Análise dos ganhos de performance na utilização do padrão Entity Component Systems contra implementações orientadas a objetos

Yuri Moraes Gavilan[1]
Bianca de Almeida Dantas[1]

[1]Faculdade de Computação – Universidade Federal do Mato Grosso do Sul (UFMS)

yuri.moraes.gavilan@ufms.com, bianca.dantas@ufms.br

Abstract. *This paper describes the performance gains of the Entity Component Systems pattern in electronic games and general real-time simulations. In this paper we will delve into the motivations behind the implementations, the hardware architecture that allows for this programming paradigm and further considerations on the flexibility and performance gains by component-based software.*

Resumo. *Este trabalho descreve os ganhos de performance na utilização do padrão Entity Component Systems em jogos e simulações em tempo-real gerais. Nele serão discutidos os motivos dos ganhos de performance, a arquitetura do hardware que permitiram as decisões das implementações e demais considerações a respeito dos ganhos de flexibilidade e performance de softwares baseados em componentes.*

1. Introdução

No desenvolvimento de jogos eletrônicos, um problema recorrente é o da estruturação dos objetos de jogo que serão operados no processo de simulação do mundo virtual. Na simulação de um mundo de jogo as entidades são muito diversas, o que gera alguns problemas aos desenvolvedores: entidades mesmo que consideravelmente semelhantes potencialmente necessitarão serem processadas em caminhos de código diferentes; implementações muito específicas podem enrijecer o código e dificultar manutenção, etc. Inicialmente, o paradigma de programação orientada a objetos se mostrou como uma solução óbvia a esse problema arquitetural, entretanto, diversos problemas surgiram, como hierarquias demasiadamente complexas, baixa flexibilidade dos objetos instanciados, projetos desnecessariamente grandes, códigos duplicados e, além de tudo, perda de performance considerável. Muitos desses problemas foram corrigidos até certo ponto ao empregar boas práticas na modelagem das classes e técnicas como o polimorfismo, além da empregar classes baseadas em componentes ao invés de hierarquias baseadas em herança [Nystrom 2011]. Entretanto, mesmo com as diversas técnicas empregadas, o problema de performance permaneceu.

Nos últimos anos, programadores e demais especialistas no desenvolvimento de jogos eletrônicos e sistemas de alta performance buscaram formas de melhorar o desempenho desses programas com diversas técnicas e paradigmas [Kirmse 2002]. Um desses

paradigmas se popularizou entre os programadores de motores de jogos, chamado *Data Oriented Design*, ou, design orientado a dados, que se baseia em desacoplar os dados da lógica de forma organizada e normalizada [Fabian 2018], havendo *tradeoff* de maior flexibilidade e performance pela relativa perda de abstração e modelagem seguindo o mundo real. A partir disso, surgiu o padrão *Entity Component Systems* (ECS), que descarta a implementação dos objetos de jogo via classes e criam componentes que são estruturas de dados puras, e sistemas, que operam sobre conjuntos densos de componentes.

Neste trabalho será demonstrada a motivação das decisões de projeto dos algoritmos, como as estruturas de dados são associadas ao hardware através da implementação e execução de simulações e jogos simples, criados inicialmente com o paradigma orientado a objetos, e posteriormente convertidos com a arquitetura ECS. Todos os jogos implementarão exatamente as mesmas simulações, de forma a demonstrar os ganhos de performance unicamente pela diferença arquitetural de como os dados são dispostos na memória e pelo desacoplamento de lógica e dados.

2. O Hardware Moderno

Ao projetar softwares que demandam boa performance, é imprescindível entender o funcionamento do hardware em que ele será executado – para o caso do ECS, esta seção explicará apenas os detalhes pertinentes no processador e na memória RAM principal.

Por muito tempo na história dos computadores, os processadores e memórias RAM (*Random Access Memory* ou Memória de Acesso Aleatório) se mantiveram próximos em questões da latência do acesso de dados, isso mudou na década de 90, onde os manufaturadores de processadores começaram a aumentar drasticamente a frequência do *clock* dos processadores. Graças a isso, a discrepância entre as CPUs (*Central Processing Units* ou Unidades Centrais de Processamento) e o barramento de memória cresceu intensamente, gerando um gargalo de performance.

Entretanto, não era viável produzir chips de memória SRAM (*Static Random Access Memory* ou Memórias Estáticas de Acesso Aleatório) com frequências maiores, pois isso aumentaria os custos exageradamente [Drepper 2007]. Para corrigir este problema, as manufaturadoras de hardware adicionaram às *CPUs* as chamadas memórias *cache* – pequenas *SRAM* adicionadas ao processador, partindo do princípio da alta probabilidade de acesso sequencial de dados. Essas memórias são pequenas mas de altíssima velocidade, sendo organizadas em diferentes níveis, de acordo com a sua capacidade de armazenamento e performance (Figura 1).

As memórias *RAM* de grande capacidade, por outro lado, são *DRAM* (Dynamic Random Acess Memory, ou, memória de acesso aleatório dinâmica), que necessita de apenas um transistor e um capacitor para cada célula. Por usar um capacitor, há a necessidade de recarregar as células de memória, o que diminui a frequência de acesso aos dados e, consequentemente, a velocidade das computações [Drepper 2007].

Os processadores modernos costumam possuir três níveis de cache: L1, L2 e L3, sendo a L1 e L2 independentes entre núcleos, e a L3 compartilhada entre todos os núcleos. Existe também a separação entre a cache de dados e de instruções, a primeira armazena informações que serão utilizadas nas operações e a segunda armazena as instruções de máquina.

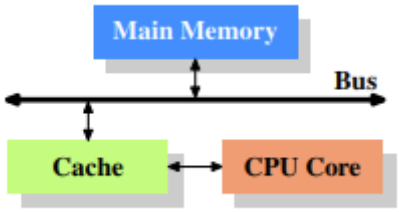

Figura 1. Configuração mínima da hierarquia de memórias [Drepper 2007]

As memórias cache *SRAM* são controladas integralmente pelo hardware (controlador da cache), utilizando métodos previsíveis e bem definidos: ele busca informações recentemente acessadas e faz cópias temporárias a partir dos princípios da localidade temporal e espacial, significando que dados e instruções são altamente prováveis de serem acessados de forma contínua e repetida. Além disso, o processador também busca, da memória, um bloco de tamanho fixo de informações próximas ao dado recentemente acessado, chamado de *cache line*, ou linha de cache. Processadores comuns ao público, como os da Intel e AMD, possuem, no geral, linhas de cache de 64 bytes, enquanto processadores recentes fabricados pela Apple, como o M1, trabalham com linhas de cache de 128 bytes.

Nestes processadores modernos, a busca de algum dado é feita da seguinte forma: a busca é feita inicialmente nos níveis mais altos de cache, caso a informação seja encontrada temos um *cache hit* e a operação é feita com sucesso. Caso contrário, temos um *cache miss* e se inicia a procura em níveis mais baixos, acessando a memória principal apenas em última instância ($L1 \rightarrow L2 \rightarrow L3 \rightarrow$ Memória principal).

Graças ao comportamento do processador na execução de instruções no acesso à memória, fica visível a seguinte implicação: a busca de dados da memória DRAM principal é extremamente lenta quando comparada às memórias cache SRAM. Desta forma, programas que não são arquitetados de forma a se beneficiarem dos princípios de localidade espacial e temporal, desperdiçam boa parte do tempo do processamento a esperar a busca dos dados da memória principal, o que é muito negativo à performance do programa. Por outro lado, programas que acessam os dados sequencialmente, são extremamente rápidos e possuem baixíssimas taxas de *cache miss*, por irem de acordo com o princípio da localidade espacial das memórias cache.

2.1. Implicações da arquitetura na performance de programas

Para debater as implicações da latência do acesso às memórias, é necessário inicialmente definir valores relativos ao custo da busca de dados em diferentes níveis de memória. Estes custos estão descritos na Tabela 1.

Portanto, fica claro que o custo de *cache misses* pode acarretar em execuções de programas dezenas ou centenas de vezes mais lentas. Para que isso seja testado empiricamente, pode-se realizar um simples experimento de latência de memória: o acesso de matrizes nos padrões *row-major* e *column-major*.

O acesso de uma matriz no padrão *row-major* se baseia em percorrer seus elementos linha por linha, enquanto a maneira *column-major* percorre os dados coluna por

Acesso	Ciclos
Registradores	≤ 1
L1	~ 3
L2	~ 14
Memória principal	~ 240

Tabela 1. Custos estimados de acesso às memórias [Drepper 2007].

coluna. Analisando visualmente a Figura 2, percebe-se que a ordem de acesso *row-major* é mais amigável ao princípio da localidade espacial, visto que os dados são acessados sequencialmente.

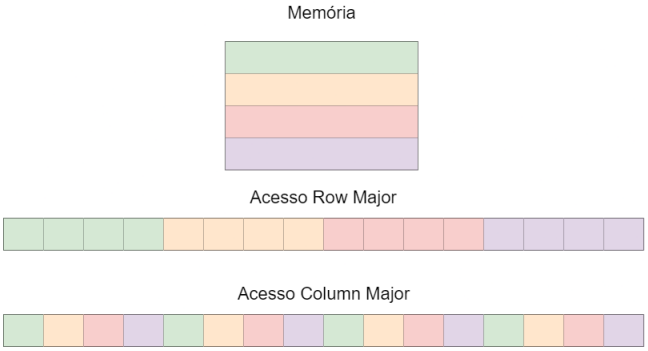

Figura 2. Diferenças entre os acessos *row-major* e *column-major*.

Para testar isso de forma prática, pode-se escrever um simples programa em C++, que realiza uma operação de soma e armazena o resultado em posições da matriz. Entretanto, se o percurso da matriz for realizado em diferentes ordens, podem ser notadas diferenças relevantes em performance e tempo de execução, mesmo que as operações fundamentais e resultados sejam os mesmos.

```
constexpr size_t matrix_size = 1024 * 8;
void RowMajor() {
  int* matrix = new int[matrix_size * matrix_size];
  for(size_t i = 0; i < matrix_size; ++i) {
    for(size_t i = 0; i < matrix_size; ++i) {
      matrix[(i * matrix_size) + j] = i + j;
    }
  }
  delete[] matrix;
}
void ColumnMajor() {
  int* matrix = new int[matrix_size * matrix_size];
  for(size_t i = 0; i < matrix_size; ++i) {
    for(size_t i = 0; i < matrix_size; ++i) {
      matrix[(j * matrix_size) + i] = i + j;
    }
  }
  delete[] matrix;
}
```

Listagem 1. Duas funções escritas em C++ que realizam as mesmas operações, a única diferença sendo a ordem de acesso das matrizes.

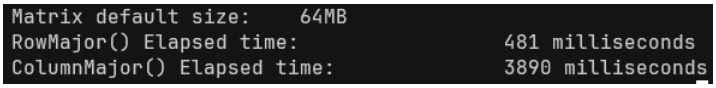

Figura 3. Resultado das diferenças de performance entre as ordens de acesso por colunas e por linhas.

Nas Figuras 1 e 3 são apresentados os códigos das duas abordagens e a discrepância dos tempos de execução, tais valores são explicados pelo número de *cache misses* e *cache hits* em cada uma das implementações [Nafack 2023].

2.2. Paralelismo e as memórias *cache*

Por muitas décadas os manufaturadores de processadores focaram em três pontos para melhorar o desempenho de seus produtos:

- Frequência de *clock*: aumentar a quantidade de ciclos executados em um segundo;
- Otimizações de execução de instruções: realizar mais trabalho **por ciclo**, *pipelining*, predição de *branches*, etc;
- Implementação das memórias *cache* para menor latência do acesso aos dados.

Todas essas melhorias representam aumento de desempenho em programas sequenciais e não-paralelizados. Contudo, no início dos anos 2000, os encarregados de projetar os novos processadores encontraram significante dificuldade em continuar aprimorando o *hardware* unicamente a partir destas estratégias citadas [Sutter 2005].

Quanto ao incremento das frequências de *clock*, principalmente três problemas surgiram: vazamento, muita dificuldade de controlar o calor do chip (descrito como muito difícil de dissipar) e consumo de energia demasiado. Por conta disso, nota-se que até mesmo atualmente, boa parte dos processadores visados ao público comum ainda se mantém na faixa de *clock* base aproximado de 3 a 4 *GHz*, mesmo essa frequência tendo sido atingida no início do século.

A partir deste ponto, os projetistas de processadores decidiram focar na paralelização da execução das instruções, adicionando *hyperthreading* e múltiplos núcleos nos *chips*. *Hyperthreading* se baseia na execução de duas ou mais *threads* em um núcleo, permitindo que várias linhas de execução sejam executadas ao mesmo tempo, contudo, estas instruções ainda compartilham boa parte dos recursos: geralmente uma unidade aritmética, uma unidade de ponto flutuante (*FPU*) entre outros componentes. Por conta disso, códigos executados em modo de *hyperthread* conseguem melhorias em média de cerca de quinze por cento [Sutter 2005] – em condições razoáveis – em relação à execução sequencial.

