Projeto Final
BitEngine

2020 / 2021

19639 Vadims Zinatulins

Licenciatura em Engenharia Informática

BitEngine

19639 Vadims Zinatulins

Julho de 2021

Orientadores:

António Manso

«Dedicatória»

Agradecimentos

Pretendo agradecer aos professores António Manso por ter acompanhado o projeto ao longo do desenvolvimento do projeto.

.

Resumo

A indústria de vídeo jogos tem vindo a crescer bastante. Existem empresas novas todos os anos a entrar na área. Para além disso existe tecnologia suficiente para que pequenas pessoas individuais, ou pequenos grupos de pessoas, também lancem jogo para o mercado. Essas tecnologias são os motores de jogos que permitem rapidamente desenvolver um jogo sem grandes conhecimentos de programação, arte, animação, música, física, etc. podendo assim focarem-se apenas nas leis do jogo. Isto significa que há uma grande responsabilidade do lado dos motores para garantir a facilidade de desenvolvimento e uma boa performance. Uma das tecnologias mais recentes que permite isto é uma arquitetura chamada Entity Component System que visa proporcionar facilidade de desenvolvimento e manutenção bem como obter um desempenho elevado.

Neste projeto implementou-se um motor de jogo, com a arquitetura Entity Compont System (ECS) e de seguida fez-se medições de performance e uma comparação com a implementação convencional com objetos.

O motor de jogo (BitEngine) é responsável por gerir a janela de jogo, o estado dos inputs, gestão de cenários, gestão de entidades, gestão de sistemas e gestão de processamento paralelo.

Palavras-Chave: Game Engine

Entity Component System

Data Oriented Design

Object Oriented Programming

Acrónimos

Sigla	Descrição
CAS	Column Address Selecion
CL	CAS Latency
DDR	Dual Data Rate
DRAM	Dynamic Random Access Memory
ECS	Entity Component System
OOP	Object Oriented Programming (Programação Orientada a Objetos)
RAS	Row Address Selection
SRAM	Static Random Acess Memory
t_{RAS}	Active to Precharge Delay
t_{RCD}	Time RAS to CAS Delay
t_{RP}	Row Precharge Time

Índice

Índice de figuras

Índice de Códigos

Índice de tabelas

1 Introdução

Os vídeos jogos são uma boa fonte de entretenimento hoje em dia e fazem concorrência ao cinema. Durante a pandemia, o número de horas de jogo teve um aumento significativo devido ao facto de as pessoas passarem mais tempo em casa e existir uma necessidade de se entreterem.

Depois temos duas vertentes a emergirem dos jogos que é a realidade aumentada e a realidade virtual. Apesar de ambas serem bastante recentes já existem grandes espectativas para o futuro de ambas as tecnologias.

Os jogos demoram bastante tempo a serem desenvolvidos e existem muitos aspetos que têm um grau de complexidade elevado (nomeadamente a questão de renderização, física, inputs, etc.) que seria impraticável desenvolver estes aspetos de raiz cada vez que se pretende fazer um jogo. Para poupar tempo e facilitar o desenvolvimento dos jogos, existem os motores de jogo que já implementam muitas das funcionalidades necessárias para o desenvolvimento de um jogo permitindo assim uma maior facilidade no desenvolvimento.

BitEngine é um motor de jogos 2D que pretende aliviar a gestão da janela, gestão dos recursos, gestão de input, gestão de cenários e outros elementos que inferem com a jogabilidade. A BitEngine é um projeto individual e como tal não pretende competir outros motores de jogo no mercado (como Unity e Unreal Engine) no entanto já permite o desenvolvimento dos jogos com maior facilidade do que se não utilizar nenhum motor de jogos.

O capítulo que se segue detalha melhor o que é um motor de jogo, quais são motores de jogo mais conhecidos no mercado, detalha o que é Entity Component System, fala com mais detalhe o que é BitEngine e por fim menciona e justifica as tecnologias utlizadas para a implementação de BitEngine.

O capítulo seguinte é dedicado à implementação de BitEngine e no final desse capítulo é feito a medição e comparação de performance relativo ao OOP.

2 Enquadramento do Projeto

Muitas das grandes empresas de jogos têm uma equipa dedicada apenas ao desenvolvimento de motor de jogo. Por exemplo, Ubisoft [1]para o desenvolvimento dos seus títulos de Assassin's Creed[2] desenvolveu um motor de jogo chamado Anvil [3]. Este motor de jogo entra na categoria de motores de jogos que são privados e/ou proprietários, o que significa que não são de uso público. No entanto existem outros motores que já são de uso público como é o caso de Unity e Unreal Engine.

2.1 Unity

Unity[4] é um dos motores de jogo gratuitos mais conhecidos. Tem alguma curva de aprendizagem, mas em geral é muito fácil e rápido desenvolver jogos e/ou protótipos para testar novas ideias.

Apesar de ser um motor que é recomendado para toda a gente que entra na área de desenvolvimento, Unity já é uma ferramenta profissional que trata de muitos sistemas (como a física, renderização, som, input, etc.) sem que seja o desenvolvedor do jogo estar a preocupar-se como implementar cada um destes sistemas e sim focar-se mais nas leis do jogo. O seu uso é gratuito até que se lança o primeiro jogo para o mercado.

Com Unity também é possível criar jogos para diversas plataformas e fazer as migrações de uma plataforma para outra com pouco ou nenhuma alteração no código. O desenvolvimento dos jogos é feito com a linguagem C#.

Alguns dos títulos mais conhecidos desenvolvidos com Unity são: Kerbal Space Program[5], Hearthstone: Heroes of Warcraft[6], Wasteland 2[7], Rust[8], Escape plan

[1] Uma das maiores empresas de jogos que opera a escala mundial. Site oficial: https://www.ubisoft.com/en-us/
[2] É uma série de títulos bem-sucedidos da Ubisoft. Site oficial: https://www.ubisoft.com/en-us/game/assassins-creed
[3] Mais informações sobre Anvil em https://en.wikipedia.org/wiki/Ubisoft_Anvil
[4] Site oficial da Unity: https://unity.com/
[5] Site oficial: https://www.kerbalspaceprogram.com/
[6] Site oficial: https://playhearthstone.com/en-us
[7] Site oficial: https://www.inxile-entertainment.com/wasteland2
[8] Site oficial: https://rust.facepunch.com/

2.2 Unreal Engine

Unreal Engine[9] é um motor de jogo muito mais complexo de Unity, no entanto foca-se bastante na performance e nos gráficos. As tecnologias que implementam são sempre de ponta e otimizadas ao máximo. Geralmente este é um motor de jogo que está à frente tecnologicamente em comparação aos outros motores de jogo que tenham uso gratuito. No entanto, este é um motor cujo a sua aprendizagem não é fácil, é possível fazer alguns jogos sem programar, no entanto, não ira tirar o máximo proveito do daquilo que o motor tem para oferecer. As grandes empresas de jogos que optam por não desenvolver o seu próprio motor de jogo costumam optar por este motor de jogos. O desenvolvimento dos jogos é feito em C++ (ou blueprints para aqueles que não programam).

Alguns dos títulos mais conhecidos de Unreal Engine são: Chivalry 2[10], Kingdom Hearts III[11], Borderlands 3[12], Gears 5[13], Killing Floor 2[14], Batman: Arkham City[15], Days Gone[16] e Star Wars Jedi: Fallen Order[17].

2.3 BitEngine

BitEngine é um motor de jogo com uma escala muito reduzida. Serve apenas para fazer jogos 2D. O desenvolvimento dos jogos tem que ser feito em C++ pois BitEngine funciona como uma biblioteca simples que:

- Faz a gestão da janela (inicializa a janela de jogo, faz o redimensionamento da mesma, alteração do título e modo de janela (fullscreen ou windowed))
- Faz a gestão dos cenários, onde cada cenário é um contexto a que as várias entidades pertencem. Por exemplo: Menu Principal, Nível 1, Nível 2, etc.

[9] Site oficial: https://www.unrealengine.com/en-US/
[10] Site oficial: https://chivalry2.com/
[11] Site oficial: https://www.kingdomhearts.com/3/pt/home/
[12] Site oficial: https://borderlands.com/en-US/
[13] Site oficial: https://www.gears5.com/
[14] Site oficial: https://www.killingfloor2.com/
[15] Site oficial: https://www.warnerbros.com/games-and-apps/batman-arkham-city
[16] Site oficial: https://www.playstation.com/en-us/games/days-gone/
[17] Site oficial: https://www.ea.com/games/starwars/jedi-fallen-order

- Gestão de Input, ou seja, indica qual o estado das teclas do teclado e dos botões do rato. Ainda indica quais as coordenadas do cursor do rato no ecrã e qual foi o movimento em relação à frame anterior.
- Gestão do ciclo principal. Ou seja, BitEngine é que tem o controlo sobre o ciclo principal do jogo e decide quando é para atualizar as entidades e quando é para fazer o rendering das entidades.
- Gestão das entidades e dos sistemas. BitEngine fornece mecanismos para indica quais são os sistemas importantes para serem executados num determinado cenário bem como criar (e apagar) as entidades necessárias para esse mesmo cenário. Isto é implementado com base em Entity Component System.
- Gestão de processamento paralelo. Apesar de internamente BitEngine ser Single Core, fornece um mecanismo que permite executar tarefas de forma paralela de forma a aproveitar os vários Cores disponíveis.

