro de compilação e estar até correto logicamente, mas, pelas condições de corrida, pode provocar um comportamento inapropriado.

Para promover um *multithreading* seguro, o *C# Job System* da *Unity* detecta possíveis situações de condições de corrida e então passa esses dados por cópia em vez de por referência para o *job* que processará aquela tarefa, garantindo ao fim que não haverá nenhuma condição de corrida (UNITY, 2022b). Como desvantagem, a cópia não é vinculada ao dado copiado, não sendo possível obter o resultado daquela operação, como solução desse detalhe, um objeto *NativeContainer* provê uma memória compartilhada, que contém um ponteiro para a região de memória copiada, dando então acesso aos resultados do *job* processado (UNITY, 2022b).

No ECS, o *Jobs* atua principalmente perto de estruturas de *System*, onde há de fato o processamento mais massivo, entretanto ele não é algo exclusivo para o DOTS e pode ser usado também em *MonoBehaviours* na abordagem DOO no geral.

2.4.4 Exemplo

Então, numa comparação entre *scripts* DOTS, usando como exemplo gerar o comportamento de um objeto transladar para a direita até um dado tempo e depois voltar, igual o exemplo escrito pelo Código-fonte 1, o *script MonoBehaviour* clássico pode ser escrito habitualmente declarando as variáveis na classe do tipo *MonoBehaviour*, inicializando e construindo o comportamento no método *Update*.

Já um *script* DOTS será dividido em pequenas estruturas, começando pelo *Component* que declara os tipos básicos da entidade, empacotando o *Component* criado e outras estruturas fundamentais num *Aspect*, escrevendo o comportamento num *System* e utilizando um *Job*, e por fim fazendo a ponte entre o *GameObject* com um *script* clássico *MonoBehaviour* e um *Baker* para transferir os dados.

Código-fonte 2 – Estrutura *Component* do *script* ECS declarando dados básicos

```
21 public struct Move : IComponentData
22 {
23     public float Duration, Speed, CountTimer;
24 }
```

Código-fonte 3 – *Aspect* unindo vários dados e definindo algumas rotinas

```
25 public readonly partial struct MoveAspect : IAspect
26 {
27     public readonly Entity entity;
28     public readonly RefRW<Move> move;
29     public readonly LocalTransform transform;
30
31     public void Translate(float deltaTime)
32     {
33         float speed = move.ValueRO.Speed;
34         float3 right = math.right();
35         transform.Position += right * speed * deltaTime;
36     }
37     public void Invert()
38     {
39         move.ValueRW.Speed *= -1f;
40         move.ValueRW.CountTimer = move.ValueRO.Duration;
41     }
42
43 }
```

Código-fonte 4 – *Job* que processará o comportamento de mover-se e voltar após um dado tempo, com *Burst* habilitado

```
44 [BurstCompile]
45 public partial struct MoveJob : IJobEntity
46 {
47     public float DeltaTime;
48     public void Execute(MoveAspect moveAspect)
49     {
50         if (moveAspect.move.ValueRO.CountTimer <= 0f)
51             moveAspect.Invert();
52         moveAspect.move.ValueRW.CountTimer -= DeltaTime;
53         moveAspect.Translate(DeltaTime);
54     }
55 }
```

Código-fonte 5 – Estrutura *System* invocando o *Job* a ser executado, com paralelismo habilidado e usando o *Burst*

```
56 [BurstCompile]
57 public partial struct MoveSystem : ISystem
58 {
59     [BurstCompile]
60     public void OnUpdate(ref SystemState state)
61     {
62         float deltaTime = SystemAPI.Time.DeltaTime;
63         new MoveJob
64         {
65             DeltaTime = deltaTime
66         }.ScheduleParallel();
67     }
68 }
```

Código-fonte 6 – Componente *MonoBehaviour* que será anexado a um *GameObject*

```
69 public class MoveMono : MonoBehaviour
70 {
71     public float Duration;
72     public float Speed;
73 }
```

Código-fonte 7 – *Baker* transferindo os dados de todo componente *moveMono* para o *script* em DOTS

```
74 public class MoveBaker : Baker<MoveMono>
75 {
76     public override void Bake(MoveMono authoring)
77     {
78         var entity = GetEntity(authoring, TransformUsageFlags.Dynamic);
79         AddComponent(entity, new Move
80         {
81             Duration = authoring.Duration,
82             Speed = authoring.Speed,
83             CountTimer = authoring.Duration
84         });
85     }
86 }
```

O código em DOTS pode parecer mais volumoso, mas vale lembrar que o DOD, embora esteja mais próximo do hardware e dos dados, trata grandes quantidades de objetos com facilidade e com paralelismo fácil de se implementar, como já dito na seção 2.2, e acaba sendo carregado lidar com grandes quantidades desde o nível mais básico de complexidade.

2.4.5 Outros pacotes

A proposta DOD da *Unity* vai mais além, oferecendo meios para lidar com física, animação *multiplayer* e demais elementos inerentes ao desenvolvimento de jogos. A exemplo disto, há o *Collections* que provê meios de lidar com estruturas de coleções; o *Mathematics* que é uma biblioteca com diversos métodos e funções matemáticas optimizadas para o compilador *Burst*; o *Entities Graphics* que faz uma ponte entre as entidades e os atuais meios de renderização da *engine* – HDRP e URP (UNITY, 2023b).

3 TRABALHOS RELACIONADOS

3.1 Application of Data-Oriented Design in Game Development

Esse artigo é uma experimentação que estuda as questões da transição de um jogo feito em DOO para DOD.

No estudo, é proposta a criação de dois jogos casuais com as mesmas mecânicas e complexidades para se obter uma comparação justa com o fim de comparar os paradigmas. O primeiro jogo feito, chamado de "*Droppy Kick*", foi usado o DOO com herança de componentes, injeção de dependência e demais designs. O segundo jogo, chamado de "*Justing Sunset*", foi usado o DOD sendo implementado por meio do *LeoECS*, um pacote criado pela comunidade para implementar *scripts* com o DOD.

O estudo foi conduzido sobre dois desenvolvedores júniors, onde um tinha um conhecimento sólido sobre temas diversos de ciência da computação, enquanto o outro não tinha esses conhecimentos, apesar de que no conhecimento técnico da *engine Unity* ambos estão no mesmo nível. Em linhas gerais, o artigo analisa a performance, manutenibilidade e o conhecimento requerido por cada um dos designs.

