Otimização de Desempenho no Desenvolvimento de Jogos

Thiago Luiz Alves Targino[1], **Nahri Moreano**[1]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul, Brasil

{thiago.targino, nahri.moreano}@ufms.br

Abstract. *The advance of the digital games industry, for entertainment and serious applications, resulted in a high demand for computing resources. Optimization techniques can be applied to enable their execution while meeting performance and hardware cost requirements. This article describes and exemplifies several optimizations that aim to improve the performance of digital games. A set of experiments is presented, where some optimizations are implemented and the performance results obtained are analyzed.*

Resumo. *O avanço da indústria de jogos digitais, para entretenimento e aplicações sérias, resultou em uma alta demanda de recursos computacionais. Técnicas de otimização podem ser aplicadas para possibilitar a execução dos mesmos atendendo a requisitos de desempenho e de custo do hardware. Este artigo descreve e exemplifica diversas otimizações que visam melhorar o desempenho de jogos digitais. Um conjunto de experimentos é apresentado, onde algumas otimizações são implementadas e os resultados de desempenho obtidos analisados.*

1. Introdução

A indústria de jogos digitais evoluiu ao longo dos anos do entretenimento para aplicações de jogos por organizações civis, comerciais e militares. Houve um aumento expressivo no desenvolvimento de jogos sérios para uma ampla variedade de propósitos, como educação, treinamento, desenvolvimento de habilidades e melhora da saúde física e psicológica [National Research Council 2010]. Mundialmente, a indústria de jogos tem se destacado dentre as indústrias criativas e culturais em termos financeiros e de perspectiva de crescimento para os próximos anos. Os jogos já ultrapassam, em faturamento, as indústrias de música e cinema somadas, e representam um importante mercado de trabalho para os profissionais da área [Mello et al. 2015, Amélio 2018, CODEMGE 2020]. Entretanto, o aumento da complexidade dos jogos, devido ao uso de modelos computacionais e simulações que produzam resultados mais precisos e realísticos, leva a uma maior demanda de recursos computacionais.

Este artigo descreve e avalia técnicas de otimização que podem ser aplicadas no desenvolvimento de jogos, com o objetivo de melhorar o desempenho dos mesmos. São apresentados exemplos de uso das técnicas em jogos reais. Pretende-se assim propiciar aos desenvolvedores de jogos uma compreensão dessas técnicas, de qual aspecto da execução é otimizado por elas e em que situações elas podem ser aplicadas.

O texto está organizado em cinco seções. A Seção 2 descreve diversas técnicas de otimização, enquanto a Seção 3 apresenta uma avaliação experimental de desempenho realizada com algumas das otimizações e analisa os resultados obtidos. Na Seção 4 são descritos outros trabalhos que tratam da aplicação de otimizações no desenvolvimento de jogos. Por fim, a Seção 5 conclui a discussão e apresenta possíveis trabalhos futuros.

2. Otimizações

As técnicas de otimização descritas abordam situações comuns em jogos digitais e visam otimizar diferentes aspectos da execução, como processamento na CPU, uso da memória, etc. O objetivo final é, quase sempre, melhorar o desempenho do jogo, reduzindo seu tempo de execução ou de resposta.

Object Pooling A técnica de *object pooling* consiste em utilizar um conjunto (*pool*) de objetos já alocados e inicializados ao invés de constantemente criar/alocar e destruir/liberar esses objetos individualmente durante a execução do jogo. Assim, quando um novo objeto é necessário, ele é obtido no conjunto, operado pelo programa e, ao final de sua utilização, desativado e colocado de volta no conjunto para ser utilizado novamente [Nystrom 2014, Unity Technologies 2019]. Essa técnica tem como objetivos melhorar o desempenho, eliminando o custo de alocar e liberar espaço na memória para os objetos durante a execução do programa, e também reduzir a fragmentação de memória. A Figura 1 exemplifica a criação de um conjunto de objetos, realizada apenas na inicialização do jogo. Assim, o custo do método *Instantiate* (para instanciar e renderizar o objeto na cena do jogo) incorre apenas no começo e não durante a execução do jogo.

```
public void CreateNewObjectsPool()
{
    poolDictionary = new Dictionary<string, Queue<GameObject>>();
    foreach (Pool pool in pools)
    {
        Queue<GameObject> objectPool = new Queue<GameObject>();

        for (int i = 0; i < pool.size; i++)
        {
            GameObject obj = Instantiate(pool.prefab);
            obj.SetActive(false);
            objectPool.Enqueue(obj);
            obj.transform.SetParent(this.transform);
        }
        poolDictionary.Add(pool.tag, objectPool);
    }
}
```

