UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

MATEUS TOMOO YONEMOTO PEIXOTO

BENCHMARKING DE MOTORES DE JOGOS 2D

CAMPO MOURÃO

2021

MATEUS TOMOO YONEMOTO PEIXOTO

BENCHMARKING DE MOTORES DE JOGOS 2D

2D Game Engine Benchmarking

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Marcos Silvano Orita Almeida

CAMPO MOURÃO

2021

MATEUS TOMOO YONEMOTO PEIXOTO

BENCHMARKING DE MOTORES DE JOGOS 2D

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 30/novembro/2020

Marcos Silvano Almeida
Doutorado
UTFPR

Juliano Henrique Foleiss
Doutorado
UTFPR

Lúcio Geronimo Valentin
Doutorado
UTFPR

CAMPO MOURÃO

2021

RESUMO

Os motores gratuitos para desenvolvimento de jogos têm sido um elemento chave para a popularização e crescimento do mercado de jogos, uma vez que agilizam o processo de produção, reduzindo complexidade e tempo de implementação. A escolha do motor a ser utilizado em um novo projeto é geralmente apoiada em dois critérios: o conjunto de recursos que oferece e a *performance* que é possível alcançar no jogo resultante. Construir testes de desempenho que possam aferir a *performance* de diferentes motores e, dessa forma, ajudar nessa decisão pode ser tornar um problema para os desenvolvedores, pois há um considerável custo de tempo envolvido. Nesse contexto, o objetivo deste trabalho foi construir aplicativos de *benchmarking* (avaliação comparativa) que permitam a realização de perfilamento de *performance* e testes de estresse sobre recursos de diferentes motores de dois motores *open source 2D*: **Phaser** e **LÖVE**. Para tanto, foram implementados três testes de estresse em sistemas diferentes (Windows, Linux e Android), que permitiram a coleta de dados para uma avaliação comparativa entre os motores. Com base na análise dos resultados, observou-se que o motor **LÖVE** obteve melhores resultados sobre os critérios de performance estabelecidos. Espera-se que os resultados deste trabalho possam auxiliar desenvolvedores a selecionar o motor mais adequado às suas necessidades.

Palavras-chaves: Motor de Jogos. Performance de Jogos. Benchmark de Motores. Teste de Estresse.

// ABSTRACT

The free engines for game development have been a key element for the popularization and growth of the games market, since they speed up the production process, reducing complexity and time of implementation. The selection of the engine to be used in a new project is generally supported by two criteria: the set of features it offers and the performance that can be achieved in the resulting game. Building performance tests that can measure the performance of different engines and, thus, help in this decision can be a problem for developers, as there is a considerable cost of time involved. In this context, the objective of this project was to build benchmarking applications (comparative evaluation) that allow the performance profiling and stress tests on resources of different engines of two open source 2D engines: **Phaser** and **LÖVE**. To this end, three stress tests were implemented on different systems (Windows, Linux and Android), which allowed data collection for a comparative assessment between the engines. Based on the analysis of the results, it was observed that the **LÖVE** engine obtained better results on the established performance criteria. It is hoped that the results of this project can assist developers in selecting the most suitable engine for their needs.

Keywords: Game Engines. Games Performance. Engine Benchmarking. Stress Tests.

LISTA DE ILUSTRAÇÕES

2.1 Ciclo de Vida do Phaser.js 13
2.2 IDE do Motor Unity 14
2.3 *Benchmark* do *Framework* Phaser.js 14
2.4 Teste de Estresse do *Benchmark* da Unity. 15

4.1 Diagrama das Etapas. 19
4.2 Gráfico de Motores Mais Populares. 21
4.3 Gráfico de Motores 2D e de Código aberto. 22
4.4 Arquitetura do Sistema 25

5.1 Fluxo do Benchmark 31
5.2 Fluxo do Starter 31
5.3 Fluxo dos Testes 31

A.1 Dispositivo Desktop - i7-7500U 52
A.2 Dispositivo Móvel - Galaxy Note 10 Lite 52
A.3 Dispositivo Desktop - Ryzen 3 2200G 53
A.4 Dispositivo Desktop - FX 6300 53
A.5 Dispositivo Móvel - Redmi 5 Plus 53
A.6 Dispositivo Móvel - Zenfone 3 53
A.7 Dispositivo Móvel - Redmi 6A 53
A.8 Dispositivo Móvel - Moto G7 Play 53
A.9 Dispositivo Móvel - Moto G8 Power Lite 54
A.10 Dispositivo Desktop - A12-9720P 54
A.11 Dispositivo Móvel - Moto G4 54
A.12 Dispositivo Desktop - i7-4790 54
A.13 Dispositivo Desktop - i5-6600 54
A.14 Dispositivo Desktop - Ryzen 5 3600 54
A.15 Dispositivo Móvel - Galaxy Note 10 Plus 55
A.16 Dispositivo Desktop - Ryzen 5 3500X 55
A.17 Dispositivo Móvel - Redmi 6 55
A.18 Dispositivo Móvel - Galaxy S10 Plus 55
A.19 Dispositivo Móvel - Galaxy S9 Plus 55
A.20 Dispositivo Desktop - i7-8700 55
A.21 Dispositivo Desktop - FX 8350 56
A.22 Dispositivo Móvel - Galaxy S10e 56

A.23 Dispositivo Desktop - i5-7300HQ 56
A.24 Dispositivo Desktop - i5-7300HQ 56
A.25 Dispositivo Móvel - Galaxy S10 56
A.26 Dispositivo Móvel - Moto G6 56
A.27 Dispositivo Móvel - Huawei P30 Lite 57
A.28 Dispositivo Desktop - i5-9400F 57
A.29 Dispositivo Móvel - Zenfone 5 57
A.30 Dispositivo Móvel - Galaxy A01 Core 57

LISTA DE TABELAS

4.1 Número de Resultados Por Motor 23
4.2 Motores Por Linguagem 23

5.1 Tecnologias JavaScript e Seus Interpretadores 34
5.2 Teste 1 com Porcentagem do Tempo em que o Motor obteve *Performance* Superior . 35
5.3 Teste 1 com Quantidade de *Sprites* na Tela 36
5.4 Teste 1 com Quantidade de Sprites Mantendo*Performance* Ideal em 60 FPS 37
5.5 Teste 2 com Porcentagem do Tempo em que o Motor obteve *Performance* Superior . 38
5.6 Teste 2 com Quantidade de Sprites na Tela 39
5.7 Teste 2 com Quantidade de Sprites Mantendo*Performance* Ideal em 60 FPS 39
5.8 Teste 1 com Porcentagem do Tempo em que o Motor obteve *Performance* Superior . 40
5.9 Teste 1 com Quantidade de Sprites na Tela 41
5.10 Teste 1 com Quantidade de Sprites Mantendo*Performance* Ideal em 60 FPS 42
5.11 Teste 2 com Porcentagem do Tempo em que o Motor obteve *Performance* Superior . 43
5.12 Teste 2 com Quantidade de Sprites na Tela 44
5.13 Teste 2 com Quantidade de Sprites Mantendo*Performance* Ideal em 60 FPS 45
5.14 Dispositivos não monotônicos 45
5.15 Teste 3 com Quantidade de *Sprites* na Tela 46

