



UNIVERSIDADE DO ESTADO DA BAHIA  
DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA I  
CURSO SUPERIOR TECNOLÓGICO EM JOGOS DIGITAIS

VINÍCIUS MATHEUS SOUZA OLIVEIRA SANTOS

***FALLING SAND SIMULATION: UMA ANÁLISE DE  
DESEMPENHO UTILIZANDO MOTOR DE JOGO *UNITY****

Salvador, Bahia

2023

VINÍCIUS MATHEUS SOUZA OLIVEIRA SANTOS

***FALLING SAND SIMULATION: UMA ANÁLISE DE  
DESEMPENHO UTILIZANDO MOTOR DE JOGO UNITY***

Trabalho de conclusão de curso apresentado  
ao Curso Superior Tecnológico de Jogos Di-  
gitais da Universidade do Estado da Bahia  
(UNEB), como requisito parcial para a obten-  
ção do título de Tecnólogo em Jogos Digitais.

Orientador: Prof. Murilo Boratto

Salvador, Bahia

2023




**UNIVERSIDADE DO ESTADO DA BAHIA**  
Autorização. Decreto nº 9237/86, DOU 18/07/96. Reconhecimento: Portaria 909/95, DOU 01/08-95  
**DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA**  
**CAMPUS I - SALVADOR**


Às **ONZE HORAS** do dia **TREZE DE DEZEMBRO DE DOIS MIL E VINTE E TRÊS**, a banca formada pelos professores **MURILO DO CARMO BORATTO (ORIENTADOR, MEMBRO INTERNO UNEB)**, **THARCÍSIO VAZ DA COSTA DE MORAES (MEMBRO INTERNO, UNEB)** e **LEANDRO COELHO CORREIA (MEMBRO EXTERNO)** se reuniram de forma remota pelo aplicativo Google Meet para julgar a defesa pública de Trabalho de Conclusão de Curso realizada por **VINÍCIUS MATHEUS SOUZA OLIVEIRA SANTOS**, como requisito parcial para obtenção de título de grau superior no curso Tecnológico em Jogos Digitais.

Às **ONZE HORAS E DEZ MINUTOS**, **VINÍCIUS MATHEUS SOUZA OLIVEIRA SANTOS** iniciou a apresentação do **TRABALHO DE CONCLUSÃO DE CURSO**, intitulado **FALLING SAND SIMULATION: UMA ANÁLISE DE DESEMPENHO UTILIZANDO MOTOR DE JOGO UNITY**, de sua autoria. Às **ONZE HORAS E TRINTA MINUTOS**, o professor **THARCÍSIO VAZ DA COSTA DE MORAES** iniciou sua arguição. Em seguida, às **ONZE HORAS E QUARENTA MINUTOS**, o professor **LEANDRO COELHO CORREIA** apresentou seus comentários e questões. Em seguida, às **ONZE HORAS E CINQUENTA MINUTOS**, o professor **MURILO DO CARMO BORATTO** apresentou seus comentários e questões. O discente **VINÍCIUS MATHEUS SOUZA OLIVEIRA SANTOS** respondeu oportunamente todas os comentários e as questões.


Após a apresentação do trabalho científico, a banca se reuniu a parte no mesmo aplicativo, para deliberar o resultado. Considerando a defesa do trabalho científico, bem com a análise do memorial descritivo e da documentação juntada, a banca considerou **APROVADO** com a nota **9,0 (NOVE)**, tendo o pleito do discente **VINÍCIUS MATHEUS SOUZA OLIVEIRA SANTOS** atendido.

Documento assinado digitalmente  
 **MURILO DO CARMO BORATTO**  
Data: 13/12/2023 12:06:53-0300  
Verifique em <https://validar.iti.gov.br>

**Murilo do Carmo Boratto**  
Professor - UNEB

Documento assinado digitalmente  
 **THARCISIO VAZ DA COSTA DE MORAES**  
Data: 13/12/2023 12:20:04-0300  
Verifique em <https://validar.iti.gov.br>

**Tharcísio Vaz da Costa de Moraes**  
Professor - UNEB

Documento assinado digitalmente  
 **LEANDRO COELHO CORREIA**  
Data: 13/12/2023 12:17:34-0300  
Verifique em <https://validar.iti.gov.br>

**Leandro Coelho Correia**  
Convidado Externo

# Resumo

No contexto de desenvolvimento de jogos digitais, tendo como pressuposto que aumentos na eficiência da utilização do recurso computacional podem ser relacionados a melhorias na experiência do usuário jogador, este trabalho propõe apresentar uma análise comparativa entre uma implementação de simulação de tipo *falling sand* de referência, utilizando a abordagem de programação convencional dentro do motor de jogo *Unity*, e implementações que fazem uso das ferramentas *IL2CPP*, *Job System* e *Burst Compiler* disponibilizadas através da iniciativa *Performance by Default*. Norteadado pelo método de teste automatizado apresentado por Borufka (2020) e aliado à metodologia de pesquisa aplicada de natureza experimental, foram realizados testes utilizando diferentes arquiteturas de processador, tamanhos de simulação e modos de compilação para investigar três implementações distintas e determinísticas de uma simulação de tipo *falling sand*. A análise dos dados produzidos nesses testes confirmaram que o uso de ferramentas da iniciativa *Performance by Default* trouxe melhora significativa no uso do recurso computacional, reduzindo o tempo de processamento consumido pelas simulações.

**Palavras-chave:** *falling sand*; jogos digitais; *unity*; *IL2CPP*; *job system*; *burst compiler*.

# Lista de ilustrações

|  |    |
|--|----|
| Figura 1 – Sandspiel, exemplo de <i>falling sand game</i> . . . . .  | 13 |
| Figura 2 – Noita, jogo de gênero <i>roguelike</i> cujo cenário é uma simulação de tipo <i>falling sand</i> . . . . .   | 14 |
| Figura 3 – Espectro de reusabilidade entre <i>software</i> especializado para um único jogo e <i>software</i> generalista capaz de construir jogos distintos . . . . . | 15 |
| Figura 4 – Diagrama da arquitetura do objeto base do motor de jogo. . . . .  | 17 |
| Figura 5 – Diagrama de classes de entidades da simulação . . . . .   | 21 |
| Figura 6 – Gerenciador de módulos do motor de jogo <i>Unity</i> . . . . .  | 23 |
| Figura 7 – Janela de configuração de projeto . . . . .   | 24 |
| Figura 8 – Gerenciador de pacotes do motor de jogo . . . . .   | 25 |
| Figura 9 – Diagrama de classes das diferentes implementações da simulação de tipo <i>falling sand</i> . . . . .  | 26 |
| Figura 10 – Ferramenta de teste de desempenho lado a lado . . . . .  | 28 |
| Figura 11 – Script de configuração de teste automatizado . . . . .   | 29 |
| Figura 12 – Histograma do tempo total de simulação por turno em milissegundos para implementações <i>Conventional</i> e <i>Jobs</i> de tamanho 128x128 . . . . .       | 33 |
| Figura 13 – Tempo acumulado de simulação em milissegundos para implementações <i>Conventional</i> e <i>Jobs</i> de tamanho 128x128 . . . . .                           | 34 |
| Figura 14 – Histograma do tempo total de simulação por turno em milissegundos para implementações <i>Conventional</i> e <i>Jobs</i> de tamanho 256x256 . . . . .       | 35 |
| Figura 15 – Tempo acumulado de execução em milissegundos para implementações <i>Conventional</i> e <i>Jobs</i> de tamanho 256x256 . . . . .                            | 35 |
| Figura 16 – Histograma do tempo total de simulação em milissegundos para implementações <i>Conventional</i> e <i>Jobs</i> de tamanho 512x512 . . . . .                 | 36 |
| Figura 17 – Tempo acumulado de simulação de simulação em milissegundos para implementações <i>Conventional</i> e <i>Jobs</i> de tamanho 512x512 . . . . .              | 37 |

# Lista de tabelas

|          |   |  |    |
|----------|---|--|----|
| Tabela 1 | – | Características das linguagens C++ e C#.   | 16 |
| Tabela 2 | – | Combinações de implementação de simulação e compilador utilizado.  | 30 |
| Tabela 3 | – | Estatísticas para tempo total de simulação em milissegundos para implementações de tamanho 128x128                                     | 33 |
| Tabela 4 | – | Estatísticas para tempo total de simulação em milissegundos para implementações de tamanho 256x256                                     | 34 |
| Tabela 5 | – | Estatísticas para tempo total de simulação em milissegundos para implementações de tamanho 512x512                                     | 36 |
| Tabela 6 | – | Estatísticas para tempo de simulação em milissegundos da etapa <i>ApplyGravity</i> , implementações <i>Jobs</i> e <i>ParallelJobs</i>  | 38 |
| Tabela 7 | – | Estatísticas para tempo de simulação em milissegundos da etapa <i>ResetStatus</i> , implementações <i>Jobs</i> e <i>ParallelJobs</i>   | 38 |
| Tabela 8 | – | Estatísticas para tempo de simulação em milissegundos da etapa <i>DrawToTexture</i> , implementações <i>Jobs</i> e <i>ParallelJobs</i> | 39 |

# Sumário

|            |  |           |
|------------|--|-----------|
| <b>1</b>   | <b>INTRODUÇÃO</b>  | <b>8</b>  |
| <b>2</b>   | <b>FUNDAMENTAÇÃO TEÓRICA</b>   | <b>11</b> |
| <b>2.1</b> | <b>Conceitos de Jogos Digitais</b>   | <b>11</b> |
| 2.1.1      | <i>Falling sand game</i>   | 12        |
| 2.1.2      | Motor de jogo  | 15        |
| 2.1.3      | <i>Unity</i> e suas tecnologias  | 16        |
| <b>2.2</b> | <b>Estado da Arte</b>  | <b>17</b> |
| <b>3</b>   | <b>METODOLOGIA</b>   | <b>19</b> |
| <b>3.1</b> | <b>Implementação de uma simulação de tipo <i>falling sand</i></b>                | <b>20</b> |
| <b>3.2</b> | <b>Implementação das ferramentas da iniciativa <i>Performance by Default</i></b> | <b>22</b> |
| 3.2.1      | IL2CPP   | 23        |
| 3.2.2      | <i>Job System</i>  | 24        |
| 3.2.3      | <i>Burst Compiler</i>  | 27        |
| <b>3.3</b> | <b>Métricas e procedimento de obtenção de dados</b>                              | <b>27</b> |
| <b>4</b>   | <b>ANÁLISE DOS RESULTADOS</b>  | <b>31</b> |
| <b>4.1</b> | <b>Comparação com a implementação convencional</b>                               | <b>32</b> |
| 4.1.1      | Simulações tamanho 128x128   | 32        |
| 4.1.2      | Simulações tamanho 256x256   | 33        |
| 4.1.3      | Simulações tamanho 512x512   | 35        |
| <b>4.2</b> | <b>Comparação entre implementação de etapas sequenciais e paralelas</b>          | <b>37</b> |
| <b>5</b>   | <b>CONCLUSÃO</b>   | <b>40</b> |
|            | <b>REFERÊNCIAS</b>   | <b>43</b> |



# 1 Introdução

Dados recentes indicam a existência de cerca de 440 milhões de dispositivos digitais no Brasil. Entre computadores, notebooks, *tablets* e *smartphones*, são aproximadamente 2 dispositivos por habitante (MEIRELLES, 2021). Estes dispositivos conquistaram um espaço abrangente no cotidiano e estão sendo utilizados em setores diversos como, por exemplo, os setores de saúde, transporte, segurança, pesquisa e, principalmente, para o objeto de interesse deste trabalho, o setor de entretenimento. Este panorama indica que o acesso a dispositivos eletrônicos capazes de executar jogos digitais é ubíquo e possibilita que faixas amplas da população, desde os mais jovens até os mais idosos, consumam jogos digitais (NEWZOO, 2021).

O desenvolvimento de jogos digitais é uma atividade de natureza multidisciplinar capaz de envolver áreas do conhecimento tão díspares quanto a psicologia e a engenharia de computação (KOSTER, 2014). Uma das preocupações durante a realização da atividade de desenvolvimento de jogos tem a ver com métricas de desempenho da execução do jogo pelo recurso computacional e o seu efeito na experiência do usuário jogador: o quão rápido o jogo responde as entradas do usuário jogador, quantos personagens são desenhados em tela simultaneamente, qual a resolução da imagem gerada na tela ou mesmo quantas vezes por segundo é possível processar a simulação física de um cenário. Dentre estas métricas, uma das principais, senão a principal, é a quantidade de imagens por segundo exibidas em tela no dispositivo de saída de imagem utilizado pelo jogo. A grande maioria dos jogos tem como alvo atingir uma taxa de 30 ou 60 quadros por segundo, o que implica na necessidade de que todo o processamento relacionado a cada imagem exibida seja realizado em menos de  $33, \bar{3}$  ou  $16, \bar{6}$  milissegundos, respectivamente, com o intuito de manter a ilusão de fluidez de movimento a partir da exibição contínua de imagens estáticas (GREGORY, 2018).

A busca contínua por melhorias nestas métricas de desempenho transformou o jogo digital numa categoria de *software* capaz de utilizar grandes quantidades de recursos computacionais, acompanhando de perto a vanguarda de avanços tecnológicos de *hardware* (SALEN; ZIMMERMAN, 2012). No início do século XXI, devido a limitações no processo de fabricação de semicondutores, fabricantes de processadores adotaram o modelo *SMP*, ou multiprocessamento simétrico, em que a *CPU* possui múltiplos núcleos de processamento idênticos num único substrato de silício. Este cenário trouxe para o primeiro plano o paradigma de programação paralela: a abordagem de programação puramente sequencial não é capaz de acessar os ganhos de desempenho trazidos por processadores modernos onde múltiplos núcleos podem cooperar para realizar tarefas mais rapidamente trabalhando em

paralelo (KIRK; HWU, 2016; GREGORY, 2018).