Figura 1. Implementação da técnica de *Object Pooling*

Uso de APIs O uso de APIs (*Application Programming Interface*) é uma prática comum no desenvolvimento de jogos digitais, uma vez que os motores gráficos utilizados possuem APIs que oferecem funcionalidades como detecção de colisão e outros métodos relacionados à física. Apesar da praticidade da utilização de APIs, elas podem causar perda de desempenho devido às suas generalizações. Por exemplo, no desenvolvimento do jogo *Adore* [Cadabra Games 2020] era necessário detectar se uma linha estava contida em um setor circular, como mostra a Figura 2. Uma API do motor gráfico adotado poderia ser utilizada para criar colisores na linha e no setor para realizar a detecção. Porém, essa implementação não possui um bom desempenho, pois acarretaria diversas chamadas para detectar objetos dentro da área de efeito dos objetos com colisores, além do tratamento de exceções para garantir que o único elemento pertinente para verificação de colisão no setor é a linha e vice-versa. Uma solução que não envolve APIs de física ou detecção de colisão é mostrada na Figura 3, onde algumas operações simples para obtenção de direção e verificação de ângulo são usadas para a detecção.

Figura 2. Problema ao capturar uma criatura no jogo *Adore*

```
float mappedResult = Mathf.Lerp(0, 360, _captureCircleInformation.externCircleImage.fillAmount);
Vector3 staffDirection = (_captureCircleInformation.transform.position -
    playerController.vfxStaff.transform.position).normalized;
staffDirection.y = 0;
redAxisDirection = _captureCircleInformation.externCircleImage.transform.right;
redAxisDirection = Quaternion.Euler(0, -(mappedResult / 2), 0) * redAxisDirection;

if (Vector3.Angle(staffDirection, -redAxisDirection) <= halfMappedResult)
{
```

Figura 3. Solução para captura sem usar APIs de física ou detecção de colisão

Operações entre Vetores As operações matemáticas entre vetores são consideravelmente mais custosas do que as operações entre dados escalares. Assim, caso existam várias operações em sequência envolvendo vetores e escalares, o desempenho pode ser melhorado realizando previamente as operações que não envolvam vetores [Kjems et al. 2016]. A Figura 4 exemplifica um cálculo envolvendo dois vetores e três variáveis do tipo *float*. A execução desse código realiza as seguintes operações: três multiplicações de vetor por *float* e uma soma de vetor com vetor. Cada multiplicação de vetor por um *float* consiste em multiplicar cada elemento do vetor pelo *float*, o que pode consumir um tempo de execução significativo. Uma solução otimizada encontra-se na Figura 5 e consiste em realizar primeiro as operações entre os *floats*, em seguida realizar uma única multiplicação de vetor pelo *float*, e por fim, a soma de vetor com vetor. Para isso, basta utilizar os parênteses para sinalizar a ordem de precedência das operações, removendo as operações desnecessárias de multiplicação entre vetor e *float* e melhorando o desempenho.

```
public void CalculatePlayerPosition(Transform playerTransform)
{
    float a = 5, b = 3, c = 1;
    Vector3 v = new Vector3(3, 3, 0);
    playerTransform.position += v * a * Mathf.Sin(b) * c;
}
```

Figura 4. Cálculo não otimizado entre vetores e escalares

```
public void CalculatePlayerPosition(Transform playerTransform)
{
    float a = 5, b = 3, c = 1;
    Vector3 v = new Vector3(3, 3, 0);
    playerTransform.position += v * (a * Mathf.Sin(b) * c);
}
```

Figura 5. Cálculo otimizado entre vetores e escalares

Dirty Flag A atualização de dados ocorre a todo momento durante a execução de um jogo, sejam estes relacionados a posição, rotação e escala de objetos ou até informações de algum personagem. Muitos dados derivados dessa atualização demandam cálculos custosos ou operações de sincronização como escrita ou leitura no disco rígido, entretanto não precisam ser recalculados imediatamente no momento em que ela acontece [Nystrom 2014]. A técnica de usar um *Dirty Flag* pode ser aplicada para evitar a atualização dos dados derivados (e as operações custosas envolvidas), a cada vez que um dado primário é modificado. Um *Dirty Flag* é um bit, que pode estar aceso ou não, associado aos dados primários dos objetos e usado para sinalizar que o dado primário foi alterado. Apenas no momento de utilização do dado derivado, é determinado se ele precisa ser atualizado, checando-se o *Dirty Flag* do dado primário correspondente. Assim, o dado derivado é atualizado apenas se o *Dirty Flag* indicar que o dado primário foi modificado, e essa atualização do dado derivado pode acontecer após o dado primário ter sido alterado várias vezes.

