CENTRO PAULA SOUZA
FATEC OURINHOS
CURSO DE TECNOLOGIA EM JOGOS DIGITAIS

Caio Delafiori Gallo
Stefano Schippa Ambrosio

TÉCNICAS DE OTIMIZAÇÃO PARA JOGOS *MOBILE* ANDROID EM UNITY 3D

OURINHOS (SP)
2018

CAIO DELAFIORI GALLO
STEFANO SCHIPPA AMBROSIO

TÉCNICAS DE OTMIZAÇÃO PARA JOGOS *MOBILE* ANDROID EM UNITY 3D

Trabalho de graduação apresentado à Faculdade de Tecnologia de Ourinhos para conclusão do Curso de Tecnologia em Jogos Digitais.

Orientador: Prof. André Luís Orlandi Fávaro

OURINHOS (SP)
2018

AGRADECIMENTOS

Agradecemos primeiramente ao professor André Luís Orlandi Fávaro por aprovar nosso trabalho e por toda a orientação durante esta pesquisa de trabalho de graduação, e às nossas famílias pelo apoio durante o desenvolvimento da pesquisa e durante o curso.

RESUMO

Este trabalho teve como objetivo aplicar técnicas e conceitos de otimização de jogos *mobile* no desenvolvimento de uma aplicação para a plataforma Android, utilizando Unity3D como a *Game Engine* para o processo. No decorrer da pesquisa foram explicados aspectos da otimização que abrangem o jogo o qual foi desenvolvido. Aspectos estes como gerenciamento eficiente de recursos, técnicas de *scripting* para uso eficiente de memória e processamento e gerenciamento eficiente de memória RAM. Todos estes pontos foram extensamente abordados e aplicados ao jogo desenvolvido nesta pesquisa, para que houvesse um estudo de caso o mais específico possível. Ao final do desenvolvimento do jogo foram realizados testes de performance na aplicação, documentando e analisando o seu desempenho em três modelos diferentes de *smartphone*, cada um com uma configuração técnica específica e única em sua categoria de preço. Ao final da análise de desempenho, os testes mostraram que as técnicas e conceitos nesta pesquisa descritos e estudados e no jogo implementados fizeram com o que o desempenho final da aplicação em dois dos modelos atingisse os objetivos planejados desde o início, que foram execução fluída e taxas de *frames* constantes e consideradas ótimas, além de o modelo restante alcançar desempenhos que superaram as expectativas, com taxas de *frames* e níveis de execução excelentes para suas capacidades.

Palavras-chave: Otimização, Jogos Mobile, Unity 3D.

ABSTRACT

The goal of this research was to apply techniques and concepts of mobile games optimization, in the development of a game for Android, using the Unity3D game engine. Throughout this research it is explained aspects of the optimization that covers the game that was developed. These aspects are efficient resource management, scripting techniques for efficient memory usage, and efficient RAM management and processing. All these subjects were extensively approached and applied to the developed game in this study, for the sake of having the most specific case study possible. At the end of the development, were performed tests to analyze the application performance. These tests were realized in three different smartphone models, each one having its unique configuration to its price range. After analyzing the performance tests, it was clear that the techniques and concepts of mobile games optimization explained in this research, and implemented in the game, made two of the smartphones reach the research initial goal. This goal was to have a good execution of the game and constant optimal frame rates. In addition to one of the models reaching results that were over the expectations, having a excellent execution and frame rate for its hardware.

Keywords: Optimization, Mobile Games, Unity 3D.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

RAM: *Random Access Memory,* ou Memória de Acesso Aleatório

GPU: *Graphics Processing Unit*, ou Unidade de Processamento Gráfico

CPU: *Central Processing Unit*, ou Unidade Central de Processamento

API: Application Programming Interface, ou Interface de Programação de Aplicativos

UML: *Unified Modeling Language*

FPS: *Frames per Second*, ou *Frames* por Segundo

SUMÁRIO

1. INTRODUÇÃO

O mercado de jogos para Android vem crescendo nos últimos tempos. Segundo McDonald (2017), diretora da empresa holandesa NEWZOO de pesquisa de mercado sobre jogos, *E-sport*, e *mobile*, o mercado global de jogos espera gerar no ano de 2017 uma receita de $108.9 bilhões de dólares, 7.8% a mais que 2016. No valor total arrecadado pelo mercado de jogos global o segmento de jogos *mobile* é responsável por 42% desse valor, cerca de $46.1 bilhões de dólares anuais.

Com o crescimento do mercado, jogos cada vez mais complexos e sofisticados graficamente e mecanicamente vem surgindo. Porém, as capacidades técnicas de *hardware* dos aparelhos atuais não acompanham este crescimento da área de desenvolvimento de jogos *mobile*, e um estudo com o objetivo de melhor utilizar toda a capacidade de *hardware* e *software* que se tem em disponibilidade atualmente é necessário para um desenvolvimento eficiente e satisfatório.

A otimização de jogos *mobile* é um ponto do desenvolvimento de *software*s de entretenimento que deve ser compreendido, estudado e praticado pelos desenvolvedores e estudiosos da área. Um jogo independente do tipo de máquina e configuração para o qual ele está sendo desenvolvido precisa ser otimizado, diferente de um *software* comercial de gerenciamento, por exemplo, que pode rodar em diversas configurações sem tantos problemas, por não apresentar recursos gráficos tão avançados.

Grande parte dessa necessidade extra de otimização é a constante requisição de recursos gráficos e chamadas de desenho, ou "*Draw Calls*", ao processador, pelo fato da maioria dos jogos serem bastante custosos no recurso computacional gráfico.

Jogos, além de utilizarem-se de grandes quantidades de capacidade de processamento e memória RAM, também fazem uso de grande parte ou toda a capacidade disponível da placa gráfica presente no computador. Isso porque, além de precisarem das operações e cálculos lógicos presentes em praticamente todos os *software*s, eles precisam processar uma parte gráfica muito desenvolvida, com gráficos complexos e que estão cada vez mais realistas.

Renderizar e processar essas imagens e modelos que tem resoluções cada vez mais altas não são tarefas nem de longe simples ou pouco custosas em questão de recursos computacionais gráficos. Somam-se ainda esses pontos ao fato de os *smartphones*, apesar de estarem evoluindo constantemente seu *hardware* e *software*, serem mais limitados em questão de recursos computacionais disponíveis, comparado às configurações de computadores que se

encontram hoje em dia.

Problema da pesquisa

Como melhorar o desempenho de jogos desenvolvidos em Unity 3D para dispositivos móveis Android?

Objetivos

Objetivo Geral

Tem-se como objetivo geral deste trabalho desenvolver um jogo mobile em Unity 3D que utilize meios de otimização de desempenho gráfico e de programação, com o propósito de analisar as melhorias e ganhos de desempenho final obtido.

Objetivos específicos

- Analisar o ganho de desempenho de CPU e GPU obtidos através do uso de técnicas de programação e *design* eficientes para o desenvolvimento de jogos;
- Identificar possíveis pontos problema em questão de otimização e desempenho final que podem ser encontrados durante a programação e desenvolvimento da arte do jogo;
- Sugerir os melhores meios para resolver esses possíveis pontos problemas encontrados durante a pesquisa.
- Discorrer sobre técnicas de otimização usadas em jogo, explicar seu funcionamento e porque elas melhoram o desempenho final de um jogo.

Justificativa

O uso de *smartphones* vem crescendo a cada dia mais no mundo todo, seja pela diminuição do preço dos aparelhos, o que os torna mais acessíveis a diversos tipos de público,

ou pelo simples fato de que o *smartphone* está se tornando peça inseparável do dia a dia do ser humano moderno.

Acompanhando o mercado dos *smartphones*, o mercado de jogos para dispositivos móveis no Brasil também vem crescendo. O mercado brasileiro de jogos para Android teve um crescimento de 780% no período de 2009 a 2014, segundo Torres (2016), e esse crescimento tende a continuar na casa dos dois dígitos por tempo indeterminado segundo as pesquisas.

"Em 2020, jogos *mobile* representarão por si só mais da metade do total do mercado de jogos global" (MCDONALD, 2017, p.1). Porém, a capacidade gráfica e de processamento dos aparelhos não vem crescendo suficientemente rápido para suprir a demanda de recursos computacionais elevados do crescente mercado de jogos *mobile*, e este é o ponto que torna relevante e justifica o desenvolvimento dessa pesquisa, pois com o uso eficiente dos recursos disponíveis atualmente, e melhorando o aproveitamento dos mesmos, projetos antes limitados pelas capacidades técnicas dos aparelhos poderão ser desenvolvidos e ter uma execução mais fluída e eficaz.

Organização do Trabalho

- **Capítulo 2:** O capítulo 2 disserta sobre as tecnologias, conhecimentos e conceitos necessários para o entendimento e contextualização do assunto sendo abordado e estudado nesta pesquisa.
- **Capítulo 3:** O capítulo 3 discorre sobre a natureza da pesquisa, e explica os métodos e meios utilizados para comprovar e quantificar os testes realizados na pesquisa, apresentando os modelos de *smartphones* utilizados e as ferramentas.
- **Capítulo 4:** O capítulo 4 apresenta o desenvolvimento da pesquisa, explicando todo o processo feito durante a implementação da aplicação e do desenvolvimento de suas funções. Ele apresenta também a realização dos testes e os resultados dos mesmos, demonstrando a conclusão do trabalho da pesquisa.
- **Capítulo 5:** O capítulo 5 apresenta as considerações finais da pesquisa e disserta sobre as conclusões com relação aos métodos e resultados obtidos por meio deste trabalho, e sobre o que ele pretendia realizar.
- **Capítulo 6:** O capítulo 6 demonstra as referências de todos os conteúdos, conhecimentos e pesquisas utilizados no decorrer deste trabalho acadêmico.

2. REVISÃO BIBLIOGRÁFICA

Este capítulo tem como objetivo apresentar o conteúdo teórico para a compreensão do trabalho. O conteúdo da revisão bibliográfica utilizou autores, livros e artigos científicos relacionados à área da pesquisa realizada.

Foram discutidas técnicas e pontos de otimização em Unity 3D, e em que sentido essas técnicas e pontos podem afetar o desempenho final de uma aplicação, e também de que maneira cada quesito da otimização afeta o jogo no produto final.

2.1 Otimização em Jogos Digitais

No mundo dos jogos digitais, o termo otimização tem algumas variâncias entre a sua definição dependendo do autor, ou da área pesquisada, mas todas as definições chegam à um mesmo ponto de conclusão, sendo ele a melhor execução possível do jogo na maior variedade possível de configurações.

Segundo a definição do estúdio de games e serviços na área de jogos QLOC, otimização significa fazer o jogo executar com a mesma taxa de FPS entre a imensa variedade de configurações de *hardware*s existentes no mercado, incluindo configurações de baixo poder computacional (QLOC *apud* THOMAN, 2016).

O ponto que faz com que haja pequenas diferenças entre as definições de otimização de jogos entre os diversos autores, produtoras e estudiosos da área é o fato de que nenhum jogo é exatamente igual, eles podem ser produzidos de maneiras diferentes, em linguagens diferentes, em *Game Engines* diferentes, o que torna cada artefato único e singular à sua maneira.

Fato que faz com que, segundo Dean Sekulic, otimizador há 20 anos da empresa de games CROTEAM, “[...] por um lado, você não pode comparar jogos diferentes e dizer com base nisso se estão otimizados ou não, mas é possível formar um julgamento a respeito disso olhando a qualidade final do produto e a performance relativa que os dois apresentam." (SEKULIC apud THOMAN, 2016, p. 1).

Ou seja, o ponto é que a comparação da otimização entre jogos distintos e as técnicas utilizadas nos mesmo não podem ser relacionadas e comparadas por si só, mas pode-se ter uma base da qualidade da otimização medindo os resultados do desempenho do produto final de cada um e, dessa maneira, comparar os dois artefatos.

2.2 Unity 3D

A *Game Engine* escolhida para o desenvolvimento do jogo no qual serão aplicadas as técnicas de otimização aqui estudadas e analisadas, é a Unity 3D. Ela foi escolhida pelo fato de ser um *Game Engine* que permite em sua licença gratuita para desenvolver um jogo completo com todos os recursos da *Engine* necessários disponíveis.

A Unity possui um sistema multiplataforma excepcional, fazendo a transição do jogo ou aplicativos de uma versão *mobile* para uma versão *desktop*, ou entre diversos sistemas operacionais *mobile*s ou não com apenas uma seleção na hora da importação final. “iOS, Android, Windows Phones, Macs, PCs, Steam, Playstation, Xbox, Wii U, etc. Há diversas plataformas nas quais o seu jogo pode ser públicado, e a Unity torna simples o processo de transferir o seu jogo para as diversas outras plataformas”. (STUDIO PEPWUPER, 2013, tradução nossa).

Além disso, a plataforma aceita duas linguagens de script diferentes sendo elas Javascript e C#, duas linguagens bastante populares no desenvolvimento de scripts e aplicações. Somando isso ao fato de, segundo a própria empresa em seu site de distribuição da aplicação, a Unity 3D pode criar jogos 2D e 3D, *mobile* ou d*esktop*, *single-player* ou *multiplayer*, deixando um leque de oportunidades de criação excepcionalmente grande.

2.3 Detectando problemas de performance

Segundo Dickinson (2015, p. 50), o primeiro ponto a ser analisado durante o processo de otimização de um jogo é a detecção dos problemas de performance.

> A avaliação de performance da maioria dos *software*s é um processo bastante científico: determina-se o máximo de métricas de performance suportados (numero de usuários concorrentes, uso máximo de memória permitido, uso da CPU, etc); são aplicados teste de carregamento da aplicação em cenários onde se tentar simular o comportamento da mesma no dia-a-dia; colhe-se dados de instrumentos de teste e analise; Analisa-se os dados coletados para tentar encontrar possíveis gargalos de performance; Define-se a causa desse problema; Faz-se as alterações necessárias na aplicação e no código para solucionar o problema; e repete-se o processo. (DICKINSON, 2015, p. 50)

O meio e instrumento para realizar essa coleta de dados necessários para a análise da aplicação dessa pesquisa, foi a ferramenta da Unity chamada Unity Profiler. A ferramenta mostra em tempo de execução dados como: uso de CPU por componente da Unity 3D Engine, estatísticas de renderização, uso da GPU nos diversos estágios do jogo, estatísticas de uso de

memória RAM, estatísticas de uso de áudio e uso e estatísticas dos motores de física.

De acordo com as ideias de Dickinson (2015), através da coleta de dados feita pelo Profiler é que se analisa onde está o possível problema de execução, e, com base nos recursos sendo utilizados, projeta-se e implementa-se uma solução para o possível problema. Após a implementação analisa-se novamente o Profiler e se o problema despareceu, continua-se para a próxima análise, caso contrário, continua-se a analisar os dados e preparar outra solução para o gargalo de performance.

2.4 Estratégias de *Scripting*

A programação ou *scripting* de um jogo, segundo Dickinson (2015) consumirá uma grande parte do tempo de desenvolvimento, tendo isso em mente, será de enorme benefício ao projeto a aplicação de boas práticas de programação. Esta é a etapa do processo de criação a qual implementará as ideias e aspectos do jogo definidos durante o desenvolvimento do projeto. É durante a fase de *scripting* que todas as mecânicas e animações do jogo, juntos com o fluxo da história serão implementadas.

Durante essa fase, existem técnicas e métodos de boa prática de programação que podem, de maneira geral, contribuir para uma execução fluída do jogo desde sua primeira codificação. Ao seguir e fazer uso destes aspectos, possíveis problemas futuros de otimização são evitados, fazendo o uso dessas técnicas algo de extremo valor e importância.

2.4.1 Funcionamento das funções *Update(), FixedUpdate()* e de Corotinas

As funções *Update(), FixedUpdate()* e as Corotinas são funções da Unity que impactam bastante no desempenho de um jogo, além de serem, na maioria das aplicações desenvolvidas em Unity, as principais funções e as funções que realizam as principais tarefas, como teste de recebimento de dados e comandos do usuário, cálculos e atualizações de física, entre outras.

Segundo Unity (2013) a função *Update()* é uma função de *callback*[1] nativa da Unity. Ela é chamada uma vez por *frame*, e o intervalo em suas execuções pode variar de *frame* a *frame*. A função *Update()* é normalmente utilizada para a movimentação de objetos da cena que não dependem de física, para implementar temporizadores simples para algum propósito

[1] Uma função de callback é uma função passada a outra função como argumento, que é então invocado dentro da função externa para completar algum tipo de rotina ou ação.

ou para o recebimento de *Input* do usuário, seja ele por teclado, mouse ou algum outro dispositivo de entrada de dados.

Porém, segundo Dickinson (2015), a função deve ser utilizada com cautela, pois ela por si só causa um gasto de recursos computacionais bem alto por tratar-se de um número elevado de chamadas por segundo. Muitas tarefas realizadas utilizando a função *Update()* podem ser efetuadas simplesmente utilizando uma Corotina, o que evitará gastos desnecessários de recurso.

Segundo a documentação oficial da Unity "Extrair valores de um *Update()* é algo que reduz a eficiência do código o qual escrevemos. As vezes é necessário fazer isso, porém é algo o qual deveríamos evitar."(UNITY, 2013)

A função *FixedUpdate()* também é uma função de *callback* nativa da Unity. Ela é chamada em um intervalo constante e inalterado de tempo, não havendo variações entre as suas invocações. A melhor maneira de utilizar este método é para realizar cálculos diretamente ligados ao motor de física da *Engine*, como, por exemplo, controle de objetos que utilizam física, movimentação de personagens por adição de força, lançamentos, colisões, entre outros.

As Corotinas em Unity são tipos de função que executam de tempos em tempos. Intervalos de tempo os quais são definidos pelo programador da função. O principal benefício de se usar uma Corotina, em vez de executar os comandos na função *Update()*, é o fato de nem todas as ações precisarem de uma execução *frame* a *frame*, e fazê-las dessa forma acarretaria em um gasto de recursos adicionais desnecessário.

2.4.2 Armazenando na *cache* referências de componentes

Algumas funções do jogo que precisam ser programadas necessitarão da recuperação do componente que elas estão usando pela Unity, ou seja, segundo a documentação da Unity (UNITY, 2017) é necessária à Engine buscar esse componente através do comando, na totalidade da cena sendo executada no momento.

Em conformidade com as ideias de Dickinson (2015), e Unity Documentation (2017), essa busca é custosa em termos de recurso de CPU, e pode ser evitada usando o seguinte princípio. Ao se referenciar um componente, utilizando-se do método "GetComponent<Componente>()", deve-se armazenar essa referência em uma variável privada, na primeira execução do código.

Ao se iniciar o código e armazenar o componente resgatado na variável, a Unity

Engine não precisará buscar na cena o componente toda a vez que necessitar utilizá-lo, só será necessária essa busca uma vez, pois, a partir dela, este componente ficará armazenado na memória *Cache* da Engine.

Por exemplo, Dickison (2015) apresenta dois códigos para serem analisados, o primeiro de como não se deve fazer, e o segundo usando os conceitos explicados neste tópico:

```
void TakeDamage() {
        if (GetComponent<HealthComponent>().health < 0) {
                GetComponent<Rigidbody>().enabled = false;
                GetComponent<Collider>().enabled = false;
                GetComponent<AIControllerComponent>().enabled = false;
                GetComponent<Animator>().SetTrigger("death");
}
}
```

No trecho de código apresentado foi criada a função *TakeDamage()* que está sendo chamada quando o personagem sofre algum dano. Porém, como essa função é chamada dentro de um *Update(), p*or se tratar de uma função, sofrer dano, que precisa ser testada continuamente, os componentes que a mesma utiliza são resgatados toda vez que a função é chamada.

E, considerando que a função *Update()* é chamada uma vez por frame, este tipo de prática de *scripting* vai causar um custo de recursos desnecessários bem alto. A seguir, o meio de *scripting* apresentado como mais adequado para esta função:

```
private HealthComponent _healthComponent;
private Rigidbody _rigidbody;
private Collider _collider;
private AIControllerComponent _aiController;
private Animator _animator;

void Awake() {

_healthComponent = GetComponent<HealthComponent>();
```

```
_rigidbody = GetComponent<Rigidbody>();
_collider = GetComponent<Collider>();
_aiController = GetComponent<AIControllerComponent>();
_animator = GetComponent<Animator>();

}

void TakeDamage() {

if (_healthComponent.health < 0) {
_rigidbody.detectCollisions = false;
_collider.enabled = false;
_aiController.enabled = false;
_animator.SetTrigger("death");

}
}
```

No trecho acima, é mostrado como esse código deve ser escrito para ser eficiente em chamadas e referências de componentes.

2.4.3 Métodos de obtenção de componentes

Existem, na Unity, três métodos para a obtenção de componentes, sendo eles:

- “GetComponent(“String”)”,
- ”GetComponent(typeof(Tipo))”, e
- “GetComponent<Componente>()”.

Com essa variedade de métodos disponíveis para uma mesma função, saber qual método é o mais rápido e menos custoso é essencial para um uso otimizado dos recursos.

Dickinson (2015) faz um estudo de caso para definir qual dos três métodos de recuperação de componente é o mais veloz. Durante o teste, foram testados os três métodos em uma mesma tarefa. A tarefa era recuperar um componente um milhão de vezes, e, de acordo com os dados recolhidos, verificar o mais rápido dos métodos.

Os dados apresentados pelo resultado da pesquisa variaram das versões 4.x, para as versões 5.x da Unity. Nas versões de Unity 4.x, o método que se apresentou mais rápido em execução foi o “GetComponent(typeof(Tipo))”. O método demonstrou ser cerca de cinco vezes mais rápido que o “GetComponent(“String”)”, e alguns significantes milissegundos mais rápido que o “GetComponent<Componente>()”, tornando este método, o recuperador de componentes ideal para se utilizar em versões da Unity 4.x.

Já nos teste com as versões 5.x, os métodos “GetComponent(typeof(Tipo))” e “GetComponent<Componente>()” apresentaram resultados praticamente idênticos, porém com alguma vantagem do método “GetComponent<Componente>()”, cerca de 1 milissegundo. Estes números em proporções cotidianos podem parecer ínfimos, mas em ciclos de máquina podem ser a diferença entre o ganho ou a perda de um *frame*.

De acordo com a Unity Documentation (2017), quando se executa um código diversas vezes em uma cena, deve se tirar o máximo de proveito deste código, para que o mesmo tenha a melhor execução e ganho de desempenho possível.

2.4.4 Removendo funções de retorno vazias

Um ponto aparentemente simples, mas que o acúmulo pode causar um uso desnecessário de ciclos de CPU são funções chamadas de funções de “*callback*” vazias no código. Por padrão o editor de código que é instalado juntamente com o Unity é o MonoDevelop. E durante a criação de um novo *script*, esse editor cria o novo *script* com duas funções prontas já inseridas nele, as funções de *Start()*, e *Update()*.

A função *Start()* é a função que é carregada no momento em que o código ou objeto em que o código está inserido é instanciado na cena. Esta função é executada apenas uma vez por instância de objeto. Já a função *Update()* é executada uma vez por *frame*, ou seja, se o jogo é executado a sessenta *frames* por segundo, a função *Update()* é executada sessenta vezes por segundo, segundo Unity Documentation (2017).

Em acordo com as ideias de Dickinson (2015) e de Martins (2014) este ponto torna importante a limpeza do código para evitar que estas funções de “*callback*” vazias sejam processadas pela CPU, que mesmo estando vazias, ocupam ciclos de máquina que poderiam estar sendo usados para a renderização de texturas do jogo, por exemplo.

2.4.5 Desabilitando *scripts* e objetos que não estão em uso

Objetos e *scripts* podem estar em execução em uma cena mesmo após sua finalidade já ter sido cumprida, e é esse o ponto desse tópico. Se um objeto já cumpriu a sua função naquele momento, se está tão distante do jogador que não faz diferença ele estar em cena ou não, ou se o objeto não está visível para o jogador, esse objeto pode ser temporariamente desativado para economia de memória e processamento.

