Ravine ECS, uma arquitetura de alto desempenho

Gabriel Lanzer Kannenberg* Fernando Marson

Universidade do Vale do Rio dos Sinos, Jogos Digitais, Brasil

RESUMO

O presente artigo descreve o desenvolvimento da Ravine ECS, uma implementação da arquitetura Entity Component System que utiliza de princípios do paradigma Data-Oriented Design (DOD) para avançar no estado da arte com a elaboração de um esquema de armazenamento e iteração de componentes de alto desempenho. Esses resultados são comprovados por testes de performance que comparam esse trabalho com uma biblioteca cujo uso já está consolidado no mercado.

Palavras-chave: ECS, High-performance, Data-Oriented Design.

1 INTRODUÇÃO

Os avanços tecnológicos na construção de microprocessadores acabam incutindo um ganho anual significativo no desempenho das Unidades Centrais de Processamento (*Central Processing Units*, CPUs), como descrito pela Lei de Moore (Figura 1). Hoje, até mesmo dispositivos móveis portam diversos núcleos de processamento (*Cores*), que podem ser utilizados amplamente para tarefas que exigem alto desempenho. Em contrapartida, um avanço significativamente menor foi visto em termos de frequência e latência no acesso à memória principal RAM, cujo desenvolvimento de circuitos integrados busca atenuar tanto pelo meio de níveis de cache temporários (L1, L2 e L3) quanto por métodos de predição no acesso de memória (com o *prefetch* de hardware).

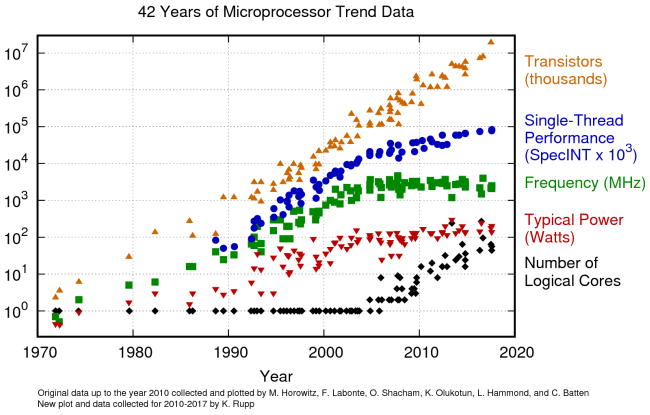

Figura 1: Exemplo da Lei de Moore. Fonte: Blog pessoal de K. Rupp (`https://bit.ly/2NU6Nec`)

De mesma forma, surgiram debates recentes sobre diversos paradigmas de *design* que se opõe. Dentre esses, destaca-se o *Design* Orientado a Objetos (*Object Oriented Design*, OOD)[8], pois facilita o entendimento e modelagem de problemas. Contudo, algumas das facilidades que a orientação a objetos traz, causam indireções que diminuem a performance da aplicação, por provocar baixo aproveitamento do cache de memória[1]. Isso acontece pela ocorrência de *cache-miss* no acesso de uma variável, pois quando o dado não se encontra em um nível de cache mais baixo, é necessário o acesso ao nível superior, o que causa uma latência no acesso.

Em reposta a isso, alguns conceitos antigos foram retomados e enfatizados por parte da comunidade C/C++. Estes evidenciam a organização e o acesso de memória como um ponto chave em aplicações de alta performance, o que caracterizou esse tipo de desenvolvimento como *Design* Orientado a Dados (*Data Oriented Design*, DOD)[1, 4, 6, 14]. Com base nessa vertente de ideias, conjecturou-se o princípio de que todo software atua em uma entrada de dados e a transforma em uma nova saída de dados.

De maneira semelhante, foi nos últimos anos que se consolidou a arquitetura *Entity Component System* (ECS), na qual é característica a distinção de: dados, denominados componentes; sistemas, que atuam transformando os componentes; e entidades, que são representações dos objetos do mundo real e caracterizadas pelos tipos de componentes as quais se associam.

Mediante esses conceitos, o presente artigo descreve a a criação e desenvolvimento da Ravine ECS[2], uma arquitetura ECS construída com base em valores defendidos pelo *Data Oriented Design*, acompanhando o estado da arte, mas também contribuindo com uma proposta em termos de organização de componentes.

1.1 Objetivo Geral

Criação de uma biblioteca ECS, com base nos princípios de design orientado à dados, juntamente da elaboração de um esquema de armazenamento com *layout* e iteração de componentes com coerência no acesso linear de memória, a fim de proporcionar alta performance com o uso eficiente de cache do processador.

1.2 Objetivos Específicos

Para atingir o objetivo geral deste projeto, é necessário alcançar e consolidar os seguintes objetivos específicos:

- Contextualização do trabalho.
- Arquitetura da biblioteca.
- Armazenamento e iteração de componentes.
- Otimizações aplicadas e possíveis.
- Validação dos resultados.

1.3 Estrutura do Artigo

Na Seção 2 são descritos os aspectos mais relevantes ao Estado da Arte a fim de contextualizar os elementos sobre os quais este projeto foi desenvolvido. São citados, também, outros trabalhos que possuem relação com o projeto desenvolvido. Já, na Seção 3 é definida a arquitetura ECS da Ravine juntamente com os diagramas associados. Na Seção 4 são ressaltados aspectos do processo de desenvolvimento, como as implementações da *Application Programming Interface* (API) e otimizações realizadas. Na Seção 5, são descritos testes de performance e casos de uso para a validação da

*e-mail: gabriellanzerlive@hotmail.com

[1] `https://bit.ly/31GQ2ef`

[2] Disponível em `https://bit.ly/2NNx39Q`

biblioteca desenvolvida, cujos resultados são descritos na Seção 6. Por fim, na Seção 7 são resumidos os produtos do trabalho, avanços no estado da arte, bem como pontos a serem melhorados (trabalhos futuros).

2 Estado da Arte e Trabalhos Relacionados

Diferente da orientação à objetos (OOD), que simplifica a modelagem de problemas relacionando classes aos objetos do mundo real, DOD tenta modelar os problemas da maneira mais adequada para organizar os dados na memória do sistema. Além da grande diferença de performance, ocasionada pela coerência de acesso ao *cache* do processador, esse método determina *buffers* com os dados que podem ser facilmente serializados em disco ou, por exemplo, transferidos para a *Graphics Processing Unit* (GPU). Além disso, essa estruturação permite o acesso paralelo aos dados de forma direta, por indexação, o que simplifica o seu uso em placas de vídeo ou em múltiplos processadores. Foi com base nesses pilares que foi consolidada a arquitetura ECS.

Mesmo tendo mais visibilidade atualmente com sua recente implementação na *Game Engine* Unity, ainda são escassos os casos de uso dessa arquitetura: ela foi utilizada em Thief: The Dark Project em 1998[12]; mais detalhada na GDC 2002 (*Game Developers Conference*) como uma abordagem em Dungeon Siege[5]; posteriormente, em 2007, a concepção da arquitetura ECS foi formalizada por sua utilização no jogo Operation Flashpoint: Dragon Rising, onde a nomenclatura foi melhor definida; por fim, na GDC 2017, ficou famosa com uma palestra sobre a implementação de Overwatch[7], contemplando um jogo AAA que obteve grande sucesso utilizando dessa arquitetura.

Vale ressaltar que, na data de produção deste artigo, existem poucas referências bibliográficas escritas sobre este assunto. Em contra-partida, são encontrados muitos *blogs*, vídeos e palestras debatendo essas temáticas ou demonstrando casos de uso e implementações da arquitetura. Por exemplo, na palestra do Overwatch[7] foram descritos o desenvolvimento do jogo e a implementação da arquitetura ECS. O jogo Spellsouls Universe, da empresa Nordeus utilizou a arquitetura ECS da Unity para criação de batalhas massivas[16].