Em *chips* que possuem mais de uma *CPU* (isto é, *Central Processing Unit*, ou Unidade Central de Processamento), é possível conseguir verdadeira execução paralela de instruções, onde cada núcleo é encarregado de executar uma ou mais *threads* (por *hyperthreading*). Desta forma, o processador pode escalar o desempenho de seu programa proporcionalmente ao número de núcleos presentes no processador, por exemplo, tendo velocidade de execução potencialmente oito vezes mais rápida num processador com oito núcleos.

Contudo, a velocidade de execução dos programas continua atrelada às memórias *cache*, e acessar a memória *RAM* principal ainda é uma operação muito custosa (Tabela

1). Além disso, a operação em múltiplos núcleos acarreta na necessidade de cada um possuir suas próprias memórias *cache*. Com isso, podem acontecer duas situações graças a diferentes núcleos tentarem acessar os mesmos dados ou as mesmas linhas de cache: **verdadeiro compartilhamento** e **falso compartilhamento**.

O **verdadeiro compartilhamento** (*true sharing*) quando múltiplas *threads* tentam acessar a mesma posição de memória e pelo menos uma delas faz uma escrita, o que faz aquela parte da *cache* ser marcada como inválida.

O **falso compartilhamento** (*false sharing*) ocorre quando múltiplas *threads* acessam variáveis diferentes mas que estão na mesma linha de *cache*. Neste caso, qualquer escrita causa invalidação da linha de *cache* inteira, que necessitará ser constantemente atualizada e sincronizada [Henessy and Patterson 2017].

A invalidação de *cache* é baseada no protocolo *MESI*, que rotula os blocos da *cache* com quatro diferentes estados: ***M**odified* (modificado); ***E**xclusive* (exclusivo); ***S**hared* (compartilhado) e ***I**nvalid* (inválido) [Al-Waisi and Agyeman 2017]. Assim, caso um bloco seja rotulado como inválido, é necessário refazer a cópia dos dados, o que gera problemas de desempenho devido ao alto tráfego de memória para garantir a coerência das linhas de *cache*.

Portanto, para garantir boa performance, é importante que não haja dependência entre os dados operados por núcleos diferentes. Para isso, é importante que os programadores responsáveis levem em consideração o funcionamento do hardware ao projetar as estruturas de dados e algoritmos que serão implementados, a fim de conseguir um bom desempenho das máquinas, algo que é importante no desenvolvimento de jogos e simulações em tempo real.

Nesta situação, nota-se que é apropriada a programação de jogos através de práticas de desenvolvimento que evitem acessos ineficazes de memória e facilite a modelagem eficiente de dados com a finalidade de alto desempenho.

3. O paradigma de Programação Orientada a Dados

Tendo em vista as limitações e possibilidades do *hardware* moderno de computadores, buscaram-se formas de projetar os softwares em torno do comportamento eficiente da memória, separando os dados do código, ou seja, na programação orientada a dados, as informações são organizadas de forma simples, previsível e linear, enquanto a parte operacional (os códigos) apenas operam nestes dados. Pode-se, então dizer que *softwares* orientados a dados são baseados apenas nas transformações de dados de um estado A para outro B.

Além disso, é preferível o armazenamento de dados homogêneos, e que serão operados em conjunto, por exemplo: na situação em uma simulação de física, onde seja preciso sempre atualizar as coordenadas tridimensionais de uma partícula. Neste caso, não faz sentido a cache ser populada com informações da cor desta partícula, portanto, é necessário separar as estruturas de dados de acordo com seu propósito e padrões de acesso, algo que é solucionado utilizando *structs of arrays*. Ademais, pela facilidade de operar os dados em conjunto, operações *SIMD* vetoriais se tornam simples e triviais [Lengyel 2016].

3.1. Bancos de dados relacionais

Jogos digitais são *softwares* de tempo real extremamente complexos, há a obrigatoriedade de processar milhões de dados por segundo, desde subsistemas de vídeo que necessitam entender as características de tudo a ser renderizado; de áudio que precisam pré-processar as amostras de som que serão enviadas aos alto-falantes de forma a atender à imersividade demandada pelos jogadores; de entrada e saída, que precisam ler as informações dos controles, teclados, armazenamento e qualquer outro periférico utilizado, entre tantos outros exemplos. De forma geral do ponto de vista do programador, essas operações podem não aparentar serem complexas isoladamente, contudo, a partir do momento em que os subsistemas precisam se comunicar entre si, a interconectividade de dados acrescenta grande complexidade aos arranjos de dados. Por exemplo, pode-se pensar em uma operação de áudio que necessita saber da posição do jogador para saber se é necessário adicionar um efeito de reverberação aos passos do personagem, isto é, o subsistema de áudio consultando dados pertencentes ao subsistema de física – acessos de dados desta forma ocorrem o tempo todo durante a execução de um jogo e, a cada recurso adicionado ao jogo, a complexidade cresce cada vez mais.

O paradigma orientado a objetos ajuda a que o programador não necessite pensar nessa complexidade da interconexão de dados a partir dos conceitos de encapsulamento e envio de mensagens. Portanto, caso o subsistema de áudio necessite de uma informação de física, basta enviar uma mensagem a ele e operar sobre o resultado respondido. Na grande maioria dos *softwares*, esta solução é confiável e rápida o suficiente, contudo, no que se diz ao desenvolvimento de jogos e demais programas que necessitam de altíssimo desempenho, o conceito de "objetos" que devem ser instanciados e atrelam o comportamento aos dados gera vários *overheads*. Um destes *overheads* é a limitação do acesso a dados por interfaces, além do fato das operações "por objeto" acarretarem no acesso ineficiente da memória, grande número de chamadas de função e *branches*. Na Listagem 2 pode-se ver um exemplo de uma classe no paradigma de orientação a objetos que representa um "Jogador".

```
1  class Player : public GameObject
2  {
3  public:
4      Player();
5      // Demais metodos publicos
6  private:
7      Vector3 m_Position;
8      int m_Health;
9      std::vector<Weapon> m_Weapons;
10     std::string m_Name;
11     Mesh *m_Mesh;
12 };
```

Listagem 2. Exemplo de um objeto "Player" num software orientado a objetos.

No livro "Data Oriented Design" [Fabian 2018], Richard Fabian apresenta uma solução quanto ao problema da complexidade e interconexão de dados, que se baseia em armazenar os dados de forma similar a bancos de dados relacionais, com dados puros (*Plain Old Data*) separados do código, que apenas irá consultar as tabelas de informação e operar sobre elas. Para entendimento desse conceito, apresenta-se uma situação hipotética de um jogo onde existem salas que possuem texturas e animações; algumas delas estão

ligadas a outras por portas e podem conter itens coletáveis dos tipos “chave”, “poção” e “armadura”, estes possivelmente contendo uma animação – iniciada a partir da interação por parte do jogador – e uma cor. Para isso, cria-se tabelas de dados como “*Rooms*” para descrever as salas e “*Pickups*” para descrever os coletáveis.

As Tabelas 2 e 3 representam os dados (não normalizados) do mundo descrito anteriormente dispostos de forma semelhante aos bancos relacionais. Num software orientado a objetos os dados ainda poderiam ser dispostos desta forma, onde cada linha representaria uma instância de um objeto que modelaria o conceito “Sala” ou “Coletável”. Contudo, pode-se notar o número considerável de entradas nulas, isto pode acarretar em perda de desempenho ao popular a memória cache e pouca flexibilidade arquitetural, visto que é possível haver dados duplicados entre tabelas. Além disso, há a necessidade de checagens para garantir que um elemento não é nulo (aumentando a chances de erros de predição de *branch*). Devido a isto é necessário realizar o processo de normalização das tabelas.

PickupID	MeshID	TextureID	PickupType	ColourTint	Anim
k1	msh_key	tex_key	KEY	Copper	anim_keybob
k2	msh_key	tex_key	KEY	Silver	anim_keybob
p1	msh_pot	tex_pot	POTION	Green	NULL
p2	msh_pot	tex_pot	POTION	Purple	NULL
a1	msh_key	tex_arm	ARMOUR	Copper	NULL

Tabela 2. A tabela “*Pickups*” [Fabian 2018].

RoomID	MeshID	TextureID	WorldPos	Pickups	Locked	IsEnd
r1	msh_rmstart	tex_rmstart	0, 0	NULL	r2, r3	true WorldPos(1,1)
r2	msh_rmtrap	tex_rmtrap	-20,10	k1	r1, r4	false
r3	msh_rm	tex_rm	-10,20	k2,p1,a1	r1, r2, r5	false
r4	msh_rm	tex_rm	-30,20	p2	r2	false
r5	msh_rmtrap	tex_rmtrap	20,10	NULL	NULL	false

Tabela 3. A tabela “*Rooms*” [Fabian 2018].

Algo que pode ser observando olhando para as Tabelas 2 e 3 é o fato de cada linha possuir uma chave primária, similar a um *set* (conjunto), onde é possível acessar um elemento através de um identificador único.

3.1.1. Primeira forma normal

Na primeira forma normal, deve-se garantir que as tabelas não sejam esparsas, isto é, garantir que não existam entradas nulas. Isto é garantido através do deslocamento dos elementos repetidos e opcionais para suas próprias tabelas [Fabian 2018]. Observando as Tabelas 4, 5 e 6 pode-se notar que não existem mais entradas nulas e que possuem vetores de valores, cada célula da tabela tem sua atomicidade garantida. Entretanto, mais tabelas são criadas, ou seja, maior gasto de memória, contudo, a eliminação de entradas nulas e duplicadas permite código mais coeso e os estados para qual os algoritmos terão

que operar são cada vez mais bem definidos, não tendo que cogitar casos extremos onde dados são opcionais nem situações de comportamento indefinido [Fabian 2018].

PickupID	MeshID	TextureID	PickupType
k1	msh_key	tex_key	KEY
k2	msh_key	tex_key	KEY
p1	msh_pot	tex_pot	POTION
p2	msh_pot	tex_pot	POTION
a1	msh_key	tex_arm	ARMOUR

Tabela 4. A tabela "*Pickups*" na primeira forma normal [Fabian 2018].

PickupID	ColourTint
k1, a1	Copper
k2	Silver
p1	Green
p2	Purple

Tabela 5. A nova tabela "*PickupTints*" [Fabian 2018].

PickupID	Anim
k1	anim_keybob
k2	anim_keybob

Tabela 6. A nova tabela "*PickupAnims*" [Fabian 2018].

No que tange o assunto deste artigo, a primeira forma normal é o suficiente para entender os conceitos necessários na implementação de Entity Component Systems, como os conjuntos esparsos (Seção 4.3) – fundamental para a utilização eficiente da memória, que é muito similar à técnica de normalização explicada aqui.

3.2. *Struct of Arrays* contra *Arrays of Structs*

No processo de projeto das estruturas de dados em um programa, é comum serem criadas estruturas que representam conceitos do mundo real, desta forma, é comum encontrar *structs* ou *classes* chamadas "Aluno" ou "Pessoa" que apresentam todos os dados que estes conceitos podem necessitar, assim como ilustrado na Listagem 3. Entretanto, isso apresenta um problema de performance: caso seja necessário operar a média geral de todos os alunos, é preciso armazenar também informações como a idade e os nomes completos de todos, desta forma, a memória cache será populada com informações irrelevantes à operação sendo realizada, aumentando o número de *cache misses* e, consequentemente, piorando a performance do programa.

Para muitos softwares, esta perda de desempenho pode não ser um problema, e o código pode ser mantido desta forma. Contudo, em softwares onde a velocidade de processamento é fundamental, como em jogos, é preciso uma abordagem mais eficiente. Para isso, é possível organizar os dados em *Struct of Arrays* (estrutura de vetores), ou *SoA*,

que consiste em uma única estrutura que contém todos os alunos, e nela, são armazenados vetores de cada um dos dados, desta forma, caso seja importante fazer algo como calcular a média geral, percorre-se o vetor de médias e a memória *cache* estará completamente populada com informações relevantes à operação. Na Listagem 4 é possível visualizar uma *Struct of Arrays*, onde todos os dados dos alunos estão armazenados de forma densa e linear.

```
struct Aluno {
  string nome_completo;
  string data_de_nascimento;
  float media;
  int idade;
};
Aluno alunos[128];
```

Listagem 3. Exemplo de uma *struct* em C++ com dados de um único aluno e um vetor que armazena todos os alunos, ou seja, *Array of Structs* (vetor de estruturas).

```
struct Alunos {
  string nome_completo[128];
  string data_de_nascimento[128];
  float media[128];
  int idade[128];
};
Alunos alunos;
```

Listagem 4. Exemplo de uma *struct* com vetores de dados de todos os alunos, ou seja, *Struct of Arrays*.

Existem diversos outros conceitos importantes ao paradigma orientado a dados, contudo, os já apresentados foram o suficiente para o nascimento de um padrão arquitetural mais apropriado para a aplicação em jogos e simulações em tempo real: os *Entity Component Systems*.