Segundo a Unity Documentation (2017), existem dois métodos de "*callback*" para cumprirem a função de desabilitar um objeto através de código quando o mesmo não está visível para as câmeras do jogo. São eles os métodos "*OnBecameVisible()*" e o "*OnBecameInvisible()*".

Esses dois métodos serão invocados quando suas respectivas condições forem cumpridas e se os mesmos estiverem sendo chamados em algum *script*. O método "*OnBecameVisible()*" é chamado quando o objeto no qual o *script* está invocando o método fica visível para a câmera, ou seja, para o jogador. Já o método "*OnBecameInvisible()*" é chamado quando o objeto no qual ele está contido fica invisível para a câmera.

Usando as possibilidades que estes dois métodos oferecem é possível habilitar e desabilitar objetos em tempo de execução de jogo, utilizando a linha de código "*OnBecameVisible(){gameObject.SetActive(true)}*" e "*OnBecameInvisible(){ gameObject.SetActive(false) }*". Utilizando-se destes dois comandos para desabilitar objetos não necessários no momento na cena, proporcionará um ganho de performance devido ao fato de haver menos objetos para serem processados na cena.

Dickinson (2015) apresenta outro meio de desabilitar objetos em cena em tempo de execução, sendo ele desabilitar objetos pela distância. Neste outro meio, segundo a ideia do autor, desabilita-se um objeto da cena por ele estar tão distante do jogador que o mesmo provavelmente não notará, e mesmo que note, o objeto desabilitado não está tendo interferência nenhuma no momento, no *gameplay* do jogador.

O princípio e comando para desabilitar o objeto aqui é o mesmo do anterior "*gameObject.SetActive(true)*" ou "*gameObject.SetActive(false)*", havendo necessidade apenas da adição de uma função de controle da distância, para dizer o momento de ativar e desativar determinado objeto.

2.5 Batching

O processo de *Batching* segundo Wloka (2003) e Korzuszek (2015) consiste no envio de um pacote de objetos similares para renderização na GPU. O envio é feito através de uma *Draw Call* é apenas são incluídos em um mesmo pacote objetos que dividem o mesmo *shader* e material. Existem dois tipos de *Batching* na Unity Engine, o *Batching* Estático e o *Batching* Dinâmico, ambos serão abordados neste tópico.

2.5.1 Draw Calls

Para compreender completamente o processo de *Batching* é necessário entender o conceito de *Draw Calls*. “Em sua forma mais básica, uma *Draw Call* é um pedido de desenho de um objeto enviado da CPU à GPU.” (DICKINSON, 2015, p. 151). Na GPU, essa *Draw Call* é direcionada a API gráfica da mesma, podendo ser a OpenGL ou Direct3D.

> *Draw Calls* requerem muitos recursos da API gráfica, necessitando de um trabalho significativo da mesma em toda *Draw Call*, causando uma sobrecarga de performance pelo lado da CPU. Isto é causado principalmente pela mudança de estado que ocorre entre as *Draw Calls* (como por exemplo, a troca de materiais), causando o uso intensivo de recursos na validação e translação pelos passos cumpridos pelo driver gráfico em cada chamada. (UNITY, 2013, p. 1)

De acordo com as constatações dos autores, *Draw Calls* são recursos utilizados em todos os desenhos requisitados pela *Engine* Unity, e são inerentemente intensivos na parte de recursos computacionais. Logo essas funções precisam ser usadas eficientemente para não causar uma sobrecarga de processos na CPU. E é exatamente sobre um meio de amenizar essa possível sobrecarga de recursos computacionais de CPU devido as chamadas de *Draw Call* que a técnica de *Batching* trata.

2.5.2 *Batching* Dinâmico

O *Batching* Dinâmico trata do processo de Batching para objetos que se movem na cena. Este processo é realizado automaticamente pela *Engine* Unity. Porém para que o processo ocorra, é necessário que estes objetos compartilhem o mesmo material e cumpram alguns critérios como:

- Realizar o processo de *Batching*, segundo Unity (2013), em objetos dinâmicos causa certa sobrecarga pelo lado da CPU, e por esse motivo o *Batching*

Dinâmicos só é aplicado em malhas que contem menos de novecentos vértices totais.

- Os objetos a serem incluídos no lote devem, geralmente, estar utilizando-se da mesma escala de tamanho. Aplicada a exceção somente à objetos do mesmo tipo que não possuem escalas que se repetem, ou seja, objetos com escalas não-uniformes.
- Usar instâncias diferentes de um mesmo material fará com que os objetos não sofram o processo de *Batching*, mesmo que o material seja uma cópia do outro que está sendo utilizado.
- Objetos com Mapas de luz têm parâmetros de renderização adicionais como o Índice de mapa de luz (*lightmap Index*), e *offset/scale* referentes ao mapa de luz. Logo, objetos dinâmicos que se utilizam de mapas de luz devem apontar exatamente para uma mesma localização do mapa de luz para ser realizado o *Batch* deles juntos, segundo Korzuszek(2015).
- O material deve usar o chamado *single-pass shader*.

Estas são as condições e critérios necessários para que ocorra o *Batching* Dinâmico. A desenvolvedora da *Engine* Unity, pode alterar esses parâmetros a qualquer momento, mas qualquer alteração que seja realizada neste tipo de função é relatada no manual e documentação do *software online*.

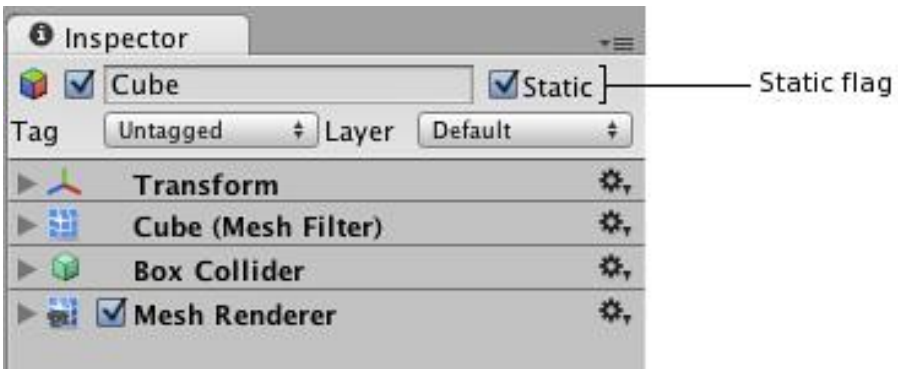
2.5.3 *Batching* Estático

O *Batching* Estático é o processo de *Batching* para objetos que não irão se mover em nenhum momento durante o jogo. É uma técnica bastante eficiente para o desenho e renderização de formas geométricas de qualquer tamanho. Ao contrário do *Batching* Dinâmico, o Estático é bastante eficiente no quesito uso de CPU, pois ele não transforma os vértices no processador. Porém este método é mais custoso no quesito de memória, pois todos os índices e cálculos que foram feitos para renderizar as formas estáticas são armazenadas diretamente na memória (UNITY, 2013).

Isso faz com que o método, se mal utilizado, acabe sobrecarregando a memória, acarretando em perda de desempenho e outros problemas de execução da aplicação. Para um objeto ser tido pela *Engine* como estático, é necessário marcá-lo como tal na região do

Inspector da cena no Unity. A figura 1 demonstra a marcação de um objeto para ser renderizado como estático.

Figura 1: *Checkbox* da propriedade estática para *Batching*



Fonte: https://docs.Unity3d.com/uploads/Main/StaticTagInspector.png

2.6 Motores de física

Os motores de física são parte essencial dos jogos nos dias atuais, seja calculando colisões entre objetos e o jogador, manuseando e direcionando interações do ambiente, ou controlando qualquer ação de natureza física que possa vir a acontecer no jogo. O principal ponto que leva à busca de melhorias no sistema de física durante a otimização de um jogo é o fato de toda oscilação na frequência de *frames*, todo fechamento inesperado do jogo e toda função que seja em questão de recursos custosa demais para uma determinada configuração de máquina, e que afete parte dos jogadores do jogo, afeta o que o jogador pensa sobre o jogo.

Esse tipo de problema causa frustração no jogador, e muitas vezes pode tirá-lo da experiência de imersão no jogo, explica Dickinson (2015). Logo, entender como os motores de física funcionam, e utilizá-los de maneira eficiente é um ponto importante para o desempenho do jogo e a experiência final geral do jogador.

2.6.1 Física e o tempo na Unity

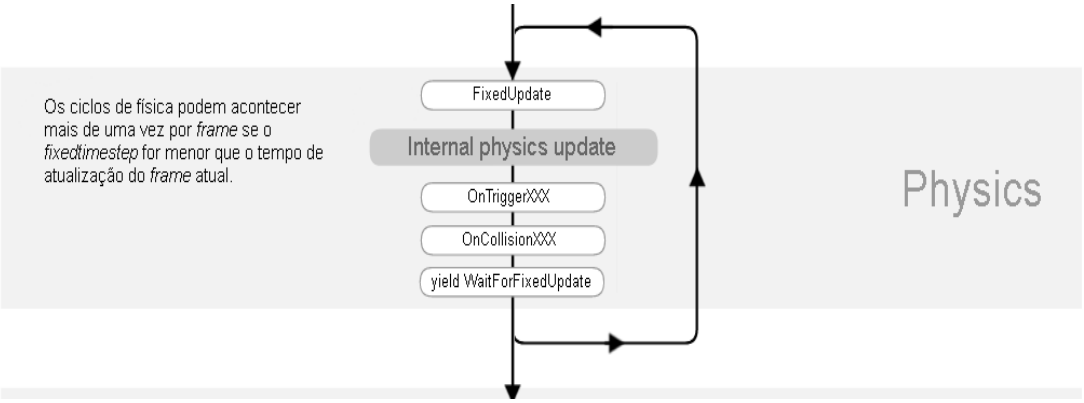
Motores de física normalmente funcionam sob o pressuposto de que as interações ao decorrer do tempo ocorram sempre em um intervalo de tempo fixo, ou seja, para que não haja alteração no intervalo de tempo em que as interações físicas ocorrem. Isso é necessário porque cálculos físicos. na maioria das aplicações. não podem ser interferidos por taxas de frame altas ou baixas.

Desta maneira, esse tipo de cálculo deve ser indiferente a tais fatores, explica Dickinson (2015). Esse tempo fixo entre interações é o intervalo entre execuções do método *FixedUpdate()*, conhecido na área de desenvolvimento em Unity como *Fixed Update timesteps*. Segundo Unity, o tempo de intervalo padrão entre as interações no *FixedUpdate()* é de 20 milissegundos.

2.6.2 Funcionamento de execução do *FixedUpdate()*

O método *FixedUpdate()* representa o momento no qual o motor de física prepara e recolhe, se houver, seus dados para serem atualizados. A figura 2 mostra a ordem de execução dos componentes da Unity, na parte de Física:

Figura 2: Ordem de prioridade de execução de métodos do motor de física



Fonte: https://docs.Unity3d.com/Manual/ExecutionOrder.html

Como demonstrado na figura 2, o método *FixedUpdate()* é invocado antes mesmo de o próprio motor de física fazer suas atualizações.

Dickinson (2015) explica o processo de invocação do *FixedUpdate()* da seguinte forma: o processo começa por um cálculo determinando se passaram 20 milissegundos desde a execução da última invocação do método. O comportamento desse processo irá variar dependendo do resultado deste cálculo.

Caso haja passado os 20 milissegundos, o método *FixedUpdate()* é invocado globalmente, e todo e qualquer interação física é logo em seguida calculada e atualizada. Seguida pelos métodos do tipo *OnTrigger()* e *OnCollision()*. E logo em seguida quaisquer Corotinas que estejam ligadas ao *FixedUpdate()* através da função *yield WaitForFixedUpdate()* são executadas conforme seus *scripts*.

Porém, caso não haja passado 20 milissegundos desde a última execução do método, então o *FixedUpdate()* que seria executado no cálculo de tempo atual é ignorado e não executado no momento. Em seguida, outros processos na fila de execução desse *frame* tomam parte e continuam a sua execução, como entradas de comandos, lógica, e renderização de objetos da cena continuam sua execução. Quando terminada, é realizado novamente o cálculo para execução do *FixedUpdate()* e o processo acima explicado se repete inúmeras vezes durante o tempo de execução.

Essa execução extra de métodos que ocasiona a segunda possibilidade explicada pode ser causada por configurações que tenham taxas de *frames* altas, fazendo com que o os métodos executem mais vezes do que em outras configurações. Apesar de "ignorar" a execução do método se a condição explicitada não for cumprida, não é possível, por parte do usuário, notar alguma diferença em performance, garantindo uma movimentação suave mesmo sem a execução do método naquele ciclo.

> Com o objetivo de garantir uma execução suave durante os *frames* onde as atualizações de física são ignoradas, alguns motores de física (incluindo o presente na Unity 2D e 3D) interpolam o estado entre o estado de física do *FixedUpdate()* anterior e o estado atual baseado em quanto tempo falta para a próxima invocação do *FixedUpdate()*. (DICKINSON, 2015, p. 211)

Ou seja, baseado na situação do estado das físicas do *FixedUpdate()* anterior, após ignorar o método *FixedUpdate()* naquele frame, o motor de física faz um cálculo para saber quanto tempo falta até a próxima invocação do método, e com base nesse tempo predizer como o estado atual deve se comportar até a próxima invocação garantindo, assim, que esses cálculos se tornem imperceptíveis na cena para o jogador.

2.6.3 Colisores e detecção de colisão

Colisores, segundo Unity (2017), definem o formato do objeto no qual estão, para propósitos de cálculos de colisão. Colisores são invisíveis ao jogador, e podem ter a mesma forma do modelo ou *sprite* o qual representam, ou podem ser compostos de formas primitivas.

"Os mais simples (e menos custosos em recursos de CPU) colisores são os chamados Colisores de Tipos Primitivos (*primitive collider types* em inglês). Em 3D são eles colisor de caixa, colisor de esfera e colisor de cápsula." (UNITY, 2017).

Colisores podem ser separados em dois tipos mais utilizados na Unity, Colisores Estáticos, que são colisores que não têm *Rigidbody* no objeto ao qual fazem parte. Esse tipo

de colisor é comumente utilizado para cenários e objetos que não irão se movimentar. Colisores estáticos interagem normalmente com outros colisores, mas não são movimentados por essas colisões.

Ao sinalizar um colisor como estático, não colocando um componente de *Rigidbody* nele, o motor de física da Unity poderá fazer cálculos e otimizações automaticamente em cima dessa característica, melhorando o desempenho desse objeto seja qual for sua função. A Unity não libera detalhes internos do seu motor de física e sobre as otimizações realizadas, são apenas afirmados estes dados com bases nos seus testes internos, e testes de usuários e empresas (DICKINSON, 2015).

Movimentar um objeto estático pelo componente *Transform* em tempo de execução é extremamente custoso em termo de CPU, salienta Unity (2017), pois ao fazer isso o motor de física deve recalcular todos os cálculos que realizou anteriormente no pressuposto de que este objeto estático não se moveria, causando diversos problemas e comportamento imprevisível por parte do objeto. Por essa razão este tipo de colisor deve ser implementado apenas em objetos que não se moverão durante a execução do jogo.

O outro tipo de colisor é chamado de Colisor Dinâmico ou Colisor de *Rigidbody*. Colisores dinâmicos são colisores em objetos que se movimentam em tempo de execução da aplicação, objetos que possuem *Rigidbody*. Este tipo de colisor é completamente controlado pela física da cena, sofrendo influência de outras colisões, de forças aplicadas, ou qualquer outra interação que venha ocorrer com o mesmo. É, segundo a Unity (2017), a configuração de colisor mais utilizada em jogos que se utilizam de física.

Existem três tipos de detecção de colisão na *Engine* Unity: Discreto, Contínuo e Contínuo Dinâmico. A detecção de colisão discreta ocorre da seguinte forma: assim como o método *FixedUpdate()* ela ocorre entre intervalos de tempo pré-determinados pela variável *Time.fixedDeltaTime*. Assim que o intervalo estabelecido se completa, o algoritmo da detecção discreta realiza cálculos de volumes na fronteira do objeto ao qual está realizando o teste. Se houver sobreposição deste objeto com algum outro neste momento então é detectada a colisão e tratada como foi programada para ser (DICKINSON, 2015).

Segundo a Unity (2017) este é o método de detecção mais rápido dentre os três, mas segundo Dickinson (2015) pode haver falhas na detecção de colisão com objetos muito pequenos se movendo a velocidades muito altas. Isso se dá pelo fato de estas colisões serem testadas em um intervalo de tempo fechado, logo, se as colisões acontecerem entre estes intervalos, ou seja, mais rápidas que a detecção, haverá a perda da detecção desta colisão.

Ambos os métodos de detecção de colisão Contínuos se comportam interpolando

objetos de sua posição inicial até sua posição final naquele determinado *timestep* (intervalo de tempo de execução de cálculos físicos) e verificando se houve alguma colisão no caminho. De acordo com Dickinson (2015) esses métodos diminuem as chances de alguma colisão ser perdida durante a execução, contudo, estes métodos são significativamente mais custosos em CPU do que o método de detecção Discreto.

A diferença entre o método de detecção Contínuo e o Contínuo Dinâmico é que o método Contínuo apenas habilita a colisão contínua com objetos de colisão estática, ou seja, que não possuem *Rigidbody*. Quando este colidir com objetos de colisão dinâmica, que possuem *Rigidbody*, a colisão será tratada como Discreta, e não Contínua.

Já o método Contínuo Dinâmico, habilita a colisão continua com qualquer outro tipo de colisor, podendo ele ser estático ou dinâmico.

2.7 Manuseio eficiente de memória

Para um manuseio eficiente da memória disponível para a execução das aplicações desenvolvidas em Unity, é necessário primeiro entender como a Unity *Engine* trata a memória que lhe é disponibilizada pelo sistema operacional do aparelho o qual está executando a aplicação.

Segundo Dickinson (2015) o espaço de memória disponível para a Unity 3D pode ser essencialmente dividido em três domínios de memória diferentes, sendo eles o Domínio Nativo, Domínio Gerenciável (*Managed Domain*) e o domínio pertencente às *DLLs (Dynamic Link Library)*, nativas ou implementadas pelo usuário.

O Domínio Nativo é o domínio pertencente ao código base o qual a *Engine* foi desenvolvida. É a parte referente a memória que foi desenvolvida pelos criadores da ferramenta, e a qual não é possível ao usuário acessá-la ou modificá-la diretamente. Esta área cuida de alocações de espaço de memória para funções como dados de texturas e malhas do projeto, sistemas de renderização, física, entradas de dados por parte do usuário, entre outros (REICH, 2013).

É nessa área da alocação de memória onde se encontram as representações nativas de objetos e componentes da cena, como os objetos de jogo (*GameObjects*) e os componentes destes objetos. Esta é a área onde estes objetos armazenam seus dados, como por exemplo, dados referentes aos componentes *Rigidbody* e *Transform*.

Não há controle direto por parte do usuário da Unity desta região de memória, mas há como influenciar essa alocação indiretamente através dos diversos níveis de alocação de

memória e boas práticas de *scripting* às quais o desenvolvedor tem controle e influencia sobre.

Todo objeto criado na região de memória gerencial tem uma representação no espaço no domínio de memória nativo, e ambos estão intrinsicamente conectados. Logo, a melhor maneira de minimizar as alocações de memória no domínio nativo é simplesmente otimizar a sua representação no domínio de memória gerenciável. (DICKINSON, 2015)

O segundo domínio de memória, o Domínio Gerenciável, é a área da memória controlada pelo compilador e linguagem a qual o usuário da *Engine* tem acesso. Todos os *scripts* de objetos e classes criadas pelo desenvolvedor são armazenados neste espaço de memória.

Neste espaço de memória ficam armazenadas representações dos objetos verdadeiros, os quais são armazenados no domínio nativo. Ou seja, o usuário manipula representações dos objetos verdadeiros, mas não tem acesso aos objetos em si. Encontra-se neste local a ponte entre o código nativo da *Engine* e o código que está sendo desenvolvido pelo usuário. A área do Domínio Gerenciável de memória é mantida e controlada pelo *Garbage Collector* (Coletor de lixo) da *Engine*.

2.7.1 *Garbage Collector*

A definição do método de *Garbage Collection* segundo a desenvolvedora da *Engine* Unity é a seguinte.

> Quando executamos nosso jogo, o mesmo usa memória para armazenar seus dados. Quando esta memória não possui mais uso, o espaço de memória o qual estes dados estavam alocados é liberado para que possa ser reutilizado. *Garbage* (Lixo, em inglês) é o termo utilizado para se referir à memória que já foi utilizada para seu propósito, porém não possui mais uso no momento. *Garbage Collection* é o nome dado ao processo de liberação de espaço para reuso da memória. (UNITY, 2017, p. 1)

Para uma melhor compreensão do sistema de *Garbage Collection* é necessário entender como a Unity trabalha com o acesso à memória.

A *Engine* tem acesso a duas regiões da memória, que são chamadas de *Stack* e *Heap*. Segundo Dickinson (2015), *Stack* é um pequeno espaço especial de memória reservado para alocações de memória de curta duração, que são automaticamente desalocados quando se encontram fora de escopo, inacessíveis pelo código.

São armazenados no *Stack* variáveis locais, e esta área da memória é responsável por

manusear o carregamento das funções quando as mesmas são invocadas. Não há necessidade de limpeza ou desalocação de memória nessa região, pois assim que um dado se torna inalcançável pelo código, novos dados simplesmente o sobrescrevem na memória, e o dado antigo é substituído.

E de acordo com Unity (2017), *Heap* ou *Managed Heap* representa todo o espaço de memória restante. Quando um dado é grande demais para ser alocado no *Stack* ele é alocado no espaço de memória do *Heap*. Dados armazenados nessa região da memória ficam ativos por mais tempo, e necessitam de uma função para desalocar dados que não estão mais sendo utilizados, e esta é a função do *Garbage Collector*.

Como os dados armazenados no *Heap* duram por mais tempo, eles não são sobrescritos por novas entradas de dados, logo a função do *Garbage Collector* é procurar espaços de memória disponíveis para novos dados, e desalocar dados em desuso. Se não há espaço suficiente disponível no *Heap* controlado pela Unity, mesmo após a limpeza dos dados, o *Garbage Collector* solicitará mais memória ao sistema operacional, podendo assim alocar o novo dado que necessita de alocação no *Heap.*

O tempo de execução da função do *Garbage Collector* aumenta conforme a quantidade de memória utilizada e os tipos de dados e funções sendo executadas. E é por esse motivo que ele deve ser controlado e invocado em horas oportunas. Segundo Unity (2017) uma boa maneira de se realizar isto é realizando a coleta de lixo manualmente, e em horas oportunas, como por exemplo, durante uma tela de carregamento, quando o jogo for pausado ou durante a execução de uma cinemática.

A maneira de se fazer isso é utilizando o comando “System.GC.Collect()”, invocando assim manualmente o *Garbage Collector* e evitando possível queda de *frame* ou congelamento da tela do jogo.

Outra maneira de tornar o coletor de lixo mais eficiente é “criando menos lixo”. Armazenar referencia de componentes e transformações na *cache*, como explicado no tópico 2.4.2 e 2.4.3 desta pesquisa, é uma maneira, de acordo com Unity (2017), de se criar menos lixo, desta forma não haverá necessidade da execução do método do *Garbage Collector*, economizando o recurso de CPU e memória que este método utilizaria. Recurso o qual é bastante custoso em termos de CPU e memória.

3. METODOLOGIA

A pesquisa realizada foi de abordagem qualitativa e natureza aplicada, pois, os conhecimentos adquiridos na elaboração desta pesquisa serão úteis para informar aqueles que se interessam pelo assunto deste trabalho acadêmico e, além disso, a pesquisa destina- se também para acadêmicos, professores, alunos, desenvolvedores e empresas que buscam aprofundar seus conhecimentos no assunto através de uma abordagem acadêmica e científica. Conforme Prodanov e Freitas (2013), o objetivo do estudo é explicativo, pois, procura identificar os fatores que causam um determinado fenômeno, aprofundando o conhecimento da realidade.