Além da produção de jogos, outros trabalhos abordaram a modelagem de problemas específicos usando a arquitetura ECS como: a construção de uma estrutura *hash-map* que não utiliza travas de *thread*[11]; a elaboração de um *framework* para criação de interfaces gráficas e interações com a arquitetura ECS[15]; o uso de traços semânticos para evitar o acoplamento de subsistemas em implementações ECS seguindo o paradigma de programação orientada à objetos[17].

Abaixo são listados alguns trabalhos, cuja relação com o presente artigo é de grande importância. Todos são implementações da arquitetura ECS e não aplicações realizadas com ela. Alguns trabalhos ainda estão em estágio preliminar, mas todos contemplam diferentes soluções para os mesmos problemas:

- EnTT[3] - Biblioteca ECS, escrita em C++17, *header-only* e com licença MIT. É a implementação mais performática e completa até o momento, com bastante flexibilidade para customização em diferentes casos de uso e também possui um sistema de eventos integrado. É utilizada no jogo Minecraft, desenvolvido pela Mojang, e no *Runtime SDKs* do software ArcGIS, da empresa Esri.

- ECS[4] - Biblioteca ECS, escrita em C++11, *header-only* e com licença MIT. Possui sistema de eventos integrado, disponibiliza interfaceamento com alocadores de memória customizados. Apenas sistemas precisam utilizar herança.

- Anax[5] - Biblioteca ECS, escrita em C++11 e licença MIT. A biblioteca utiliza módulos da Boost, e visa sua utilização para desenvolvimento de jogos. Possui eventos para criação e remoção de entidades. Componentes e sistemas precisam usar herança.

- EntityX[6] - Biblioteca ECS, escrita em C++11 e licença MIT. Possui sistema de eventos integrado, uma classe para gerenciamento da aplicação (que facilita o desenvolvimento rápido), as entidade são representadas como índices, a biblioteca exige que os sistemas utilizem herança.

- Ginseng[7] - Biblioteca ECS, escrita em C++17 e licença MIT. Não exige herança, define entidades como índices. Permite remoção e inserção de componentes (e entidades) durante um *update* de sistema.

- Unity DOTS[8] - *Framework* integrado na Unity 3D, Escrito em C#. Contempla a primeira implementação de ECS oficial de uma grande *game-engine*. A arquitetura foi desenvolvida com suporte ao C# *Job System* para *multi-threading* e possui um compilador próprio (*Burst Compiler*) para otimizações de baixo nível.

- Entitas[9] - Biblioteca ECS, escrita em C# e licença MIT. Foi uma das primeiras bibliotecas ECS de C#, possui um módulo de suporte para Unity 3D (que surgiu antes da implementação oficial).

3 Arquitetura

A arquitetura desenvolvida se embasa nos dois conceitos descritos na Seção 2 (*Entity Component System* e *Data-Oriented Design*), com ênfase em performance e praticidade de utilização da API. Além disso, é tido como destaque a modularidade da arquitetura, em reflexo ao design ECS. A arquitetura vincula os sistemas da aplicação, definindo as bases atreladas à aplicação do usuário.

Uma das vantagens de se trabalhar com sistemas que atuam em componentes é desvincular as entidades da cena (e portanto suas representações de objetos do mundo) à forma como seus componentes são utilizados pelos sistemas. Isso possibilita que um sistema gráfico, por exemplo, não esteja relacionado com a lógica do jogo, evitando abstrações e facilitando a modelagem de problemas, uma vez que esse sistema não acessa diferentes classes de objetos, por exemplo: Guerreiro, Mago, Espada, Botão, etc.

As implementações possíveis de ECS são estritamente vinculadas a linguagem de programação utilizada para o desenvolvimento. Nesse contexto, optou-se pela linguagem C++11, por garantir mais controle sobre os dados, possuir boa compatibilidade e permitir a construção de *wrappers* para outras linguagens. A fim de facilitar o desenvolvimento e evitar erros por parte do programador (em especial nas implementações de sistemas), utilizou-se um recurso denominado *variadic template*, que possibilita a utilização de declaração de classes com um número variado de *templates*. O diagrama de classes deste projeto é representado na Figura 2.

3.1 Componentes

Os componentes são pequenas unidades de dados que definem propriedades específicas de uma representação de objeto. De forma análoga, pode-se associar um componente a um bloco essencial e comum na composição de um tipo de objeto. Por exemplo, no contexto de jogos, os mais diversos objetos de mundo tem posição em

[3] https://github.com/skypjack/entt
[4] https://github.com/redxdev/ECS
[5] https://github.com/miguelmartin75/anax
[6] https://github.com/alecthomas/entityx
[7] https://github.com/apples/ginseng
[8] https://unity.com/pt/dots
[9] https://github.com/sschmid/Entitas-CSharp

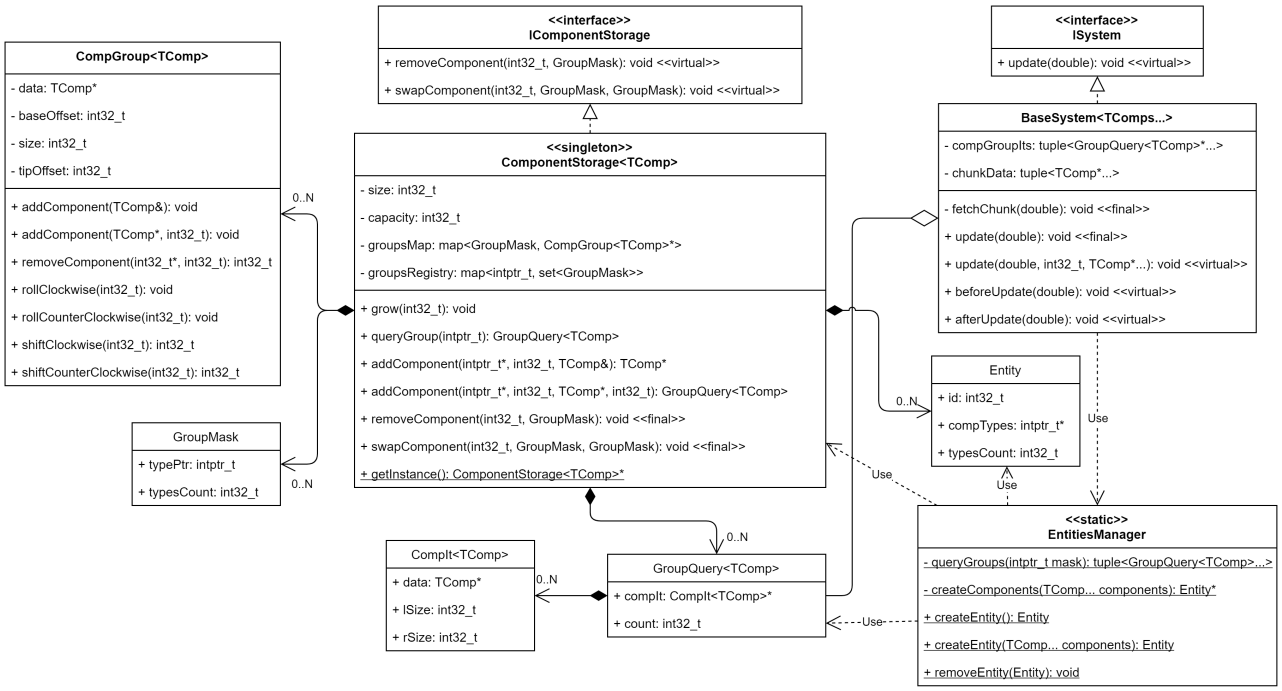

Figura 2: Diagrama de classes específico, Arquitetura ECS. Fonte: Autor.

cena e um nome de identificação, esses são dois tipos de componentes muito comuns e que, na orientação a objetos, costumam ser atributos de uma classe.