4. *Entity Component Systems*

Os *Entity Component Systems* representam um padrão arquitetural com base nos conceitos advindos da programação orientada a dados que visam melhorar o desempenho de aplicações em tempo real, principalmente jogos eletrônicos que demandam o processamento de um número expressivo de dados a cada iteração do laço de atualização do estado de jogo, como: jogos de mundo aberto, multijogadores, simuladores. Esta seção irá tratar os conceitos básicos dos ECS e os detalhes gerais da implementação da biblioteca utilizada nos testes, a EnTT, criada por Michele Caini [Caini 2019].

Para o entendimento inicial acerca do que é e como funcionam os *Entity Component Systems* necessita-se entender o significado dos três conceitos presentes neste padrão: as entidades, os componentes e os sistemas.

4.1. Entidades, componentes e sistemas

No contexto de jogos eletrônicos, pode-se dizer que **entidades** representam tudo que existe no mundo simulado, seja um personagem que é controlado pelo jogador, um monstro, um carro, uma espaçonave, uma casa, ou qualquer objeto inserido pelos desenvolvedores. No desenvolvimento de um software orientado a objetos, a solução para a

representação destas entidades é simples: é criada uma classe chamada “Jogador” que herda a classe “Humano”, a classe “Carro” que herda a classe “Automóvel”, etc.

Entretanto, além de esta prática poder ferir os princípios de acessos lineares e efetivos de memória vistos na Seção 3, é comum que as classes comecem a se entrelaçar de formas não esperadas e gerem códigos repetidos e difíceis de manter e expandir. Além disso, existe a possibilidade de novas entidades precisarem herdar comportamentos de outros conceitos não relacionados para funcionar [Nystrom 2011]. Outro caso onde isso pode causar a falta de flexibilidade na alteração de código, é na alteração de um conceito do projeto do jogo: caso o *game designer* entenda que o jogador se interessará mais em controlar um monstro do que um humano, modificar o código e mudar a classe herdada de “Humano” para “Monstro” pode causar vários problemas de compatibilidade no envio das mensagens entre os objetos.

Já os **componentes**, são representações reutilizáveis de dados, isto é, agrupamentos de informações que possuem alguma característica em comum e que podem ser relacionados a vários tipos de entidades, garantindo também a facilidade de manutenção do código, visto que caso seja preciso mudar o comportamento de um componente que seja compartilhado por diversas classes, o código está localizado num lugar bem definido, não necessitando editar vários pontos distintos da base de código.

Como solução deste problema de manutenção de classes no contexto de jogos, adotou-se o conceito de “composição acima de herança”, em que as superclasses são os comportamentos em si, de forma a desacoplar e evitar a duplicação de código quando há a necessidade de duas entidades diferentes compartilharem as maneiras de funcionamento. Esta forma arquitetural é nomeada de composição orientada a objetos, e possui benefícios sobre a orientação a objetos pura, visto que com ela obtemos componentes reutilizáveis e flexíveis mas ainda não é o suficiente para o desacoplamento completo desejado em uma arquitetura orientada a dados, visto que as informações e o comportamento ainda estão entrelaçados [Romeo 2016].

No processo de conversão da composição orientada a objetos ao paradigma orientado a dados, é necessário remover a relação entre os comportamentos e os dados. Para isso, são criados os **sistemas**: funções simples que irão operar sobre dados homogêneos e lineares. Portanto, os componentes se tornarão estruturas *Plain Old Data*, isto é, dados simples e puros, sem nenhum comportamento atrelado a eles. Um exemplo básico de um componente reutilizável é um que representa a posição de uma entidade no mundo, sendo geralmente uma estrutura de três números de ponto flutuante, cada um armazenando as coordenadas dos eixos do sistema tridimensional.

Retornando ao exemplo do primeiro parágrafo desta seção, em que é mencionado a mudança da classe “Jogador” para herdar “Monstro” ao invés de “Humano”, com a utilização de componentes, basta modificar ou remover os componentes necessários para o funcionamento de um humano, e substituí-los pelos que compõe um monstro, sem gerar atrito entre as interfaces, visto que os componentes são operados de forma homogênea, não importando a classe a qual eles pertencem.

No que se refere ao padrão dos *Entity Component Systems*, por buscar melhor localidade espacial (e consequentemente melhor aproveitamento da cache), as entidades são apenas índices únicos apontando posições em vetores homogêneos de componentes (que

estão armazenados da forma *Struct of Arrays*) [Caini 2019]. Logo, há apenas uma estrutura de dados de arquitetura "*Property-centric design*" [Gregory 2018], assim, elimina-se qualquer classe "*Entity*" ou "*GameObject*" e obtém-se dados puros que serão operados pelos sistemas de forma efetiva, paralelizável e sem desperdício da cache.

Percebe-se, entretanto, que com um vetor para cada componente e com uma posição reservada para cada entidade, há possibilidade de grande desperdício de memória nos vetores, visto que nem todos objetos da simulação estarão atribuídos a todos os componentes, contudo, o programa irá alocar os recursos para isso. Esta condição é um malefício intrínseco aos ECS, e que deve ser levado em consideração pelo desenvolvedor, já que jogos podem ter como plataformas-alvo dispositivos de baixa capacidade de memória RAM, como celulares e consoles relativamente antigos (jogos imensos como *Grand Theft Auto V* e *The Last of Us* tinham como requisitos de memória apenas 256 MB DRAM, no *Playstation 3* [Gregory 2018]) e nesta situação, o ganho de performance pode ser irrelevante quando comparado à possibilidade do jogo sequer ser compatível com o hardware alvo. Além disso, os vetores em geral se tornam esparsos, isto é, apresentam várias lacunas entre os dados utilizados, impedindo a boa utilização da cache [Caini 2019]. Outro problema desta implementação vem da situação em que necessitamos saber as entidades que possuem certos conjuntos de componentes.

4.1.1. O problema da contiguidade

Ademais, surge outro problema que, neste trabalho se chamará "problema da contiguidade", no qual, na necessidade de operar vários componentes ao mesmo tempo. Para fazer isto, é difícil manter os vetores arranjados de forma contígua e homogênea para os sistemas realizarem as operações, já que algum sistema $S1$ pode querer operar sobre os componentes "A" e "B", enquanto outro sistema $S2$, quer operar sobre os componentes "A", "B" e "C" e não pode haver nenhuma sobreposição entre os componentes operados por $S1$ e $S2$. Desta forma, é difícil evitar a coincidência de dados na memória e garantir que cada sistema acessará sempre vetores uniformes.

A partir dos problemas mencionados anteriormente, três vertentes de ECS surgem, buscando resolvê-los, sempre focando na flexibilidade e, principalmente, no desempenho, sendo eles, os *frameworks* de *bitset*, os arquétipos e o que será utilizado nos experimentos deste artigo, os conjuntos esparsos. Por serem pouco utilizados, os ECS baseados em *bitsets* não serão explicados em detalhes, ainda assim, é justo mencionar sua existência.

4.2. Os arquétipos

Em uma situação imaginária existem dois componentes, chamados "A" e "B". Neste mundo, três entidades estão vivas, $E0$ e $E2$ abrigam o conjunto de componentes $[A, B]$ enquanto $E1$ apenas $[A]$. Assim, caso seja necessário um sistema operar sobre $[A, B]$, o arranjo de memória – como demonstrado na Tabela 7 – é desfavorável ao acesso contíguo dos dados [Mertens 2020].

	$E0$	$E1$	$E2$
A	✓	✓	✓
B	✓		✓

Tabela 7. As entidades $E0$, $E1$, $E2$ e seus componentes.

Neste caso, é possível apenas reordenar os vetores, de forma que as colunas $E0$ e $E2$ fiquem uma ao lado da outra.

	$E0$	$E2$	$E1$
A	✓	✓	✓
B	✓	✓	

Tabela 8. As entidades $E0$, $E2$, $E1$ e seus componentes, após o rearranjo de dados.

Este rearranjo de dados soluciona de forma satisfatória o "problema da contiguidade" num mundo com dois componentes, porém, conforme mais componentes são adicionados, apenas a ordenação de dados em vetores com lacunas não é o suficiente para garantir a contiguidade dos dados. Neste caso, é possível criar vetores por tipo, ao contrário de criá-los por componente, ou seja, as entidades $E0$ e $E2$ seriam classificadas como donas do conjunto $[A, B]$ e a $E1$ do $[A]$. Estes tipos criados a partir da posse de conjuntos são chamados de arquétipos. Desta forma, é possível armazenar os componentes da forma identificada na Listagem 5.

```
// Tipo do conjunto [A]
A a[2];

// Tipo do conjunto [A, B]
A a[2];
B b[2];

// Tipo do conjunto [A, C]
A a[2];
C c[2];
```

Listagem 5. Pseudocódigo que apresenta representações distintas de como os arquétipos podem ser definidos [Mertens 2020].

No que diz respeito ao mapeamento de entidades para componentes, é necessário armazenar informações relativas ao que compõe os arquétipos. Implementações diferem, mas as soluções em geral se baseiam em atribuir identificadores aos componentes e guardar estas combinações em vetores. É importante também ordená-los, para que não haja diferença entre os conjuntos $[A, B]$ e $[B, A]$, visto que eles expressam o mesmo tipo. Assim, a estrutura de dados de um arquétipo pode ser escrita da seguinte forma:

```
using Column = vector<T>;

struct Archetype {
  ArchetypeId id;
  Type type;
  vector<Column> components;
```

```
};
```

Listagem 6. Pseudocódigo da implementação de um arquétipo [Mertens 2020].

Pode-se notar, na Listagem 6, que basta armazenar o identificador único de um arquétipo (relativo à combinação exclusiva de componentes), o tipo (um vetor dos índices dos componentes em si) e, então, um vetor de vetores de componentes, que é onde de fato estarão armazenados os dados pertinentes: os componentes. Após a definição das estruturas de dados que definem um arquétipo, basta mapear as entidades (que são apenas números inteiros e únicos) às *structs*, isto pode ser feito de forma eficiente utilizando *Hash Maps* não-ordenados e um identificador relativo à coluna da tabela onde o componente necessário está presente.

Um ECS baseado em arquétipos bastante difundido é o desenvolvido pela *Unity Technologies* [Unity-Technologies 2024b] e disponibilizado ao uso comum pelos usuários da *Unity Engine*, nesta implementação, há o conceito de *chunks*, blocos uniformes de memória de 16*KiB* consistentes de vetores de componentes e um adicional com os identificadores das entidades proprietárias dos dados contidos neste *chunk*. Com isso, numa remoção de uma entidade do arquétipo por qualquer motivo, basta mover os componentes presentes no final dos vetores para preencher a lacuna criada, garantindo sempre que a memória *cache* esteja completamente populada com informações relevantes na execução dos sistemas [Unity-Technologies 2024a].

Portanto, do ponto de vista arquitetural, o fluxo de trabalho proposto pela solução dos arquétipos é satisfatória e, do ponto de vista computacional, potencialmente muito rápida. Entretanto, apresenta complexidade de código e é necessário grande cuidado na implementação pela necessidade de muitos vetores de componentes, criação dinâmica de tabelas no estilo orientado a dados e organização de memória constante. Por exemplo, uma operação comum como a simples remoção de um componente pertencente a uma entidade pode implicar na criação de um arquétipo totalmente novo, gerando a necessidade de alocar memória dinamicamente e atualizar índices.

4.3. Conjuntos esparsos

A partir dos problemas citados anteriormente acerca dos arranjos de memória e formas de dispor os componentes de forma amigável aos acessos sequenciais de memória, surgiram vertentes diferentes a fim de garantir os ganhos de performance propostos inicialmente. Uma solução comum baseia-se em agrupar componentes por arquétipos, entretanto, a resolução descoberta por diversos programadores funciona através da representação eficiente de conjuntos esparsos, que utiliza dois vetores: um esparso e um denso. O vetor esparso contém as posições dos elementos presentes do vetor denso. O vetor denso, por sua vez, contém um inteiro que representa o índice utilizado no outro vetor [Briggs and Torczon 1993]. Essa estrutura de dados permite a inicialização, busca e inserção em tempo $O(1)$ e iteração em $O(n)$. Com os conjuntos esparsos, gasta-se tempo indexando valores nos vetores densos e esparsos, entretanto, as vantagens de ter dados densos e homogêneos irão, na grande maioria dos casos, deixar irrelevantes os atrasos de indexação. A Figura 4 exibe a relação entre os dois vetores, em que o valor 0 no vetor esparso define a posição no vetor denso que contém o inteiro 3, e vice-versa.