O procedimento realizado neste trabalho é de abordagem prática, aplicando os conhecimentos e técnicas de base teórica em uma situação prática criada para testar a eficiência e veracidade dos conhecimentos. A situação prática foi o desenvolvimento de um jogo *mobile* Android 3D utilizando a *Engine* estudada neste caso, Unity. A coleta das informações de desempenho e dados relevantes para a conclusão da pesquisa foi realizada utilizando a ferramenta *Profiler* da própria Unity. Ferramenta ao qual o funcionamento foi explicado durante o tópico 2.2.1 deste trabalho.

Os testes foram realizados em três aparelhos celulares diferentes sendo eles *Samsung Galaxy J5 Prime*, *Asus Zenfone 5* e *Samsung Galaxy J7*. Os aparelhos foram escolhidos pelo critério de capacidades técnicas, onde tendo o *Samsung Galaxy J7* como configuração mais avançada para os testes, e o *Galaxy J5 Prime* e o *Asus Zenfone 5* como configuração mais modestas. Com o propósito de testar a aplicação e as técnicas aplicadas em configurações mais abrangentes.

O objetivo dos testes foi a execução fluida da aplicação nos aparelhos sem haver queda de *frames* ou sobrecarga da capacidade de processamento ou da capacidade gráfica dos aparelhos.

O jogo ao qual as técnicas estudadas aqui foram aplicadas foi desenvolvido em Unity 3D para a plataforma *mobile* Android, na linguagem de programação *C#*. O método de desenvolvimento foi por prototipação. O método foi escolhido pelo motivo de facilitar o entendimento dos requisitos, e melhor elaborar os conceitos e funcionalidades do *software*. Além de facilitar a distribuição das técnicas por área da otimização, aplicando-as de acordo com as áreas dos protótipos sendo desenvolvidos.

Foi escrito para a documentação do jogo o GDD completo (Apêndice A), contendo

todas as informações técnicas e criativas do jogo, seu roteiro, os requisitos funcionais e não funcionais, e os diagramas de Classes e Casos de Uso. Os diagramas foram desenvolvidos na ferramenta de criação de diagramas de UML *Astah Community*.

A parte artística e de design do jogo foi desenvolvida utilizando a ferramenta de modelagem *Blender*, e o *software* de edição de imagens da *Adobe Photoshop CS6*.

Segundo Prodanov e Freitas (2013), é imprescindível correlacionar a pesquisa com um universo teórico, optando por um modelo que sirva de embasamento à interpretação do significado dos dados e fatos colhidos e levantados. Por meio do procedimento citado acima, estudou-se e avaliou-se a efetividade da aplicação de técnicas e conhecimentos de otimização no desenvolvimento de jogos para dispositivos Android desenvolvidos em Unity3D.

4. RESULTADOS E DISCUSSÕES

Este capitulo da pesquisa discorre sobre como ocorreu o desenvolvimento do jogo e sobre a aplicação das técnicas e conceitos de otimização abordados e estudados durante a pesquisa.

Além de apresentar os resultados dos testes de performance realizados da aplicação em diferentes modelos de smartphone, como explicado no capitulo 3 desta pesquisa, validando e demonstrando de forma prática os ganhos de performance e os pontos de atuação de cada aspecto da otimização abordado no decorrer do trabalho.

4.1 Desenvolvimento do jogo

O primeiro passo no desenvolvimento da aplicação foi definir como as estruturas básicas do jogo como, movimentação do personagem, cenário e dificultadores, seriam desenvolvidas. A criação do cenário foi proposta da seguinte forma: o personagem se movimentará sobre um chão que o seguirá durante o seu percurso, logo, o pedaço do chão o qual o personagem tem contato será sempre o mesmo, economizando assim recursos de memória e renderização.

O personagem se movimentará pelo chão, seguindo o fluxo da fase, e em determinado momento, o chão se movimentará para a frente do personagem, criando assim um fluxo infinito de terreno o qual o jogador pode se movimentar sobre. Essa movimentação, por conta de o terreno ser homogêneo, é imperceptível ao mesmo, dando a impressão de um terreno contínuo e infinito.

A figura 3 mostra o *script "FollowPlayer"* que foi desenvolvido para desempenhar a função do chão já descrita. O *script* contem uma variável privada que armazena as coordenadas da localização do jogador, em uma variável do tipo *Transform*. No método *Start* é atribuído um valor à essa variável utilizando o método *FindGameObjectWithTag* da classe *GameObject*. Esse é um método da própria Unity, que procura na cena um objeto que contenha a *Tag* que foi passada para o método, no nosso caso, a *Tag "Player".*

Após ter o jogador como referência na variável *playerTransform*, é iniciada a Corotina *MoveGround.* Essa corotina, de tipo privado, aguarda seis segundos após sua chamada de execução, e após esse tempo, movimenta o chão no eixo Z, para a posição atual do jogador. Ou seja, a posição do chão é reiniciada em relação ao jogador, e pode percorrer novamente toda a extensão do objeto do chão indefinidamente.

Esse *script* está ligado com o chão do cenário e o chão o qual o jogador percorre, para que a sensação de imersão que o mesmo causará abranja todos os aspectos da movimentação do ambiente.

Figura 3: Representação do *script* "*FollowPlayer*".

```
1  using System.Collections;
2  using UnityEngine;
3
4  public class FollowPlayer : MonoBehaviour {
5
6      private Transform playerTransform;
7
8      void Start () {
9          playerTransform = GameObject.FindGameObjectWithTag("Player").transform;
10         StartCoroutine(MoveGround());
11     }
12
13     private IEnumerator MoveGround()
14     {
15         yield return new WaitForSeconds(6f);
16         transform.position = new Vector3(0, transform.position.y,
17             1 *playerTransform.position.z);
18         StartCoroutine(MoveGround());
19         yield return null;
20     }
21 }
22
```

Fonte: Elaborada pelos autores

O próximo ponto relativo ao ambiente são os objetos de cenário que estarão presentes na cena. Durante o decorrer da fase, que se passa em uma floresta, o jogador corre continuamente e passa por inúmeras árvores. Esse é um ponto que se não for corretamente desenvolvido e implementado, pode causar um grande desperdício de recursos computacionais, pois como serão diversas arvores, tratando-se de uma floresta, logo cada arvore ocupará um espaço de memória. Essa mecânica tem de ser implementada para utilizar a memória e Draw Calls da maneira mais eficiente possível.

Essa mecânica do cenário foi desenvolvida da seguinte forma. Um número relativamente grande de árvores foi necessário para suprir o papel delas na imersão do jogador no ambiente da floresta durante o jogo. Foram usados vinte e oito objetos de arvore no cenário ao todo. A sensação de continuidade do mesmo foi feita através do script "GlacierSpawner" demonstrado na imagem a seguir. O script é ligado à um objeto vazio chamado "Forest", e este será responsável por carregar o código desenvolvido, e os objetos filhos que serão as

vinte e oito arvores.

Figura 4: Representação do *script* "*GlacierSpawner*"

```
using UnityEngine;

public class GlacierSpawner : MonoBehaviour {

    private const float DISTANCE_TO_RESPAWN = 3.0f;

    public float totalLenght;
    public bool IsScrolling { set; get; }

    private Transform playerTransform;

    private void Start()
    {
        playerTransform = GameObject.FindGameObjectWithTag("Player").transform;
    }

    private void Update()
    {
        if (!IsScrolling)
            return;

        if(transform.GetChild(0).position.z <
            playerTransform.position.z - DISTANCE_TO_RESPAWN)
        {
            transform.GetChild(0).localPosition += Vector3.forward * totalLenght;
            transform.GetChild(0).SetSiblingIndex(transform.childCount);

            transform.GetChild(0).localPosition += Vector3.forward * totalLenght;
            transform.GetChild(0).SetSiblingIndex(transform.childCount);
        }
    }
}
```

Fonte: Elaborada pelos autores

O script tem uma sequência constante de funcionamento. Ele possui quatro variáveis essenciais para sua execução, que são elas "DISTANCE_TO_RESPAWN", uma constante privada do tipo float que dita a distância a qual a arvore precisa estar do jogador para poder reaparecer à frente no cenário. "totalLenght", uma variável pública do tipo float que define o tamanho total desse cenário em comprimento. "IsScrolling", uma propriedade pública do tipo booleana que retorna o valor dizendo se o cenário está se movimentando ou não, e "playerTransform", que tem exatamente a mesma função explicada no script anterior.

O início do script tem a mesma função do anterior, que é atribuir a posição do jogador à variável "playerTransform". Seguido pela função Update, que é onde acontece a

lógica principal deste script. No início da função, é realizado um teste com a propriedade “IsScrolling”, para saber se seu valor é verdadeiro ou falso. Se falso, acontece o retorno da função, e nada da lógica escrita abaixo é executado, nada acontece com os objetos manipulados por esse código.

Se verdadeiro, o código continua sua execução seguido por um teste condicional if para testar se está na hora de um dos objetos filhos de “Forest” se movimentar. O código “transform.GetChild(0).position.z” retorna a posição em Z da primeira arvore ao lado esquerdo mais próxima ao jogador, em seguida é feito o teste se esse valor retornado é menor que a posição do jogador em Z menos a distância que precisa ser passada para a árvore retornar à frente do cenário, que é o valor “DISTANCE_TO_RESPAWN”.

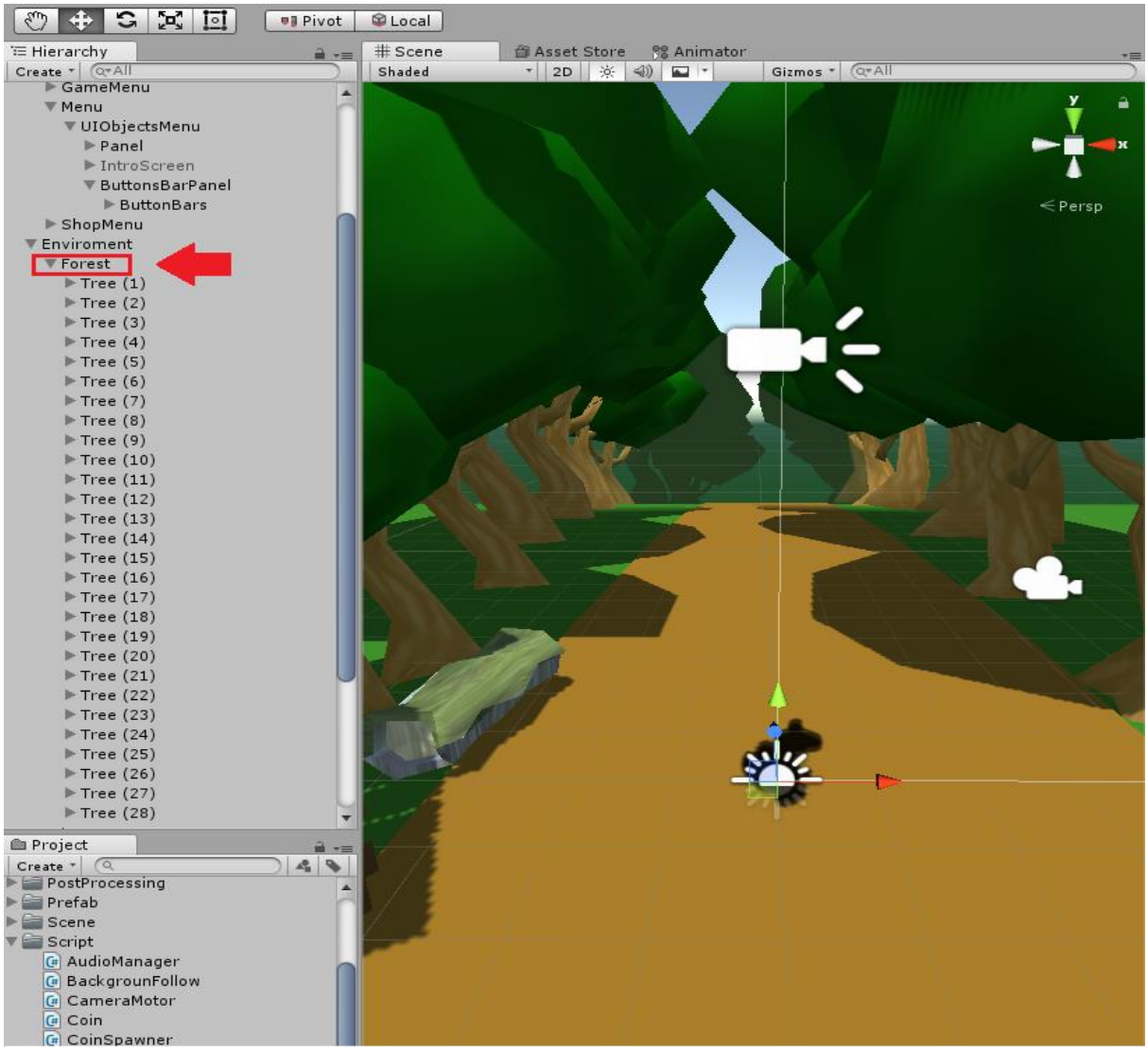
Se esse teste for verdadeiro, é executado o código de dentro do if, que pega essa mesma árvore que acabou de ser testada, e a manda para o final do cenário, que são as arvores à frente do jogador, as quais o mesmo ainda não alcançou. Após posicioná-la à frente no cenário, é mudado seu Index de objeto em relação as outras árvores filhas de “Forest” e está arvore se torna a última da fila.

O mesmo código executado acima é executado novamente no mesmo bloco de if, fazendo as mesmas alterações, só que agora para a arvore que estava em segundo lugar na fila, porque assim que o índice da árvore anterior foi alterado para o último, o índice da segunda arvore virou o primeiro, continuando assim o ciclo.

Finalizando assim o bloco if, e o código do Update que precisava ser executado neste frame. O script foi desenvolvido intencionalmente para movimentar duas árvores de cada vez, e as arvores estão posicionadas no objeto “Forest” de modo que a primeira esteja do lado esquerdo, a segunda no direito, e assim sucessivamente intercalando os objetos. Dessa forma, a movimentação das árvores fica mais fluída, e contínua, contribuindo para a sensação de um mapa infinito e constante passada pelo jogo.

A seguir a representação do objeto “Forest” na hierarquia de cena do Unity:

Figura 5: Representação do objeto "*Forest*", na cena do jogo



Fonte: Elaborada pelos autores

Tratar os objetos de cenário dessa forma faz com que seja economizada uma grande quantidade de recursos computacionais, principalmente recursos gráficos, pois, com todos os objetos do cenário já renderizados em cena não são necessárias novas Draw Calls para objetos dessa natureza. E a única interação que esses objetos têm com o motor de física e a Unity em si, são movimentações ocasionais, como demonstrado na explicação anterior, consumindo poucas chamadas de desenho, menos do que seriam necessárias utilizando outro método para esta função, e utilizando um poder de processamento da CPU consideravelmente menor.

Durante a pesquisa e o desenvolvimento do jogo, foi pensada esta como a melhor forma para a otimização do cenário nesse caso específico de jogo e temática.

O próximo grande aspecto referente ao cenário do jogo a ser desenvolvido foram os dificultadores. Existem três tipos de dificultadores, que são o tronco, o qual o jogador precisa

desviar, a lápide, a qual o jogador precisa pular, e a placa de madeira, a qual é preciso deslizar por baixo para evitá-la. Cada dificultador instanciado na cena ocupará um determinado espaço e quantidade de memória, além de utilizar *Draw Calls* para ser renderizado na cena em tempo de execução.

E esse aspecto da implementação, se não for planejado visando a performance, no caso do desenvolvimento *mobile*, pode causar um gargalo enorme de consumo de memória e recursos gráficos. A solução pensada para esse ponto do desenvolvimento foi a aplicação de um conceito chamado o *Object Pooling*.

O *Object Pooling* consiste em criar um *script* para gerenciar objetos que serão ativados e desativados diversas vezes em cena. Esse tipo de ação é bastante custosa em termos de recursos computacionais, então, criar um *script* que gerenciará isso e o fará da maneira mais eficiente possível aumentará consideravelmente a performance da aplicação e evitará que futuros gargalos de performance ocorram na área a qual o script gerencia.

Fazendo o *Object Pooling* dos obstáculos, é possível gerenciar esses objetos e utiliza-los de uma forma que consuma uma quantidade menor de memória e realize menos *Draw Calls*. Essa técnica foi implementada da seguinte forma neste projeto.

Os obstáculos foram divididos em partes menores chamadas peças, foi criada uma classe *"Piece"* para representar a parte gráfica destas peças. A figura 6 mostra a estrutura da classe:

Figura 6: Representação da estrutura da classe "*Piece*"

```
1   using UnityEngine;
2
3   public enum PieceType
4   {
5       none = -1,
6       ramp = 0,
7       longblock = 1,
8       jump = 2,
9       slide = 3
10  }
11
12  public class Piece : MonoBehaviour {
13
14      public PieceType type;
15      public int visualIndex;
16  }
17
```

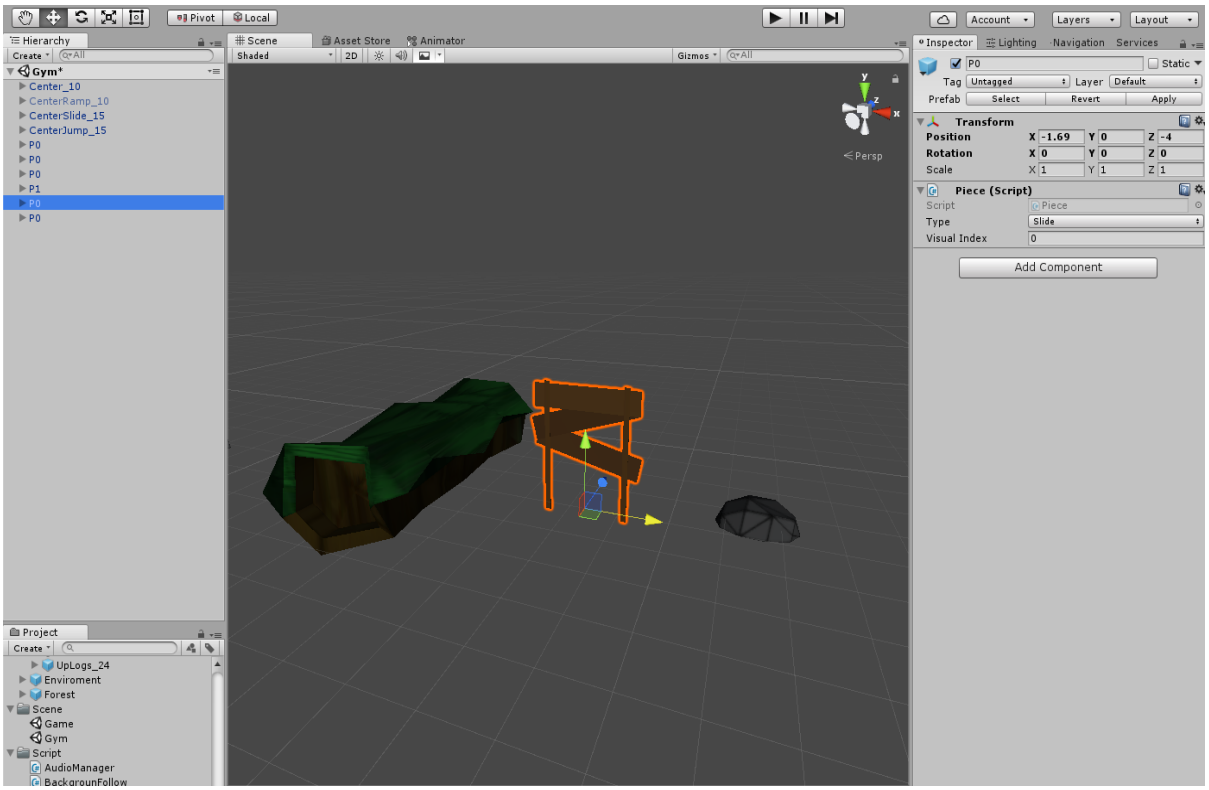
Fonte: Elaborada pelos autores

Esta classe é composta por um *enum PieceType* fora da classe, que é um tipo de variável em C#, que neste caso utilizamos apenas para definir o tipo da peça em que o *script* está ligado, sendo estes tipos nenhum, rampa, bloco longo, pulo, e deslizar, representados por *none*, *ramp*, *longblock*, *jump* e *slide* respectivamente.

No corpo da classe em si há apenas duas variáveis, sendo elas uma variável pública do tipo *PieceType* para representar o tipo da peça, e uma variável pública do tipo *int* chamada *visualIndex* para indicar o componente gráfico referente àquela peça.

A figura 7 mostra três peças, com o *script* já ligado a elas, para fins ilustrativos. São demonstradas uma peça do tipo bloco, uma peça do tipo pulo e uma peça do tipo deslizar.

Figura 7: Exemplos de objetos os quais tem o *script* "*Piece*" ligado a eles



Fonte: Elaborada pelos autores

A parte funcional que trata a colisão do jogador com as peças é gerenciada por uma classe chamada *"PieceSpawner"* apresentada na figura 8.

Figura 8: Representação do *script* "*PieceSpawner*"

```
3   public class PieceSpawner : MonoBehaviour {
4
5       public PieceType type;
6       private Piece currentPiece;
7
8       public void Spawn()
9       {
10          int amtObj = 0;
11          switch (type)
12          {
13              case PieceType.jump:
14                  amtObj = LevelManager.Instance.jumps.Count;
15                  break;
16
17              case PieceType.slide:
18                  amtObj = LevelManager.Instance.slides.Count;
19                  break;
20
21              case PieceType.longblock:
22                  amtObj = LevelManager.Instance.longblocks.Count;
23                  break;
24
25              case PieceType.ramp:
26                  amtObj = LevelManager.Instance.ramps.Count;
27                  break;
28          }
29
30
31          //Pegar uma Peça aleatoria da Object Pool
32          currentPiece = LevelManager.Instance.GetPiece(type, Random.Range(0, amtObj));
33          currentPiece.gameObject.SetActive(true);
34          currentPiece.transform.SetParent(transform, false);
35      }
36
37      public void Despawn()
38      {
39          currentPiece.gameObject.SetActive(false);
40
41      }
42  }
```

Fonte: Elaborada pelos autores

Ela contém duas variáveis, sendo elas uma *PieceType* pública para armazenar o tipo da peça que será instanciada, e uma variável do tipo *Piece* privada chamada *currentPiece*, que manterá a referência para a peça sendo instanciada no momento.

Está classe é responsável por instanciar os colisores das peças, utilizando o *PieceType* que será passado para a mesma pela classe *"Segment"*, classe esta que será abordada após esta explicação. Utilizando o *PieceType* recebido a classe *"PieceSpawner"* através do método *Spawn()*, testa esse *PieceType* através de um *switch* e dependendo do

resultado referente ao tipo da peça, é executado no *case* um comando para receber da classe *"LevelManager"* a quantidade de peças desse tipo presentes na *Pool*.