SUMÁRIO

1 Introdução 8
1.1 Considerações Preliminares 8
1.2 Problema de Pesquisa 9
1.3 Objetivo 10
1.3.1 Objetivos Específicos 10
1.4 Contribuições 11
1.5 Organização do Texto 11
2 Conceitos 12
2.1 Motores de Jogos 12
2.1.1 *Framework* 12
2.1.2 Editor Visual ou Ambiente Integrado 13
2.2 *Benchmark* 14
2.3 Teste de Estresse 15
2.4 Considerações Finais 15
3 Trabalhos Relacionados 17
3.1 *A Comparison of Game Engines and Languages* 17
3.2 *Comparison and Evaluation of 3D Mobile Game Engines* 18
3.3 Considerações Finais 18
4 Materiais e Métodos 19
4.1 Seleção dos Motores 19
4.2 Definição dos Testes 24
4.2.1 Construção dos Aplicativos de *Benchmark* 25
4.3 Execução dos Testes 26
4.4 Coleta de Dados 26
4.5 Análise dos Dados 26
5 Resultados 28
5.1 Tecnologias Auxiliares 28
5.2 Aplicativos de *Benchmark* 28
5.2.1 Dificuldades Encontradas e Soluções Utilizadas 32
5.3 Interpretadores das Linguagens 33
5.4 Discussão e Análise 34
5.4.1 Dispositivos Desktop 35
5.4.2 Dispositivos Móveis 40
5.4.3 Teste 3: Tilemap Com Várias Camadas 46
5.4.4 Considerações Finais 46

6 Conclusões 47
Referências 49
Apêndices 51
A Resultados de Cada Dispositivo 52

1 INTRODUÇÃO

Este capítulo tem como objetivo realizar uma introdução ao leitor sobre o assunto abordado neste trabalho. Na Seção 1.1 são apresentados algumas considerações preliminares que auxiliam o leitor na compreensão da proposta deste trabalho. A Seção 1.2 exibe qual o problema de pesquisa abordado. O objetivo deste trabalho é apresentado na Seção 1.3. Por fim, na Seção 1.4 é mostrado as contribuições que este trabalho oferece.

1.1 Considerações Preliminares

O mercado de jogos digitais movimenta anualmente bilhões de dólares no mundo. Somente em 2018, movimentou-se 89 bilhões de dólares, com uma taxa de crescimento projetada de 6,3% ao ano. A importância dos jogos digitais não se limita apenas ao entretenimento, geração de emprego e renda, mas também se estende à promoção da inovação tecnológica com aplicações voltadas a diferentes contextos. Nesse sentido, jogos tipicamente percorrem diferentes setores, como arquitetura e construção civil, áreas da saúde, educação, treinamento e capacitação, entre outros (LEMES et al., 2012).

A construção de um jogo envolve uma série de etapas executadas por pessoas de diferentes papéis a fim de produzir artefatos que, somados, formam um jogo digital (BITTENCOURT; OSÓRIO, 2006). Os profissionais envolvidos compreendem usualmente programadores, artistas gráficos (2D/3D), *game designers* (projetistas de jogos), músicos e sonoplastas, testadores e redatores. Este grupo multidisciplinar de pessoas trabalham usualmente em um processo iterativo e incremental que culmina na entrega de um jogo digital. Tais fases são usualmente organizadas em: pré-produção, produção e pós-produção. A complexidade técnica envolvida na construção de jogos incentivou o surgimento de ferramentas que pudessem acelerar o processo como um todo. Essas ferramentas, comumente chamadas de motores de jogos[1], tiveram um papel significativo no surgimento e crescimento do mercado de desenvolvimento de jogos independentes.

Os desenvolvedores independentes, popularmente conhecidos por *indies*, são tidos como aqueles que não possuem o apoio financeiro de grandes estúdios. Em contrapartida, eles possuem total liberdade para conceber o jogo à sua vontade. Tipicamente, os *indies* constroem os jogos com considerável limitação de recursos financeiro e humanos (CASTRO et al., 2015). Jogos como `Angry Birds`, `Limbo` e `Fez` são exemplos de títulos criados por desenvolvedores independentes que alcançaram grande sucesso de mercado. `Angry Birds` foi desenvolvido com o motor de física

[1] Ambiente de desenvolvimento feito para desenvolver jogos.

`Box2D`[2], `Limbo` foi construído com o motor `Unity`[3] e para construir o `Fez` utilizou-se o motor XNA[4].

Segundo Lipkin (2012), a criação e disponibilização de ferramentas de acesso gratuito e a popularização dos mecanismos de distribuição digital de jogos, como a `Steam`[5], contribuíram significativamente para o sucesso dos pequenos desenvolvedores. De uma forma geral, os motores gratuitos de jogos têm sido um elemento chave para a popularização e crescimento do desenvolvimento independente, tal como pode ser visto nas listas de jogos desenvolvidos com ferramentas como `Unity`, `Cocos2d-x` e `Monogame` (UNITY, 2005; COCOS2D-X, 2008; MONOGAME, 2009).

Os motores proveem uma estrutura comum e um conjunto de utilitários que padronizam, facilitam e agilizam a produção de jogos (COSTA, 2014). Eles permitem criar jogos e aplicações multimídia de forma rápida e ágil, reduzindo drasticamente o esforço de criação e programação necessárias por parte de programadores, *designers* de jogos e artistas. Os motores podem se apresentar de duas formas: como um *framework*[6] ou um ambiente integrado. A escolha de um motor para a produção de um jogo é de extrema importância, pois pode significar a diferença entre entregar o jogo finalizado ou remover recursos por falta de tempo ou complexidade em demasia. Usualmente, um dos principais critérios que guia tal escolha é a *performance*, ou seja, o desempenho do motor, porém nem sempre este é o critério absoluto.

A escolha de um motor que melhor se encaixe a um projeto envolve alguns critérios como *performance* (desempenho do tempo de execução de um programa) e consumo de memória e usualmente é necessário realizar estudos e análises comparativas de motores sobre tais critérios. Para auxiliar nessa decisão, aplicativos de *benchmark*[7] são comumente utilizados.

1.2 Problema de Pesquisa

A escolha de um motor é usualmente um dos passos iniciais no processo de produção de jogos. Usualmente, a escolha do motor vem após as discussões de possibilidades de *design* do jogo ou projeto. Os critérios para fazer tal escolha são normalmente centrados em *performance* que é, por vezes, a questão técnica chave no desenvolvimento de jogos (BLOW, 2004). De modo geral, a escolha prematura de um motor de desenvolvimento pode gerar problemas em etapas posteriores. Por exemplo, quando se está no início de um projeto de jogos, a complexidade das cenas ainda é bastante pequena. A medida que o projeto avança, o mundo do jogo se torna gradativamente mais complexo e é nesse momento que tendem a surgir problemas de performance, tanto em termos de uso de CPU/GPU quanto em consumo de memória. O custo de migrar o jogo em desenvolvimento para outro motor é muito alto, muitas vezes forçando o projeto a reiniciar. Tais problemas podem ser agravados se existir a pretensão de publicar o jogo para diferentes plataformas. Além disso, é incomum que

[2] <http://box2d.org/>
[3] <https://unity3d.com/pt/>
[4] <https://www.microsoft.com/en-us/download/details.aspx?id=20914>
[5] <https://store.steampowered.com/?l=portuguese>
[6] Estrutura de suporte que guia a construção de um jogo.
[7] Ferramentas utilizadas para avaliar *performance*.

a equipe possua recursos para implementar protótipos em diferentes motores a fim de avaliá-las para embasar a escolha da que será utilizada em todo o projeto. Lamarche (2014) teve problemas em relação a limite de memória durante o desenvolvimento do jogo *Republic Sniper* e foi obrigado a trocar de motor. Segundo o desenvolvedor, meses de trabalho foram perdidos, pois o código previamente escrito para o motor que foi substituído não pôde ser reaproveitado na nova ferramenta. Mesmo os *assets* (artefatos do jogo), como texturas, modelos e animações de personagens, objetos e cenários, geraram grande retrabalho para serem reaproveitados no novo motor.

