UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE RAMOS DE BARROS

UM COMPARATIVO ENTRE MÉTODOS DE OTIMIZAÇÃO DE VIZINHANÇA EM DISPOSITIVOS MÓVEIS USANDO O UNITY DOTS

RIO DE JANEIRO
2024

FELIPE RAMOS DE BARROS

UM COMPARATIVO ENTRE MÉTODOS DE OTIMIZAÇÃO DE VIZINHANÇA EM DISPOSITIVOS MÓVEIS USANDO O UNITY DOTS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Silvana Rossetto

RIO DE JANEIRO

2024

CIP - Catalogação na Publicação

B277c

Barros, Felipe Ramos de
Um comparativo entre métodos de otimização de vizinhança em dispositivos móveis usando o Unity DOTS / Felipe Ramos de Barros. -- Rio de Janeiro, 2024.
54 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. boids. 2. Unity DOTS. 3. jogos digitais em celulares. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

FELIPE RAMOS DE BARROS

UM COMPARATIVO ENTRE MÉTODOS DE OTIMIZAÇÃO DE VIZINHANÇA EM DISPOSITIVOS MÓVEIS USANDO O UNITY DOTS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 19 de agosto de 2024

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
SILVANA ROSSETTO
Data: 22/08/2024 14:42:58-0300
Verifique em https://validar.iti.gov.br

Silvana Rossetto
Instituto de Computação - UFRJ

gov.br Documento assinado digitalmente
JOAO CARLOS PEREIRA DA SILVA
Data: 22/08/2024 16:49:11-0300
Verifique em https://validar.iti.gov.br

João Carlos Pereira da Silva
Instituto de Computação - UFRJ

gov.br Documento assinado digitalmente
DANIEL GREGORIO ALFARO VIGO
Data: 22/08/2024 21:06:06-0300
Verifique em https://validar.iti.gov.br

Daniel Gregorio Alfaro Vigo
Instituto de Computação - UFRJ

Dedico este trabalho aos meus pais Fernando e Simone que sempre me apoiaram e trabalharam para que eu pudesse ter capacidade de realizar tudo que sonhei.

AGRADECIMENTOS

Agradeço a oportunidade de conclusão desta etapa aos professores da Universidade Federal do Rio de Janeiro, em especial a Prof. Silvana Rossetto que acreditou na minha capacidade realizar este trabalho e me deu apoio durante todo o processo.

Agradeço também aos meus colegas da GDP que me incentivaram e em momentos cruciais souberam compartilhar dos seus conhecimentos e dar boas sugestões.

Agradeço pelo suporte emocional que pude ter durante parte deste processo com minha psicóloga Kelly Azevedo, que me ajudou a me entender como uma pessoa neuroatípica, e me munir com ferramentas para que pudesse concluir minha graduação mesmo com os obstáculos diários que tive que enfrentar.

Agradeço também a minha família por acreditar em mim e querer sempre o meu melhor. Sou grato também pelos incentivos recebidos pelo meu namorado Léo e nossos dois gatos, Bentinho e Minduim, que me fizeram companhia enquanto escrevia esse trabalho.

RESUMO

Com a popularização dos jogos de celular e o avanço tecnológico dos aparelhos disponíveis para o público surgiu a necessidade de desenvolver jogos que sejam capazes de utilizar o máximo da capacidade que esses dispositivos têm a oferecer e criar experiências inovativas para os jogadores. Esse trabalho teve como objetivo fazer um comparativo entre estratégias de otimização do algoritmo de *boids* dentro de uma arquitetura orientada a dados utilizando a ferramenta Unity DOTS em celulares. O algoritmo de *boids* é um algoritmo de agentes distribuídos cujo objetivo é simular o comportamento de bandos de animais como pássaros e peixes. Por sua simplicidade de implementação e resultados fáceis de serem conferidos, esse algoritmo é usado para testar a capacidade de sistemas de processar uma simulação com grande número de objetos que interagem entre si. As estratégias avaliadas, que tiveram como foco otimizar a descoberta de vizinhos de um *boid*, foram: força-bruta, *hashing* espacial e utilizando uma estrutura de dados didimensional. Ao avaliar a média de *frames por segundos* (FPS) e os intervalos de tempo de processamento de CPU e GPU para cada uma das estratégias experimentadas em casos de testes com diferentes números de elementos na simulação, foi encontrada uma vantagem para a estrutura de dados bidimensional em relação às outras estratégias.

Palavras-chave: algoritmo de boids; Unity DOTS; jogos digitais em celulares

ABSTRACT

With the popularization of mobile games and the technological advancements of devices available to the public, there has been a need to develop games that can utilize the full capabilities of these devices and create innovative experiences for players. This work aimed to compare optimization strategies for the *boids* algorithm within a data-oriented architecture using Unity DOTS on mobile phones. The *boids* algorithm is a distributed agent algorithm designed to simulate the behavior of groups of animals such as birds and fish. Due to its simplicity of implementation and easy-to-verify results, this algorithm is used to test the capability of systems to process simulations with a large number of interacting objects. The evaluated strategies, which focused on optimizing the discovery of neighboring *boids*, were: brute force, spatial hashing, and using a two-dimensional data structure. By evaluating the average *frames per second* (FPS) and CPU and GPU processing times for each of the strategies tested in simulations with varying numbers of elements, a performance advantage was found for the two-dimensional data structure compared to the other strategies.

Keywords: boids; Unity DOTS; mobile games

LISTA DE ILUSTRAÇÕES

Figura 1 – Captura de tela da implementação de Boids pela Unity 28
Figura 2 – Captura de tela da implmentação em execução com 16384 boids 29
Figura 3 – Exemplo de Hashing Espacial 32
Figura 4 – Exemplo de representação de uma simulação em um grid de vizinhança 32
Figura 5 – Vizinhança extendida de Moore que define os vizinhos de um boid na matrix 33
Figura 6 – Samsung S10e - FPS 35
Figura 7 – Redmi Note 11 - FPS 36
Figura 8 – Samsung S10e - CPU 37
Figura 9 – Redmi Note 11 - CPU 37
Figura 10 – Samsung S10e - GPU 38
Figura 11 – Redmi Note 11 - GPU 39

LISTA DE CÓDIGOS

Código 1 Exemplo de definição da estrutura boid em pseudo-código 16
Código 2 Exemplo do laço de simulação em pseudo-código 16
Código 3 Exemplo de implementação da regra de coesão em Pseudo-código . 17
Código 4 Exemplo de implementação da regra de separação em pseudo-código 17
Código 5 Exemplo de implementação da regra de alinhamento em pseudo-código 17
Código 6 Exemplo de IJob 19
Código 7 Exemplo de IJobParallelFor 20
Código 8 Scheduling dos Jobs 21
Código 9 Burst Compile 22
Código 10 Implementação em MonoBehaviour 23
Código 11 Implementação em ECS 24
Código 12 Boid Component 43
Código 13 Boid Authoring 44
Código 14 SteerBoidJob - Brute Force 45
Código 15 InitialPerBoidJob - Brute Force 46
Código 16 ApplyRules - Brute Force 47
Código 17 InitialPerBoidJob - Spatial Hashing 48
Código 18 HashPosition - Spatial Hashing 48
Código 19 ApplyRules - Spatial Hashing 49
Código 20 InitialPerBoidJob - Sort 50
Código 21 ApplyRules - Sort 51
Código 22 RowSort - Sort 52
Código 23 ColumnSort - Sort 53
Código 24 PartialSort - Sort 54
Código 25 FullSort - Sort 54

LISTA DE TABELAS

Tabela 1 – Média de FPS limitado a 60 - Samsung S10e 36
Tabela 2 – Média de FPS limitado a 60 - Redmi Note 11 36
Tabela 3 – Tempo de execução em CPU (ms) - Samsung S10e 37
Tabela 4 – Tempo de execução em CPU (ms) - Redmi Note 11 38
Tabela 5 – Tempo de execução em GPU (ms) - Samsung S10e 38
Tabela 6 – Tempo de execução em GPU (ms) - Redmi Note 11 38

LISTA DE ABREVIATURAS E SIGLAS

ECS	Entity Component System
DOTS	Data Oriented Technology Stack
SIMD	Single Instruction, Multiple Data
FPS	Frames por segundo

SUMÁRIO

1 INTRODUÇÃO 13

2 CONCEITOS BÁSICOS 15
2.1 ALGORITMO DE BOIDS 15
2.1.1 O Boid 15
2.1.2 Fluxo Principal 16
2.1.3 Coesão 16
2.1.4 Separação 17
2.1.5 Alinhamento 17
2.1.6 Definição de Vizinhança 18
2.1.7 Comportamento Esperado 18
2.2 UNITY DOTS 18
2.2.1 C# Job System 19
2.2.2 Burst Compiler 22
2.2.3 Entity Component System 22

3 BREVE HISTÓRICO 25
3.1 ALGORITMO DE BOIDS 25
3.2 UNITY DOTS 26

4 IMPLEMENTAÇÃO 28
4.1 IMPLEMENTAÇÃO PRÓPRIA DA UNITY 28
4.2 DESCRIÇÃO DA IMPLEMENTAÇÃO 29
4.2.1 BoidComponent 30
4.2.2 BoidAuthoring 30
4.2.3 Systems 31
4.3 FERRAMENTAS E AMBIENTES DE DESENVOLVIMENTO 33

5 AVALIAÇÃO E RESULTADOS 34
5.1 DETALHES TÉCNICOS 34
5.2 METODOLOGIA 34
5.3 RESULTADOS 35
5.3.1 FPS 35
5.3.2 Tempo de frame - CPU 36
5.3.3 Tempo de frame - GPU 36

6 CONCLUSÃO 40

REFERÊNCIAS **41**

APÊNDICE A – EXCERTOS DE CÓDIGO DA IMPLEMENTAÇÃO **43**

1 INTRODUÇÃO

Ano após ano os dispositivos móveis têm se tornado mais potentes com novas tecnologias em processamento, telas e múltiplos núcleos de CPU. Isso tem chamado a atenção de desenvolvedores de jogos que querem explorar o mercado de jogadores *mobile*. Entretanto os sistemas de dispositivos móveis ainda é um fator limitante para o que é possível desenvolver e executar de forma otimizada, tornando-se necessário buscar formas de melhor aproveitar os componentes presentes nos aparelhos. O que se espera é possibilitar que os jogos desenvolvidos mantenham sua qualidade em uma gama maior de dispositivos compatíveis com diferentes capacidades de processamento, o que por sua vez, resulta em um número maior de jogadores.

A ferramenta de desenvolvimento escolhida por grande parte dos times de desenvolvimento de jogos, especialmente os desenvolvedores de jogos de celular, é a *Unity Engine*[1] por ser um motor de jogos bastante acessível e versátil para times de desenvolvimento de pequeno porte. Outros motores como *Godot*[2] e *Unreal Engine*[3] também são bastante utilizados.