Entre os muitos tipos de ferramentas utilizados no desenvolvimento de jogos digitais estão os motores de jogo e, dentre estes, um dos mais utilizados é o motor *Unity*. No sentido de permitir melhorias na experiência do usuário jogador e tornar possível ao usuário desenvolvedor atingir um patamar maior de desempenho computacional através da utilização dos recursos existentes em processadores modernos, *Unity Technologies*, a empresa responsável pelo desenvolvimento do motor de jogo *Unity* assumiu, a partir de 2018, o lema *Performance by Default* e tem implementado e tornado disponíveis novas ferramentas com o foco em prover acesso à construtos de programação paralela acessíveis, normalmente, apenas em linguagens denominadas de baixo nível, ou seja, que trabalham com um nível menor de abstração e modelam de forma mais aproximada o funcionamento do *hardware* (KROGH-JACOBSEN, 2018). As principais ferramentas desta iniciativa são:

- a) *Intermediate Language to C++*, ou IL2CPP, um transpilador capaz de traduzir código MSIL para código na linguagem C++;
- b) *Burst Compiler*, um compilador capaz de utilizar um subconjunto da linguagem C# para gerar código de máquina com alto grau de otimização;
- c) *Job System*, uma biblioteca para codificação de algoritmos paralelizáveis e gerenciamento de execução destes em tarefas. Também fornece acesso à execução de código utilizando a *thread pool* interna do motor de jogo;
- d) *Entity Component System*, uma biblioteca para codificação de sistemas de jogo que segue o paradigma de programação de mesmo nome.

A junção destas ferramentas e da aplicação de técnicas de programação paralela pode fornecer um caminho para uma utilização mais eficiente do recurso computacional e, conseqüentemente, para uma melhora em métricas de desempenho com influência direta na experiência do usuário jogador.

Este trabalho apresenta um estudo de desempenho e escalabilidade de uma implementação de simulação de tipo *falling sand* utilizando as ferramentas convencionais da linguagem C# em comparação com implementações que utilizem as ferramentas IL2CPP, *Burst Compiler* e *Job System* dentro do motor de jogo *Unity*, mensurando o impacto do uso destas ferramentas no desempenho da simulação e na utilização dos recursos computacionais em duas plataformas de *hardware* distintas, uma utilizando arquitetura de instrução AMD<sup>®</sup> x86-64 e a outra ARM<sup>®</sup> aarch64 e, por fim, analisar os resultados obtidos para determinar se é plausível utilizar estas ferramentas para obter ganhos de eficiência no uso do recurso computacional, influenciando positivamente a experiência do usuário jogador.

O texto está organizado da seguinte forma: na seção 2 serão contextualizados as fundamentações e os referenciais teóricos utilizados no projeto, na seção 3 serão apresentados as características do projeto, na seção 4 serão apresentados os resultados experimentais obtidos nesta pesquisa. Finalizamos este trabalho na seção 5 com conclusões e trabalhos futuros.

## 2 Fundamentação Teórica

Neste capítulo o trabalho propõe fundamentar conceitos importantes para a construção deste processo de pesquisa. Serão discutidos definições de jogos digitais, detalhes do motor do jogo *Unity* e alguns trabalhos relacionados.

### 2.1 Conceitos de Jogos Digitais

Em seu livro “Regras do Jogo: Fundamentos do Design de Jogos” Salen e Zimmerman (2012, p.102) discorrem sobre como o jogo digital pode ser analisado “como um sistema formal de regras, como um sistema experimental de jogo ou como um sistema contextual incorporado nos sistemas maiores da cultura” e ao descrever características do objeto acabam por estabelecer uma relação entre jogo digital, *hardware* e *software*:

Os jogos digitais são sistemas [...] O meio físico do computador é um elemento que compõe o sistema do jogo, mas não representa todo o jogo. O *hardware* e o *software* do computador são apenas os materiais dos quais o jogo é composto. (SALEN; ZIMMERMAN, 2012, p.102)

Para os autores, portanto, “a tecnologia digital em si é uma parte do sistema, mas certamente não a constitui inteiramente” e as “relações cognitivas e psicológicas, físicas e emocionais que surgem entre um jogador e o jogo” (SALEN; ZIMMERMAN, 2012, p.103) também são elementos constituintes do jogo digital. Outro aspecto relevante trazido pelos autores é a diversidade de formas que estes elementos constituintes podem assumir:

Os jogos digitais e eletrônicos assumem uma grande quantidade de formas e são projetados para muitas plataformas de computadores diferentes. Esses incluem jogos para computadores pessoais ou consoles de jogo conectados à TV, tais como o *Playstation* da *Sony* ou o *Microsoft XBox*; dispositivos de jogos portáteis, como, por exemplo, o *Nintendo Game Boy Advance* ou os portáteis de apenas um jogo; jogos para PDAs ou telefones celulares; e jogos para máquinas *arcade* ou parques de diversões. Jogos digitais e eletrônicos podem ser projetados para um único jogador, para um pequeno grupo de jogadores ou para uma grande comunidade. (SALEN; ZIMMERMAN, 2012, p. 102)

Ao fazer uma análise das qualidades pertencentes ao jogo digital que o diferenciam de outras mídias, os autores destacam quatro elementos: interatividade imediata porém restrita, manipulação da informação, sistemas automatizados complexos e comunicação

em rede. Para o propósito deste trabalho é interessante focar no terceiro aspecto, sistemas automatizados complexos:

Talvez, a característica mais predominante dos jogos digitais seja que eles podem automatizar procedimentos complexos e, assim, facilitar a disputa do jogo que seria muito complicada em um contexto não informatizado. Na maioria dos jogos não digitais, os jogadores têm de fazer avançar a partida a cada passo, através da manipulação das peças ou comportando-se de acordo com as instruções explícitas descritas pelas regras. Em um jogo digital, o programa pode automatizar esses procedimentos e fazer avançar o jogo sem a entrada direta de um jogador. [...] Esse é exatamente o tipo de complexidade com o qual os computadores lidam com facilidade. (SALEN; ZIMMERMAN, 2012, p.105)

Juul (2005) dialoga com este aspecto quando caracteriza o jogo digital como “um sistema formal baseado em regras, com resultados quantificáveis e variáveis”<sup>1</sup>, parte de uma dualidade composta de um mundo ficcional e de regras reais.

O gênero de jogo de interesse deste trabalho, *falling sand game*, trabalha de forma intensa o conceito de jogo como sistema automatizado complexo e faz uso do recurso computacional para aplicar regras de complexidade arbitrária muitas vezes por segundo.

### 2.1.1 *Falling sand game*

*Falling sand game*, também conhecido como *sand art game* ou *sand painting game*, é um gênero de jogo que combina elementos de simulação física e autômato celular em um *canvas* interativo onde cada *pixel* da imagem representa um material específico e o jogador pode alterar livremente propriedades de cada *pixel* no cenário utilizando como método de interação um pincel virtual capaz de pintar diretamente as estruturas de dados que contem os *pixels* e os materiais que estes representam. Expandindo sobre estes elementos constituintes do gênero, é possível justificar:

- a) Simulação física, pois cada *pixel* do *canvas* tem armazenado em sua estrutura de dados valores que representam propriedades físicas como, por exemplo, estado físico, cor, velocidade, densidade e sofrem ação de processos análogos a processos naturais existentes como gravidade, troca de calor, reações químicas e físicas;
- b) Autômato celular, pois, partindo da definição de Wolfram (1983):

Autômatos celulares são idealizações matemáticas simples de sistemas naturais. Construídos a partir de uma sequência de estruturas idênticas, cada uma capaz de assumir um valor único dentre um conjunto finito de possibilidades distintas. Os valores nestas estruturas mudam em intervalos de tempo

<sup>1</sup> No original: “1. a rule-based formal system; 2. with variable and quantifiable outcomes;”

discretos em função de regras determinísticas que definem o próximo valor da estrutura em função dos valores existentes em estruturas vizinhas. (WOLFRAM, 1983, p.4, tradução nossa)<sup>2</sup>

É possível verificar que estas características estão presentes na simulação de tipo *falling sand* já que cada *pixel* é armazenado numa célula dentro de uma estrutura de dados em formato de matriz, seu comportamento varia em intervalos de tempo discretos e seu valor computado é influenciado pelos valores contidos nas células vizinhas imediatas.

Figura 1 – Sandspiel, exemplo de *falling sand game*



Fonte: Bittker (2019).

A partir da experiência de desenvolvedores envolvidos na criação de jogos pertencentes ao gênero é possível identificar interseções com conceitos previamente expostos. Max Bittker, criador de *Sandspiel*<sup>3</sup>, visto na Figura 1, descreve sua perspectiva sobre o ato de jogar um jogo deste gênero e a forma com que elementos de jogabilidade emergente vêm a tona a medida que o jogador experimenta as diferentes combinações de materiais possíveis no cenário:

Em cada partida, o jogador tem acesso a uma paleta de materiais sólidos, líquidos e gases virtuais e cabe a ele pintá-los na tela para descobrir como eles interagem entre si. Ao brincar com estes elementos o jogador estará criando hipóteses, construindo experimentos e inventando histórias e jogos dentro da sua própria imaginação. (BITTKER, 2019, tradução nossa)<sup>4</sup>

<sup>2</sup> No original: “Cellular automata are simple mathematical idealizations of natural systems. They consist of a lattice of discrete identical sites, each site taking on a finite set of, say, integer values. The values of the sites evolve in discrete time steps according to deterministic rules that specify the value of each

Figura 2 – Noita, jogo de gênero *roguelike* cujo cenário é uma simulação de tipo *falling sand*



Fonte: Nolla Games (2020).

*Noita*<sup>5</sup>, visto na figura Figura 2, é outro exemplo de jogo recente, de maior proeminência e pertencente ao gênero. Em entrevista a um portal especializado em desenvolvimento de jogos, Petri Purho, programador envolvido no desenvolvimento do jogo *Noita* discorre sobre a natureza deste tipo de simulação e como suas regras são definidas:

Essencialmente, é um autômato celular complexo. Cada *pixel* no jogo segue um conjunto de regras que, embora sejam simples isoladas, quando combinadas geram resultados surpreendentes e inesperados. Por exemplo, *pixels* representando materiais líquidos primeiro verificam se eles podem se mover para baixo. Caso não seja possível, eles verificam à direita e à esquerda para determinar se é possível o deslocamento nestas direções. Com estas regras é possível simular, de forma rudimentar, um fluido bidimensional. [...] O fogo também segue regras simples. Quando um pixel está pegando em chamas ele irá gerar uma direção aleatória e determinar se é possível incendiar o pixel vizinho nesta direção. Óleo é inflamável e água é não-inflamável de forma que o fogo se espalha através do óleo mas não se espalha através da água. Quando o fogo entra em contato com a água ele é convertido em vapor. (PURHO, 2019b, tradução nossa)<sup>6</sup>

site in terms of the values of neighboring sites.”

<sup>3</sup> Disponível em: <<https://sandspiel.club>>

<sup>4</sup> No original: “In each game, you’re given a palette of virtual solids, liquids, and gasses, and it’s up to you to paint them onto the screen to discover how they interact. [...] Playing with the elements means asking questions, building experiments, and inventing stories and games of your own.”

<sup>5</sup> Disponível em: <<https://noitagame.com>>.

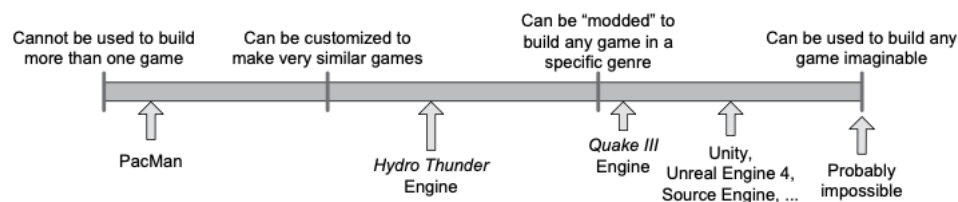
<sup>6</sup> No original: “Essentially, it’s complex cellular automata. Every pixel in the game follows rather simple rules, but when you combine them together you get surprising and unexpected results. For example, liquid pixels first check if they can go down. If not, they check left and right to see if they can move

Uma simulação deste tipo foi escolhida pelo autor para servir como carga de teste de comparação entre implementações utilizando as novas tecnologias oferecidas pela iniciativa *Performance by Default* do motor de jogo Unity

### 2.1.2 Motor de jogo

O termo motor de jogo, ou *game engine*, surgiu na década de 90 junto com os jogos de tiro em primeira pessoa desenvolvidos pela *id Software* e adveio da constatação da existência de benefícios funcionais à atividade de desenvolvimento com a separação do *software* responsável por sistemas centrais<sup>7</sup> do *software* responsável pelas regras do jogo e do conteúdo do jogo representado por seus *assets*<sup>8</sup> (GREGORY, 2018).

Figura 3 – Espectro de reusabilidade entre *software* especializado para um único jogo e *software* generalista capaz de construir jogos distintos



Fonte: Gregory (2018).

De forma sucinta, o motor de jogo é um “*software* que é passível de ser estendido e pode ser utilizado como base para o desenvolvimento de jogos diversos sem necessitar de modificações consideráveis”<sup>9</sup> (GREGORY, 2018, p.12, tradução nossa). Como ilustrado na Figura 3, *Unity* é um motor de jogo generalista, capaz de ser utilizado na criação de jogos de gêneros diversos.

that way. With those rules, you get a rudimentary 2D liquid. [...] The fire is kind of simple as well. When something is on fire it will look in a random direction to see if it can ignite that pixel. Oil is flammable, but water is not so the fire will spread in the oil, but stop when it hits the water. If the fire tries to ignite the water it will convert into steam instead.”

<sup>7</sup> O autor cita, como exemplo, os sistemas de renderização tridimensional, detecção de colisões e de reprodução de sons.

<sup>8</sup> O autor cita, como exemplo, arquivos contendo texturas, modelos, sons, mapas, dados de balanceamento que compõem os dados a serem utilizados pelo motor de jogo para executar o jogo em si.

<sup>9</sup> No original: “software that is extensible and can be used as the foundation for many different games without major modification”



### 2.1.3 *Unity* e suas tecnologias

O *Unity* é um motor de jogo proprietário, de código fonte fechado, multiplataforma<sup>10</sup>, desenvolvido pela empresa *Unity Technologies* (UNITY, 2020b). Para o propósito deste trabalho é interessante focar nos aspectos técnicos relacionados ao *Unity* como ambiente de programação e sua relação com as linguagens C++ e C#.