Benchmarks são meios popularmente utilizados na indústria para avaliar a *performance* de jogos ou ferramentas em diferentes plataformas. Alguns jogos disponibilizam testes de *benchmark* para verificar a *performance* no computador do usuário, como por exemplo `Resident Evil 5`, `Final Fantasy XIV` e `Lost Planet 2`. Esses *benchmarks* embutidos em jogos são popularmente utilizados para dois propósitos: para verificar o desempenho do jogo no computador do usuário e para criar bancos de dados da *performance* de jogos em diferentes configurações de computadores a fim de ajudar usuários na compra de *hardware* para jogos. Na prática, a escolha de um motor é usualmente guiado por dois fatores: o sucesso de outros jogos já existentes que o utilizaram e a popularidade no meio dos desenvolvedores. O jogo DOOM (2016) é um exemplo, pois ajudou a promover o motor `IDTech6`.

Inserido no contexto apresentado, este trabalho tem como questão principal auxiliar na escolha de um motor a ser utilizado em um projeto. Para condução deste trabalho, a seguinte questão de pesquisa foi elaborada:

- **QP**: É possível construir aplicativos de *benchmark* que realizam testes similares de estresse sobre diferentes recursos de motores de jogos 2D de forma que auxilie na escolha do motor em um projeto?

1.3 Objetivo

O objetivo deste trabalho consistiu na construção e utilização de aplicativos de *benchmark*. Estes aplicativos permitiram a realização de testes de estresse sobre recursos de alguns motores de jogos 2D. Os aplicativos foram executados em vários dispositivos, que enviaram os dados coletados para um servidor. Esses dados foram utilizados para discussão e avaliação da performance dos motores selecionados.

1.3.1 Objetivos Específicos

- Definição de critérios de escolha dos motores;
- Definição de testes a serem realizados;
- Construção de aplicativos de *benchmark client-side*[8] em motores de jogos 2D selecionados;

[8] Também conhecido como *front-end*, refere-se às operações que são realizadas pelo cliente em relacionamento cliente-servidor.

- Construção de um *software server-side*[9] para recebimento e armazenamento de dados dos *benchmarks*;
- Coleta de dados de performance das execuções dos aplicativos;
- Construção de gráficos para visualização dos dados armazenados no servidor;
- Análise dos resultados dos testes.

1.4 Contribuições

Neste trabalho, foram construídos aplicativos de *benchmark* para testar e comparar diferentes motores de jogos 2D. A partir dos aplicativos de *benchmark* foram realizados testes de estresse, pelos quais foram coletados dados de performance e posteriormente utilizados para análise. O código fonte do aplicativo de *benchmark* e dos testes serão disponibilizados para uso posterior por outros desenvolvedores e pesquisadores através do `GitHub`[10]. Espera-se que as discussões abordadas neste trabalho fomentem a disseminação da cultura de produção de jogos e o emprego de tecnologias gratuitas e/ou de código aberto para o desenvolvimento de *software* de jogos. Além disso, os aplicativos de *benchmark* em si e a infraestrutura será de alta valia a desenvolvedores, pois servirão para avaliar a performance dos motores nos dispositivos alvos de seus projetos. Dessa forma, os ajudarão a selecionar o motor mais adequado às suas necessidades em termos de desempenho.

1.5 Organização do Texto

No Capítulo 1 foi introduzido o contexto da proposta, assim como o problema de pesquisa, os objetivos e as contribuições deste trabalho. Alguns conceitos referentes ao trabalho são descritos no Capítulo 2. Os trabalhos relacionados são discutidos no Capítulo 3. A metodologia utilizada é mostrada no Capítulo 4. Os resultados, bem como a construção dos aplicativos são exibidos no Capítulo 5 e, por fim, no Capítulo 6 são apresentadas as conclusões obtidas ao decorrer do desenvolvimento deste trabalho.

[9] Também conhecido como *back-end*, é um termo usado para designar operações que são feitas no servidor.
[10] <https://github.com/>

2 CONCEITOS

Este capítulo tem como objetivo descrever ao leitor alguns conceitos abordados neste trabalho, visando auxiliá-lo na compreensão do texto. A Seção 2.1 define melhor o conceito de motores de jogos. Na Seção 2.2 é mostrado como um *benchmark* funciona e por fim, na Seção 2.3 é apresentado o conceito de teste de estresse.

2.1 Motores de Jogos

O desenvolvimento de jogos não é considerado uma atividade trivial (PEREIRA et al., 2017). A complexidade inerente às técnicas oriundas de diversas áreas, em especial da matemática, física, lógica algorítmica e produção áudio visual, levam a construção de ferramentas que podem auxiliar os times de desenvolvimento em seus ofícios. Os motores de jogos são ferramentas que dão suporte ao desenvolvimento de jogos, permitindo ao desenvolvedor agilizar a produção e reaproveitar recursos previamente construídos (BITTENCOURT; OSÓRIO, 2006). Motores de jogos 2D proveem recursos aos desenvolvedores, tais como: gerenciamento de ciclo de vida de objetos de jogo, gerenciamento e carregamento de componentes, aplicação da física, *sprites*[1], animação, partículas, câmera, som, *tilemap*[2], utilitários de geometria e trigonometria. De uma forma geral, os motores de jogos podem se apresentar de duas formas: como um *framework* ou como um editor visual.

2.1.1 *Framework*

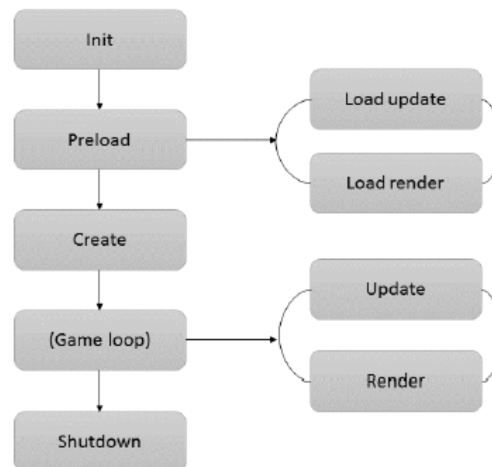
Uma das formas de apresentação de um motor de jogo é como um *framework*. Essencialmente, *framework* é uma coleção de utilitários e estrutura de classes que pode ser especializada para auxiliar na construção de uma aplicação. Para Pereira et al. (2017), *frameworks* são estruturas de classes que interagem entre si para um conjunto de aplicações de um domínio.

Existem inúmeros *frameworks* para desenvolvimento de jogos, como `Phaser`[3], `Cocos2d-x`[4], `Monogame`[5], LÖVE[6], entre outros. Todos esses *frameworks* possuem características principais semelhantes, como por exemplo a inicialização de recursos, métodos para carregar, atualizar e renderizar objetos do jogo e gerenciadores de memória, que ajudam a descarregar recursos inativos (COSTA, 2014).

