Atividade Orientada de Ensino: continuação da análise dos ganhos de performance na utilização do padrão Entity Component Systems contra implementações orientadas a objetos

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

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

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

Abstract. *This paper describes the development and results of an experiment to measure the performance gain on the utilization of Entity Component Systems for real-time simulations that need to process high ammounts of data compared to traditionally object-oriented implementations.*

Resumo. *Este trabalho descreve o desenvolvimento e os resultados de um experimento para medir o ganho de desempenho na utilização de Entity Component Systems para simulações em tempo real que necessitam processar altíssimas quantidades de dados comparado a implementações tradicionalmente orientadas a objetos.*

1. Introdução

"Entity Component Systems" (ECS) é um padrão de software que tem como objetivo fornecer aos desenvolvedores de jogos digitais maior flexibilidade arquitetural e ganho de desempenho por padrão, mesmo contra implementações orientadas a objetos que são baseadas em componentes [Nystrom 2011]. A premissa do ECS é garantir que os dados estejam sempre arranjados de forma a garantir eficientes acessos à memória, e diminuir o número de erros de *cache*.

Para entender melhor os possíveis ganhos de desempenho a partir da utilização do ECS, foram construídas duas simulações bi-dimensionais em C++ que possuem regras de mundo idênticas: uma orientada a objetos com componentes e outra baseada em ECS. O objetivo é realizar medições em números crescentes de entidades e compreender o comportamento computacional de cada uma das simulações.

2. A regras de mundo da aplicação

A aplicação desenvolvida é uma simulação de partículas que se comportam de formas diferentes de acordo com os componentes que as definem. Nesta simulação, existem dois tipos de entidades: luzes e caçadores de luz. Os caçadores de luz tem como objetivo se movimentar em direção a alguma fonte de luz. Contudo, a forma da movimentação deles é definida por algum componente de movimento, que podem ser **movimentação em ondas** ou **movimentação normal**. As luzes são objetos estáticos e que possuem a informação de intensidade de sua própria luz.

A **movimentação em ondas** é definida a partir dos seguintes passos: é calculado o vetor normalizado de direção até a luz inicial:

$$\vec{direcao} = \frac{posicaoLuz - posicaoParticula}{||posicaoLuz - posicaoParticula||} \tag{1}$$

Então é definido um vetor para movimentação perpendicular:

$$\vec{perpendicular} = (-direcao.y, direcao.x) \tag{2}$$

Para então fazer o cálculo da nova posição a cada passo de simulação:

$$\begin{aligned} x = (&tempo \cdot direcao.x + amplitude \cdot \\ &sin(frequencia \cdot tempo) \cdot perpendicular.x) \cdot delta \cdot 2 \end{aligned} \tag{3}$$

$$\begin{aligned} y = (&tempo \cdot direcao.y + amplitude \cdot \\ &cos(frequencia \cdot tempo) \cdot perpendicular.y) \cdot delta \cdot 2 \end{aligned} \tag{4}$$

Onde:

- $tempo$ é o valor em **segundos** desde o início da aplicação;
- $amplitude$ é um valor aleatório no intervalo $[100, 200]$ que descreve a amplitude de movimentação da partícula em torno da luz;
- $frequencia$ é um valor $\frac{1,0}{[20,60]}$;
- $delta$ é o tempo em **segundos** da duração total do último passo de simulação do mundo, é utilizado para garantir comportamento consistente sem depender da velocidade que a aplicação está rodando.