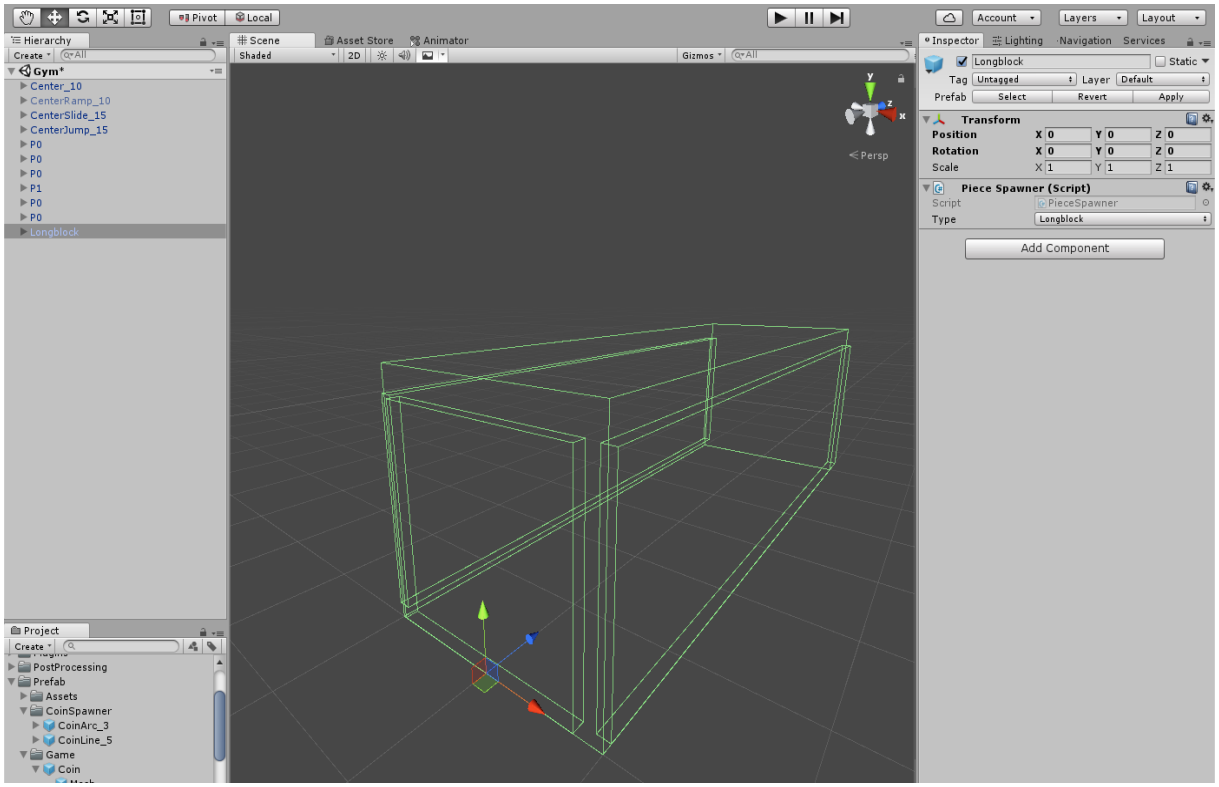
Esse numero é armazenado em uma variável temporária do tipo *int* chamada *amtObj*. Finalizado esse *switch*, é atribuído a variável *currentPiece*, o valor do retorno do método *GetPiece()* da classe *"LevelManager"*, método esse que será explicado quando está classe estiver sendo analisada. É passado para este método o tipo da peça sendo instanciada, e um numero aleatório gerado entre zero e o numero contido na variável *amtObj*, que é a quantidade de objetos do tipo da peça sendo instanciada, que estão presentes na *Pool.*

As próximas duas linhas de comando ativam esse objeto em cena, e tornam o objeto como objeto filho do segmento o qual ele está sendo instanciado. Está são as funções do método *Spawn()* da classe *"PieceSpawner"*.

O outro método desta classe é o *Despawn()* este método apenas desativa o *GameObject* da peça à qual ele está sendo invocado sobre, é um método de desativação das peças para quando as mesmas não tem mais uso em cena.

A figura 9 demonstra um objeto do tipo *"PieceSpawner"* representado por uma peça do tipo bloco longo.

Figura 9: Representação de um objeto do tipo "*PieceSpawner*"



Fonte: Elaborada pelos autores

As peças da classe *Piece* são agrupadas em um conjunto maior chamado de segmento. Estes segmentos serão blocos padrões com varias peças ligadas a ele. Esses segmentos serão como blocos de mapa. Por exemplo, um segmento pode conter uma rampa e um bloco longo, e no decorrer do jogo esse segmento pode reaparecer na *Pool* de objetos instanciados em cena, e ele conterá a mesma rampa e bloco longo, só que em um ponto diferente do jogo, dando a sensação de serem obstáculos totalmente novos, e não os mesmos utilizados a algum tempo atrás.

Para gerenciar esses segmentos foi criada a classe *"Segment"* apresentada a seguir na figura 10:

Figura 10: Representação da classe "*Segment*"

```
1  using UnityEngine;
2
3  public class Segment : MonoBehaviour {
4
5      public int SegId { set; get; }
6      public bool transition;
7
8      public int lenght;
9      public int beginY1, beginY2, beginY3;
10     public int endY1, endY2, endY3;
11
12     private PieceSpawner[] pieces;
13
14     private void Awake()
15     {
16         pieces = gameObject.GetComponentsInChildren<PieceSpawner>();
17     }
18
19     public void Spawn()
20     {
21         gameObject.SetActive(true);
22
23         for (int i = 0; i < pieces.Length; i++)
24             pieces[i].Spawn();
25     }
26
27     public void Despawn()
28     {
29         gameObject.SetActive(false);
30         for (int i = 0; i < pieces.Length; i++)
31             pieces[i].Despawn();
32     }
33 }
34
```

Fonte: Elaborada pelos autores

A classe "*Segment*" apresentada na figura 10 possui dez variáveis necessárias para sua execução, sendo elas *"SegId",* uma propriedade pública do tipo *int* que armazena o identificador único de cada segmento. "*transition",* uma booleana pública que demarca um segmento como sendo segmento de transição ou não, *"lenght"* uma variável pública do tipo *int* que armazena o comprimento do segmento no qual o *script* está ligado.

Já o grupo de variáveis do tipo *int, beginY1, beginY2* e *beginY3* armazenam em seus respectivos valores se a faixa um, dois ou três do segmento começa no chão, ou em cima de algum obstáculo. O jogo é dividido em três faixas, as quais o jogador pode se movimentar livremente sobre. Esses valores para saber se o segmento começa no chão ou terreno elevado são necessários para o algoritmo conectar corretamente os próximos segmentos sendo gerados e instanciados. Conectando segmentos que terminam no chão com segmentos que iniciam no chão, e segmentos que terminam em terreno elevado com segmentos que se iniciam em terrenos elevados.

O grupo de variáveis *endY1, endY2* e *endY3* tem a mesma função das anteriores, só que ao invés de armazenar os valores que o segmento começa, elas armazenam os valores em que o segmento termina.

A última variável é uma lista privada de objetos do tipo *PieceSpawner* chamada *piece.* Esta é a lista que armazena a referência das peças que compõe o segmento em questão.

O primeiro método da classe *"Segment"* é o método *Awake(),* um método padrão da Unity que é executado assim a instância a qual o *script* está ligado entra em cena. O código presente neste método serve para "inicializar" os segmentos. Ele atribui à lista *pieces* os componentes *PieceSpawner* que busca nos objetos filhos do *GameObject* ao qual o *script* está ligado. Ao ter essas referências, o método *Spawn()* da classe *PieceSpawner* pode ser invocado para cada objeto deste segmento.

O próximo método é o método *Spawn(),* porém este método é diferente do *Spawn()* da classe *PieceSpawner*. Este método ativa o objeto no qual o *script "Segment"* está ligado, e executa um laço de repetição *for* indo de zero até o tamanho da lista *pieces,* que é a quantidade de peças deste segmento em questão. E pra cada *PieceSpawner* dentro de *Piece,* é executado o método *Spawn()* da classe *PieceSpawner* desta vez, instanciando assim, para cada peça, seus respectivos colisores.

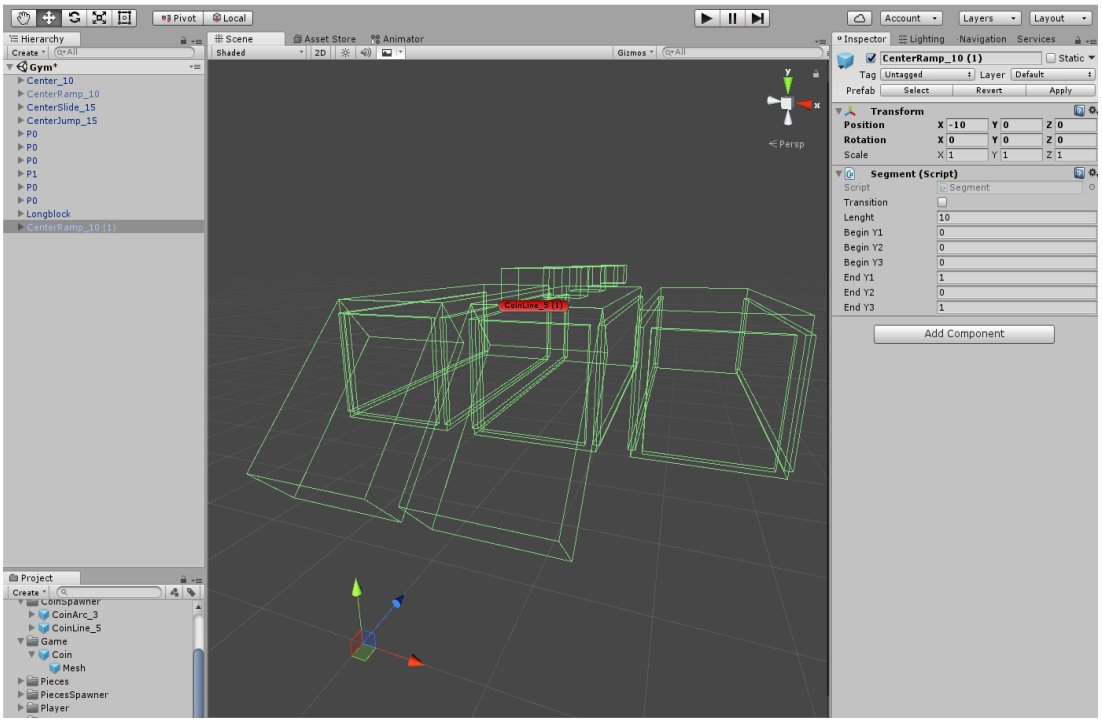
Ambos os métodos possuem um mesmo nome por causa da similaridade e ligação em suas funções. O método *Spawn()* da classe, *"PieceSpawner"* nunca será chamado em outro lugar a não ser pelo método *Spawn()* da classe *"Segment",* que por sua vez, só é chamado pelo método *SpawnSegment()* da classe *"LevelManager"*, desta forma não havendo

confusões com relação a nomes de métodos na hora de invocá-los.

O método seguinte da classe é o *Despawn(),* e ele tem exatamente a mesma função do método *Spawn(),* só que ao invés de ativar as peças, ele desativa as mesmas. A lógica é exatamente a mesma nos dois métodos, a única diferença é que um ativa os objetos e o outro desativa.

A figura 11 mostra o exemplo de um segmento. Apenas as colisões estão instanciadas no segmento pois as partes gráficas das peças do segmento são instanciadas em tempo de execução no jogo.

Figura 11: Exemplo de Segmento em cena, sem a arte



Fonte: Elaborada pelos autores

A mecânica de *Object Pooling* desta pesquisa foi desenvolvida em cima destes segmentos, são eles que fazem parte da *Pool* para que ocorra a diminuição de consumo de memória, *Draw Calls* e poder de processamento.

Está mecânica foi implementada da seguinte forma: são instanciados dez segmentos iniciais aleatórios dentre os segmentos disponíveis para o jogo, e conforme o jogador vai avançando pelo jogo, e deixando estes segmentos para trás, eles vão sendo desativados e novos segmentos gerados aleatoriamente à frente na cena. Para um novo segmento ser instanciado ele precisa ser do tipo que é aleatoriamente definido, precisa encaixar o seu início com o segmento anterior a ele, no quesito de terminar e começar na mesma altura sendo ela

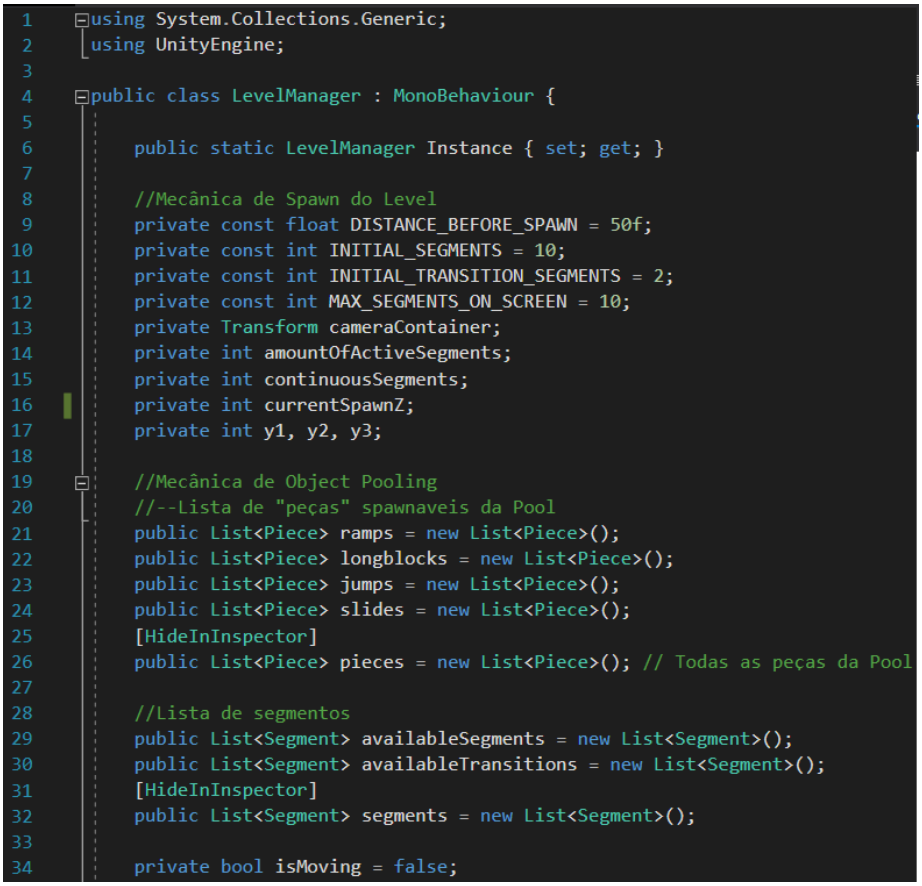
no chão ou em cima de algum obstáculo, além de não poder já estar em uso em outra parte da cena.

Se todas estas condições forem cumpridas, o segmento é instanciado à frente, se não, um novo segmento precisa ser gerado para suprir as necessidades de instância que a peça seguinte requer. Logo, o novo segmento é gerado e posicionado. Os que já foram “utilizados” pelo jogador ficam em cena, porém são desativados assim que o jogador passa por eles. O fato de eles serem apenas desativados, faz com que possam ser reutilizados em futuras instâncias ao decorrer do jogo, utilizando menos chamadas de desenho e processamento, além de já estarem ocupando um espaço na memória, facilitando o trabalho de gerenciamento de memória e do *GarbageCollector*.

Isso faz com que o *Object Pooling* seja o meio mais eficientes de gerenciar, em termos de recursos, objetos que aparecerão em cena repetidas vezes e em grande quantidade, fazendo com que os recursos sejam melhor aproveitados utilizando essa técnica de desenvolvimento.

A seguir será abordado o funcionamento e implementação desta técnica, que é parte crucial no desenvolvimento e otimização da aplicação desta pesquisa. A figura 12 apresenta a parte das variáveis do código da classe *“LevelManager”*, responsável pela implementação da mecânica de *Pooling*:

Figura 12: Variáveis da classe "*LevelManager*"



```
using System.Collections.Generic;
using UnityEngine;

public class LevelManager : MonoBehaviour {

    public static LevelManager Instance { set; get; }

    //Mecânica de Spawn do Level
    private const float DISTANCE_BEFORE_SPAWN = 50f;
    private const int INITIAL_SEGMENTS = 10;
    private const int INITIAL_TRANSITION_SEGMENTS = 2;
    private const int MAX_SEGMENTS_ON_SCREEN = 10;
    private Transform cameraContainer;
    private int amountOfActiveSegments;
    private int continuousSegments;
    private int currentSpawnZ;
    private int y1, y2, y3;

    //Mecânica de Object Pooling
    //--Lista de "peças" spawnaveis da Pool
    public List<Piece> ramps = new List<Piece>();
    public List<Piece> longblocks = new List<Piece>();
    public List<Piece> jumps = new List<Piece>();
    public List<Piece> slides = new List<Piece>();
    [HideInInspector]
    public List<Piece> pieces = new List<Piece>(); // Todas as peças da Pool

    //Lista de segmentos
    public List<Segment> availableSegments = new List<Segment>();
    public List<Segment> availableTransitions = new List<Segment>();
    [HideInInspector]
    public List<Segment> segments = new List<Segment>();

    private bool isMoving = false;
```

Fonte: Elaborada pelos autores

A classe abordada na figura 12 possui uma quantidade significativamente maior de variáveis que as anteriores, porque é ela quem implementa e manuseia toda a lógica central do sistema de *Pool*, tendo as outras classes já abordadas como classes de suporte a essa.

A primeira variável da classe *"LevelManager"* é uma propriedade pública e estática que guarda uma referência a ela mesma, para que outras classes possam acessá-la através dessa referência, e para que todas as classes que tentem acessar seus dados acessem a mesma instância da classe, para não haver discrepâncias nos dados e valores recebidos da mesma.

O primeiro bloco, são as variáveis responsáveis pelos dados e valores referentes à instância dos segmentos na cena. A primeira delas é uma constante privada do tipo *float* *"DISTANCE_BEFORE_SPAWN"*, que é o valor da distância que o jogador deve ultrapassar para instanciar um novo segmento a frente na cena. As duas próximas variáveis são constantes privadas do tipo *int* chamadas *"INITIAL_SEGMENTS"*, que é a variável que armazena a quantidade de segmentos que serão instanciados no início do jogo, e *"INITIAL_TRANSITION_SEGMENTS"* que armazena a quantidade de segmentos de

transição que serão instanciados no inicio do jogo. Segmentos de transição são trechos da fase onde não haverão obstáculos, serão áreas de "descanso" para o jogador aliviar de trechos que requerem bastante concentração.

E a ultima constante é do tipo *int* também, chamada *"MAX_SEGMENTS_ON_SCREEN"* que armazena a quantidade máxima de segmentos que poderão ser instanciados na tela de uma vez. Essa variável armazena um valor muito importante referente à utilização de recursos gráficos, e o valor dela precisa ser pensado para estabelecer um equilíbrio entre utilização de recursos e a imersão do jogador.

Por exemplo, se existem muito segmentos instanciados na tela, será utilizada uma quantidade muito alta de recursos gráficos para instanciar e renderizar todas elas, causando gargalos de performance e queda de *frames.* Agora, se existem muito poucos segmentos na tela, o jogador conseguirá ver o horizonte e os futuros segmentos sendo instanciados, fazendo com que isso prejudique a experiência de imersão e continuidade do jogador.

A próxima variável é do tipo *Transform* chamada *cameraContainer*. Ela guarda uma referência aos valores de posição da câmera do jogo, e através destes valores são feitos os cálculos para saber se o jogador já passou a distância pré-estabelecida para instanciar um novo segmento.

As próximas quatro variáveis são do tipo *int*, sendo elas *"amountOfActiveSegmets"*, que armazena a quantidade de segmentos ativos na tela no momento, *"continuousSegments"*, que é uma variável usada para gerar um numero de chance aleatória para decidir se será instanciada uma transição ou não, *"currentSpawnZ"*, variável que armazena o valor atual do ponteiro do eixo Z. Esse ponteiro armazena uma referência à localização onde o próximo segmento deverá ser instanciado para dar continuidade ao fluxo da cena. E as últimas delas são um conjunto de variáveis chamadas de *y1, y2* e *y3* que serão usadas para armazenar os valores de *endY1, endY2* e *endY3* dos segmentos que serão instanciados.

A próxima sequência de variáveis são as responsáveis pelo núcleo da mecânica de *Object Pooling*. São um total de oito listas. As quatro primeiras listas são do tipo *Piece*, seus nomes são *ramps, longblocks, jumps e slides*, e essas quatro listas armazenam as referências a todos os obstáculos que estarão presentes na *Pool*, todos os blocos, rampas, pedras e placas. E a quinta lista, do tipo *Piece*, é chamada de *pieces* e armazena nela todas as peças das quatro listas anteriores, é uma lista com todos os objetos disponíveis da *Pool*.

As três ultimas listas são do tipo *"Segment"* e são elas *"availableSegments",* que armazena as referências de todos os segmentos que estão disponíveis para serem instanciados no momento, *"availableTransitions"*, que assim como a anterior, armazena uma referência à

todos os segmentos que estão disponíveis para serem instanciados no momento, porém, esta armazena apenas os segmentos considerados de transição. E a última lista, chamada *"segments"*, armazena todos os segmentos das duas listas nela.

A ultima variável da classe é uma booleana privada chamada *"isMoving"* que diz ao *"LevelManager"* se o jogador está se movimentando ou não. Está variável será utilizada para iniciar e parar as instâncias de segmentos na cena.

A próxima parte desta classe sendo abordada são os métodos e o funcionamento de cada um deles. A figura 13 mostra os métodos "*Awake()", "Start()"* e *"Update()"*, e seus respectivos códigos.

Figura 13: Métodos "*Awake()", "Start()"* e *"Update()"* da classe "*LevelManager*"

```
36      private void Awake()
37      {
38          Instance = this;
39          cameraContainer = Camera.main.transform;
40          currentSpawnZ = 0;
41      }
42
43      private void Start()
44      {
45          for (int i = 0; i < INITIAL_SEGMENTS; i++)
46              if (i < INITIAL_TRANSITION_SEGMENTS)
47                  SpawnTransition();
48              else
49                  GenerateSegment();
50      }
51
52      private void Update()
53      {
54          //Spawna segmentos automaticamente
55          if (currentSpawnZ - cameraContainer.position.z < DISTANCE_BEFORE_SPAWN)
56              GenerateSegment();
57
58          //Despawna segmentos automaticamente
59          if(amountOfActiveSegments >= MAX_SEGMENTS_ON_SCREEN)
60          {
61              segments[amountOfActiveSegments - 1].Despawn();
62              amountOfActiveSegments--;
63          }
64      }
```

Fonte: Elaborada pelos autores

O primeiro método da classe, *"Awake()",* inicializa algumas variáveis. Atribuindo "*this"* à propriedade "*Instance"* faz com que a instancia da classe atual seja armazenada nessa

propriedade para poder ser utilizada por outras classes, e para que todas utilizem a mesma instância. A variável *"cameraContainer"* recebe a referência da câmera presente na cena, e *"currentSpawnZ"* é inicializada com zero.

O próximo método é o "*Start()*", ele é responsável por instanciar todos os obstáculos no início do jogo. Um laço de repetição *for* que vai de zero ao valor da variável *"INITIAL_SEGMENTS"* é executado no início do método. A cada laço deste *for* é testado se a variável de controle de laço *i* é menor que o valor da variável "*INITIAL_TRANSITION_SEGMENTS"*, se sim, é instanciada uma transição chamando o método *"SpawnTransition()"*, se não, é instanciado um segmento normal chamando o método *"GenerateSegment()"*. Seguindo está lógica haverão sempre no inicio da fase dois segmentos de transição e oito segmentos normais.

Continuando o fluxo da classe vem o método *"Update()"*. Este é o método responsável por fazer as chamadas para instanciar e desativar automaticamente os segmentos no decorrer do jogo. Ele começa com um teste condicional *if* verificando se o valor da variável *"currentSpawnZ"* menos o valor da posição em Z da variável *"cameraContainer"* é menor que o valor da variável *"DISTANCE_BEFORE_SPAWN"*. Se sim, é hora de instanciar um novo segmento, então é chamado o método *"GenerateSegment()"*. Este é o trecho responsável por instanciar segmentos automaticamente.

Em seguida vem o trecho responsável por desativar os segmentos deixados para trás. Ele começa com um teste condicional *if* para saber se o numero de segmentos ativos na tela no momento é maior ou igual ao valor da variável *"MAX_SEGMENTS_ON_SCREEN"*, que é o valor máximo de segmentos na tela permitidos.