A Figura 2.1 representa o ciclo de vida do *framework* `Phaser`, demonstrando como cada processo trabalha em conjunto para construir um ciclo de funcionamento. O método `Init` permite

[1] Objetos gráficos que fazem parte de uma cena.
[2] Popular no desenvolvimento de jogos 2D, consiste em construir o mundo do jogo como uma matriz de blocos retangulares.
[3] <https://phaser.io/>
[4] <http://www.cocos2d-x.org/>
[5] <http://www.monogame.net/>
[6] <https://love2d.org/>

Figura 2.1. Ciclo de Vida do Phaser.js



Fonte: An Introduction To HTML5 Game Development With Phaser.js

criar e preparar o motor. O `Preload` carrega os *assets* do jogo. O método `Create` constrói e configura os objetos do jogo. Já o `Game loop` tem a função de atualizar a lógica do jogo para cada objeto, no qual tipicamente, é chamado 60 vezes por segundo para cada objeto na cena. Por fim, o método `Shutdown` destrói objetos e libera recursos previamente alocados para o jogo.

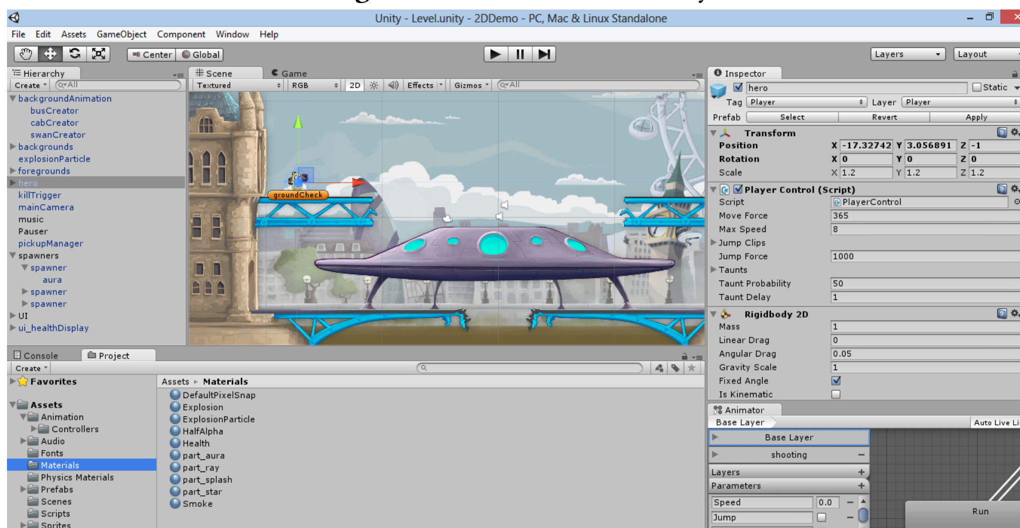
2.1.2 Editor Visual ou Ambiente Integrado

Outra forma de apresentação de um motor de jogos é como *Integrated Development Environment*(IDE). Assim como o nome sugere, a principal diferença para o *framework* é a de que as IDEs provêm características e ferramentas integradas e usualmente acessíveis pelo editor principal com o objetivo de agilizar o processo de desenvolvimento.

Um motor de jogo com IDE usualmente possui quatro componentes: *interação*, *comunicação*, *controle* e *visualização* (BITTENCOURT; OSÓRIO, 2006). O componente de *interação* é responsável pelo tratamento de eventos gerados por periféricos, como *mouse* e teclado. O componente de *comunicação* permite que o jogo esteja em rede com múltiplos participantes. O componente de *controle* é o que mantém a lógica do jogo e manipula os objetos. Por fim, o visualizador é quem apresenta as imagens por meio de dispositivos de vídeo, como um monitor.

A Figura 2.2 mostra como exemplo a IDE do motor `Unity`, na qual é possível observar várias características e ferramentas, muitas vezes não encontradas em *frameworks*, que auxiliam o desenvolvedor, como por exemplo, um menu com as características dos objetos da cena e um visualizador que permite arrastar objetos para a cena e gerenciá-los através do menu ao lado.

Figura 2.2. IDE do Motor Unity

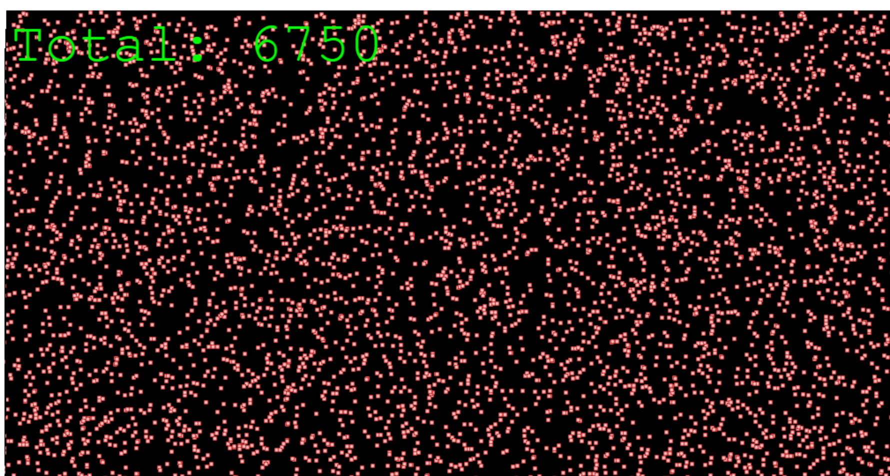


Fonte: (POLSINELLI, 2013)

2.2 *Benchmark*

Benchmarks são meios popularmente utilizados no mercado de *software* e *hardware* de computadores para avaliar comparativamente produtos similares, tendo por foco a execução de testes semelhantes em sistemas diferentes(RAMOS, 2008). No contexto de jogos, os aplicativos de *benchmark* são tidos como ferramentas utilizadas para aferir características, como a taxa de quadros e o consumo de memória, tendo como principal alvo a *performance* do jogo produzido.

Figura 2.3. *Benchmark* do *Framework* Phaser.js



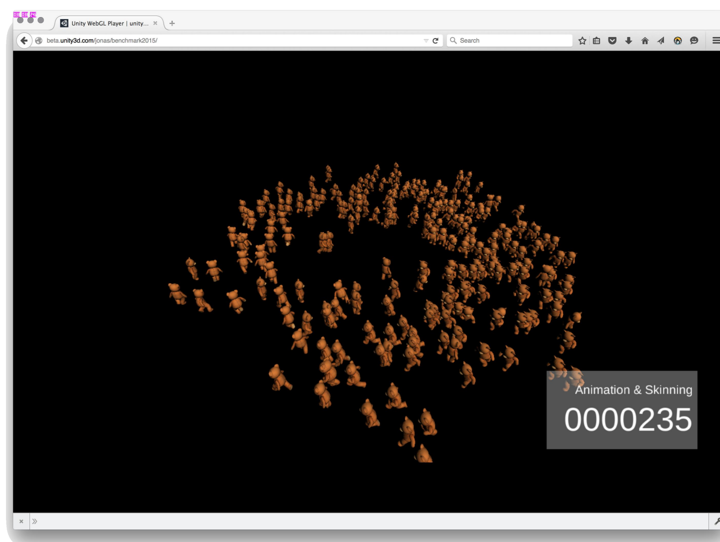
Fonte: (PHASER, 2013)

A Figura 2.3 mostra um teste de *benchmark* do *framework* `Phaser`, na qual vários pontos são renderizados na tela, aumentando gradativamente para que se possa acompanhar o comportamento do jogo no sistema a medida que a cena se torna mais complexa e, consequentemente, mais custosa de ser executada pelo sistema.

2.3 Teste de Estresse

Testes de estresse são testes intensos utilizados para determinar a estabilidade e também a *performance* de um sistema ou uma entidade, observando o comportamento dos sistemas de *hardware* e de *software* em condições de funcionamento além do habitual(CUNHA, 2010).