Tabela 1 – Características das linguagens C++ e C#.

|                          | C++                | C#                               |
|--------------------------|--------------------|----------------------------------|
| Paradigma                | Orientado a objeto | Orientado a objeto               |
| Gerenciamento de memória | Manual             | Automatizado com coletor de lixo |
| Compiladores             | gcc, clang, msvc   | mcs, csc, roslyn                 |
| Modo de Compilação       | AOT                | JIT                              |

Fonte: Elaborada pelo autor.

O motor de jogo *Unity* é construído utilizando a linguagem C++ nos seus sistemas centrais e expõe, para seus usuários desenvolvedores, uma API<sup>11</sup> que utiliza a linguagem C#. Embora ambas pertençam à mesma família de linguagens descendentes do C, existem diferenças mecânicas, explicitadas na Tabela 1, que impedem que executáveis e bibliotecas criadas utilizando estas linguagens possam ser associados dinamicamente, o que acaba por impedir a possibilidade de interoperar objetos criados utilizando estas duas linguagens de forma eficiente. Esta natureza dual do motor de jogo traz consigo uma série de peculiaridades que afetam o desempenho, em tempo de execução, e a ergonomia, durante o processo de desenvolvimento:

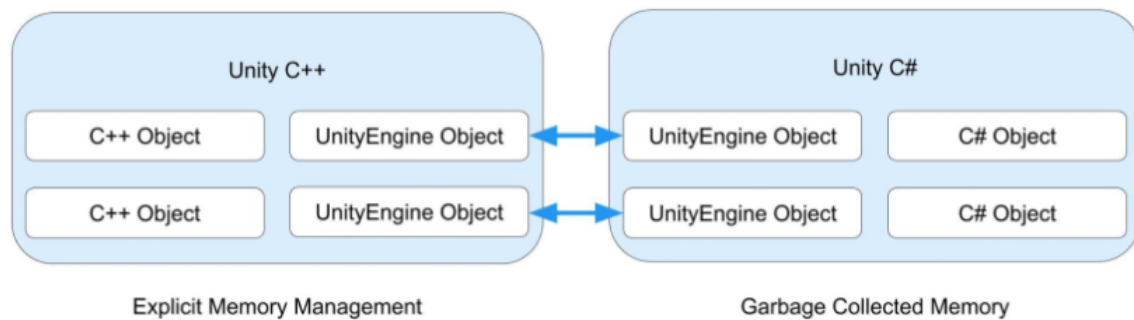
- a) `UnityEngine.Object` é o objeto base do motor de jogo e, como pode ser visto na Figura 4, é um objeto especialmente projetado para cruzar a barreira de interoperabilidade entre as duas linguagens.
- b) Os objetos que herdam de `UnityEngine.Object` possuem duas representações em memória, uma em cada um dos lados do motor, que são armazenados em endereços de memória diferentes (UNITY, 2020a; MUTEL, 2018)

Essas diferenças mecânicas implicam na necessidade de realizar processos serialização e desserialização quando há a necessidade de comunicação entre os dois lados do motor de jogo, impactando negativamente o desempenho de aplicações desenvolvidas no *Unity*. Este aspecto é importante para o trabalho pois parte das ferramentas da iniciativa

<sup>10</sup> Numa lista não exaustiva é possível citar as plataformas: PC (Windows, MacOS, Linux), WebGL, Android, iOS, Nintendo Switch, Xbox (One, One X, Series S/X), Playstation (4, 5).

<sup>11</sup> Sigla em inglês para Interface de Programação de Aplicação.

Figura 4 – Diagrama da arquitetura do objeto base do motor de jogo.



Fonte: Unity (2020a).

*Performance by Default* têm como efeito reduzir a necessidade de comunicação, trazendo elementos que antes existiam do lado C# para o lado C++ do motor de jogo.

É válido ressaltar as diferenças do modo de compilação entre as duas linguagens: a linguagem C++ é compilada utilizando a estratégia *ahead-of-time* ou AOT, onde o processo de compilação transforma o código escrito pelo usuário programador em código de máquina que pode ser executado diretamente pelo processador (CHAMBERS, 2002); em comparação, a linguagem C# é compilada utilizando a estratégia *just-in-time* ou JIT, onde o processo de compilação transforma código escrito pelo usuário programador em uma representação intermediária, independente de arquitetura, que é transformada em código nativo durante o tempo de execução (MICROSOFT, 2021). Uma das formas que o motor de jogo utiliza para mitigar o efeito desta diferenças reside no conceito de transpilação: quando o código de uma linguagem de programação é convertido para outra linguagem de programação utilizando um *software* especializada para esta tarefa. No caso do motor de jogo *Unity* este processo é realizado pela ferramenta IL2CPP que será discutido de forma mais aprofundada na subseção 3.2.1.

## 2.2 Estado da Arte

Durante o processo de ideação deste trabalho foram buscados estudos semelhantes no intuito de informar quais caminhos já haviam sido explorados por outros pesquisadores e auxiliar na formação de hipóteses:

- a) *Exploring Game Industry Technological Solutions to Simulate Large-Scale Autonomous Entities within a Virtual Battlespace*, neste paper McCullough et al. (2019) propuseram testar a utilização de ferramentas da iniciativa *Performance*

*by Default* no motor de jogo *Unity* para simular 1 milhão de agentes autônomos. Estes agentes foram programados para se deslocar em um ambiente virtual modelado a partir da captura de imagens de alta resolução de uma localização real. Para efeitos de comparação, os agentes foram implementados utilizando duas abordagens: uma convencional orientada a objeto que foi utilizada como referência e uma abordagem baseada no design orientado a dados, utilizando as ferramentas *Entity Component System*, *Job System* e *Burst Compiler*. Utilizando como mesa de teste um computador montado com peças disponíveis no mercado consumidor, constataram que o uso das ferramentas propostas trouxe ganho de desempenho significativo. A implementação de referência conseguiu simular cerca de 2000 agentes enquanto a simulação proposta conseguiu simular cerca de 150.000 agentes.

- b) *Performance Testing Suite for Unity DOTS*, é o trabalho norteador desta pesquisa. Nesta dissertação de mestrado Borufka (2020) propõe uma metodologia de teste de desempenho automatizada com geração de resultados em arquivos *csv*<sup>1</sup> para posterior análise estatística. O autor utiliza a metodologia de teste proposta para identificar quais os impactos, em termos de utilização do recurso computacional, das recomendações presentes na documentação das ferramentas da iniciativa *Performance by Default* e, posteriormente, faz uma série de recomendações próprias baseadas no resultado dos testes: reforça que as ferramentas *Job System* e *Burst Compiler* devem ser usadas em conjunto; indica a utilização de uma estratégia de alocação persistente quando utilizando os tipos genéricos derivados de `NativeArray<T>` do namespace `Unity.Collections` entre outras recomendações menos relevantes para este processo de pesquisa.
- c) Análise de Desempenho do Algoritmo de *Flocking* em uma Implementação Baseada no Design Orientado a Dados, neste trabalho de conclusão de curso, Alves (2020) investiga as características de desempenho de um algoritmo de tipo *flocking* implementado utilizando duas abordagens: uma abordagem orientada a objeto e outra baseada no design orientado a dados. Neste trabalho, o autor não conseguiu identificar diferenças significativas no desempenho das duas implementações porém sugere que utilização conjunta das ferramentas *Job System* e *Burst Compiler* poderia mudar este cenário.

O estudo dos conceitos e dos trabalhos científicos abordados neste capítulo formaram a base do conhecimento que amparou o processo de produção desta pesquisa.

---

<sup>1</sup> Do inglês, *comma-separated values*: um padrão de formatação tabular de dados escrita em texto

### 3 Metodologia

A introdução de novas ferramentas focadas na melhoria de desempenho para o motor de jogo *Unity*, parte da iniciativa *Performance by Default*, e os trabalhos de McCullough et al. (2019), Alves (2020) e Borufka (2020) possibilitam sugerir, como hipótese, que a utilização destas técnicas é capaz de proporcionar uma utilização mais eficiente do recurso computacional e, por sua vez, uma melhora na experiência do usuário jogador no contexto de uma simulação de tipo *falling sand*.

Este trabalho teve como objetivos, a partir da implementação de uma simulação de tipo *falling sand*:

- a) Realizar testes comparando desempenho e escalabilidade entre uma implementação base, sem fazer uso das ferramentas da iniciativa *Performance by Default* (a qual será chamada daqui por diante de abordagem convencional), e implementações que utilizam as ferramentas *IL2CPP*, *Job System* e *Burst Compiler*;
- b) Determinar se o uso dessas ferramentas possibilitou uma utilização mais eficiente do recurso computacional;

Para tanto, o trabalho foi realizado nos moldes de uma pesquisa aplicada de natureza experimental. Segundo Silva, Guerra et al. (2011), na pesquisa aplicada “o investigador é movido pela necessidade de contribuir para fins práticos mais ou menos imediatos, buscando soluções para problemas concretos” e, segundo Gil (2002), a sua natureza experimental “consiste em determinar um objeto de estudo, selecionar as variáveis que seriam capazes de influenciá-lo, definir as formas de controle e de observação dos efeitos que a variável produz no objeto”.

Neste sentido, o processo de desenvolvimento da pesquisa se deu da seguinte forma: em um primeiro momento foi realizada a implementação de uma simulação de tipo *falling sand* simplificada, que simulava apenas dois tipos de material, como prova de conceito; esta simulação inicial foi modificada de forma que se tornasse determinística, ou seja, que não existissem fontes de aleatoriedade e que a partir de uma determinada entrada sempre fosse produzido o mesmo resultado final na simulação; em seguida, a simulação foi modificada para implementar mais tipos de material, totalizando seis tipos, como pode ser visto no `enum Kind` na figura Figura 5; a partir desta implementação utilizando a abordagem convencional foram criadas duas novas implementações (*Jobs* e *ParallelJobs*)

que utilizam as ferramentas da iniciativa *Performance by Default*; seguindo recomendações realizadas por Borufka (2020) foi criada uma infraestrutura de testes e coleta de dados automatizada, capaz de executar testes utilizando as três implementações e compilar os resultados em arquivos de tipo *csv*; com os resultados dos testes foram criados *scripts* para extrair informações estatísticas da massa de dados e gerar visualizações em forma de gráficos; por fim os dados foram analisados com o intuito de inferir conclusões.

A pesquisa, as implementações das simulações e os testes de desempenho foram escritos na linguagem de programação C# e realizados utilizando, como ambiente de desenvolvimento, a versão 2022.3.7f1 do motor de jogo *Unity* com a adição do módulo IL2CPP e dos pacotes *Burst* versão 1.8.11 e *Collections* versão 2.1.4. Para análise dos resultados e geração de tabelas e gráficos foram utilizados os pacotes *pandas*, *numpy*, *scipy* e *matplotlib* na linguagem Python: foram criados *scripts*<sup>1</sup> que analisam os arquivos de resultado e, utilizando das ferramentas das bibliotecas citadas acima, geram valores de média, mediana, desvio padrão, erro padrão de intervalo de confiança<sup>2</sup> para as implementações testadas.

### 3.1 Implementação de uma simulação de tipo *falling sand*

Considerando o conceito de *falling sand game* apresentado na fundamentação teórica, a implementação de uma simulação desta natureza implica na criação de estruturas de dados que retratem o mundo virtual a ser simulado em memória e que possibilitem modelar a interação de elementos existentes em formato de algoritmos. Com este objetivo foram implementadas as estruturas<sup>3</sup> *Kind*, *Status*, *Velocity*, *Pixel* e *World*, como pode ser visto na Figura 5:

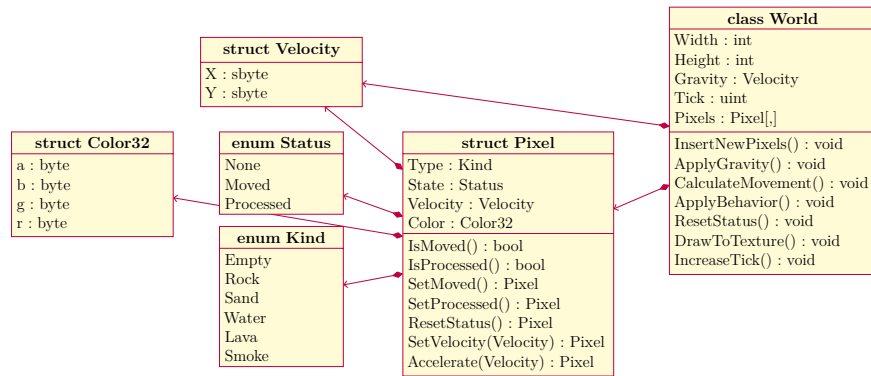
- a) *Kind* é uma enumeração que indica de qual o tipo de material o *pixel* é feito. O tipo de material é utilizado para determinar qual o comportamento da entidade durante a simulação;
- b) *Status* é uma enumeração de tipo sinalizador que indica por quais etapas da simulação um *pixel* foi processado. Os *pixels* iniciam cada turno com o estado *None* e tem seu estado modificado de forma que suas rotinas de movimento e comportamento sejam executadas apenas uma vez por *pixel* por turno;
- c) *Velocity* é uma estrutura que representa uma velocidade em unidades por turno para os eixos horizontal e vertical;

<sup>1</sup> Estes *scripts* podem ser encontrados em <<https://github.com/vimsos/unity-pbd-comparison/tree/main/plot>>.

<sup>2</sup> Para o cálculo de intervalo de confiança foi utilizado o valor de  $T^*$  de 3.6.

<sup>3</sup> A estrutura *Color32* é implementada pelo motor de jogo *Unity* e representa uma cor em termos de seus componentes de cores primárias e opacidade.

Figura 5 – Diagrama de classes de entidades da simulação



Fonte: Elaborada pelo autor.