Na performance, foram usadas como métricas: Taxa de quadros por segundo (FPS), tempo de uso da CPU, distribuição de carga entre as *threads*, intensidade de uso da CPU e uso de memória. Um *iPhone X* com 6 núcleos e 6 GB de RAM foi usado como máquina a testar os dois jogos. No geral, o DOD apresentou melhor desempenho em relação ao jogo feito em DOO. Em média, o *Justing Sunset* – jogo feito usando DOD – apresentou 58,3 FPS contra 56,9 FPS do *Droppy Kick*. Em processamento, o *Justing Sunset* se mostrou mais eficiente, usando apenas 25.11% da CPU, em média, contra 33.26% do *Droppy Kick*. Além disso, o jogo feito em DOD se mostrou melhor distribuído entre as *threads*.

Em manutenibilidade de código, os dois designs tiveram medidas muito parecidas, apesar de que foi observado uma dificuldade maior para entender o código e o seu fluxo no DOD, em contrapartida foi percebido que o DOD facilita o reuso de módulos do código, ponto importante quando falamos de manutenibilidade.

Sobre o conhecimento requerido por cada um dos designs, enquanto DOO exige menos conhecimento técnico de ciência da computação e muito de padrões de projeto, o DOD faz o contrário ao não necessitar de conhecimentos de nenhum *pattern* fora o ECS e importar mais com conhecimentos de algoritmos, sistemas operacionais, arquitetura de computadores,

compiladores etc.

A relação com este trabalho acontece quando ambos abordam a questão do DOD em alternativa ao DOO por meio de uma comparação de elementos. No entanto, o DOD foi alcançado por meio do *LeoECS*, este trabalho propõe uma comparação usando DOTS, que tem suporte oficial da *Unity*.

3.2 A Performance Comparison for 3D Crowd Rendering using an Object-Oriented system and Unity DOTS with GPU Instancing on Mobile Devices

Esse trabalho comparou a performance para renderização de multidões virtuais entre DOO e DOD, com DOTS. Multidões virtuais são várias entidades, geralmente *NPCs*, que têm função somente visual, não necessitam de mecânicas complexas ou interação com jogador. Exemplos disso são as torcidas em jogos de esportes.

As duas implementações foram feitas usando *Unity*. Os pacotes do DOTS se encontravam na versão 0.5.1 (*Entities*) e 1.2.1 (*Burst*). Os testes foram feitos em dois *smartphones*, ambos *iPhone*, dos quais um era o modelo S6 de 32 GB de armazenamento e outro um *XR* de 64 GB, todos operando no *iOS* versão 13.4.1. Apesar da diferença de resolução entre as duas máquinas – o modelo *6S* com 1334 x 750 e o modelo *XR* com 1792 x 828 –, foi feita uma escala de renderização para reduzir as disparidade, valores de 0.81 para o *6S* e 0.665 para o *XR* foram aplicados. Para extrair as informações do teste, foi usado *xCode* (v. 11.4.1), onde *frame-rate (FPS)*, uso de *CPU* e *GPU* foram os valores buscados.

Fora os *FPS*, o uso de CPU e GPU foram obtidos em ms (milissegundos), uma vez que as máquinas limitavam um *frame-rate* maior que 30, e para ficar claro o quanto de trabalho a CPU e a GPU teriam, mesmo em *frame-rates* muito baixos e até mesmo bem parecidos entre as duas implementações.

Em comum, os testes consistiram em renderizar várias quantidades entidades humanoides animadas num no plano formado pelos eixos x e z.

No primeiro cenário, os mesmos modelos e o animações foram usados, já no segundo cenário, o número de modelos e animações a variar foram aumentados para nove.

Os resultados mostraram uma performance superior na maioria dos casos para as versões feitas em DOTS, mas não em todos. Nos casos onde os testes consistiam em renderizar um número de personagens, com as mesmas animações e modelos, a versão DOTS se saiu inferior em *frame-rate* no *XR*, o mesmo aconteceu quando foi testado a renderização usando os

nove diferentes modelos e animações, também no *XR*. Apesar disso, nesses casos a diferença de *FPS* foi pequena. Os demais testes mostraram que o DOTS obteve uma larga vantagem sobre a versão DOO, tanto em *frame-rate*, quanto no aproveitamento de CPU e GPU.

Os trabalham se relacionam a medida que buscam comparar o desempenho das implementações DOO e DOD, feita usando os pacotes DOTS. Entretanto, os testes desse trabalho foram feitos em plataforma *mobile*, enquanto este trabalho tem um escopo focado na plataforma PC.

3.3 Unity DOTS in Production – DOTS pathfinding implementation in VR AR

Este trabalho busca responder questões pertinentes sobre se o DOTS encontra-se viável para uso em projetos na fase de produção, para tal, busca enfocar no algoritmo de *pathfinding* A*. Vale ressaltar que o trabalho foi publicado em 2021, portanto a versão do pacote *Entities* usada foi a 0.17.

O processamento consiste em gerar entidades aleatoriamente enquanto o FPS é maior que 10, essas entidades então executarão o algorítimo A* para encontrar um caminho até outro ponto do mapa. Além disso, há presença também de animações a serem processadas. Os testes incluem tanto a versão DOO quanto a DOD, este em maior quantidade para testar o impacto do *Burst* e do *multitheading*. Os testes para *PC* e *Mobile* foram executados em duas máquinas com características distintas.

- *PC*: Máquina com mais poder de paralelismo, dado o seu processador, e outra máquina com menos núcleos em sua *CPU*, porém mais poder de processamento em cada deles;
- *Mobile*: Um dispositivo mais moderno e potente (Google Pixel 3A), e outro mais antigo e menos potente (*LG Nexus 5X*)

Como o esperado, os testes que foram executados usando paralelização com o *Burst* se saíram bem melhores que os demais. A versão *multithread* se mostrou 13 vezes mais rápida no *Desktop* e 9 vezes mais rápido no *Notebook*, isso é um resultado interessante mostrando o benefício que o DOTS aproveita de uma *CPU* com vários núcleos. O mesmo se repete nos dispositivos *Mobile*, o *Google Pixel*, o qual tem maior quantidade de núcleo em relação ao seu concorrente (*LG Nexus*) também se saiu melhor.

Apesar dos ganhos expressivos de performance, o trabalho concorda com os próprio mantenedores – *Unity Technologies* – que não recomendam o uso do DOTS, na versão estudada no trabalho, para produção.

A relação aqui é que ambos os trabalhos compararam o DOO com o DOD usando DOTS, entretanto a diferença, inicialmente, é a versão do *Entities*, que neste trabalho é a 1.0.10, a qual já se encontra hábil para produção, além disso a mecânica permeada para os testes é diferente.