Figura 2.4. Teste de Estresse do *Benchmark* da Unity.



Fonte: https://beta.unity3d.com/jonas/WebGLBenchmark/ Acesso em: 15 abr. 2018.

A Figura 2.4 mostra um exemplo de teste de estresse utilizado no *benchmark* do motor `Unity`, no qual vários modelos 3D animados de personagens são renderizados na tela. A quantidade de modelos é gradativamente aumentada para aferir a capacidade máxima de funcionamento estável do *software* em execução e do *hardware*. Funcionamento estável refere-se a valores aceitáveis de taxa de quadros e consumo de memória. Isso depende do projeto que está sendo desenvolvido, por exemplo, em alguns casos, 60 de taxa de quadros seria o valor aceitável, em outros, seria 30.

2.4 Considerações Finais

Os motores de jogos auxiliam no desenvolvimento de jogos, trazendo vários benefícios aos desenvolvedores, oferecendo funcionalidades que podem ser prontamente usadas no contexto do jogo. Motores de jogos ajudam os times de desenvolvimento a publicar jogos mais rápido, reduzindo o tempo de desenvolvimento (COSTA, 2014). Os *benchmarks* ajudam desenvolvedores a detectar problemas de *performance* e a selecionar motores mais adequados aos seus projetos. Dessa forma,

neste trabalho é proposto e realizado a construção de aplicativos de *benchmark*, na qual foi possível coletar alguns dados de *performance* que posteriormente foram utilizados para análise e discussão, auxiliando na escolha de um motor mais adequado a um projeto.

3 TRABALHOS RELACIONADOS

Este capítulo discute alguns trabalhos encontrados com propósitos similares ou relacionados a esta pesquisa. Ele tem por objetivo apresentar um panorama do cenário na qual este trabalho está inserido.

3.1 *A Comparison of Game Engines and Languages*

A pesquisa de Jónsdóttir (2010) descreve a comparação de três diferentes motores, utilizando como metodologia a comparação quantitativa e qualitativa de características, incluindo:

- Pontos fortes e pontos fracos de cada IDE, segundo a perspectiva do autor;
- Tamanho dos arquivos gerados pelo motor, como o executável;
- Tempo de compilação e empacotamento do jogo em um programa executável;
- Consumo de memória durante o uso do jogo;
- Suporte disponível por parte dos desenvolvedores de cada motor.

Para guiar suas escolhas, Jónsdóttir (2010) leva em consideração seu conhecimento prévio de programação e o fato de ter pouco ou nenhum dinheiro, eliminando qualquer ferramenta que exija conhecimento avançado e que não ofereçam um período de experimentação suficiente para criar um jogo simples, já que o tempo de implementação que o autor possui é de um mês. Dessa forma, os motores selecionadas foram: `Microsoft XNA`, `Panda3D` e `Adobe Flash`. Outro fator que incentivou o autor a adotá-las foi o fato de que empregam linguagens de programação diferentes.

No trabalho, o autor realiza uma avaliação comparativa, construindo um mesmo jogo em três motores distintos. O jogo se chama *Mugwamps* e o objetivo consiste em tentar encontrar o *Mugwamp* clicando em locais da cena nos quais ele pode estar escondido. Para que a comparação fosse mais homogênea possível, as 3 versões usaram os mesmos *assets*.

Como conclusão, os resultados obtidos mostram que os motores `Microsoft XNA` e `Panda3D` são mais flexíveis e fáceis para um programador novato, além de se assemelharem bastante em alguns aspectos, apesar de usarem linguagens diferentes. O motor `Adobe Flash` mostrou-se um pouco mais difícil de ser utilizado devido sua IDE pesada e por possuir pouco suporte da comunidade de desenvolvedores que o utilizam.

A pesquisa de Jónsdóttir (2010) segue uma ideia similar deste trabalho, que é construir o mesmo jogo em diferentes motores. Contudo, Jónsdóttir (2010) se preocupa com a produtividade e recursos e não em aferir *performance* com testes de estresse. Por outro lado, alguns aspectos considerados pelo autor são relevantes ao presente trabalho, como por exemplo as variáveis de comparação.

3.2 *Comparison and Evaluation of 3D Mobile Game Engines*

O foco do trabalho de Pattrasitidecha (2014) é criar uma matriz de comparação, com o objetivo de ajudar desenvolvedores a escolherem o motor de plataforma móvel que melhor atenda seus propósitos.

Para a escolha dos motores são utilizado critérios, nos quais são verificados algumas características. O autor selecionou somente motores que permitem gerar jogos 3D para dispositivos móveis com casos de sucesso no mercado, deixando de lado todos os motores que não possuem essas características. Com os motores selecionados, foram realizados as comparações utilizando características, como por exemplo, bibliotecas de programação, usabilidade, plataforma de desenvolvimento e preço.

Através da matriz de comparação, o autor conclui que não existe um motor definitivo e que a melhor opção é aquela que mais se adequa ao propósito do desenvolvedor. Cada motor possui suas vantagens e desvantagens.

Os trabalhos de Pattrasitidecha (2014) e de Jónsdóttir (2010) realizam a comparação de motores de jogos, o que por sua vez, estabelece a relação destes trabalhos com o aqui proposto. A principal diferença é que o foco de Pattrasitidecha (2014) está em comparar os recursos que cada motor oferece, não se preocupando com a *performance*.

3.3 Considerações Finais

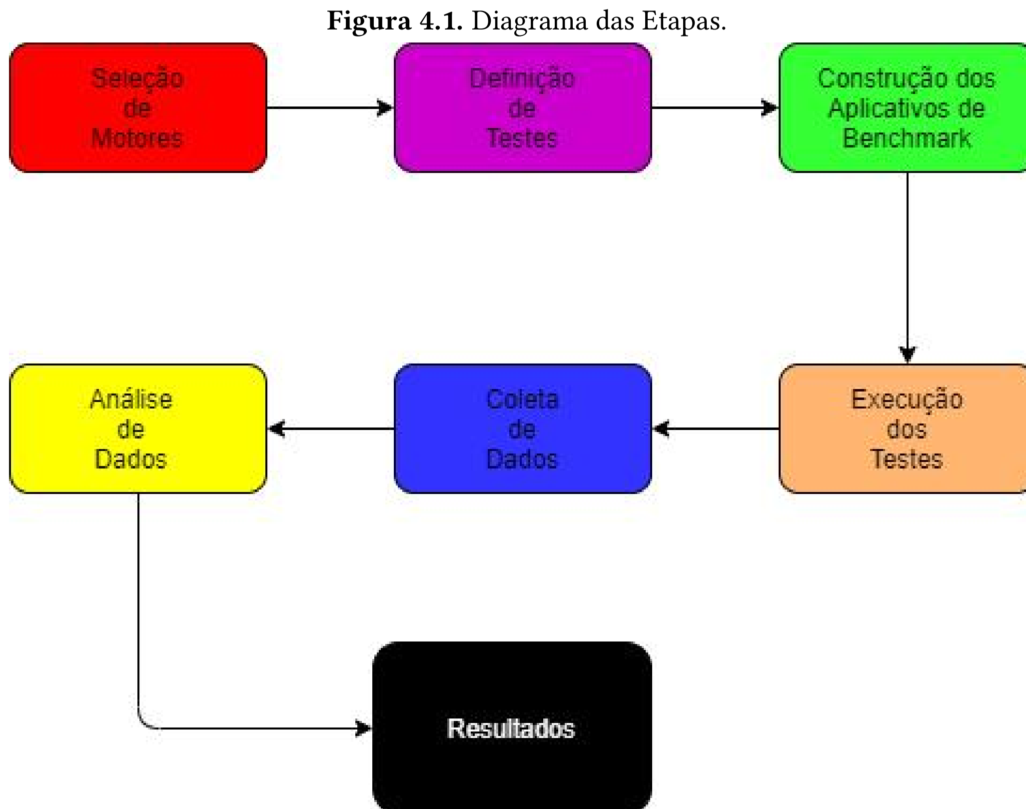
Os trabalhos abordados neste capítulo mostraram comparações entre motores de jogos, porém com foco voltado para uma análise dos recursos disponibilizados pelos motores, não se preocupando com o resultado de *performance* que cada motor oferece. Neste trabalho foi realizado a construção de aplicativos de *benchmark* para aferir a *performance* do resultado que pode ser gerado pelos motores analisados.