Em 2018 a Unity Engine anunciou o Data Oriented Technology Stack (DOTS)[4] para auxiliar na criação de jogos ainda mais otimizados. *DOTS* é formado pelo seguinte conjunto de tecnologias: Entity Component System (ECS), uma arquitetura voltada a aumentar a localidade de informações na memória e assim alcançar melhor desempenho; o *Burst Compiler*, um compilador capaz de identificar possíveis pontos de otimização através do uso de instruções SIMD; e o *C# Job System* que cria uma camada de alto nível que facilita a escrita de código concorrente correto e de bom desempenho. Esse conjunto de tecnologias encontra-se ainda em fase experimental, porém apresenta potencial de exploração acerca de suas aplicações, especialmente em dispositivos móveis. Vê-se então necessário estudos que aprofundem o conhecimento dessa ferramenta e avaliações quantitativas que mostrem a extensão de sua utilidade também neste universo.

Dado um conjunto de números, há métodos elementares para calcular o seu Entity Component System, que é abreviado ECS. Este processo é similar ao utilizado para o Entity Component System (ECS).

Atualmente existem jogos que revelam um potencial de uso de arquiteturas orientadas a dados em dispositivos móveis. Torna-se ainda mais relevante explorar os potenciais que esse uso possibilita tendo em vista os casos de sucesso de jogos como Overcrowded:

[1] https://unity.com/
[2] https://godotengine.org/
[3] https://www.unrealengine.com/
[4] https://unity.com/dots

Tycoon [5] e Survivor.io [6], os quais apresentam um grande número de elementos simultâneos que interagem entre si.

O objetivo deste trabalho é fazer um comparativo entre estratégias de otimização do algoritmo de *boids* (REYNOLDS, 1987) dentro de uma arquitetura orientada a dados utilizando a ferramenta Unity DOTS em celulares. O algoritmo de *boids* é um algoritmo de agentes distribuídos cujo objetivo é simular o comportamento de bandos de animais como pássaros e peixes. Por sua simplicidade de implementação e resultados fáceis de serem conferidos, esse algoritmo é usado para testar a capacidade de sistemas de processar uma simulação com grande número de objetos que interagem entre si.

As estratégias avaliadas, que tiveram como foco otimizar a descoberta de vizinhos de um *boid*, foram: força-bruta, *hashing* espacial e utilizando uma estrutura de dados bidimensional. Ao avaliar a média de *Frames por segundo (FPS)* e os intervalos de tempo de processamento de CPU e GPU para cada uma das estratégias experimentadas em casos de testes com diferentes números de elementos na simulação, foi encontrada uma vantagem para a estrutura de dados bidimensional em relação às outras estratégias.

O restante deste texto está organizado da seguinte forma. No Capítulo 2 são apresentados conceitos básicos do escopo tratado. No Capítulo 3 alguns trabalhos relacionados são descritos. No Capítulo 4 os objetivos e decisões de implementação do protótipo elaborado são discutidas. No Capítulo 5 descreve-se a metodologia de avaliação e os resultados obtidos. Por fim, no Capítulo 6 as conclusões do trabalho são apresentadas.

[5] https://www.zeptolab.com/games/overcrowded-tycoon
[6] https://www.habby.com/game/detail/survivor

2 CONCEITOS BÁSICOS

Nesse capítulo serão apresentados os conceitos básicos que serão utilizados para realizar o comparativo proposto. Primeiramente será introduzido o algoritmo de simulação de *boids* que será utilizado como base da implementação e a definição de vizinhança onde se concentram as estratégias de otimização. Em seguida, temos uma breve apresentação do *Unity DOTS* com seu conjunto de ferramentas utilizadas na implementação final.

2.1 ALGORITMO DE BOIDS

O algoritmo de *boids* é um algoritmo de agentes distribuídos proposto por Reynolds com o objetivo de simular o comportamento de bandos de animais como pássaros e peixes, entre outros (REYNOLDS, 1987). O algoritmo parte do princípio de comportamento emergente, isto é: quando um número de indivíduos interagem entre si de forma direta ou indireta gerando resultados muito mais complexos que suas interações unitárias. Um comportamento emergente possui um resultado final difícil de prever mesmo quando as regras de interações entre objetos são simples (ERNEHOLM, 2011). Para gerar resultados próximos da realidade, Reynolds desenvolveu apenas três regras (Separação, Alinhamento e Coesão) que são aplicadas a cada agente da simulação para calcular sua velocidade no próximo frame, se baseando na posição relativa e direção de movimento de agentes próximos.

Por sua simplicidade de implementação e resultados fáceis de serem conferidos, esse algoritmo é bastante usado para testar a capacidade de sistemas diversos em processar uma simulação com grande número de objetos que interagem entre si, porém sem muita complexidade em seus comportamentos individuais.

2.1.1 O Boid

Cada agente na simulação, denominado "boid", é definido por Reynolds com três atributos necessários para o seu funcionamento:

a) Posição: Definida por um vetor bidimensional ou tridimensional que guarda as coordenadas do *boid* no espaço da simulação;

b) Direção: definida pelo ângulo que indica a direção do movimento do *boid*;

c) Velocidade: definido pelo módulo da velocidade do *boid* na direção de movimento.

De forma equivalente, podemos representar a direção e velocidade utilizando um único vetor de velocidade.

Código 1 – Exemplo de definição da estrutura boid em pseudo-código

```
define boid:
    vec2 pos
    float dir
    float vel
```

2.1.2 Fluxo Principal

Como podemos ver no Código 2, a simulação consiste em um único laço para cada *frame*. As iterações da simulação, quando mostradas em rápida sucessão, criam a ilusão de continuidade. Primeiramente, a partir do estado atual dos boids calcula-se as influências das regras de interação: coesão, separação e alinhamento. Então, com as alterações de direção e velocidade calculadas, a posição de cada *boid* é atualizada para que represente uma mudança em um curto espaço de tempo, condizente com suas propriedades atuais. E por fim, o estado resultante é desenhado na tela.

Código 2 – Exemplo do laço de simulação em pseudo-código

```
simulate(deltaTime):
    for each frame do:
        for each boid do:
                cohesion(boid)
                separation(boid)
                alingment(boid)
        for each boid do:
                boid.pos.x <- boid.pos.x + cos(boid.dir) * boid.vel *
                   deltaTime
                boid.pos.y <- boid.pos.y + sen(boid.dir) * boid.vel *
                   deltaTime
                draw(boid)
```

2.1.3 Coesão

O Código 3 mostra como é calculada a regra de coesão que é a tendência dos *boids* de convergir na posição média dos membros de seu bando local. Essa regra é a que permite que o bando se mantenha unido e não perca membros. Cria-se uma força de atração para o centro da vizinhança de um *boid* que é adicionada na direção geral do seu movimento.

Código 3 – Exemplo de implementação da regra de coesão em Pseudo-código

```
cohesion(boid):
    result <- (0,0)
    for each neighbour in neighbourhood(boid) do:
        result <- result + neighbour.pos
    result <- result / neighbourhood(boid).length
    boid.dir <- direction(result)
    boid.vel <- magnitude(result)
```

2.1.4 Separação

O objetivo da regra de separação entre os *boids* é impedir um agrupamento muito denso e evitar colisões entre eles. Isso é alcançado criando uma força de repulsão entre um *boid* e seus vizinhos que altera a sua direção de movimento como descrito no Código 4.

Código 4 – Exemplo de implementação da regra de separação em pseudo-código

```
separation(boid):
    result <- (0,0)
    for each neighbour in neighbourhood(boid) do:
        result <- result + boid.pos - neighbour.pos
    result <- result / neighbourhood(boid).length
    boid.dir <- direction(result)
    boid.vel <- magnitude(result)
```

2.1.5 Alinhamento

A regra de alinhamento (Código 5) existe para manter todos os *boids* de um bando se movendo com velocidades e direções parecidas, dessa forma evita-se que um *boid* tenha mudanças muito bruscas de movimento e se separe do resto do bando.

Código 5 – Exemplo de implementação da regra de alinhamento em pseudo-código

```
alignment(boid):
    dir <- 0
    vel <- 0
    for each neighbour in neighbourhood(boid) do:
        dir <- dir + boid.dir - neighbour.dir
        vel <- vel + boid.vel - neighbour.vel
    boid.dir <- boid.dir + dir / neighbourhood(boid).length
    boid.dir <- boid.vel + vel / neighbourhood(boid).length
```

2.1.6 Definição de Vizinhança

A forma como a vizinhança de um *boid* é definida é crucial para o funcionamento da simulação já que ela é o conjunto de quais outros *boids* devem ser considerados para os seus cálculos em cada regra. Cada regra deve possuir sua própria implementação de vizinhança, atentando-se ao fato de que se as regras de coesão e separação operarem sobre o mesmo conjunto elas cancelarão uma a outra e não teríamos o comportamento adequado. Portanto, é importante que a regra de coesão tenha sempre um alcance maior que a regra de separação. A forma simples de computar a vizinhança de cada *boid*, a qual consiste em iterar por todos os outros *boids* na simulação e verificar um a um se pertence ou não ao conjunto desejado, possui uma complexidade de tempo de $O(n^2)$, onde n representa o número de boids na simulação. É justamente nessa complexidade que muitas tentativas de otimização visam trabalhar.

2.1.7 Comportamento Esperado

Para definir qual o comportamento adequado da simulação podemos usar as características definidas por Erneholm (2011): tomar por base um bando como sendo um grupo de corpos que se movem de modo que estão próximos entre si e com aproximadamente a mesma velocidade e direção. Importante notar que não existe uma definição formal de um comportamento de bando correto, porém ele propõe alguns pontos que podemos observar em uma boa simulação desse tipo de movimento:

a) os corpos formem um bando ao se encontrarem;

b) membros de um bando não colidam entre si;

c) um bando não deve perder membros espontaneamente;

d) um bando não deve se dividir em outros bandos espontaneamente.

2.2 UNITY DOTS

A Unity Engine define o DOTS como "Uma combinação de tecnologias e pacotes que entrega uma estratégia de design orientada a dados para desenvolver jogos na Unity"[1]. Ele é dividido em três sistemas principais:

a) C# Job System;

b) Burst Compiler;

c) Entity Component System (ECS).

[1] https://unity.com/dots

2.2.1 C# Job System

O C# Job System é uma ferramenta que disponibiliza uma nova sintaxe para escrever códigos de multiplos fluxos de execução com mais garantias de segurança diretamente no projeto em C#. O Job System auxilia na melhor utilização de recursos de paralelismo presente em dispositivos alvo, gerando uma melhora no desempenho do aplicativo, especialmente em aparelhos móveis.