- d) **Pixel** é uma estrutura que representa o elemento mínimo da simulação, contém o estado de um *pixel* armazenando o seu tipo de material, qual o status, sua velocidade e sua cor. Provê métodos que encapsulam rotinas para ler e modificar o estado;
- e) **World** é a classe que representa o mundo a ser simulado, possui propriedades que determinam a o tamanho da simulação em altura e largura, a intensidade da gravidade dentro da simulação, contabiliza quantos turnos foram simulados e armazena numa matriz bidimensional todo o estado dos *pixels* dentro do mundo virtual sendo que os índices de acesso dos elementos da matriz representam as coordenadas horizontal e vertical de cada *pixel*. Provê, também, métodos responsáveis por expor as etapas da simulação;

Com estas estruturas foi possível criar algoritmos que, à partir de um estado inicial, modificam os valores em memória para gerar um novo estado da simulação. Estes algoritmos foram divididos em etapas e estas são executadas de forma sequencial para avançar a simulação em um turno:

1. **InsertNewPixels**: adiciona novos *pixels* na simulação<sup>4</sup>;
2. **ApplyGravity**: altera a velocidade de *pixels* existentes dentro da simulação de acordo com o valor de gravidade configurado na classe **World**;
3. **CalculateMovement**: calcula as trajetórias e colisões dos *pixels* existentes dentro da simulação, nesta etapa o status de cada *pixel* processado é modificado de forma a indicar que ele já teve seu movimento calculado;

<sup>4</sup> Numa simulação interativa este seria o momento onde as ações do usuário jogador seriam inseridas dentro da simulação.

4. **ApplyBehavior**: calcula qual o comportamento dos *pixels* de acordo com os seus vizinhos imediatos<sup>5</sup>, nesta etapa o status de cada *pixel* processado é modificado de forma a indicar que ele já teve seu comportamento calculado;
5. **ResetStatus**: modifica o status de todos os *pixels* para o valor **None**, em preparação para o processamento de um novo turno da simulação;
6. **DrawToTexture**: lê o estado atual e copia os valores de cor de cada *pixel* para uma textura que pode ser utilizada para visualização do estado da simulação;
7. **IncreaseTick**: incrementa o valor da variável **Tick** da classe simulação para indicar que um turno de simulação foi executado por completo.

Tendo em vista que as implementações da simulação seriam compiladas utilizando diferentes compiladores e utilizando diferentes APIs do motor de jogo, um cuidado especial foi tomado para que as diferentes implementações da simulação produzissem resultados finais idênticos quando fossem submetidas as mesmas entradas e executadas pela mesma quantidade de turnos. Para tanto, todas as propriedades das estruturas de dados que são parte de operações aritméticas foram implementadas utilizando variáveis de tipo número inteiro em vez de tipo ponto flutuante.

### 3.2 Implementação das ferramentas da iniciativa *Performance by Default*

A iniciativa *Performance by Default* teve seu primeiro lançamento público com a versão 2018.1 do motor de jogo (KROGH-JACOBSEN, 2018), quando tornou públicas ferramentas que oferecem alternativas para implementação de algoritmos e lógica de jogo com um desempenho otimizado, capaz de melhor utilizar o recurso computacional em comparação com implementações convencionais utilizando os recursos oferecidos pelo motor de jogo até então.

As ferramentas analisadas no contexto deste trabalho foram: *IL2CPP*, *Job System*, *Burst Compiler* e *Entity Component System*. Por serem ferramentas que foram tornadas disponíveis inicialmente em caráter experimental, estas passaram por mudanças significativas em suas APIs entre o momento que foram inicialmente lançadas e a data de realização deste trabalho, com o pacote *Entity Component System* tendo sua API estabilizada apenas recentemente com o lançamento da versão 1.0.8 em Abril de 2023 (UNITY, 2023b).

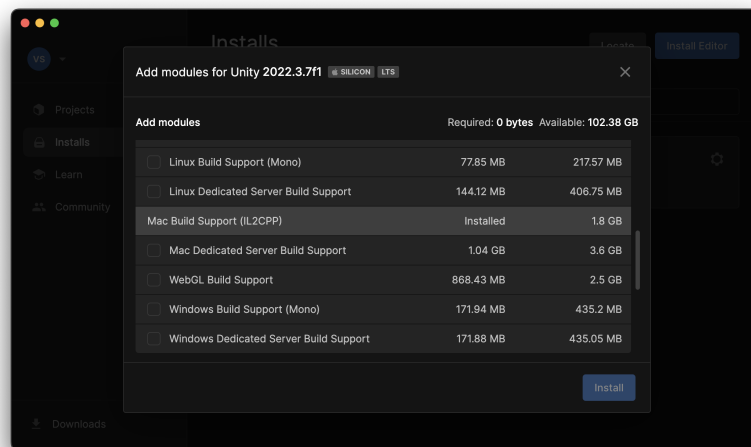
---

<sup>5</sup> Por exemplo, um *pixel* do material lava pode transformar um *pixel* adjacente de tipo água para fumaça, ou de tipo areia para pedra.

Com a implementação da simulação de tipo *falling sand* utilizando a abordagem convencional do motor de jogo, um processo de análise foi realizado para avaliar quais ferramentas da iniciativa seriam utilizadas no processo de construção dos testes de desempenho. Chegou-se à conclusão que para fazer uso da ferramenta *Entity Component System* seriam necessárias modificações muito extensas no código da simulação e, portanto, seria mais proveitoso para o processo de pesquisa excluir a ferramenta *Entity Component System*. Logo, seguiu-se em frente buscando testar as hipóteses com a implementação das ferramentas *IL2CPP*, *Job System* e *Burst Compiler*.

### 3.2.1 IL2CPP

Figura 6 – Gerenciador de módulos do motor de jogo *Unity*



Fonte: Elaborada pelo autor.

*Intermediate Language to C++*, ou *IL2CPP*, é uma ferramenta categorizada como *scripting backend*<sup>6</sup> pela documentação oficial do motor de jogo. Seu principal objetivo é permitir que aplicações construídas utilizando o motor de jogo *Unity* possam ser executadas em plataformas que não possuem suporte à compilação e execução JIT. Para tanto, a ferramenta realiza um processo de transpilação do código MSIL (gerado pela compilação do código original em C#) em código C++ e, em seguida, executa um processo de compilação AOT utilizando o compilador *clang*, gerando código de máquina específico para uma determinada plataforma alvo. Um efeito do uso desta ferramenta é a criação de executáveis que podem ser mais eficientes no uso do recurso computacional quando comparados com

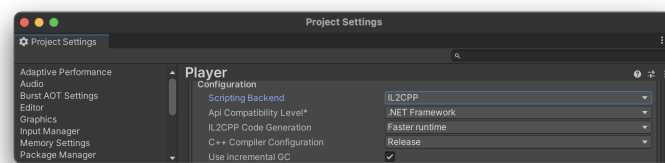
<sup>6</sup> Ferramenta interna do motor de jogo responsável por transformar os scripts criados pelo usuário programador em código executável (UNITY, 2020b).



executáveis gerados pelo *scripting backend* padrão *mono* (UNITY, 2020b; PETERSON, 2015). Neste trabalho, as referências feitas aos diferentes *scripting backends* serão realizadas os chamando de compiladores.

A ferramenta IL2CPP é instalada como módulo durante a instalação do motor de jogo, como pode ser visto na Figura 6. Para configurar um projeto de forma a utilizar a ferramenta IL2CPP é necessário acessar a janela *Project Settings*, dentro da aba *Player*, e modificar a linha *Scripting Backend*, selecionando a opção IL2CPP, como pode ser visto na Figura 7. A partir deste momento, os executáveis gerados serão compilados utilizando a estratégia AOT e o compilador IL2CPP. Não foi necessário realizar modificações no código da aplicação para fazer uso desta ferramenta.

Figura 7 – Janela de configuração de projeto



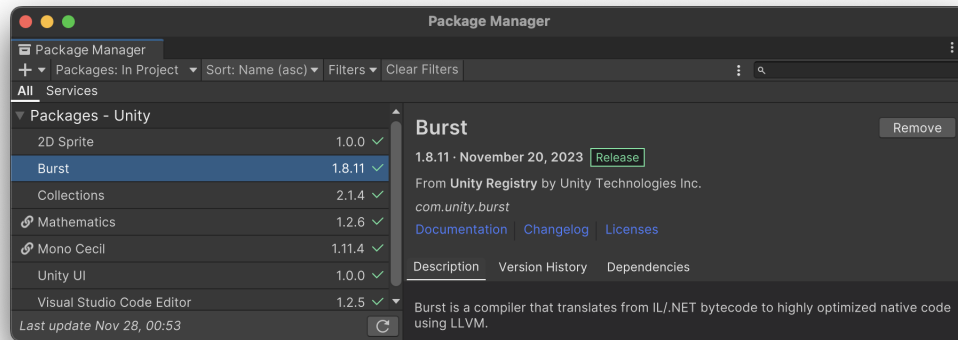
Fonte: Elaborada pelo autor.

### 3.2.2 Job System

*Job System* é uma biblioteca cujo objetivo é fornecer ferramentas para codificação de algoritmos paralelizáveis e gerenciamento de execução destes em tarefas. Uma etapa, dentro do contexto da biblioteca, é chamada de *Job* e sua execução é análoga à execução de um método no paradigma orientado à objeto, porém, devido à biblioteca ter acesso aos recursos de execução do lado C++ do motor de jogo, esses *jobs* podem executar o seu código com um grau maior de eficiência e, caso sejam agendados desta maneira, podem ser executados de maneira paralela. Para tanto, a biblioteca oferece algumas interfaces que podem ser implementadas pelo usuário programador para criar seus próprios *jobs* e agenda-los durante a execução da aplicação (UNITY, 2023c):

- a) *IJob*, para executar uma tarefa de forma sequencial;
- b) *IJobParallelFor*, para executar uma tarefa de forma paralela;
- c) *IJobParallelForTransform*, para executar uma tarefa de forma paralela com acesso ao componente *Transform*, responsável por armazenar e manipular a localização espacial de *GameObjects*;

Figura 8 – Gerenciador de pacotes do motor de jogo



Fonte: Elaborada pelo autor.

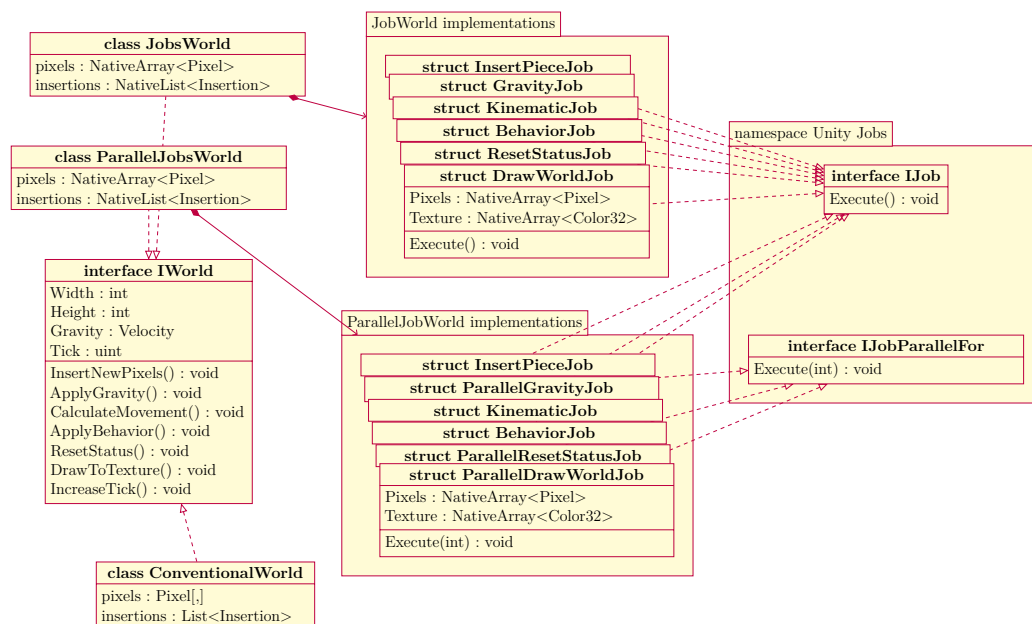
- d) **IJobFor**, para executar uma tarefa de forma paralela porém oferece a opção de agendar a execução sequencial de forma programática.

Para utilizar a ferramenta *Job System* no projeto é necessário instalar o pacote *Collections* utilizando o gerenciador de pacotes, como pode ser visto na Figura 8. Após a instalação torna-se necessário realizar modificações no código da simulação. Neste ponto, uma decisão foi tomada para que o código da implementação convencional fosse copiado e modificado de forma separada, no intuito de preservar a implementação de referência para os testes e permitir a criação de duas novas implementações, *Jobs* e *ParallelJobs*, como pode ser visto no diagrama de classes na Figura 9. As modificações realizadas se deram de tal maneira:

1. Foi criada uma interface **IWorld** para representar de forma genérica uma simulação de tipo *falling sand* expondo as etapas da simulação como métodos desta interface. Todas as implementações das simulações utilizam esta interface;
2. As etapas que foram modeladas como métodos na implementação convencional foram transformadas em estruturas que utilizam as interfaces da biblioteca *Job System*. Por exemplo, o método da etapa *ApplyBehavior* foi reimplementado como uma estrutura de nome **BehaviorJob**, com o método **Execute** da interface **IJob** realizando o processamento da etapa correspondente;
3. Substituição de estruturas de dados implementadas pelo motor de jogo pertencentes ao namespace **UnityEngine** por estruturas do namespace **Unity.Mathematics**. Por exemplo, os usos do tipo **Vector2Int** foram substituídos pelo tipo **int2**;

4. Substituição de métodos de funções matemáticas implementadas pelo motor de jogo pertencentes ao namespace `UnityEngine` por métodos do namespace `Unity.Mathematics`. Por exemplo, os usos do método `Mathf.Abs` foram substituídos pelo método `math.abs`;
5. Substituição de estruturas de tipo contêiner genérico implementados pela linguagem `C#` pertencentes ao namespace `System.Collections.Generic` por tipos pertencentes ao namespace `Unity.Collections`. Por exemplo, os usos do tipo `Array<T>` foram substituídos pelo tipo `NativeArray<T>`;
6. As etapas anteriores foram repetidas para a implementação da simulação *ParallelJobs* utilizando a interface `IJobParallelFor`. As etapas *ApplyGravity*, *ResetStatus* e *DrawToTexture* foram implantadas de maneira à paralelizar o processamento dos *pixels*, porém, o processo de implementação de algoritmo paralelo para as etapas *InsertNewPixels*, *CalculateMovement* e *ApplyBehavior* se mostrou não trivial e, portanto, o código da implementação sequencial *Jobs* foi reutilizado na implementação *ParallelJobs*.