4 MATERIAIS E MÉTODOS

Este capítulo tem como objetivo apresentar a metodologia empregada para a construção e aplicação dos aplicativos de *benchmark*.

A realização do trabalho foi organizada em etapas de execução, como mostra o diagrama da Figura 4.1. Tais etapas são discutidas e detalhadas nas subseções seguintes.

Figura 4.1. Diagrama das Etapas.



4.1 Seleção dos Motores

O primeiro passo compreendeu a seleção dos motores a serem perfilados. Os critérios utilizados para essa seleção foram:

- Basear-se na experiência prévia do orientador, uma vez que este proveria o suporte para resolver problemas de implementação nos motores.
- Motores de alta popularidade no mercado.
- Motores gratuitos (de código aberto ou proprietários);
- Motores centrados (ou com bom suporte) em jogos 2D;
- Motores que permitam a construção de qualquer tipo de jogo;
- Motores de linguagens diferentes: considerou-se obter um indício da *performance* que é possível obter ao optar por desenvolver jogos com determinada linguagem.

Os candidatos inicialmente selecionados para construção dos testes de *performance* foram os motores **Phaser**, **LÖVE** e **Unity**. O primeiro passo foi realizar testes iniciais com o intuito de verificar a complexidade e o tempo necessário para a implementação dos aplicativos de *benchmark* nos três motoros. Com isso, optou-se por reduzir a abrangência do estudo deste trabalho a dois motores, além de concentrar os esforços naqueles centrados em jogos 2D e de código aberto. Portanto, os motores **Phaser** e **LÖVE** foram selecionados para a investigação de *performance*.

Para averiguar a popularidade dos motores, foram utilizados três critérios:

- Ocorrências dos nomes dos motores em listas de sites especializados em sugerir ferramentas de produção de jogos a desenvolvedores independentes;
- Quantidade de resultados encontrados em buscas no `Google`[1] sobre os motores;
- Relevância dos nomes dos motores em resultados de buscas no `Google` quanto associados às linguagens de programação que empregam.

No texto que segue são discutidos os resultados encontrados para cada um dos três critérios de popularidade.

Busca de *Sites* que Listam Motores

Uma das abordagens para atestar a popularidade dos motores selecionados foi constatar sua presença em listagens de *sites*[2] especializados em sugerir motores de jogos a desenvolvedores independentes. Para isso, foram realizadas pesquisas no `Google` utilizando os seguintes termos de busca: *"most used game engines"*, *"most popular 2d game engines"* e *"most used 2d game engines"*. Para cada listagem resultante, foram selecionados os cinco primeiros *sites* em ordem de relevância. Os resultados das buscas são mostrados abaixo. Os sites recorrentes nos resultados de mais de um termo de busca estão marcados em negrito.

- *"Most used game engines"*:
  - <https://www.gamedesigning.org/career/video-game-engines/>
  - <https://www.perforce.com/blog/vcs/most-popular-game-engines>
  - <https://www.gamasutra.com/blogs/MarcusToftedahl/20190930/350830/Which_are_the_most_commonly_used_Game_Engines.php>
  - <https://itch.io/game-development/engines/most-projects>
  - **<https://gamedevacademy.org/best-game-engines/>**
- *"Most popular 2d game engines"*:
  - **<https://careerkarma.com/blog/2d-game-engines/>**
  - **<https://www.slant.co/topics/341/ best-2d-game-engines>**
  - <https://thomasgervraud.com/best-2d-game-engine/>
  - <https://www.pcgamer.com/the-best-2d-game-engines/>
  - <https://www.godigitley.com/blog/best-2d-game-engines>

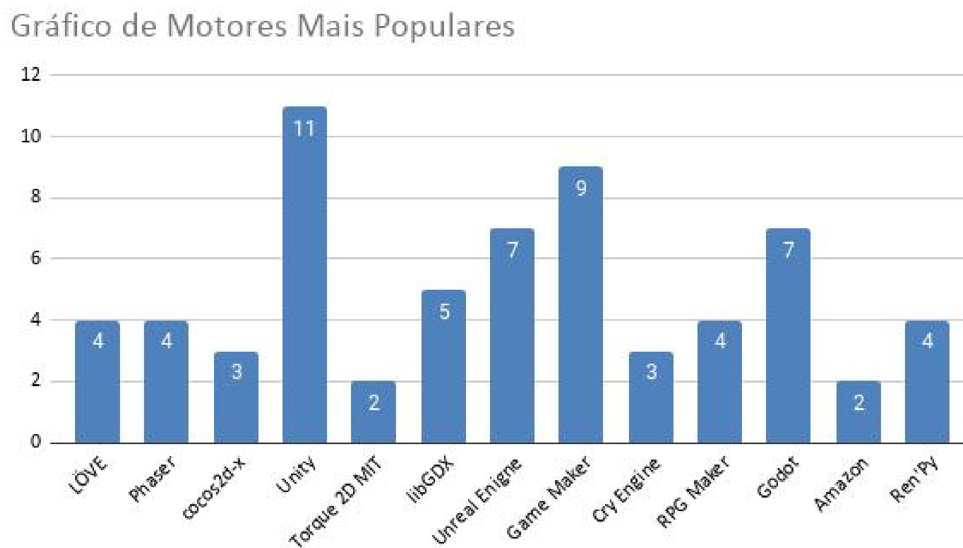
[1] <https://www.google.com/>

[2] Conhecido também como sítio, é um conjunto de páginas web.

- *“Most used 2d game engines”*:
  - **<https://careerkarma.com/blog/2d-game-engines/>**
  - **<https://thomasgervraud.com/best-2d-game-engine/>**
  - **<https://www.slant.co/topics/341/ best-2d-game-engines>**
  - <https://www.freecodecamp.org/news/what-2d-game-engine-to-use-for-your-next-game/>
  - **<https://gamedevacademy.org/best-game-engines/>**