Para criar um Job para realizar uma tarefa de maneira otimizada em vários núcleos é necessário construir uma `struct` que implementa o método Execute da interface `IJob`[2]. Uma instância da `struct` é criada e preenchida com os dados necessários para a execução e escalonada para execução na thread principal através do método `Schedule`.

Importante ressaltar que um Job opera apenas sobre variáveis de tipo *blittable* ou `NativeContainers` para evitar condições de corrida e assegurar que os dados serão transferidos da thread principal para a thread auxiliar através de cópia.

NativeContainers são estruturas de dados que encapsulam pedaços de memória nativa de forma que possam ser manipuladas em um ambiente gerenciado. A estrutura mais comum desse tipo é o `NativeArray`[3], que equivale a um array simples, mas outras estruturas mais complexas existem como por exemplo `NativeHashMap` que é utilizado em implementações de *spatial hashing*.

Abaixo temos um exemplo de dois Jobs (Códigos 6 e 7) que dado um array de valores decimais (input) eles calculam o quadrado de cada elemento do array e retorna esse valor armazenado no array de saída (output).

Código 6 – Exemplo de IJob

```
public struct Job : IJob
{
    [ReadOnly]
    public NativeArray<float> input;

    [WriteOnly]
    public NativeArray<float> output;

    public void Execute ()
    {
        for (int i = 0; i < input.Length; i++)
        {
            output[i] = input[i] * input[i];
        }
    }
}
```

[2] https://docs.unity3d.com/2022.2/Documentation/ScriptReference/Unity.Jobs.IJob.html
[3] https://docs.unity3d.com/2022.2/Documentation/ScriptReference/Unity.Collections.NativeArray_1.html

Código 7 – Exemplo de IJobParallelFor

```
public struct JobParellelFor : IJobParallelFor
{
    [ReadOnly]
    public NativeArray<float> input;

    [WriteOnly]
    public NativeArray<float> output;

    public void Execute (int i)
    {
        output[i] = input[i] * input[i];
    }
}
```

O método `Execute` das duas `structs`, apesar de terem a mesma finalidade, executam de forma ligeiramente distinta. Ambos executam fora da thread principal mas o `JobParallelFor`[4] é capaz de realizar o cálculo de vários elementos em paralelo em múltiplas threads, enquanto o *Job* executa sequencialmente em uma única thread auxiliar. A forma como esses *Jobs* são invocados é exemplificada pelo Código 8.

[4] https://docs.unity3d.com/2022.2/Documentation/ScriptReference/Unity.Jobs.IJobParallelFor.html

Código 8 – Scheduling dos Jobs

```
//Cria o array de entrada
NativeArray<float> input = new NativeArray<float>(1000, Allocator.
    TempJob);

//Creates output array for the sequential Job
NativeArray<float> output = new NativeArray<float>(1000, Allocator.
    TempJob);

//Creates output array for the parallel job
NativeArray<float> outputParallel = new NativeArray<float>(1000,
    Allocator.TempJob);

//Instantiates Job struct with necessary data
Job job = new Job
{
    input = input,
    output = output
};

//Instantiates JobParallelFor struct with necessary data
JobParellelFor jobParallelFor = new JobParellelFor
{
    input = input,
    output = outputParallel
};

//Schedule jobs for execution
JobHandle handle = job.Schedule();
//For the parallel job is necessary to specify its input struct size
//and the size of the execution batch
JobHandle handleParallel = jobParallelFor.Schedule(input.Length, 64);

//Awaits end of execution of jobs
handle.Complete();
handleParallel.Complete();

//Frees memory used in NativeArrays
input.Dispose();
output.Dispose();
outputParallel.Dispose();
```

2.2.2 Burst Compiler

O Burst é um compilador de alto desempenho integrado na Unity Engine, desenvolvido para otimizar a execução de código em processadores modernos e GPUs. Ele converte códigos em IL (Intermediate Language) ou .NET bytecode originalmente escritos em C# para código de máquina altamente otimizado para o sistema de destino, garantindo melhorias significativas de desempenho em relação ao código gerenciado tradicional. No Código 9 temos um exemplo de uso.

O Burst é uma ferramenta útil para melhorar o desempenho sem dificultar a implementação, pois funciona de forma automática e é nativamente compatível entre plataformas. De acordo com Borufka (2020), o uso do Burst em conjunto com o Job System é recomendado sempre que possível, já que o seu uso tem potencial para resultar em melhorias de desempenho.

Código 9 – Burst Compile

```
[BurstCompile]
struct MyJob : IJobParallelFor
{ ... }
```

O código demonstra o uso de um atributo `BurstCompile` que indica ao compilador que deve compilar o `MyJob` usando o Burst.

2.2.3 Entity Component System

O *Entity Component System (ECS)*[5] traz para o desenvolvimento na Unity Engine uma abordagem diferenciada de estruturar o código visando obter os benefícios de um código orientado a dados, isto é, otimizando o uso da cache da CPU ao utilizar estruturas de dados que exploram o princípio de localidade dos dados (LLOPIS, 2009). Apesar de existir em oposição ao modelo convencional de orientação a objetos, o ECS pode ser utilizado em conjunto com ele de forma híbrida.

Ele é composto por três partes essênciais:

a) *Entity*, ou entidade, é um identificador único que identifica um conjunto de componentes.

b) *Component*, ou componente, é composto por dados e não possui nenhum tipo de lógica interna.

c) *System*, ou sistema, age em coleções de componentes ou tuplas de componentes executando operações sobre eles.

Esse tipo de arquitetura mantém os dados e a lógica do sistema totalmente separados. Os componentes do mesmo tipo podem então ser armazenados sequencialmente na

[5] https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html

memória facilitando o uso da cache, diminuindo a taxa de falhas de leitura na memória e permitindo a utilização de instruções do tipo SIMD.

Código 10 – Implementação em MonoBehaviour

```
using UnityEngine;

public class MoveObject : MonoBehaviour
{
    public Vector3 direction;
    public float speed;

    void Update()
    {
        transform.position += direction * speed * Time.deltaTime;
    }
}
```

No código acima (Código 10) temos um exemplo simples usando MonoBehaviour, a classe base de todos os objetos de jogo na Unity Engine que comumente mistura tanto a lógica quanto os dados necessários. Uma implementação de movimento de um objeto com uma certa direção (direction) e velocidade (speed).

Código 11 – Implementação em ECS

```
using Unity.Entities;
using Unity.Mathematics;
using Unity.Transforms;

public struct MoveData : IComponentData
{
    public float3 direction;
    public float speed;
}

public class MoveSystem : SystemBase
{
    protected override void OnUpdate()
    {
        float deltaTime = Time.DeltaTime;

        Entities.ForEach((ref Translation translation, in MoveData
            moveData) =>
        {
            translation.Value += moveData.direction * moveData.speed *
                deltaTime;
        }).Schedule();
    }
}
```

No exemplo acima (Código 11), temos uma implementação usando ECS para a mesma lógica encontrada na classe MoveObject. Os dados e a lógica estão separados entre um *component* e um *system*. No componente MoveData ficam registrados a direção e velocidade de uma entidade e o sistema MoveSystem itera sobre todos as entidades que possuem componentes do tipo *Translation* e *MoveData*, aplicando a lógica de movimentação em cada um deles.

3 BREVE HISTÓRICO

Nesse capítulo serão analizados trabalhos anteriores que tratam sobre o algoritmo de *boids* e várias estratégias que já foram exploradas para melhorar o seu desempenho. Partimos de sua primeira idealização por Craig Raynolds e continuamos por diversas propostas de melhorias por diferentes autores. Em seguida, detalhamos de forma resumida as principais características e evolução do *Unity DOTS*.

3.1 ALGORITMO DE BOIDS

Em 1987, Craig Reynolds propôs o algoritmo de boids para simular o comportamento de grupos de animais baseando-se em agentes independentes (REYNOLDS, 1987). O algoritmo destaca-se por sua simplicidade e possui resultados bem próximos da realidade. Cada agente na simulação observa o estado atual do seu ambiente e usa esses dados para calcular sua próxima ação. Usando essa ideia, ele consegue gerar um comportamento emergente a partir de cálculos simples distribuídos entre todos os agentes, observando a distância entre um *boid* e seus vizinhos.

O principal problema presente no algoritmo se deve à complexidade de $O(n^2)$ necessária para percorrer todos os pares de agentes a cada iteração da simulação. Na época, de acordo com o próprio autor, isso tornava impraticável o uso desse algoritmo para simulações em tempo real como jogos digitais. Reynolds et al. (1999) e Reynolds (2000) incrementaram ainda o modelo para incluir regras mais complexas e o uso da estratégia de *hashing* espacial. Essa estratégia consiste em agrupar elementos próximos espacialmente com uma mesma chave *hash*, reduzindo o número de *boids* que precisam ser considerados em sua vizinhança.

Hastings, Mesit e Guha (2005) usaram o algoritmo de *boids* para demonstrar a capacidade de otimização da estratégia de *hashing* espacial no uso de simulações com um número elevado de agentes independentes com resposta em tempo real. Por meio dessa técnica conseguiram otimizar os cálculos necessários de colisão objeto a objeto, objeto com terreno, renderização, e decisões de cada *boid* em sua análise de vizinhos próximos. Entretanto, o uso de *hashing* espacial, apesar de trazer uma melhora significativa, permanece sensível a densidade de *boids* no espaço simulado, com a complexidade se aproximando novamente de $O(n^2)$ quanto maior o número de *boids* em um único *hash*.

Mavhemwa e Nyangani (2018) investigaram a possibilidade de usar subdivisão espacial uniforme como tentativa de melhorar a checagem de proximidade entre vizinhos no algoritmo de *boids*, entretanto a solução proposta não demonstrou melhora significativa comparando a implementação ingênua do algoritmo.

A estratégia explorada por Erneholm (2011) focou em não mudar a forma que a vizi-

nhança de um *boid* é computada, mas sim alterar sua própria definição. Em seu trabalho o autor compara duas definições: todos os *boids* dentro de um certo raio de distância e uma alternativa onde são olhados apenas os N *boids* mais próximos, independente de sua distância. Essa mudança na definição de vizinhança levou a uma divergência entre o resultado obtido e o comportamento esperado do sistema, devido a um conflito entre as regras de coesão e separação.

Zhou e Zhou (2004) propuseram um modelo de subdivisão espacial onde cada subdivisão existia em um dos computadores de um cluster processando de forma paralela. Nessa implementação foi possível dividir a carga computacional igualmente entre cada CPU com o uso de uma estratégia de balanceamento que particiona o espaço de simulação dinamicamente entre todos os processadores. Por meio do balanceamento, evitou-se tanto a sensibilidade em relação a densidade espacial dos *boids* quanto também a necessidade de comunicação entre os processadores a cada frame da simulação. Porém, a arquitetura em cluster torna a solução pouco escalável e de difícil acesso para sistemas onde comumente rodam jogos digitais.