Na Ravine, qualquer estrutura do tipo *plain-old-data* (POD) pode ser utilizada como componente, ponteiros são possíveis, mas não devem ser gerenciados (alocados/desalocados) pelo componente. Apesar disso, na prática, é comum que os componentes possuam funções auxiliares, como sobrecarga de operadores para interação entre componentes.

Outro aspecto cujas implementações variam é a forma de armazenamento dos componentes, descrita em detalhes na Seção 4. Esse armazenamento se dá na classe **ComponentStorage**, que também gerencia o ciclo de vida dos mesmos. Internamente, o *storage* organiza os componentes em grupos, classe **ComponentsGroups**. Na iteração dos componentes a estrutura **GroupQuery** é utilizada temporariamente para acessar vários grupos e a classe **CompIt** ajuda a acessar os blocos de componentes dentro deles.

Além disso, a fim de possibilitar algumas iterações independentes do tipo de componente, foi criada uma interface **IComponentStorage**. Isso viabiliza, a partir do ponteiro do *storage*, remover um componente ou trocar a qual grupo ele pertence. Para identificação desses componentes, durante a chamada da função, é necessário o identificador (**int32_t**) da entidade e uma máscara (**GroupMask**) para achar o grupo no registro.

3.2 Entidades

Sendo uma representação de um objeto, uma entidade é o que relaciona um conjunto de componentes entre si. Apesar de cada tipo de componente identificar uma característica, são conjuntos de componentes que efetivamente representam um objeto do jogo. A classe **Entity** armazena o identificador da entidade e os tipos de componente que ela possui. É pelo meio delas que o usuário da arquitetura cria e destrói componentes.

A grande importância da representação de uma entidade é a ordenação dos componentes dentro dos *storages* de seus respectivos tipos. Nesta implementação, uma entidade é que define (pela posse de um conjunto de componentes) a qual grupo os componentes devem pertencer. Todas as entidades são gerenciadas pelo **EntitiesManager** que, por sua vez, garante a criação dessas por chamadas aos diversos *storages* de componente.

3.3 Sistemas

As partes atuantes em uma arquitetura ECS são os sistemas. Cada sistema faz uso de um ou mais tipos de componentes mas atua nesses conjuntos de forma isolada. Na prática, um sistema itera pelas listas dos componentes acessando vários tipos de componente a cada vez, computando as alterações, e modificando os componentes conforme o resultado.

As classes de sistemas constituem uma herança de **BaseSystem** com os *templates* de classes componentes desejados. A fim de poder listar e gerenciar todos os sistemas da mesma forma, foi criado mais um nível de indireção **ISystem**. De tal forma, **BaseSystem** apresenta a implementação final da função **update**, da interface **ISystem**, e gera uma função virtual de tipo semelhante passando um ponteiro para cada listas de componentes do tipo declarados pelo *template*.

Opcionalmente, foram integradas funções virtuais chamadas antes da execução do sistema. Além disso, alternativamente, é chamada uma outra função **update** que passa parâmetros úteis dos grupos, como o tamanho do bloco de componentes sendo processado e o *offset* do grupo, relativo ao início do *array* de dados do *storage*.

4 DESENVOLVIMENTO

Nessa seção são descritos, em mais detalhes, os aspectos técnicos de desenvolvimento que permitem garantir a eficiência da arquitetura. Na Seção 4.1 é abordada a problemática do armazenamento de componentes e a elaboração de uma solução altamente eficaz, bem como os prós e contras dessa implementação. Em seguida, na Seção 4.2, é descrita a metodologia escolhida para alta performance no acesso dos componentes e otimizado para o *layout* de memória definido na seção anterior. Na Seção 4.3 são apresentados mais detalhes sobre o tratamento de entidades no sistema indicado.

Por fim, a Seção 4.4, elabora melhor pontos onde foram aplicadas otimizações e onde são previstas melhorias.

4.1 Armazenamento de Componentes

Para o desenvolvimento do armazenamento de componentes, é necessário escolher uma maneira de identificar os tipos de componente em tempo de execução, descrita na Seção 4.1.1, bem como um *layout* de memória para o armazenamento, Seção 4.1.2. Em seguida, é descrita uma organização interna dos *arrays* de dados (Seção 4.1.3), vital no desempenho do sistema. Na Seção 4.1.4, são descritas as operações que atuam sobre esse armazenamento. Por fim, na Seção 4.1.5, são delineadas as formas de registro que facilitam e otimizam o acesso à diferentes tipos de componentes pelos sistemas.

4.1.1 Identificando Componentes

Em C#, a inferência de um tipo a partir de uma instância de objeto é um recurso nativo da linguagem (denominado Reflection). Infelizmente, em C++ esse recurso ainda não é um padrão da linguagem, e são diversas as soluções para esse problema. Bibliotecas como o EnTT costumam utilizar de identificadores gerados em tempo de compilação ou macros que auto-incrementam um índice armazenado na classe que precisa ser identificada. Outros métodos usam *variadic template expansion* para compilar uma *hash* a partir do nome do arquivo (que é injetado pelo compilador).

Para a prototipação dessa arquitetura, foi utilizada uma técnica onde a criação de um *storage* ocorre em uma variável estática por uma função com *template* do tipo de componente na sua assinatura. Isso gera um endereço único e imutável de memória para cada tipo de componente. Dessa forma, é possível identificar um componente com o ponteiro do *storage* ao qual ele pertence. De maneira inversa, tendo um identificador de um tipo de componente, basta interpretá-lo como um endereço de memória para acessar o seu *storage*.

4.1.2 *Layout* de Memória

Na implementação da Unity 3D[10], os componentes são organizados pelos chamados *archtypes*, ou arquétipos. Essencialmente, esses descrevem uma combinação única de tipos de componentes. Cada arquétipo representa uma *pool* com alocações de blocos de tamanho fixo. Essa organização é denominada de *Array of Structs (AoS)*.

Isso otimiza o *layout* de memória para sistemas que acessam vários tipos de componentes, mas faz com que um sistema mais simples tenha baixa utilização do cache, quando acessando entidades com muitos componentes. Além disso, o acesso à diferentes *pools* de componentes, que é muito comum em sistemas simples, significa um grande pulo no acesso da memória. Esse problema é atenuados pela utilização de múltiplas *threads* para o processamento desses sistemas (com o *Jobs System*)[14], onde diferentes *pools* são, normalmente, acessadas por diferentes núcleos físicos, por decorrência, utilizam-se memórias cache fisicamente distintas.

Outro motivo para essa organização é a redução do uso de *prefetch streams* do processador, uma vez que o acesso simultâneo de muitos endereços de memória pode ocasionar uma queda drástica na performance. Em alguns processadores Intel, o *prefetcher* pode reconhecer até oito *streams* concorrentes[9]. Já alguns processadores ARM, para dispositivos móveis, conseguem até doze *streams* concorrentes[3].

A biblioteca EnTT realiza alocações em *arrays* lineares de memória por tipo de componente e usa *look-up tables* para achar ou marcar posições vazias, os chamados *sparse sets*[10]. Esse padrão de organização é denominado *Structure of Arrays (SoA)*, onde componentes de uma mesma entidade estão em *arrays* separados da memória. Por conta das tabelas de mapeamento de posições essa biblioteca tem destaque pela sua alta eficácia na criação e remoção de componentes e entidades, mas faz com que o padrão de armazenamento e acesso não seja sempre linear. Por consequência, essa biblioteca pode ser altamente customizada, mas possui um *overhead* no acesso de componentes.