Estrutura Multidimensional vs. Vetor de Vetores Ao armazenar dados multidimensionais, estruturas multidimensionais costumam ser usadas, por apresentarem uma sintaxe mais intuitiva e serem mais facilmente manipuladas. Entretanto, acessar uma estrutura multidimensional tem, em geral, um desempenho pior, do que acessar um vetor de vetores. A Figura 6 mostra métodos para

atribuir um valor a um elemento de um vetor de vetores e de uma estrutura multidimensional. Utilizando um *disassembler* com esse código, é possível obter o código em linguagem intermediária gerado para cada um dos métodos, mostrado na Figura 7. Para o vetor de vetores, são usadas apenas instruções simples de escrita e leitura, enquanto que para a estrutura multidimensional é realizada a chamada de um método, o que pode ser mais custoso [Unity Technologies 2021].

```
static void SetElementAt(int[][] array, int i, int j, int value)
{
    array[i][j] = value;
}

static void SetElementAt(int[,] array, int i, int j, int value)
{
    array[i, j] = value;
}
```

Figura 6. Atribuição de elemento em vetor de vetores e em estrutura multidimensional

```
.method private hidebysig static void  SetElementAt(int32[][] 'array', int32 i, int32 j,
                                                    int32 'value') cil managed
{
  // Code size       7 (0x7)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  ldelem.ref
  IL_0003:  ldarg.2
  IL_0004:  ldarg.3
  IL_0005:  stelem.i4
  IL_0006:  ret
} // end of method Program::SetElementAt

.method private hidebysig static void  SetElementAt(int32[0...,0...] 'array', int32 i, int32 j,
                                                    int32 'value') cil managed
{
  // Code size       10 (0xa)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  ldarg.3
  IL_0004:  call       instance void int32[0...,0...]::Set(int32, int32, int32)
  IL_0009:  ret
} // end of method Program::SetElementAt
```

Figura 7. Código em linguagem intermediária gerado para métodos da Figura 6

Struct* de Vetores vs. Vetores de *Struct A utilização de um vetor de *structs* para armazenar propriedades de uma coleção de elementos é uma prática comum no desenvolvimento de jogos. Se todas as propriedades dos elementos são acessadas em conjunto, essa organização é adequada. Entretanto, ela pode trazer problemas de desempenho se o padrão de acessos é outro. Por exemplo, se uma função modifica apenas uma propriedade de todos os elementos, as demais propriedades também são trazidas para cache mas não são utilizadas. Tal alocação reduz a localidade espacial no acesso aos dados e pode levar a ocorrência de mais falhas na cache [Savas 2017]. Uma alternativa é utilizar um *struct* de vetores. Ao acessar o vetor da propriedade desejada de todos os elementos, apenas essa propriedade é trazida para cache. Ao tirar proveito da localidade espacial no acesso a esses dados, reduz-se as falhas na cache. Essas duas organizações podem ser vistas na Figura 8.

Uso de APIs que não Alocam Espaço para Física Algumas funções oferecidas pelas APIs dos motores gráficos alocam espaço na memória para realizar tarefas relacionadas à física. Por exemplo, a função *Physics.SphereCast* disponibilizada pela Unity detecta todos os colisores presentes

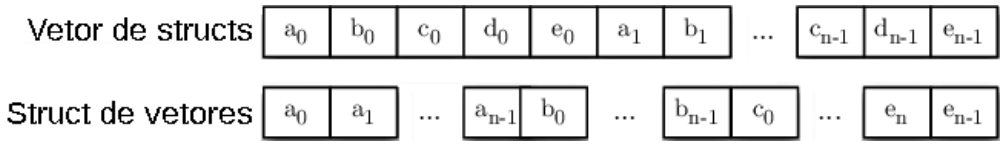


Figura 8. *Layout* na memória do armazenamento de propriedades de n elementos na forma de vetor de *structs* e de *struct* de vetores

em uma esfera indicada pelos parâmetros passados. Os colisores detectados são armazenados em um vetor alocado na função e retornado no final da mesma. Assim, toda chamada realiza uma nova alocação, o que pode gerar fragmentação na memória [Unity Technologies 2021]. Uma alternativa é utilizar *non-allocating APIs*. Por exemplo, a função *Physics.SphereCastNonAlloc* evita essa alocação e um de seus parâmetros é o vetor onde serão armazenados os resultados.