Figura 9 – Diagrama de classes das diferentes implementações da simulação de tipo *falling sand*



Fonte: Elaborada pelo autor.

### 3.2.3 *Burst Compiler*

*Burst Compiler* é um compilador capaz de utilizar um subconjunto da linguagem C# para gerar código de máquina altamente otimizado. Foi criado, primariamente, para ser utilizado junto com a biblioteca *Job System* e os tipos contidos no namespace `Unity.Collections`. Para utilizar esta ferramenta é necessário adicionar o pacote *Burst* no gerenciador de pacotes como pode ser visto na Figura 8. A utilização desta ferramenta requer realizar modificações no código das simulações, porém essas modificações são triviais de serem feitas: é necessário apenas adicionar o atributo `[BurstCompile]` às estruturas as quais o usuário programador deseja que sejam compiladas utilizando a otimização *Burst*. As documentações do motor de jogo e da ferramenta (UNITY, 2023a) sugerem que as ferramentas *Job System* e *Burst Compiler* sejam utilizadas sempre em conjunto, portanto, as implementações *Jobs* e *ParallelJobs* fazem uso de ambas e estas não foram testadas separadamente.

## 3.3 Métricas e procedimento de obtenção de dados

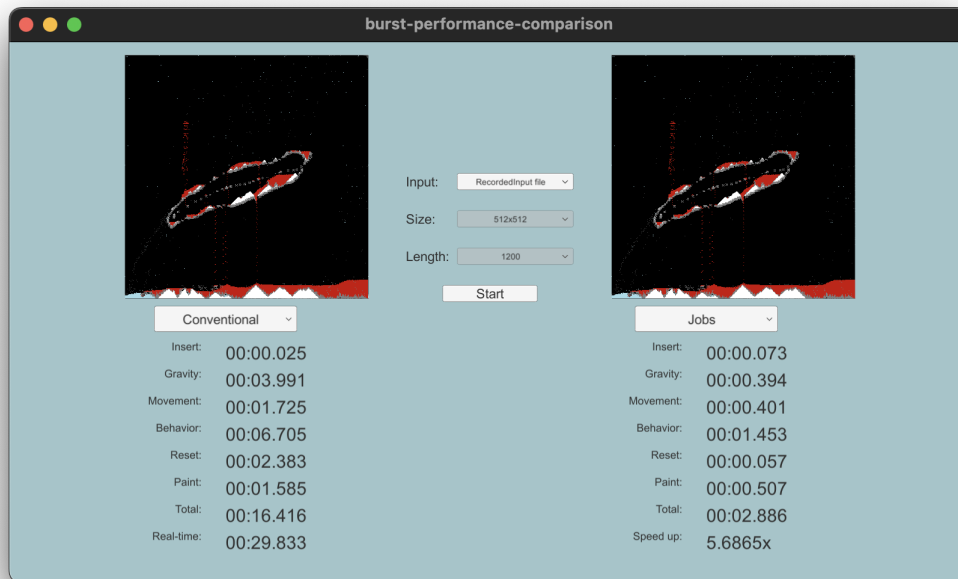
Para este trabalho, a métrica base para realização das análises foi o tempo de execução de cada etapa do turno de simulação. Para tanto, a abstração de cada implementação e a interface `IWorld` foram criadas para permitir que as etapas pudessem ser executadas de forma individual e sem realizar a renderização, com o intuito de que o tempo medido fosse exclusivamente o tempo de processamento da simulação.

Durante o processo de desenvolvimento da pesquisa foi criada uma ferramenta para avaliar, de forma ágil, os resultados parciais de comparação das simulações, como pode ser visto na Figura 10. Essa ferramenta permitiu executar duas simulações simultaneamente, lado a lado, e observar em tempo real o tempo consumido por cada etapa da simulação assim como identificar quaisquer discrepâncias que existissem no estado das simulações, que deveriam ser sempre iguais.

Seguindo recomendações realizadas por Borufka (2020), foi criado uma segunda ferramenta responsável por realizar a execução automatizada dos testes. Neste *script* foi possível configurar parte do procedimento de coleta de dados, cuja configuração final foi, como pode ser visto na Figura 11:

- a) As simulações foram testadas na seguinte ordem, primeiro *Conventional*, em seguida *Jobs* e por fim *ParallelJobs*;
- b) As simulações foram submetidas a uma entrada determinística gerada aleatoria-

Figura 10 – Ferramenta de teste de desempenho lado a lado



Fonte: Elaborada pelo autor.

mente<sup>7</sup> por 900 turnos;

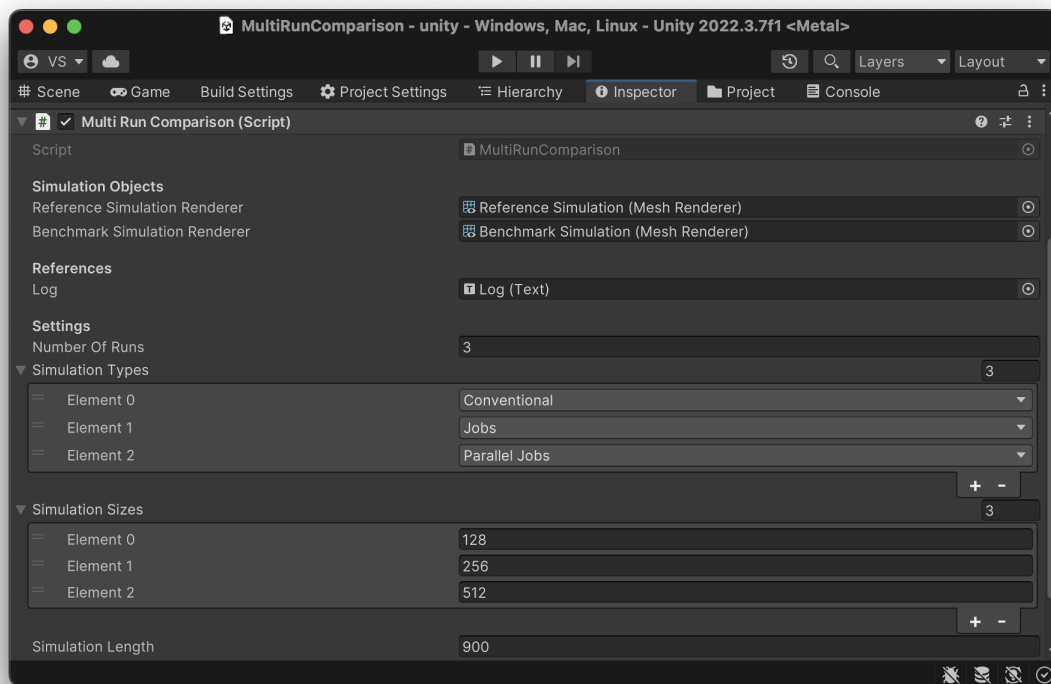
- c) As simulações foram executadas em três tamanhos de resolução diferentes, 128 por 128, 256 por 256 e 512 por 512;
- d) Os testes foram repetidos três vezes.

Para a obtenção dos dados finais foram seguidas recomendações para testes de desempenho realizadas por Akinshin (2019) e Borufka (2020) e alguns cuidados foram tomados para que os testes realizados fossem afetados o mínimo possível por elementos externos à simulação, contribuindo para a precisão dos resultados:

- a) O tempo de execução foi medido para cada etapa da simulação utilizando o temporizador de alta resolução da classe `System.Diagnostics.Stopwatch`;
- b) As estruturas de dados responsáveis por armazenar os resultados foram pré-alocadas de modo a evitar alocação de memória durante a execução dos testes;
- c) O coletor de lixo automatizado, responsável por gerenciar memória do lado C# do motor de jogo, foi invocado manualmente antes e depois de cada teste, com

<sup>7</sup> As resoluções de mesmo tamanho foram sempre submetidas às mesmas entradas, estas foram geradas aleatoriamente porém utilizando o mesmo valor de *seed* inicial.

Figura 11 – Script de configuração de teste automatizado



Fonte: Elaborada pelo autor.

o intuito de evitar que o processo de coleta de lixo ocorresse durante a execução dos testes;

- d) Um processo de *warm up* foi realizado antes da execução de cada teste, executando parte da simulação a ser testada, de forma a reduzir a influência de processos de compilação JIT que podem ocorrer do lado C# do motor de jogo;
- e) Todas as aplicações que estivessem em execução no computador foram fechadas antes da execução dos testes;
- f) Como os testes foram executados utilizando dois *notebooks*, ambos estavam com a bateria completamente carregada e conectados a energia elétrica durante a execução dos testes.

Foram gerados dois executáveis, um gerado utilizando o compilador *mono* e outro utilizando o compilador IL2CPP. Fazendo uma análise combinatória utilizando como variáveis o compilador e a implementação testada é possível gerar 6 combinações distintas para a realização dos testes, como pode ser visto na Tabela 2. Devido as implementações *Jobs* e *ParallelJobs* fazerem uso do mesmo esquema de compilação IL2CPP independente

Tabela 2 – Combinações de implementação de simulação e compilador utilizado.

|                     | <i>mono</i> | IL2CPP |
|---------------------|-------------|--------|
| <i>Conventional</i> | ✓           | ✓      |
| <i>Jobs</i>         | ✗           | ✓      |
| <i>ParallelJobs</i> | ✗           | ✓      |

Fonte: Elaborada pelo autor.

de qual *scripting backend* esteja selecionado nas configurações do projeto, os dados gerados pelos testes dessas implementações quando compilados com *scripting backend mono* não foram utilizadas na análise dos dados.

Finalmente, os testes foram executados utilizando dois computadores: um Macbook Air com processador de arquitetura Arm64, modelo Apple M1, sistema operacional MacOS Ventura, e um Acer com processador de arquitetura x86-64, modelo Intel i7-11800H, sistema operacional Windows 11.

## 4 Análise dos Resultados

Os dados obtidos para fundamentar esta pesquisa foram colhidos executando testes em três implementações de tipo *falling sand* distintas, utilizando três tamanhos diferentes de simulação, dois modos de compilação distintos e foram executados em dois computadores com arquiteturas de CPU distintas. Com a conclusão dos testes foi possível agregar uma massa de dados capaz de subsidiar comparativos e gerar visualizações, e, à partir destas informações, foi possível confirmar a hipótese principal deste trabalho: que o uso de ferramentas da iniciativa *Performance by Default*, no motor de jogo *Unity*, possibilitam uma utilização mais eficiente do recurso computacional numa simulação de tipo *falling sand*.

A seguir, são demonstradas as análises comparativas do desempenho das diferentes implementações:

- a) Análise comparativa do tempo total e do tempo médio por turno das implementações que utilizam as ferramentas da iniciativa *Performance by Default* em comparação com a implementação convencional;
- b) Análise comparativa do tempo médio por turno das etapas *ApplyGravity*, *ResetStatus*, *DrawWorld* entre as implementações *Jobs* e *ParallelJobs*.

Os dados que baseiam as análises estão dispostos em tabelas que contém informações sobre tamanho da simulação, em qual arquitetura de CPU o teste foi realizado, qual compilador foi utilizado e também qual a implementação da simulação foi testada. Cada linha das tabelas tem dados sobre o tempo total da execução da simulação (ou de uma etapa específica da simulação), o valor da média, mediana, erro padrão, desvio padrão e intervalo de confiança para o tempo de execução de um turno da simulação, e, por fim, o valor calculado de *speedup* para a implementação da linha.

Vale ressaltar que, de modo geral, as comparações foram feitas entre linhas que possuem o mesmo tamanho de simulação e tipo de CPU. Para fins de análise, as variáveis independentes são o tipo de compilador e de implementação.



## 4.1 Comparação com a implementação convencional

As implementações de tipo *Conventional*, *Jobs* e *ParallelJobs* geradas com o compilador IL2CPP, foram comparadas com a implementação *Conventional* gerada utilizando o compilador *mono*. Para auxiliar a análise foram criadas duas visualizações:

- a) Histograma do tempo de execução de simulação por turno, sendo o eixo horizontal representante do tempo em milissegundos e o eixo vertical representante do número de ocorrências<sup>1</sup>;
- b) Gráfico do tempo acumulado de execução da simulação, sendo que o eixo horizontal representa a quantidade de turnos simulados e o eixo vertical representa a quantidade de tempo acumulado em segundos. Para fins de comparação, foram plotados em pontilhado os valores referentes a simulações hipotéticas que executem exatamente em 30 e 60 turnos por segundo.

É importante frisar que os dados referentes a implementação *ParallelJobs* foram omitidos das visualizações pois seus valores estão em sua maioria sobrepostos com os valores da implementação *Jobs*. Os dados referentes a implementação *ParallelJobs* serão discutidos na seção 4.2.

Para todos os casos, as implementações que fazem uso de ferramentas da iniciativa *Performance by Default* registraram redução no valor total de tempo de execução da simulação, e, por consequência, registraram valores de *speedup*<sup>2</sup> acima de 1.

### 4.1.1 Simulações tamanho 128x128

Com base na Tabela 3, para as simulações com CPU x86-64, a utilização do compilador IL2CPP gerou uma redução no tempo de execução de 60.75%, a implementação *Jobs* gerou uma nova redução de 67.69%. A implementação *ParallelJobs* x86-64 não gerou melhoria de desempenho quando comparada com a implementação *Jobs* x86-64 para este tamanho de simulação.

Para as simulações com CPU Arm64, a utilização do compilador IL2CPP gerou uma redução no tempo de execução de 74.88%, a implementação *Jobs* gerou uma nova redução de 22.62%. A implementação *ParallelJobs* Arm64 também não gerou melhoria de desempenho quando comparada com a implementação *Jobs* Arm64 para este tamanho de simulação.