Com o objetivo de mitigar a perda de desempenho em implementações de hashing espacial, Passos et al. (2010) propuseram uma implementação em GPU com uma nova estrutura de dados que permite que o cálculo de vizinhança de um *boid* seja feito em tempo constante. Essa otimização se dá por meio da modelagem do espaço de simulação como um autômato celular que é obtido de uma passada incompleta de um algoritmo de classificação. A estrutura de dados resultante é uma aproximação discreta da relação de posição entre os *boids*, permitindo que a vizinhança seja facilmente extraída utilizando os elementos circundantes na matriz. Os autores demonstram que a implementação funciona bem em arquiteturas de GPU e apresentam um ganho significativo em relação a implementações utilizando *hashing* espacial e a forma ingênua. Em trabalhos subsequentes os autores demonstraram que a estrutura de dados proposta pode ser estendida para três dimensões sem perdas significativas no desempenho (JOSELLI et al., 2009) e que a estrutura também é adequada para uso em dispositivos móveis mais atuais (JOSELLI et al., 2012).

3.2 UNITY DOTS

Por ser uma ferramenta anunciada em 2018 e ainda em desenvolvimento tendo sido desponibilizada para o público recentemente existem, até o momento, poucos trabalhos publicados a respeito do seu uso e aplicações. Turpeinen (2020) usou o problema de renderização de multidões em 3D para comparar o desempenho de uma implementação usando DOTS e outra usando a arquitetura orientada a objetos convencional da Unity Engine, além do uso ou não de instanciamento por GPU. Os testes foram executados em dois *smartphones* considerados atuais, um iPhone XR e um iPhone 6S onde em quase

todos os cenários testados mostrou-se uma melhora significativa do DOTS em relação ao modelo orientado a objetos.

Borufka (2020) desenvolveu um conjunto de testes de desempenho para implementações feitas na Unity Engine e utilizou os testes para comparar implementações convencionais, com e sem o uso do Job System, e implementações utilizando o modelo de arquitetura orientada a dados do DOTS (ECS) com e sem o auxílio do compilador Burst. A partir do estudo e análise dos resultados obtidos, o autor propõe uma série de recomendações para melhor aproveitar os benefícios disponibilizados pelo sistema estudado.

4 IMPLEMENTAÇÃO

O objetivo da parte prática deste trabalho foi criar uma implementação do algoritmo de Boids em duas dimensões, aplicando os conceitos aprendidos de design orientado a dados, e utilizando as ferramentas disponíveis pelo Unity DOTS para execução em dispositivos móveis e obtenção de métricas importantes para análise e comparação. Sendo assim, esse capítulo está organizado da seguinte maneira: primeiramente temos uma breve descrição da implementação da Unity que foi usada como base para as implementações desse trabalho. Em seguida apresenta-se uma descrição geral das três versões de implementação que foram desenvolvidas e que serão avaliadas e comparadas no próximo capítulo. Por fim, apresenta-se uma descrição das ferramentas e do ambiente de desenvolvimento utilizado durante a implementação.

4.1 IMPLEMENTAÇÃO PRÓPRIA DA UNITY

Em seus materiais de exemplo do sistema DOTS, a Unity disponibiliza uma versão do algoritmo de boids em três dimensões, visando demonstrar a capacidade do sistema de gerar o comportamento de grupos de peixes em movimento. A implementação está disponível em código aberto pela Unity no github[1].

Figura 1 – Captura de tela da implementação de Boids pela Unity

Fonte: Unity Technologies

O resultado apresentado mostrou um bom desempenho no editor com modelos 3D texturizados e animados e efeitos de pós processamento gráfico, conseguindo manter a

[1] https://github.com/Unity-Technologies/EntityComponentSystemSamples

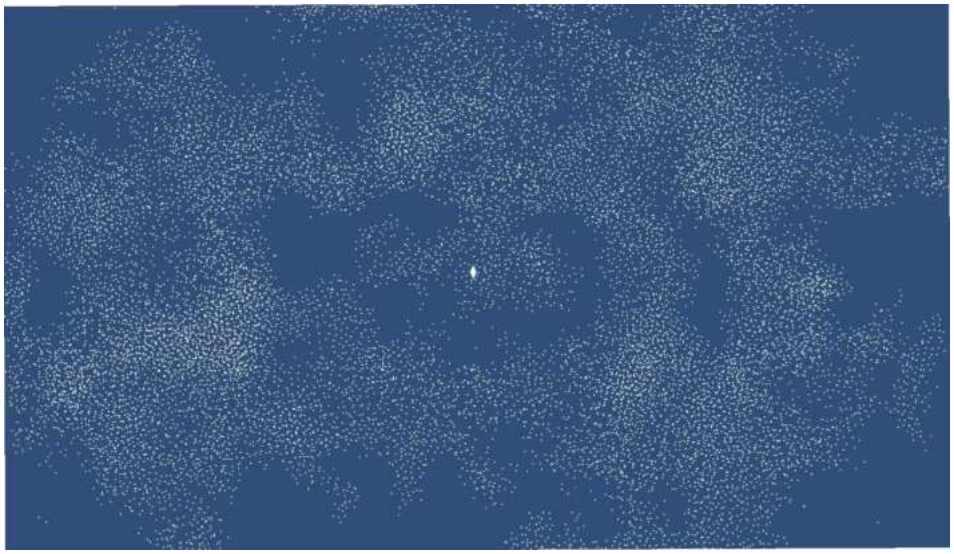
faixa dos 30 FPS com mais de 4096 peixes simulados. Entretanto, uma análise mais aprofundada do código fonte demonstrou que a implementação não seguia à risca as descrições das regras criadas por Reynolds (1987), como por exemplo, a falta de uma regra de Coesão. Por conta disso, a simulação não condizia com o comportamento esperado descrito por Erneholm (2011).

4.2 DESCRIÇÃO DA IMPLEMENTAÇÃO

Foram implementadas três versões de simulação de boids utilizando a Unity DOTS que se diferenciam na decisão de otimização das suas definições de vizinhança. São elas:

- *Força-bruta*, onde todos os boids olham para todos os outros boids para calcular sua vizinhança. Essa versão é utilizada como referencial de comparação para os ganhos obtidos pelas otimizações nas outras duas versões;
- *Spatial Hashing*, onde utiliza-se a estratégia de hashing espacial para descobrir boids presentes em uma região próxima do espaço de simulação. Essa é a estratégia mais utilizada, inclusive pela Unity na sua implementação, servindo então como exemplo das estratégias atuais;
- E *Sort*, onde utiliza-se da estrutura de dados proposta por Passos et al. (2010) para encontrar os boids mais próximos. Essa versão foi criada para avaliar a sua compatibilidade com a arquitetura ECS e descobrir se sua vantagem sobre o *hashing* espacial se mantem nesse ambiente de desenvolvimento.

Figura 2 – Captura de tela da implmentação em execução com 16384 boids



4.2.1 BoidComponent

O componente de tipo Boid define os parâmetros de comportamento de uma categoria de boid e se manteve com a mesma estrutura para as três versões (Código 12). Os seus campos são:

- *CohesionWeight*, define o quanto a regra de coesão influencia na direção do boid no próximo frame;
- *SeparationWeight*, define o quanto a regra de separação influencia na direção do boid no próximo frame;
- *AlignmentWeight*, define o quanto a regra de alinhamento influencia na direção do boid no próximo frame;
- *CohesionDistance*, define a distância mínima entre dois boids para que a regra de coesão seja considerada;
- *SeparationDistance*, define a distância mínima entre dois boids para que a regra de separação seja considerada;
- *AlignementDistance*, define a distância mínima entre dois boids para que a regra de alinhamento seja considerada;
- *MoveSpeed*, define a velocidade em unidades por segundo que o boid se move;
- *WallAvoidanceDistance*, define a distância que um boid começa a sentir uma influência contrária (de evitação) às bordas do espaço de simulação para evitar que este escape;
- *WallAvoidanceWeight*, define o quanto a regra de evitação das bordas do espaço de simulação influencia na direção do boid no próximo frame;
- *CellRadius*, define o tamanho da subdivisão do espaço onde as posições dos boids serão mapeados para uma mesma *hash* que será utilizada na versão de *Spatial Hashing*;
- *MooreDistance*[2], que define o tamanho da vizinhança extendida de Moore utilizada nas estratégias de *Spatial Hashing* e *Sort*;
- *WorldBounds*, define os limites do espaço de simulação;

4.2.2 BoidAuthoring

O `GameObject` do tipo `Boid` contém a representação visual de um Boid que será renderizado e um `BoidAuthoring` que transformará esse `GameObject` em uma Entidade com os Componentes necessários de um Boid.

[2] https://mathworld.wolfram.com/MooreNeighborhood.html

4.2.3 Systems

O comportamento central `SteerBoidJob` responsável por controlar a atualização da posição e direção de cada Boid possui a mesma estrutura de execução alterando-se somente a implementação do método `ApplyJob`. Podemos ver sua implementção para a versão de Força bruta no Código 14.

a) Força Bruta

Na implementação da estratégia de força bruta, temos uma inicialização bem simples como podemos ver no Código 15. Apenas guardamos as posições e direções atualizadas de cada boid numa lista antes de calcular o resultado do frame no `BoidSteerJob` (Código 14).

A aplicação das regras nessa estratégia acontece iterando por todos os elementos das listas de posições e direções e calculando as influências para cada um dos boids (Código 16).

b) Spatial Hashing

Na estratégia de *Spatial Hashing*, assim como na estratégia anterior, guardamos as posições e direções de cada boid no início de cada frame da simulação. Mas também é necessário inicializar a estrutura de hash `ParallelHashMap` que irá guardar o indice de todos os boids dentro de uma mesma seção do espaço de simulação (Código 17). Para encontrar o hash da posição de um boid, utilizamos o método auxiliar `HashPosition` cuja implementação está demonstrada no Código 18.

Na Fig 3 temos um exemplo de como o espaço de simulação é subdividido e as estruturas correspondentes de `HashMap` e do índice de cada objeto nesta hash map.