2.4 Acesso à memória

Os padrões de acesso à memória têm um grande impacto na performance de um Software. Por isso muitas das decisões tomadas durante o desenvolvimento de BitEngine derivam deste facto. Esta secção dedica-se ao estudo e análise de performance da memória principal com o objetivo de compreender os padrões de acesso e como tirar proveito das mesmas.

2.4.1 DRAM

O mercado de hoje em dia é constituído maioritariamente por SRAM (Static RAM) e DRAM (Dynamic RAM). SRAM é muito mais eficiente e fornece as mesmas funcionalidades que DRAM, no entanto têm um custo de desenvolvimento muito mais elevado. Por esse motivo este tipo de tecnologia é encontrado em memória cache ou em equipamentos que requerem um elevado desempenho (ex.: equipamentos de rede). Por outro lado, DRAM é utilizado na memória principal e por esse motivo é a tecnologia que se analisa neste documento.

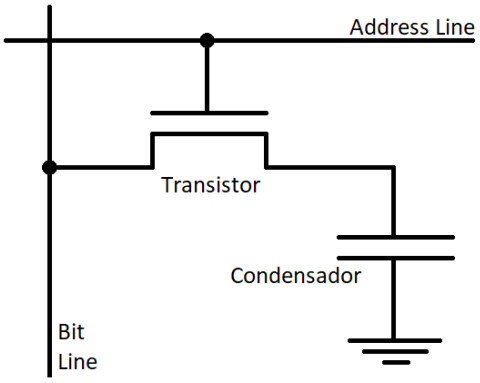

Figura 1 Circuito de uma célula de DRAM

A Figura 1 Circuito de uma célula de DRAM mostra o circuito eletrónico que forma uma célula de DRAM. O componente mais importante no circuito é o condensador pois é responsável pelo armazenamento do bit lógico. O condensador apesar de tornar o circuito muito mais simples (comparado ao SRAM) e ter um custo de desenvolvimento mais barato tem alguns problemas:

- O condensador perde a carga ao longo do tempo. Por isso é preciso recarregar o condensador de vez em quando para que o estado prevaleça. Durante o tempo de recarga as operações de leitura são proibidas.
- As operações de escrita têm um tempo associado. Isto é, quando se pretende guardar um bit lógico é preciso dar tempo ao condensador para que este carregue (ou descarregue).
- As operações de leitura também têm um tempo associado. Isto porque o estado da célula não é acessível diretamente. O valor quando sai da célula tem que passar por um amplificador que por sua vez descodifica o valor lógico da célula.

2.4.2 Acesso à DRAM

Para aceder aos dados é preciso indicar o seu endereço. É completamente impraticável ter uma linha de endereço para aceder a um endereço específico. Isto requeria ter 2^{32} linhas de endereço para aceder a 4GB. Por isso os endereços são codificados e requerem o uso de desmultiplexer para desmultiplexar os endereços. Desta forma, de 2^{32} linhas de endereço passamos a ter 32 linhas de endereço. No entanto como o espaço físico é um recurso precioso, existe uma necessidade de reduzir este número ainda mais. Quando o controlador de memória pretende enviar o endereço à RAM, este parte o endereço em duas partes e envia cada parte um de cada vez. Assim passamos a ter 16 linhas de endereço para aceder a 4GB. Internamente como as células estão agrupadas

numa matriz, uma metade do endereço indica a linha e outra metade do endereço indica a coluna. Isto por sua vez tem implicância no desempenho ao acesso à memória.

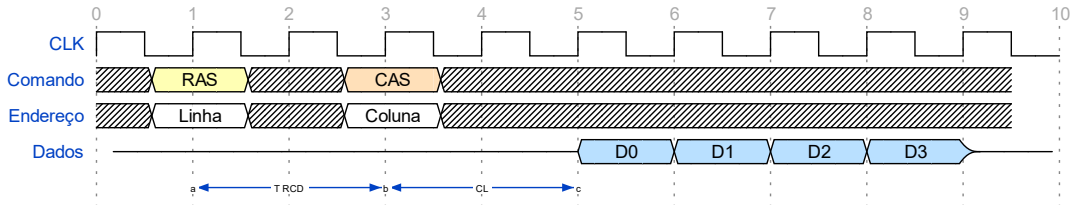

Figura 2 Tempos de acesso à memória

A Figura 2 Tempos de acesso à memória mostra a sequência típica para aceder a 4 bytes (no caso das tecnologias DDR a única coisa que muda é que num único ciclo é possível transmitir 2, 4 ou mais bytes). É primeiro indicado a linha com o comando RAS (Row Address Selection) juntamente com o respetivo endereço. Depois é indicado a coluna com o comando CAS (Column Address Selection) juntamente com o seu endereço. No entanto, existe um tempo t_{RCD} (Time RAS-to-CAS Delay) que a memória necessita para assimilar internamente a linha, só depois deste tempo é que a coluna pode ser definida (2 ciclos no exemplo anterior). O mesmo acontece para a coluna, existe um tempo CL (CAS Latency) que é o tempo que a memória necessita para assimilar internamente a coluna. No final destes tempos é que os dados estão disponíveis. Isto significa que em 8 ciclos apenas 4 ciclos são efetivamente leituras de dados.

Existem mais dois tempos de espera associados ao endereçamento e ambos relacionados com o carregamento da linha. Para se carregar uma nova linha é primeiro preciso desativar a atual. Existe um tempo associado a este processo, no entanto é possível mitigar o seu impacto pois é permitido efetuar leituras durante este tempo. A Figura 3 Tempos de acesso à memória com precharge mostra a leitura de 4 bytes onde os bytes menos significativos não estão contíguos em relação aos mais significativos o que implica alteração da linha e da coluna. O tempo t_{RP} (Row Precharge Time) é de 3 ciclos na figura e é possível ver que durante esse tempo é efetuado uma leitura. No entanto ainda existe 1 ciclo de espera antes de se carregar uma nova linha.

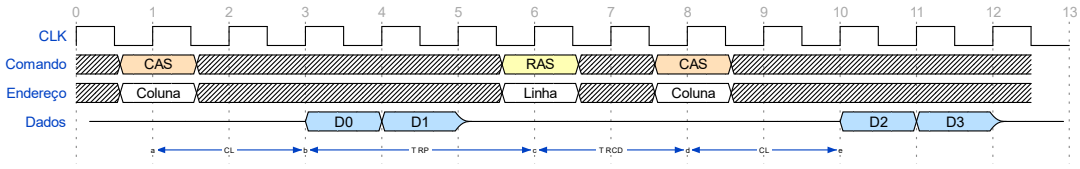

Figura 3 Tempos de acesso à memória com precharge

O último tempo é designado por t_{RAS} e é o número de ciclos de espera sem ser possível efetuar outro precharge. Por outras palavras, é o tempo (em ciclos) obrigatório que uma linha tem que estar com um determinado valor. Este tempo geralmente costuma ser 2 ou 3 vezes maior que t_{RP}. Os fabricantes costumam anunciar estes tempos no formato w-x-y-z. Por exemplo a memória RAM Crucial Ballistix RGB 64GB (2x32GB) DDR4-3200MHz CL16 (PC Diga, 2021) anuncia os seguintes tempos 16-18-18-36 e significam o seguinte:

Tabela 1 – Tempos de acesso

w	16	CAS Latency (CL)
x	18	RAS-to-CAS Delay (t_{RCD})
y	18	RAS Precharge (t_{RP})
z	36	Active to Precharge Delay (t_{RAS})

Em termos práticos, isto significa que se os dados estiverem dispersos pela memória a probabilidade de esperar vários ciclos para obter os dados é elevada e existe uma penalização na performance. Enquanto se os dados estiverem contíguos a penalização é drasticamente reduzida.

2.4.3 Teste de Performance

O teste de performance baseia-se num array de 12MB e aceder a cada um dos elementos (sendo cada elemento 1 byte). Array apesar de ser sequencial é tratado como uma matriz.

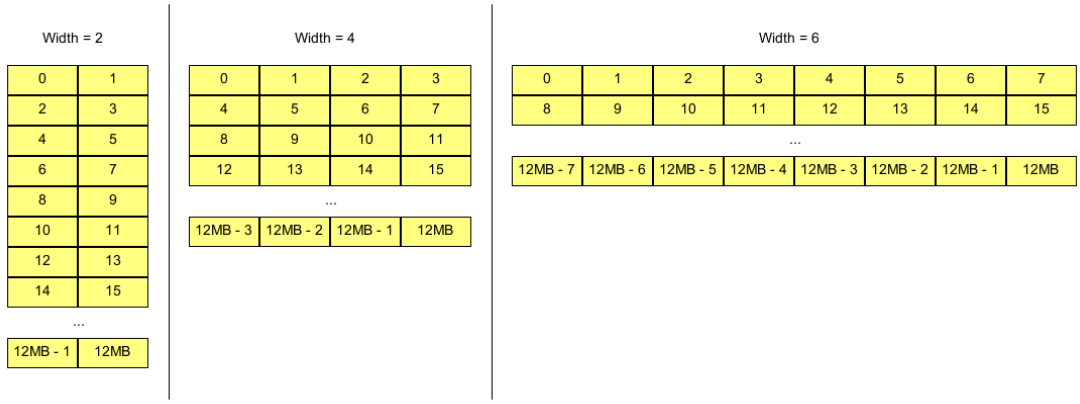

Figura 4 Dimensões das matrizes

Para cada teste muda apenas a dimensão da matriz (mas mantem sempre os 12MB).