<sup>1</sup> Foram removidas ocorrências que estivessem a uma distância absoluta do valor da média superior em três desvios padrão.

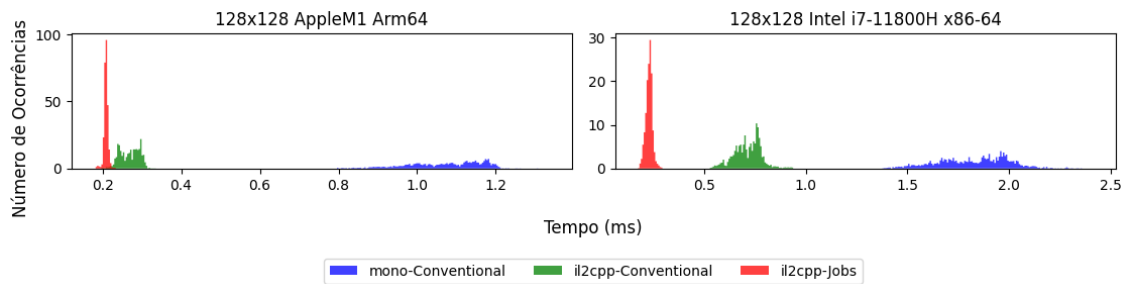
<sup>2</sup> Os valores calculados de *speedup* nas Tabelas 3, 4 e 5 têm como base de comparação a implementação convencional utilizando o compilador *mono* de mesmo tamanho de simulação e tipo de CPU.

Tabela 3 – Estatísticas para tempo total de simulação em milissegundos para implementações de tamanho 128x128

| tamanho | cpu    | compilador | implementação | total (ms) | média (ms) | mediana (ms) | erro padrão (ms) | desvio padrão (ms) | intervalo confiança (ms) | speedup |
|---------|--------|------------|---------------|------------|------------|--------------|------------------|--------------------|--------------------------|---------|
| 128x128 | x86-64 | mono       | Conventional  | 4945.0821  | 1.8315     | 1.8529       | 0.0036           | 0.1890             | (1.8184, 1.8446)         | 1.0000  |
| 128x128 | x86-64 | il2cpp     | Conventional  | 1940.8579  | 0.7188     | 0.7240       | 0.0015           | 0.0754             | (0.7136, 0.7241)         | 2.5479  |
| 128x128 | x86-64 | il2cpp     | Jobs          | 627.1306   | 0.2323     | 0.2321       | 0.0004           | 0.0214             | (0.2308, 0.2338)         | 7.8853  |
| 128x128 | x86-64 | il2cpp     | ParallelJobs  | 768.1684   | 0.2845     | 0.2851       | 0.0007           | 0.0340             | (0.2821, 0.2869)         | 6.4375  |
| 128x128 | Arm64  | mono       | Conventional  | 2894.1246  | 1.0719     | 1.0867       | 0.0018           | 0.0934             | (1.0654, 1.0784)         | 1.0000  |
| 128x128 | Arm64  | il2cpp     | Conventional  | 727.0259   | 0.2693     | 0.2712       | 0.0005           | 0.0246             | (0.2676, 0.2710)         | 3.9808  |
| 128x128 | Arm64  | il2cpp     | Jobs          | 562.5743   | 0.2084     | 0.2091       | 0.0002           | 0.0083             | (0.2078, 0.2089)         | 5.1444  |
| 128x128 | Arm64  | il2cpp     | ParallelJobs  | 613.9947   | 0.2274     | 0.2274       | 0.0002           | 0.0126             | (0.2265, 0.2283)         | 4.7136  |

Fonte: Dados da pesquisa.

A Figura 12 sugere que a implementação convencional gerada pelo compilador *mono* possui uma distribuição de característica platô e a implementação *Jobs* gerada pelo compilador IL2CPP possui uma distribuição de característica unimodal.

Figura 12 – Histograma do tempo total de simulação por turno em milissegundos para implementações *Conventional* e *Jobs* de tamanho 128x128

Fonte: Dados da pesquisa.

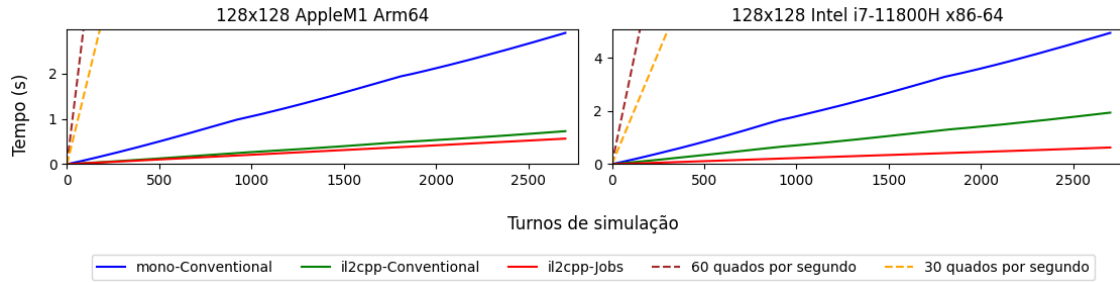
É possível observar nas Figuras 12 e 13 que todas as implementações para este tamanho de simulação possuem um valor médio de tempo de processamento de um turno confortavelmente abaixo do valor máximo para manter uma taxa de exibição de 60 quadros por segundo, ou seja, abaixo de  $16, \bar{6}$  milissegundos.

Na Figura 13, ao comparar o ângulo formado entre as linhas das implementações geradas utilizando o compilador IL2CPP (linhas verde e vermelha), entre os dois tipos de CPU, percebe-se que o ângulo formado para as implementações testadas na arquitetura x86-64 é maior. Isso indica que a melhoria gerada através da implementação das ferramentas *Job System* e *Burst Compiler* é mais acentuada para CPUs com arquitetura x86-64.

#### 4.1.2 Simulações tamanho 256x256

Com base na Tabela 4, para as simulações com CPU x86-64, a utilização do compilador IL2CPP gerou uma redução no tempo de execução de 59.66%, a implementação

Figura 13 – Tempo acumulado de simulação em milissegundos para implementações *Conventional* e *Jobs* de tamanho 128x128



Fonte: Dados da pesquisa.

*Jobs* gerou uma nova redução de 69.38%. A implementação *ParallelJobs* x86-64 não gerou melhoria de desempenho quando comparada com a implementação *Jobs* x86-64 para este tamanho de simulação.

Tabela 4 – Estatísticas para tempo total de simulação em milissegundos para implementações de tamanho 256x256

| tamanho | cpu    | compilador | implementação | total (ms) | média (ms) | mediana (ms) | erro padrão (ms) | desvio padrão (ms) | intervalo confiança (ms) | speedup |
|---------|--------|------------|---------------|------------|------------|--------------|------------------|--------------------|--------------------------|---------|
| 256x256 | x86-64 | mono       | Conventional  | 20365.9935 | 7.5430     | 7.5959       | 0.0148           | 0.7675             | (7.4898, 7.5961)         | 1.0000  |
| 256x256 | x86-64 | il2cpp     | Conventional  | 8216.1099  | 3.0430     | 3.0649       | 0.0059           | 0.3068             | (3.0217, 3.0643)         | 2.4788  |
| 256x256 | x86-64 | il2cpp     | Jobs          | 2516.2702  | 0.9320     | 0.9388       | 0.0016           | 0.0824             | (0.9262, 0.9377)         | 8.0937  |
| 256x256 | x86-64 | il2cpp     | ParallelJobs  | 2710.3906  | 1.0038     | 0.9834       | 0.0021           | 0.1114             | (0.9961, 1.0116)         | 7.5140  |
| 256x256 | Arm64  | mono       | Conventional  | 12148.2425 | 4.4993     | 4.5934       | 0.0080           | 0.4178             | (4.4704, 4.5283)         | 1.0000  |
| 256x256 | Arm64  | il2cpp     | Conventional  | 3361.2788  | 1.2449     | 1.2870       | 0.0027           | 0.1399             | (1.2352, 1.2546)         | 3.6142  |
| 256x256 | Arm64  | il2cpp     | Jobs          | 2431.4045  | 0.9005     | 0.9213       | 0.0013           | 0.0667             | (0.8959, 0.9051)         | 4.9964  |
| 256x256 | Arm64  | il2cpp     | ParallelJobs  | 2482.1823  | 0.9193     | 0.9373       | 0.0013           | 0.0684             | (0.9146, 0.9241)         | 4.8942  |

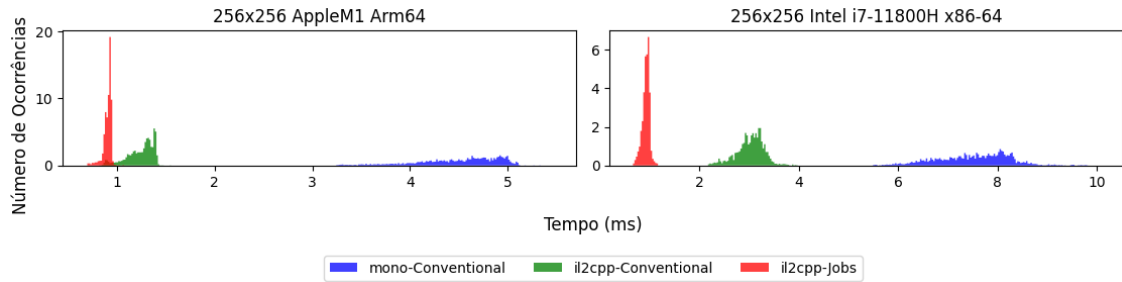
Fonte: Dados da pesquisa.

Para as simulações com CPU Arm64, a utilização do compilador IL2CPP gerou uma redução no tempo de execução de 72.33%, a implementação *Jobs* gerou uma nova redução de 27.66%. A implementação *ParallelJobs* Arm64 também não gerou melhoria de desempenho quando comparada com a implementação *Jobs* Arm64 para este tamanho de simulação.

A Figura 14 sugere que a implementação convencional gerada pelo compilador *mono* possui uma distribuição de característica platô e as implementações geradas com compilador IL2CPP possuem uma distribuição de característica unimodal.

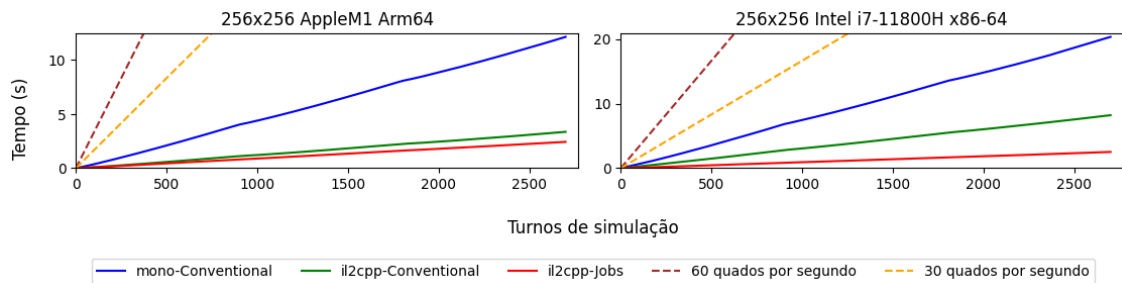
É possível observar nas Figuras 14 e 15 que as implementações convencionais demandam, em média, um percentual significativo no tempo de execução máximo de um turno para que se atinja a taxa de exibição de 60 quadros por segundo: 26.99% na arquitetura Arm64 e 45.26% na arquitetura x86-64. Já as implementações que fazem uso das ferramentas da iniciativa *Performance by Default* possuem um valor médio de tempo

Figura 14 – Histograma do tempo total de simulação por turno em milissegundos para implementações *Conventional* e *Jobs* de tamanho 256x256



Fonte: Dados da pesquisa.

Figura 15 – Tempo acumulado de execução em milissegundos para implementações *Conventional* e *Jobs* de tamanho 256x256



Fonte: Dados da pesquisa.

de processamento de um turno confortavelmente abaixo do valor máximo para manter uma taxa de exibição de 60 quadros por segundo.

Na Figura 15, ao comparar o ângulo formado entre as linhas das implementações geradas utilizando o compilador IL2CPP, entre os dois tipos de CPU, percebe-se que o ângulo formado para as implementações testadas na arquitetura x86-64 é maior. Análogo às simulações de tamanho 128x128, isso indica que a melhoria gerada através da implementação das ferramentas *Job System* e *Burst Compiler* é mais acentuada para CPUs com arquitetura x86-64.

### 4.1.3 Simulações tamanho 512x512

Com base na Tabela 5, para as simulações com CPU x86-64, a utilização do compilador IL2CPP gerou uma redução no tempo de execução de 57.29%, a implementação *Jobs* gerou uma nova redução de 71.96%. A implementação *ParallelJobs* x86-64 gerou redução de 1.35% no tempo de execução quando comparada com a implementação *Jobs* x86-64 para este tamanho de simulação.

Para as simulações com CPU Arm64, a utilização do compilador IL2CPP gerou uma redução no tempo de execução de 69.98%, a implementação *Jobs* gerou uma nova redução de 33.29%. A implementação *ParallelJobs* Arm64 gerou redução de 1.02% no tempo de execução quando comparada com a implementação *Jobs* Arm64 para este tamanho de simulação.