3.3.1 Tabela comparativa entre os trabalhos

Tabela 1 – Tabela comparativa entre os trabalhos

Trabalho	Mecânica	Ferramenta	Plataforma	Critérios
(FEDOSEEV *et al.*, 2020)	Diversas	*LeoECS*	*Mobile*	FPS, CPU, *threads* e memória
(TURPEINEN, 2020)	Multidões virtuais	*Entities* 0.5.1	*Mobile*	Latência
(NäYKKI, 2021)	*Pathfinding* A*	*Entities* 0.17	PC (VR) e *Mobile*	FPS
Este Trabalho	Diversas	*Entities* 1.0.10	PC	FPS, CPU e *threads*

4 METODOLOGIA

4.1 Planejamento do experimento

O simulador de batalhas, nomeado de "*Metal Rain*", faz instanciações massivas de entidades, que também repete isso em seus ataques atirando nos inimigos. Vários tipos de tanques serão gerados, uns com maior cadência de tiro, outros com menor, criando uma variedade maior de mecanismos funcionando.

A priori foi feito um pequeno planejamento da simulação e seus requisitos, para se ter em mãos uma visão do caminho a seguir.

4.2 Definir os critérios de comparação

FPS é uma métrica bem importante, mas não suficiente para responder todas as questões deste trabalho. Foi observado também a duração para processar os *frames*, balanceamento de carga entre os núcleos da CPU, assim como o seu uso em si. A variação desses dados conforme cada vez mais entidades serão instanciadas será algo interessante de ser observado também, além disso outras métricas também foram observadas, como o uso de memória da placa de vídeo e a de RAM.

4.3 Preparo dos ambientes de desenvolvimento

A *Unity Engine* versão 2022.3.0f1 foi usada, em conjunto com os pacotes *Entities Graphics* 1.0.10, *Entities* 1.0.10, *Unity Physics* 1.0.10, *Burst* 1.8.4 e suas respectivas dependências. A versão da *engine* e dos pacotes foram a mesma tanto para a implementação DOD quanto para a DOO. O trabalho começou exatamente na versão 1.0 do *Entities*, porém foi atualizado periodicamente até a versão 1.0.10, visto que pouquíssimas mudanças foram necessárias para aproveitar as atualizações. Os *scripts* serão escritos usando o *Visual Studio 2022*.

4.4 Implementação da versão DOO e DOD

Tanto a versão DOO quanto a DOD são o mesmo simulador de batalha, ou seja, a mesma mecânica, modelo, arte, textura etc, menos a parte da codificação, onde de fato é o escopo do projeto e onde foi comparado duas maneiras de se chegar ao mesmo simulador.

Apesar de dependerem da habilidade do desenvolvedor e de serem influenciada pelas duas abordagens, tanto a versão DOD quanto a DOO possuem otimizações e uso de boas práticas. Uma das qualidade do DOD é o fácil paralelismo e uso de memória cache, sem isso estaríamos usando somente uma parte do que o DOTS provê. Então para uma comparação mais justa é imprescindível que as duas implementações se aproximem em termos de otimização.

4.5 Preparo do ambiente de testes

A forma como se compara as duas implementações precisa ter uma atenção especial, pois de nada adianta medir algo se o sensor ou termômetro não é autêntico. Em outras palavras, será necessário obter as ferramentas mais adequadas para extrair as estatísticas, além de um ambiente isolado, estável e igual para as duas implementações.

Os softwares *MSI Afterburn* e *Rivatuner* foram utilizados para a captura das métricas. O simulador teve 3 *builds* criadas para a comparação:

- Versão Mono DOO, com somente *scripts MonoBehaviour*;
- Versão DOTS DOD em *thread* principal, com scripts ECS tendo *Jobs* e *Systems* programados sem pensar em *multithreading*;
- Versão DOTS DOD sem limitações, com *scripts* ECS tendo *Jobs* e *Systems* usando *multihreading* quando conveniente;

Os parâmetros e variações para cada um dos testes foram passados por meio de um arquivo *.json* que é lido pelo simulador ao iniciar. O arquivo se encontra no caminho provido pela variável *Application.persistentDataPath*.

Além disso, foi necessário verificar se os sistemas operacionais de cada máquina de teste estão em bom funcionamento, assim como garantir que não haja nenhum processo em segundo plano competindo recursos com o simulador. Além disso, é essencial que o hardware esteja em bom funcionamento. Tudo isso é necessário para que não aconteça interferências externas que possam distorcer a medição.

4.6 Execução de testes

Os testes foram executados em três máquinas com características distintas e com resolução de 1080p. Quem possuía resolução menor em sua própria tela, que é o caso de *Notebooks*, foi utilizado um monitor externo com resolução 1080p, com conexão via HDMI,

onde somente será usado a tela externa, sem duplicação de tela, extensão ou qualquer outro elemento que interfira no desempenho.

Tabela 2 – Especificações de cada máquina

Nome	CPU	GPU	Memória RAM
DT1	Core i5 4440	RX 6600 8GB	16GB DDR3 1600MHz Dual Channel
DT2	Core i7 10700	RTX 2060 6GB	16GB DDR4 1600MHz Dual Channel
NB	Ryzen 5 3500U	RX Vega 2GB	8GB DDR4 2400MHz Dual Channel

Fonte: elaborada pelo autor.

Cada máquina possui uma configuração distinta para abranger diferentes cenários. O PC *Desktop* DT1 tem uma placa de vídeo mais robusta, enquanto sua CPU é mais antiga, o que pode criar um gargalo em jogos que precisam de muita CPU. Já o *Notebook* NB possui uma CPU mais moderna e com vários núcleos, ótimo para *multithreading*, entretanto tem uma GPU menos potente, o que também pode causar gargalo, mas agora em jogos que fazem muito uso de GPU. O PC *Desktop* DT2 possui a configuração mais equilibrada, com CPU e GPU potentes.

4.7 Análise dos resultados dos testes

Nesta última etapa, foram reunidos todos os dados extraídos dos testes e então feitas inferências e conclusões a partir deles.

Os testes em que os FPS permaneceram menor ou igual a 1 minuto de tempo decorrido deste o início do monitoramento por foram considerados como injogáveis e foram encerrados após este tempo.

5 SIMULADOR “METAL RAIN”

Metal Rain foi feito com o intuito de simular uma batalha de tanques, e ao mesmo tempo buscar mecânicas para comparar o DOD com o DOO. Portanto não se trata de um simulador com elementos de realismo, e sim mais casual, a fim de se aproximar da ideia de um jogo real e ter uma comparação determinística entre as duas versões.