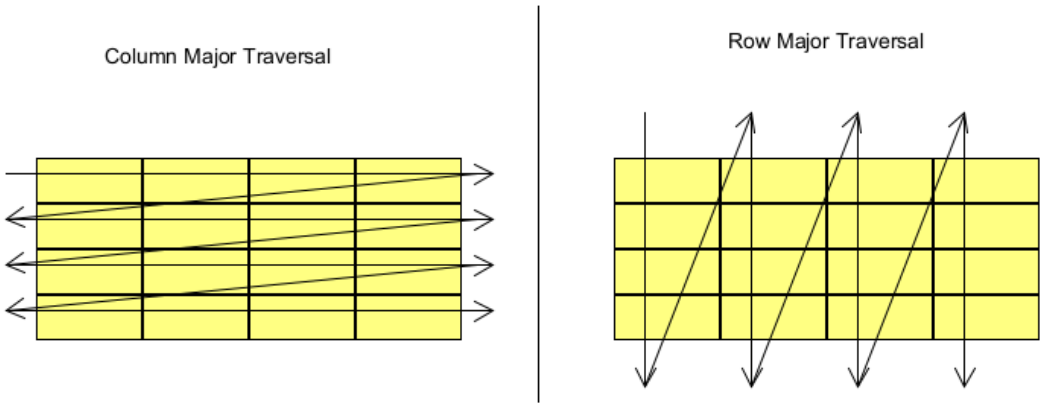

Figura 5 Formas de percorrer a matriz

A matriz é sempre percorrida em coluna e em linha. Estes dois factos causam diferentes padrões de acesso à memória. Por exemplo, em *Row Major* com uma largura de 4 bytes é lido 1 byte de 4 bytes em 4 bytes. A Figura 6 Tempos de acesso em Column Major e Row Major face à largura da matriz mostra o tempo medido para aceder a todos os bytes individualmente.

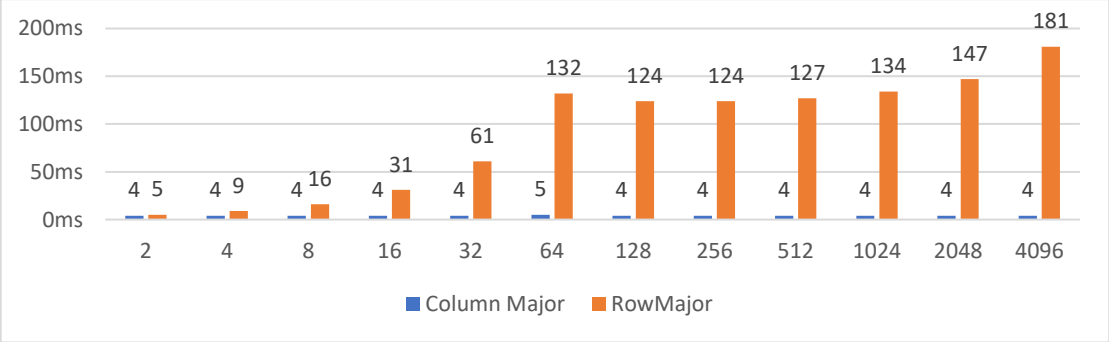

Figura 6 Tempos de acesso em Column Major e Row Major face à largura da matriz

Várias coisas se podem concluir com estes resultados. Primeiro, o acesso em *Column Major* mantêm-se sempre constante devido ao facto de que esta forma satisfazer as condições para o bom uso da memória cache. Para além disso, o mecanismo de prefetch também auxilia este resultado. Por outro lado, quando o acesso é feito em Row Major existe grandes penalizações de performance. Isto deve-se ao facto de má utilização da memória cache. Na máquina em que os testes foram executados, a linha de cache tem o tamanho de 64 bytes, o que significa que quando a largura da matriz é 2 bytes, existe 1 byte que é desperdiçado. Quando a largura é 64 bytes significa que 63 bytes são

desperdiçados a cada linha de cache. O que significa que o segundo nível de cache é acedido. Isto é visível na Figura 6 Tempos de acesso em Column Major e Row Major face à largura da matriz entre as larguras de 64bytes e 2048 bytes. Cache de L2 é acedido com a maior frequência e por esse motivo existe penalização de desempenho.

2.4.4 Conclusão

Em suma, se os dados estiverem dispersos em memória, não é possível utilizar a memória cache de forma eficiente e por isso mais acessos à memória principal serão feitos que por sua vez é uma operação que demora tempo e leva a que o CPU fique muito tempo parado a espera dos dados. Por outro lado, se os dados estiverem contíguos em memória e o seu acesso for sequencial, consegue-se tirar maior proveito da memória cache e do mecanismo de prefetch reduzindo as penalizações e aumentando o desempenho do CPU.

2.5 Entity Component System

Entity Component System é um padrão de desenvolvimento implementado em jogos. Segue o princípio de composição sobre herança o que significa que as entidades são definidas não pelo seu "tipo", mas por um conjunto de dados mais conhecidos como componentes que são associados às entidades o que permite uma maior flexibilidade. Os sistemas são responsáveis por fazer o processamento dos dados escolhendo que entidades estão interessados em processar dependendo dos componentes que os constituem. Por exemplo, um sistema de físico atua sobre todas as entidades que têm os componentes de massa, velocidade e posição. As entidades podem ter um comportamento dinâmico atribuindo/removendo componentes. Isto remove a ambiguidade existente que existem com grandes hierarquias de OOP.

2.5.1 Características de ECS

ECS consiste em 3 componentes:

- **Entidades**:
    - As entidades funcionam como um identificador. Servem para identificar quais os componentes que em conjunto formam uma entidade no jogo.
- **Componentes**:
    - Os componentes são apenas dados (ex.: vida, velocidade, gravidade, posição, etc.) que são guardados em arrays. Os componentes não têm

qualquer comportamento associado. Um determinado componente é identificado por uma entidade.

- **Sistemas**:
    - Os sistemas são o que atuam sobre os componentes interessados. Por exemplo, um sistema gravitacional pode atuar sobre os componentes "posição", "velocidade" e "gravidade". Os sistemas geralmente não têm a necessidade de saber a quem um determinado componente pertence.

2.5.2 Vantagem de ECS sobre OOP

Supondo que temos os seguintes objetos durante o desenvolvimento de um jogo:

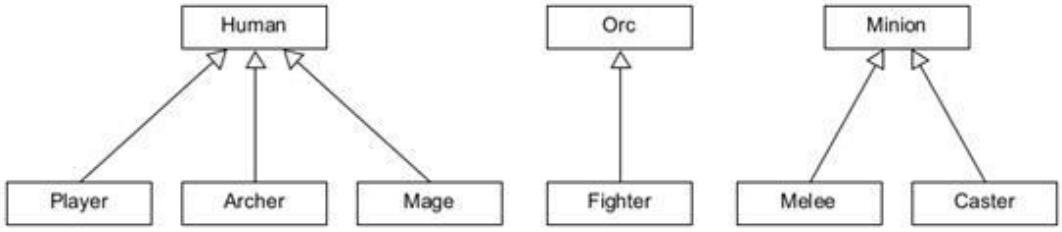

Figura 7 Diagrama inicial

Mas a uma dada altura decidiu-se que os Ogres também podem ser arqueiros. Para implementar tal solução fez-se a seguinte restruturação:

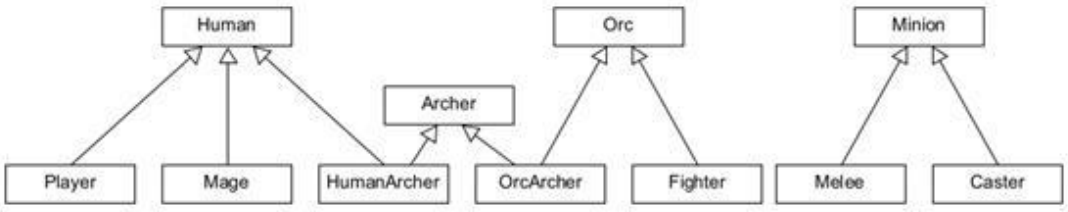

Figura 8 Diagrama com uma possível solução

Isto tem dois problemas associados. Primeiro problema é a herança múltipla. O segundo problema está relacionado com a dimensão do projeto. Num projeto pequeno uma alteração destas pode ser insignificante, mas em projetos de grande dimensão fazer este tipo de alterações não é viável.