Na palestra da GDC sobre o desenvolvimento de Overwatch[7], foi observado que a maior parte dos sistemas não acessa tantos tipos de componentes. Ainda assim, quando isso acontece, o acesso se dá nos chamados *Singleton Componentes*, que são, efetivamente, componentes únicos e não um *array* de componentes, então costumam caber no cache. Dessa maneira, a quantidade de *prefetch streams* utilizadas concorrentemente em um jogo não ultrapassa os limites de processadores modernos. Por isso, para o desenvolvimento da Ravine ECS, optou-se pela organização dos dados no *layout* SoA, similar à biblioteca EnTT, evitando a criação de *pools* de alocação que podem estar muito distantes na memória RAM e, por consequência, reduzindo a chance de *cache-misses*.

4.1.3 Grupos de Componentes

Como descrito na seção anterior, o armazenamento dos componentes ocorre em *arrays* lineares por tipo de componente. Contudo, diferente de outras implementações, os componentes alocados nesse *array* são separados em regiões representativas de seus arquétipos (grupos coloridos, na Figura 3). Possibilitando o acesso indexado à componentes que representam qualquer arquétipo em diferentes *storages*.

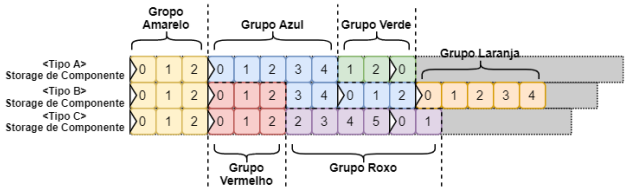

Figura 3: Grupos de Componentes. Fonte: Autor.

No decorrer do desenvolvimento desse trabalho, a biblioteca EnTT implementou uma nova maneira de organizar os componentes, denominada de *Component Groups*. A implementação descrita no blog do criador da biblioteca[11] é muito semelhante a aqui representada, mas se limita a um caso específico onde há o acesso de somente dois tipos de componentes (um arquétipo específico). A organização proposta na Ravine ECS estende essa concepção para qualquer combinação simultânea de tipos de componentes. Contudo, a publicação do EnTT definiu a nomenclatura para um conjunto de componentes que representam algum arquétipo, a qual foi seguida nesse trabalho.

A Figura 3 é uma representação de três *storages* de componentes com seus respectivos grupos. Cada bloco significa um componente e, para um grupo nos diferentes *arrays*, o acesso de um mesmo índice (descrito nos blocos) é relativo aos componentes pertencentes à mesma entidade. De forma análoga, uma entidade tem seus componentes identificados por um índice dentro de um grupo de componentes. A fim de evitar a realocação de todo o *array* quando um novo componente é adicionado, cada grupo é organizado de maneira cíclica. O início de cada grupo é indicado pelo triângulo branco, chamado de ponta (*tip*) (Figura 3). Na prática, essa posição é marcada por um deslocamento com relação à posição inicial do grupo no *storage*.

Essa organização mantém os componentes em um padrão de acesso linear, mas pode possuir eventuais quebras entre componentes, como exemplificado no pulo de acesso para as entidades 2 e 3 no grupo azul. Isso exige uma correção no índice antes do acesso ao

[10] https://bit.ly/3dStqdj

[11] https://bit.ly/3gpFzIC

componente e, em troca de maior flexibilidade na adição e remoção de componentes, pode provocar um eventual *cache-miss* (quando os grupos são muito grandes). Além disso, destaca-se a escolha de posicionar grupos de componentes de arquétipos mais complexos (com mais tipos de componentes) à esquerda. É importante ressaltar que a inserção e remoção de entidades mais simples se torna mais otimizada, por estarem localizadas no final do *array*.

4.1.4 Operações de Armazenamento

Para que a representação em grupos de componentes funcione, é necessário que as operações de remoção e inserção de componentes mantenham coerência interna dos grupos. Além disso, a fim de otimizar essas operações, foi seguido um *design* orientado à dados. Sendo assim, tanto a inserção quanto a remoção de componentes são compostas por outras pequenas operações realizadas repetidamente para cada grupo. Ainda assim, a inserção e remoção de componentes desloca, eventualmente, um número considerável de componentes, sendo esse um ponto fraco desta implementação.

Em termos de organização interna de um grupo, foram criadas quatro operações básicas possíveis.

- Mover: Move um conjunto de elementos na direção desejada; mantém o deslocamento relativo da ponta; mantém o tamanho do grupo.
- Rolar: Move os componentes de maneira circular (sentido horário ou anti-horário); incrementa/decrementa a posição do início do *array*; muda o deslocamento relativo da ponta; mantém o tamanho do grupo;
- Deslizar: Move (para esquerda/direita) os componentes antes/depois da ponta; rola aqueles que passarem do início/fim do grupo; mantém o deslocamento relativo da ponta; aumenta o tamanho do grupo.
- Comprimir: Move componentes na direção relativa à ponta; rola componentes no sentido anti-horário pra preencher espaços vazios; muda o deslocamento relativo da ponta; diminui o tamanho do grupo.

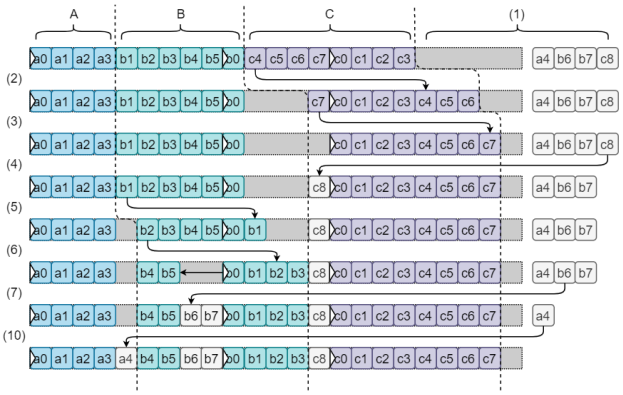

Figura 4: Inserção/Criação de Componentes. Fonte: Autor.

A operação de inserção (ou criação) de componentes em um *storage* com os grupos A, B e C está representada na Figura 4. A lista de componentes inseridos (a4, b6, b7, c8) é ordenada com a mesma lógica dos grupos. Por definição, a fim de não influenciar a ordem dos componentes já existentes no grupo, a inserção deve ocorrer no final do grupo (antes da ponta), de maneira cíclica. As operações ocorrem da direita para a esquerda. As etapas para a inserção, conforme enumeradas na figura, são:

1. Garantir que o *storage* seja grande o suficiente (senão realocar o *array* inteiro).
2. Rolar o grupo C no sentido horário pela quantidade de componentes restantes nos outros grupos (A e B): 3.
3. Deslizar, para a esquerda, os elementos à esquerda da ponta de C para quantos elementos precisam ser inseridos: Deslizar 1 componente, 1 posição.
4. Copiar todos os elementos de C para a ponta: ”c8”.
5. Rolar o grupo B no sentido horário pela quantidade de componentes restantes nos outros grupos (A): 1.
6. Deslizar, para a esquerda, os elementos à esquerda da ponta de B para quantos elementos precisam ser inseridos: Deslizar 4 componentes, 2 posições.
7. Copiar todos os elementos de B para a ponta: ”b6, b7”.
8. Rolar o grupo A no sentido horário pela quantidade de componentes restantes nos outros grupos: 0.
9. Deslizar, para a esquerda, os elementos à esquerda da ponta de A para quantos elementos precisam ser inseridos: Deslizar 0 componentes, 1 posição.
10. Copiar todos os elementos de A para a ponta: ”a4”.

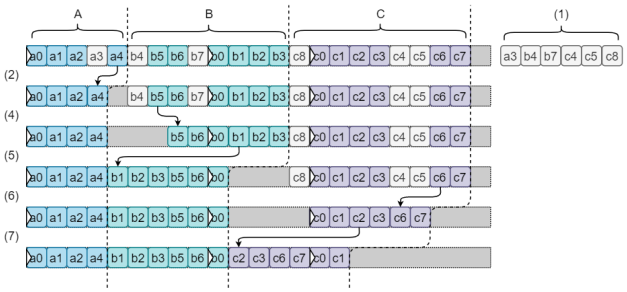

Figura 5: Remoção/Deleção de Componentes. Fonte: Autor.