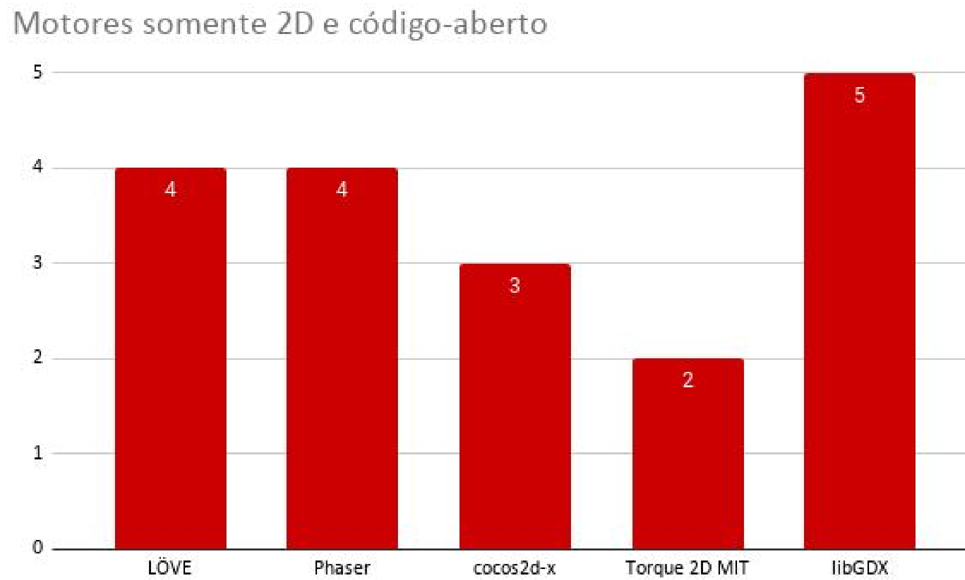
O gráfico abaixo exibe o número de aparições dos motores sugeridos nos *sites* acima. Neste gráfico, o eixo horizontal representa os motores sugeridos pelos *sites* e o eixo vertical representa o número de vezes que os motores foram sugeridos. As repetições encontradas nos sites de sugestões de motores, bem como, motores com apenas uma ocorrência, foram desconsiderados na montagem do gráfico.

Figura 4.2. Gráfico de Motores Mais Populares.



O gráfico da Figura 4.2 mostra que a Unity se destaca, tendo onze aparições nos *sites* de sugestões de ferramentas de jogos. Logo em seguida vem o motor Game Maker Studio com 9 aparições, seguido por Unreal e Godot, ambos com 7 aparições.

Figura 4.3. Gráfico de Motores 2D e de Código aberto.



A Figura 4.3 apresenta os motores centrados em 2D e de código aberto. A `lib`GDX mostrou-se mais popular, com 5 aparições. Em seguida, aparecem ambos motores selecionados, **Phaser** e **LÖVE**, com 4 aparições nos *sites* sugeridos. O motor `Ren'Py` foi desconsiderado porque é um motor específico para construção *visual novels* e não serve para criação de qualquer tipo de jogo 2D.

Número de Páginas Encontradas Pelo Nome do Motor

A segunda abordagem utilizada para averiguar a popularidade dos motores foi obter o número de páginas retornadas em pesquisas no `Google` utilizando o nome dos motores. Dessa forma, foram obtidos os números de páginas resultantes que o `Google` retornou ao buscar o nome de cada motor. Essa abordagem foi aplicada somente aos motores que mais se destacaram na abordagem anterior e aos motores selecionados. A Tabela 4.1 mostra os resultados obtidos nessa abordagem.

Tabela 4.1. Número de Resultados Por Motor

Motores	Número de Resultados
Game Maker Studio	133.000.000
Unreal Engine	86.300.000
Unity Game Engine	56.900.000
Godot	7.260.000
Löve	1.790.000
Phaser Game Engine	839.000

Fonte: Autoria própria

Através da utilização desta abordagem, foi possível observar que o motor `Game Maker Studio` se destaca, com 133 milhões de páginas retornadas na busca, sendo seguida pelo motor `Unreal` com 86 milhões e o motor `Unity`, com quase 57 milhões. Dentre os motores selecionados, o motor **LÖVE** mostrou-se mais popular com quase 2 milhões de páginas retornadas, já o motor **Phaser**, retornou quase 1 milhão de páginas.

Lista de Motores Por Linguagem

A última abordagem realizada para atestar a popularidade dos motores selecionados foi listar motores por linguagem em que os motores oferecem para desenvolver os jogos. Para isso, foram utilizados os termos de busca <*javascript game engine*> e <*lua game engine*>, como mostra a Tabela 4.2.

Tabela 4.2. Motores Por Linguagem

Javascript Game Engine	*Lua Game Engine*
Phaser	**LÖVE**
MelonJS	Corona
CraftyJS	Defold
BabylonJS	Moai
Impact	Solar2D

Fonte: Autoria própria

Os resultados contém a lista dos motores cujos *sites* oficiais foram apresentados como resultado da busca com os termos indicados. Como pode ser observado, o motor **Phaser** possui maior popularidade entre os motores que oferecem a linguagem *JavaScript* para o desenvolvimento de jogos. O mesmo vale para o motor **LÖVE** na linguagem *Lua*.

O objetivo desta primeira etapa da metodologia foi atestar a popularidade dos motores selecionados. Ambos os motores escolhidos possuem alta popularidade no quesito motores de jogos centrados em 2D. Vale ressaltar que em suas respectivas linguagens usadas para desenvolver jogos, no caso *javascript* e *lua*, ambos os motores são os mais populares.

4.2 Definição dos Testes

Para aferir a *performance*, foi determinado como critério encontrar recursos em comum, olhando para os recursos que cada motor oferece. Nos dois motores selecionados, **Phaser** e **LÖVE**, é possível observar alguns recursos em comum, como o uso de física, *tilemap*, sons e animações. No plano inicial, os *benchmarks* implementados teriam os seguintes testes:

1. *Sprites* em movimento, sem uso de física;
2. *Sprites* em movimento, com uso de física;
3. *Sprites* com animação;
4. *Sprites* com alpha[3] e transformações.
5. *Sprites* em movimento pelo uso de interpolação (*tweenings*[4]).
6. *Tilemap*[5], com uma camada e câmera;
7. *Tilemap* com várias camadas;
8. Partículas;
9. Objetos de texto;
10. Objetos de desenho vetorial.

Grande parte destes testes citados acimas já haviam sido implementados no **Phaser**. No entanto, durante a implementação no **LÖVE**, deparou-se com algumas dificuldades. De maneira geral, os recursos e facilidades que ambos motores oferecem são distintos. O motor **Phaser** possui recursos avançados já implementados, ao passo que o **LÖVE** é muito manual, apresentando apenas recursos básicos. **LÖVE** é essencialmente uma biblioteca que permite a captura de entradas do usuário, a execução de áudio, carregamento e exibição de imagens (texturas) com a aplicação de transformações. Por outro lado, o **Phaser** é um *framework* com diversos recursos pré implementados e prontos para uso. Além disso, a herança de classes do **Phaser** facilita consideravelmente as construções de jogos. Dessa forma, para haver equivalência de recursos nos testes propostos, foi necessário realizar a construção de vários recursos já disponíveis no **Phaser** sobre a biblioteca **LÖVE**.

Na escolha e implementação dos testes, o tempo foi um fator decisivo e, por esta razão, optou-se por diminuir o número de testes e evitar redundâncias a fim de deixar aqueles mais expressivos. Sendo assim, mesclou-se alguns testes, como os testes 1 e 2 com o teste 3 e 4. Preferiu-se também mesclar os testes de *tilemap*. Não existe um motor de física padrão no **LÖVE**, desse modo, retirou-se o uso de física dos testes. O teste de *tweening* foi removido pois não existe no **LÖVE** e seria muito trabalhoso sua implementação. Devido a esses obstáculos encontrados e também ao tempo, os testes de fato implementados foram:

[3] Alpha faz com que uma imagem se misture com o fundo tornando-a parcialmente transparente ou renderizando a imagem como 100% de transparência.

[4] *Tweening* é um processo de interpolação pelo qual um valor é gradativamente transformado em outro. Em animações, é comumente o processo de geração de quadros intermediários entre duas imagens para dar a aparência de que uma imagem evolui gradativamente para outra.