Com ECS a abordagem é diferente. Podemos pensar numa tabela onde as colunas representam os componentes e as linhas representam as várias entidades do jogo. O "X" de cada célula indica se uma determinada entidade tem um determinado componente.

	Orc	Human	Minion	Archer	Fighter	Mage	Melee	Caster	Player
Entity 0	X			X					
Entity 1		X		X					
Entity 2			X					X	X
...									
Entity N		X				X			

Figura 9 ECS como uma tabela

Desta forma é possível fazer quaisquer tipos de combinação. Podemos dizer que um Ogre pode ser arqueiro e mago ao mesmo tempo. Também é possível, em runtime, dizer que esse mesmo Ogre perdeu as suas capacidades mágicas e retirar esse componente.

Isto permite uma maior flexibilidade para gerir as várias entidades.

2.5.3 Archetypes

Imaginemos o seguinte caso onde existem 6 entidades e 3 componentes:

	Entity 1	Entity 2	Entity 3	Entity 4	Entity 5	Entity 6
Positions	X		X	X		X
Healths			X	X		
Velocities	X					X

Figura 10 ECS dispersão dos dados

Um sistema de física que esteja interessado nos componentes de Posições e Velocidades vai processar esses mesmos arrays. O problema é que nem todas as entidades têm estes componentes. Isto implica duas, o carregamento da memória cache com dados desnecessários e saltos para endereços de memória distintos mais frequentes. Isto acontece porque os dados não estão verdadeiramente contíguos em memória.

Para resolver este problema recorre-se a Archetypes. Archetypes são um conjunto de arrays de componentes que estão sempre contíguos em memórias. Pegando no caso anterior, o equivalente utilizando Archetypes fica de seguinte maneira:

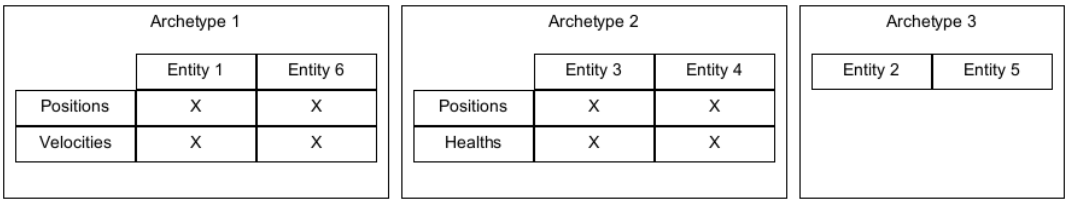

Figura 11 ECS com Archetypes

Archetype 1 apenas tem os componentes de Posições e Velocidades, Archetype 2 tem de Posições e Vidas e o Archetype 3 não tem quaisquer componentes. Caso seja retirado o componente de Velocidade à Entidade 6, seria criado um novo Archetype só com o componente de Posições, Archetype 1 deixaria de ter os componentes da Entidade 6 pois estes passavam a ser guardados no novo Archetype criado. Desta forma os componentes estão sempre contíguos em memória.

2.5.4 Potenciais otimizações

2.5.4.1 Processamento paralelo

Como os dados estão em arrays, é muito fácil e intuitivo criar e utilizar algoritmos de processamento paralelo o que nos permite tirar o proveito dos Cores do computador aumentando assim o desempenho do jogo.

2.5.4.2 SIMD

Os processadores de hoje em dia têm capacidade de fazer operações numa única instrução a 4, 8 e 16 valores diferentes. Mas para tal é primeiro preciso fazer uma preparação dos dados, depois faz-se a operação (tem que ser a mesma operação para todos os valores) e por fim guardar os resultados da operação o que significa que é preciso fazer um pouco de trabalho do lado do processador. No com este trabalho extra geralmente existem aumentos de performance. Com ECS os dados vêm em arrays o que significa que já vêm preparados para serem executados com operações SIMD, logo não existe esse esforço extra para preparar os dados. O mesmo acontece com os resultados, como estes têm que ser guardados em arrays, a escrita destes pode ser feita diretamente nas mesmas.

Devido à tecnologia implementada no BitEngine, é possível tirar o proveito das instruções SIMD.

2.6 Tecnologias implementadas

A implementação de BitEngine é feita em C++ pelo facto de esta ser uma linguagem de alto nível, mas ao mesmo tempo permitir ter acesso a coisas de baixo nível. Para alem disso, a flexibilidade que C++ permite para gestão da memória é fundamental para a implantação de ECS e tal como visto anteriormente, é possível obter melhores performances com bons padrões de acesso à memória e para tal esta tem que estar bem organizada.

Para a gestão da janela, renderização, input, etc. utilizou-se a biblioteca SDL2 pelo simples facto de haver uma familiarização com esta biblioteca e por ser de uso gratuito. Para alem desta biblioteca ainda é utilizado SDL2_image para carregar as imagens para a memória.

3 Trabalho de Projeto

3.1 Implementação de BitEngine

Tal como referido anteriormente BitEngine foi desenvolvido em C++ e com o uso das bibliotecas SDL2 e SDL2_image. Para recordar, BitEngine tem os seguintes objetivos:

- Gestão da janela
- Gestão e renderização de texturas
- Gestão de input (rato e teclado)
- Gestão de cenários
- Gestão de entidades
- Gestão do ciclo principal
- Gestão de processamento multithread

3.1.1 Diagrama de classes

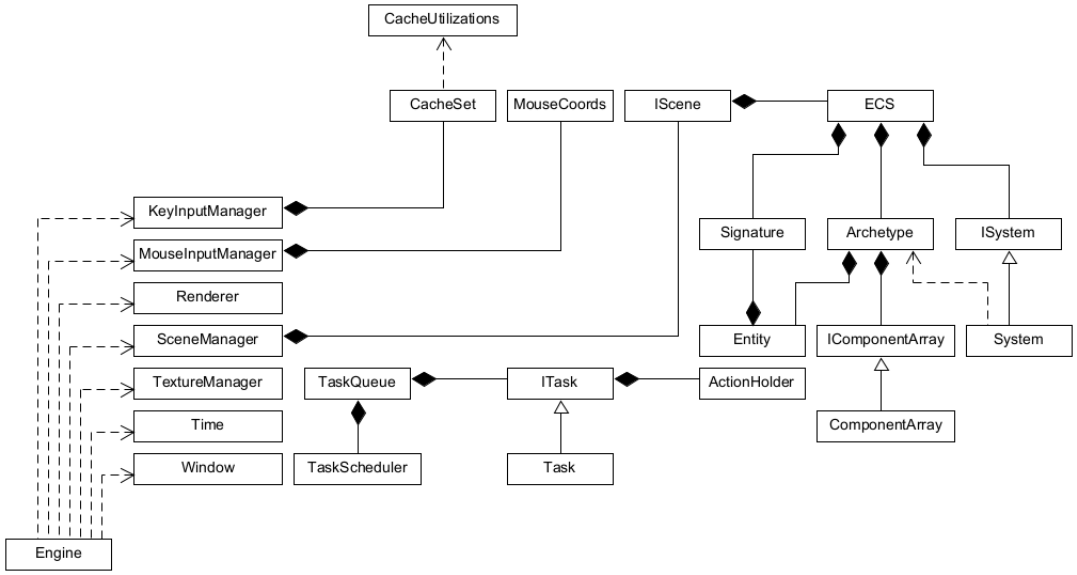

Figura 12 Diagrama de classes de BitEngine

Em seguida segue uma breve descrição sobre estas classes. Na secção seguinte estas classes são mais detalhadas consoante o seu uso na BitEngine.

3.1.1.1 Class Engine

Esta é a classe principal para pode utilizar a BitEngine. Os jogos terão que herdade desta class. Internamente esta classe é responsável por tudo o que acontece na engine.

É nesta clases que é feita toda a inicialização, gestão do loop principal, sincronização entre vários componentes da BitEngine e a finalização.

3.1.1.2 Class Window

Esta classe é responsável pela gestão da janela. Isto é, pelo seu modo, resolução, título e pelo número máximo de FPS.

3.1.1.3 Class Time

Esta classe é responsável por fazer a gestão do tempo. Serve para indicar quanto tempo passou desde o início do jogo e para indicar a diferença de tempo entre a última frame e a frame atual.

3.1.1.4 Class TextureManager

Esta classe é responsável por fazer a gestão das texturas. Isto é, é responsável por carregar as imagens, convertê-las para SDL_Texture. Para além disso, é feito o caching das texturas o que significa se carregar duas vezes a mesma imagem essa imagem só será carregada efetivamente uma vez, à segunda vez é devolvido a textura que está em cache. Por último esta classe também é responsável por desalocar toda a memória das texturas carregadas.

3.1.1.5 Class SceneManager

Esta classe é responsável por gestão dos cenários do jogo. Ou seja, indica qual é o cenário atual que deve ser atualizado e renderizado. Para além disso controla o tempo atividade do jogo, isto é, é responsável por indicar quando é que o jogo deve terminar.

3.1.1.6 Class Renderer

Esta é uma classe de abstração ao SDL_Renderer que é utilizado para vários efeitos. Por isso esta classe é um Singleton de forma a tornar SDL_Renderer global e seguro.