No que diz respeito à implementação de conjuntos esparsos para a iteração e processamento de componentes, pode-se ter uma estrutura de dados com três vetores: um

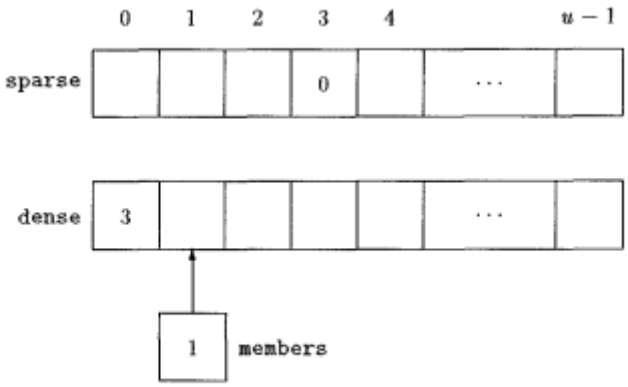

Fig. 1. A set with one member.

Figura 4. Um conjunto esparso composto por seus dois vetores. [Briggs and Torczon 1993]

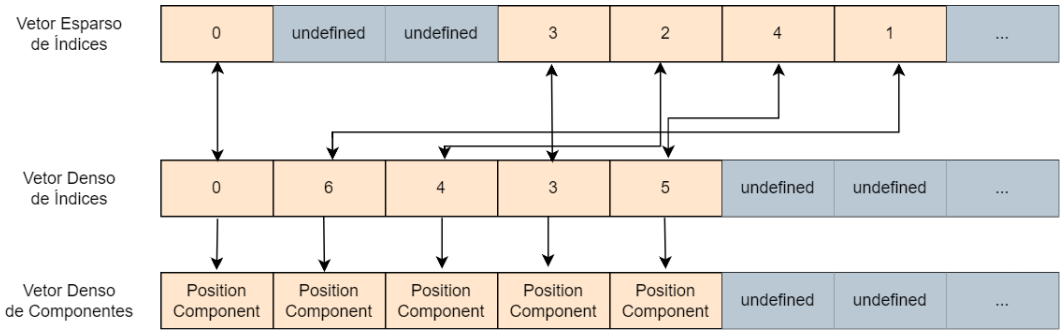

Figura 5. Um conjunto esparso com vetor denso, que é o único vetor importante para o processamento de dados, enquanto os outros existem somente para mapear a posse dos dados.

vetor esparso de índices e dois outros vetores densos, um de índices para o esparso e outro dos componentes em si. Os vetores de índices têm como finalidade manter a relação das entidades com os componentes e identificar quem são os donos de cada dado, com isso, os sistemas podem operar unicamente nos dados dos vetores densos de componentes, sem a necessidade de realizar indexação prévia [Caini 2019]. A Figura 5 demonstra um conjunto esparso de componentes de posição, neste caso, se for necessário atualizar a posição de todas as entidades que sejam donas de um deles, basta iterar entre a posição 0 e 4 sem a necessidade de checar se a posição é válida ou demais verificações.

4.4. Operação de múltiplos componentes

As propriedades dos conjuntos esparsos solucionam o problema das lacunas nos vetores de componentes, entretanto, o "problema da contiguidade", mencionado na Seção 4.1.1, ainda existe e afeta expressivamente o desempenho dos sistemas.

A solução trivial, neste caso, é percorrer o vetor denso que seja mais curto entre eles e, durante a iteração, verificar quais elementos estão ativos em ambos (ou seja, que devem ser operados). Entretanto, os malefícios são expressivos: caso existam 200 mil entidades que contém conjuntos diversificados de componentes, haverá a necessidade de

sempre verificar e validar – possivelmente até mais de uma vez – a interseção de posse entre elas, o que irá gerar grande carga de computações desnecessárias e, consequentemente, poderá piorar o tempo de execução expressivamente.

A solução encontrada na implementação da EnTT [Caini 2024] se baseia na funcionalidade de agrupamento, a qual parte da premissa de organizar os vetores densos de acordo com as combinações de componentes. Neste caso, se na simulação existirem mil entidades que são proprietárias dos dados "Position" e "Velocity", existirão intervalos idênticos nos vetores que corresponderão a estas entidades. Portanto, é necessário manter os dados sempre ordenados e agrupados, que pode causar problemas de desempenho em situações de constante atribuição e remoção de componentes, contudo, estas operações não são comuns, e o tempo de execução poderá se tornará melhor. Ademais, não são necessários passos de organização em tempo de processamento dos sistemas, já que a responsabilidade de organizar os dados na memória fica a cargo das funções de inserção e remoção dos componentes, livrando a necessidade de passos extras ao chamar os sistemas. O desempenho utilizando a funcionalidade de agrupamento pode melhorar substancialmente, pois além dos sistemas poderem atuar sobre vetores completamente populados por dados relevantes (diminuindo o número de *cache misses* desnecessárias a zero) não são necessários passos de indireção ou atualizar os vetores de índices [Caini 2019].

4.5. Simples exemplo hipotético de agrupamento

Para melhor entendimento do conceito de agrupamento e atuação dos sistemas no ECS, é necessário demonstrar o comportamento da organização dos dados na memória de acordo com a adição e remoção de componentes em uma situação similar ao que pode ser encontrado em um *software real*. Com esse objetivo, as circunstâncias supostas são as seguintes: num jogo hipotético existem três tipos de entidades, sendo dois monstros não-jogáveis e um humano jogável (isto é, controlado pelo usuário). Todos eles possuem em comum três componentes, sendo estes os de posição, velocidade e renderização (*PositionComponent*, *VelocityComponent* e *RenderComponent*, respectivamente), contudo, ambos os tipos de monstros exibem comportamentos diferentes ao se aproximar do jogador: um dos tipos de monstros muda de cor (utilizando um *ColorComponent*), enquanto o outro tipo muda de tamanho (utilizando um *ScaleComponent*). Com a finalidade de facilitar referenciar cada tipo de monstro, chama-se o primeiro mencionado de *MonsterColor* e o segundo *MonsterScale*.

A Figura 6 apresenta as associações entre estes componentes e sistemas, são elas:

- *SimpleRenderingSystem*: atua sobre os componentes de renderização e posição, ou seja, sobre todas as entidades que não possuem o componente de cor nem de escala.
- *ColorRenderingSystem*: é encarregado de processar os componentes de renderização, posição e cor;
- *ScaleRenderingSystem*: possui a tarefa de renderizar todas as entidades que possuem os componentes de renderização, posição e escala;
- *PhysicsSystem*: tem como função atualizar a posição de cada entidade, para isso, são necessários os componentes de posição e velocidade;
- *ColorMonsterTriggerSystem*: é o sistema que deve mudar as cores do monstro caso ele esteja perto do jogador, ele opera sobre os componentes de posição e cor;

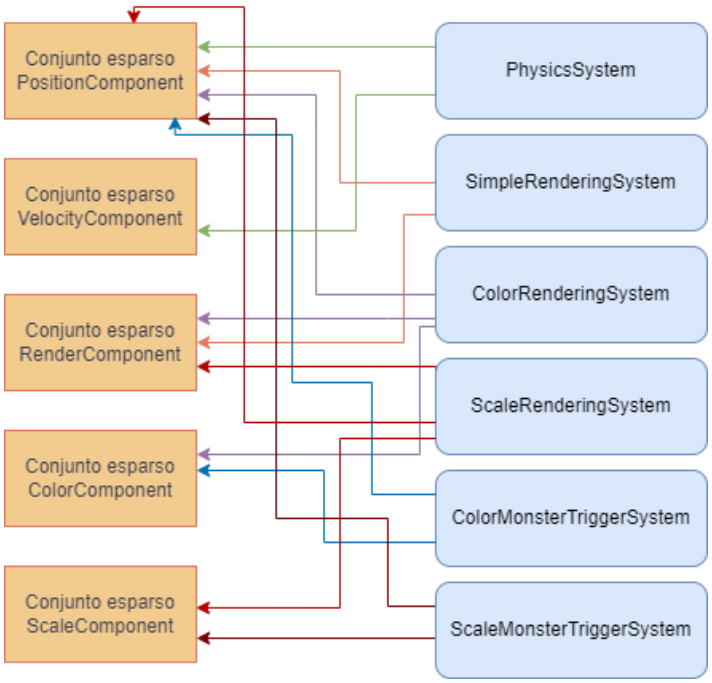

Figura 6. Relações entre os sistemas e os conjuntos esparsos de componentes.

- *ScaleMonsterTriggerSystem*: muda a escala do monstro em caso de proximidade com o jogador, operando sobre os componentes de posição e escala.

Nesta subseção, serão mostrados os passos de execução da simulação e o arranjo dos dados e da memória conforme os dados são incluídos e também o comportamento da funcionalidade de agrupamento dos vetores densos:

1. Suponha-se o início da simulação apenas com a existência da entidade "jogador", portanto, a inclusão de três componentes, sendo estes os de posição, renderização e velocidade. O resultado deste passo é visto na Figura 7;
2. Agora são instanciadas 10 mil entidades do tipo *ColorMonster*, ou seja, com os componentes de posição, renderização, velocidade e cor. Para isso, deve-se analisar os vetores densos e os reordenar caso necessário. Como ainda não existiam instâncias deste tipo de monstro, é possível apenas incluí-los nas posições 1 até 10.000 dos vetores, sendo esses valores os limites do grupo;
3. Neste passo são criados 20 mil monstros do tipo *ScaleMonster*, com os componentes de posição, velocidade, renderização e escala. Assim como o item anterior, não é necessário ordenar os vetores visto que não existiam entidades do mesmo tipo vivas no mundo. Os limites desse grupo serão 10.001 até 30.000 dos vetores, esses valores serão os limites do grupo. O resultado deste passo é visto na Figura 8;
4. Visto que nesta etapa os componentes estão devidamente organizados na memória, pode-se executar todos os sistemas necessários, basta consultar os limites dos grupos (assim como demonstrado na Figura 8) e iterar por estes componentes. Desta forma, caso seja desejado efetuar o sistema "*ColorMonsterTriggerSystem*" é apenas necessário o componente de posição do jogador (para a comparação de distancia) e iterar pelos componentes de posição e cor dos "*ColorMonsters*". Aqui,

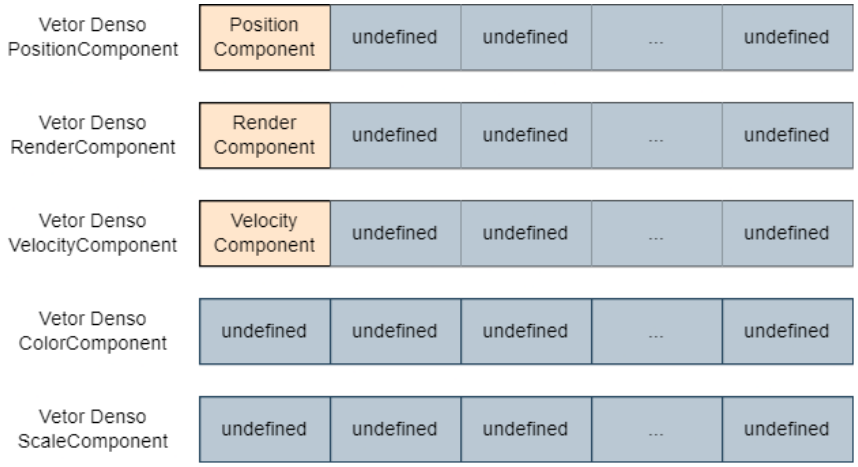

Figura 7. Estado dos vetores densos após o passo 1 da execução. Os dados presentes são correspondentes à entidade "jogador".