A **movimentação normal** é definida da seguinte forma: é determinado o vetor normalizado de direção, assim como na Equação 1, então o cálculo da nova posição a cada passo de simulação e definido por:

$$\begin{aligned} x &= delta \cdot direction.x \\ y &= delta \cdot direction.y \end{aligned} \tag{5}$$

Além dos cálculos de movimento, a cada cinco segundos de simulação, todas as partículas atualizam as luzes que estão seguindo a partir do seguinte algoritmo:

Algorithm 1 Descobrimento da fonte de luz mais próxima por uma partícula

```
distancia ← inf
for i ← 0 to quantLuzes − 1 do
    novaDistancia ← vetorDistancia(posicao, posicaoLuz)
    if novaDistancia < distancia then
        distancia ← novaDistancia
        luzIndice ← i
    end if
end for
```

2.1. Implementação da simulação ECS

Para desenvolvimento da aplicação ECS, foi utilizada a biblioteca de código aberto EnTT [Caini 2024] que implementa funcionalidades de ECS baseados em conjuntos esparsos [Briggs and Torczon 1993]. Nele foram definidos seis componentes, que definem a propriedade visual, posição no mundo, entre outras propriedades possíveis para uma partícula. A Figura 1 apresenta os dados de cada um dos componentes utilizados na simulação ECS. Note como o *NormalMovementComponent* funciona apenas como um rótulo, para identificação dos sistemas.

```
struct RenderableComponent
{
  rl::Rectangle rect;
  rl::Color color;
};

struct PositionComponent
{
  rl::Vector2 pos;
};

struct LightSeekerComponent
{
  uint8_t lightIndex;
};

struct LightComponent
{
  float intensity;
};

struct WaveMovementComponent
{
  float amplitude;
  float frequency;
  float speed;
};

struct NormalMovementComponent {
};
```

Figura 1. Componentes definidos para a execução da simulação ECS

Na execução da simulação, foram implementados três sistemas simples: *WaveLightSeekerSystem*, que simula as entidades que se movem em padrões ondulares e elipsais, e as renderiza; *NormalLightSeekerSystem*, que simula as entidades que movimento direto e as renderiza; e *LightSystem*, que é responsável apenas por renderizar as luzes

estáticas. A Figura 2 apresenta a implementação das partículas de movimento ondular, desde a captura de todas as entidades a partir de uma *view*, até a iteração por todos os componentes.

```
static void WaveLightSeekerSystem(double time, bool shouldTick)
{
  auto waveView = world.view<PositionComponent, LightSeekerComponent, WaveMovementComponent>();
  for(const entt::entity e : waveView) {
    PositionComponent& position     = world.get<PositionComponent>(e);
    LightSeekerComponent& light     = world.get<LightSeekerComponent>(e);
    WaveMovementComponent& movement = world.get<WaveMovementComponent>(e);

    rl::Vector2 lightTarget = lights[light.lightIndex].pos;

    rl::Vector2 normalizedDirection = rl::Vector2Normalize(rl::Vector2Subtract(lightTarget, position.pos));
    rl::Vector2 perpendicular = { -normalizedDirection.y, normalizedDirection.x };
    position.pos.x += (time * normalizedDirection.x + movement.amplitude * sin(movement.frequency * time) * perpendicular.x) * rl::GetFrameTime();
    position.pos.y += (time * normalizedDirection.y + movement.amplitude * cos(movement.frequency * time) * perpendicular.y) * rl::GetFrameTime();
    rl::DrawRectangle((int)position.pos.x, (int)position.pos.y, size, size, rl::Color{0, 0, 200, 100});

    if(shouldTick) {
      int newLight = -1;
      float pastDistance = FLT_MAX;
      for(uint8_t i = 0; i < 4; ++i) {
        float newDistance = rl::Vector2Distance(position.pos, lights[i].pos);
        if(newDistance < pastDistance) {
          pastDistance = newDistance;
          light.lightIndex = i;
        }
      }
    }
  }
}
```

Figura 2. Sistema de simulação das partículas de movimentação ondular.

2.2. Implementação da simulação orientada a objetos

Para a implementação da simulação orientada a objetos, não foi utilizada nenhuma biblioteca externa para arranjo dos objetos de jogos e componentes, foram empregados apenas *containers* padrões da linguagem C++.