3.1.1.7 Class MouseInputManager

Esta classe é responsável por guardar todos os registos do rato. Isto é, as coordenadas atuais do rato, a diferença de movimento da última frame para a frame atual e o estado dos botões do rato. Esta classe é um Singleton de forma que possa ser acedida de forma global e segura.

3.1.1.8 Class KeyInputManager

Esta classe é responsável por guardar o registo das teclas pressionadas no teclado. Tem a capacidade de indicar se a tecla foi pressionada num determinado instante, se está a ser pressionada durante um logo período de tempo ou se foi libertada num determinado instante. Para a sua implementação esta classe utiliza CacheSet. Esta classe é um Singleton de forma que possa ser acedida de forma global e segura.

3.1.1.9 Class CacheSet e CacheUtilization

CacheSet é uma estrutura de dados cujo seu objetivo é guardar os dados numa única linha de cache, de forma a ser eficiente, e cujo seus valores não sejam repetidos.

CacheUtilization especifica qual a taxa de ocupação que deve ser feito da memória cache. As possibilidades são 100%, 50% ou 25% da memória cache.

3.1.1.10 Class IScene

Esta é uma classe abstrata que tem de ser herdada por todos os cenários do jogo e implementar todos os seus métodos. Os cenários são uteis para fazer a diferenciação de menu principal, pausa, nível 1, nível 2, etc.

3.1.1.11 Class ECS

Esta classe centraliza Entity Component System. É através desta classe que se criam novas entidades, associam-se componentes às entidades e se registam novos sistemas. Internamente esta classe garante que os componentes das entidades se encontram em Archetypes corretos.

3.1.1.12 Class Signature

Esta classe é responsável por identificar os Archetypes e os respetivos componentes que os Archetypes guardam. A cada componente é atribuído um ID, esse ID é utilizado como um índice num array de bits para determinar se um determinado Archetype guarda ou não um determinado componente.

3.1.1.13 Class Archetype

Esta classe é responsável por guardar os diversos arrays de componentes diferentes bem como uma referência para as entidades. Também garante que todos os array de componentes são array contíguos. Se um novo componente é inserido este vai para a última posição do array. Quando um componente a meio do array é retirado, o último

elemento do array ocupa o índice do elemento retirado mantendo assim os arrays sempre contíguos.

3.1.1.14 Class Isystem e System

Classe Isystem é a classe base que guarda a assinatura do sistema (o que por sua vez identifica quais os componentes a que o sistema está interessado).

A classe System é uma classe template que identifica os diversos componentes sobre os quais atua. Todos os sistemas herdam desta class.

3.1.1.15 Class Entity

Esta classe identifica uma entidade do jogo. Para além disso guarda tem um Signature que identifica o Archetype a que pertence e tem um índice que indica qual é o índice nos arrays do Archetype dos seus componentes.

3.1.1.16 Class TaskScheduler

Esta classe é responsável pela gestão das threads existentes e por agendamento das várias tarefas executadas em threads diferentes. Cada tarefa é executada em uma thread. Se existirem 8 threads significa que podem ser executadas 8 tarefas ao mesmo tempo.

3.1.1.17 Class TaskQueue

Esta classe é responsável por ter uma fila única de tarefas a executar em threads diferentes. Primeira tarefa a entrar na fila é a primeira tarefa a ser executada.

3.1.1.18 Class Itask e Task

Itask é a classe abstrata para Task. Esta classe é responsável por executar uma tarefa em paralelo. Para além disso também é responsável por colocar na TaskQueue outras tarefas cujo seu input depende do output da tarefa executada.

3.1.1.19 Class ActionHolder

Esta classe é responsável por delegar a colocação de Task na TaskQueue para outras Tasks quando necessário.

3.1.2 Implementação de ECS

Nesta secção é explicada de forma detalhada a implementação de ECS bem como algumas das decisões tomadas ao longo da sua implementação.

3.1.2.1 Signature

Antes de entrar em grandes detalhes de ECS é primeiro necessário perceber o que são as Signatures e como funcionam.

Uma Signature é apenas um conjunto de bits que indica a presença de um determinado componente ou a sua ausência. A cada tipo de componente é atribuído um ID e esse mesmo valor acabar por determinar o índice correspondente na Signature que indica a sua presença. Vejamos o seguinte exemplo: Supondo que tempos os seguintes tipos de componentes com os seus respetivos Ids.

Tabela 2 – ID por tipo de component

Componente	ID
Posição	0
Velocidade	1
Vida	2

Isto implica que todas as Signatures terão 3 bits. Na tabela seguinte existem 3 Signatures:

Tabela 3 – Exemplo de Signatures

Signature	Bit 2	Bit 1	Bit 0
S0	0	1	0
S1	1	1	1
S2	1	0	1

A Signature S0 indica que a componente “Velocidade” faz parte da sua Signature enquanto as outras duas não.

A Signature S1 indica que todos os componentes fazem parte da sua Signature.

A Signature S2 indica que a componente “Vida” e “Posição” fazem parte da sua Signature, mas o componente “Velocidade” não.

Na implementação de ECS as Signature têm os seguintes objetivos:

- Identificar quais são as componentes que fazem parte de uma determinada Entidade
- Identificar quais são os tipos de componentes que um determinado Archetypes guarda
- Identificar quais são os componentes que um determinado Sistema está interessado em processar

A implementação de Signatures é feita através da implementação de estrutura de dados BitSet de C++ e é feito da seguinte maneira:

```
using Signature = std::bitset<MAX_NUM_COMPONENTS>;
```

Código 1 – Definição de Signature

3.1.2.2 Entity

As Entidades funcionam apenas como identificadores e são definidos da seguinte forma:

```
struct Entity
{
    std::size_t index;
    Signature signature;
};
```

Código 2 – Definição de Entity

Onde o atributo "signature" para além de identificar os componentes que pertencem à Entidade, mas também identifica a que Archetype essa Entidade pertence.

O atributo "index" identifica o índice de todos os componentes que fazem parte da Entidade no respetivo Archetype (explorado mais detalhadamente nas próximas secções).

3.1.2.3 ComponentArray

ComponentArray é responsável por guardar um conjunto de componentes do mesmo tipo em forma de array. Esta classe é uma classe template o que significa que `ComponentArray<Position>` irá guardar um array de posições e `ComponentArray<Velocity>` irá guardar um array de velocidades. No entanto existe um problema associado aqui. No exemplo anterior apesar de ambos serem

`ComponentArray` ambos são diferentes o que significa que é impossível ter um array de `ComponentArray`. Para resolver o problema, criou-se uma outra classe `IcomponentArray` que ser como uma classe basse para ComponentArray. Assim quando se pretender ter um array de `ComponentArray` isto pode ser alcançado através de um array de `IcomponentArray*` da seguinte forma:

```
IcomponentArray *componentes[10];
```

Código 3 – Array de IcomponenteArray

O exemplo anterior permite guardar 10 `ComponentArray` independentemente se é `ComponentArray<Position>` ou `ComponentArray<Velocity>`.

A classe IcomponentArray é uma classe vazio sem atributos. A classe ComponentArray é definida da seguinte forma:

```
template<typename T>
struct ComponentArray final : public IcomponentArrat
{
    T *21efault21es21;
};
```

Código 4 – Definição de ComponentArray

3.1.2.4 Archetype

A classe Archetype é responsável por guardar um conjunto de `ComponentArray` (que por sua vez guardam um conjunto de componentes) e as Entidades que têm esses mesmos componentes.

A classe Archetype é definida da seguinte forma:

```
class Archetype
{
    std::size_t m_count;
    std::unordered_map<std::size_t,IcomponentArray*>
m_components;
    EntityPtr *m_entities;
};
```

Código 5 – Definição de Archetpye

Onde o atributo “m_count” identifica o número de Entidades pertencentes ao Archetype.

O atributo “m_components” é uma tabela de hash onde a chave é o ID do componente e o valor é o ComponentArray*. Desta forma é possível o array correspondente a um determinado componente. Referindo à tabela de componentes da secção 3.1.2.1, teríamos a chave 0 que correspondia ao array de componentes de posições (`ComponentArray<Position>*`).

Por fim o último atributo “m_entities” é um array de referências para as entidades. Como Archetype tem o poder de alterar os atributos das Entidades é necessário guardar as referências das Entidades para as alterações sejam propagadas para todos o que têm uma referência para a mesma Entidade.

3.1.2.5 System

Os Sistemas são responsáveis por processar um conjunto de componentes que são do interesse deles. Por exemplo, um Sistema gravitacional poderia estar interessado nos componentes de Posição e Gravidade fazendo assim aplicar a gravidade a todas as Entidades. A implementação deste sistema é feita da seguinte forma:

```
struct GravitySystem final : public System<Position, Gravity>
{
    void update(const std::vector<ArchetypePtr> &22efault22es)
override;
    void render(const std::vector<ArchetypePtr> &22efault22es)
override;
};
```

Código 6 – Exemplo de definição de um Sistema

Existem dois aspetos a ter em conta aqui. Primeiro um sistema tem que herdade da classe `System` e ao fazer isso tem que indicar quais são os componentes a que está interessado em processar.