A operação de remoção (ou deleção) de componentes em um *storage* com os grupos A, B e C, está representada na Figura 5. A lista de componentes removidos (a3, b4, b7, c4, c5, c6) é ordenada com a mesma lógica dos grupos. Diferente da inserção, a remoção pode ser efetuada em um componente em qualquer posição, afetando os índices dos outros componentes. Na prática, isso não causa incoerências no acesso dos componentes, visto que a remoção de uma entidade provoca a remoção dos componentes em todos os *storages*. Diferente da inserção, a remoção ocorre da esquerda para a direita. As etapas para remoção, conforme enumeradas na figura, são:

1. Calcular as operações de compressão (posição, quantidade e direção) usando a lista de deleção.
2. Comprimir todos os elementos de A para a ponta: Mover 1 componente para esquerda por 1 posição.
3. Rolar A no sentido anti-horário para preencher as posições vazias na ponta: 0 posições.
4. Comprimir todos os elementos de B para a ponta: Mover 2 componentes para a direita por 1 posição.
5. Rolar B no sentido anti-horário para preencher as posições vazias na ponta: 3 posições.
6. Comprimir todos os elementos de C para a ponta: Mover 0 componentes para a direita por 1 posição; Mover 2 componentes para a esquerda por 2 posições.
7. Rolar C no sentido anti-horário para preencher as posições vazias na ponta: 4 posições.

Na prática, foi implementada uma estrutura que guarda as informações de um grupo e possui as implementações de cada uma dessas operações. Além disso, as operações de remoção e inserção devem ser executadas antes ou depois de um sistema acessar os componentes. Cada operação foi testada e está disponível no *branch* **component_group_done** do repositório do GitHub[12].

4.1.5 Registro de Grupos

A fim de ordenar e organizar os grupos dentro dos *storages* de componentes foi implementado um sistema de registro de grupos. Na prática, como um grupo é uma representação de um arquétipo, ou seja, um conjunto de tipos de componentes, relembrando que um tipo de componente é identificado pelo ponteiro do *storage* daquele componente, um grupo seria uma combinação de ponteiros. Além disso, um arquétipo mais complexo significa uma combinação de mais ponteiros. Então a operação de ordenação leva em consideração primeiro a quantidade de ponteiros e depois o valor do identificador.

Dessa forma, os identificadores dos grupos são calculados por uma *hash* dos ponteiros de cada tipo de componente do arquétipo em conjunto com a quantidade de ponteiros. Como o arquétipo é o mesmo para um conjunto de tipos de componentes, indiferente da ordem, a operação de *hash* escolhida não pode depender da ordem. Além disso, como os valores dos ponteiros costumam ser altos, ela deve lidar bem com *overflow* da representação do identificador. Na Ravine ECS, essa operação foi simplesmente a soma dos valores dos ponteiros, armazenados na estrutura **intptr** do C++. O mapeamento de todos os grupos de um *storage* é realizado por meio de uma estrutura *set*, ordenado, restringindo a existência de só um grupo por arquétipo.

Por definição um arquétipo complexo representa, nas possíveis combinações de seus tipos de componentes, arquétipos mais simples. Por exemplo, o arquétipo {A,B} representa dois "sub-arquétipos": {A} e {B}. Isso se torna relevante, pois sistemas que atuam em arquétipos simples devem atuar, também, nos grupos de arquétipos mais complexos que os representam. Para evitar uma busca pelos grupos que contém um "sub-arquétipo"específico, essa informação é pré-computada com o cálculo de todas as possíveis combinações de arquétipos quando um novo grupo é criado. Então essa informação é registrada em uma estrutura de *hash-map*.

Essa estrutura confere um acesso de tempo constante à uma lista de todos os grupos que representam, no mínimo, o arquétipo requisitado pela operação. Internamente, cada chave é mapeada para um *set* ordenado pelos identificadores dos grupos. Cada *storage* possui o registro relativo aos outros tipos de componentes. O *storage* de tipo A, com base no exemplo anterior, tem registros de grupos com identificador B (representando o grupo do arquétipo {A,B}) e com identificador zerado (representam o grupo do arquétipo puro {A}).

4.2 Iteração de Componentes

Por definição, muitas arquiteturas ECS abordam a ideia de que os sistemas iteram sobre entidades que possuem arquétipos específicos. Na prática, as implementações possibilitam que sistemas possam filtrar por arquétipos específicos de componentes com regras de exclusão e inclusão de tipos de componentes. Como descrito na seção anterior, para o protótipo da Ravine, cada grupo é registrado para cada um dos "sub-arquétipos"que representa.

Inicialmente, o acesso aos componentes foi implementado com um iterador para cada tipo de componente e os sistemas recebiam uma chamada de **update** por *tick* de simulação, que passava esses iteradores. Essa estrutura calculava, com um único índice, a localização do grupo que representava essa entidade, bem como a posição ajustada dentro do grupo (em função do *array* cíclico). Na prática o cálculo mostrou-se, na maior parte das vezes, mais complexo do que a própria operação efetuada pelos sistemas, causando baixíssimo desempenho.

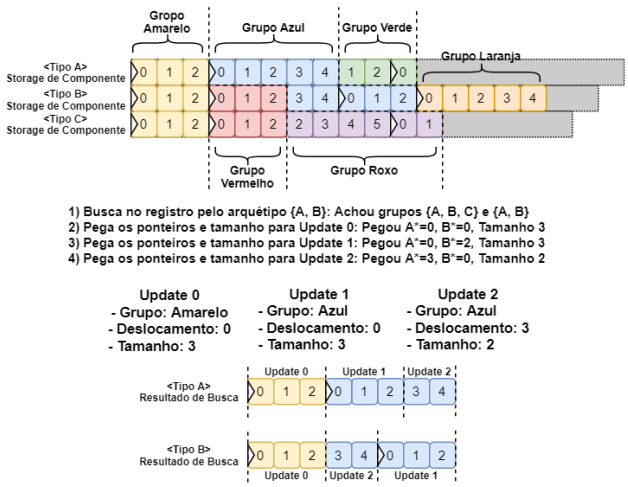

Figura 6: Iteração/Update de Componentes. Fonte: Autor.

Foi testada então uma abordagem onde os iteradores só calculavam a posição ajustada dentro do grupo. Os sistemas recebiam várias chamadas de **update** por *tick* de simulação, uma para cada grupo. Nesse contexto, os índices utilizados pelo usuário são só significativos dentro de cada grupo. Ainda assim, a soma e resto de divisão utilizados para o ajuste do índice num determinado grupo mostraram-se *overhead* demais em comparação com sistemas simples, onde o acesso das tabelas de indireção do EnTT conseguiam melhor performance.

Por fim, ao invés da utilização de iteradores, optou-se por aumentar ainda mais os números de chamadas de **update** por *tick* de simulação, ajustando os ponteiros das chamadas para alinhar com o início de cada grupo (Figura 6). Dessa forma, não há a necessidade de uma conversão do índice utilizado dentro da chamada **update** que o usuário sobrescreve, o acesso ao componente dentro do *array* dá-se pelo operador padrão e um mesmo índice é mapeado para os componentes representativos de uma mesma entidade.

4.3 Entidades como Componentes

Como descrito na Seção 4.1.3, as entidades da Ravine, são definidas por um grupo, representativo do arquétipo de seus componentes, e um índice dentro desse grupo. Na prática, para adicionar e remover componentes (ou as próprias entidades) é necessário acessar os *storages* de cada componente que ela representa. Por isso, é necessário armazenar uma lista de cada tipo de componente, ao invés do identificador calculado (que não pode ser revertido).