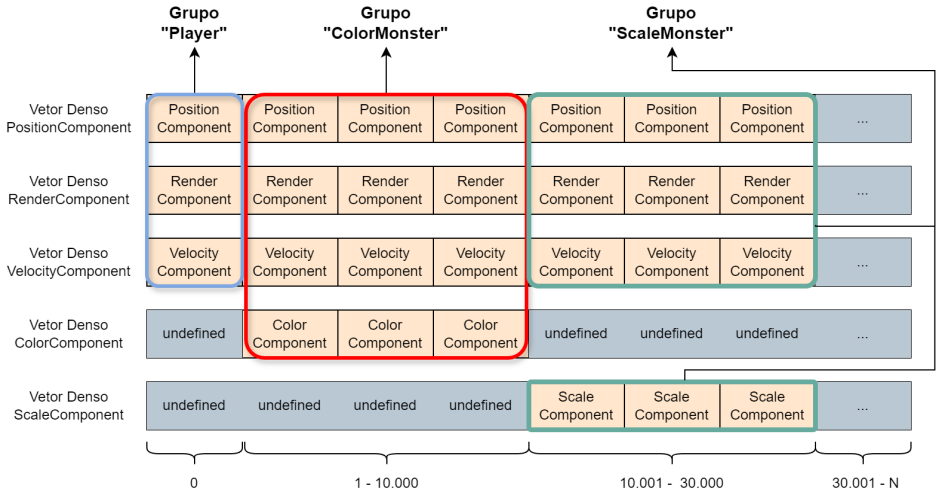

Figura 8. Estado dos vetores densos após o passo 3 da execução. Os dados presentes são correspondentes aos arquétipos "jogador", "*ColorMonster*" e "*ScaleMonster*".

```
static void SimpleRenderingSystem(void) {
  // Recebe o iterador (ou seja, os limites) do grupo.
  EntityGroup group = ecs.group<PositionComponent, RenderComponent>();
  for(Entity entity : group) {
    // Itera por todas entidades do grupo e as renderiza.
    auto& position = group.get<PositionComponent>(entity);
    auto& render = group.get<RenderComponent>(entity);
    RenderSimpleEntity(position, render);
  }
}

static void ColorRenderingSystem(void) {
  EntityGroup group = ecs.group<PositionComponent, RenderComponent, ColorComponent>();
  for(Entity entity : group) {
    auto& position = group.get<PositionComponent>(entity);
    auto& render = group.get<RenderComponent>(entity);
    auto& color = group.get<ColorComponent>(entity);
    RenderColorEntity(position, render, color);
  }
}

static void ScaleRenderingSystem(void) {
  EntityGroup group = ecs.group<PositionComponent, RenderComponent, ScaleComponent>();
  for(Entity entity : group) {
    auto& position = group.get<PositionComponent>(entity);
    auto& render = group.get<RenderComponent>(entity);
    auto& scale = group.get<ScaleComponent>(entity);
    RenderScaleEntity(position, render, scale);
  }
}
```

Figura 9. Exemplo de código em C++ de uma implementação dos sistemas descritos no passo 4.

também, se for desejado renderizar os três tipos de entidade, serão executados os sistemas *SimpleRenderingSystem*, *ColorRenderingSystem* e *ScaleRenderingSystem*, o primeiro atua sobre os componentes de renderização e posição na posição 0 dos vetores (visto que apenas o jogador é de renderização simples); o segundo atua sobre as posições 1 até 10.000 dos vetores densos dos *PositionComponent*, *RenderComponent* e *ColorComponent*; o último atua nos limites 10.001 até 30.000 dos vetores densos *PositionComponent*, *RenderComponent* e *ScaleComponent*. A Figura 9 exibe um código de uma implementação hipotética de sistemas com ECS.

4.6. A flexibilidade arquitetural do ECS

Indo além das capacidades computacionais do ECS, é interessante também mencionar as funcionalidades e benefícios arquiteturais da utilização desse padrão. Para isso, serão mencionados os padrões anteriores a ele (principalmente o modelo de "GameObject" orientado a objetos).

Dos anos 90 até o início dos anos 2000, programadores de jogos em geral utilizavam modelos arquiteturais estritamente orientados a objetos, sem a existência do conceito de componentes, o que dificultava a modificação e expansão do código. Jogos são programas altamente complexos, podendo apresentar enorme variedade de comportamentos e nuances, desta forma, um objeto presente pode ser 90% idêntico ao outro, mas apresentar pequenas características diferentes. Desta forma, procurou-se maneiras de criar códigos e comportamentos desacoplados.

À maneira orientada a objetos, é possível criar uma classe base `GameObject` que apresenta apenas comportamentos padrões e obrigatórios a todas as entidades do mundo, e, na necessidade de criar outros objetos, criam-se classes herdeiras que sobrescreverão

os métodos base. Entretanto, num processo de desenvolvimento altamente volátil como é o de jogos, a rigidez imposta por este fluxo de trabalho pode acabar gerando atrasos, acoplamento desnecessário de comportamentos e subsistemas, além de baixa reutilização dos códigos e coerência entre classes que são parecidas, ademais, os dados de cada objetos seriam fixos e não poderiam ser alterados em tempo de execução, necessitando recompilar os projetos em caso de alteração ou a implementação de ferramentas de *scripting* [Nystrom 2011].

A partir destes desafios arquiteturais, surgiu o padrão de composição orientada a objetos, citada previamente na Seção 4.1, onde os conceitos e comportamentos implementados por uma entidade são unificados em componentes que implementam aquele determinado conceito e as classes de objetos de jogo se tornam apenas *containers* destes componentes. No motor de jogo *Unity*, todas entidades são implementadas a partir deste padrão arquitetural, o que mostrou diminuir bastante a complexidade de desenvolvimento e garantir boa flexibilidade arquitetural [Unity-Technologies 2024b].

Contudo, o padrão "`GameObject`" ainda apresenta certos problemas de complexidade, principalmente pela atribuição de significado aos dados, isto é, baixa flexibilidade da forma como eles podem ser reaproveitados para a criação de novos algoritmos. Na palestra "*Overwatch: Gameplay architecture and Netcode*" [Ford 2017], o programador chefe *Timothy Ford* cita que o acoplamento entre significado (e consequentemente do comportamento) e os dados, dificulta o desenvolvimento de sistemas concisos, pequenos e de baixa complexidade. Desta forma, eles utilizaram um ECS personalizado para o desenvolvimento do jogo *Overwatch*, o que, segundo *Ford*, facilitou imensamente o processo de desenvolvimento e manutenção do jogo, impedindo que o código gerasse fricção no desenvolvimento, garantindo, assim, uma arquitetura de código muito flexível, simples de entender, de baixíssimo acoplamento e, acima de tudo, fácil de estender, algo fundamental para jogos multijogador que necessitam da adição de conteúdo por muitos anos. As Figuras 10 e 11 demonstram exemplos dos sistemas utilizados para a criação do jogo *Overwatch* e a simplicidade e desacoplamento dos códigos que, utilizando o padrão orientado a objetos, sobreporiam vários conceitos e comportamentos para a execução de um simples subsistema.

No ECS implementado para *Overwatch*, os componentes possuem métodos `Create()` polimórficos utilizados para definir dados-base e pequenas funções de acesso, no mais, não há nenhum comportamento atrelado a eles; estes são exclusivamente descritos nos sistemas, que não possuem nenhum estado. O estado global é gerenciado por uma classe `EntityAdmin` que possui um vetor de sistemas e um *hash map* que mapeia identificadores únicos a ponteiros para objetos do tipo *Entity*. As entidades são definidas por classes (o que foge um pouco da definição de entidades nos *frameworks* mais utilizados), que possuem um identificador único e um vetor de ponteiros aos componentes atrelados a ela [Ford 2017].

Ademais, bibliotecas modernas de ECS, como *EnTT* [Caini 2024] e *Flecs* [Mertens 2024], possibilitam a adição e remoção dinâmica de componentes, isto é, em tempo de execução. Com isso, a testagem de sistemas de jogo e simulação se torna potencialmente mais fácil, além de aumentar a flexibilidade das funcionalidades presentes na simulação. Além disso, algumas bibliotecas também permitem hierarquia de entidades, permitindo que conceitos advindos da orientação a objetos também sejam possíveis com

ECS.

4.7. Paralelismo

Pelos conceitos vistos na Seção 2.2, no âmbito da computação paralela de alta performance, é fundamental pensar nos arranjos de memória e evitar dependências de memória e estados de falso compartilhamento, para minimizar invalidações da memória *cache* e, com isso, necessitar a sincronização entre núcleos e novos acessos à memória principal [Drepper 2007].

Como citado na Seção 4, o ECS ajuda arquiteturalmente a desacoplar os significados e comportamentos dos dados – algo advindo diretamente da programação orientada a dados (Seção 3) –, assim, pode-se utilizar os componentes de diversas formas diferentes, ou seja, uma única estrutura de dados simples pode ter diversos comportamentos distintos quando inserido em conjuntos de componentes diferentes. Exemplo: um componente de posição é atualizado numa simulação de física, mas é apenas lido em um sistema de renderização, neste caso, os exatos mesmos dados tiveram funções diferentes. Consequentemente, teoricamente pode-se separar os sistemas que acessam apenas dados independentes, e desta forma, enviar estas funções independentes a cada *thread* [Romeo 2016].

Uma proposta da biblioteca *ECST* [Romeo 2017] para facilitar a paralelização dos *Entity Component Systems* é de separar as atividades paralelizáveis em dois níveis: **"paralelismo interno"** e **"paralelismo externo"**.

O paralelismo externo define o conceito de executar paralelamente sistemas completamente independentes. Estes são identificados através de grafos acíclicos direcionados de componentes ou dados, no qual são identificadas as cadeias independentes de vértices e, assim, são ditas próprias à paralelização. Um exemplo é onde se tem os dados de **Aceleração**, **Velocidade**, **Forma**, **Crescimento**, **Rotação** e **Renderização**. Para realizar o processo de renderização são necessários os dados de **Velocidade, Crescimento e Rotação**; a **Velocidade**, por sua vez, necessita da **Aceleração** e o **Crescimento** necessita da **Forma** [Romeo 2016].

Assim, tem-se as seguintes cadeias de dependência: Renderização $\rightarrow$ Velocidade $\rightarrow$ Aceleração; Renderização $\rightarrow$ Crescimento $\rightarrow$ Forma; Renderização $\rightarrow$ Rotação. Desta forma, cada uma dessas cadeias pode ser executada paralelamente antes da etapa de renderizar, sem que nenhum dado seja compartilhado entre elas.

No **paralelismo interno**, pode-se separar um único sistema em subtarefas, pois muitos deles não contém efeitos colaterais e realizam cálculos independentes. Contudo, este tipo de computação é mais restrita a problemas ditos "facilmente paralelizáveis" [Foster 1995], visto que há a necessidade de atualizar algumas estruturas de dados internas. Por fim, todas essas tarefas são executadas por *threads* trabalhadoras que ficam aguardando suas atribuições (os sistemas em si).

Bibliotecas diferem seus pormenores quanto à paralelização, mas em geral, a grande maioria utiliza conceitos parecidos como os de paralelização interna e externa, identificando os sistemas ou *pipelines* independentes e as repassando às *threads* trabalhadoras correspondentes. A biblioteca Flecs permite que o programador defina fases e *pipelines* e as paralelize [Mertens 2024].

Figura 10. Pequeno exemplo dos sistemas e componentes utilizados no desenvolvimento do jogo *Overwatch* [Ford 2017].

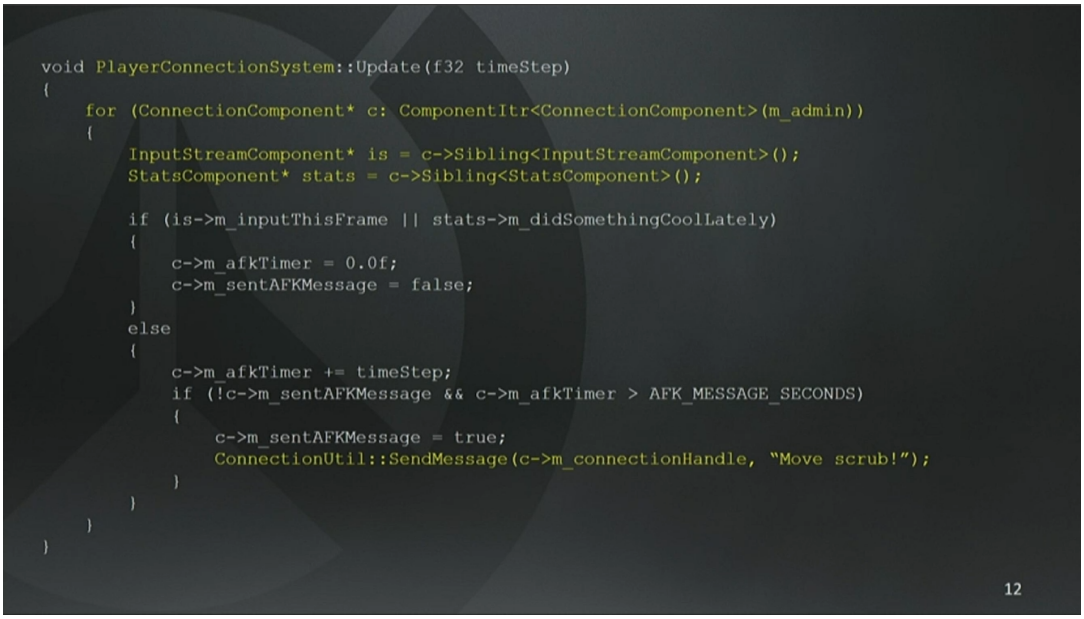

Figura 11. Simples sistema utilizado no jogo *Overwatch* para o monitoramento de *AFK* (*Away from keyboard* ou, longe do teclado, isto é, um usuário que está inativo há algum tempo) [Ford 2017].

5. O ECS no mundo real

Como visto nas seções anteriores, o paradigma ECS pode apresentar grandes benefícios de desempenho e flexibilidade arquitetural para programas que demandam altíssimo desempenho. Para melhor entender os *softwares* que se beneficiam dos aspectos do ECS, convém olhar projetos e produtos reais que o utilizam.

A *Unity Game Engine* [Unity-Technologies 2024b], popular motor de jogo, principalmente entre os jogos independentes, lançou em 2019 o *Unity Data Oriented Technology Stack* (Pilha de Tecnologia Orientada a Dados da Unity), com o objetivo de apresentar soluções orientadas a dados para os usuários, garantindo melhor desempenho para futuros projetos. Entre as soluções apresentadas, estão: o *burst compiler*, que compila o *bytecode* gerado do C# para código nativo altamente otimizado; o sistema de trabalhos, que permite melhor paralelização dos *scripts* desenvolvidos, e a própria implementação de ***Entity Component Systems*** da *Unity*, o ***Unity* ECS**.