A ideia do simulador consiste em dois exércitos de tanques – time vermelho e time verde – atacarem um ao outro até eliminar todos do time inimigo.

A versão final apresenta três tipos de tanques:

- Tanque balanceado: Dispara seu canhão em intervalos de tempo. Significa o equilíbrio;
- Tanque leve: Disparos rápidos de balas continuamente até destruir o inimigo, porém com pouco dano. Significa leveza;
- Tanque caçador: Dispara um super tiro letal (de muito dano) em intervalos relativamente longos de tempo. Significa a letalidade;

Figura 14 – Tanques balanceado, caçador e leve com respectivamente cores verde, preto e amarelo

Fonte: Capturado pelo autor

A ordem dos acontecimentos do simulador é a seguinte:

1. Carregamento do cenário: É literalmente o início, onde o cenário é carregado e o simulador fica esperado algum comando para ir para a próxima etapa;
2. *Spawn* dos dois times: Ficou definido que a tecla **R** carrega os tanques do time vermelho no mapa e a tecla **G** o time verde. Cada time fica agrupado com seus tanques mirando na direção do time inimigo;
3. Combate: A tecla **Espaço** inicia o combate, cada tanque então escolhe um inimigo baseado numa série de critérios compõem a IA do tanque e atira no inimigo até destruí-lo;
4. Fim: Quando o último tanque inimigo é destruído acontece o fim, o time que destruiu

todos os tanques é considerado o vencedor;

A câmera possui uma posição fixa durante toda a simulação na duas versões, além de contarem também com o mesmo posicionamento e ângulo de visão.

Os arquivos fonte do simulador pode ser encontrados em https://github.com/mvinicius2k/Metal-Rain, podem ser baixados e compilados por qualquer um.

5.1 Assets

Assets são basicamente os recursos do mundo do simulador, como modelos 3D, texturas, áudio, música. No caso deste experimento não há a presença de efeitos visuais e nem sonoros, ou seja, somente *assets* de modelos 3D e textura foram usados.

Os modelos e textura dos tanques foram obtidos por meio de *asset* de terceiro baixado na plataforma *cgtrader*, feito pelo usuário OK3D (disponível em *https://www.cgtrader.com/products/low-poly-armored-vehicles-pack*), ao todo 3 modelos foram usados. A textura dos tanques possui uma leve alteração da coloração para facilitar a distinção entre as classes de tanques. O *skybox* (imagem de plano de fundo) também foi obtido por *assets* de terceiros, na *Unity Asset Store*, feito pelo usuário *PULSAR BYTES* (disponível em *https://assetstore.unity.com/packages/2d/textures-materials/sky/starfield-skybox-92717*). As balas que os tanques disparam, por outro lado, foram produzidos pelo próprio autor e possuem um modelo próprio para cada classe de tanque. O plano e textura sob o qual os tanques ficam também foi produzido pelo próprio autor deste projeto.

Desse modo os tanques ficam num espaço estelar em cima de um plano de textura bem abstrata. A ideia por trás dessa escolha artística é fazer parecer que tudo é uma simulação.

5.2 Combate

É a etapa mais complexa da simulação, por isso merece uma atenção especial. A primeira coisa que um tanque faz ao iniciar o combate é procurar um inimigo, para isso ele lê os dados de todos os tanques do time inimigo e calcula e distância para cada um deles, após isso ordena os tanques inimigos pela distância em sua memória e testa se são atingíveis, isto é, se não há nenhum tanque amigo no meio do percurso do qual as balas poderão se movimentar.

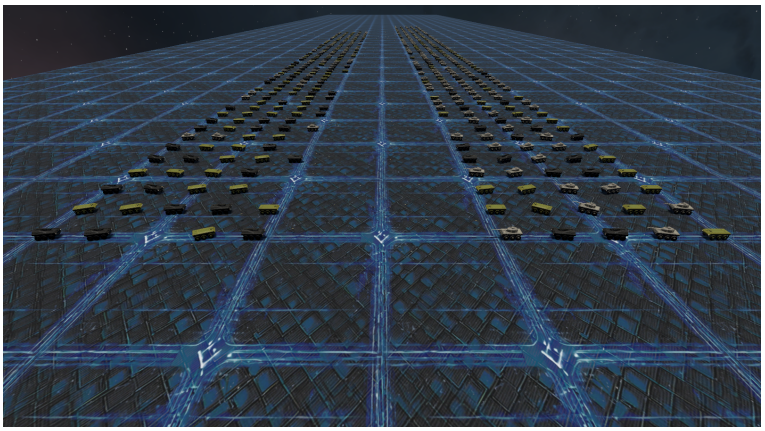
Cada tanque possui uma precisão distinta: tanques caçadores são ultra precisos, já tanques leves o contrário. Isso se reflete sobre a IA de cada tanque, onde dependendo da precisão,

o tanque pode acabar tendo que escolher aleatoriamente entre os 3 inimigos atingíveis mais próximos, 5 ou mais, depende da classe. Isso se traduz em código numa lista de tamanho *n*, tal que *n* é igual ao valor da precisão, após preencher esta lista, um destes tanques é escolhido aleatoriamente como inimigo.

Com o inimigo definido, o tanque começa atirar na direção deste inimigo. As balas viajam pelo espaço, e ao colidirem com o tanque inimigo (ou qualquer tanque inimigo), os pontos de vida são decrementados de acordo com o dano do tanque dono da bala.

A IA não é baseada em nenhum algoritmo e a implementação foi feita com base na conveniência de ser algum nível de complexidade e ao mesmo tempo ser simples dado o escopo e tempo para a realização do trabalho.

Figura 15 – Tanques dos dois times atirando uns contra os outros



Fonte: Capturado pelo autor

Quando um tanque é destruído, todos os tanques que estavam mirando nele reiniciam o processo para buscar um novo tanque a mirar.

As interações que o jogador faz é somente controlar quando cada ação do simulador acontece, ou seja, ao apertar o botão “R” do teclado os tanques do time vermelho são gerados, ao apertar “G” os tanques do time verde são gerados e ao apertar espaço os tanques iniciam o combate. Para garantir que a simulação seja igual para as duas versões, o controle do jogador foi reduzido até este ponto.