Reorganização dos Dados para Redução de Falhas na Cache No desenvolvimento de jogos, é comum que um conjunto de dados seja utilizado em vários *frames* seguidos. Assim, é importante que dados acessados em conjunto e com frequência sejam identificados pelo desenvolvedor para serem alocados em conjunto na cache e evitar falhas na mesma. Por exemplo, ao portar o jogo *The Witcher 3* para o *Nintendo Switch*, os desenvolvedores identificaram situações em que a reorganização dos dados proporcionou ganho de desempenho ao reduzir falhas na cache. A redução da quantidade de membros (e em consequência o tamanho da estrutura) de uma classe utilizada com frequência foi uma das estratégias adotadas, mostrada na Figura 9. A classe possuía 32 bytes e havia cerca de 1,3 milhão de instâncias da classe na execução do jogo. O membro *m_class* podia ser acessado a partir de *m_object*, portanto foi removido; o membro *m_weakRefCount* era utilizado em apenas uma chamada, onde foi substituído por *m_strongRefCount*, e também foi removido. Assim o tamanho da classe diminuiu para 16 bytes [Lebedev 2020].

```
class ReferencableInternalHandle {
       Red::Threads::CAtomic< Int32 > m_weakRefCount;
       Red::Threads::CAtomic< Int32 > m_strongRefCount;
       volatile Uint32                m_flags;
       IReferencable*                 m_object;
       const CClass*                  m_class;
};
sizeof(ReferencableInternalHandle) == 32
```

```
class ReferencableInternalHandle {
       Red::Threads::CAtomic< Int32 > m_strongRefCount;
       volatile Uint32                m_flags;
       IReferencable*                 m_object;
};


sizeof(ReferencableInternalHandle) == 16
```

Figura 9. Redução de tamanho de estrutura [Lebedev 2020]

Outra estratégia foi agrupar em uma mesma linha da cache os membros de uma classe acessados no *hot path* (sequência de instruções executadas com frequência). Utilizou-se a ferramenta *PadAnalyzer* para depurar o programa e identificar os dados acessados frequentemente. A Figura 10 mostra as mudanças no código para agrupar esses membros e garantir que sejam alocados na mesma linha da cache. O acesso aos membros *m_frameTracker* e *m_drawableFlags* estava causando falhas na cache e eles foram portanto agrupados com outros membros constantemente acessados [Lebedev 2020].

```
class IRenderProxyBase : public IRenderProxy, public IRenderPrefetchable {
protected:
    // padding 24, 16 byte alignment for Box
    Box                             m_boundingBox; // offset 32
    // cache line 1
    Matrix                          m_localToWorld; // offset 64
    // cache line 2
    CRenderSceneEx*                 m_scene; // offset 128
    Uint32                          m_entryID;
    Float                           m_autoHideDistance;
    Float                           m_autoHideDistanceSquared;
    GlobalVisID                     m_umbraProxyId;
    Red::Threads::CAtomic< Int32 >  m_registrationRefCount;
    SFrameTracker                   m_frameTracker;
```

```
class IRenderProxyBase : public IRenderProxy {
protected:
    // no padding
    SFrameracker                    m_frameTracker; // offset 16
    Red::Threads::CAtomic< Int32 >  m_registrationRefCount;
    Uint32                          m_entryID;
    Uint8                           m_drawableFlags;
    Uint8                           m_lightChannels;
    Uint8                           m_renderingPlane;
    Box                             m_boundingBox;
    // cache line 1
    Matrix                          m_localToWorld;
    // cache line 2
    CRenderSceneEx*                 m_scene;
```

Figura 10. Agrupamento de dados acessados com frequência [Lebedev 2020]

3. Avaliação Experimental de Desempenho

Um conjunto de experimentos foi realizado para avaliar algumas das otimizações descritas. Os resultados apresentados nesta seção foram obtidos utilizando um computador pessoal com um

processador AMD Ryzen 7 3800X com 16 *threads* e oito núcleos, cache L1 de 512KB, cache L2 de 4MB e cache L3 de 32MB, 32GB de memória RAM e o sistema operacional Windows 10 Pro Versão 20H2. O motor gráfico Unity e a linguagem C# foram utilizados, assim como o *profiler* fornecido pelo Unity [Unity Technologies 2021].