O *Unity* ECS é implementado com base nos arquétipos (Seção 4.2) que são organizados em blocos contíguos de memória definidos pelo objeto *ArchetypeChunk*, onde um arquétipo pode possuir um ou mais *chunks*. Além disso, a *Unity* estende grandemente o escopo de ECS, permitindo sua integração com os sistemas de paralelização de baixo nível do motor *Unity*, e associa outros significados aos conceitos básicos do ECS, como permitir diferentes interfaces para componentes – como os *IComponentData*, *IBufferElementData*, *ISharedComponentData*, entre vários outros –, a fim de otimizar ainda mais o desempenho dos jogos e simulações desenvolvidos nela.

Citado previamente na Seção 4.6, o jogo de tiro em primeira pessoa *Overwatch*, também foi desenvolvido com um ECS proprietário, focado em flexibilidade arquitetural e velocidade de desenvolvimento por parte da equipe de programadores [Ford 2017].

O *Minecraft*, também utiliza o ECS em sua versão "*Bedrock Edition*" através da biblioteca *EnTT* (a mesma utilizada para implementação dos códigos de teste deste trabalho), para administrar o grande número de entidades presentes no seu mundo aberto e gerado proceduralmente, ainda garantindo boa performance aos jogadores e facilidade de programação aos desenvolvedores [Mojang-Studios 2024].

A aplicação do ECS não se limita apenas a desenvolvimento de jogos. Em um artigo publicado em 2023 por Tianshi Cheng, Tong Duan e Venkata Dinavahi [Tianshi Cheng 2023], descreve-se a construção de uma plataforma de simulação de Sistemas Ciberfísicos de Energia, com enfoque para pesquisas a fim de evoluir a efetividade dos sistemas elétricos existentes. Esta plataforma foi completamente desenvolvida com base em um ECS, dando nome à plataforma: *ECS-Grid* [Tianshi Cheng 2023]. As motivações principais pela escolha pelo uso do ECS foram a alta flexibilidade e extensibilidade do código, onde os dados da simulação podem ser utilizados de diversas formas, sem estarem ligados a uma classe ou interfaces rígidas.

6. Comparação experimental de desempenho entre Orientação a Objetos e ECS

Para entender de forma prática a diferença de performance e arquitetural, foram criadas duas simples simulações em tempo real, cada uma com duas versões, uma tradicionalmente orientada a objetos e outra utilizando a biblioteca de *Entity Component Sys-*

tems chamada *EnTT*, baseada em conjuntos esparsos, explicados na Seção 4.3. Ambas simulações implementam exatamente os mesmos algoritmos e lógicas, sendo a única diferença entre elas puramente arquitetural e quanto ao arranjo de memória discutidos anteriormente.

O primeiro exemplo foi desenvolvido em *C++* com o auxílio da biblioteca *Raylib*, que facilita o desenvolvimento de jogos para múltiplas plataformas e *APIs* gráficas. Ele consiste em uma simples simulação de partículas que são atraídas a um ponto em um espaço bidimensional, representado pela posição do cursor. O algoritmo obtém o vetor de distância entre a partícula e o ponto, então utiliza a distância para calcular a velocidade a ser atribuída ao componente. Após isso, a posição da partícula é atualizada de acordo com a velocidade calculada previamente.

Em ambas as versões, são definidos três tipos de componentes: de física, de posição e de cor. Os dois primeiros são utilizados na movimentação da partícula no espaço bidimensional, o último é utilizado no processo de renderização, para permitir melhor visualização de cada entidade durante a simulação.

Na versão orientada a objetos, foi criada a classe *ParticleEntity*, que representa uma única partícula da simulação. A declaração desta classe pode ser vista na Figura 12, com métodos padrões para garantir o encapsulamento. A lógica responsável pela atualização do estado interno das partículas está no método *Update*, como visto na Figura 13. De modo classicamente orientado a objetos, cada partícula gerencia seu estado interno e nada mais, qualquer informação necessária a ser passada ou obtida é feita por meio do envio de mensagens.

Já na versão baseada em ECS, tem-se os mesmos componentes, mas não existe o conceito de um objeto partícula, e sim uma classe *ParticleScene*, que implementa um cenário de jogo, no qual no método *update*, são executados os sistemas de simulação e de renderização. Desta forma, os dados de cada entidade estão guardados internamente pelo *EnTT* e, pela funcionalidade do agrupamento (Seção 4.5), estes dados são acessados e atualizados com o mesmo algoritmo visto na Figura 13.

Para a execução do software, é necessário, na linha de comando, a passagem do parâmetro `-entity <numero>` com a quantidade de partículas que serão simuladas. Neste trabalho, **o primeiro teste de performance** será a medição da média de *frametime* (tempo médio da duração do laço principal) durante sessenta segundos, iniciando com dez mil entidades com incrementos de trinta mil a cada teste. Além disso, foi implementado um "modo puro de CPU", que elimina as chamadas ao renderizador 2D, tirando qualquer perda de desempenho por parte de chamadas gráficas à placa de vídeo.

Todos os testes foram realizados utilizando um computador com as seguintes características:

- Processador: AMD Ryzen 5 5600 com clock base de 3.5 GHz. 6 núcleos e 12 *threads*. Cache L1 de 384 KB; Cache L2 de 3MB e Cache L3 de 32 MB;
- GPU: NVIDIA Geforce GTX 1060 com 3GB de *VRAM* (memória RAM de vídeo);
- 24GB de memória RAM principal, com 3200 MHz.

O *software* utilizado para medir as médias de *frametime* se chama "*CapFrameX*", frequentemente utilizado para *benchmarks* de jogos e simulações. Ademais, todos os

```
class ParticleEntity : public GameObject {
 public:

  ParticleEntity(const std::string& name, bool isStar);
  ~ParticleEntity();

  void update(float delta);
  void draw();

  const std::string& getName() const { return m_Name; }

  void setMousePosition(Vector2 mousePosition) {
    m_MousePosition = mousePosition;
  }

  Color getColor(void) const { return m_Color; }
  void setColor(Color newColor);

  PositionComponent getPosition() const;
  void setPosition(Vector2 newPosition);

  PhysicsComponent getPhysics() const;
  void setVelocity(Vector2 newVelocity);

 private:
  std::string m_Name;

  ColorComponent m_Color;
  PhysicsComponent m_Physics = {0};
  PositionComponent m_Position = {0};

  Vector2 m_MousePosition = {0};
};
```

Figura 12. Declaração em C++ da classe *ParticleEntity*.

```
void ParticleEntity::update(float delta) {
  Vector2 differenceVector;

  differenceVector.x = m_Position.x - m_MousePosition.x;
  differenceVector.y = m_Position.y - m_MousePosition.y;

  float dist = Vector2Distance({m_Position.x, m_Position.y}, m_MousePosition);

  if (dist == 0.0f) dist = 1;
  Vector2 normal = {differenceVector.x * (1 / dist),
                    differenceVector.y * (1 / dist)};

  dist = fmax(dist, 0.5);
  m_Physics.velocity.x -= normal.x / dist;
  m_Physics.velocity.y -= normal.y / dist;

  m_Physics.velocity.x *= 0.99;
  m_Physics.velocity.y *= 0.99;

  m_Position.x += m_Physics.velocity.x * 100 * delta;
  m_Position.y += m_Physics.velocity.y * 100 * delta;

  auto screenWidth = GetScreenWidth();
  auto screenHeight = GetScreenHeight();

  if (m_Position.x < 0) m_Position.x += screenWidth;
  if (m_Position.x >= screenWidth) m_Position.x -= screenWidth;

  if (m_Position.y < 0) m_Position.y += screenHeight;
  if (m_Position.y >= screenHeight) m_Position.y -= screenHeight;
}
```

Figura 13. Implementação do método *update* da classe *ParticleEntity*.

Entidades	*Frametime*	Subida de *Frametime*
10.000	1,3 *ms*	—
40.000	4,7 *ms*	3,4
70.000	8,1 *ms*	3,4
100.000	11,6 *ms*	3,5
130.000	14,9 *ms*	3,3
160.000	18,3 *ms*	3,4
190.000	21,6 *ms*	3,3
220.000	24,9 *ms*	3,3
250.000	28,4 *ms*	3,5
280.000	31,5 *ms*	3,1
310.000	35 *ms*	3,5
—	**Subida média:**	3,37

Tabela 9. Média de *frametime* com diferentes quantidades de entidades na simulação orientada a objetos.

Entidades	*Frametime*	Subida de *Frametime*
10.000	0,4 *ms*	—
40.000	0,9 *ms*	0,5
70.000	1,5 *ms*	0,6
100.000	2 *ms*	0,5
130.000	2,5 *ms*	0,5
160.000	3,1 *ms*	0,6
190.000	3,7 *ms*	0,6
220.000	4,2 *ms*	0,5
250.000	4,8 *ms*	0,6
280.000	5,3 *ms*	0,5
310.000	5,9 *ms*	0,6
—	**Subida média:**	0,55

Tabela 10. Média de *frametime* com diferentes quantidades de entidades na simulação baseada em ECS.

testes foram realizados no modo de "*Pure CPU*", isto é, sem chamadas às *APIs* gráficas, de modo a medir apenas o trabalho efetuado pelo processador, que é onde o ECS atua.

Sobre a métrica de *frametime*, é fundamental entender que as plataformas de jogos e os clientes possuem requisitos diferentes: jogos "cinematográficos" de alta fidelidade gráfica geralmente focam em conseguir uma média de $33{,}3$ milissegundos por iteração da simulação. Jogos competitivos de sucesso como *Counter-Strike 2* e *Valorant*, que movimentam dezenas de milhões de dólares por ano, focam no maior desempenho possível, onde os jogadores profissionais visam números abaixo de 2 milissegundos por quadro. Jogos de realidade virtual, por sua natureza de "enganar" o cérebro do jogador, necessitam de máximo de 10 milissegundos por quadro a fim de evitar *motion sickness*. Desta forma, é visto que boa parte dos jogos possuem a necessidade de trabalhar com tempos de execução baixíssimos e o melhor desempenho possível.

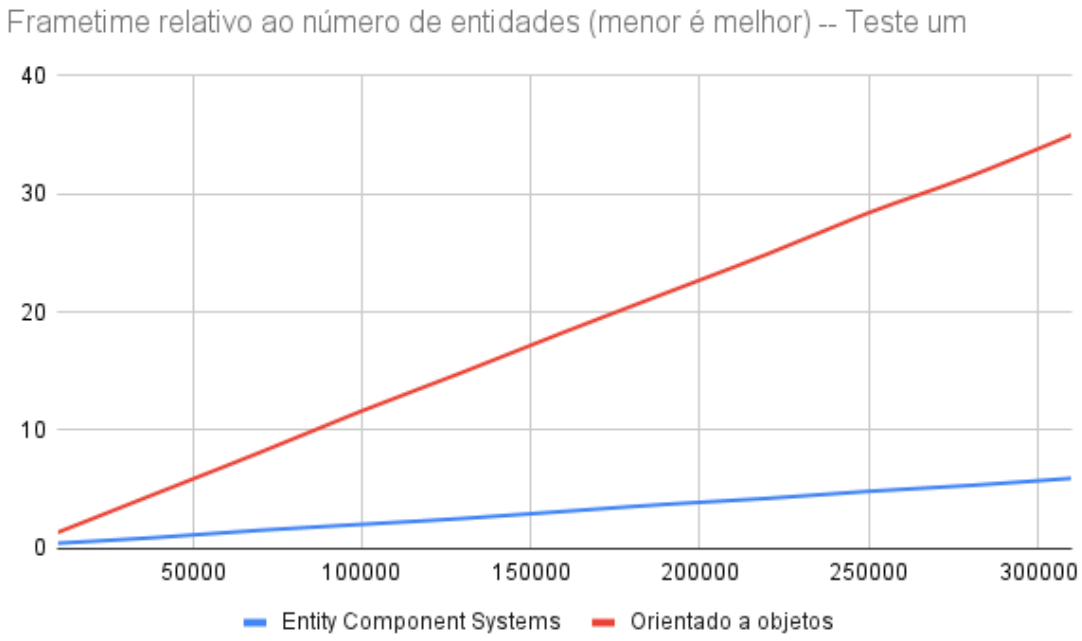

Figura 14. Diferenças de *frametime* da primeira simulação entre as implementações OO e ECS.

Em análises comerciais de desempenho, é muito comum a utilização da métrica de quadros por segundo (QPS), que define a quantidade de iterações do laço principal de simulação que uma aplicação consegue realizar em um segundo. Contudo, ela não será considerada uma métrica relevante neste trabalho devido a ser uma medida inversa ao *frametime*. Este fato acarreta não-linearidade de gráficos semelhantes a funções matemáticas $\frac{1}{X}$, mesmo em situações onde a perca no desempenho é linear e constante.