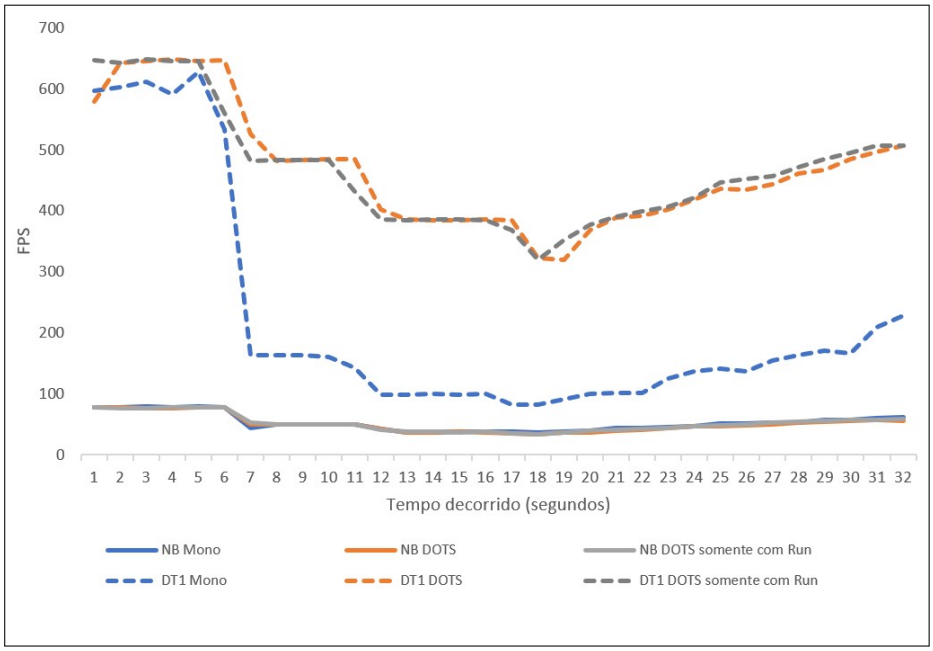
6 RESULTADOS

Os dados extraídos mostraram que a versão DOD se saiu com desempenho superior nos testes, além de usar de forma mais eficiente os núcleos da CPU. As métricas de memória pouco variaram entre as versões.

6.1 Experimento com 240 tanques

Neste teste, foram instanciados 120 tanques do time verde e 120 tanques do time vermelho, totalizando 240 tanques ao total. No, PC DT1 e DT2 (as versões DOD) se saíram bem melhor que a versão DOO, alcançando altas taxas de quadro e lidando com folga com a carga de trabalho. Entretanto na máquina NB os testes tiveram pouquíssimas variações.

Figura 16 – Gráfico de FPS comparando o teste no PC DT1 e no *Notebook* NB



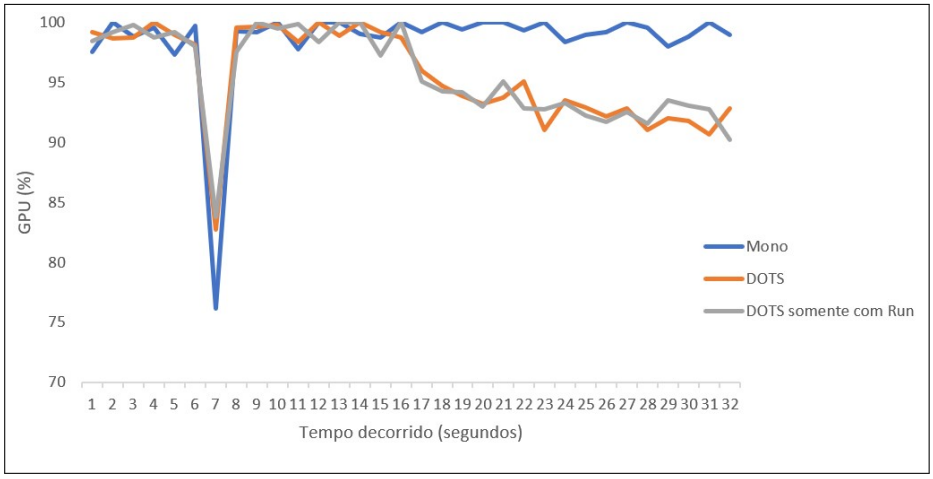
Fonte: Elaborada pelo autor.

Analisando mais estatísticas, ocorre uma situação de gargalo na GPU, onde ela demora mais a terminar as tarefas que a CPU, atrasando todo o fluxo. A Figura 18 mostra, em todas as versões, todos os núcleos operando numa porcentagem baixa, enquanto, na Figura 17, a GPU está sempre beirando os 100% de uso.

Isso acontece porque o Design Orientado a Dados traz fortes otimizações para a CPU, fazendo com que ela trabalhe de forma mais eficiente. Como máquina NB possui já um processador forte que termina as tarefas antes da GPU para o caso do experimento, o ganho de

desempenho em FPS fica imperceptível entre as versões DOO e DOD. Apesar disso, a versão DOD entrega mais eficiência, podendo trazer ganhos de energia por exemplo.

Figura 17 – Gráfico do uso de GPU da máquina NB



Fonte: Elaborada pelo autor.