Object Pooling No desenvolvimento do jogo *Adore*, uma situação recorrente é a ativação de efeitos visuais para os ataques de criaturas. A situação analisada é a ativação de 20 efeitos visuais na tela no mesmo *frame*, supondo que existem quatro criaturas na tela, cada uma com um ataque que atinge cinco inimigos, resultando na necessidade da ativação de vinte efeitos visuais. A Figura 11 mostra os resultados produzidos pelo *profiler* do Unity, com a criação das instâncias dos efeitos visuais para um *frame*, sem a utilização da técnica de *Object Pooling*. A linha destacada mostra o método *Instantiate* que cria as instâncias. A coluna *Time ms* mostra o tempo de execução (em milissegundos) considerando todas as invocações do método, que é 6,85ms, e a coluna *Total* indica a porcentagem que isso representa em relação ao tempo total gasto naquele *frame*), que é 12,6%. Vale ressaltar que, além do custo de instanciação dos objetos, há também o custo de destruição dos mesmos. Na Figura 12, a técnica de *Object Pooling* foi utilizada e as instâncias dos efeitos visuais foram criadas na inicialização do jogo. Assim, é necessário apenas retirar os objetos de um *pool*, ativando-os e, após o uso, desativá-los. Os métodos de ativação e desativação consumiram 0,16ms e 0,13ms, respectivamente, correspondendo juntos a 0,4% do tempo total gasto no *frame*. Esses resultados correspondem à execução de apenas um *frame* e essa situação ocorre diversas vezes durante a execução do jogo, portanto a técnica de *Object Pooling* permitiu um ganho de desempenho considerável.

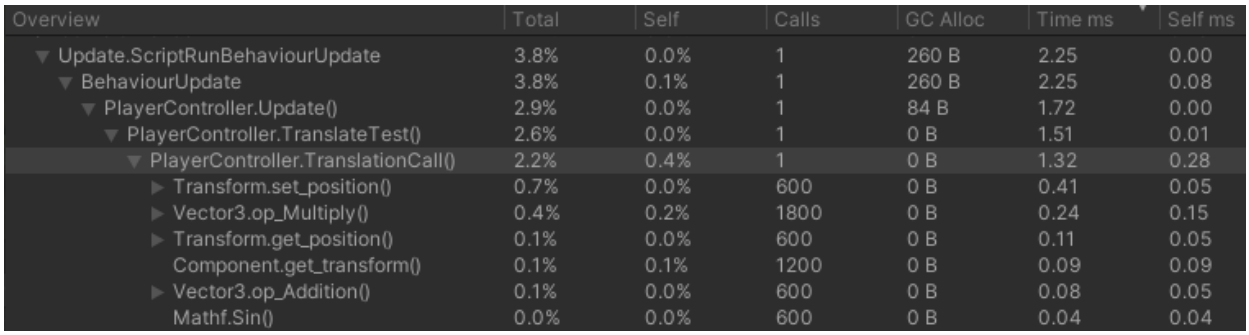
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
EditorLoop	61.6%	61.6%	2	0 B	33.36	33.36
▼ PlayerLoop	38.3%	0.3%	2	16.4 KB	20.76	0.17
▼ Update.ScriptRunBehaviourUpdate	14.1%	0.0%	1	1.1 KB	7.66	0.00
▼ BehaviourUpdate	14.1%	0.1%	1	1.1 KB	7.66	0.08
▼ PlayerController.Update() [Invoke]	13.5%	0.3%	1	0.9 KB	7.34	0.18
▶ Instantiate	12.6%	0.0%	20	0 B	6.85	0.03
▶ Mono.JIT	0.5%	0.5%	4	112 B	0.30	0.29