Como as entidades precisam estar armazenadas em algum lugar, faz sentido representar uma entidade como um tipo de componente. Dessa forma, cada entidade criada implica na alocação de um componente do tipo **Entity** juntamente com os componentes que ela possui. Por consequência, o componente da entidade pertence a um grupo que representa o mesmo arquétipo que seus componentes e é alocado na mesma posição. Para manter os dados de entidades atualizados, foi implementada uma especialização de *template* para a operação de remoção dos grupos de entidades que, além de realizar as operações citadas, atualiza o índice das entidades.

4.4 Otimizações

No contexto de otimizações, a utilização de *variadic template* juntamente com declarações de funções como *inline* ou *constexpr* faz com que o compilador consiga reduzir a quantidade de chamadas

[12] https://bit.ly/2Zv2riZ

de funções no código, trocando tempo de compilação por performance. Além disso, a fim de evitar problemas com predição de condicionais nos código, as operações de grupos (Seção 4.1.4) foram todas implementadas sem o uso de *if/else*. Isso é possível com a utilização de *bitmasks*, e usando o bit de sinal da representação numérica **int32_t**. As operações são implementadas com laços de repetição e chamada **memmove** e **memcpy**, esses dão suporte para argumentos de tamanho zero, não executando operações. Isso possibilita que o compilador consiga otimizar ainda mais o código.

Além das otimizações já implementadas, são identificadas diversas outras otimizações possíveis. Dentre essas, destaca-se a oportunidade de processamento concorrente de dados. Para isso, são discutidos alguns pontos-chave, considerando a organização da memória e concorrência de dados, bem como a organização dos fluxos da aplicação. É necessário compreender as dependências de acesso aos componentes por diferentes sistemas, de forma a não acessar os mesmos componentes simultaneamente a partir de sistemas diferentes e/ou com ordem equivocada entre sistemas. A Unity, em sua arquitetura DOTS, aborda os sistemas de mesma maneira, e apresenta a criação de barreiras de acesso para forçar pontos de sincronia[2]. A seguir são indicadas algumas ideias de otimizações com paralelismo.

4.4.1 Sistemas Paralelos

É possível tanto realizar a execução de um sistema totalmente em outra *thread* como também gerar blocos de trabalho (*jobs*), cada qual processando parte do *array*. Esses são despachados para um conjunto de *threads*, a fim de aproveitar melhor os recursos de *hardware* dos diferentes núcleos de processamento.

4.4.2 Ordenamento Paralelo

De maneira semelhante aos acessos paralelos por sistemas, uma vez que as dependências dos acessos sejam estabelecidas, é possível utilizar uma *thread* dedicada para acessar componentes não utilizados no momento e reordenar o *array* circular de cada grupo, para que todos fiquem alinhados linearmente. Uma vez alinhados, o acesso desses componentes ocasiona a menor quantidade possível de *cache-misses*.

5 Validação

Nessa seção são descritos tanto os testes de performance, direcionados para a organização de componentes, quanto o desenvolvimento de duas aplicações em Open Graphics Library (OpenGL), uma delas seguindo o design orientado a objetos (OOD) e outra seguindo o design orientado a dados (DOD), como estudo de caso da Ravine.

5.1 Testes de Performance

Com o intuito de averiguar as vantagens dos conceitos empreendidos no desenvolvimento da arquitetura, em especial no acesso dos componentes, foram elaborados testes com vários padrões de acesso de sistemas e um número variado de entidades, simulando diferentes cenários de utilização[13]. Foram desenvolvidos os seguintes cenários de teste:

- Um sistema acessando um arquétipo puro {A}, atualizando somente A.
- Um sistema acessando dois arquétipos puros {A} e {B}, atualizando A, depois B.
- Um sistema acessando um arquétipo duplo {A,B}, atualizando A e B.
- Um sistema acessando um arquétipo triplo {A,B,C}, atualizando A e B e C.
- Dois sistema acessando dois arquétipos duplos {A,B} e {B,C}, atualizando A e B, depois B e C.

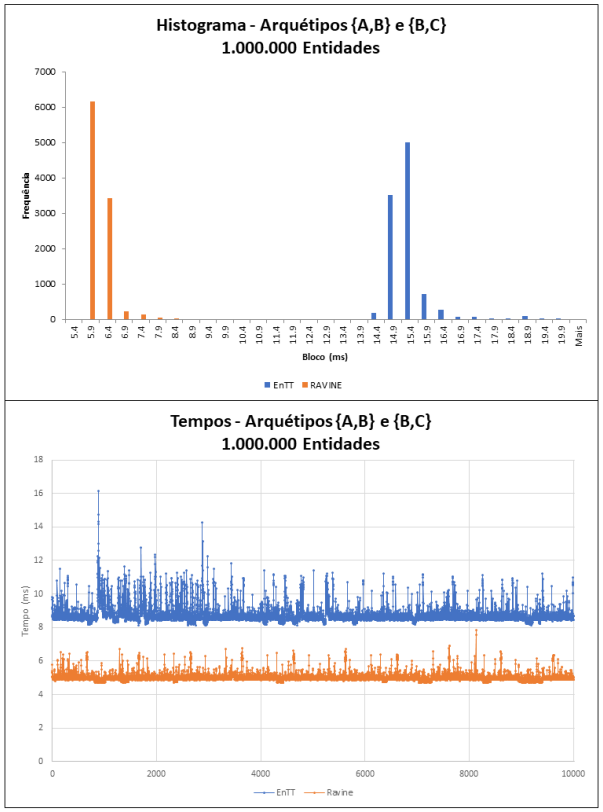

Figura 7: Histograma e Tempos 1 milhão de entidades, dois pares de arquétipos ({A,B} e {B,C}), com dois sistemas. Fonte: Autor.

A fim de otimizar o tempo de desenvolvimento, foi consultado um *benchmark*[14] comparando vários sistemas, muitos dos quais são citados nos trabalho relacionados. Com base neste *benchmark* foi escolhido o sistema que apresentou os melhores resultados, o EnTT, como base de comparação.

Foi definida uma classe de abstração para as etapas de: configuração, execução e limpeza dos dados. Cada teste segue esse fluxo de chamadas a fim de garantir que não haja interferência entre as execuções. Além disso, a etapa de execução roda 10.000 vezes (*ticks*), cada qual com o tempo cronometrado. Isto é feito para diferentes números de entidades (1.000, 10.000, 100.000 e 1.000.000). Cada sistema aplica uma operação simples de somar o tempo decorrido entre o *frame* anterior e o atual em cada componente.

O computador utilizado possui um processador Intel i7-7700HQ (6Mb de L3 cache, 1.5Mb de L2 cache e 256Kb de L1 cache) com 32GB de RAM (2400MHz, CL17). Os testes foram realizados com apenas uma *thread* de aplicação. Os tempos foram medidos com o cronômetro de alta precisão da biblioteca padrão do C++, **chrono**, e gravados em formato CSV. Os dados foram analisados com a ferramenta Microsoft Excel, de onde foram gerados gráficos com os tempos e histogramas de frequência (Figura 7).

5.2 Estudo de Caso

Para contemplar um estudo de caso similar à um jogo, foram desenvolvidas duas aplicações gráficas, com a Open Graphics Library

[13] Disponível em `https://bit.ly/38pmEe2`

[14] `https://github.com/abeimler/ecs_benchmark`

(OpenGL)[15]. Uma delas segue os conceitos do design orientada a dados (DOD), com a utilização da arquitetura ECS da Ravine. Em contra-partida, a outra segue a concepção do problema a partir do design orientada a objetos (OOD). O objetivo é, além de exemplificar o uso da biblioteca, também comparar os dois princípios de *design* supracitados, em termos de processo de desenvolvimento, complexidade de código e facilidade de otimização.