Se sim, é selecionado o ultimo segmento da lista *"segments"* e executado o seu método *"Despawn()"* desativando-o da cena. É então subtraído 1 do valor da variável *"amountOfActiveSegments",* variável que mantém o controle dos segmentos ativos no momento. A figura 14 mostra os próximo métodos da classe, que são *"GenerateSegment()"*, e "*SpawnSegment()"*.

Figura 14: Métodos “*GenerateSegment()*” e “*SpawnSegment()*” da classe “*LevelManager*”

```
66  private void GenerateSegment()
67  {
68      SpawnSegment();
69
70      if(Random.Range(0f, 1f) < (continuousSegments * 0.25f))
71      {
72          //Spawnar segmento de transição
73          continuousSegments = 0;
74          SpawnTransition();
75      }
76      else
77      {
78          continuousSegments++;
79      }
80
81  }
82
83  private void SpawnSegment()
84  {
85      List<Segment> possibleSeg = availableSegments.
86          FindAll(x => x.beginY1 == y1 || x.beginY2 == y2 || x.beginY3 == y3);
87      int id = Random.Range(0, possibleSeg.Count);
88
89      Segment s = GetSegment(id, false);
90
91      y1 = s.endY1;
92      y2 = s.endY2;
93      y3 = s.endY3;
94
95      s.transform.SetParent(transform);
96      s.transform.localPosition = Vector3.forward * currentSpawnZ;
97
98      currentSpawnZ += s.lenght;
99      amountOfActiveSegments++;
100     s.Spawn();
101 }
```

Fonte: Elaborada pelos autores

O método *“GenerateSegment()”* começa invocando o método *“SpawnSegment()”*, e em seguida, realiza um teste condicional *if* para definir se será instanciada uma transição após este segmento, ou não. É gerado um numero aleatório entre zero e um, com a função *“Random.Range(0f, 1f)”*, e depois testado se esse número é menor que o valor da variável *“continuousSegments”* multiplicada por 0.25, o que significa um total de 25% de chance de instanciar uma transição. Se essa sentença for verdadeira, então a variável *“continuousSegments”* recebe zero, e é chamado o método *“SpawnTransition()”* para instanciar uma transição. Se a sentença for falsa, a variável *“continuouSegments”* é incrementada em 1, e a execução do código continua normalmente.

O método *“SpawnSegment()”* é o método responsável por de fato instanciar os segmentos. Ele começa criando uma variável temporária do tipo lista chamada *“possibleSeg”* e atribuindo à ela o valor do método *“FindAll()”* executado na lista de segmentos disponíveis *“availableSegments”*. O método *“FindAll()”* vai procurar na lista um segmento que, uma das variáveis *“BeginY1”, “BeginY2”* ou *“BeginY3”* deste segmento seja igual a variável de classe “y1”, “y2” ou “y3” respectivamente.

Se alguma delas for igual, significa que este possível segmento tem um caminho que termina onde o possível segmento seguinte se esse inicia, logo, este possível segmento em

questão está apto a ser o próximo instanciado. Todos os segmentos aptos a isso são armazenados nessa lista *"possibleSeg"*.

Em seguida é criada uma variável temporária do tipo *int* chamada *"id"*, que será a responsável por selecionar aleatoriamente o segmento entre os possíveis candidatos a serem os próximos. É armazenado nesta variável um numero aleatório entre zero e o numero de segmentos na lista *"possibleSeg"*, esse número será o *"id"* do próximo segmento.

Definido o *"id"*, na próxima linha de comando é criada uma variável temporária do tipo *"Segment"* chamada *"s"*, ela será responsável por armazenar temporariamente uma referência ao segmento sendo instanciado atualmente. Está variável recebe como valor o resultado do método *"GetSegment()"*, onde é passado para ele o *"id"* gerado, e uma booleana identificando se é um segmento de transição ou não.

Em seguida, é armazenada em *"y1", "y2"* e *"y3"* os valores da variável *"endY1", "endY2"* e *"endY3"* respectivamente, para manter o controle de onde o segmento anteriormente instanciado termina para a próxima execução deste método, auxiliando assim a busca do método *"FindAll()"*.

Agora que o próximo segmento da fase está selecionado e pronto para ser colocado em cena, as próximas linhas de código tratam dessa parte. Primeiro a variável *"s"* é colocada como filha do objeto onde o *script* está ligado. Em seguida é definida sua posição colocando-a na distância ditada pelo valor da variável *"currentSpawnZ"*. Essa variável em seguida é incrementada com a soma do tamanho do segmento *"s"* em seu valor, para que o próximo segmento seja instanciado a frente de *"s"*.

A variável *"amountOfActiveSegments"* recebe o incremento de 1, já que o segmento instanciado será automaticamente um segmento ativo. E por fim, na ultima linha, é invocado o método *"Spawn()"* do segmento *"s"*, para coloca-lo em cena totalmente funcional e pronto para quando o jogador passar pelo mesmo.

A figura 15 demonstra os dois métodos seguintes da classe, o método *"SpawnTransition()"*, e o método *"GetSegment()"*.

Figura 15: Métodos *"SpawnTransition()"* e *"GetSegment()"*

```
103   private void SpawnTransition()
104   {
105       List<Segment> possibleTransition = availableTransitions.
106           FindAll(x => x.beginY1 == y1 || x.beginY2 == y2 || x.beginY3 == y3);
107       int id = Random.Range(0, possibleTransition.Count);
108
109       Segment s = GetSegment(id, true);
110
111       y1 = s.endY1;
112       y2 = s.endY2;
113       y3 = s.endY3;
114
115       s.transform.SetParent(transform);
116       s.transform.localPosition = Vector3.forward * currentSpawnZ;
117
118       currentSpawnZ += s.lenght;
119       amountOfActiveSegments++;
120       s.Spawn();
121   }
122
123   public Segment GetSegment(int id, bool transition)
124   {
125       Segment s = null;
126       s = segments.Find(x => x.SegId == id && x.transition == transition && !x.gameObject.activeSelf);
127
128       if(s == null)
129       {
130           GameObject go = Instantiate((transition) ?
131               availableTransitions[id].gameObject : availableSegments[id].gameObject) as GameObject;
132           s = go.GetComponent<Segment>();
133
134           s.SegId = id;
135           s.transition = transition;
136
137           segments.Insert(0, s);
138       }
139       else
140       {
141           segments.Remove(s);
142           segments.Insert(0, s);
143       }
144
145       return s;
146
147   }
```

Fonte: Elaborada pelos autores

O método *"SpawnTransition()"* faz exatamente a mesma coisa que o método *"SpawnSegment()"* porém ele instancia segmentos de transição ao invés de segmentos normais. Por sua funcionalidade ser exatamente a mesma, além de as mesmas linhas de código não há necessidade da explicação em detalhes deste método. A única diferença notável entre os dois é durante a criação da variável que representa o segmento, *"s"*, onde o método *"GetSegment()"* recebe a booleana *true* ao invés de *false.*

O método seguinte é o *"GetSegment()"*, ele é responsável por pegar um segmento desativado presente na *Pool* e reativa-lo novamente como um novo segmento. Caso não haja um segmento com as características necessárias ao novo segmento sendo instanciado, é necessário então instanciar um novo, buscando-o na lista de segmentos disponíveis.

Este método é iniciado criando uma variável temporária do tipo *"Segment"* chamada *"s"*, semelhante à dos métodos anteriores, e a inicializando com o valor nulo. O próximo

passo é executar nesta variável criada o método *"Find()"* na lista de classe *"segments"*, que é a lista de todos os segmentos presentes. O método irá buscar na lista por um segmento que possua o mesmo *"id"*, seja ou não uma transição estando de acordo com o valor da booleana *"transition"* passada na invocação do método *"GetSegment()"*, e por último, esse segmento só será atribuído a variável *"s"* se não estiver ativo em cena no momento da execução deste método.

Se *"s"* tiver um valor nulo, o que significa que o método *"Find()"* não achou nenhum segmento com as características necessárias, então é instanciado um novo segmento a partir dos disponíveis presentes na lista *"availableSegments"*. As linhas seguintes do bloco *true* do condicional *if* instanciam esse novo segmento e o adicionam na lista de *"segments"* para manter controle e referência de todos os novos segmentos instanciados.

Agora, se o valor de *"s"* não é nulo, significa que o método *"Find()"* encontrou um segmento disponível desativado com as características necessárias para a próxima instância. Logo, o bloco *else* é executado e o segmento em questão removido do fim da lista *"segments"* e colocado no inicio da mesma, pelo fato de agora ser o segmento mais novo ativo. Por fim, o método retorna o novo segmento instanciado ou encontrado na lista de segmentos disponíveis.

O ultimo método da classe *"LevelManager"* é o método *"GetPiece()"*, apresentado na figura 16.

Figura 16: Método "*GetPiece()*" da classe "*LevelManager*"

```
148     public Piece GetPiece(PieceType pt, int visualIndex)
149     {
150         Piece p = pieces.Find(x => x.type == pt &&
151             x.visualIndex == visualIndex && !x.gameObject.activeSelf);
152 
153         if(p == null)
154         {
155             GameObject go = null;
156             if (pt == PieceType.ramp)
157                 go = ramps[visualIndex].gameObject;
158             else if (pt == PieceType.longblock)
159                 go = longblocks[visualIndex].gameObject;
160             else if (pt == PieceType.jump)
161                 go = jumps[visualIndex].gameObject;
162             else if (pt == PieceType.slide)
163                 go = slides[visualIndex].gameObject;
164 
165             go = Instantiate(go);
166             p = go.GetComponent<Piece>();
167             pieces.Add(p);
168         }
169 
170         return p;
171     }
172 }
```

Fonte: Elaborada pelos autores

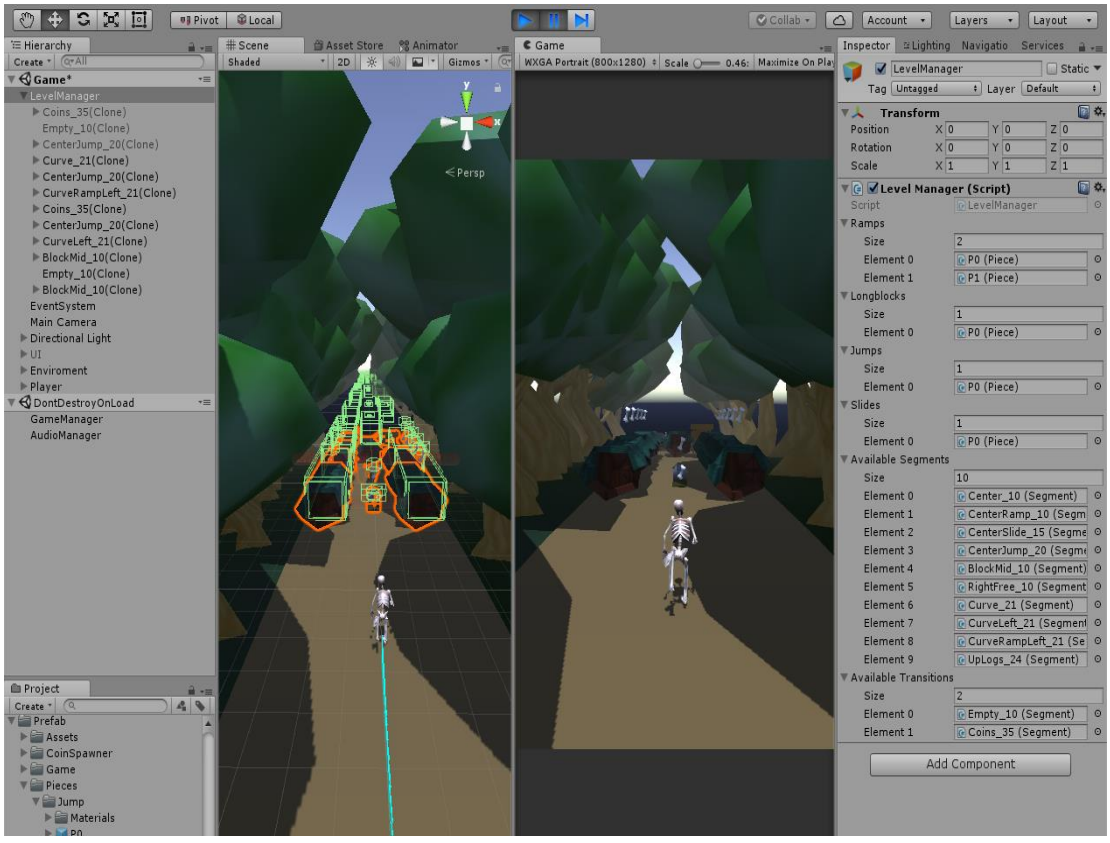
O funcionamento deste método é bem semelhante ao do método anterior. O método é iniciado com a criação de uma variável temporária do tipo *"Piece"* chamada *"p"*, e é atribuído a ela o valor de retorno do método *"Find()"*, executado na lista *"pieces"*, que é a lista de todas as peças disponíveis da *Object Pool*. O método irá buscar na lista uma peça que seja do mesmo tipo passado para o *"GetPiece()"* em sua invocação, tenha o mesmo *"visualIndex"* e não esteja ativa em cena no momento.

Se o valor de *"p"* for nulo, significa que não há uma peça, no momento, na lista que cumpra os requisitos, logo, uma nova peça precisará ser instanciada. É executada então uma cadeia de *if else* para definir o tipo da peça, e a partir deste tipo, buscar a peça na lista apropriada, usando como índice para a lista o *"visualIndex"* passado para a o método em sua invocação.

Definido seu tipo, a peça é instanciada e adicionada à lista *"pieces"* para a manutenção do controle e referência para futuras instâncias. O método ao final retorna a peça criada ou buscada.

A figura 17 demonstra o *"LevelManager"* em ação na execução do jogo.

Figura 17: Classe "*LevelManager*" em execução



Fonte: Elaborada pelos autores

Na tela à esquerda está a representação do jogo na cena do editor do Unity, mostrando em traços verdes os colisores dos segmentos de obstáculos gerenciados pelo *"LevelManager"*. Já na tela à direita, está a representação de como ficará a tela no dispositivo Android, esta é a visão que o jogador terá.

Na hierarquia de objetos da cena, abaixo de "*LevelManager*" no canto superior esquerdo da tela, é possível observar o comportamento do objeto *"LevelManager"* que instanciou os segmentos de obstáculos que estão logo abaixo dele, tendo doze objetos instanciados e apenas nove ativos. A quantidade de objetos ativos e desativados segue as métricas da classe *"LevelManager"* explicada anteriormente, a imagem representa a parte prática da teoria explicada.

E assim finalizada a explicação da classe *"LevelManager"*, a classe principal responsável pela mecânica de *Object Pooling*. Esse é um dos principais e mais relevantes aspectos no que tange a otimização do jogo em questão, pois, esse aspecto pode, se não for cuidadosamente desenvolvido, ser o ponto mais problemático e que possivelmente inviabilize a aplicação. Contínuas instâncias de objetos complexos como os segmentos de mapa, consomem quantidades relevantes de memória, poder de processamento e principalmente capacidade gráfica do aparelho *Mobile*.

O método de desenvolvimento e implementação desta parte da aplicação desenvolvida nesta pesquisa utiliza propriedades e funções da *Engine* de desenvolvimento, Unity3D, que são menos custosos em questão de recursos computacionais como a desativação e reativação de segmentos já utilizados. Os segmentos possuem uma "propriedade de instância única", o que significa, que uma vez já instanciados eles são guardados e não precisam ser instanciados novamente. Então a quantidade de *Draw Calls* e renderização feitas em tempo de execução diminui consideravelmente, contribuindo assim para atingir o objetivo de taxa de *frames* por segundo da otimização da aplicação.

O último grande ponto a ser discutido à respeito da otimização da performance do jogo, é um ponto crucial referente a parte de consumo de GPU do jogo, que é a texturas dos objetos. Cada objeto do jogo, seja ele o personagem, os obstáculos ou as arvores possui um arquivo de textura que dá a "cara" do objeto. Seja ele o formato e *design* do tronco, os ossos do personagem principal, e para cada objeto instanciado, é instanciado com ele uma textura, logo, quanto mais objetos instanciados em cena, mais texturas instanciadas junto.

Arquivos de texturas são bastante custosos em questão de GPU, pois geralmente são arquivos bastante detalhados e que se instanciados em grandes quantidades podem causar uma

overhead[2] no dispositivo gráfico do aparelho em que a aplicação está sendo executada.

Para resolver esse problema foi utilizado o recurso da Unity chamado *"GPU Instancing"*. Este recurso pode ser utilizado quando é necessário instanciar muitas cópias de um mesmo material em uma cena. O *"GPU Instancing"* faz isso utilizando um numero muito menor de *Draw Calls* que o normal, renderizando os objetos e agrupando-os o máximo possível, ocasionando em um ganho de performance muito relevante nesse aspecto.

Esse recurso foi utilizado em objetos como as texturas dos obstáculos do jogo, tais como as rampas, os troncos, e todas as outras peças, além de ser utilizado nos ossos que o jogador coleta ao longo da fase. Os ossos são colecionáveis do jogo que atribuem pontos ao jogador, e são instanciados uma quantidade considerável de vezes.

Cada osso tem uma chance de 50% de ser instanciado em um máximo de 15 por segmento, sendo que estão ativos ao mesmo tempo dez segmentos na cena. Considerando as chances haverão em média 75 ossos instanciados em cena, ou seja, 75 texturas e *Draw Calls* serão otimizadas utilizando o método de *"GPU Instancing"*.

Além do fato de haver pelo menos dez segmentos ativos em tela, cada um com uma média de quatro peças, e vinte e oito arvores com duas texturas cada, uma para a folha e outra para o tronco. Somando tudo isso são mais 96 texturas otimizadas em cena com esse método. Sem a utilização desta técnica haveriam 171 texturas independentes entre si instanciadas consumindo cada uma sua porção de processamento, *Draw Calls* e chamadas de renderização da GPU, o que poderia causaria uma sobrecarga na parte gráfica do aparelho.

Os conceitos e técnicas que abrangem a otimização acima destacados neste capítulo da pesquisa são os aspectos relevantes à otimização do jogo desenvolvido. Os outros pontos do desenvolvimento, com relação a mecânicas gerais do jogo não possuem impactos significativos ou relevantes o suficiente na performance da aplicação para justificar o seu detalhamento como os anteriormente descritos. Os pontos explicados neste capítulo são cruciais para o funcionamento consistente da aplicação com as métricas de otimização estabelecidas e com uma taxa constante de *frames* por segundo (FPS).

4.2 Resultados dos testes de performance

Chegando ao ponto final da pesquisa, que são os testes da aplicação finalizada em

[2] Overhead é geralmente considerado qualquer processamento ou armazenamento em excesso, seja de tempo de computação, de memória, de largura de banda ou qualquer outro recurso que seja requerido para ser utilizado ou gasto para executar uma determinada tarefa.

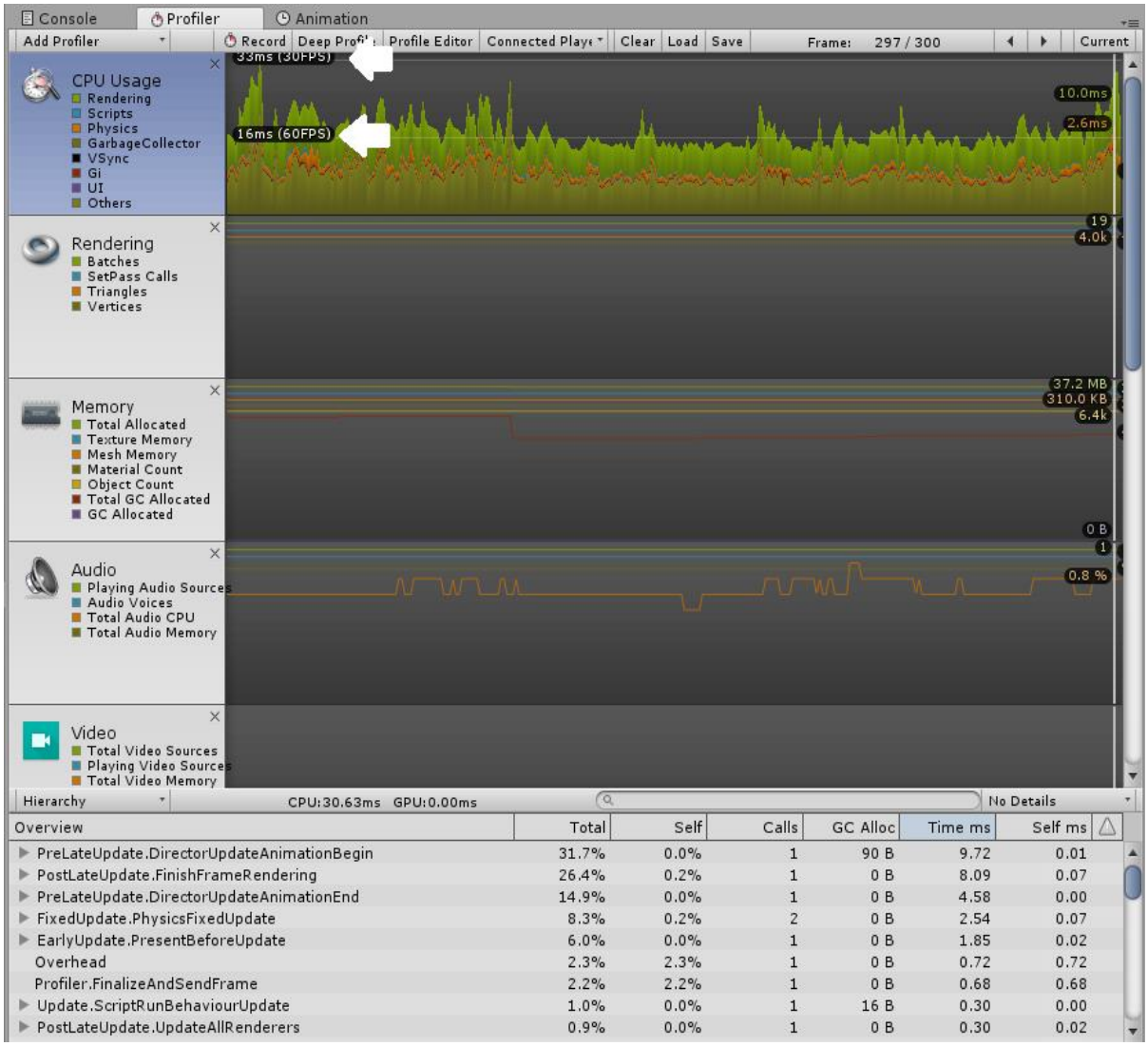
três modelos de *smartphone* diferentes, cada um com uma especificação técnica diferente e em diferentes faixas de preços, para variar e atingir o máximo de usuários possíveis com o espaço amostral de aparelhos para testes disponíveis.

Os testes foram realizados da seguinte forma: foram coletados 300 *frames* de informação do jogo, o que equivale a cinco segundos de jogo, de quatro partes diferentes do ciclo de vida da aplicação. As informações serão coletadas utilizando a ferramenta *Profiler*, da Unity, como explicado anteriormente. Serão coletadas informações da tela inicial, do início do jogo, também chamado de *"EarlyGame"*, de um tempo de jogo mediano, cerca de 45 segundos a 1 minuto de *gameplay* chamado de "*MidGame"*, e, por último, informações do jogo quando o jogador já o jogou por cinco minutos ou mais, chamado de "*LateGame"*.

A coleta de informações foi realizada dessa forma pois em cada um dos períodos citados a aplicação pode se comportar de maneira diferente com relação ao uso de recursos, logo, analisando essas quatro principais etapas do ciclo de vida da aplicação pode se ter uma noção geral do desempenho total do jogo desta pesquisa.

O primeiro teste foi realizado em um *smartphone* da marca ASUS, modelo Zenfone 5. Esse é o aparelho que tem as configurações e capacidades técnicas mais modestas dos três, por já ser um modelo relativamente antigo. A figura 18, a seguir, mostra as informações referentes a tela inicial do jogo.