Figura 11. Instanciação sem a utilização de *Object Pooling*

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
EditorLoop	70.3%	70.3%	2	0 B	38.15	38.15
▼ PlayerLoop	29.5%	0.3%	2	138.9 KB	16.04	0.17
▶ PostLateUpdate.PlayerUpdateCanvases	15.9%	0.0%	1	126.8 KB	8.67	0.00
▶ Camera.Render	6.0%	0.3%	7	0 B	3.30	0.18
▼ Update.ScriptRunBehaviourUpdate	2.3%	0.0%	1	2.1 KB	1.24	0.00
▼ BehaviourUpdate	2.3%	0.1%	1	2.1 KB	1.24	0.06
▼ PlayerController.Update() [Invoke]	1.7%	0.4%	1	2.0 KB	0.92	0.22
▶ Mono.JIT	0.6%	0.6%	4	104 B	0.36	0.36
▶ GameObject.Activate	0.2%	0.0%	20	0 B	0.16	0.00
▶ ObjectPooler.DeactivatePoolObjectCoroutine()	0.2%	0.0%	20	0 B	0.13	0.02

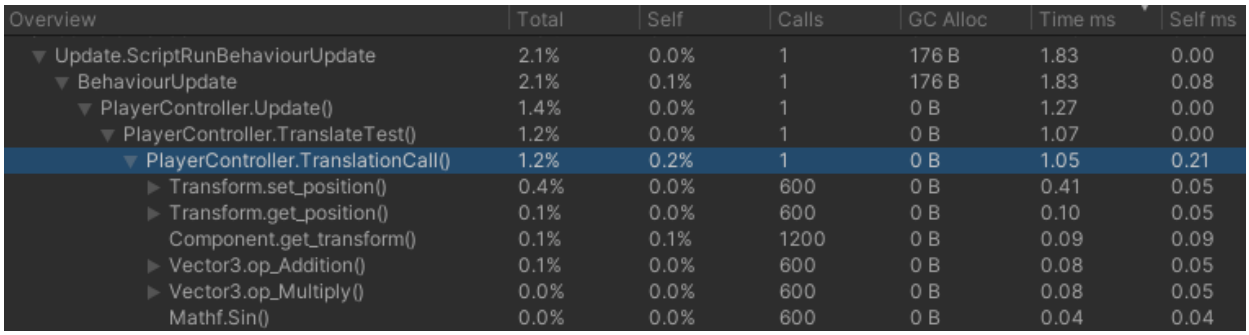
Figura 12. Instanciação utilizando a técnica de *Object Pooling*

Operações entre Vetores Supondo um jogo onde a posição de 20 elementos (personagem principal, projéteis, criaturas aliadas e inimigas, entre outros) é atualizada em um *frame*, com uma operação de 30 *frames* por segundo, são ao todo 600 chamadas da função de atualização. A posição dos elementos foi representada por vetores de três posições e atualizada com as funções apresentadas no exemplo da Seção 2. A Figura 13 mostra os resultados da atualização das posições sem a otimização das operações entre vetores. São gastos 1,32ms na execução das funções relacionadas à atualização e a operação de multiplicação entre vetores é chamada 1.800 vezes. A atualização das posições com a otimização das operações entre vetores produziu os resultados da Figura 14, onde são gastos 1,05ms na execução das funções relacionadas à atualização e a operação de multiplicação entre vetores é chamada 600 vezes.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
Update.ScriptRunBehaviourUpdate	3.8%	0.0%	1	260 B	2.25	0.00
BehaviourUpdate	3.8%	0.1%	1	260 B	2.25	0.08
PlayerController.Update()	2.9%	0.0%	1	84 B	1.72	0.00
PlayerController.TranslateTest()	2.6%	0.0%	1	0 B	1.51	0.01
PlayerController.TranslationCall()	2.2%	0.4%	1	0 B	1.32	0.28
Transform.set_position()	0.7%	0.0%	600	0 B	0.41	0.05
Vector3.op_Multiply()	0.4%	0.2%	1800	0 B	0.24	0.15
Transform.get_position()	0.1%	0.0%	600	0 B	0.11	0.05
Component.get_transform()	0.1%	0.1%	1200	0 B	0.09	0.09
Vector3.op_Addition()	0.1%	0.0%	600	0 B	0.08	0.05
Mathf.Sin()	0.0%	0.0%	600	0 B	0.04	0.04

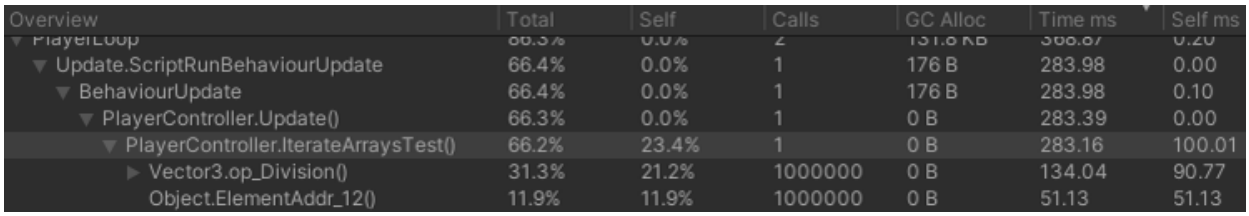
Figura 13. Operação entre vetores sem otimização