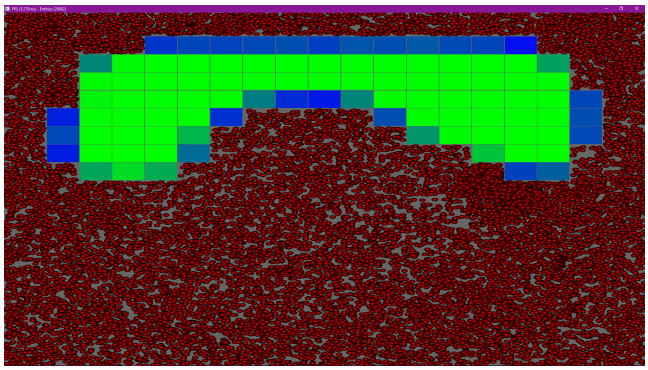

Figura 8: Aplicação para estudo de caso (DOD). Fonte: Autor.

A temática da aplicação se baseou no jogo Breakout de Atari[16], onde o objetivo é destruir todos os tijolos na fase, rebatendo uma ou mais bolas em uma barra controlada pelo usuário. Contudo, diferente da proposta original do jogo, essa aplicação é uma demonstração para um número muito grande de entidades (no caso as bolas). Sendo assim, a proposta é lançar um grande número delas, onde os blocos só quebram depois de atingidos diversas vezes (cuja vida é representada pelas cores verde, quando cheia, e azul, quando acabando). Além disso, não existe uma barra e as bolas nunca saem fora da tela (não são destruídas). A Figura 8 mostra a aplicação implementada com a Ravine ECS.

As interações possíveis são: criar novas bolas na direção do mouse na tela com o botão esquerdo do mouse; criar um campo gravitacional que atrai as bolas próximas com o botão direito do mouse. Cada iteração foi desenvolvida de maneira diferente em cada uma das aplicações. Essas atuações são realizadas pelos sistemas, na aplicação da arquitetura ECS, e pela própria classe **balls**, na aplicação desenvolvida com a programação orientada a objetos.

Vale ressaltar que a implementação de colisão possui complexidade *Big-O* de O(n*m), para **n** como o número de bolas e **m** o número de tijolos; cada tijolo é com todas as bolinhas, sendo essa a parte menos performática nas aplicações. A operação de atração dos objetos é implementada com complexidade O(n), para **n** como o número de bolas; a posição de cada bola é comparada com a posição do mouse.

6 Análise de Resultados

Nessa seção são apresentados os resultados provenientes dos testes de performance e do processo de desenvolvimento do estudo de caso, descritos na seção anterior. Também são elaboradas as interpretações associando às possíveis causas dos resultados.

6.1 Resultados dos Testes

A fim de compreender os dados, foi realizada uma análise estatística da frequência de ocorrência dos tempos por meio de histogramas. Os demais gráficos, assim como as tabelas das análises, podem ser consultados nos anexos[17] do trabalho. Com isso, percebeu-se que a distribuição dos valores não segue uma curva normal, mas sim uma distribuição de Poisson.

De forma a simplificar a análise dos dados, aproximou-se a curva à uma normal pela eliminação de picos, fazendo-se então uma análise com média e desvio padrão, gerando uma tabela que representa propriamente os resultados (Figura 9). Optou-se pela seleção de 95% dos valores mais representativos para o cálculo da média e desvio padrão, eliminando alguns picos, considerados *outliers*. Contudo, foi mantida a soma dos 10.000 valores, para comparação do processo de testes como um todo. Por exemplo, no teste de 1.000.000 de entidades com dois pares de arquétipos ({A,B} e {B,C}) os tempos totais do EnTT e da Ravine foram, respectivamente, 150606.98ms e 59221.74ms.

Nesse teste, foram registrados tempos médios até 2.5x menores que a base de comparação. Além disso, os histogramas indicaram maior estabilidade no acesso dos componentes, com menor frequência de tempos maiores. Isso também fica visível no desvio padrão amostral, em especial com mais entidades. É importante destacar que os tempos aumentam com o número de entidades, para ambas as bibliotecas.

No teste com apenas um arquétipo puro {A}, a biblioteca EnTT teve melhor desempenho pela utilização dos ponteiros internos. Na prática, isso só é possível em casos bem específicos, onde a iteração ocorre por componentes não utilizados por entidades, o que pode causar comportamento indefinido. Por exemplo, esse dado não pode ser utilizado para desenhar objetos em tela, pois objetos inexistentes seriam representados.

Os testes estão organizados por ordem de complexidade crescente, denotado pelo maior acesso concorrente de arquétipos. O ganho de performance da biblioteca Ravine ECS em relação a do EnTT se torna maior à medida que os sistemas aumentam em complexidade. Atribui-se isso à linearidade e simplicidade de iteração pelos *storages* de componentes na Ravine, em oposição ao acesso pela tabela de pesquisa no EnTT e a limitação de dois tipos de componentes para a organização em grupos.

Tem-se como interpretação que a biblioteca EnTT possui um sistema de armazenamento e iteração de componentes que causa mais incoerência no acesso de memória. Com isso, o *prefetcher* de *hardware* não consegue deixar o dado futuro no cache, ocasionando em *cache-misses*. Como a aplicação precisa esperar a latência de acesso para ter os dados, é comum que o sistema operacional jogue o processo para o fim da fila de escalonamento, o que explicaria os picos de tempo durante algumas execuções.

Outros pontos onde a implementação desse artigo se destaca é na serialização desses dados. Uma vez que a biblioteca EnTT precisa da tabela de pesquisa (*look-up table*) é necessário serializar essa informação junto com o *array* de dados. Em contra-partida, na Ravine, seria só necessário serializar os dados dos grupos de componente, cujo tamanho é consideravelmente menor, uma vez que cada grupo representa um conjunto de componentes de um arquétipo.

6.2 Resultados do Estudo de Caso

Durante o desenvolvimento das duas aplicações do estudo de caso, os primeiros benefícios percebidos foram na possibilidade de otimização. O fato das posições estarem em blocos lineares de dados possibilitou a utilização da estrutura *Shader Storage Buffer Object* (**SSBO**) para armazenar as posições na memória da GPU. Isso se torna útil com a chamada **glDrawArraysInstanced** que desenha múltiplos objetos iguais (instências, no caso do *mesh* da bola) e passa o índice da instância para o *vertex shader*.

Na prática, esse índice é utilizado no acesso do *buffer* de posições, mas, para um jogo complexo, esse *array* poderia conter os estados do *frame* atual para cada objeto. Como representado na Figura 8, são processadas e desenhados 20.002 entidades (bolas) interagindo com 95 tijolos no tempo de 5,27ms (189 FPS). Enquanto na aplicação com design orientado a objetos cerca de 3000 bolas são

[15]Disponível em `https://bit.ly/2YTY1CW`
[16]`https://bit.ly/2D57nn8`
[17]Anexos disponíveis em `https://bit.ly/3gkfiv5`