Figura 18: Dados da tela inicial, *Asus Zenfone 5*



Fonte: Elaborada pelos autores

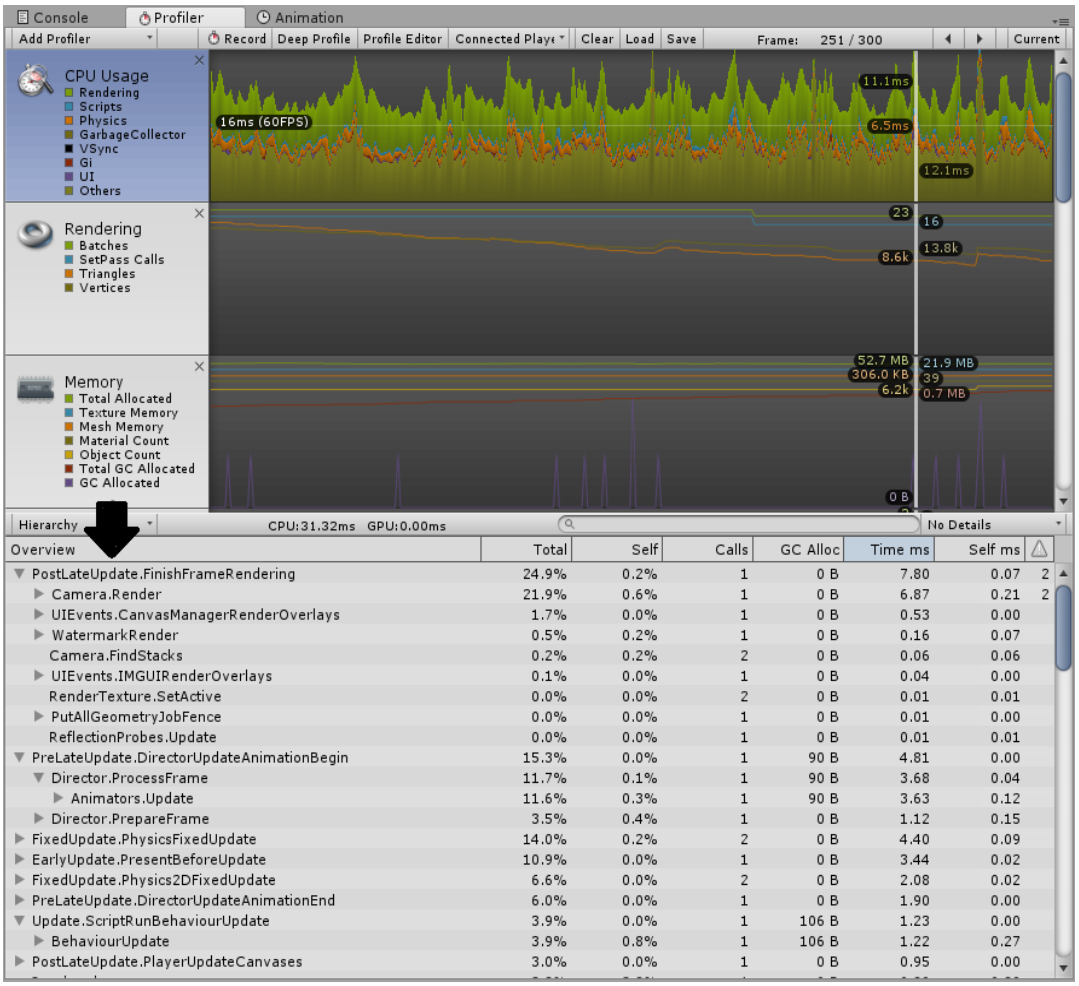
A figura 18 mostra o gráfico da performance do aparelho. As setas brancas mostram as marcações de a quantos *frames* por segundo está a aplicação. Como pode ser observado a aplicação mantem uma média constante de 45 a 60 *FPS*, quantidade considerada excelente, pois a quantidade de *FPS* considerada ótima para jogos em *smartphones* é de 20 a 30 *frames* por segundo, logo, o gráfico mostra que apesar dos recursos e funções já implementados e desenvolvidos neste jogo, o mesmo ainda possui bastante espaço e capacidade para expansão, melhorias e adição de recursos sem impacto significativo prejudicial na performance.

Esses pequenos picos de performance no início do gráfico são a aplicação carregando ainda, por se tratar da tela inicial, e os picos seguintes são pequenas atualizações do "*Animator*" da Unity, componente responsável por gerenciar as animações de interface de usuário na aplicação apresentada.

Além de durarem apenas um *frame*, o que neste caso é irrelevante para a performance e imperceptível, eles não abaixam a taxa de *frames* para menos de 30 *FPS* em nenhum momento, como demonstrado no gráfico, mantendo assim a aplicação acima do esperado até nos gargalos de performance da tela inicial.

A figura 19 mostra a performance do jogo em seu início, como explicado anteriormente.

Figura 19: Dados do "*EarlyGame*", *Asus Zenfone 5*



Fonte: Elaborada pelos autores

Como pôde ser observado, o padrão do gráfico já é bem diferente do anterior, pois ao mesmo tempo estão acontecendo ativações de objetos, instanciação dos mesmos, a mecânica de *Object Pooling*, etc. A aplicação ainda mantém uma média de 45 *FPS* durante a execução do jogo em seu início, apresentando picos de performance a cada 15 *frames* em média. Esses picos são causados, como demonstrado na figura 19 pela seta preta, pelo "*PostLateUpdate*" da Unity, essa função está gerenciando e cuidando das operações

relacionadas a renderização de texturas e objetos, sendo eles de cena ou de interface de usuário.

Logo, por se tratar da parte gráfica, diz-se que a aplicação está "*GPU Bound*" no momento, pelo fato de a parte gráfica estar em execução no momento e o processador estar em espera, ocioso, até a parte gráfica ser finalizada naquele *frame* ou *frames*. A parte gráfica dos *smartphones* modernos vem evoluindo bastante, mas, por diversos fatores, ainda é bastante inferior comparada a dos computadores, causando essa dificuldade quando o assunto é gráfico.

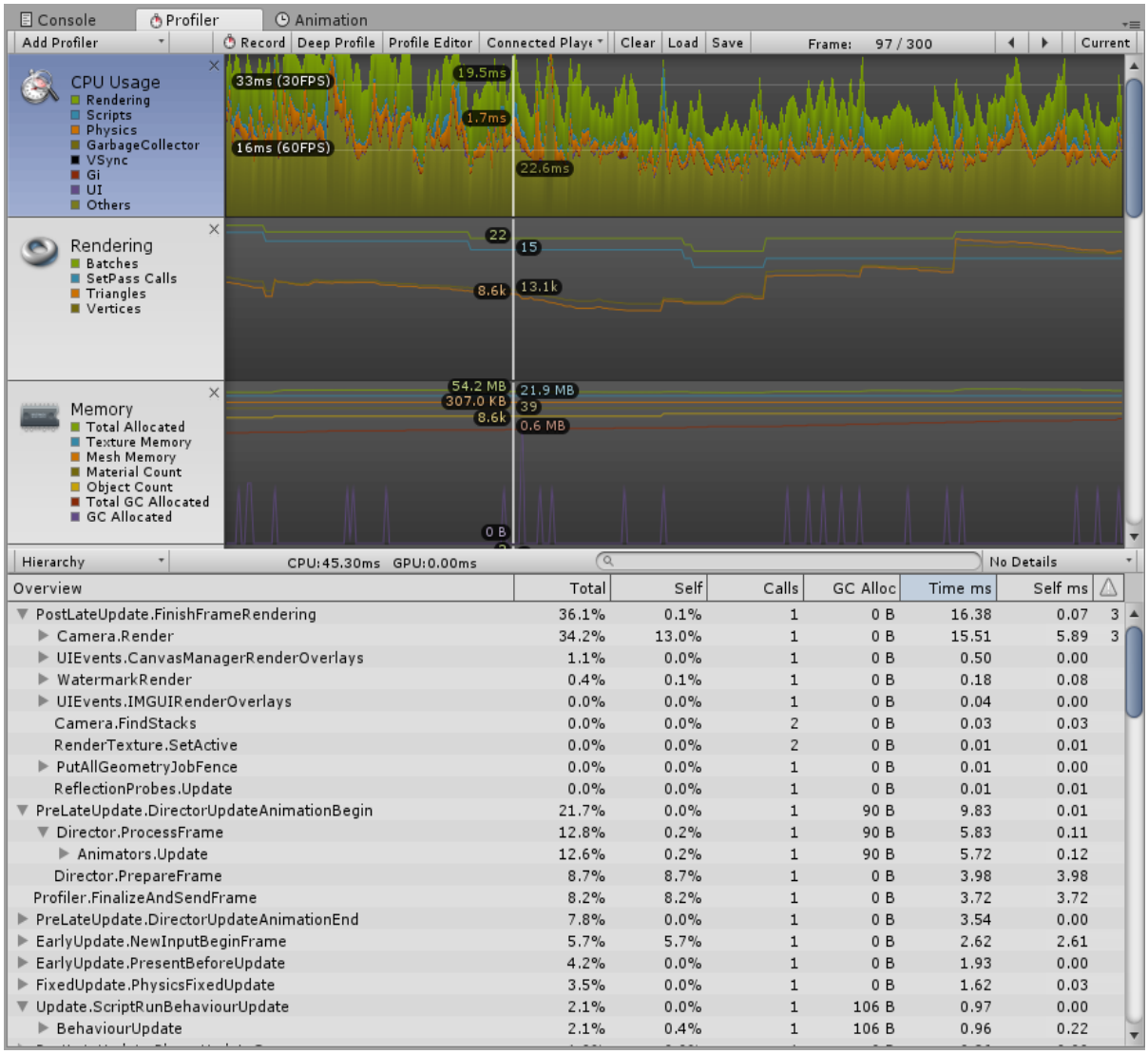
Mas como explicado no gráfico da figura 19 esses picos não levam a aplicação em nenhum momento a menos de 30 *FPS*, mantendo a mesma na região ótima para o jogo e plataforma desenvolvidos.

A figura 20, demonstrando a performance no "*MidGame*" do jogo, mostra no início diversos picos de performance, alguns chegando na margem dos 20 *FPS*. Isso se deve ao fato de o "*MidGame*" ser o período do jogo que pode ser considerado mais custoso. Pelo fato de que é durante ele que o *GarbageCollector* começará a apresentar um padrão constante pois a memória começa a estabilizar seu uso nesse período.

Este também é o momento onde o *script* de *Object Pooling* ainda não possui todas as peças instanciadas, havendo assim segmentos para instanciar ainda, além de outras pequenas transições internas da aplicação que tornam esse período de "transição" mais conturbado no quesito da performance no *smartphone* testado atualmente.

Porém, mesmo com os diversos picos visíveis o jogo não cai abaixo de 20 *FPS*, se mantendo mesmo em seus momentos conturbados na faixa ideal de performance para um jogo Android, no que tange a quantidade de *frames* por segundo.

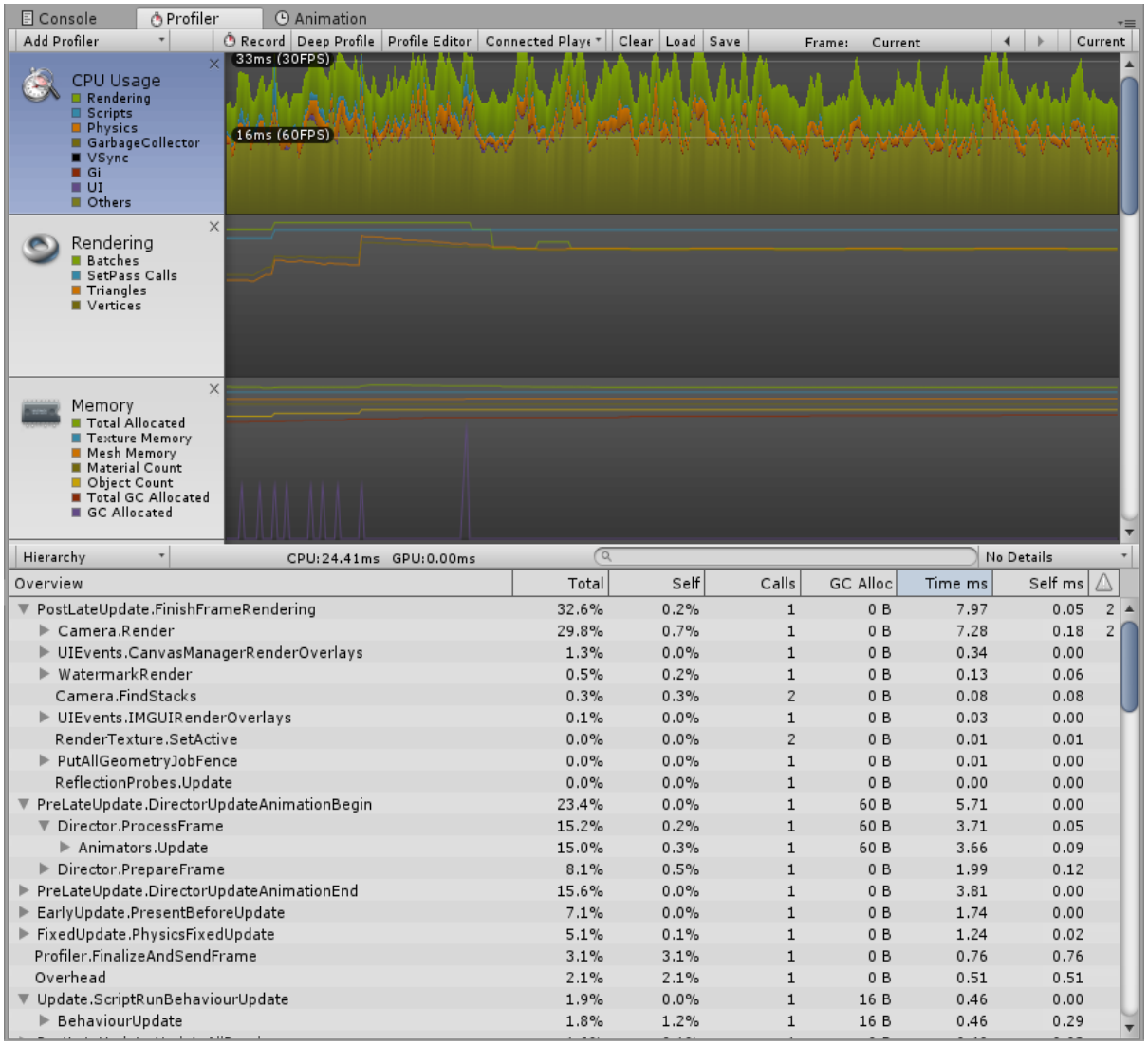
Figura 20: Dados do “*MidGame*”, *Asus Zenfone 5*



Fonte: Elaborada pelos autores

A figura 21, representando o “*LateGame*”, como pode ser observado o jogo já alcançou uma estabilidade, e apesar de não ter a mesma performance do “*EarlyGame*”, não fica tão atrás. Agora que o que foi explicado na imagem anterior já se estabilizou o jogo chegou em seu estágio final, fazendo com que não haja nenhuma mudança significativa no quesito de recursos até que o usuário perca o jogo.

Figura 21: Dados do "*LateGame*", *Asus Zenfone 5*



Fonte: Elaborada pelos autores

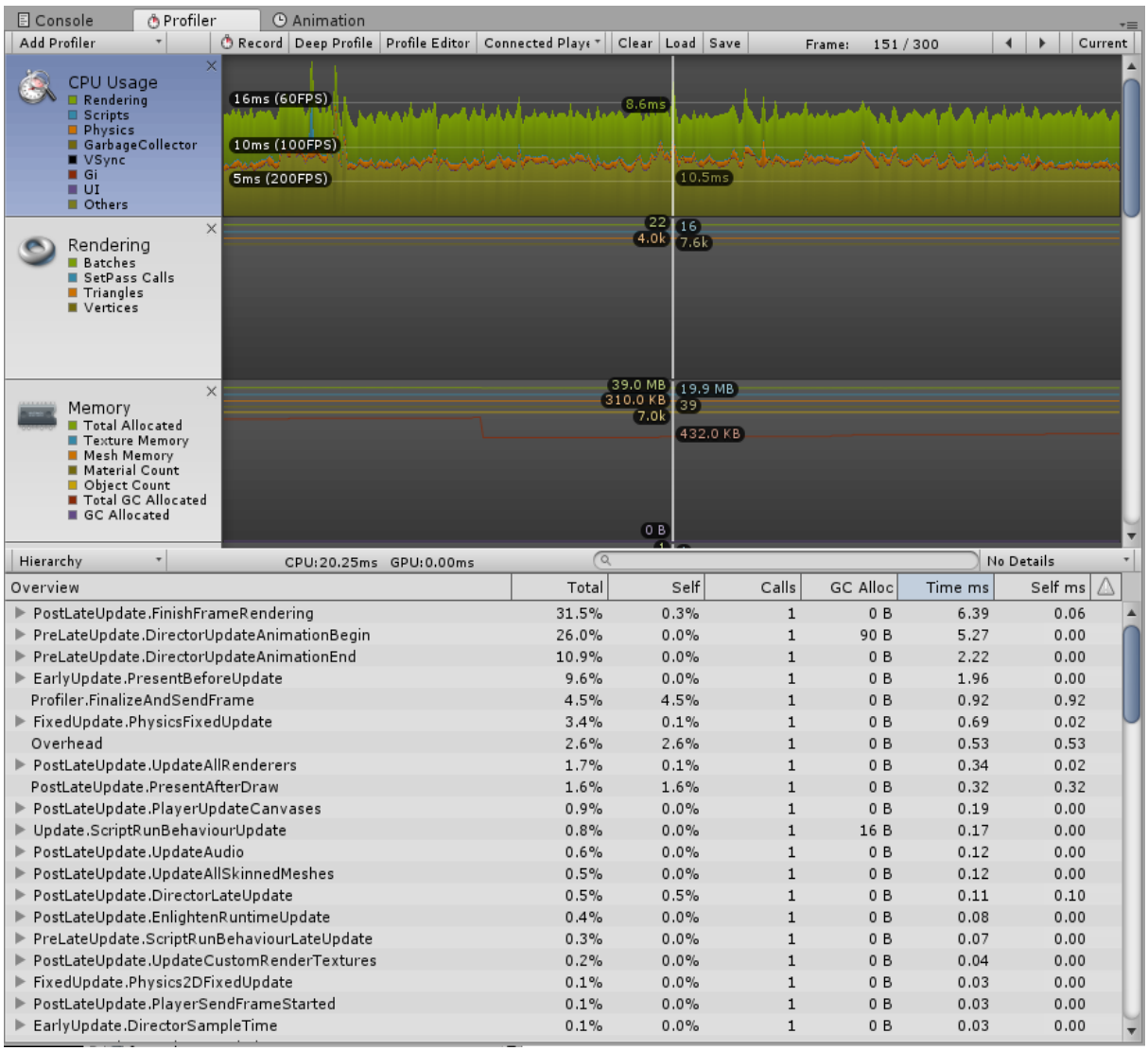
Nesse estágio, apresentado na figura 21, o jogo está estável e mantém uma performance média de 45 *FPS*, atingindo um padrão bem alto de performance para *games mobile* Android. As funções utilizadas durante a performance são as mesmas durante o processo todo, e as relevantes ao tópico foram explicadas anteriormente na análise do "*EarlyGame*".

Os resultados dos testes desta pesquisa no aparelho Zenfone 5 foram extremamente satisfatórios, considerando que ele é o aparelho, em termos de recurso, mais modesto, e o que se esperava uma performance abaixo do ideal.

O próximo aparelho que foi testado é um *smartphone* da marca Samsung, modelo *Galaxy J5 Prime*. Esse é o aparelho de testes considerado, em questão de recursos e preço, mediano.

A figura 22 demonstra os dados coletados da aplicação rodando na tela inicial do jogo.

Figura 22: Dados da tela inicial, *Samsung Galaxy J5 Prime*



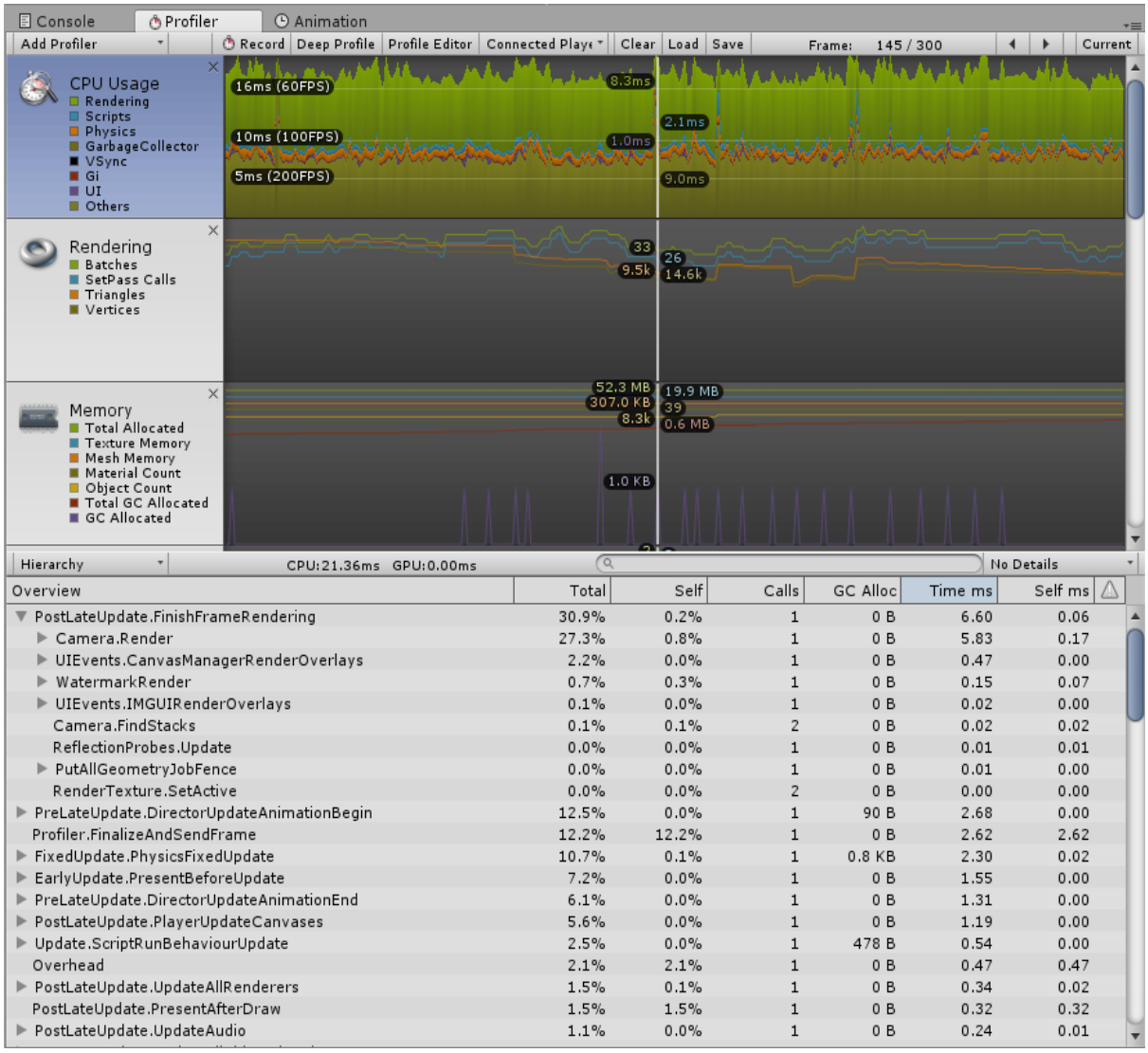
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PostLateUpdate.FinishFrameRendering	31.5%	0.3%	1	0 B	6.39	0.06
PreLateUpdate.DirectorUpdateAnimationBegin	26.0%	0.0%	1	90 B	5.27	0.00
PreLateUpdate.DirectorUpdateAnimationEnd	10.9%	0.0%	1	0 B	2.22	0.00
EarlyUpdate.PresentBeforeUpdate	9.6%	0.0%	1	0 B	1.96	0.00
Profiler.FinalizeAndSendFrame	4.5%	4.5%	1	0 B	0.92	0.92
FixedUpdate.PhysicsFixedUpdate	3.4%	0.1%	1	0 B	0.69	0.02
Overhead	2.6%	2.6%	1	0 B	0.53	0.53
PostLateUpdate.UpdateAllRenderers	1.7%	0.1%	1	0 B	0.34	0.02
PostLateUpdate.PresentAfterDraw	1.6%	1.6%	1	0 B	0.32	0.32
PostLateUpdate.PlayerUpdateCanvases	0.9%	0.0%	1	0 B	0.19	0.00
Update.ScriptRunBehaviourUpdate	0.8%	0.0%	1	16 B	0.17	0.00
PostLateUpdate.UpdateAudio	0.6%	0.0%	1	0 B	0.12	0.00
PostLateUpdate.UpdateAllSkinnedMeshes	0.5%	0.0%	1	0 B	0.12	0.00
PostLateUpdate.DirectorLateUpdate	0.5%	0.5%	1	0 B	0.11	0.10
PostLateUpdate.EnlightenRuntimeUpdate	0.4%	0.0%	1	0 B	0.08	0.00
PreLateUpdate.ScriptRunBehaviourLateUpdate	0.3%	0.0%	1	0 B	0.07	0.00
PostLateUpdate.UpdateCustomRenderTextures	0.2%	0.0%	1	0 B	0.04	0.00
FixedUpdate.Physics2DFixedUpdate	0.1%	0.0%	1	0 B	0.03	0.00
PostLateUpdate.PlayerSendFrameStarted	0.1%	0.0%	1	0 B	0.03	0.00
EarlyUpdate.DirectorSampleTime	0.1%	0.0%	1	0 B	0.03	0.00

Fonte: Elaborada pelos autores

Como pode ser observado nos gráficos de desempenho, o jogo consegue se manter acima da taxa dos 60 *FPS* constantemente, mostrando que com uma melhora nos recursos computacionais de *hardware*, o aparelho consegue manter uma taxa mais que otimizada de execução, sem problema nenhum na tela inicial.