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
Update.ScriptRunBehaviourUpdate	2.1%	0.0%	1	176 B	1.83	0.00
BehaviourUpdate	2.1%	0.1%	1	176 B	1.83	0.08
PlayerController.Update()	1.4%	0.0%	1	0 B	1.27	0.00
PlayerController.TranslateTest()	1.2%	0.0%	1	0 B	1.07	0.00
PlayerController.TranslationCall()	1.2%	0.2%	1	0 B	1.05	0.21
Transform.set_position()	0.4%	0.0%	600	0 B	0.41	0.05
Transform.get_position()	0.1%	0.0%	600	0 B	0.10	0.05
Component.get_transform()	0.1%	0.1%	1200	0 B	0.09	0.09
Vector3.op_Addition()	0.1%	0.0%	600	0 B	0.08	0.05
Vector3.op_Multiply()	0.0%	0.0%	600	0 B	0.08	0.05
Mathf.Sin()	0.0%	0.0%	600	0 B	0.04	0.04

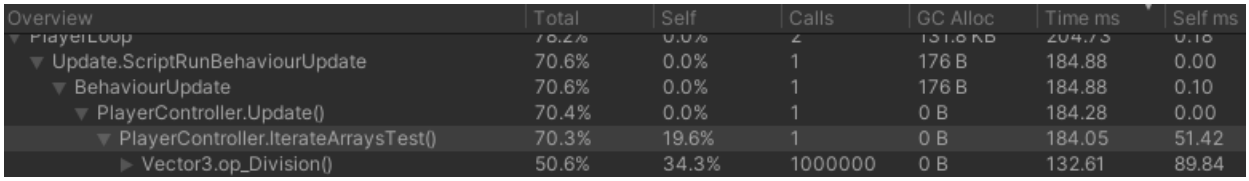
Figura 14. Operação entre vetores com otimização

Estrutura Multidimensional vs. Vetor de Vetores Para avaliar essa otimização, analisou-se a operação de iterar por uma estrutura de $100 \times 100 \times 100$ elementos e realizar uma divisão em cada elemento, simulando a posição, escala e rotação de um conjunto de *GameObjects*, por exemplo. Inicialmente, uma estrutura multidimensional foi usada e os resultados da Figura 15 obtidos, com o tempo de execução da iteração completa em 283,16ms. Em seguida, um vetor de vetores de vetores foi usado e os resultados da Figura 16 foram alcançados, com o tempo de execução em 184,05ms. A chamada da função adicional *ElementAddr* apenas para poder iterar entre os elementos da estrutura multidimensional é muito custosa. Portanto, apesar da simplicidade de sua utilização, esta é muito ineficiente. Esse resultado é consideravelmente impactante, dado que o processo de iteração por estruturas é extremamente comum.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	66.3%	0.0%	2	131.8 KB	368.87	0.20
Update.ScriptRunBehaviourUpdate	66.4%	0.0%	1	176 B	283.98	0.00
BehaviourUpdate	66.4%	0.0%	1	176 B	283.98	0.10
PlayerController.Update()	66.3%	0.0%	1	0 B	283.39	0.00
PlayerController.IterateArraysTest()	66.2%	23.4%	1	0 B	283.16	100.01
Vector3.op_Division()	31.3%	21.2%	1000000	0 B	134.04	90.77
Object.ElementAddr_12()	11.9%	11.9%	1000000	0 B	51.13	51.13

Figura 15. Iteração realizada em uma estrutura multidimensional



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	78.2%	0.0%	2	131.8 KB	204.73	0.18
Update.ScriptRunBehaviourUpdate	70.6%	0.0%	1	176 B	184.88	0.00
BehaviourUpdate	70.6%	0.0%	1	176 B	184.88	0.10
PlayerController.Update()	70.4%	0.0%	1	0 B	184.28	0.00
PlayerController.IterateArraysTest()	70.3%	19.6%	1	0 B	184.05	51.42
Vector3.op_Division()	50.6%	34.3%	1000000	0 B	132.61	89.84