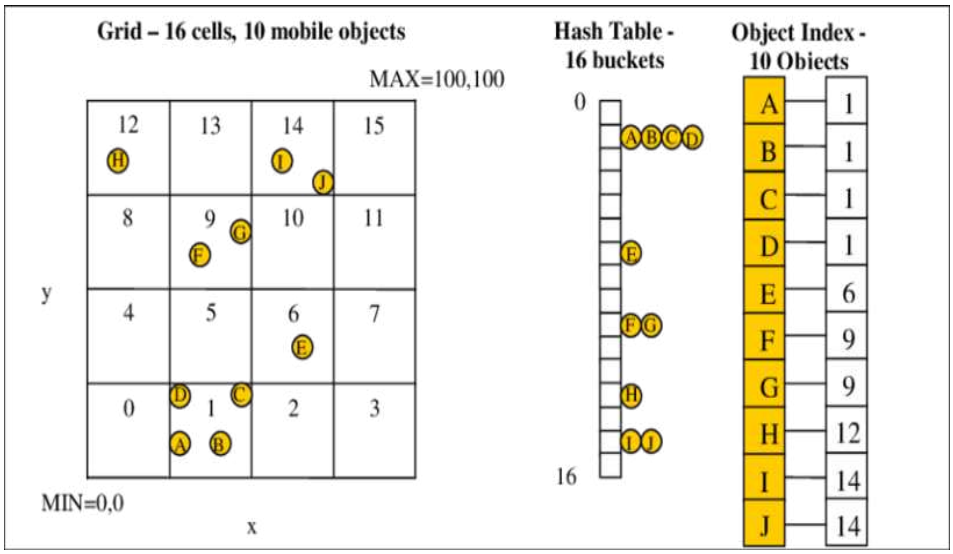
Para encontrar a vizinhança de um boid nessa estratégia iteramos por todos os outros boids que possuem o mesmo hash que ele assim como também aqueles que estão nas regiões adjacentes, utilizando-se do conceito de Vizinhança Extendida de Moore. (Código 19).

c) Sort

Para a estratégia de *Sort* a inicialização é idêntica a versão de Força Bruta, guardando apenas a posição e alinhamento dos boids em uma lista (Código 20). A diferença consiste em como essas posições serão ordenadas a cada frame para garantir que elementos próximos na matriz representam elementos próximos no espaço de simulação, como podemos observar no exemplo presente na Fig. 4.

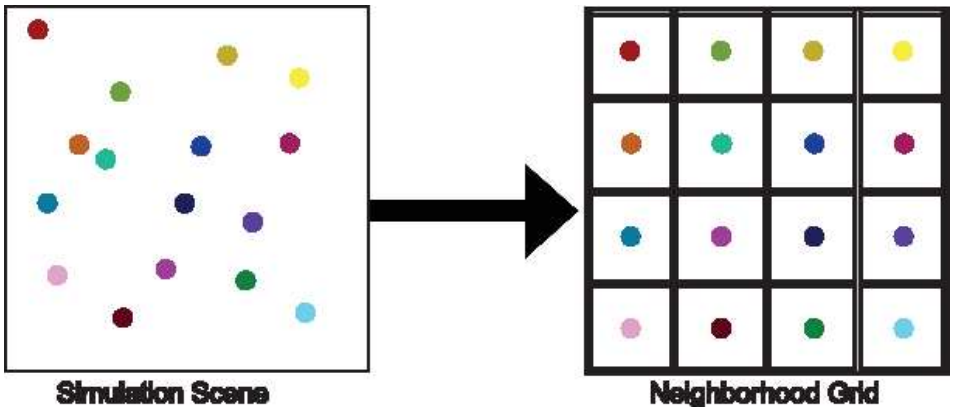
Encontrar a vizinhança de um boid e aplicar as regras de simulação nessa estratégia é simples pois sabemos que as matrizes de posição e alinhamento já estão ordenadas.

Figura 3 – Exemplo de Hashing Espacial



Fonte: Hastings, Mesit e Guha (2005)

Figura 4 – Exemplo de representação de uma simulação em um grid de vizinhança



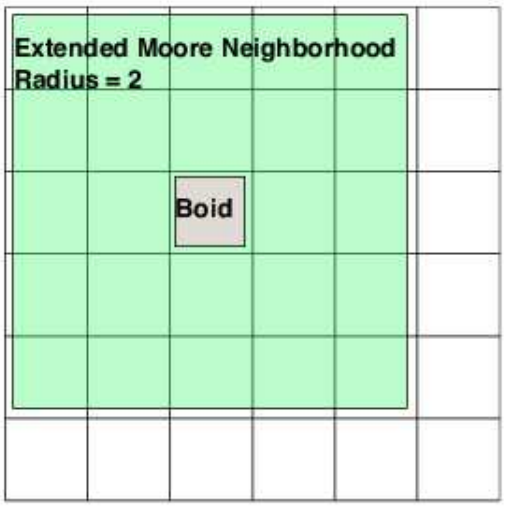
Fonte: Joselli et al. (2012)

Iteramos pelos vizinhos desse boid na matriz e calculamos suas influências como podemos ver no Código 21 e exemplificado na Fig. 5.

Para realizar a ordenação das matrizes de posições e alinhamentos foi escolhido o algoritmo de ordernação Par-Ímpar[3] por ser um algoritmo paralelo de ordenação eficiente e de fácil implementação. O algoritmo foi adaptado para funcionar numa matriz bidimensional operando alternadamente entre as linhas (Código 22) e colunas (Código 23). Para alcançar uma ordenação completa, deve-se repetir essas iterações até não acontecerem mais alterações na matriz. Entretanto foi constatado por Passos et al. (2010) que somente realizar uma ordenação completa no início da simulação e uma ordenação parcial para cada frame subsequente resulta

[3] https://pt.wikipedia.org/wiki/Odd-even_sort

Figura 5 – Vizinhança extendida de Moore que define os vizinhos de um boid na matrix



Fonte: Passos et al. (2010)

em erros pontuais que não atrabalham a qualidade da simulação. Podemos ver as implementações da ordenação parcial no Código 24 e completa no Código 25.

Podemos observar também que após a ordenação perdemos a ligação entre um boid e sua posição na matriz por não terem mais o mesmo índice. Isso não é um problema, dado que todos os boids são idênticos, a consequência disso faz com que dois boids sejam trocados de posição entre um frame e outro, o que é algo imperceptível.

4.3 FERRAMENTAS E AMBIENTES DE DESENVOLVIMENTO

Foi utilizada a versão *2022.2.15f* do editor da Unity, partindo como base do projeto o template 3D Core (URP). Esse template faz uso da Pipeline de Renderização Universal necessária para utilizar o pacote *Entities* que disponibiliza as ferramentas de desenvolvimento do ECS.

As versões dos pacotes utilizados foram: *Entities* 1.0.0-pre.65 e *Burst* 1.8.4 ambos de 22 de Março de 2023. Não foi necessário adicionar um pacote para o uso do C# Job System visto que ele já configura como parte do editor.

5 AVALIAÇÃO E RESULTADOS

Neste trabalho serão avaliadas três versões de implementação da simulação. Primeiro, será analisado o desempenho do algoritmo implementado usando apenas as ferramentas disponibilizadas pelo sistema DOTS (*Força Bruta*). Em seguida, serão consideradas duas alternativas de otimização para o problema do cálculo de vizinhança: uma versão utilizando hashing espacial (*Spatial Hashing*) e outra utilizando a estrutura de dados proposta por Passos et al. (2010) (*Sort*).

5.1 DETALHES TÉCNICOS

Todas as versões de implementação foram construídas utilizando a Unity Engine 2022.2.15f Personal contendo os pacotes de Entities versão 1.0.0-pre.65 e Burst versão 1.8.4 ambos de 22 de Março de 2023. As versões para os testes foram compiladas para Android utilizando a Unity para os seguintes dispositivos:

- **Redmi Note 11** com o sistema operacional **Android 11**, **8GB** de RAM, processador **Snapdragon 680 Octa-core** e placa gráfica **Qualcomm Adreno™ 610 GPU**;
- **Galaxy S10e** com o sistema operacional **Android 12**, **6GB** de RAM, processador **SAMSUNG Exynos 9 Octa 9820** e placa gráfica **Mali-G76 MP12**.

A escolha dos dispositivos foi limitada apenas aos que estavam disponíveis para teste durante a execução dos testes.

5.2 METODOLOGIA

Cada versão da simulação, com ou sem otimização, foi executada no dispositivo por cinco minutos com 256, 1024, 4096 e 16384 boids. Esses valores foram escolhidos por serem múltiplos de 4 e proporcionarem diferença significativa entre o desempenho das versões analisadas.

Para cada execução foram então coletadas a quantidade média de Frames por segundo (FPS) e a duração média em milissegundos do processamento em CPU e em GPU. Essas métricas foram escolhidas por serem comumente utilizadas na literatura (TURPEINEN, 2020; HASTINGS; MESIT; GUHA, 2005; PASSOS et al., 2010) e facilmente coletadas utilizando uma ferramenta disponibilizada pela Unity dentro do próprio aplicativo.

5.3 RESULTADOS

A seguir iremos avaliar separadamente os resultados dos testes em ambos os disppositivos para as métricas de FPS, tempo de CPU e tempo de GPU. Em jogos digitais, o FPS é geralmente limitado à 60 *frames* por segundo.

5.3.1 FPS

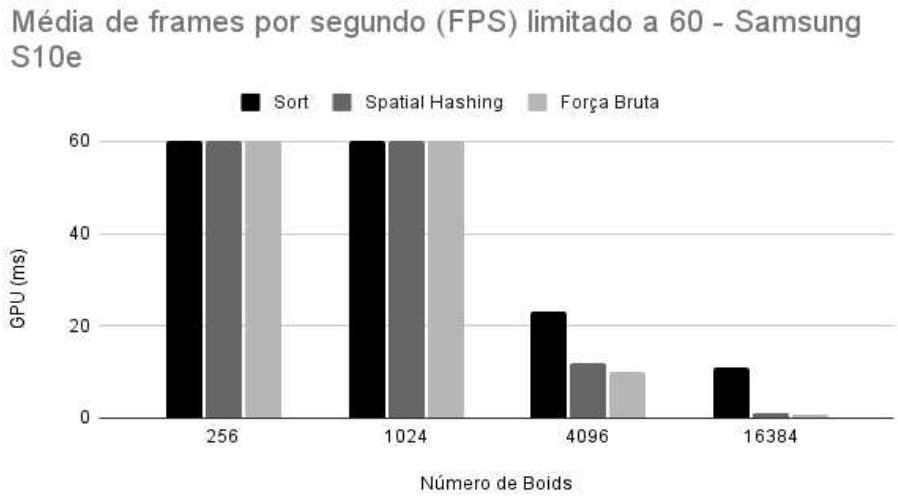
Analisando a métrica FPS para cada uma das estratégias de definição de vizinhança dos boids vemos que, para ambos os dispositivos, a estratégia de *Sort* desempenha melhor dentre as três, seguida pela estratégia de *Spatial Hashing*. A estratégia de *Sort* é a única que consegue alcançar uma taxa de frames por segundo que pode ser considerada interativa ao subir o número de boids para 4096.

Apesar de resultados similares nos dois dispositivos, o dispositivo Samsung apresenta uma estabilidade maior nas simulações com 256 e 1024 boids e uma queda mais abrupta ao aumentar o tamanho do espaço de teste para 4096 (Figura 6 e Tabela 1).