6.2 Experimento com 1700 tanques

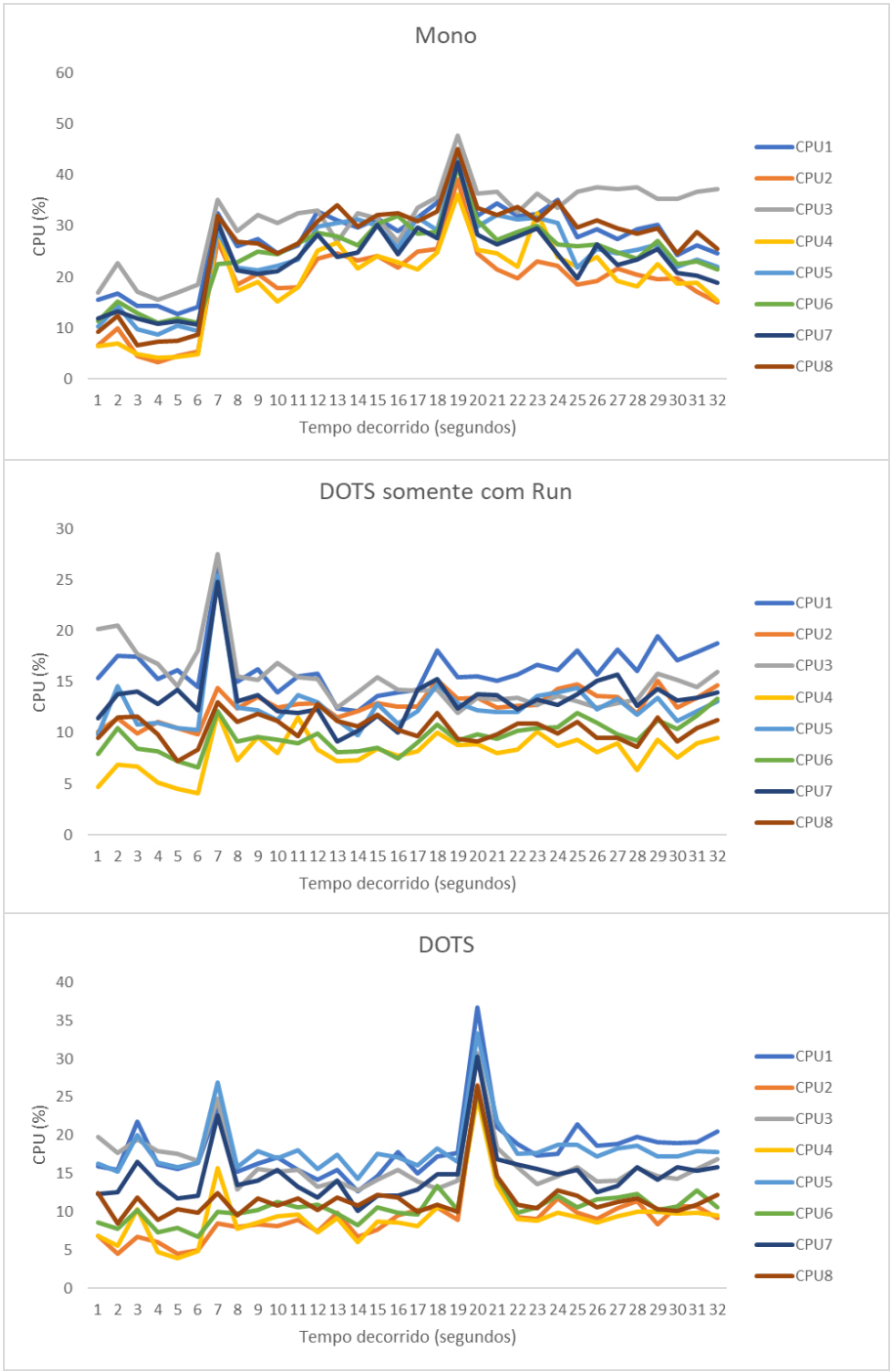
Aumentando o número de instanciações nos testes, quando configurado para gerar 700 tanques para o time verde e 1000 para o verde, as versões *MonoBehaviour* ficaram injogáveis para NB e DT1, com FPS abaixo de 5 por minutos.

As versões DOD do PC DT1 conseguiram suportar bem este teste. A versão onde os *Jobs* são executados na *thead* principal sofreu para manter uma taxa de quadros fluido, chegando a 3 FPS no início do combate (onde tem maior caos) e a medida que os tanques iam se destruindo, os FPS aumentavam. Já a versão DOD sem limitações conseguiu se manter acima de 60 FPS durante todo este teste, além disso distribui bem a carga entre os vários núcleos disponíveis da CPU.

Para o Notebook NB, o cenário foi uma pouco menos fluído que o DT1, chegando a 10 FPS na versão DOD sem limitações e a 3 FPS na com execução de *Jobs* na *thread* principal. Em ambos a taxa de quadro aumenta gradativamente até chegar no gargalo da GPU, estacionando nos 40 FPS.

No PC DT2, a versão *MonoBehaviour* conseguiu suportar o teste sem travar um quadro por vários segundos. Apesar de ter chegado a 3 FPS no ápice do caos da cena, gradativamente foi ganhando fluidez conforme os tanque diminuíam na cena. Porém longe de ter fluidez até que maior parte dos tanques estivessem destruídos. A versão DOD com *Run* executou todo o

Figura 18 – Gráfico do uso de todos os núcleos da CPU da máquina NB no experimento com 240 tanques



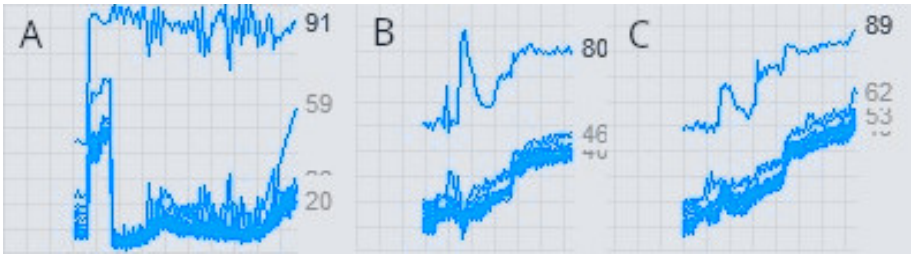
Fonte: Elaborada pelo autor.

teste acima de 30 FPS e a versão sem limitações acima de 75 FPS.

A Figura 19 mostra como as versões DOD balanceiam bem a carga entre os núcleos da CPU, enquanto a versão DOO tem um único núcleo trabalhando com quase toda a capacidade e os demais com baixo uso. O gráfico C, que é a versão DOD sem limitações, mostra um melhor aproveitamento dos núcleos da CPU. Apesar de estar ligeiramente usando mais a CPU que o

gráfico B – versão DOD com *Jobs* executados em uma única *thread* –, isso pode ser explicado pelos mecanismos internos da *engine* que operam sem a responsabilidade do programador.

Figura 19 – Gráfico do uso de todos os núcleos da CPU da máquina DT2 no experimento com 1700 tanques, tal que A é a versão *MonoBehaviour*, B a versão DOD com *Jobs* na *thread* principal e C a versão DOD sem limitações.



Fonte: Capturado pelo autor.

Para este trabalho, a análise estática de código não foi feita, pois o *scripts* DOTS, principalmente os *Systems* e *Aspects* fazem uso de classes parciais do C#. Classes parciais são classes que podem ser complementadas posteriormente em outros arquivos de código fonte C#, o compilador une todas as classes parciais num só tipo. Os analisadores de código estático acabam capturando as partes de classes parciais do próprio DOTS, entregando estatísticas que estão fora do escopo do trabalho.

7 CONCLUSÃO

Este trabalho reafirma a eficiência do DOD em entregar alto desempenho e consequentemente eficiência no uso de recursos de *hardware*, sendo muito bem vindo para jogos complexos que precisam de muito processamento para funcionar.

No entanto, dependendo dos requisitos e objetivos de um jogo, a situações do gargalo de GPU pode acontecer, não sendo possível aproveitar todos os benefícios que o DOD traz. Por outro lado, a eficiência é onipresente em casos com ou sem gargalo, sendo uma escolha interessante para quem quer um uso eficiente e equilibrado da CPU.