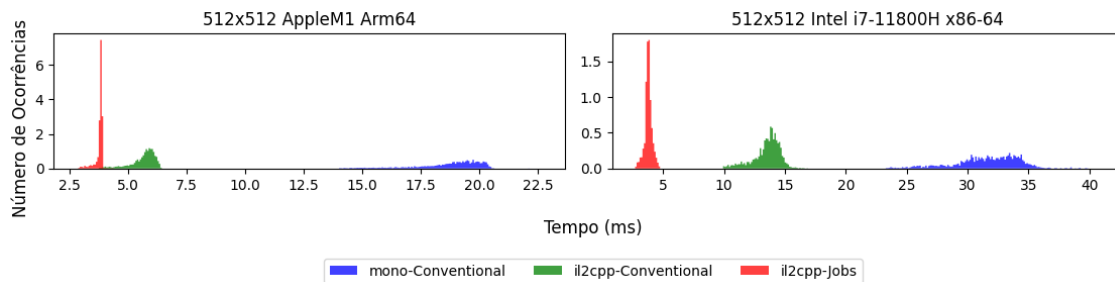
Tabela 5 – Estatísticas para tempo total de simulação em milissegundos para implementações de tamanho 512x512

| tamanho | cpu    | compilador | implementação | total (ms) | média (ms) | mediana (ms) | erro padrão (ms) | desvio padrão (ms) | intervalo confiança (ms) | speedup |
|---------|--------|------------|---------------|------------|------------|--------------|------------------|--------------------|--------------------------|---------|
| 512x512 | x86-64 | mono       | Conventional  | 85420.8265 | 31.6373    | 31.9443      | 0.0531           | 2.7576             | (31.4463, 31.8284)       | 1.0000  |
| 512x512 | x86-64 | il2cpp     | Conventional  | 36485.1503 | 13.5130    | 13.7616      | 0.0227           | 1.1817             | (13.4311, 13.5949)       | 2.3412  |
| 512x512 | x86-64 | il2cpp     | Jobs          | 10229.6470 | 3.7888     | 3.7912       | 0.0065           | 0.3371             | (3.7654, 3.8121)         | 8.3503  |
| 512x512 | x86-64 | il2cpp     | ParallelJobs  | 10091.1777 | 3.7375     | 3.7479       | 0.0056           | 0.2913             | (3.7173, 3.7577)         | 8.4649  |
| 512x512 | Arm64  | mono       | Conventional  | 50306.6568 | 18.6321    | 19.0739      | 0.0296           | 1.5383             | (18.5255, 18.7387)       | 1.0000  |
| 512x512 | Arm64  | il2cpp     | Conventional  | 15102.9098 | 5.5937     | 5.7755       | 0.0118           | 0.6117             | (5.5513, 5.6361)         | 3.3309  |
| 512x512 | Arm64  | il2cpp     | Jobs          | 10074.4567 | 3.7313     | 3.8145       | 0.0055           | 0.2876             | (3.7114, 3.7512)         | 4.9935  |
| 512x512 | Arm64  | il2cpp     | ParallelJobs  | 9971.2392  | 3.6931     | 3.7755       | 0.0054           | 0.2796             | (3.6737, 3.7124)         | 5.0452  |

Fonte: Dados da pesquisa.

De forma similar aos tamanhos de simulação anteriores, a Figura 16 sugere que a implementação convencional gerada pelo compilador *mono* possui uma distribuição de característica platô e as implementações geradas com compilador IL2CPP possuem uma distribuição de característica unimodal.

Figura 16 – Histograma do tempo total de simulação em milissegundos para implementações *Conventional* e *Jobs* de tamanho 512x512



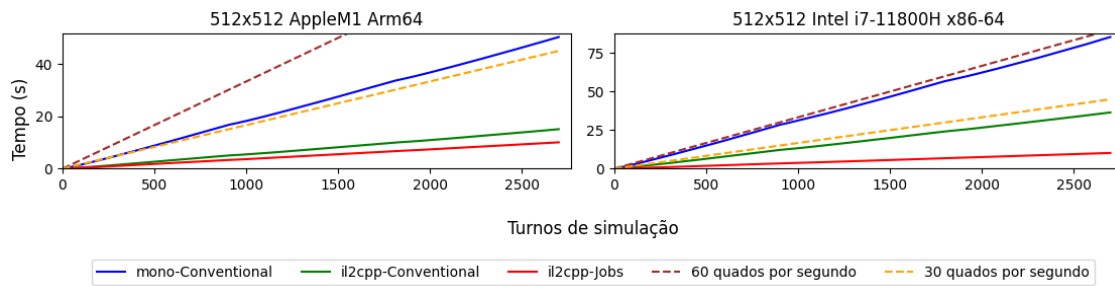
Fonte: Dados da pesquisa.

É possível observar nas Figuras 16 e 17 que as implementações convencionais, em ambas arquiteturas de CPU, são incapazes de manter uma taxa de exibição de 60 quadros por segundo. A implementação convencional gerada com compilador *mono* para arquitetura x86-64 sugere também ser incapaz de manter uma taxa de exibição de 30 quadros por segundo: esta implementação demanda, em média, 94.91% do valor máximo para manter uma taxa de exibição de 30 quadros por segundo, 33,3 milissegundos. Da mesma forma, a implementação convencional gerada com compilador IL2CPP para arquitetura x86-64

sugere ser incapaz de manter uma taxa de exibição de 60 quadros por segundo, demandando, em média, 81.08% do valor máximo para manter uma taxa de exibição de 60 quadros por segundo.

Para este tamanho de simulação, apenas as implementações *Jobs* e *ParallelJobs* possuem um valor médio de tempo de processamento de um turno capaz de manter uma taxa de exibição de 60 quadros por segundo.

Figura 17 – Tempo acumulado de simulação de simulação em milissegundos para implementações *Conventional* e *Jobs* de tamanho 512x512



Fonte: Dados da pesquisa.

De maneira análoga às simulações de tamanho 128x128 e 256x256, a Figura 17 indica que a melhoria gerada através da implementação das ferramentas *Job System* e *Burst Compiler* é mais acentuada para CPUs com arquitetura x86-64.

## 4.2 Comparação entre implementação de etapas sequenciais e paralelas

Nesta seção são apresentados as comparações de etapas específicas das implementações *Jobs* e *ParallelJobs* que fizeram uso das ferramentas da iniciativa *Performance by Default*, implementando o *Job System* e o *Burst Compiler*.

Como discutido na subseção 3.2.2 e representado na Figura 9, a implementação *ParallelJobs* paralelizou as etapas *ApplyGravity*, *ResetStatus* e *DrawToTexture* da simulação, reutilizando o algoritmo sequencial das etapas *InsertNewPixels*, *CalculateMovement*, *ApplyBehavior* da implementação *Jobs*. Logo, a comparação<sup>3</sup> que foi feita nesta seção se dá considerando apenas as etapas *ApplyGravity*, *ResetStatus* e *DrawToTexture*, das implementações *Jobs* e *ParallelJobs* e geradas com o compilador IL2CPP.

<sup>3</sup> Os valores calculados de speedup nas Tabelas 6, 7 e 8 têm como base de comparação a implementação *Jobs* utilizando o compilador IL2CPP de mesmo tamanho de simulação e tipo de CPU.

Tabela 6 – Estatísticas para tempo de simulação em milissegundos da etapa *ApplyGravity*, implementações *Jobs* e *ParallelJobs*

| tamanho | cpu    | compilador | implementação | gravity total<br>(ms) | média<br>(ms) | mediana<br>(ms) | erro padrão<br>(ms) | desvio padrão<br>(ms) | intervalo confiança<br>(ms) | speedup |
|---------|--------|------------|---------------|-----------------------|---------------|-----------------|---------------------|-----------------------|-----------------------------|---------|
| 128x128 | x86-64 | il2cpp     | Jobs          | 47.1779               | 0.0175        | 0.0167          | 0.0001              | 0.0034                | (0.0172, 0.0177)            | 1.0000  |
| 128x128 | x86-64 | il2cpp     | ParallelJobs  | 61.6767               | 0.0228        | 0.0205          | 0.0001              | 0.0064                | (0.0224, 0.0233)            | 0.7649  |
| 128x128 | Arm64  | il2cpp     | Jobs          | 34.1784               | 0.0127        | 0.0126          | 0.0000              | 0.0008                | (0.0126, 0.0127)            | 1.0000  |
| 128x128 | Arm64  | il2cpp     | ParallelJobs  | 46.9955               | 0.0174        | 0.0184          | 0.0001              | 0.0036                | (0.0172, 0.0177)            | 0.7273  |
| 256x256 | x86-64 | il2cpp     | Jobs          | 179.8895              | 0.0666        | 0.0664          | 0.0003              | 0.0173                | (0.0654, 0.0678)            | 1.0000  |
| 256x256 | x86-64 | il2cpp     | ParallelJobs  | 148.5027              | 0.0550        | 0.0538          | 0.0004              | 0.0211                | (0.0535, 0.0565)            | 1.2114  |
| 256x256 | Arm64  | il2cpp     | Jobs          | 141.2932              | 0.0523        | 0.0526          | 0.0000              | 0.0019                | (0.0522, 0.0525)            | 1.0000  |
| 256x256 | Arm64  | il2cpp     | ParallelJobs  | 119.7396              | 0.0443        | 0.0448          | 0.0001              | 0.0033                | (0.0441, 0.0446)            | 1.1800  |
| 512x512 | x86-64 | il2cpp     | Jobs          | 735.7730              | 0.2725        | 0.2505          | 0.0014              | 0.0702                | (0.2676, 0.2774)            | 1.0000  |
| 512x512 | x86-64 | il2cpp     | ParallelJobs  | 425.1375              | 0.1575        | 0.1540          | 0.0007              | 0.0345                | (0.1551, 0.1598)            | 1.7307  |
| 512x512 | Arm64  | il2cpp     | Jobs          | 566.2474              | 0.2097        | 0.2111          | 0.0002              | 0.0097                | (0.2091, 0.2104)            | 1.0000  |
| 512x512 | Arm64  | il2cpp     | ParallelJobs  | 242.0473              | 0.0896        | 0.0891          | 0.0002              | 0.0088                | (0.0890, 0.0903)            | 2.3394  |

Fonte: Dados da pesquisa.

Tabela 7 – Estatísticas para tempo de simulação em milissegundos da etapa *ResetStatus*, implementações *Jobs* e *ParallelJobs*

| tamanho | cpu    | compilador | implementação | reset total<br>(ms) | média<br>(ms) | mediana<br>(ms) | erro padrão<br>(ms) | desvio padrão<br>(ms) | intervalo confiança<br>(ms) | speedup |
|---------|--------|------------|---------------|---------------------|---------------|-----------------|---------------------|-----------------------|-----------------------------|---------|
| 128x128 | x86-64 | il2cpp     | Jobs          | 18.4096             | 0.0068        | 0.0065          | 0.0000              | 0.0022                | (0.0067, 0.0070)            | 1.0000  |
| 128x128 | x86-64 | il2cpp     | ParallelJobs  | 60.3633             | 0.0224        | 0.0209          | 0.0001              | 0.0066                | (0.0219, 0.0228)            | 0.3050  |
| 128x128 | Arm64  | il2cpp     | Jobs          | 10.2600             | 0.0038        | 0.0037          | 0.0000              | 0.0006                | (0.0038, 0.0038)            | 1.0000  |
| 128x128 | Arm64  | il2cpp     | ParallelJobs  | 34.9719             | 0.0130        | 0.0130          | 0.0000              | 0.0022                | (0.0128, 0.0131)            | 0.2934  |
| 256x256 | x86-64 | il2cpp     | Jobs          | 54.7303             | 0.0203        | 0.0197          | 0.0001              | 0.0033                | (0.0200, 0.0205)            | 1.0000  |
| 256x256 | x86-64 | il2cpp     | ParallelJobs  | 97.7427             | 0.0362        | 0.0339          | 0.0002              | 0.0101                | (0.0355, 0.0369)            | 0.5599  |
| 256x256 | Arm64  | il2cpp     | Jobs          | 34.9703             | 0.0130        | 0.0131          | 0.0000              | 0.0011                | (0.0129, 0.0130)            | 1.0000  |
| 256x256 | Arm64  | il2cpp     | ParallelJobs  | 79.1091             | 0.0293        | 0.0297          | 0.0000              | 0.0024                | (0.0291, 0.0295)            | 0.4421  |
| 512x512 | x86-64 | il2cpp     | Jobs          | 191.4468            | 0.0709        | 0.0688          | 0.0004              | 0.0183                | (0.0696, 0.0722)            | 1.0000  |
| 512x512 | x86-64 | il2cpp     | ParallelJobs  | 228.4324            | 0.0846        | 0.0793          | 0.0005              | 0.0238                | (0.0830, 0.0863)            | 0.8381  |
| 512x512 | Arm64  | il2cpp     | Jobs          | 131.9323            | 0.0489        | 0.0511          | 0.0001              | 0.0038                | (0.0486, 0.0491)            | 1.0000  |
| 512x512 | Arm64  | il2cpp     | ParallelJobs  | 158.1718            | 0.0586        | 0.0582          | 0.0002              | 0.0082                | (0.0580, 0.0592)            | 0.8341  |

Fonte: Dados da pesquisa.

Os resultados expostos nas Tabelas 6, 7 e 8 indicam que o desempenho das etapas paralelizadas tendem a melhorar quando executados em simulações de maior tamanho porém, na maioria dos casos testados não trouxeram redução no tempo de execução quando comparados com a implementação sequencial das etapas, representadas pela implementação *Jobs*. Como pode ser visto na Tabela 6, ganhos significativos são observados apenas na etapa *ApplyGravity* testada na resolução 512x512, com as arquiteturas x86-64 e Arm64 atingindo uma redução de tempo de execução, mostrado na coluna “gravity total”, de 42.21% e 57.25% respectivamente.