Depois tem que reescrever dois métodos (`update` e `render`) cujo seu argumento são todos os Archetypes que têm os componentes a que o sistema está interessado.

É de notar que a classe `System` é também uma classe template o que significa que também ela tem uma classe base `Isystem` pelos mesmos motivos que a classe `ComponentArray`. A classe System é defenida da seguinte forma:

```
template<typename …Queries>
struct System : public Isystem
{
     System() : Isystem(genSignature<Queries…>()) {}
     virtual ~System() {}

     virtual   void   update(const   std::vector<ArchetypePtr>
&archtypes) = 0;
     virtual   void   render(const   std::vector<ArchetypePtr>
&archtypes) = 0;
};
```

Código 7 – Definição de System

Os dois métodos são métodos abstratos o que significa que têm que ser implementados por quem herda desta class. Para além disso a classe herda de `Isystem` e ao chamar o construtor da classe gera uma Signature com base nos componentes interessados. A classe `Isystem` é definida da seguinte forma:

```
struct Isystem
{
     Isystem(Signature s) : signature(s) {}
     virtual ~Isystem() {}
     virtual   void   update(const   std::vector<ArchetypePtr>
&archtypes) = 0;
     virtual   void   render(const   std::vector<ArchetypePtr>
&archtypes) = 0;
     Signature signature;
     ECS *ecs{ nullptr };
};
```

Código 8 – Definição de Isystem

É de notar que tem um atributo Signature que identifica quais são os componentes a que o Sistema está interessado. Para além disso tem uma referência para uma classe `ECS` que permite gerir todo o mecanismo de Entity Component System (explicado em seguida). Este atributo é necessário pois permite aos sistemas criar novas entidades, removê-las ou até alterar os seus componentes.

3.1.2.6 ECS

Esta é a classe principal através da qual se faz toda a interação com Entity Componente System. É através desta classe que se identificam os vários sistemas, criam-se novas entidades, associam-se componentes às entidades e removem-se as entidades.

Esta classe tem dois atributos:

```
std::unordered_map<Signature, ArchetypePtr> m_archetypes{};
std::vector<std::pair<Isystem*,std::vector<ArchetypePtr>>>
m_systems;
```

Código 9 – Atributos de ECS

O primeiro atributo é uma tabela de hash que associa uma Signature a um determinado Archetype. O segundo atributo é um vetor de pares onde o primeiro elemento do par é o Sistema e o segundo elemento do par é um vetor de Archetypes. A razão para o segundo atributo é evitar fazer pesquisas todas as frames. Isto é, todas as frames é necessário indicar aos sistemas quais são os Archetypes do seu interesse e para tal é preciso fazer uma pesquisa por todos os Archetypes cujo a Signature seja compatível coma Signature do sistema. Para evitar estas pesquisas, cada vez que é criado um novo Archetype, ele é associado imediatamente aos sistemas interessados.

Existe um método chamado “update” e outro “render” que para todos os pares de m_systems chamam os seus respetivos métodos com os Archetypes correspondentes. Estes métodos são declarados e definidos da seguinte forma:

```
void update()
{
     for (auto const &pair : m_systems)
     {
          pair.first->update(pair.second);
     }
```

```
}

void render()
{
      for (auto const &pair : m_systems)
      {
            pair.first->render(pair.second);
      }
}
```

Código 10 – Definição de "update" e "render" de ECS

É da responsabilidade da classe Engine chamar estes dois métodos do ECS no loop principal.

3.1.3 Gestão de Cenários

Para facilitar a diferenciação do menu principal, nível 1, nível 2, etc. criaram-se os cenários. Para que seja criado um novo cenário, este tem que herdar de Iscene que é definido da seguinte forma:

```
struct Iscene
{
      Iscene() = 25efault;
      virtual ~Iscene() = 25efault;

      virtual void initialize() = 0;
      virtual void shutdown() = 0;

      ECS ecs;
};
```

Código 11 – Definição de Iscene

É de notar que os cenários que herdarem desta classe serão forçados a definir o método "initialize" e o "shutdown". É nestes dois métodos que fazem toda a inicialização e o fecho necessário. Para além disso existe um atributo "ecs" associado ao cenário.

A gestão dos vários cenários é feita através de uma classe Singleton chamada SceneManager. Está classe para além de gerir os cenários também indica se é para terminar o jogo ou não.

Quando é pretendido mudar para um cenário específico, isso é feito da seguinte forma:

```
SceneManager::getInstance().changeScene<MainMenuScene>();
```

Código 12 – Exemplo de mudança de cenário

E na próxima frame do jogo, SceneManager irá assumir esta mudança.

3.1.4 Loop Principal

A BitEngine controla o loop principal do jogo e tem a seguinte ordem:

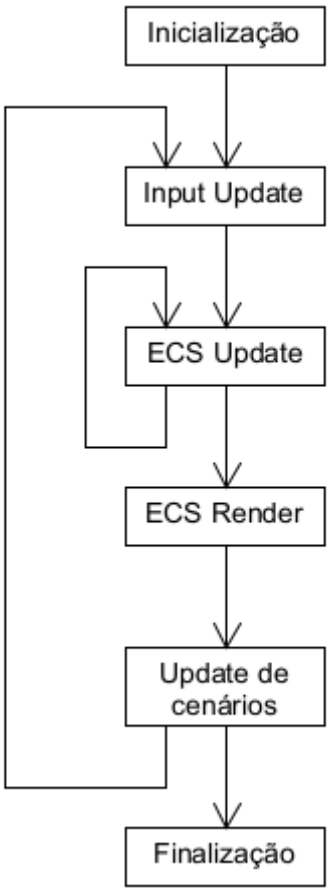

Figura 13 BitEngine Ciclo Principal

- Inicialização: é aqui que BitEngine inicializa SDL2, a janela e o sistema de multithread.
- Input Update: é aqui que é feito a atualização do estado dos inputs.

- ECS Update: é aqui que a classe Engine pede ao SceneManager o cenário atual e pega no ECS e chama o método "update" desse ECS. Este método pode ser chamado várias vezes na mesma frame (que depende do frame rate definido).
- ECS Render: a mesma coisa que no ECS Update mas aqui é chamado o ECS Render e é chamado apenas uma vez a cada frame. É aqui que é feito todo o rendering.
- Update de cenários: é aqui que é feito a mudança de cenários que seja necessário
- Finalização: é aqui que BitEngine finaliza o sistema multithread, a janela e o SDL2

3.1.5 Gestão de Input

Os eventos de SDL fornecem informação sobre quais as teclas que foram pressionadas e quais foram libertadas, mas, no entanto, não é informação suficiente para determinar o estado de um determinado input. Por isso existem dois Singletons chamados KeyInputManager e MouseInputManager que server para guardar o estado dos inpits bem como facilitar o seu acesso.

3.1.5.1 KeyInputManager

Esta classe serve para guardar o estado das teclas pressionadas. Para isso, é utilizado uma outra classe chamada CacheSet que permite com que um set esteja sem em uma linha cache diminuído assim o tempo de acesso a esse mesmo set. Para além disso, é possível indicar qual a taxa de ocupação do set em uma linha de cache utilizando CacheUtilization.

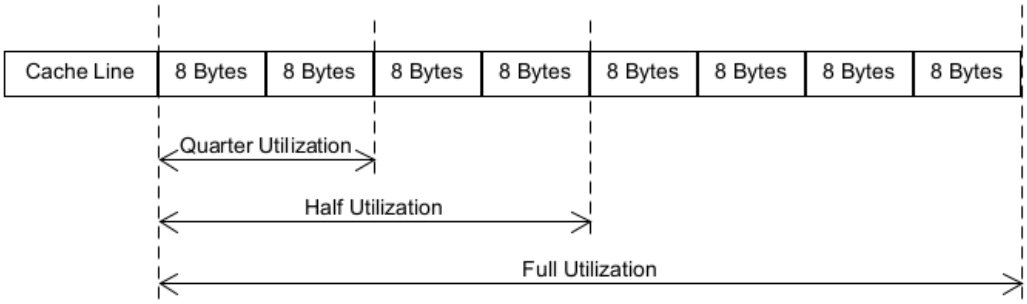

Figura 14 Utilização da linha de cache

O exemplo anterior, supondo que uma linha de cache é de 64bytes, podemos observar quais serão as dimensões de cada set. No case em que a utilização seja $\frac{1}{4}$ da linha de cache, teremos então 16 bytes para guardar os dados. No caso em que a utilização seja $\frac{1}{2}$ então teremos 32 bytes para guardar os dados. Por fim, quando a utilização da linha ache é total, teremos os 64 bytes para guardar os dados.

Como a representação das teclas é feita através de um Enum que ocupa 4 bytes, para $\frac{1}{2}$ da utilização da linha de cache, teremos então 8 teclas possíveis que podem ser guardadas em cache, o que é suficiente (não é comum encontrar jogos que requerem que mais do que 4 teclas sejam pressionadas ao mesmo tempo).