Jogos *CPU Bound*, isto é, que fazem uso intenso da CPU, são os que mais podem se aproveitar do Design Orientado a Dados, as otimizações e ganhos de desempenho são perceptíveis e podem resolver inclusive problemas de gargalo em CPU.

7.1 Limitações

Comparar paradigmas é algo difícil de alcançar um resultado que diga quem é o melhor, e este projeto não é sobre quem é o melhor. Esta é uma questão talvez sem resposta dada a quantidade de possibilidades que este assunto está ligado. Jogos são complexos e interligam, além de várias artes, vários problemas de programação. Este trabalho buscou analisar dentro de um cenário o Design Orientado a Dados e o Design Orientado a Objetos.

Por se tratar de um paradigma emergente, o pouco conhecimento pode impactar diretamente na qualidade de código, resolução de problemas e até mesmo no próprio desempenho, não se aproveitando então de todos os benefício que o DOD provê. Além disso, o autor deste projeto teve contato com o Design Orientado a Dados por pouco tempo em relação ao Design Orientado a Dados.

Dessa forma, por ser um experimento de uma pessoa, a diferença de conhecimento entre os dois paradigmas pode ter algum impacto no resultado final. Em termos práticos isso se traduz em otimizações e soluções mais eficiente de problemas, coisas que dependem da habilidade do desenvolvedor.

7.2 Trabalhos futuros

Sobre o tema ainda há um longo caminho a percorrer. Tecnologias emergentes de aumento de desempenho estão constantemente sendo lançadas. Ainda não está claro o impacto

que o DOD tem dentro do ambiente de uma desenvolvera, uma vez que o DOO é o paradigma vigente na indústria de *software* como um todo. Como os desenvolvedores se adaptariam e qual qualidade de código produzido pelo paradigma DOD são questões que precisam de um estudo.

REFERÊNCIAS

ACTON, M.; GAMES, I. Data-oriented design and c++. **Luento. CppCon**, 2014.

AMPATZOGLOU, A.; CHATZIGEORGIOU, A. Evaluation of object-oriented design patterns in game development. **Information and Software Technology**, v. 49, n. 5, p. 445–454, maio 2007. ISSN 09505849. Disponível em: https://linkinghub.elsevier.com/retrieve/pii/S0950584906000929.

BARI, M. A. **Creating Complex AI Behavior in 'Stellaris' Through Data Driven Design**. 2017. GDC. Disponível em: https://www.gdcvault.com/play/1024223/Creating-Complex-AI-Behavior-in. Acesso em: 4 jul. 2023.

BAYLISS, J. D. The Data-Oriented Design Process for Game Development. **Computer**, v. 55, n. 5, p. 31–38, maio 2022. ISSN 0018-9162, 1558-0814. Disponível em: https://ieeexplore.ieee.org/document/9771161/.

COLLIN, D. **Culling the Battlefield: Data Oriented Design in Practice**. 2011. GDC. Disponível em: https://www.gdcvault.com/play/1014491/Culling-the-Battlefield-Data-Oriented. Acesso em: 4 jul. 2023.

FABIAN, R. **Fabian - Data-oriented design**. [*S. n.*], 2018. v. 21. Disponível em: https://www.dataorienteddesign.com/dodbook/node2.html.

FARYABI, W. **Data-oriented Design approach for processor intensive games**. Dissertação (Mestrado) – NTNU, 2018.

FEDOSEEV, K.; ASKARBEKULY, N.; UZBEKOVA, A. E.; MAZZARA, M. Application of Data-Oriented Design in Game Development. **Journal of Physics: Conference Series**, v. 1694, n. 1, p. 012035, dez. 2020. ISSN 1742-6588, 1742-6596. Disponível em: https://iopscience.iop.org/article/10.1088/1742-6596/1694/1/012035.

GAMMA, E.; HELM, R.; JOHNSON, R.; JOHNSON, R. E.; VLISSIDES, J.; others. **Design patterns: elements of reusable object-oriented software**. [*S. l.*]: Pearson Deutschland GmbH, 1995.

GOLDSTONE, W. **Unity 3. x game development essentials**. [*S. l.*]: Packt Publishing Ltd, 2011.

HUSSAIN, A.; SHAKEEL, H.; HUSSAIN, F.; UDDIN, N.; GHOURI, T. L. Unity Game Development Engine: A Technical Survey. p. 10, jun. 2020.

HäRKöNEN, T. ADVANTAGES AND IMPLEMENTATION OF ENTITY-COMPONENT-SYSTEMS. p. 31, abr. 2019.

JULIANI, A.; BERGES, V.-P.; TENG, E.; COHEN, A.; HARPER, J.; ELION, C.; GOY, C.; GAO, Y.; HENRY, H.; MATTAR, M.; LANGE, D. **Unity: A General Platform for Intelligent Agents**. arXiv, 2020. ArXiv:1809.02627 [cs, stat]. Disponível em: http://arxiv.org/abs/1809.02627.

LINTRAMI, T. **Unity 2017 Game Development Essentials: Build fully functional 2D and 3D games with realistic environments, sounds, physics, special effects, and more!** [*S. l.*]: Packt Publishing Ltd, 2018.

LLOPIS, N. Data-oriented design (or why you might be shooting yourself in the foot with OOP). **Game Developer Magazine**, v. 16, n. 8, 2009. Disponível em: https://gamesfromwithin.com/data-oriented-design.

MICROSOFT. **O que é "código gerenciado"?** 2022. Disponível em: https://learn.microsoft.com/pt-br/dotnet/standard/managed-code. Acesso em: 13 jul. 2022.

NAKOV, S.; KOLEV, V. **Fundamentals of Computer Programming with C#: The Bulgarian C# Book**. [*S. l.*]: Faber Publishing, 2013.

NIKOLOV, S. Oop is dead, long live data-oriented design. **CppCon.(Retrieved 02.02. 2019)**, 2018.

NäYKKI, A. Unity DOTS in production: DOTS pathfinding implementation in VR & AR. 2021.

PARVIAINEN, N. Dependency injection in unity3d. Jyväskylän ammattikorkeakoulu, 2017.

PENNYCOOK, J.; HAMMOND, S.; WRIGHT, S.; HERDMAN, A.; MILLER, I.; JARVIS, S. An investigation of the performance portability of opencl. **Journal of Parallel and Distributed Computing**, v. 73, p. 1439–1450, 11 2013.

SALMELA, T. Game Development Using the Open-Source Godot Game Engine. p. 67, 2022.

TANENBAUM, A. **Sistemas Operacionais Modernos**. Pearson Universidades, 2015. ISBN 9788543005676. Disponível em: https://books.google.com.br/books?id=0bnzzwEACAAJ.