O dispositivo Redmi, por outro lado, já apresenta uma queda em sua taxa de frames por segundo a partir de 1024 boids, porém apresenta desempenho levemente superior para o caso de 4096 boids (Figura 7 e Tabela 2).

Podemos observar que nenhum dos testes obteve uma taxa de frames por segundo acima de 60. Isso acontece porque na indústria de jogos a taxa de frames por segundo é comumente limitada a esse valor e por causa disso a Unity já realiza esse corte de forma automática.

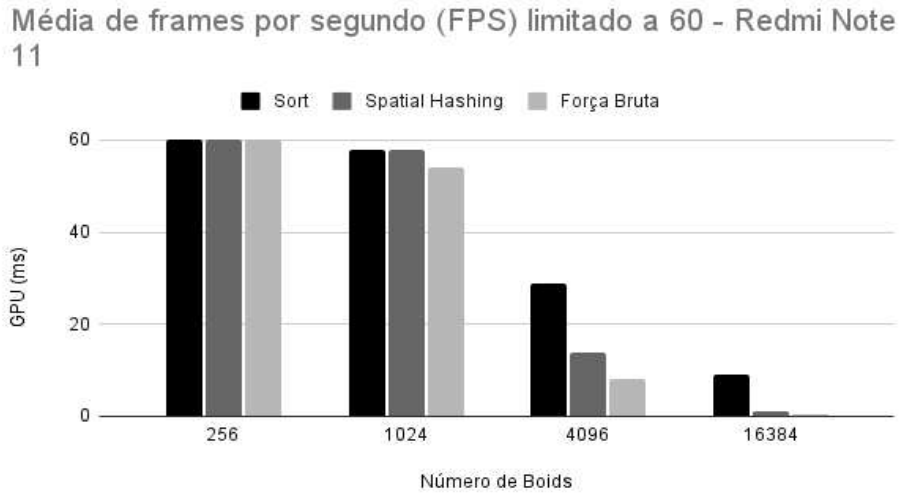
Figura 6 – Samsung S10e - FPS



N.º de boids	Sort	Spatial Hashing	Força Bruta
256	60	60	60
1024	60	60	60
4096	23	12	10
16384	11	1	0,7

Tabela 1 – Média de FPS limitado a 60 - Samsung S10e

Figura 7 – Redmi Note 11 - FPS



N.º de boids	Sort	Spatial Hashing	Força Bruta
256	60	60	60
1024	58	58	54
4096	29	14	8
16384	9	1	0,6

Tabela 2 – Média de FPS limitado a 60 - Redmi Note 11

5.3.2 Tempo de frame - CPU

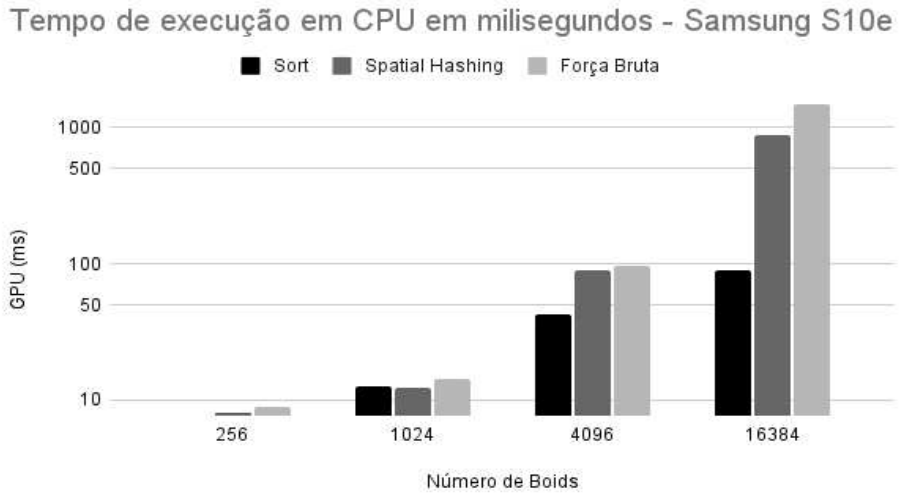
Os resultados obtidos para a métrica de tempo de frame na CPU acompanham a tendência encontrada nos valores obtidos de FPS. A implementação *Sort* aparece consistentemente com os menores valores dentre as três estratégias de otimização.

No caso específico do dispositivo Samsung (Figura 8 e Tabela 3) os valores da estratégia de *Spatial Hashing* se aproxima bastante dos valores para a implementação de *Força Bruta*. Já no dispositivo Redmi (Figura 9 e Tabela 4) a estratégia de *Spatial Hashing* alcança um desempenho melhor, apesar de ainda ser pior que a estratégia de *Sort*.

5.3.3 Tempo de frame - GPU

O tempo gasto na GPU a cada frame teve um comportamento distinto para cada dispositivo. No dispositivo Samsung (Figura 10 e Tabela 5) para todos os casos de teste e

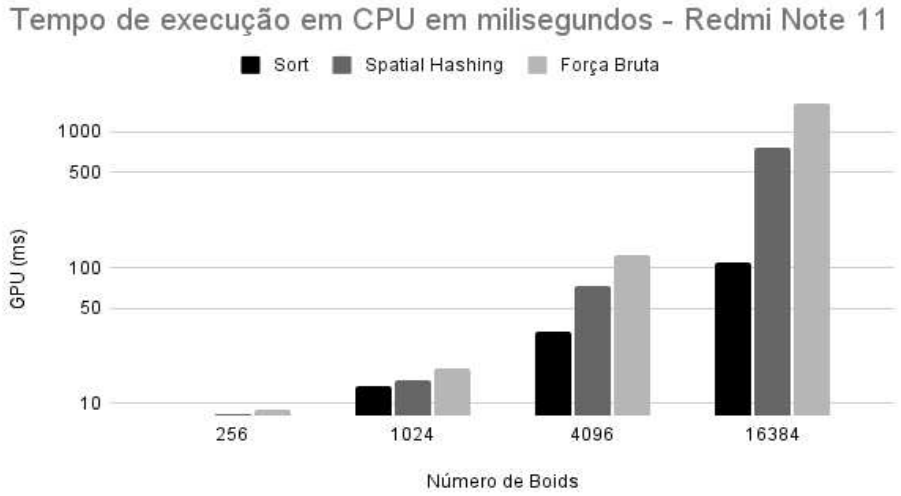
Figura 8 – Samsung S10e - CPU



N.º de boids	Sort (ms)	Spatial Hashing (ms)	Força Bruta (ms)
256	7,6	8	8,9
1024	12,5	12,4	14,2
4096	43,1	90,2	96,4
16384	88,8	889,7	1471,6

Tabela 3 – Tempo de execução em CPU (ms) - Samsung S10e

Figura 9 – Redmi Note 11 - CPU



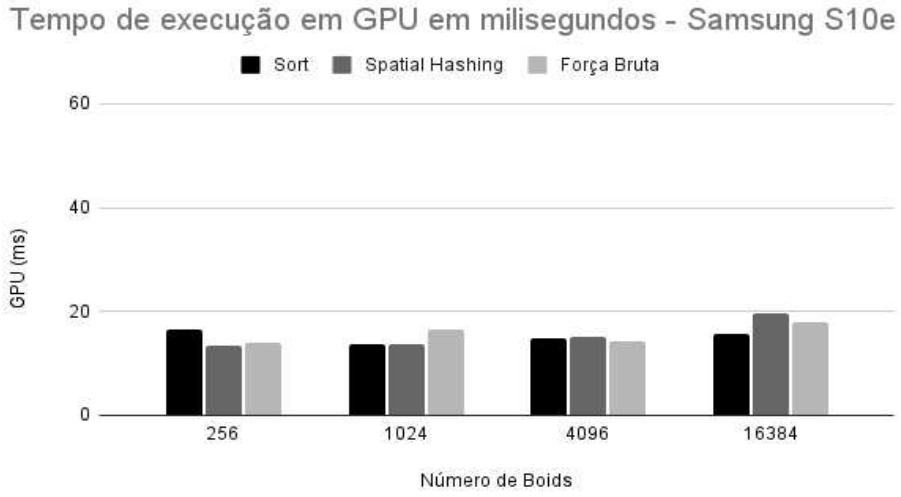
no dispositivo Redmi (Figura 11 e Tabela 6) para os casos com 256 e 1024 boids o tempo se manteve próximo a 15ms. Porém, no dispositivo Redmi, os valores começaram a subir com 4096 boids. E no caso de 16384 boids as estratégias se inverteram, com *Sort* tendo o pior desempenho e *Força Bruta* o melhor.

Os resultados da avaliação demonstram que a estratégia proposta por Passos et al.

N.º de boids	Sort (ms)	Spatial Hashing (ms)	Força Bruta (ms)
256	8,1	8,3	9
1024	13,6	14,9	18,3
4096	33,9	73,2	112,4
16384	109,5	756,4	1599,7

Tabela 4 – Tempo de execução em CPU (ms) - Redmi Note 11

Figura 10 – Samsung S10e - GPU



N.º de boids	Sort	Spatial Hashing	Força Bruta
256	16,5	13,5	14
1024	13,6	13,6	16,5
4096	14,9	15,1	14,4
16384	15,8	19,7	17,9

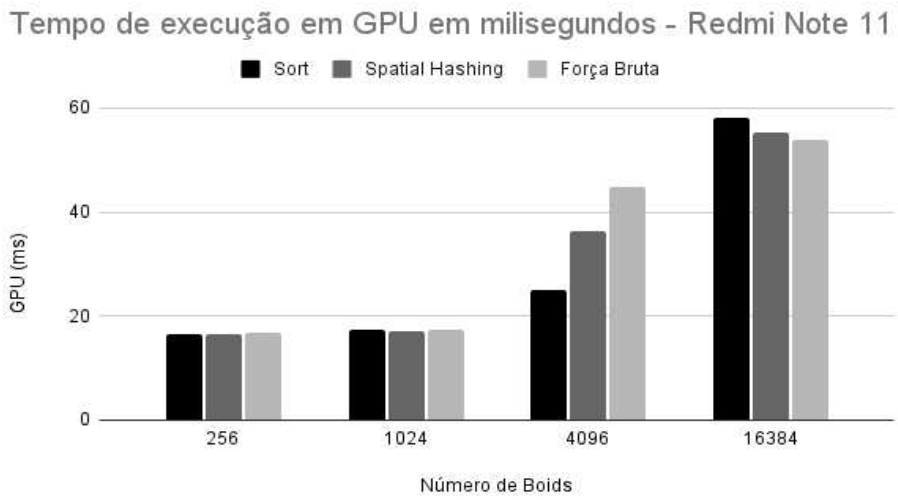
Tabela 5 – Tempo de execução em GPU (ms) - Samsung S10e

N.º de boids	Sort	Spatial Hashing	Força Bruta
256	16,6	16,5	16,
1024	17,3	17	17,5
4096	25,1	36,3	44, 9
16384	58,3	55,2	53,9

Tabela 6 – Tempo de execução em GPU (ms) - Redmi Note 11

(2010) (*Sort*) se mantém com vantagens sobre a estratégia de (*Spatial Hashing*) quando utilizadas em conjunto com o ECS do *Unity Dots*. Porém os resultados obtidos por trabalhos anteriores (PASSOS et al., 2010; JOSELLI et al., 2009; JOSELLI et al., 2012) foram superiores aos obtidos nesse comparativo, o que abre a oportunidade para trabalhos futuros que investiguem a razão para isso.