O estado das teclas é guardado em 2 CacheSets a $\frac{1}{2}$ da ocupação da linha de cache fazendo assim o uso total da linha de cache. Ambas são necessárias porque uma guarda o estado da tecla na frame atual e outra guarda o estado da tecla na última frame. Estes dois sets têm o nome de "m_currFrameKeys" e "m_oldFrameKeys" e funcionam da seguinte forma: quando uma tecla é pressionada, é inserida na "m_currFrameKeys", no início de cada frame o estado de "m_currFrameKeys" é copiado para "m_oldFrameKeys". Quando se pretende obter o estado de uma tecla, ambos os sets são observados. Podemos interrogar 3 estado diferentes de uma tecla: se a tecla foi pressionada, se a tecla está a ser pressionada e se a tecla foi libertada. Para obter o estado das teclas supondo que temos os seguintes valores em cada um dos sets:

m_currFrameKeys	W	A		
m_oldFrameKeys	A	D		

Figura 15 Estado de input

Então podemos deduzir que:

- A tecla "W" foi pressionada
- A tecla "A" está a ser pressionada
- A tecla "D" foi libertada

3.1.5.2 MouseInputManager

Para o caso do rato, é também utilizado um CacheSet para determinar o estado dos botões do rato da mesma forma como funciona para as teclas. Mas para além do estado das teclas do rato, MouseInputManager também guarda a posição do cursor na janela de jogo e qual é o movimento em relação à última frame.

3.1.6 Gestão de texturas

A classe TextureManager é um Singleton que é responsável pela gestão das texturas. Para além disso, evita que duas texturas iguais sejam carregadas para a memória fazendo assim caching das texturas. Isto é, se uma determinada textura já esteja em memória então quando se pedir para carregar outra vez a mesma textura, esta irá devolver a mesma textura que esteja em cache. Na finalização do BitEngine, todas as texturas são libertadas da memória. Também é possível fazer a libertação destas texturas a qualquer momento do jogo.

Para carregar as texturas, é utilizado a biblioteca SDL2_image que carrega uma textura para um formato `SDL_Surface*`, depois é necessário converter esse `SDL_Surface*` para `SDL_Texture*`.

É possível renderizar `SDL_Surface*` e assim não seria necessário estar a fazer conversões, mas o problema é que `SDL_Surface*` tem problemas de performance, por isso é que é feita a conversão.

3.1.7 Gestão de multithread

Apesar de BitEngine internamente não fazer uso de multithread, este permite o uso para os que tiverem interessados em processamento paralelo com o objetivo de aumentar a performance.

O seu funcionamento consiste numa fila (FIFO) de tarefas a executar.

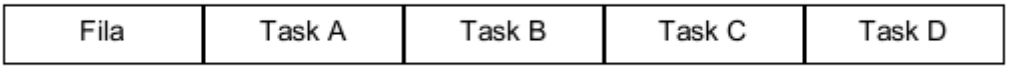

Figura 16 Fila de tarefas

Cada thread, retira da fila uma tarefa e executa-a. Mas existem operações mais complexas associada à gestão de multithread. Existem tasks que depende do resultado

de outras tasks. Na figura seguinte, a tarefa E depende do resultado da Tarefa B, por isso a tarefa E apenas será colocada na fila após o término da tarefa B. Neste exemplo, tarefa E apenas depende de uma única tarefa, mas pode depende de um número ilimitado de tarefas.

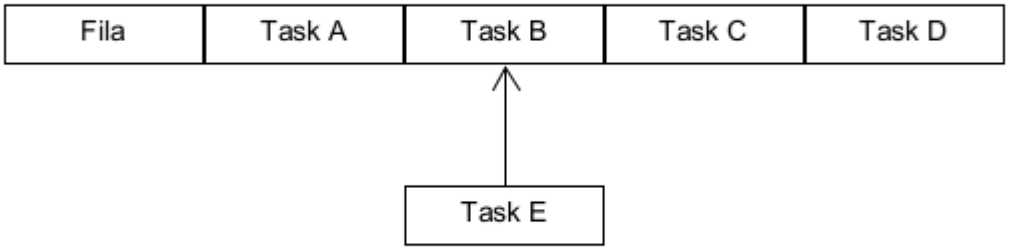

Figura 17 Fila de tarefas com dependência

A complexidade aumenta ainda mais quando uma tarefa devolve não um resultado em concreto, mas sim, uma outra tarefa. Voltado ao exemplo anterior, supondo que B na realidade devolva uma tarefa, assim quando B terminar, a tarefa E passa a depender dessa nova tarefa criada pela tarefa B e por sua vez só entra na fila depois desta nova tarefa terminar.

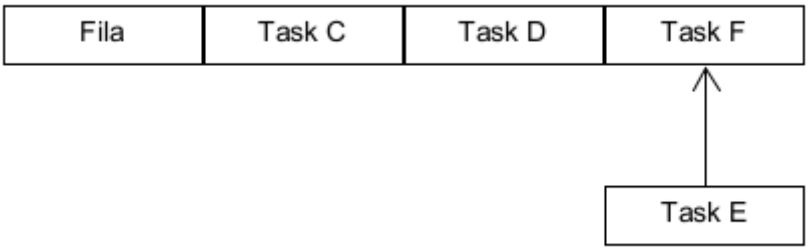

Figura 18 Nova fila de tarefas com dependência

A criação de tarefa é simples:

```
auto task1 = make_task([](){ return 4; });
auto task2 = make_task([]() { return 6; });
auto task3 = make_task([](int a, int b) { return a + b; }, task1,
task2);
std::cout << task3.result() << std::endl;
```

Código 13 – Exemplo de criação de Tasks

No exemplo anterior são criadas 3 tarefas. A criação de cada tarefa é apenas uma função lambda. A primeira tarefa devolve o valor 4. A segunda tarefa devolve 6. A terceira tarefa tem como argumento dois valores, devolve a soma desses dois valores e ainda depende do resultado da primeira tarefa e da segunda tarefa. Isto faz com que a tarefa 3 seja colocada na fila de execução após a execução das duas primeiras tarefas.

3.1.7.1 Task

A classe Task funciona como um wrapper às tarefas a executar e tem 3 componentes:

- A tarefa a executar
- Número de tarefas pendentes (contador)
- Referência para as tarefas dependentes da mesma

A Task só entra a fila de Tasks quando o número de tarefas pendentes for igual a zero. Este contador indica quantas tarefas pendentes ainda têm que terminar a sua execução. A referência para as tarefas pendentes server para quando a tarefa terminar a sua execução, decrementa o contador das tarefas dependentes de forma que estas possam ser colocadas na fila.

3.1.7.2 ActionHolder

Esta classe serve como um conector entre as tarefas pendentes e as dependentes. Quando uma tarefa termina, é através do ActionHolder que é feito a indicação para as tarefas dependentes de que a tarefa terminou e ao mesmo tempo envia os argumentos necessários para a execução da tarefa dependente.

3.1.7.3 TaskQueue

Esta classe é a classe que representa a fila de tarefas a executar. A fila é thread safe, o que significa que uma thread de cada vez é que pode retirar uma task da fila. Quando não existirem tarefas na fila, a thread é adormecida. Quando surgir uma tarefa, é acordada uma thread e é enviada a tarefa a executar.

3.1.7.4 TaskScheduler

Esta classe guarda todas as threads e a fila de tasks a executar. É através desta classe que as tarefas são colocadas na fila.

3.1.7.5 Nota final de gestão de multithread

Apesar de existir forma de gerir funções que correm em paralelo a BitEngine não faz o seu uso. Ou seja, BitEngine é single core. Um dos objetivos pretendidos para multithread era correr os vários sistemas não relacionados (ou seja, que não partilhem o interesse nos mesmos componentes ou que se partilhassem o acesso seria apenas de leitura) em threads diferentes para aumentar a performance e o uso dos vários Cores disponíveis. Por falta causa das restrições de tempo tal não foi possível implementar.

3.2 Medição de performance

Para medir a performance, foram criados dois jogos similares, mas um deles faz o uso do ECS do BitEngine (projeto chamado Castle Defender ECS) e o outro faz o uso dos objetos de OOP (projeto chamado Castle Defender OOP). A implementação dos projetos não é detalhada neste relatório, mas o código fonte pode ser encontrado no GitHub do projeto (em anexo).

As medições de performance baseiam-se nos FPS (Frames Per Second) dependendo da quantidade de entidades que existem por processar.