A fim de seguir os padrões arquiteturais comumente vistos na indústria (e também a flexibilidade do ECS), a classe que define os objetos de jogo é apenas um "desenho", isto é, nela não é determinado nenhum dado ou comportamento acerca das entidades a serem simuladas, apenas existe um *container* comum para armazenamento dos componentes que podem ser adicionados e removidos em tempo de execução, e assim, os objetos são simulados conforme os dados que os definem. Neste caso, existe uma classe *Entity*, que possui um *container* de ponteiros para *IComponent*, a fim de utilizar o polimorfismo oferecido pelo C++ [Committee]. A Figura 3 demonstra como os objetos de jogo são inicializados.

A classe *IComponent* é uma classe abstrata que deve ser sobrescrita pelos componentes a serem criados pelo desenvolvedor. A Figura 4 apresenta a definição da base e da classe filha *WaveMovementComponent*, que é utilizada nas partículas que possuem movimento ondular. Desta forma, para criar uma entidade caçadora de luz que possui movimento ondular, basta instanciar uma entidade vazia e adicionar a ela os componentes *LightSeekerComponent* e *WaveMovementComponent*, e caso haja a necessidade de convertê-la em uma partícula de movimentação normal, basta remover o *WaveMovementComponent* e adicionar um *NormalMovementComponent*, sem a necessidade de remover o objeto e instanciar um novo.

```cpp
std::size_t entityCounter = 0;
printf("Creating %d Light Seekers with WaveMovement\n", m_EntityNum/2);
for(std::size_t i = 0; i < m_EntityNum/2; ++i)
{
  // Creating Light Seekers with Wave Movement
  float x = (float)(positionDistribution(gen));
  float y = (float)(positionDistribution(gen));
  Entity e = CreateEntity(x, y);
  e.AddComponent<LightSeekerComponent>(0);
  e.AddComponent<WaveMovementComponent>(
      static_cast<float>(amplitudeDistribution(gen)),
      static_cast<float>(1.0f / frequencyDistribution(gen)),
      static_cast<float>(speedDistribution(gen) / 20.0f)
  );
  m_Entities.push_back(e);
  entityCounter++;
}
```

Figura 3. Inicialização de um objeto de jogo partícula de movimentação ondular na simulação orientada a objetos.

```cpp
#pragma once

class IComponent
{
public:
  virtual void Update(float delta) = 0;
  virtual const char* GetName() = 0;
private:
};
```

```cpp
#pragma once

#include "IComponent.h"
#include "PositionComponent.h"

class WaveMovementComponent : public IComponent
{
public:
  WaveMovementComponent(float amplitude, float frequency, float speed);

  void Update(float delta) override;
  const char* GetName() override { return "WaveMovementComponent"; }

  float m_Amplitude, m_Frequency, m_Speed;
};
```

Figura 4. A definição da classe base *IComponent* e a classe *WaveMovement-Component*, que a sobrescreve e define o movimento ondular de partículas.

3. Resultados experimentais

Os testes foram realizados da seguinte forma: ambas simulações orientadas a objetos e ECS são executadas separadamente a partir de trinta mil entidades, com incremento de trinta mil entidades a cada execução. Para medir a diferença de desempenho de ambas as simulações, foi utilizado o software *CapFrameX*, que permite medir a média de *frametime* (tempo de duração de um quadro, ou a iteração do laço de simulação) durante a execução de um programa por tempo determinado, para os testes feitos foi utilizado duração de sessenta segundos.

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

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

Ademais, toda renderização da parte gráfica foi feita pela biblioteca de código aberta *Raylib* [Santamaria 2024], que permite fácil renderização em várias plataformas e sistemas operacionais.