Figura 11 – Redmi Note 11 - GPU



6 CONCLUSÃO

Nos últimos anos os jogos desenvolvidos para dispositivos móveis têm se tornado mais ambiciosos tendo em vista o avanço acelerado nas tecnologias presentes nesses dispositivos. Apesar disso, esses sistemas ainda apresentam várias limitações de consumo energético e processamento.

Em vista disso, esse trabalho teve como objetivo fazer um comparativo em dispositivos móveis de três implementações do algoritmo de *boids* com diferentes estratégias de otimização desenvolvidas com o auxílio das ferramentas do *Unity DOTS*.

Levando em consideração os resultados obtidos na avaliação e comparando as três estratégias implementadas podemos concluir que a estratégia utilizando a estrutura de dados bidimensional proposta por Passos et al. (2010) possui um melhor desempenho quando implementada dentro da arquitetura ECS do Unity DOTS.

A arquitetura ECS apresenta grande potencial na otimização de simulações de grande número de agentes e suas aplicações em jogos digitais. Embora o ambiente do ECS no Unity DOTS ainda se encontre em desenvolvimento e apresente alguma instabilidade, existem outras ferramentas de desenvolvimento baseadas em ECS que podem ser exploradas em trabalhos futuros.

Há também a oportunidade de tabalhos futuros avaliarem o desempenho do ECS e dos algoritmos de otimização discutidos aqui em outros dispositivos móveis com especificações mais diversas e em outras aplicações.

REFERÊNCIAS

BORUFKA, R. **Performance testing suite for Unity DOTS**. Dissertação (Mestrado) — Department of Software and Computer Science Education, Charles University, Prague, 2020.

ERNEHOLM, C.-O. **Simulation of the Flocking Behavior of Birds with the Boids Algorithm.** 2011. https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand11/Group5Lars/carl-oscar_erneholm.pdf. Acessado: 2024-08-21.

HASTINGS, E. J.; MESIT, J.; GUHA, R. K. Optimization of large-scale, real-time simulations by spatial hashing. In: CITESEER. **Proc. 2005 Summer Computer Simulation Conference**. Cherry Hill, New Jersey, USA: Society for Modeling Simulation International (SCS), 2005. v. 37, n. 4, p. 9–17.

JOSELLI, M. et al. A flocking boids simulation and optimization structure for mobile multicore architectures. In: SBC. **XI Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2012 – Computing Track**. Brasilia, DF, Brasil: sbgames.org, 2012.

JOSELLI, M. et al. A neighborhood grid data structure for massive 3d crowd simulation on gpu. In: SBC. **2009 VIII Brazilian Symposium on Games and Digital Entertainment**. Rio de Janeiro, RJ, Brazil: IEEE, 2009. p. 121–131.

LLOPIS, N. **Data-oriented design**. 2009. Disponível em: https://gamesfromwithin.com/data-oriented-design.

MAVHEMWA, P. M.; NYANGANI, I. Uniform spatial subdivision to improve boids algorithm in a gaming environment. **International Journal for Advance Research and Development**, IJARnD, v. 3, n. 10, p. 49–57, 2018.

PASSOS, E. B. et al. A bidimensional data structure and spatial optimization for supermassive crowd simulation on gpu. **Computers in Entertainment (CIE)**, ACM New York, NY, USA, v. 7, n. 4, p. 1–15, 2010.

REYNOLDS, C. W. Flocks, herds and schools: A distributed behavioral model. In: **Proceedings of the 14th annual conference on computer graphics and interactive techniques**. Anaheim: ACM, 1987. p. 25–34.

REYNOLDS, C. W. Interaction with groups of autonomous characters. In: **Game Developers Conference**. https://gdconf.com/: www.cs.princeton.edu/, 2000. v. 2000, p. 449–460.

REYNOLDS, C. W. et al. Steering behaviors for autonomous characters. In: CITESEER. **Game developers conference**. https://gdconf.com/, 1999. v. 1999, p. 763–782.

TURPEINEN, M. **A Performance Comparison for 3D Crowd Rendering using an Object-Oriented system and Unity DOTS with GPU Instancing on Mobile Devices.** 2020. "https:https://www.diva-portal.org/smash/get/diva2:1467022/FULLTEXT01.pdf", note = "Acessado em: 2024-08-21".

ZHOU, B.; ZHOU, S. Parallel simulation of group behaviors. In: **Proceedings of the 2004 Winter Simulation Conference**. Washington, DC,USA: IEEE, 2004. v. 1, p. 370.

APÊNDICE A – EXCERTOS DE CÓDIGO DA IMPLEMENTACÃO

Código 12 – Boid Component

```
[Serializable]
[WriteGroup(typeof(LocalToWorld))]
public struct Boid : ISharedComponentData
{
    public float CohesionWeight;
    public float SeparationWeight;
    public float AlignmentWeight;
    public float CohesionDistance;
    public float SeparationDistance;
    public float AlignmentDistance;
    public float MoveSpeed;
    public float WallAvoidanceDistance;
    public float WallAvoidanceWeight;
    public float CellRadius;
    public int MooreDistance;
    public float4 WorldBounds;
}
```

Código 13 – Boid Authoring

```
using Unity.Entities;
using UnityEngine;

public class BoidAuthoring : MonoBehaviour
{
    public float CohesionWeight = 1.0f;
    public float SeparationWeight = 1.0f;
    public float AlignmentWeight = 1.0f;
    public float CohesionDistance = 1.0f;
    public float SeparationDistance = 1.0f;
    public float AlignmentDistance = 1.0f;
    public float WallAvoidanceDistance = 1.0f;
    public float WallAvoidanceWeight = 10f;
    public float MoveSpeed = 25.0f;
    public Vector4 WorldBounds = new(-18f, -12f, 18f, 12f);
    public float CellRadius = 10f;
    public int MooreDistance = 2;
}

public class BoidAuthoringBaker : Baker<BoidAuthoring>
{
    public override void Bake (BoidAuthoring authoring)
    {
        Entity entity = GetEntity(authoring, TransformUsageFlags.Dynamic
            | TransformUsageFlags.Renderable);
        AddSharedComponent(entity, new Boid
        {
            CohesionWeight = authoring.CohesionWeight,
            SeparationWeight = authoring.SeparationWeight,
            AlignmentWeight = authoring.AlignmentWeight,
            CohesionDistance = authoring.CohesionDistance,
            SeparationDistance = authoring.SeparationDistance,
            AlignmentDistance = authoring.AlignmentDistance,
            MoveSpeed = authoring.MoveSpeed,
            WallAvoidanceDistance = authoring.WallAvoidanceDistance,
            WallAvoidanceWeight = authoring.WallAvoidanceWeight,
            WorldBounds = authoring.WorldBounds,
            CellRadius = authoring.CellRadius,
            MooreDistance = authoring.MooreDistance
        });
    }
}
```

Código 14 – SteerBoidJob - Brute Force

```
[BurstCompile]
partial struct SteerBoidJob : IJobEntity {
    [ReadOnly] public NativeArray<float2> BoidAlignment;
    [ReadOnly] public NativeArray<float2> BoidPosition;
    [ReadOnly] public Boid CurrentBoidVariant;
    [ReadOnly] public float DeltaTime;
    [ReadOnly] public float MoveDistance;
    [ReadOnly] public float4 WorldBounds;
    [ReadOnly] public int Count;

    void Execute (ref LocalToWorld localToWorld) {
        float2 alignment = localToWorld.Up.xy;
        float2 position = localToWorld.Position.xy;
        float3 wallAvoidanceResult = GetWallAvoidance(position);
        (float3 cohesionResult, float3 separationResult, float3
           alignmentResult) =
            ApplyRules(position, alignment);
        float3 normalHeading = math.normalizesafe(cohesionResult +
           separationResult + alignmentResult + wallAvoidanceResult);
        float2 nextHeading = math.normalizesafe(alignment + DeltaTime *
           (normalHeading.xy - alignment));
        float2 nextPosition = position + nextHeading * MoveDistance;
        localToWorld = new LocalToWorld {
            Value = float4x4.TRS(
                new float3(nextPosition, 0f),
                quaternion.LookRotationSafe(math.forward(), new float3(
                   nextHeading, 0f)),
                new float3(0.1f, 0.1f, 0.1f))
        };
    }

    float3 GetWallAvoidance (float2 position) {...}
    (float3, float3, float3) ApplyRules (float2 position, float2
       alignment) {...}
    static int IncrementIf (bool condition) => math.select(0, 1,
       condition);
    static float3 AddIf (bool condition, float2 value) => new(math.
       select(0f, value, condition), 0f);
}
```

Código 15 – InitialPerBoidJob - Brute Force

```
[BurstCompile]
        partial struct InitialPerBoidJob : IJobEntity
        {
            [ReadOnly] public NativeArray<int> ChunkBaseEntityIndices;
            [NativeDisableParallelForRestriction] public NativeArray<
               float2> BoidAlignment;
            [NativeDisableParallelForRestriction] public NativeArray<
               float2> BoidPosition;
            void Execute ([ChunkIndexInQuery] int chunkIndexInQuery, [
               EntityIndexInChunk] int entityIndexInChunk, in
               LocalToWorld localToWorld)
            {
                int entityInQueryIndex = ChunkBaseEntityIndices[
                   chunkIndexInQuery] + entityIndexInChunk;
                BoidAlignment[entityInQueryIndex] = localToWorld.Up.xy;
                BoidPosition[entityInQueryIndex] = localToWorld.Position
                   .xy;
            }
        }
```