Especificamente sobre a versão orientada a objetos (Tabela 9), inicia-se com o *frametime* de $1{,}3$ milissegundos e há subida quase constante, na média de $3{,}4$ milissegundos, finalizando com 35 milissegundos para o último teste com 310.000 entidades. Mesmo com o alto número de entidades, 35 milissegundos já não atenderia aos requisitos de jogos de alta fidelidade gráfica ($33{,}3$ milissegundos), ainda mais sendo apenas uma simulação simples de partículas.

Quanto aos números de *frametime* na versão ECS (Tabela 10), já se inicia com desempenho três vezes mais rápido comparado à orientada a objetos. Inicia-se com $0{,}4$ milissegundos de *frametime* e sobe-se para $0{,}9$ a partir do primeiro incremento de 30.000 entidades. Há subida média de $0{,}55$ a cada incremento, isto é, pode-se afirmar que, a cada adição de 30.000 entidades à simulação, cada laço torna-se apenas cerca de 550 microssegundos mais demorado. O último teste finaliza com $5{,}9$ milissegundos para 310.000 entidades. Algo que deve ser salientado é a pouca perda de desempenho para a adição de um número expressivo de entidades a cada incremento, desta forma, numa situação hipotética de desenvolvimento de um jogo com data de lançamento definida, os desenvolvedores poderiam adicionar novos recursos sem grande preocupação de não atender às demandas de performance para as plataformas alvo (discutidas previamente).

Ao observar a Figura 14 que apresenta um gráfico com as medidas de *frametime* de ambas implementações, nota-se que as duas exibem comportamento de crescimento linear, contudo, a versão orientada a objetos possui maior inclinação, indicando que a perda de desempenho relacionada ao incremento de dados a serem processados é ruim quando comparada à versão ECS.

Para o segundo experimento, foi criada uma simples simulação tridimensional utilizando o motor de jogo *Unity*. Assim como citado na Seção 5, a *Unity* provê dois tipos de paradigmas de desenvolvimento: o padrão, baseado em "*GameObjects*", – que possuem seus comportamentos descritos por componentes e se comunicam através do envio de mensagens entre os objetos de jogo –, e a ***Unity ECS***, que permite a utilização dos *Entity Component Systems* de forma oficial no motor de jogo. Portanto, assim como no experimento anterior, o mesmo jogo foi desenvolvido separadamente utilizando ambas as tecnologias, de forma a entender e medir as diferenças de desempenho, flexibilidade arquitetural. A simulação desenvolvida se baseia em três tipos de objetos que se movimentam em formas diferentes pelo espaço, existindo milhares de instâncias de cada um deles. Neste exemplo, os objetos definidos são:

- Cubos: objetos que aumentam linearmente seu valor de escala até um certo limiar superior aleatório no intervalo de $[10, 12]$ e, então, diminuem até um limiar inferior aleatório no intervalo de $[0.1, 1]$, e o processo é repetido indefinidamente. Desta forma, os cubos da simulação apresentam diferenças visuais, mesmo estando executando o mesmo método `Update`;
- Esferas: objetos que circulam um certo ponto no espaço que é determinado aleatoriamente no processo de criação do objeto. Além disso, é definido um raio aleatório e uma velocidade de rotação aleatória, de forma a garantir diferenças visuais e de simulação entre os dados de cada uma das esferas;
- Cilindros: são uma mistura entre os dois tipos anteriores, pois alteram sua própria escala e rotacionam em torno de um ponto aleatório do espaço.

O primeiro desafio na implementação tradicional deste jogo foi a instanciação do grande número de objetos de jogo. A *Unity*, por ser um motor de jogo onde as funcionalidades são implementadas em *scripts* C# com *garbage collection*, não é veloz nas operações de `Instantiate` e `Destroy`, portanto, é necessário implementar a técnica de otimização chamada *Object Pooling* [Nystrom 2011], na qual todos os objetos são alocados em uma lista durante a inicialização do jogo e apenas são marcados como ativados e desativados durante o tempo de execução, sem a necessidade do motor realizar operações custosas de memória. Para a criação de objetos de jogos na *Unity*, há o conceito de *prefabs*: moldes prontos de um *GameObject*, que pode gerar instâncias independentes em tempo de execução.

Para a inicialização do jogo, é criado um objeto `GameManager`, que se encarrega do processo de enviar mensagens às *pools* de objetos. Antes da execução, é definido um valor `amountToSpawn` que determina o total de objetos a serem simulados. O método `Start()` executa três laços, cada um instanciando $\frac{amountToSpawn}{3}$ objetos de cada tipo.

Para a simulação, o comportamento de alteração de escala (para cubos e cilindros) funciona apenas somando e subtraindo uma pequena constante a cada passo da atualização de física (a implementação deste comportamento pode ser vista na Figura 15). Já o comportamento de rotação em torno de um eixo (para esferas e cilindros), funci-

```
public class ScalingCube : MonoBehaviour
{
    private bool scalingUp = true;
    private float lowerThreshold = 10f;
    private float upperThreshold = 10f;
    void Start()
    {
        float x = Random.Range(-120f, 120f);
        float z = Random.Range(-120f, 120f);
        lowerThreshold = Random.Range(0.1f, 1f);
        upperThreshold = Random.Range(2f, 12f);
        gameObject.transform.position = new Vector3(x, 2, z);
    }
    void Update()
    {
        if (transform.localScale.x > upperThreshold)
        {
            scalingUp = false;
        }
        else if (transform.localScale.x ⩽ lowerThreshold)
        {
            scalingUp = true;
        }
        if (scalingUp)
        {
            float currentScale = gameObject.transform.localScale.x + 0.05f;
            gameObject.transform.localScale = new Vector3 ( currentScale, currentScale, currentScale );
        } else
        {
            float currentScale = gameObject.transform.localScale.x - 0.05f;
            gameObject.transform.localScale = new Vector3 ( currentScale, currentScale, currentScale );
        }
    }
}
```

Figura 15. *Script* referente aos cubos na implementação tradicional da *Unity*.

ona atualizando as posições nos eixos X e Z de acordo com as equações trigonométricas de um caminho circular:

$$x = \text{centroX} + \cos(\theta) \cdot R$$
$$z = \text{centroZ} + \sin(\theta) \cdot R$$

onde:

- θ é o ângulo em graus;
- R é o raio;
- centroX é o ponto no eixo X no qual o objeto irá circular;
- centroZ é o ponto no eixo Z no qual o objeto irá circular.

A implementação do comportamento descrito é ilustrada na Figura 16.

```
public class CircularSphere : MonoBehaviour
{
    // Start is called before the first frame update
    public float targetX, targetZ;
    public float radius = 2f;
    public float speed = 1f;
    private float angle = 0f;
    void Start()
    {
        radius = Random.Range(5f, 15f);
        speed = Random.Range(2f, 20f);
        targetX = Random.Range(-120f, 120f);
        targetZ = Random.Range(-120f, 120f);
    }

    // Update is called once per frame
    void Update()
    {
        float x = targetX + Mathf.Cos(angle) * radius;
        float y = gameObject.transform.position.y;
        float z = targetZ + Mathf.Sin(angle) * radius;

        gameObject.transform.position = new Vector3(x, y, z);
        angle += speed * Time.deltaTime;
    }
}
```

Figura 16. *Script* referente às esferas na implementação tradicional da *Unity*.

Em contrapartida ao fluxo de trabalho tradicional, na implementação baseada em ECS, não se pensa em objetos individuais e separados que são donos de seus próprios dados (Seção 4.1). Para criar um ambiente ECS na *Unity*, é necessário instalar pacotes referentes às técnicas orientadas a dados, como o *Unity Entities* e *Unity Entities Graphics*. O fluxo de trabalho com essas tecnologias demanda o trabalho com o processo de *Baking* (como visto na Figura 17), que transforma os objetos de jogo tradicionais em entidades de ECS [Unity-Technologies 2023a] e os componentes tradicionais são agrupados em *chunks* baseados nos arquétipos existentes.

No *Unity* ECS, existem diversos tipos de componentes para serem utilizados:

- **Não-gerenciados**: herdam de *IComponentData* e podem armazenar apenas alguns tipos de dados. Mesmo com essas restrições, são o tipo de componente mais comum e rápido, por ser implementado com *struct* e, assim, ser alocado na pilha;
- **Gerenciados**: também herdam de *IComponentData*, mas são alocados na *heap* por serem classes. Eles podem armazenar qualquer tipo de dado, mas são mais custosos de operar;
- **Compartilhados**: agrupam entidades em blocos homogêneos de forma a eliminar a duplicação de dados;
- **De limpeza**: são componentes que continuam a existir mesmo após a chamada de *Destroy* em uma entidade, para realizar algum processo necessário de limpeza antes dos dados serem de fato apagados;
- **Marcação**: são componentes sem nenhum dado, que servem para a marcação de uma entidade como sendo de algum tipo;
- ***Buffer***: componentes que agem como um vetor de tamanho variável, para quando uma entidade necessita acessar dados que mudam dinamicamente durante a execução de um jogo, como pontos no mundo para o cálculo de algum sistema;
- ***Chunk***: componentes que armazenam valores por bloco em vez de por entidade. Existem com o propósito de otimizar operações;
- **Ativáveis**: componentes que podem ser desativados ou ativados em tempo de execução. Úteis para trabalho com estados mutáveis;
- ***Singleton***: componentes que apenas possuem uma única instância.

Para a implementação da simulação dos objetos mencionados previamente são utilizados dois componentes não-gerenciados "*ScalingComponent*" e "*RotatingComponent*", que contêm as informações necessárias para as operações de alteração de escala e da movimentação em torno de um eixo, respectivamente.

Quanto aos sistemas da *Unity ECS*, existem principalmente dois tipos:

- ***ISystem***: sistemas não-gerenciados em *structs*, desta forma só podendo atuar sobre tipos limitados de componentes. O maior diferencial é a compatibilidade com o compilador *Burst* da *Unity DOTS*, que permite compilação dos sistemas para código nativo e agendamento para execução paralela.
- ***SystemBase***: sistemas gerenciados em *classes*, que possuem certas funcionalidades a mais, mas não podem ser compilados em código nativo pelo compilador *Burst*.

Existem várias formas de acessar os dados da simulação a partir dos sistemas. Neste exemplo em específico, serão utilizados simples *foreaches* a partir de *queries* (isto é, consultas) ao objeto *SystemAPI* oferecido pelo *Unity ECS*. *A SystemAPI* é uma classe

Entidades	*Frametime*	Subida de *Frametime*
5.000	5,6 *ms*	—
10.000	12,1 *ms*	6,5
15.000	21 *ms*	8,9
20.000	29,4 *ms*	8,4
25.000	37,3 *ms*	7,9
30.000	45 *ms*	7,7
35.000	53,9 *ms*	8,9
40.000	61,2 *ms*	7,3
—	**Subida média:**	7,94

Tabela 11. Médias de *frametime* com diferentes quantidades de entidades na simulação orientada a objetos utilizando a *Unity* tradicional.

que provê métodos para garantir o acesso aos dados do mundo simulado, sendo todos os tipos de componentes, entidades, *EntityCommandBuffers*, etc.

Uma consulta realizada através da *SystemAPI* retorna uma coleção de dados que podem ser iterados, o que permite operar apenas pelos dados necessários, assim como na funcionalidade de agrupamento provida pela *EnTT* (Seção 4.5).

A fim de realizar o processo de inicialização, é criada uma entidade chamada "*ObjectSpawner*", que tem como papel realizar a operação de criar as entidades, assim como o objeto "*GameManager*" da implementação tradicional. Para isso, precisa-se realizar o processo de *baking*, criando um *script* "*ObjectSpawnerAuthoring*" que associa a entidade ao componente *singleton* "*ObjectSpawner*" que contém os *prefabs* dos objetos e as dimensões da câmera.

Para a inicialização da simulação, existe um sistema gerenciado *ObjectSpawnerSystem*, que é executado apenas uma vez. Ele instancia um objeto *EntityCommandBuffer* provido pela *API* de ECS da *Unity* que registra comandos a serem executados em lotes, de forma a minimizar operações custosas de memória.

As entidades são criadas enviando mensagens `CreateEntity` para o *EntityCommandBuffer* e os componentes são adicionados pela mensagem `AddComponent`.

O processo de criação de entidades é semelhante à versão tradicional: são feitos três laços que criam separadamente os três tipos de entidades. Aos cubos é atribuído o componente *ScalingComponent*, à esfera é atribuída o componente *RotatingComponent* e aos cilindros são adicionados os dois.