A figura 23 apresenta os dados referentes ao "*EarlyGame*" do jogo no aparelho.

Figura 23: Dados do “*EarlyGame*”, *Samsung Galaxy J5 Prime*

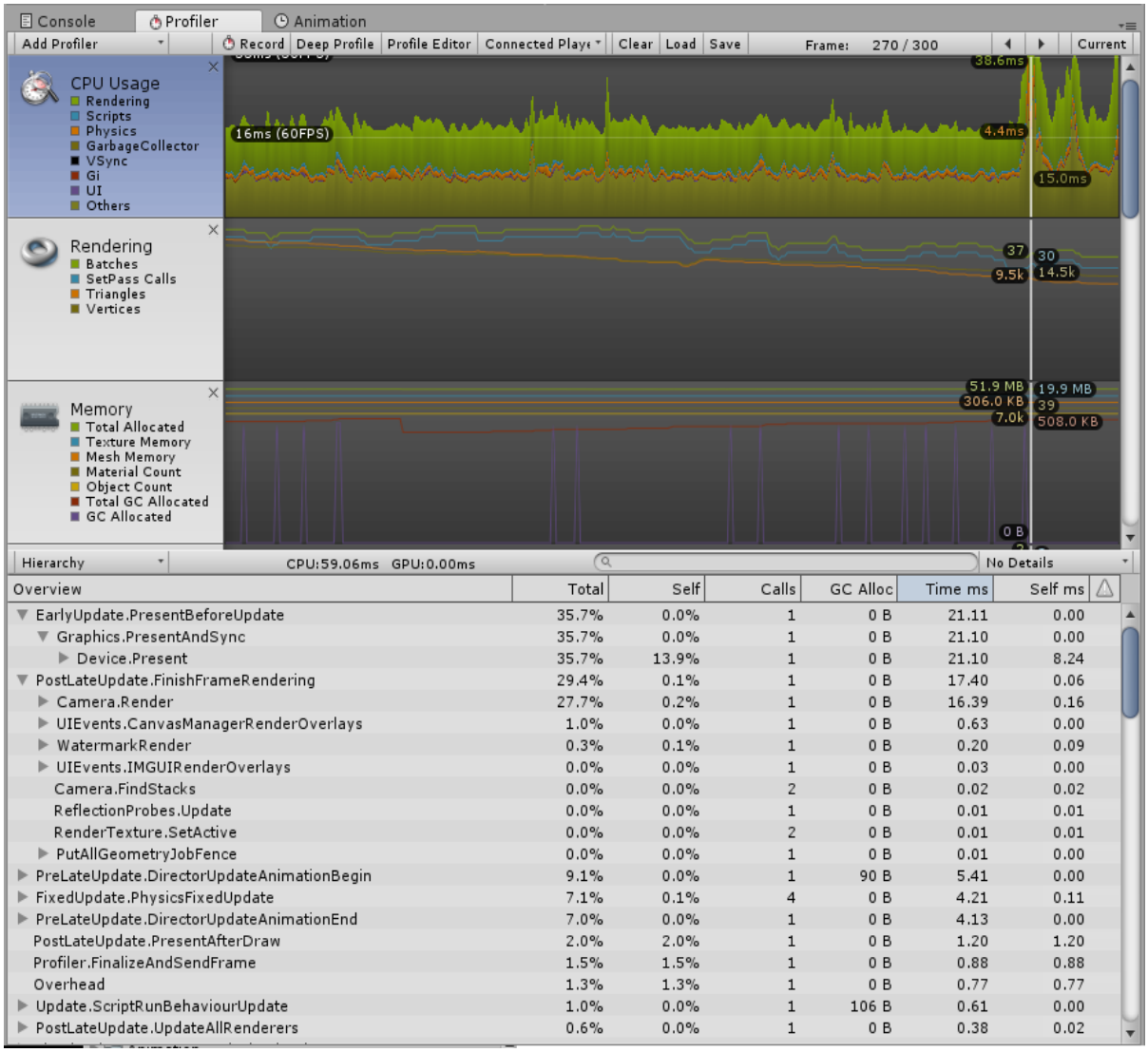


Fonte: Elaborada pelos autores

Como demonstra os gráficos de desempenho, comparado ao primeiro aparelho testado, já houve significativa melhora na performance geral do jogo, conseguindo se manter na média dos 60 *FPS* consistentemente, não havendo sequer picos de performance durante a execução da aplicação no “*EarlyGame*”.

Este fato demonstra que a aplicação está em um nível de otimização muito bom para aplicações *mobile* Android, tendo espaço para inserção de recursos gráficos e de mecânicas de jogo ainda mantendo um padrão alto de performance.

Figura 24: Dados do “*MidGame*”, *Samsung Galaxy J5 Prime*

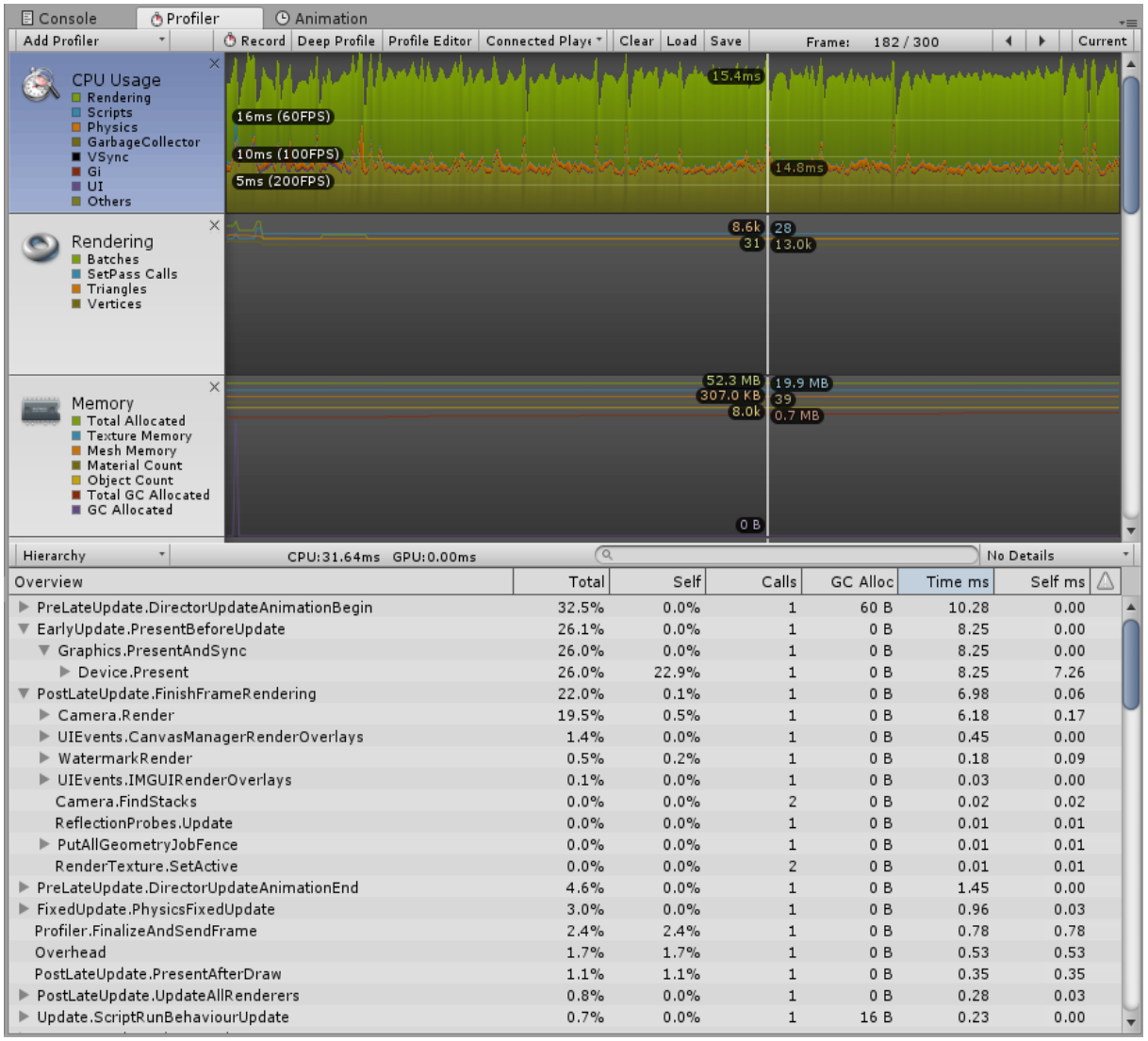


Fonte: Elaborada pelos autores

A figura 24, mostrando o “*MidGame*” do jogo, apresenta uma performance constante, mantendo uma taxa de *frames* por segundo bastante parecida com a da figura 23, mostrando que nesse dispositivo a aplicação lida melhor com a transição do “*EarlyGame*” para o “*MidGame*”, mantendo sua performance constante durante o período analisado da execução.

São observados apenas dois picos de performance no período dos dados analisados, e eles são causados por um “*EarlyUpdate*” executando “*Graphics.PresentAndSync*” e segundo Unity (2013), essa função é chamada quando o processador já terminou suas funções naquele *frame* e está apenas ocioso esperando a *GPU* terminar as suas funções. Como a capacidade gráfica dos dispositivos moveis é menor do que a de processamento, ela leva mais tempo, deixando o processador ocioso nesse período. Mais um sinal de uma aplicação “*GPU Bound*”.

Figura 25: Dados do “*LateGame*”, *Samsung Galaxy J5 Prime*



Fonte: Elaborada pelos autores

Na figura 25, são apresentados os dados da aplicação em seu “*LateGame*”. Apresentando um comportamento como já observado no mesmo período do dispositivo anterior, a aplicação por já estar sendo executado a algum tempo, e estar utilizando recursos de memória, renderização e diversos outros fatores perde um pouco da performance a longo termo.

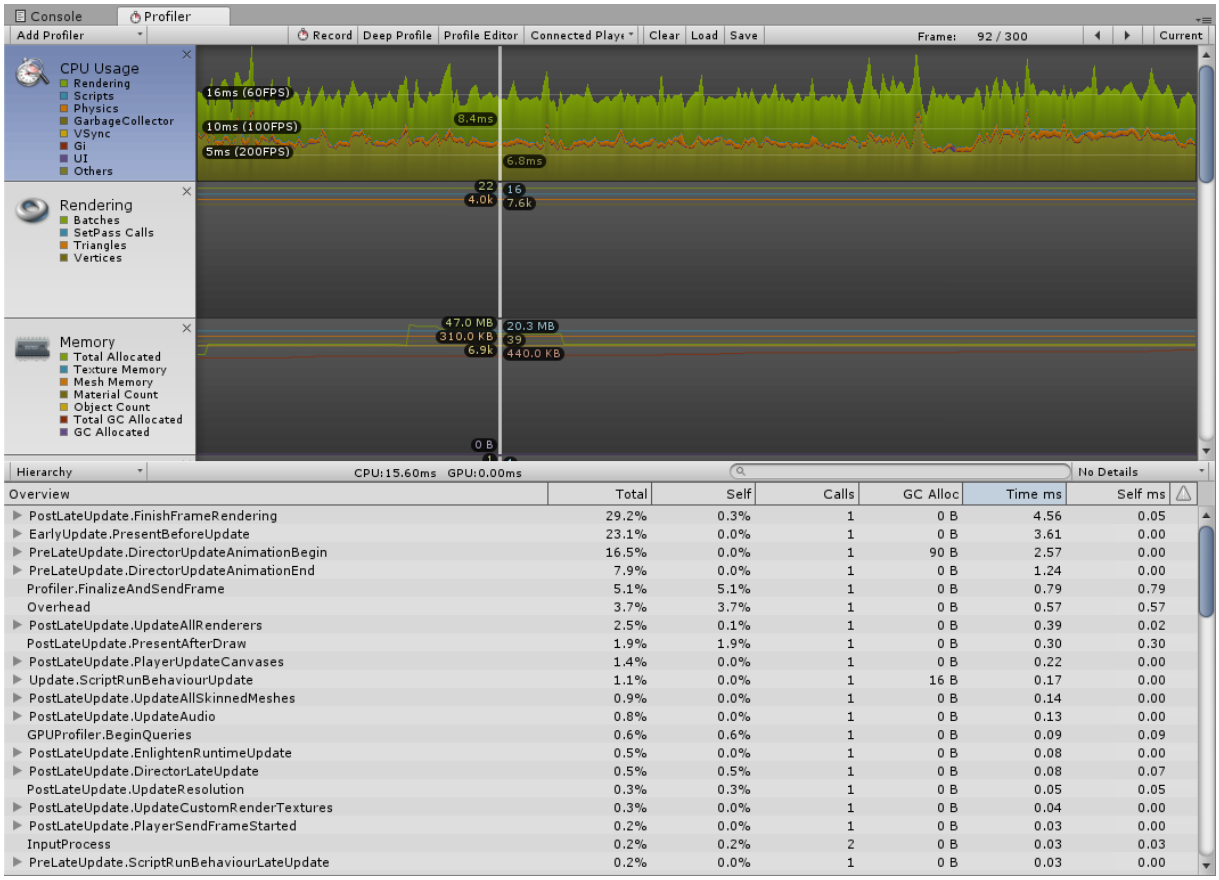
Mas essa “perda” de performance é extremamente relativa pois mesmo neste período mais conturbado a aplicação mantém a taxa dos 45 *FPS*, valor bem alto para o desempenho da aplicação *mobile.*

Os testes no aparelho *J5 Prime* apresentaram resultados acima dos esperados na pesquisa, mostrando que quando trabalhando com dispositivos de capacidade computacional média, ainda há uma quantidade imensa de espaço para os desenvolvedores trabalharem a

aplicação, se seguirem o trabalho descrito nesta pesquisa.

O próximo e ultimo aparelho é da marca Samsung, assim como o anterior, porém o modelo é um *Samsung Galaxy J7*. Este é o dispositivo de testes mais robusto e com o maior preço entre os três. A figura 26 apresenta o gráfico do desempenho na tela inicial da aplicação.

Figura 26: Dados da tela inicial, *Samsung Galaxy J7*



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PostLateUpdate.FinishFrameRendering	29.2%	0.3%	1	0 B	4.56	0.05
EarlyUpdate.PresentBeforeUpdate	23.1%	0.0%	1	0 B	3.61	0.00
PreLateUpdate.DirectorUpdateAnimationBegin	16.5%	0.0%	1	90 B	2.57	0.00
PreLateUpdate.DirectorUpdateAnimationEnd	7.9%	0.0%	1	0 B	1.24	0.00
Profiler.FinalizeAndSendFrame	5.1%	5.1%	1	0 B	0.79	0.79
Overhead	3.7%	3.7%	1	0 B	0.57	0.57
PostLateUpdate.UpdateAllRenderers	2.5%	0.1%	1	0 B	0.39	0.02
PostLateUpdate.PresentAfterDraw	1.9%	1.9%	1	0 B	0.30	0.30
PostLateUpdate.PlayerUpdateCanvases	1.4%	0.0%	1	0 B	0.22	0.00
Update.ScriptRunBehaviourUpdate	1.1%	0.0%	1	16 B	0.17	0.00
PostLateUpdate.UpdateAllSkinnedMeshes	0.9%	0.0%	1	0 B	0.14	0.00
PostLateUpdate.UpdateAudio	0.8%	0.0%	1	0 B	0.13	0.00
GPUProfiler.BeginQueries	0.6%	0.6%	1	0 B	0.09	0.09
PostLateUpdate.EnlightenRuntimeUpdate	0.5%	0.0%	1	0 B	0.08	0.00
PostLateUpdate.DirectorLateUpdate	0.5%	0.5%	1	0 B	0.08	0.07
PostLateUpdate.UpdateResolution	0.3%	0.3%	1	0 B	0.05	0.05
PostLateUpdate.UpdateCustomRenderTextures	0.3%	0.0%	1	0 B	0.04	0.00
PostLateUpdate.PlayerSendFrameStarted	0.2%	0.0%	1	0 B	0.03	0.00
InputProcess	0.2%	0.2%	2	0 B	0.03	0.03
PreLateUpdate.ScriptRunBehaviourLateUpdate	0.2%	0.0%	1	0 B	0.03	0.00

Fonte: Elaborada pelos autores

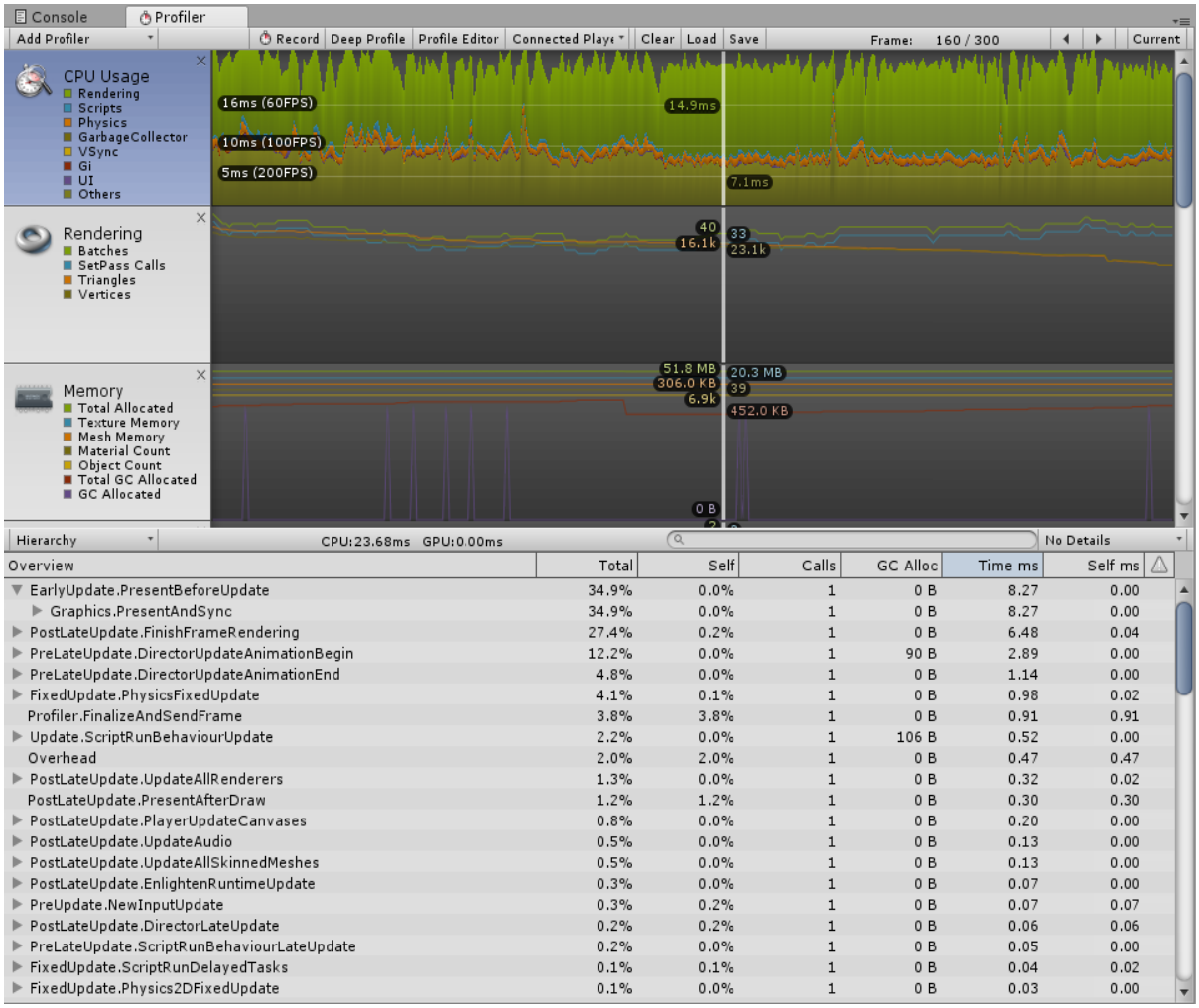
Como explicado nos testes anteriores e demonstrado também na figura 26, não há muita coisa acontecendo na tela inicial, logo a performance tende a ser bastante alta para esse período do ciclo de vida do jogo, girando em torno de 60 *FPS* no aparelho em questão, quantidade acima do considerado ótimo. Vale lembrar que a variação de performance entre os dispositivos testados varia não apenas por suas capacidades técnicas, mas também por arquivos pessoais instalados nos dispositivos, temperatura, consumo de memória atual, entre diversos outros.

Os *smartphones* testados são de voluntários para os testes da pesquisa, logo possuem arquivos pessoais, músicas, fotos, aplicações, etc. E tudo isso pode também afetar o

desempenho do jogo. Assim também como fatores externos como temperatura do aparelho, nível de conservação, etc.

A figura 27 aborda os dados do "*EarlyGame*" da aplicação.

Figura 27: Dados do "*EarlyGame*", *Samsung Galaxy J7*



Fonte: Elaborada pelos autores

À primeira vista, analisando os dados do gráfico, pode-se pensar que a performance da aplicação neste aparelho foi pior que em seu antecessor, o *Galaxy J5*, por manter uma taxa de *FPS* de em média 35. Mas é nesse momento que é necessária a interpretação dos gráficos e das informações apresentadas nele.

Como se pode observar na figura 27, na seção "*Overview*", 34.9% do gráfico de performance é ocupado pela função "*Graphics.PresentAndSync*", função já explicada no aparelho anterior. Isso não significa que o jogo está tendo uma queda de performance, significa que a aplicação está no momento puramente ligada à *GPU*, ou "*GPU Bound*".