Código 16 – ApplyRules - Brute Force

```
(float3, float3, float3) ApplyRules (float2 position, float2 alignment)
{
    float3 cohesionResult = float3.zero, separationResult = float3.zero,
        alignmentResult = float3.zero;
    int cohesionCount = 0, separationCount = 0, alignmentCount = 0;

    for (int i = 0; i < Count; i++)
    {
        float2 otherPosition = BoidPosition[i];
        float2 otherAlignment = BoidAlignment[i];
        float dist = math.distance(position, otherPosition);
        cohesionResult += AddIf(dist < cohesionDistance && dist >
            separationDistance, otherPosition - position);
        cohesionCount += IncrementIf(dist < cohesionDistance && dist >
            separationDistance);
        separationResult += AddIf(dist < separationDistance, position -
            otherPosition);
        separationCount += IncrementIf(dist < separationDistance);
        alignmentResult += AddIf(dist < alignmentDistance,
            otherAlignment - alignment);
        alignmentCount += IncrementIf(dist < alignmentDistance);
    }

    cohesionResult = cohesionResult / cohesionCount * cohesionWeight;
    separationResult = separationResult / separationCount *
       separationWeight;
    alignmentResult = alignmentResult / alignmentCount * alignmentWeight
       ;

    return (cohesionResult, separationResult, alignmentResult);
}
```

Código 17 – InitialPerBoidJob - Spatial Hashing

```
[BurstCompile]
        partial struct InitialPerBoidJob : IJobEntity
        {
            [ReadOnly] public NativeArray<int> ChunkBaseEntityIndices;
            [NativeDisableParallelForRestriction] public NativeArray<
               float2> BoidAlignment;
            [NativeDisableParallelForRestriction] public NativeArray<
               float2> BoidPosition;
            public NativeParallelMultiHashMap<int, int>.ParallelWriter
               ParallelHashMap;
            public float InverseBoidCellRadius;
            void Execute ([ChunkIndexInQuery] int chunkIndexInQuery, [
               EntityIndexInChunk] int entityIndexInChunk, in
               LocalToWorld localToWorld)
            {
                int entityInQueryIndex = ChunkBaseEntityIndices[
                   chunkIndexInQuery] + entityIndexInChunk;
                BoidAlignment[entityInQueryIndex] = localToWorld.Up.xy;
                BoidPosition[entityInQueryIndex] = localToWorld.Position
                   .xy;
                int hash = HashPosition(localToWorld.Position.xy,
                   InverseBoidCellRadius);
                ParallelHashMap.Add(hash, entityInQueryIndex);
            }
        }
```

Código 18 – HashPosition - Spatial Hashing

```
static int HashPosition (float2 pos, float InverseBoidCellRadius)
{
    return (int)math.hash(new int2(math.floor(pos *
       InverseBoidCellRadius)));
}
```

Código 19 – ApplyRules - Spatial Hashing

```
(float3, float3, float3) ApplyRules (float2 position, float2 alignment)
{
    float3 cohesionResult = float3.zero, separationResult = float3.zero,
        alignmentResult = float3.zero;
    int cohesionCount = 0, separationCount = 0, alignmentCount = 0;
    for (int i = -MooreDistance; i <= MooreDistance; i++)
        for (int j = -MooreDistance; j <= MooreDistance; j++)
        {
            float2 pos = position + new float2(i, j) * BoidCellRadius;
            int hash = HashPosition(pos, InverseBoidCellRadius);

            NativeParallelMultiHashMap<int, int>.Enumerator neighbours =
                ParallelHashMap.GetValuesForKey(hash);
            foreach (int neighbour in neighbours) {
                float2 otherPosition = BoidPosition[neighbour];
                float2 otherAlignment = BoidAlignment[neighbour];
                float dist = math.distance(position, otherPosition);
                cohesionResult += AddIf(dist < cohesionDistance && dist
                    > separationDistance, otherPosition - position);
                cohesionCount += IncrementIf(dist < cohesionDistance &&
                    dist > separationDistance);
                separationResult += AddIf(dist < separationDistance,
                    position - otherPosition);
                separationCount += IncrementIf(dist < separationDistance
                    );
                alignmentResult += AddIf(dist < alignmentDistance,
                    otherAlignment - alignment);
                alignmentCount += IncrementIf(dist < alignmentDistance);
            }
        }
    cohesionResult = cohesionResult / cohesionCount * cohesionWeight;
    separationResult = separationResult / separationCount *
        separationWeight;
    alignmentResult = alignmentResult / alignmentCount * alignmentWeight
        ;
    return (cohesionResult, separationResult, alignmentResult);
}
```

Código 20 – InitialPerBoidJob - Sort

```
[BurstCompile]
        partial struct InitialPerBoidJob : IJobEntity
        {
            [ReadOnly] public NativeArray<int> ChunkBaseEntityIndices;
            [NativeDisableParallelForRestriction] public NativeArray<
               float2> BoidAlignment;
            [NativeDisableParallelForRestriction] public NativeArray<
               float2> BoidPosition;
            void Execute([ChunkIndexInQuery] int chunkIndexInQuery, [
               EntityIndexInChunk] int entityIndexInChunk, in
               LocalToWorld localToWorld)
            {
                int entityInQueryIndex = ChunkBaseEntityIndices[
                   chunkIndexInQuery] + entityIndexInChunk;
                BoidAlignment[entityInQueryIndex] = localToWorld.Up.xy;
                BoidPosition[entityInQueryIndex] = localToWorld.Position
                   .xy;
            }
        }
```

Código 21 – ApplyRules - Sort

```
(float3, float3, float3) ApplyRules (int boidIndex, float2 position,
    float2 alignment)
{
    (int x, int y) = GetCoords(boidIndex);
    float3 cohesionResult = float3.zero, separationResult = float3.zero,
        alignmentResult = float3.zero;
    int cohesionCount = 0, separationCount = 0, alignmentCount = 0;

    for (int i = -MooreDistance; i <= MooreDistance; i++)
    {
        int z = x + i;
        if (z < 0 || z >= MatrixSize)
            continue;
        for (int j = -MooreDistance; j <= MooreDistance; j++)
        {
            if (i == 0 && j == 0) continue;
            if (y + j < 0 || y + j >= MatrixSize) continue;
            int neighbourIndex = GetIndex(z, y + j);
            float2 otherPosition = BoidPosition[neighbourIndex];
            float2 otherAlignment = BoidAlignment[neighbourIndex];
            float dist = math.distance(position, otherPosition);
            cohesionResult += AddIf(dist < cohesionDistance,
                otherPosition - position);
            cohesionCount += IncrementIf(dist < cohesionDistance);
            separationResult += AddIf(dist < separationDistance,
                position - otherPosition);
            separationCount += IncrementIf(dist < separationDistance);
            alignmentResult += AddIf(dist < alignmentDistance,
                otherAlignment - alignment);
            alignmentCount += IncrementIf(dist < alignmentDistance);
        }
    }

    cohesionResult = cohesionResult / cohesionCount * cohesionWeight;
    separationResult = separationResult / separationCount *
        separationWeight;
    alignmentResult = alignmentResult / alignmentCount * alignmentWeight
        ;
    return (cohesionResult, separationResult, alignmentResult);
}
```

Código 22 – RowSort - Sort

```
[BurstCompile]
struct RowSort : IJobParallelFor {
    public int Size;
    public int Parity;
    [NativeDisableParallelForRestriction] public NativeArray<float2>
       Positions;
    [NativeDisableParallelForRestriction] public NativeArray<float2>
       Alignments;
    [NativeDisableParallelForRestriction] public NativeArray<bool>
       Sorted;
    public void Execute (int index) {
        (int x, int y) = GetCoords(index);
        if (y + 1 >= Size)
            return;
        int a = GetIndex(x, y);
        int b = GetIndex(x, y + 1);
        if (Positions[a].y < Positions[b].y ||
            Positions[a].y == Positions[b].y && Positions[a].x >
               Positions[b].x)
            Swap(a, b);
    }

    int GetIndex (int x, int y) => y * Size + x;
    (int, int) GetCoords (int index) => (index % Size, 2 * (index / Size
       ) + Parity);
    void Swap (int a, int b) {
        (Positions[a], Positions[b]) = (Positions[b], Positions[a]);
        (Alignments[a], Alignments[b]) = (Alignments[b], Alignments[a]);
        Sorted[0] = false;
    }
}
```

Código 23 – ColumnSort - Sort

```
[BurstCompile]
struct ColumnSort : IJobParallelFor {
    public int Size;
    public int Parity;
    [NativeDisableParallelForRestriction] public NativeArray<float2>
       Positions;
    [NativeDisableParallelForRestriction] public NativeArray<float2>
       Alignments;
    [NativeDisableParallelForRestriction] public NativeArray<bool>
       Sorted;
    public void Execute (int index) {
        (int x, int y) = GetCoords(index);
        if (x + 1 >= Size)
            return;
        int a = GetIndex(x, y);
        int b = GetIndex(x + 1, y);
        if (Positions[a].x > Positions[b].x ||
            Positions[a].x == Positions[b].x && Positions[a].y <
               Positions[b].y)
            Swap(a, b);
    }

    int GetIndex (int x, int y) => y * Size + x;
    (int, int) GetCoords (int index) => ( 2 * index % Size + Parity, 2 *
        index / Size);
    void Swap (int a, int b) {
        (Positions[a], Positions[b]) = (Positions[b], Positions[a]);
        (Alignments[a], Alignments[b]) = (Alignments[b], Alignments[a]);
        Sorted[0] = false;
    }
}
```

Código 24 – PartialSort - Sort

```
[BurstCompile]
bool PartialSort (NativeArray<float2> positions, NativeArray<float2>
    alignments, int size) {
     NativeArray<bool> sorted = new NativeArray<bool>(1, Allocator.
         Persistent);
     sorted[0] = true;

     ColumnSort evenColumn = new() { Positions = positions, Alignments =
         alignments, Parity = 0, Size = size, Sorted = sorted };
     ColumnSort oddColumn = new() { Positions = positions, Alignments =
         alignments, Parity = 1, Size = size, Sorted = sorted };
     RowSort evenRow = new() { Positions = positions, Alignments =
         alignments, Parity = 0, Size = size, Sorted = sorted };
     RowSort oddRow = new() { Positions = positions, Alignments =
         alignments, Parity = 1, Size = size, Sorted = sorted };

     int elementCount = size * size / 2;
     JobHandle evenColumnHandle = evenColumn.Schedule(elementCount, 32);
     JobHandle oddColumnHandle = oddColumn.Schedule(elementCount, 32,
         evenColumnHandle);
     JobHandle evenRowHandle = evenRow.Schedule(elementCount, 32,
         oddColumnHandle);
     JobHandle oddRowHandle = oddRow.Schedule(elementCount, 32,
         evenRowHandle);
     oddRowHandle.Complete();

     bool result = sorted[0];
     sorted.Dispose();
     return result;
}
```

Código 25 – FullSort - Sort

```
[BurstCompile]
void FullSort (NativeArray<float2> positions, NativeArray<float2>
    alignments, int size)
{
     bool sorted = false;
     while (!sorted)
          sorted = PartialSort(positions, alignments, size);
     performedFullSort = true;
}
```