Figura 16. Iteração realizada em um vetor de vetores de vetores

4. Trabalhos Relacionados

Esta seção apresenta o relato de otimizações realizadas por desenvolvedores em diferentes jogos, com o objetivo de melhorar o desempenho dos mesmos. Os desenvolvedores do jogo *INSIDE* mostraram diversas otimizações relacionadas a carregar cenas, gerenciamento de memória, etc [Kjems et al. 2016]. Entre elas, a otimização para evitar operações entre vetores proporcionou uma redução de $\sim$17% no tempo de execução, um resultado próximo do obtido na Seção 3. Os

desenvolvedores do Unity apresentaram algumas otimizações que podem ser realizadas para obter ganho de desempenho no uso do seu motor gráfico [Unity Technologies 2021]. Entre elas, a substituição de uma estrutura tridimensional de $100 \times 100 \times 100$ elementos por um vetor de vetores levou a uma redução no tempo de execução de 3470ms para 730ms, resultado esse próximo do obtido na Seção 3.

A diferença de desempenho ao utilizar um *struct* de vetores ao invés de um vetor de *structs* é analisada no desenvolvimento de um protótipo de um motor gráfico para jogos. A organização na forma de *struct* de vetores ocuparia um terço de linhas da cache quando comparada com o vetor de *structs* [Savas 2017]. Os desenvolvedores do jogo *The Witcher 3: Wild Hunt* realizaram diversas otimizações de CPU e de memória para portar o jogo para o *Nintendo Switch*, devido às limitações de recursos de hardware do *console*. A redução do tamanho de uma classe com $\sim$1,3 milhão de instâncias de 32B para 16B economizou $\sim$20MB de memória. Além disso, ao agrupar em uma mesma linha da cache os membros de uma classe acessados com frequência, reduziu-se as falhas na cache, o que resultou em um ganho de 1,5ms por *frame* [Lebedev 2020].

5. Conclusão

Este artigo descreveu uma série de otimizações que podem ser aplicadas no desenvolvimento de jogos digitais para melhorar seu desempenho, focando em diferentes aspectos como processamento da CPU, uso da memória, etc. Diversas das técnicas apresentadas são adotadas em jogos comerciais conhecidos. Uma avaliação experimental de desempenho de algumas otimizações foi realizada para demonstrar sua importância e eficácia. As otimizações demonstradas proporcionaram uma melhora considerável no desempenho.

O foco desse trabalho foi o estudo de otimizações para jogos que possam ser aplicadas em diversas plataformas. Outras plataformas, como GPUs e dispositivos móveis, possuem características particulares, o que traz diferentes desafios para otimizar o desempenho dos jogos nas mesmas. Assim, otimizações específicas para essas plataformas devem ser investigadas em trabalhos futuros.

Referências

Amélio, C. (2018). A Indústria e o Mercado de Jogos Digitais no Brasil. In *SBGames*, pages 1497–1506.

Cadabra Games (2020). Adore. `https://www.cadabragames.com`.

CODEMGE (2020). Série de Estudos Setoriais em Economia Criativa – Games. Technical report, Companhia de Desenvolvimento de Minas Gerias.

Kjems, K., Pedersen, E., and Madsen, S. (2016). Tools, Tricks and Technologies for Reaching Stutter Free 60 FPS in INSIDE. *Unite*, `https://www.youtube.com/watch?v=mQ2KTRn4BMI&t=1715s`. Acessado em junho/2022.

Lebedev, R. (2020). 'Witcher 3' on the Nintendo Switch: CPU & Memory Optimization. *Game Developers Conference*, `https://gdcvault.com/play/1026635/-Witcher-3-on-the`. Acessado em junho/2022.

Mello, G. et al. (2015). Como a indústria brasileira de jogos digitais pode passar de fase. *BNDES Setorial*, 42:337–382.

National Research Council (2010). *The Rise of Games and High-Performance Computing for Modeling and Simulation*. The National Academies Press.

Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.

Savas, N. (2017). Nomad Game Engine: Part 4.3 – AoS vs SoA. `https://medium.com/@savas/nomad-game-engine-part-4-3-aos-vs-soa-storage-5bec879aa38c`. Acessado em junho/2022.

Unity Technologies (2019). Introduction to Object Pooling. `https://learn.unity.com/tutorial/introduction-to-object-pooling`. Acessado em junho/2022.

Unity Technologies (2021). Unity User Manual 2021.3. `https://docs.unity3d.com`.