Os resultados podem ser vistos na Tabela 1 e na Figura 5, que demonstra a diferença de *frametimes* entre cada simulação. É importante frisar que para os padrões da indústria, $33{,}3$ milissegundos é o ideal para jogos de alta fidelidade gráfica e de cálculos intensivos, portanto, ultrapassar esses valores não é interessante para produtos a serem comercializados. Ao analisar a Tabela 1 nota-se que os testes da versão orientada a objetos já são iniciados com valores aproximadamente quatro vezes maiores do que a ECS, e apresentam crescimento maior a cada etapa de testes, finalizando com o *frametime* de $175{,}9$ milissegundos para 330.000 entidades, o que é inaceitável para qualquer jogo eletrônico. Já a versão ECS inicia os testes com $2{,}7milissegundos$ e finaliza com $29{,}6$, atendendo aos requisitos ditos anteriormente para jogos comerciais de $33{,}3$ milissegundos. Ademais, ao observar a Figura 5, é possível perceber que ambas implementações apresentam comportamento linear quanto ao *frametime* em quantidades crescentes de entidades, contudo, a versão orientada a objetos apresenta inclinação maior quando comparado à versão ECS, dando a entender que o custo de operar muitos dados é constantemente maior quando os dados não são organizados com padrões eficientes às memórias (principalmente as memórias cache) em mente.

4. Conclusão

A conclusão notada ao observar os dados apontados pelos experimentos é de que ambas simulações apresentam *frametimes* com crescimento linear ao aumentar o número de entidades, entretanto, a aplicação feita com a biblioteca EnTT [Caini 2024] oferece bom desempenho em situações onde é necessário processar altíssimas quantidades de dados. A versão orientada a objetos, por outro lado, apresenta crescimento linear consideravelmente alto e com métricas não desejáveis para jogos eletrônicos comerciais, portanto, nota-se que o ECS é uma solução válida para situações onde a performance computacional é necessária juntamente com a flexibilidade arquitetural provida pelo modelo de entidades descritas por componentes.

Entidades	*Frametime* OOP	*Frametime* ECS
30.000	10,8 *ms*	2,7 *ms*
60.000	25 *ms*	5,4 *ms*
90.000	42,2 *ms*	8,1 *ms*
120.000	59,8 *ms*	10,8 *ms*
150.000	76,3 *ms*	13,4 *ms*
180.000	92,8 *ms*	16,1 *ms*
210.000	109,1 *ms*	18,8 *ms*
240.000	125,3 *ms*	21,4 *ms*
270.000	143 *ms*	24,1 *ms*
300.000	160 *ms*	26,7 *ms*
330.000	175,9 *ms*	29,6 *ms*

Tabela 1. Medidas de *frametime* relativas às quantidades incrementais de entidades em ambas simulações (OO e ECS).

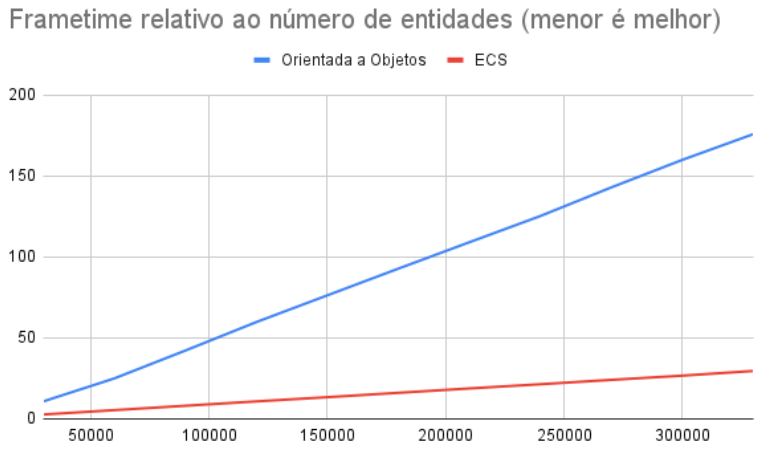

Figura 5. Diferenças de *frametime* utilizando orientação a objetos e *ECS*.

Referências

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

Caini, M. (2024). Entt. `https://github.com/skypjack/entt`. [online: acesso em 17-Novembro-2024].

Committee, C. C++ polymorphism. `https://cplusplus.com/doc/tutorial/polymorphism/`. [online: acesso em 17-Novembro-2024].

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

Santamaria, R. (2024). Raylib. `https://www.raylib.com/`. [online: acesso em 17-Novembro-2024].