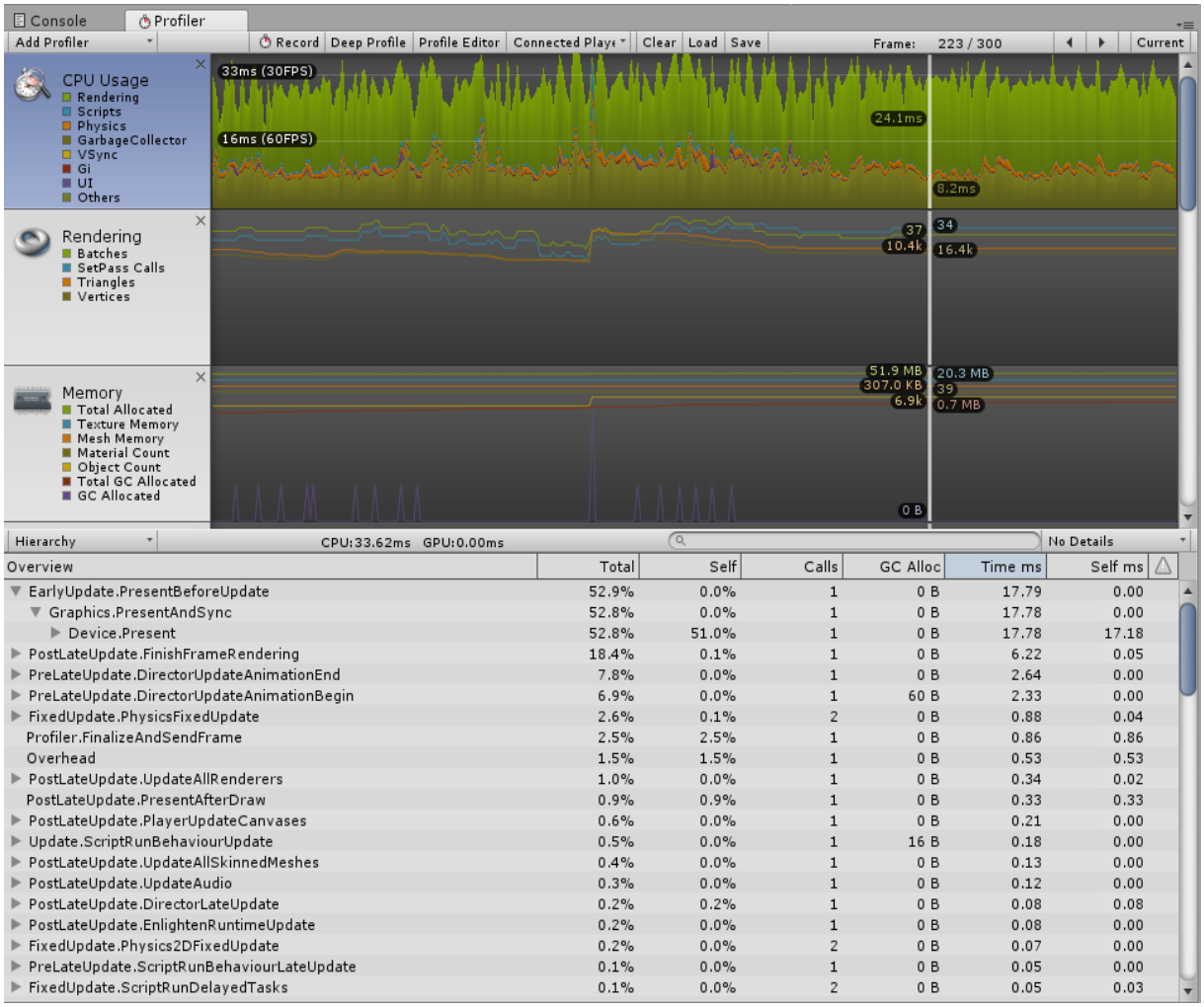
Mas se for considerado o crescimento da capacidade de processamento de um modelo para o outro, com o tempo que os mesmos ficaram "*GPU Bound*", a performance dos dois modelos nesse período do jogo foi praticamente a mesma, uma média de 60 *FPS*.

Não é possível demonstrar mais a fundo os números exatos do que foi explicado em questão de uso de GPU, pelo fato de as fabricantes de placas gráficas não darem muito suporte a ferramentas de *profile*, como o *Profiler* da Unity. Tornando ainda mais difícil a análise dos dados para as empresas e desenvolvedores. Apenas algumas placas mais modernas possuem essa compatibilidade.

Esse tipo de informação do gráfico apresentado demonstra que apesar de a capacidade de processamento ter avançado bastante do modelo *J5* para o *J7*, demonstrado pela rapidez com que o processador termina sua execução e já transfere as funções para a *GPU* terminar as dela, mas também demonstra como a capacidade gráfica dos aparelhos não cresceu na mesma intensidade ou sequer evoluiu.

A capacidade de processamento dos *smartphones* está longe de ser um problema no que tange ao desenvolvimento de jogos *mobile*, o que dificulta a vida dos desenvolvedores, como demonstram os gráficos nos aparelhos mais robustos, é a capacidade gráfica dos dispositivos, fazendo-se necessária a otimização da parte gráfica e de programação para este tipo de aplicação.

Figura 28: Dados do "*MidGame*", *Samsung Galaxy J7*

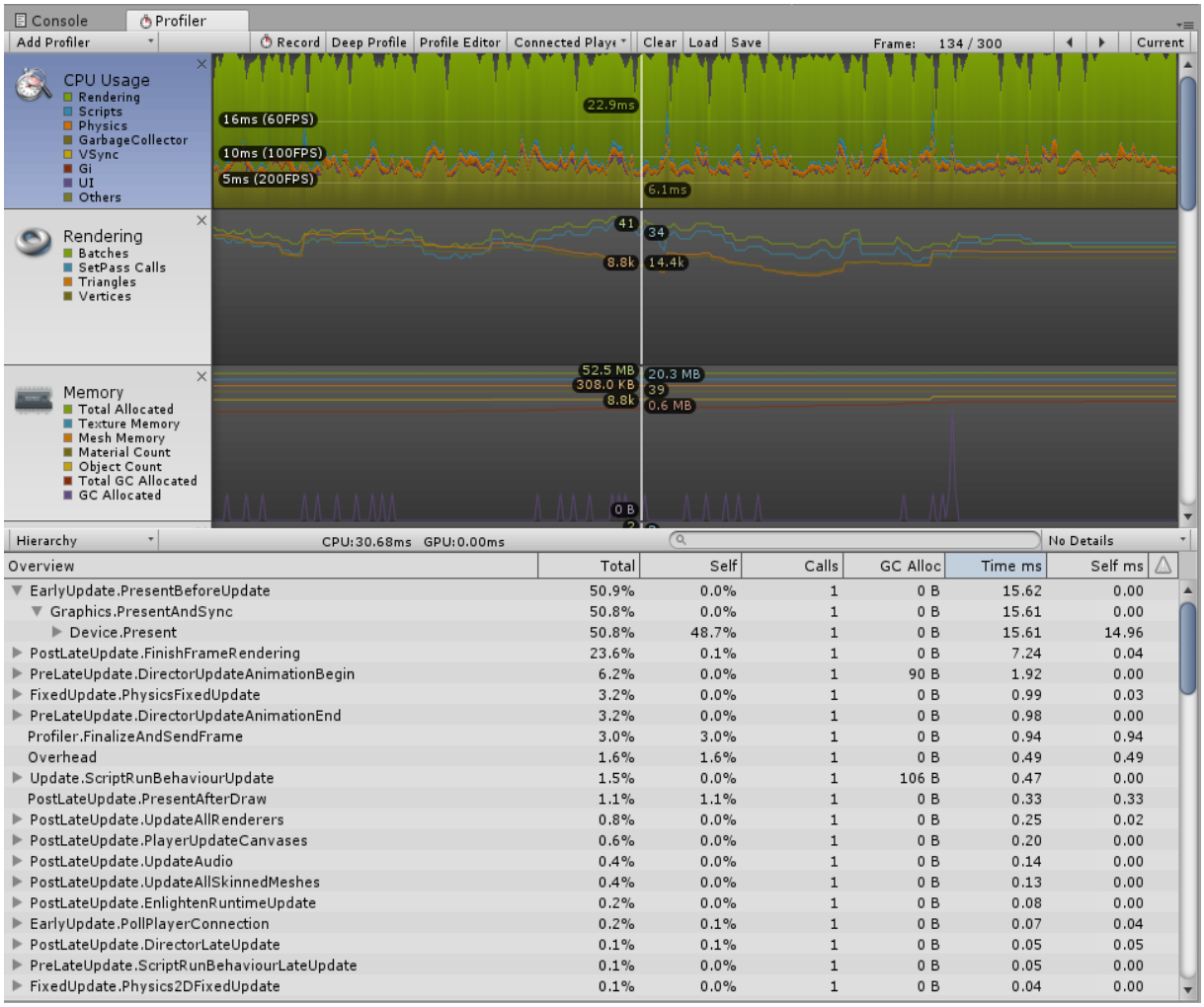


Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
EarlyUpdate.PresentBeforeUpdate	52.9%	0.0%	1	0 B	17.79	0.00
Graphics.PresentAndSync	52.8%	0.0%	1	0 B	17.78	0.00
Device.Present	52.8%	51.0%	1	0 B	17.78	17.18
PostLateUpdate.FinishFrameRendering	18.4%	0.1%	1	0 B	6.22	0.05
PreLateUpdate.DirectorUpdateAnimationEnd	7.8%	0.0%	1	0 B	2.64	0.00
PreLateUpdate.DirectorUpdateAnimationBegin	6.9%	0.0%	1	60 B	2.33	0.00
FixedUpdate.PhysicsFixedUpdate	2.6%	0.1%	2	0 B	0.88	0.04
Profiler.FinalizeAndSendFrame	2.5%	2.5%	1	0 B	0.86	0.86
Overhead	1.5%	1.5%	1	0 B	0.53	0.53
PostLateUpdate.UpdateAllRenderers	1.0%	0.0%	1	0 B	0.34	0.02
PostLateUpdate.PresentAfterDraw	0.9%	0.9%	1	0 B	0.33	0.33
PostLateUpdate.PlayerUpdateCanvases	0.6%	0.0%	1	0 B	0.21	0.00
Update.ScriptRunBehaviourUpdate	0.5%	0.0%	1	16 B	0.18	0.00
PostLateUpdate.UpdateAllSkinnedMeshes	0.4%	0.0%	1	0 B	0.13	0.00
PostLateUpdate.UpdateAudio	0.3%	0.0%	1	0 B	0.12	0.00
PostLateUpdate.DirectorLateUpdate	0.2%	0.2%	1	0 B	0.08	0.08
PostLateUpdate.EnlightenRuntimeUpdate	0.2%	0.0%	1	0 B	0.08	0.00
FixedUpdate.Physics2DFixedUpdate	0.2%	0.0%	2	0 B	0.07	0.00
PreLateUpdate.ScriptRunBehaviourLateUpdate	0.1%	0.0%	1	0 B	0.05	0.00
FixedUpdate.ScriptRunDelayedTasks	0.1%	0.0%	2	0 B	0.05	0.03

Fonte: Elaborada pelos autores

Como pode ser observado na figura 28 do gráfico do "*MidGame*", este modelo também possui uma transição tranquila vindo do ciclo anterior, mostrando até uma melhora de performance entre os 40 e 45 *FPS*. Como pode também ser observado, 50% da performance é ocupada pela função "*Graphics.PresentAndSync*", mostrando que nesse tipo de aplicação a *GPU* é um fator de extrema importância e parte crucial para uma performance ideal do jogo.

Figura 29: Dados do "*LateGame*", *Samsung Galaxy J7*



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
EarlyUpdate.PresentBeforeUpdate	50.9%	0.0%	1	0 B	15.62	0.00
Graphics.PresentAndSync	50.8%	0.0%	1	0 B	15.61	0.00
Device.Present	50.8%	48.7%	1	0 B	15.61	14.96
PostLateUpdate.FinishFrameRendering	23.6%	0.1%	1	0 B	7.24	0.04
PreLateUpdate.DirectorUpdateAnimationBegin	6.2%	0.0%	1	90 B	1.92	0.00
FixedUpdate.PhysicsFixedUpdate	3.2%	0.0%	1	0 B	0.99	0.03
PreLateUpdate.DirectorUpdateAnimationEnd	3.2%	0.0%	1	0 B	0.98	0.00
Profiler.FinalizeAndSendFrame	3.0%	3.0%	1	0 B	0.94	0.94
Overhead	1.6%	1.6%	1	0 B	0.49	0.49
Update.ScriptRunBehaviourUpdate	1.5%	0.0%	1	106 B	0.47	0.00
PostLateUpdate.PresentAfterDraw	1.1%	1.1%	1	0 B	0.33	0.33
PostLateUpdate.UpdateAllRenderers	0.8%	0.0%	1	0 B	0.25	0.02
PostLateUpdate.PlayerUpdateCanvases	0.6%	0.0%	1	0 B	0.20	0.00
PostLateUpdate.UpdateAudio	0.4%	0.0%	1	0 B	0.14	0.00
PostLateUpdate.UpdateAllSkinnedMeshes	0.4%	0.0%	1	0 B	0.13	0.00
PostLateUpdate.EnlightenRuntimeUpdate	0.2%	0.0%	1	0 B	0.08	0.00
EarlyUpdate.PollPlayerConnection	0.2%	0.1%	1	0 B	0.07	0.04
PostLateUpdate.DirectorLateUpdate	0.1%	0.1%	1	0 B	0.05	0.05
PreLateUpdate.ScriptRunBehaviourLateUpdate	0.1%	0.0%	1	0 B	0.05	0.00
FixedUpdate.Physics2DFixedUpdate	0.1%	0.0%	1	0 B	0.04	0.00

Fonte: Elaborada pelos autores

O gráfico do "*LateGame*", na figura 29, apresenta o mesmo padrão do modelo anterior, com leve queda de performance pelo gerenciamento prolongado de recursos em constante mudança. Apresenta uma taxa de 30 a 40 *FPS*, taxa ainda acima dos padrões de otimização, mas o modelo deixou bastante a desejar na parte gráfica, ficando para trás em comparação ao seu antecessor.

Finalizados o desenvolvimento da aplicação e os testes pode-se concluir através dessa pesquisa que, através dos conceitos e técnicas de otimização de jogos *mobile Android* desenvolvidos em Unity 3D, discutidos nessa pesquisa, pode-se atingir um padrão bem alto de performance geral, deixando ainda assim espaço para incrementações extras e melhorias na aplicação.

Os resultados dos testes foram extremamente satisfatórios, com o aparelho mais modesto e o de meio termo superando as expectativas, mostrando um desempenho excelente

em uma aplicação de testes relativamente custosa.

O único teste que deixou um pouco a desejar foram os do ultimo modelo. O fato de que sua capacidade de processamento é tão superior à sua capacidade gráfica, faz com que haja uma discrepância notável nesses pontos, fazendo com que isso cause impacto significativo na performance geral.

O aparelho que apresentou melhor performance com o jogo da pesquisa foi o *Galaxy J5 Prime*, seguido pelo *ASUS Zenfone 5*. Isso dá-se pelo fato de sua performance se manter excepcional comparado as suas capacidades técnicas, e em comparação ao modelo mais robusto do espaço amostral, o que fez com que suas performances se sobressaíssem ainda mais. Porém, isso não muda o fato de a performance no *Samsung Galaxy J7* ter sido próxima de perfeita, mas, era esperado mais do modelo devido à suas capacidades.

O APK do jogo desenvolvido na pesquisa foi disponibilizado para download no link: https://drive.google.com/open?id=1-8xl27TXl-O9fZYnXbCRTG1cb1Cm1WiG.

5. CONSIDERAÇÕES FINAIS

Por meio do desenvolvimento desta pesquisa pôde-se analisar todos os pontos que abrangem a otimização de jogos em si. Desde o pensamento e planejamento da estrutura da aplicação, todos os pontos que englobam o desenvolvimento, desde os *scripts* até a implementação dos mesmos e como eles se comportam em relação ao jogo como um todo.

O desenvolvimento de uma aplicação de testes para a pesquisa ajuda a demonstrar os pontos abordados durante a explicação teórica, demonstrando cada aspecto que poderia e foi melhorado, cada possível gargalo de performance ao longo do desenvolvimento. E até a identificação desses gargalos antes mesmo de acontecerem, por meio da lógica de programação e estudo aprofundado da *engine* a qual a aplicação foi desenvolvida.

Após a finalização do desenvolvimento e implementação, os testes foram planejados e executados de forma que poderiam cobrir o maior leque de possibilidades. Desde a escolha dos modelos, para que fosse abrangido um publico de usuários de três níveis de configurações de aparelhos diferentes. Até a própria execução dos testes, que foi pensada para englobar todos os ciclos de vida da aplicação, que são, tela inicial, inicio do jogo, meio do jogo, e final do jogo. No caso do tema do jogo da pesquisa esses períodos são demarcados por tempo de jogo, por se tratar de um jogo do gênero *Runner*.

A coleta de dados dessa maneira permitiu analisar como o jogo se comporta, em seus níveis mais internos, em cada um desses períodos do seu ciclo. Mostrando as características e pontos intrínsecos à cada um deles.

Pode se observar como a tela inicial e os primeiros momentos da execução desse tipo de aplicação são os menores consumidores de recursos e geram menos problemas de performance. Mas também demonstrou a quão turbulenta pode ser a transição para o meio e final do jogo, onde os recursos e mecânicas do jogo já estão mais estabilizados.

Cada conceito do desenvolvimento desta pesquisa foi pensado para que se possa ter o melhor e mais claro entendimento possível sobre o tema, tentando ser o mais claro e simples possível nas partes técnicas, porém sem deixar nem os mínimos detalhes de fora da explicação.

A conclusão final deste trabalho é que os resultados planejados foram obtidos com sucesso, e os conceitos e técnicas aqui abordados abrangem de maneira satisfatória os pontos da otimização de jogos *mobile Android* desenvolvidos em Unity3D.

6. REFERÊNCIAS

DICKINSON, Chris. **Unity 5 Game Optimization: Master performance optimization for Unity3D applications with tips and techniques that cover every aspect of Unity3D Engine.** 1.ed. Birmingham, UK: Packt Publishing Ltd., 2015;

KORZUSZEK, Piotr. ***Mobile* Optimization – Batching in Unity**, 2015. Disponível em:< http://blog.theknightsofUnity.com/*mobile*-optimization-batching-Unity/>. Acesso em 28 de abril de 2017;

MARTINS, André. **Otimização de jogos com Unity 3D,** 2014. Disponível em: <http://intentor.com.br/otimizacao-de-jogos-com-Unity-3d/>. Acesso em 24 de abril de 2017;

MCDONALD, Emma. **THE GLOBAL GAMES MARKET WILL REACH $108.9 BILLION IN 2017 WITH *MOBILE* TAKING 42%**, 2017. Disponível em: <https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-*mobile*-taking-42/>. Acesso em 21 de abril de 2017;

PRODANOV, Cleber Cristiano; FREITAS, Ernani Cesar de. **Metodologia do trabalho científico: métodos e técnicas da pesquisa e do trabalho científico.** 2.ed. Nova Hamburgo – Rio Grande Sul: Feevale, 2013;

REICH, Wendelin. **C# Memory Management for Unity Developers (part 1 of 3)**, 2013. Disponível em:< http://www.gamasutra.com/blogs/WendelinReich/20131109/203841/C_Memory_Management_for_Unity_Developers_part_1_of_3.php>. Acesso em 20 de abril de 2017;

_____, Wendelin. **C# Memory Management for Unity Developers (part 2 of 3)**, 2013. Disponível em:< http://www.gamasutra.com/blogs/WendelinReich/20131119/203842/C_Memory_Management_for_Unity_Developers_part_2_of_3.php>. Acesso em 20 de abril de 2017;

_____, Wendelin. **C# Memory Management for Unity Developers (part 3 of 3)**, 2013. Disponível em:< http://www.gamasutra.com/blogs/WendelinReich/20131127/203843/C_Memory_Management_for_Unity_Developers_part_3_of_3.php>. Acesso em 20 de abril de 2017;

STUDIO PEPWUPER. **Top 10 Reasons to Choose Unity 3D for App and Game Development**, 2013. Disponível em: <http://www.pepwuper.com/top-10-reasons-to-choose-Unity-3d-for-app-and-game-development/>. Acesso em 24 de abril de 2017;

TEC. **Brasil é o terceiro país do mundo que fica mais tempo on-line no celular,** 201**5.** Folha de São Paulo. Disponível em:<http://www1.folha.uol.com.br/tec/2015/09/1679423-

brasil-e-terceiro-pais-do-mundo-que-fica-mais-tempo-on-line-no-celular.shtml>. Acesso em 14 de maio de 2017;

THOMAN, Peter. **What 'optimization' really means in games**, 2016. Disponível em: <http://www.pcgamer.com/what-optimization-really-means-in-games/>. Acesso em 24 de abril de 2017;

TORRES, Filipe. **O Crescimento do Mercado de Jogos Mobile,** 2016. Disponível em:< http://congressodemobile.com.br/os-grandes-motivos-do-crescimento-do-mercado-de-jogos-mobile/>. Acesso em 14 de maio de 2017;

UNITY. **CollisionDetectionMode**, 2017. Disponível em:< https://docs.Unity3d.com/ScriptReference/CollisionDetectionMode.html>. Acesso em 23 de abril de 2017;

_____. **Colliders**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/CollidersOverview.html>. Acesso em 22 de abril de 2017;

_____. **Draw Call Batching**, 2013. Disponível em:< https://docs.Unity3d.com/430/Documentation/Manual/DrawCallBatching.html>. Acesso em 18 de abril de 2017;

_____. **Draw call batching**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/DrawCallBatching.html>. Acesso em 26 de abril de 2017;

_____. **Execution Order of Event Functions**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/ExecutionOrder.html>. Acesso em 12 de abril de 2017;

_____. **MonoBehaviour.Update()**, 2017. Disponível em:< https://docs.Unity3d.com/ScriptReference/MonoBehaviour.Update.html>. Acesso em 19 de abril de 2017;

_____. **Overview: Performance Optimization**, 2017. Disponível em: <https://docs.Unity3d.com/410/Documentation/ScriptReference/index.Performance_Optimization.html>. Acesso em 25 de abril de 2017;

_____. **Optimizing scripts in Unity games,** 2017**.** Disponível em: <https://Unity3d.com/pt/learn/tutorials/topics/performance-optimization/optimizing-scripts-Unity-games?playlist=44069**>.** Acesso em 25 de abril de 2017;

_____. **Optimizations**, 2017. Disponível em: < https://docs.Unity3d.com/Manual/*Mobile*Optimisation.html>. Acesso em 21 de abril de 2017;

_____. **Optimizing Garbage Collection in Unity games**, 2017. Disponível em:<

https://Unity3d.com/pt/learn/tutorials/topics/performance-optimization/optimizing-garbage-collection-Unity-games>. Acesso em 22 de abril de 2017;

_____. **Optimizing Scripts**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/*Mobile*OptimizationPracticalScriptingOptimizations.html>. Acesso em 18 de abril de 2017;

_____. **Pratical guide to optmization for *mobile*s**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/*Mobile*OptimizationPracticalGuide.html>. Acesso em 16 de abril de 2017;

_____. **Rigidbody overview**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/RigidbodiesOverview.html>. Acesso em 23 de abril de 2017;

_____. **Scripting and Gameplay Methods**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/*Mobile*OptimizationScriptingMethods.html>. Acesso em 20 de abril de 2017;

_____. **Time and Framerate Management**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/TimeFrameManagement.html>. Acesso em 23 de abril de 2017;

_____. **Time Manager**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/class-TimeManager.html>. Acesso em 20 de abril de 2017;

_____. **Understanding Automatic Memory Management**, 2017. Disponível em:< https://docs.Unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html>. Acesso em 22 de abril de 2017;

‘**Update and FixedUpdate – Unity Official Tutorials**’. Produção: Unity, 2013. Disponível em:< https://www.youtube.com/watch?v=ZukcUv3pyXQ>. Acesso em 20 de abril de 2017;

WLOKA, Matthias. **“Batch, Batch, Batch:” What does it really mean?**, 2003. Disponível em:< https://www.nvidia.com/docs/IO/8228/BatchBatchBatch.pdf>. Acesso em 22 de abril de 2017;