TURPEINEN, M. A Performance Comparison for 3D Crowd Rendering using an Object-Oriented system and Unity DOTS with GPU Instancing on Mobile Devices. p. 19, jun. 2020.

UNITY. **What are the optimizations that Burst compiler do?** 2021. Disponível em: https://forum.unity.com/threads/what-are-the-optimizations-that-burst-compiler-do.1208473/#post-7715599. Acesso em: 3 jul. 2022.

UNITY. **Burst**. 2022. Disponível em: https://docs.unity3d.com/2022.2/Documentation/Manual/com.unity.burst.html. Acesso em: 21 nov. 2022.

UNITY. **C Job System**. 2022. Disponível em: https://docs.unity3d.com/Manual/JobSystem.html. Acesso em: 5 jul. 2022.

UNITY. **Performance Testing Extension for Unity Test Runner**. 2022. Disponível em: https://docs.unity3d.com/Packages/com.unity.test-framework.performance@1.3/manual/index.html. Acesso em: 24 nov. 2022.

UNITY. **Profiler overview**. 2022. Disponível em: https://docs.unity3d.com/2022.2/Documentation/Manual/Profiler.html. Acesso em: 20 nov. 2022.

UNITY. **Scripting backends**. 2022. Disponível em: https://learn.microsoft.com/pt-br/dotnet/standard/managed-code. Acesso em: 02 nov. 2022.

UNITY. **What is DOTS and why is it important?** 2022. Disponível em: https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important. Acesso em: 01 jul. 2022.

UNITY. **Are Physics Scenes multithreaded?** 2023. Disponível em: https://forum.unity.com/threads/are-physics-scenes-multithreaded.1405525/#post-8855257. Acesso em: 06 jul. 2023.

UNITY. **Entities package concepts**. 2023. Disponível em: https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-intro.html. Acesso em: 26 jun. 2023.

ZHOU, H. L. Z. **Deep Dive into Data-Oriented Design for a Cross-platform UGC Game Engine (Yahaha): Stateful, Assets, Synchronization, Performance (Presented by YAHAHA Studios)**. 2023. GDC. Disponível em: https://www.gdcvault.com/play/1029021/Deep-Dive-into-Data-Oriented. Acesso em: 4 jul. 2023.

APÊNDICE A – DOCUMENTO DE PLANEJAMENTO DO SIMULADOR *METAL RAIN*

O documento de planejamento contém alguns detalhes de como o simulador *Metal Rain* foi idealizado.

Introdução

Metal Rain é um jogo de batalha simulada de tanques entre dois lados. Inicialmente cada lado começa com uma quantidade de pontos de energia para investir no seu exército e então o libera em partes para enfrentar o outro exército. Quem perder todas as entidades do exército, perde o jogo.

Elementos

Tanque comum

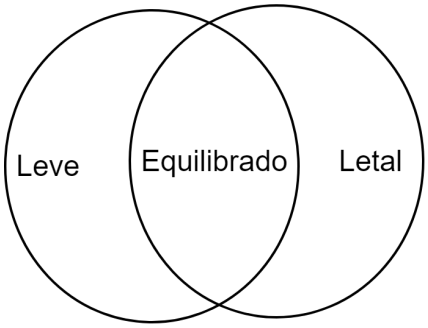
Dispara seu canhão em intervalos de tempo. Significa equilíbrio.

Tanque metralhador

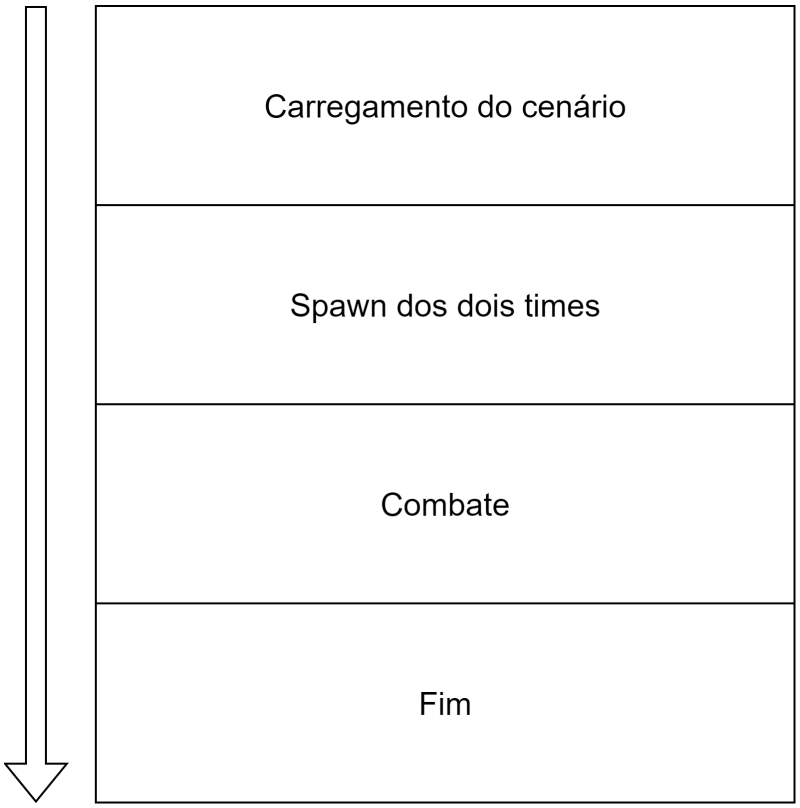
Dispara rajadas rápidas de balas continuamente até destruir o inimigo. Significa leveza.

Tanque destroçador

Dispara um super tiro letal (de muito dano) em intervalos relativamente longos de tempo. Significa a letalidade. Custa 9 R.



Ações



Fluxo

Configurações de Combate

O cenário é carregado e o simulador fica aguardando o comando para gerar os times.

Spawn dos dois times

Os tanques são spawnados, um time de cada vez.

Combate

Logo após se iniciar a partida, o jogador é livre para mover os tanques como bem entender e spawnar novos tanques, desde que hajam recursos R para isso.

Combate

Os tanques atiram nos inimigos até o não sobrar mais nada.

Fim

Algum time tem seus tanques todos destruídos e então perde.

Controles

Os controles são basicamente o fluxo da simulação

Descrição	Comando
R	Spawn do time vermelho
G	Spawn do time verde
Espaço	Iniciar batalha

História

Guerras agora não são mais feitas com pessoas, e sim com máquinas. A vitória não é mais decidida pelos soldados, mas sim pela tecnologia e as habilidades do comandante.