Tabela 8 – Estatísticas para tempo de simulação em milissegundos da etapa *DrawToTexture*, implementações *Jobs* e *ParallelJobs*

| tamanho | cpu    | compilador | implementação | draw total<br>(ms) | média<br>(ms) | mediana<br>(ms) | erro padrão<br>(ms) | desvio padrão<br>(ms) | intervalo confiança<br>(ms) | speedup |
|---------|--------|------------|---------------|--------------------|---------------|-----------------|---------------------|-----------------------|-----------------------------|---------|
| 128x128 | x86-64 | il2cpp     | Jobs          | 111.4462           | 0.0413        | 0.0391          | 0.0002              | 0.0083                | (0.0407, 0.0418)            | 1.0000  |
| 128x128 | x86-64 | il2cpp     | ParallelJobs  | 149.2904           | 0.0553        | 0.0498          | 0.0002              | 0.0126                | (0.0544, 0.0562)            | 0.7465  |
| 128x128 | Arm64  | il2cpp     | Jobs          | 64.2088            | 0.0238        | 0.0236          | 0.0000              | 0.0010                | (0.0237, 0.0238)            | 1.0000  |
| 128x128 | Arm64  | il2cpp     | ParallelJobs  | 73.0097            | 0.0270        | 0.0268          | 0.0000              | 0.0018                | (0.0269, 0.0272)            | 0.8795  |
| 256x256 | x86-64 | il2cpp     | Jobs          | 419.9097           | 0.1555        | 0.1577          | 0.0003              | 0.0175                | (0.1543, 0.1567)            | 1.0000  |
| 256x256 | x86-64 | il2cpp     | ParallelJobs  | 407.2772           | 0.1508        | 0.1457          | 0.0003              | 0.0156                | (0.1498, 0.1519)            | 1.0310  |
| 256x256 | Arm64  | il2cpp     | Jobs          | 235.0183           | 0.0870        | 0.0867          | 0.0000              | 0.0016                | (0.0869, 0.0872)            | 1.0000  |
| 256x256 | Arm64  | il2cpp     | ParallelJobs  | 248.1779           | 0.0919        | 0.0915          | 0.0001              | 0.0028                | (0.0917, 0.0921)            | 0.9470  |
| 512x512 | x86-64 | il2cpp     | Jobs          | 1532.2999          | 0.5675        | 0.5587          | 0.0009              | 0.0466                | (0.5643, 0.5708)            | 1.0000  |
| 512x512 | x86-64 | il2cpp     | ParallelJobs  | 1493.4577          | 0.5531        | 0.5407          | 0.0007              | 0.0367                | (0.5506, 0.5557)            | 1.0260  |
| 512x512 | Arm64  | il2cpp     | Jobs          | 897.2030           | 0.3323        | 0.3318          | 0.0001              | 0.0040                | (0.3320, 0.3326)            | 1.0000  |
| 512x512 | Arm64  | il2cpp     | ParallelJobs  | 963.4560           | 0.3568        | 0.3549          | 0.0003              | 0.0139                | (0.3559, 0.3578)            | 0.9312  |

Fonte: Dados da pesquisa.



## 5 Conclusão

O jogo digital é uma categoria de *software* capaz de consumir grandes quantidades de recursos computacionais e acompanha de perto a vanguarda dos avanços tecnológicos realizados na fabricação de semicondutores. Essa característica configurou o processo de desenvolvimento de jogos digitais como uma atividade sensível às mudanças advindas das transformações de *hardware* ocorridas a longo do século XXI. Uma dessas mudanças foi a adoção de modelos paralelos de processamento, como nos modelos SMP e SIMD, criando um cenário em que, com o propósito de fazer melhor uso do recurso computacional, se tornou necessário adotar o paradigma de programação paralela.

Seguindo esta tendência, em 2018, a empresa *Unity Technologies*, responsável pelo motor de jogo *Unity*, adotou publicamente uma iniciativa de nome *Performance by Default*. Esta iniciativa tem o intuito de possibilitar ao usuário programador do motor de jogo fazer melhor uso do recurso computacional, tornando disponíveis ferramentas que trabalham num nível de abstração mais baixo do que o paradigma de programação tradicional do motor de jogo *Unity* e que tornam mais acessível a adoção do paradigma de programação paralela no contexto da produção de um jogo digital. Uma utilização eficiente do recurso computacional pode ser relacionada a melhorias na experiência do usuário jogador, visível em métricas que podem ser quantificadas como, por exemplo, uma redução no consumo de energia para um jogo *mobile* (estendendo a duração da bateria do dispositivo, reduzindo a emissão de calor), um aumento na quantidade de elementos interativos no cenário do jogo, aumento da resolução do jogo ou da quantidade de quadros por segundo que o jogo é capaz de exibir.

Neste contexto, a propósito de avaliar o impacto que o uso destas ferramentas traz no uso do recurso computacional, foi implementada uma simulação determinística de tipo *falling sand* utilizando a abordagem de programação convencional dentro do motor de jogo *Unity*. Essa implementação foi utilizada como referência para implementação de versões da simulação que fazem uso das ferramentas da iniciativa *Performance by Default*: *IL2CPP*; *Job System* e *Burst Compiler*. A partir da criação de uma infraestrutura para realização de testes de desempenho automatizados e de scripts para análise estatística, foram colhidos dados de tempo de execução das diferentes implementações, realizando o experimento com três tamanhos diferentes de simulação, dois modos de compilação distintos e em duas arquiteturas de CPU distintas.

Após realização dos testes verificou-se que todas as implementações que fizeram uso de ferramentas da iniciativa trouxeram ganhos de desempenho quando comparadas

com uma implementação utilizando a abordagem de programação convencional do motor de jogo *Unity* e, diferentemente da implementação convencional, se mantiveram com o tempo médio de processamento de um turno de simulação abaixo do valor necessário para manter uma taxa de 60 quadros por segundo, 16,  $\bar{6}$  milissegundos.

A análise dos dados permitiu formular algumas conclusões: houveram ganhos muito significativos com a utilização das ferramentas propostas. Nos testes realizados na arquitetura x86-64 foi possível observar valores de *speedup* de até 8.46<sup>1</sup>, significando que esta implementação foi quase 8,5 vezes mais rápida do que a implementação convencional para a mesma arquitetura de processador. Já nos testes realizados na arquitetura Arm64 foram atingidos valores de *speedup* de até 5.14<sup>2</sup> em comparação com a implementação convencional para a arquitetura Arm64; uma segunda análise permitiu concluir também que a utilização da abordagem de programação paralela, representada nas etapas *ApplyGravity*, *ResetStatus* e *DrawToTexture* (vide Figura 9) da implementação *ParallelJobs*, necessita de uma carga de trabalho maior do que a fornecida nos testes realizados para se tornar mais eficiente do que a abordagem sequencial, seja trabalhando uma simulação de resolução maior ou realizando uma quantidade maior de trabalho por elemento processado; com a implementação apenas da ferramenta IL2CPP é possível atingir valores significativos de *speedup* de até 3.98<sup>3</sup> sem necessitar de realizar mudanças no código da simulação, apenas mudando uma configuração do projeto; o uso das ferramentas *Job System* e *Burst Compiler*, presente nas implementações *Jobs* e *ParallelJobs*, traz ganhos comparativamente maiores para a arquitetura x86-64 em relação aos ganhos atingidos na arquitetura Arm64. Para uma aplicação que tenha como alvo exclusivo processadores Arm64, a maior parte dos ganhos podem ser obtidos usando apenas a ferramenta IL2CPP.

Os ganhos observados fornecem insumos para afirmar que o uso das ferramentas IL2CPP, *Job System* e *Burst Compiler* são capazes de reduzir sensivelmente o tempo de processamento utilizado pela simulação para todos os tamanhos de simulação e arquiteturas de CPU testadas<sup>4</sup>. Esta redução, quando aplicada no contexto de desenvolvimento de um jogo, cria oportunidades para que o tempo de processamento economizado seja utilizado para melhorar a experiência do usuário jogador, seja criando uma experiência com maior grau de interatividade, seja por oferecer uma experiência de jogo mais fluída com uma quantidade maior de quadros por segundo exibidos em tela, ou mesmo por reduzir o consumo de energia da aplicação, validando a afirmação inicial de que uso dessas ferramentas é

<sup>1</sup> CPU x86-64, tamanho 512x512, implementação *ParallelJobs* gerada com compilador IL2CPP.

<sup>2</sup> CPU Arm64, tamanho 128x128, implementação *Jobs* gerada com compilador IL2CPP.

<sup>3</sup> CPU Arm64, tamanho 128x128, implementação convencional gerada com compilador IL2CPP.

<sup>4</sup> O projeto *Unity* com as implementações da simulação de tipo *falling sand*, os resultados dos testes realizados e os scripts de análise de dados podem ser encontrados no repositório disponível em <<https://github.com/vimsos/unity-pbd-comparison>>.

benéfico no processo de desenvolvimento de jogos digitais.

Em uma possível continuação deste processo de pesquisa sugere-se aprofundar a implementação do paradigma de programação paralela: utilizando técnicas de particionamento de espaço pode ser possível paralelizar todas as etapas da simulação, como sugerido por Purho (2019a) em sua palestra “*Exploring the Tech and Design of Noita*” no evento GDC de 2019.

# Referências

- AKINSHIN, A. **Pro .NET Benchmarking**. 1. ed. [S.l.]: Apress, 2019. 690 p. ISBN 978-1-4842-4940-6.
- ALVES, R. V. L.  
**Análise de Desempenho do Algoritmo de *Flocking* em uma Implementação Baseada no Design Orientado a Dados** — Universidade do Estado da Bahia, Salvador, 2020.
- BITTKER, M. **Making Sandspiel**. 2019. Disponível em: <<https://maxbittker.com/making-sandspiel>>. Acesso em: 2 set. 2021.
- BORUFKA, R. **Performance Testing Suite for Unity DOTS**. Dissertação (Mestrado) — Charles University Faculty of Mathematics and Physics, Prague, 2020.
- CHAMBERS, C. Staged compilation. In: **PEPM '02: Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation**. [S.l.: s.n.], 2002. v. 37, p. 1–8.
- GIL, A. C. **Como Elaborar Projetos de Pesquisa**. 4. ed. São Paulo: Atlas, 2002.
- GREGORY, J. **Game Engine Architecture**. 3. ed. New York: CRC Press, 2018.
- JUUL, J. **Half-Real: Video Games between Real Rules and Fictional Worlds**. Massachusetts: MIT Press, 2005. ISBN 9780262101103.
- KIRK, D.; HWU, W. **Programming Massively Parallel Processors: A Hands-on Approach: Third Edition**. [S.l.]: Elsevier Inc., 2016. ISBN 9780128119860.
- KOSTER, R. **A Theory of Fun for Game Design**. 2. ed. [S.l.]: O'Reilly, 2014. 244 p.
- KROGH-JACOBSEN, T. **2018.1 is now available**. 2018. Disponível em: <<https://blog.unity.com/technology/2018-1-is-now-available>>. Acesso em: 02 set. 2021.
- MCCULLOUGH, K. et al. Exploring Game Industry Technological Solutions to Simulate Large-Scale Autonomous Entities within a Virtual Battlespace. In: **Proceedings of the I/ITSEC 2019 Conference**. Orlando, FL: [s.n.], 2019. p. 12. Disponível em: <<https://www.xcdsystem.com/iitsec/proceedings/index.cfm?Year=2019&AbID=28302&CID=48>>.
- MEIRELLES, F. S. **Uso da TI nas Empresas - Panorama e Indicadores**. 2021. Disponível em: <<https://eaesp.fgv.br/sites/eaesp.fgv.br/files/u68/fgvcia2021pesti-relatorio.pdf>>. Acesso em: 19 set. 2021.
- MICROSOFT. **Managed Execution Process**. [S.l.], 2021. Disponível em: <[https://learn.microsoft.com/en-us/dotnet/standard/managed-execution-process#compiling\\_msil\\_to\\_native\\_code](https://learn.microsoft.com/en-us/dotnet/standard/managed-execution-process#compiling_msil_to_native_code)>. Acesso em: 15 dec. 2023.

- MUTEL, A. **Porting the Unity Engine to .NET CoreCLR**. 2018. Disponível em: <<https://xoofx.com/blog/2018/04/06/porting-unity-to-coreclr/>>. Acesso em: 31 oct. 2021.
- NEWZOO. **Newzoo's Generations Report: How Different Generations Engage with Games**. 2021. Disponível em: <<https://newzoo.com/insights/trend-reports/newzoos-generations-report-how-different-generations-engage-with-games/>>. Acesso em: 20 set. 2021.
- NOLLA GAMES. **Noita - a roguelike in which every pixel is simulated**. 2020. Disponível em: <<http://noitagame.com>>. Acesso em: 17 oct. 2021.
- PETERSON, J. **An introduction to IL2CPP internals**. 2015. Disponível em: <<https://blog.unity.com/engine-platform/an-introduction-to-ilcpp-internals>>. Acesso em: 28 nov. 2023.
- PURHO, P. **Exploring the Tech and Design of Noita**. 2019. Disponível em: <<https://www.youtube.com/watch?v=prXuyMCgbTc>>. Acesso em: 30 nov. 2023.
- PURHO, P. **Noita: a Game Based on Falling Sand Simulation**. 2019. Disponível em: <<https://80.lv/articles/noita-a-game-based-on-falling-sand-simulation/>>. Acesso em: 17 oct. 2021.
- SALEN, K.; ZIMMERMAN, E. **Regras do jogo: fundamentos do design de jogos**. São Paulo: Blucher, 2012. v. 1: Principais Conceitos. ISBN 9788521206262.
- SILVA, F. S. d.; GUERRA, R. A. T. et al. **Cadernos CB Virtual 2**. 2. ed. João Pessoa: Ed. Universitária UFPB, 2011. ISBN 978-85-7745-902-5.
- UNITY TECHNOLOGIES. **Unity architecture**. [S.l.], 2020. Disponível em: <<https://docs.unity3d.com/Manual/unity-architecture.html>>. Acesso em: 28 oct. 2021.
- UNITY TECHNOLOGIES. **Unity User Manual 2020.3**. [S.l.], 2020. Disponível em: <<https://docs.unity3d.com/Manual/index.html>>. Acesso em: 28 oct. 2021.
- UNITY TECHNOLOGIES. **About Burst**. [S.l.], 2023. Disponível em: <<https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html>>. Acesso em: 28 nov. 2023.
- UNITY TECHNOLOGIES. **Entities Package Changelog**. [S.l.], 2023. Disponível em: <<https://docs.unity3d.com/Packages/com.unity.entities@1.2/changelog/CHANGELOG.html#108---2023-04-17>>. Acesso em: 27 nov. 2023.
- UNITY TECHNOLOGIES. **Job system overview**. [S.l.], 2023. Disponível em: <<https://docs.unity3d.com/2022.3/Documentation/Manual/JobSystemOverview.html>>. Acesso em: 28 nov. 2023.
- WOLFRAM, R. **Cellular Automata**. Los Alamos National Laboratory, 1983. Disponível em: <<https://la-science.lanl.gov/lascience09.shtml>>. Acesso em: 21 oct. 2021.