		Arquétipo {A}		Arquétipos {A} e {B}		Arquétipo {A,B}		Arquétipo {A,B,C}		Arquétipos {A,B} e {B,C}	
		EnTT	Ravine	EnTT	Ravine	EnTT	Ravine	EnTT	Ravine	EnTT	Ravine
1.000 Entidades	Média	0.0012	0.0018	0.0045	0.0037	0.0069	0.0042	0.0079	0.0068	0.0138	0.0081
	Desvio Pad A	0.0001	0.0001	0.0001	0.0001	0.0002	0.0001	0.0003	0.0003	0.0004	0.0003
	Soma (ms)	13.27	19.69	48.03	39.17	72.75	45.41	84.11	72.85	147.22	86.31
10.000 Entidades	Média	0.0116	0.0121	0.0433	0.0246	0.0716	0.0294	0.0819	0.0494	0.1436	0.0620
	Desvio Pad A	0.0004	0.0004	0.0024	0.0006	0.0073	0.0028	0.0092	0.0044	0.0158	0.0088
	Soma (ms)	123.06	125.87	447.30	256.02	741.16	307.28	843.24	515.49	1462.70	639.59
100.000 Entidades	Média	0.1175	0.1191	0.4488	0.2419	0.7268	0.2919	0.8251	0.4824	1.4625	0.5853
	Desvio Pad A	0.0120	0.0121	0.0285	0.0181	0.0386	0.0225	0.0355	0.0293	0.0604	0.0335
	Soma (ms)	1207.31	1215.24	4552.01	2479.36	7331.84	2963.86	8332.33	4886.95	14768.34	5951.57
1.000.000 Entidades	Média	1.2772	1.3195	5.3104	2.6025	7.4870	2.9428	8.7002	4.9982	14.9732	5.8788
	Desvio Pad A	0.0573	0.0839	0.1661	0.1413	0.1947	0.1313	0.2918	0.1317	0.3561	0.1304
	Soma (ms)	12908.77	13383.59	53473.86	26146.69	75375.42	29521.55	87808.73	50345.03	150606.98	59221.74

Figura 9: Resultados dos testes de performance (em milissegundos). Fonte: Autor.

processadas e desenhadas em tempos maiores que 6ms (166 FPS). Essa drástica diferença é atribuída ao tempo de CPU utilizado para tanto atualizar os dados de cada objeto e testar as colisões, quanto para a realização de muitas chamadas para a OpenGL.

Outro aspecto destacado é que quatro dos seis sistemas desenvolvidos são independentes da biblioteca gráfica escolhida. Diferente da implementação com os objetos, onde é necessário armazenar os índices dos *buffers* para que cada objeto consiga realizar a operação de desenho. É fato que implementações mais elaboradas conseguem abstrair a API gráfica, contudo, o desenvolvimento ressaltou a facilidade de criar códigos pouco eficientes seguindo a orientação a objetos.

Na tentativa de seguir os princípios S.O.L.I.D[13], foi despendido muito tempo na organização de uma arquitetura de aplicação, agregando complexidade de entendimento de código. Essa construção de um complexo modelo mental dos problemas agrega esforço do desenvolvedor na compreensão dos códigos e restringe a flexibilidade de expansão de novas funcionalidades. Por exemplo, a elaboração de uma nova ação do jogador exige, não raramente, a criação de uma interface, sua herança pela classe responsável (que há de ser definida, visto que a hierarquia pode ficar bem grande) e respectiva implementação apropriada.

Em termos de depuração de código, as interfaces ocasionam deferimento de responsabilidades. Isso significa que um erro pode propagar de uma classe de objeto para uma classe de gerenciamento que atua com inúmeras instâncias de tipos diferentes, por consequência com diferentes implementações plausíveis de erro. O código não fica explícito como nos sistemas da arquitetura ECS.

7 Considerações Finais e Trabalhos Futuros

Durante a realização desse trabalho, estudou-se um novo princípio de design de aplicações (*Data-Oriented Design*) e uma nova arquitetura de software (*Entity Component System*). Partindo desses dois temas, realizou-se uma pesquisa aprofundada sobre diferentes implementações existentes e seus pontos positivos e negativos. Com base no conteúdo compreendido, foi possível avançar o Estado da Arte com um inovador esquema de armazenamento de componentes. Pelo meio da implementação da arquitetura, foi possível comprovar a performance do sistema com testes que simulam diferentes cenários de uma aplicação.

Uma arquitetura ECS exige, diversas ferramentas para flexibilizar sua adequação aos mais diversos casos de uso. É interessante ressaltar que a implementação aqui descrita é inicial e não busca representar um produto final, sendo assim, carece de recursos como, por exemplo, um sistema de eventos. Dentre os trabalhos relacionados, é destacada a biblioteca EnTT, cujo criador, Michele Caini, foi contatado pelo autor, de onde surgiu uma possibilidade de integração do esquema de armazenamento aqui descrito com uma arquitetura cujo uso já é consolidada no mercado.

Além das otimizações citadas, um grande desafio em termos de trabalhos futuros é a reestruturação da forma de armazenamento. É de interesse que as operações de inserção e remoção de componentes sejam compostas apenas pela troca de poucos componentes, em oposição à movimentação de blocos de componentes. Outro ponto pouco elaborado na implementação atual é na representação dos tipos de componentes. Esse aspecto é essencial na concepção de uma arquitetura real, sendo um ponto chave na criação de cenas e para salvar e carregar o estado da aplicação.

Por fim, muitos dos recursos e das ferramentas destacados no desenvolvimento do jogo Overwatch[7] abrangem trabalhos futuros que, certamente, consolidarão a arquitetura para que fique pronta para a produção de aplicações em nível profissional. A integração nativa com diversas bibliotecas gráficas, além da inclusão de uma interface gráfica de usuário são objetivos almejados pela Ravine como um *framework*, em oposição ao escopo desse trabalho, que objetiva a Ravine ECS como um módulo em forma de biblioteca.

Referências

[1] M. Acton. Data-Oriented Design and C++. `www.youtube.com/watch?v=rX0ItVEVjHc`, CPPCon 2014. Acessado 02/07/2020.

[2] M. Acton. A Data Oriented Approach to Using Component Systems. `https://www.youtube.com/watch?v=p65Yt20pw0g`, GDC 2018. Acessado 02/07/2020.

[3] ARM. *ARM® Cortex®-A75 Core Technical Reference Manual*, chapter A6.5. ARM Limited, 2017. Issue 0200-00.

[4] E. Baumel. Understanding data-oriented design for entity component systems. `www.youtube.com/watch?v=0_Byw9UMn9g`, GDC 2019. Acessado 02/07/2020.

[5] S. Bilas. A Data-Driven Game Object System. `www.gamedevs.org/uploads/data-driven-game-object-system.pdf`, GDC 2002. Acessado 02/07/2020.

[6] R. Fabian. *Data-oriented design: software engineering for limited resources and short schedules*. Richard Fabian, self-published, 2018.

[7] T. Ford. Overwatch Gameplay Architecture and Netcode. `www.youtube.com/watch?v=W3aieHjyNvw`, GDC 2017. Acessado 02/07/2020.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, Massachusetts, USA, 1994.

[9] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, chapter 3.6.6. Intel Corporation, 2020. Número de Ordem 248966-043.

[10] T. Johansson. Job System & Entity Component System. www.youtube.com/watch?v=kwnb9Clh2Is, GDC 2018. Acessado 02/07/2020.

[11] P. Lange, R. Weller, and G. Zachmann. Wait-free hash maps in the entity-component-system pattern for realtime interactive systems. In *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 1–8, 2016.

[12] T. Leonard. Postmortem: Thief - The Dark Project. https://bit.ly/3gpE3G6, 1999. Acessado 02/07/2020.

[13] R. C. Martin. *Agile Software Development - Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, Nova Jersey, USA, 2002.

[14] S. McMillan. Designing for Efficient Cache Usage. www.youtube.com/watch?v=3-ityWN-FdE, Pacific++ 2018. Acessado 02/07/2020.

[15] T. Raffaillac and S. Huot. Polyphony: Programming interfaces and interactions with the entity-component-system model. *Proc. ACM Hum.-Comput. Interact.*, 3(EICS), June 2019.

[16] T. Rakic and S. Stetic-Kozic. Massive Battle in the Spellsouls Universe. https://www.youtube.com/watch?v=GEuT5-oCu_I, Unite Austin 2017. Acessado 02/07/2020.

[17] D. Wiebusch and M. E. Latoschik. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. In *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 25–32, 2015.