Para os passos de simulação em si, são definidos dois sistemas não-gerenciados (*ISystem*) chamados *ScalingSystem* e *RotatingSystem* (implementações vistas nas Figuras 18 e 19, respectivamente). O primeiro realiza uma consulta que retorna todos os componentes de escala, verifica se o valor deve ser incrementado ou decrementado de acordo com o limiar aleatório gerado e, então, modifica a variável de transformação da entidade. O sistema de rotação realiza a consulta e apenas calcula a nova posição da entidade de acordo com um sistema de coordenadas circular, depois atualiza a variável de angulação de acordo com a velocidade aleatória determinada no início da aplicação.

```
class CubeAuthoring : MonoBehaviour
{
    public float lowerThreshold;
    public float upperThreshold;
    bool scalingUp;

    class CubeAuthoringBaker : Baker<CubeAuthoring>
    {
        public override void Bake(CubeAuthoring authoring)
        {
            float lt = Random.Range(0.1f, 1.0f);
            float ut = Random.Range(2f, 12f);
            Entity entity = GetEntity(TransformUsageFlags.Dynamic);
            AddComponent(entity, new ScalingComponent { lowerThreshold = lt, upperThreshold = ut, scalingUp = true });
        }
    }

}

public struct ScalingComponent : IComponentData
{
    public float lowerThreshold;
    public float upperThreshold;
    public bool scalingUp;
};
```

Figura 17. Código de *Baking* no *Unity ECS* para os cubos e o componente de escala.

```
public partial struct ScalingSystem : ISystem
{
  void OnUpdate(ref SystemState state)
  {
    foreach((RefRW<ScalingComponent> scaling, RefRW<LocalTransform> transform) in SystemAPI.Query<RefRW<ScalingComponent>, RefRW<LocalTransform>>())
    {
      if (transform.ValueRO.Scale > scaling.ValueRO.upperThreshold)
      {
        scaling.ValueRW.scalingUp = false;
      }
      else if (transform.ValueRO.Scale ≤ scaling.ValueRO.lowerThreshold)
      {
        scaling.ValueRW.scalingUp = true;
      }
      if (scaling.ValueRO.scalingUp)
      {
        float currentScale = transform.ValueRO.Scale + 0.05f;
        transform.ValueRW.Scale = currentScale;
      }
      else
      {
        float currentScale = transform.ValueRO.Scale - 0.05f;
        transform.ValueRW.Scale = currentScale;
      }
    }
  }
}
```

Figura 18. Sistema para simulação de todas as entidades que tenham o componente de escala.

```
public partial struct RotatingSystem : ISystem
{
  void OnUpdate(ref SystemState state)
  {
    foreach(var (rotating, transform) in SystemAPI.Query<RefRW<RotatingComponent>, RefRW<LocalTransform>>())
    {
      float x = rotating.ValueRO.targetX + Mathf.Cos(rotating.ValueRO.angle) * rotating.ValueRO.radius;
      float y = transform.ValueRO.Position.y;
      float z = rotating.ValueRO.targetZ + Mathf.Sin(rotating.ValueRO.angle) * rotating.ValueRO.radius;

      transform.ValueRW.Position = new Vector3(x, y, z);
      rotating.ValueRW.angle += rotating.ValueRO.speed * SystemAPI.Time.DeltaTime;
    }
  }
}

public struct RotatingComponent : IComponentData
{
  public float targetX, targetZ;
  public float radius;
  public float speed;
  public float angle;
}
```

Figura 19. Sistema para simulação de todas as entidades que tenham o componente de rotação.

Entidades	*Frametime*	Subida de *Frametime*
5.000	4,6 *ms*	—
10.000	6 *ms*	1,4
15.000	7,3 *ms*	1,3
20.000	8,6 *ms*	1,3
25.000	9,6 *ms*	1
30.000	10,6 *ms*	1
35.000	11,7 *ms*	1,1
40.000	12,9 *ms*	1,2
—	**Subida média:**	1,18

Tabela 12. Médias de *frametime* com diferentes quantidades de entidades na simulação realizada com a *Unity* ECS.

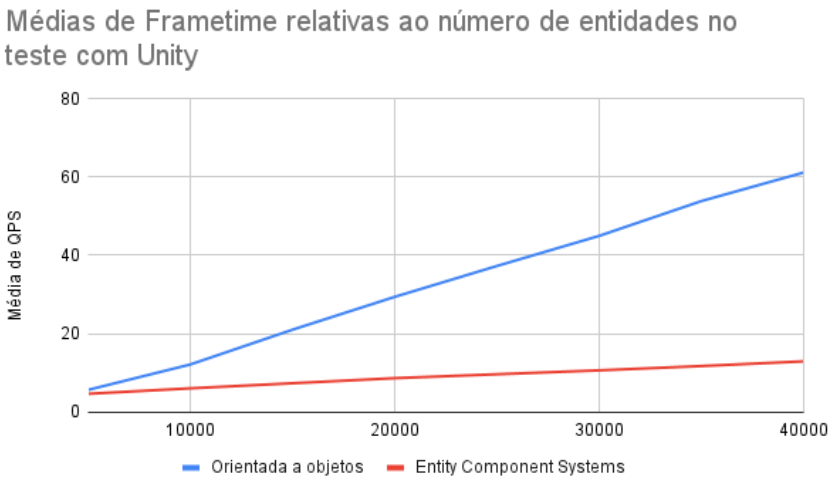

Figura 20. Diferenças *frametime* da simulação utilizando a *Unity* entre a versão tradicional e ECS. Menor é melhor.

O teste realizado para medição e comparação do desempenho computacional entre as duas implementações é similar ao primeiro experimento: a aplicação é iniciada com número bem definido de entidades e as médias de *frametime* são registrados por um minuto com a ferramenta "*CapFrameX*". A cada teste aumenta-se em 5.000 o número de entidades; iniciando com 5.000 e finalizando com 40.000.

Para os testes utilizando a versão tradicional da *Unity* (Tabela 11), inicia-se a simulação com 5.000 entidades com $5{,}6$ milissegundos de *frametime* de média durante a execução de um minuto. Na segunda etapa, com 10.000 entidades, há subida de de $6{,}5$ milissegundos, obtendo a média de $12{,}1$ milissegundos durante a execução. Algo importante a ser evidenciado, são os *frametimes* acima de $33{,}3$ milissegundos já a partir das 25.000 entidades, o que não é desejado para produtos comerciais, visto que esse limiar de tempo é visto como o mínimo mesmo para jogos de alta fidelidade gráfica e carga computacional. Os testes são finalizados com a média de $61{,}2$ milissegundos com 40.000 entidades, um número altíssimo para a execução de uma simulação trivial, mesmo que com quantidade expressiva de entidades.

Já sobre os testes utilizando o *Unity ECS*, inicia-se a primeira simulação com média de $4{,}6$ milissegundos, 1 milissegundo a menos que a versão tradicional. Para o primeiro incremento, há incremento de $1{,}4$ milissegundos, atingindo o número de 6 *ms*. O experimento é finalizado com a execução da simulação com as 40.000 entidades atingindo uma média de *frametime* de $12{,}9$, com uma subida média de $1{,}18$ milissegundos. É bom também perceber que mesmo com 40.000 entidades ainda se obtém boas métricas para jogos de alto desempenho, que demandam *frametimes* menores que $16{,}6$ milissegundos.

Com isso, é visível a diferença expressiva de desempenho entre as simulações que realizam a mesma tarefa, apenas com diferentes abordagens de acesso efetivo à memória do computador. A Figura 20 evidencia a diferença e reforça o comportamento visto na Figura 14 entre ambas estratégias, quando observada a medida de *frametime*: as duas apresentam comportamento linear, contudo, a versão baseada em ECS tem menor inclinação de crescimento, ou seja, garante bons números de desempenho mesmo com a simulação

de muitas entidades simultaneamente.

7. Conclusão

Após todas as informações apresentadas, nota-se que o modelo ECS é uma alternativa interessante para os desenvolvedores de sistemas de tempo-real que precisam ativamente considerar as restrições de desempenho do produto final enquanto simultaneamente necessita-se de alta flexibilidade arquitetural, principalmente em ambientes de desenvolvimento em que as rápidas iterações e mudanças de projeto são fundamentais. Ademais, a adoção deste paradigma por equipes importantes da indústria de jogos eletrônicos demonstra a robustez dos conceitos apresentados e a capacidade de gerar produtos sólidos e de sucesso, garantindo bom desempenho e rapidez nos processos de desenvolvimento mesmo em jogos de propostas ambiciosas e que necessitam da comunicação da simulação do estado interno para com servidores, como o *Overwatch*. É, contudo, importante ressaltar que as soluções disponíveis ao público ainda são relativamente complicadas e necessitam de boa compreensão por parte do programador do fundamento do problema solucionado pelos ECS e entender se é realmente a decisão correta para o projeto. O *Unity* ECS, por exemplo, por ser algo novo, é uma ferramenta ainda considerada suplementar pela comunidade de desenvolvedores, cujo uso é estritamente para processamento de grande número de entidades em situações especificas. Ou seja, há a necessidade de conciliar os dois paradigmas (tradicional e ECS) no desenvolvimento de um grande projeto *Unity*.

Demais soluções como *Flecs* e *EnTT* possuem vários recursos para a produção de jogos e simulações, porém, são focados para projetos onde a equipe de desenvolvimento seja capaz de criar seu próprio motor de jogo, algo que não é comum entre pequenas e médias equipes, onde a utilização de motores prontos como a *Unity* e *Godot* são necessários para diminuir o tempo de desenvolvimento, por providenciarem grande variedade de tecnologias robustas focadas em jogos e simulações.

Neste trabalho foi apresentado apenas o cerne do que compõe um sistema ECS, contudo, conforme o mercado de tecnologia caminha cada vez mais a soluções paralelas, exploram-se soluções *multithreading* utilizando este paradigma, como o *Job System* da *Unity* [Unity-Technologies 2023b]. Pela natureza particular de manter sistemas independentes para funcionamento, em simulações onde estes são muitos, há a possibilidade de utilizar GPUs para acelerar estes processos. Assim, o desenvolvimento de um ECS acelerado por GPU é um excelente candidato para trabalhos futuros e implementações novas que combinam os conceitos já estudados de computação paralela com a lógica de operações utilizadas por jogos e demais simulações de tempo-real.

Referências

Al-Waisi, Z. and Agyeman, M. O. (2017). An overview of on-chip cache coherence protocols. *Intelligent Systems Conference 2017*.

Briggs, P. and Torczon, L. (1993). An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*.

Caini, M. (2019). ECS back and forth. part 2, insights. `https://skypjack.github.io/2019-03-21-ecs-baf-part-2-insights/`. [online: acesso em 01-Maio-2024].

Caini, M. (2024). EnTT code repository. `https://github.com/skypjack/entt`. [online: acesso em 30-Maio-2024].

Drepper, U. (2007). What every programmer should know about memory. *Red Hat Inc.*

Fabian, R. (2018). *Data-Oriented Design*.

Ford, T. (2017). Overwatch gameplay architecture and netcode. In *Game Developers Conference*, São Francisco, EUA.

Foster, I. (1995). *Designing and Building Parallel Programs*. Ian Foster.

Gregory, J. (2018). *Game Engine Architecture*. A K PETERS.

Henessy, J. L. and Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach 6th Edition*. Morgan Kaufmann.

Kirmse, A. (2002). Optimization for c++ games. In *Game Programming Gems 2*. Charles River Media.

Lengyel, E. (2016). *Game Engine Gems 3*. CRC Press.

Mertens, S. (2020). Building an ECS 2: Archetypes and vectorization. `https://ajmmertens.medium.com/building-an-ecs-2-archetypes-and-vectorization-fe21690805f9`. [online: acesso em 02-Maio-2024].

Mertens, S. (2024). Flecs code repository. `https://github.com/SanderMertens/flecs`. [online: acesso em 03-Junho-2024].

Mojang-Studios (2024). Minecraft attributions. `https://www.minecraft.net/en-us/attribution`. [online: acesso em 03-Junho-2024].

Nafack, B. (2023). Analyze the performance of dense matrix transposition: Row-major and column-major access in C++.

Nystrom, R. (2011). *Game Programming Patterns*. Genever Benning.

Romeo, V. (2016). Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time entity-component-system C++14 library.

Romeo, V. (2017). ECST code repository. `https://github.com/vittorioromeo/ecst`. [online: acesso em 05-Junho-2024].

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*.

Tianshi Cheng, Tong Duan, V. D. (2023). ECS-Grid: Data-oriented real-time simulation platform for cyber-physical power systems. *IEEE Transactions on Industrial Informatics, Volume 19, Edição 11*.

Unity-Technologies (2023a). Entities Overview. `https://docs.unity3d.com/Packages/com.unity.entities@1.3/manual/index.html`. [online: acesso em 17-Junho-2024].

Unity-Technologies (2023b). Job System Manual. `https://docs.unity3d.com/Manual/JobSystem.html`. [online: acesso em 27-Agosto-2024].

Unity-Technologies (2024a). Archetypes concepts. `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-archetypes.html`. [online: acesso em 07-Maio-2024].

Unity-Technologies (2024b). Website. `https://unity.com/`. [online: acesso em 29-Maio-2024].