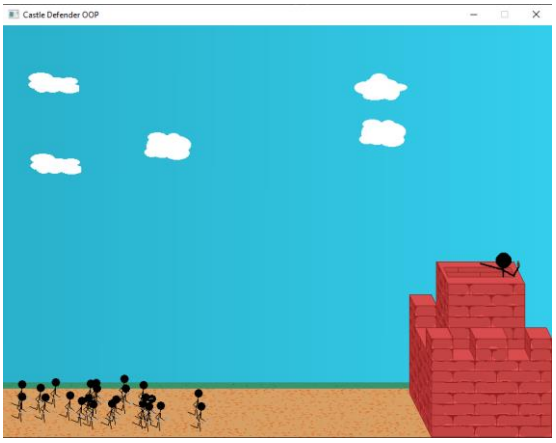

Figura 19 Jogo feito com OOP

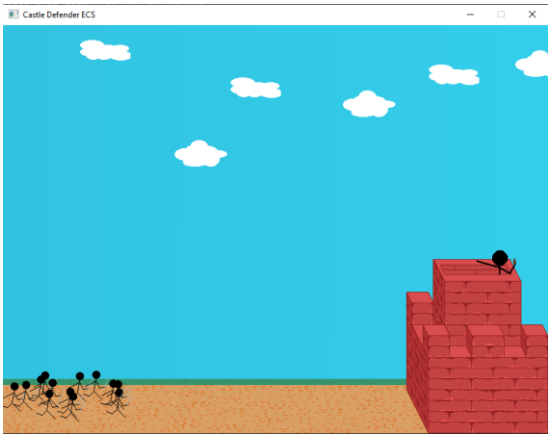

Figura 20 Jogo feito com ECS

Após a execução de ambos os jogos com quantidades de entidades diferentes temos os seguintes resultados:

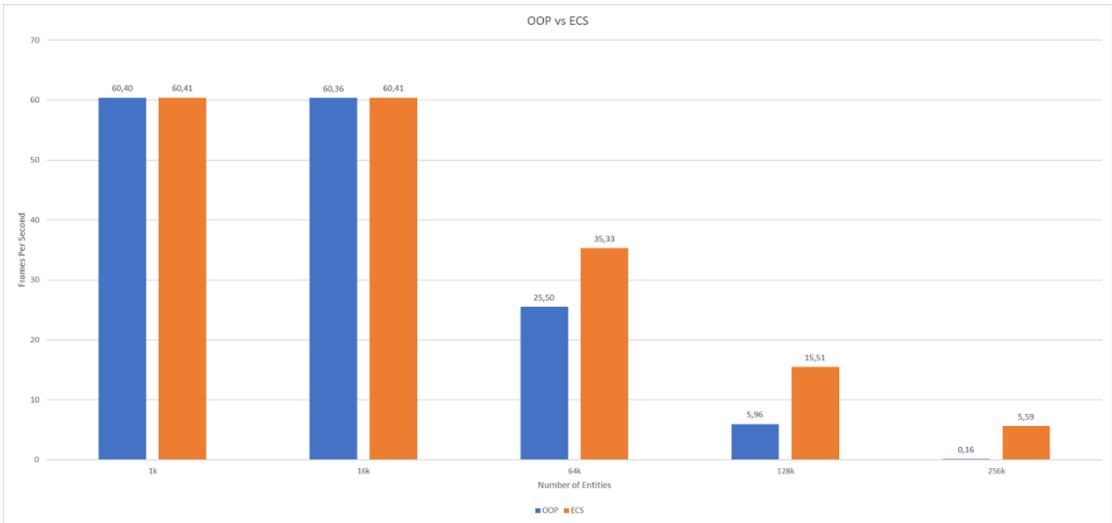

Figura 21 Gráfico de performance OOP vs ECS

É de notar que para 1024 entidades e 16384 entidades, a performance é semelhante e temos o máximo de FPS (que é definido na implementação).

Mas quando tempo 65536 entidades para serem processadas, a performance para ambas baixa drasticamente, mas enquanto na implementação de ECS os FPS ainda rondam os 35 na implementação de OOP temos a volta de 25 FPS. O que mostra uma melhoria em termos de performance. Para os restantes números de entidades os FPS baixam bastante em ambos os casos.

4 Conclusões

É seguro afirmar que BitEngine foi concluído com sucesso, a gestão da janela é feita de forma simples, bem como a gestão de input, ciclo principal, gestão das texturas, gestão dos cenários e a gestão de processamento paralelo. Para além disso, a implementação de ECS também foi um sucesso, é muito fácil criar novas entidades e atribuir/revogar componentes a estes de foram a que eles tenham comportamentos dinâmicos em runtime. A criação de sistemas também é simples e permite com que cada sistema esteja dedicado a um processamento específico sem ter a preocupação da existência de outros sistemas. Por fim, os resultados de performance são positivos, existe uma clara melhoria em relação aos métodos convencionais de programação (OOP neste caso) o que demonstra que os padrões de acesso à memória são efetivamente importantes.

Mas apesar de todo o sucesso, ainda existem alguns inconvenientes e aspetos que podem ser melhorados. Os Archetypes apesar de serem fundamentais no ECS têm uma complexidade extremamente elevada. Existem formas de reduzir essa complexidade e assim melhorar ainda mais o desempenho de BitEngine. Esta complexidade é o motivo pela qual os compiladores terão dificuldade em implementar operações SIMD automaticamente, o que significa que para tirar o proveito de SIMD o programador terá que o fazer explicitamente. Para além disso, a forma como ECS está implementado não é possível com que um Archetype, Entity ou System operem sobre dois componentes do mesmo tipo. Isto é uma entidade não pode ter dois componentes do tipo `Velocidade` o que pode ser visto como inconveniente e reduz a flexibilidade de desenvolvimento dos jogos. Um outro aspeto que é alvo de melhoria é o facto de que os arrays de componentes de todos os Archetypes ocupem um número igual de elementos, mas, no entanto, existem muitas situações em que existe apenas uma entidade que tenha uma combinação especifica de componentes. Um exemplo: um Archetype que guarde os seguintes tipos de componentes: `Player`, `Position`, `Velocity` e `Texture`. Isto implica que para cada tipo irá existir um array com 1024 (ou mais) elementos. Para um jogo de Multiplayer isto pode ser desejado, mas para um jogo Singleplayer isto é um enorme desperdício de memória porque de todos esses elementos apenas seria necessário um elemento. Um último aspeto que pode ser melhorado no BitEngine é o facto de este ser apenas single Core. Apesar de permitir processamento paralelo a essência do BitEngine é apenas single Core. Para tirar o

máximo proveito dos vários Cores a ideia inicial era fazer com que os sistemas fizessem o seu processamento em paralelo. Mas para tal, apenas os sistemas que não partilhassem o interesse pelos mesmos tipos de componentes (ou cujo acesso seria apenas de leitura) é que poderiam correr em paralelo. E por falta de tempo não se conseguiu implementar esta funcionalidade.

Em geral, os objetivos de BitEngine foram todos cumpridos. O trabalho futuro do projeto requeria trabalhar apenas nos inconvenientes e nos aspetos a melhorar.

Este foi um dos projetos mais complexos em que trabalhei devido ao facto de ser necessário de fazer um estudo à memória para perceber como é que é possível reduzir a quantidade de acesso à mesma reduzindo assim o tempo de espera do CPU pelos dados a processar. Para além disso, a implementação de ECS também não foi fácil. A primeira versão da implementação tinha uma performance pior em relação a OOP o que numa fase inicial me levou a começar a duvidar da implementação.

Bibliografia

Acton, M. (10 de 7 de 2021). *Data-Oriented Design and C++*. Obtido de YouTube: https://www.youtube.com/watch?v=rX0ItVEVjHc

Assassinscreed Fandom. (10 de 7 de 1021). *Anvil (game engine)*. Obtido de https://assassinscreed.fandom.com: https://assassinscreed.fandom.com/wiki/Anvil_(game_engine)

Epic Games. (10 de 7 de 2021). *The world's most open and advanced real-time 3D creation tool*. Obtido de Unreal Engine: https://www.unrealengine.com/en-US/

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

GDC. (10 de 7 de 2021). *Overwatch Gameplay Architecture and Netcode*. Obtido de YouTube: https://www.youtube.com/watch?v=W3aieHjyNvw

Härkönen, T. (10 de 7 de 2021). ADVANTAGES AND IMPLEMENTATION. *Faculty of Information Technology and Communication Sciences*.

Lazy Foo' Productions. (10 de 7 de 2021). *Beginning Game Programming v2.0*. Obtido de Lazy Foo: https://lazyfoo.net/tutorials/SDL/

MORLAN, A. (10 de 7 de 2021). *A SIMPLE ENTITY COMPONENT SYSTEM (ECS) [C++]*. Obtido de AUSTIN MORLAN: https://austinmorlan.com/posts/entity_component_system/#introduction

PC Diga. (28 de Abril de 2021). *Crucial Ballistix RGB 64GB (2x32GB) DDR4-3200MHz CL16*. Obtido de PC Diga: https://www.pcdiga.com/memoria-ram-crucial-ballistix-rgb-64gb-2x32gb-ddr4-3200mhz-cl16-branca-bl2k32g32c16u4wl?gclid=CjwKCAjwj6SEBhAOEiwAvFRuKHocI4weY6ayc7Dvw-TRbLaTlm_aVGlnLuEkw-aoxun8FgcxF6BjrhoCHMwQAvD_BwE

Sinicki, A. (10 de 7 de 2021). *What is Unity? Everything you need to know*. Obtido de Android Authority: https://www.androidauthority.com/what-is-unity-1131558/

Wikipedia. (10 de 7 de 2021). *Entity component system*. Obtido de Wikipedia: https://en.wikipedia.org/wiki/Entity_component_system

Anexo 1 Repositório GitHub

Link para o repositório de GitHub: https://github.com/vadimsZinatulins/BitEngine-Project