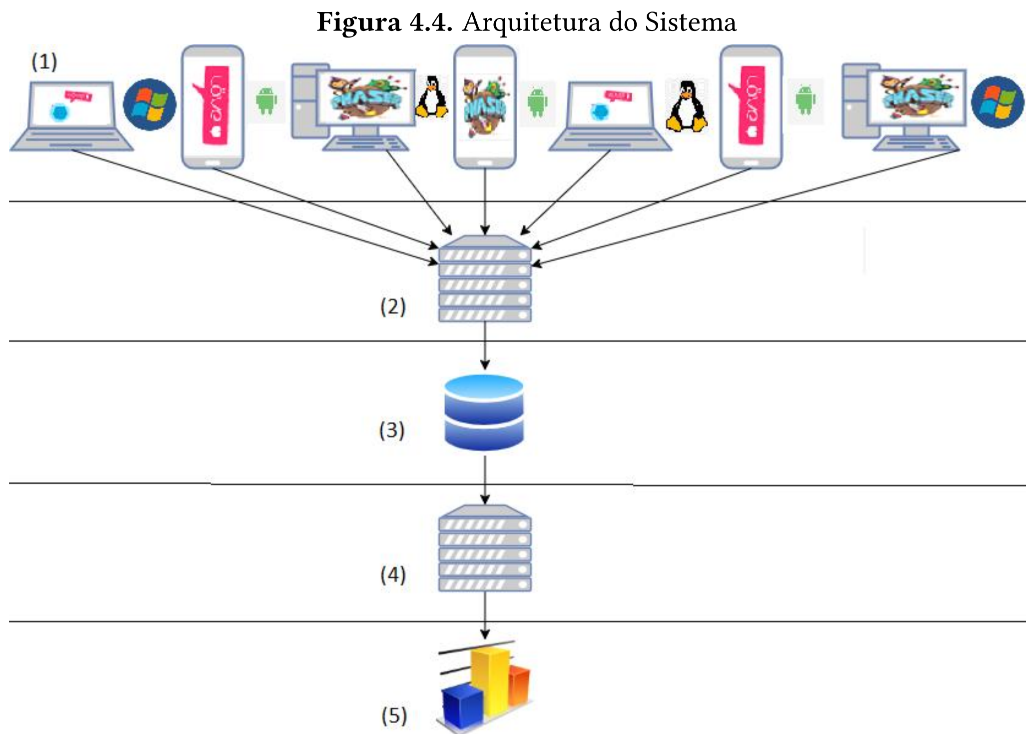
[5] *Tilemap* é uma técnica que consiste na construção do mundo ou mapa de níveis de um jogo a partir de pequenas imagens de formato regular chamada *tiles*

- *Sprites* em movimento sem alpha;
- *Sprites* em movimento com alpha;
- *Tilemap* com várias camadas e câmera.

4.2.1 Construção dos Aplicativos de *Benchmark*

Os aplicativos de *benchmark* foram desenvolvidos utilizando-se dos recursos oferecidos pelos motores selecionados e para duas plataformas: Desktop(Windows e Linux) e Dispositivo Móvel(Android).

Figura 4.4. Arquitetura do Sistema



A Figura 4.4 representa a arquitetura do sistema. (1) A primeira camada simboliza o lado do cliente, ou seja, são os usuários executando os aplicativos de *benchmark* em seus respectivos dispositivos e diferentes plataformas. (2) e (4) A segunda e quarta camada correspondem ao servidor, que tem como função o recebimento de dados coletados pelos aplicativos e também é através deste servidor que são gerados as visualizações. (3) A terceira camada caracteriza o banco de dados, utilizado para o armazenamento dos dados recebidos do servidor. (5) A quinta e última camada constitui na geração de gráficos para visualização dos dados coletados, para a análise e discussão referente a *performance* que os motores oferecem.

Para diminuir o viés que pode ser causado pela solução de implementação utilizada pelo autor em cada um dos motores, foram utilizadas algumas estratégias:

- Os códigos foram preferivelmente curtos. O foco é testar os recursos dos motores pela *Application Programming Interface* (API) disponibilizada. Portanto, evitou-se algoritmos complexos, quando possível;
- Os códigos utilizaram as soluções de implementação apresentadas na documentação, incluindo exemplos, dos motores. Dessa forma, usou-se soluções algorítmicas sugeridas pelos autores dos motores. Além disso, considerou-se que este é o tipo de solução que será provavelmente empregada em um ambiente real, por utilizadores dos motores.

4.3 Execução dos Testes

Com os aplicativos de *benchmark* construídos e os testes definidos, a próxima etapa foi a aplicação desses testes. Não existiu uma configuração de máquina alvo, pois a ideia foi testar em diversos dispositivos diferentes para verificar as diferenças entre os motores em um mesmo dispositivo. O público alvo da aplicação dos testes foram pessoas próximas ao autor. No total obteve-se resultados de 30 dispositivos, incluindo do autor.

4.4 Coleta de Dados

Usualmente, para aferir a *performance* de jogos, alguns dados são imprescindíveis, como *Frame Per Second* (FPS). Como mostrado na arquitetura do sistema, os dados coletados são enviados via protocolo HTTP para o servidor (*backend*). Como um dos objetivos deste trabalho consiste em avaliar a performance dos motores selecionados, os dados coletados a partir dos aplicativos de *benchmark* e dos dispositivos são:

- *Frame Per Second* (FPS);
- Quantidade de *sprites* na tela;
- Tempo corrido dos testes em milissegundos;
- Processador;
- Quantidade de Núcleos Lógicos;
- Quantidade de Núcleos físicos;
- Plataforma (Desktop ou Móvel);
- Sistema Operacional;
- Quantidade de memória RAM;
- Modelo Placa de vídeo.

4.5 Análise dos Dados

Como etapa final, foi feita a análise dos dados coleados. Os dados foram coletados e armazenados em um servidor. Assim que todos os dados foram coletados, gerou-se gráficos (e.g. progressão do FPS de cada motor) que posteriormente serviram para análise e discussão. Além disso, montou-se um

banco de dados de *benchmark*, que será útil a desenvolvedores, pois podem verificar o alcance de público-alvo que o jogo possuirá para cada motor. O desenvolvimento dos *benchmarks* e os resultados obtidos são discutidos no próximo capítulo.

5 RESULTADOS

Neste capítulo, o objetivo é apresentar, analisar e discutir todos os resultados obtidos, desde a construção dos aplicativos de *benchmark* até os resultados coletados na execução dos experimentos.

5.1 Tecnologias Auxiliares

No desenvolvimento dos aplicativos de *benchmark* foram utilizadas tecnologias auxiliares, escolhidas por popularidade e também por conhecimento prévio de seus funcionamentos. Tais tecnologias são enumeradas abaixo.

1. **Node.js**[1], ambiente de execução *server-side*, que permite criar aplicações *web* escaláveis e de alta *performance* utilizando `JavaScript`. Foi utilizado para criação do servidor e conexão com o banco de dados.
2. **NW.js**[2], aplicação em tempo de execução que deriva-se do `Node.js`, que permite criar aplicativos nativos para Windows, Linux e Mac, utilizando-se de tecnologias *web* e pacotes do Node.js. Foi aplicado no desenvolvimento da versão Desktop do **Phaser**.
3. **Apache Cordova**[3], plataforma de desenvolvimento móvel feito com HTML, HTML e `JavaScript`, no qual seu código pode ser compilado para diversas plataformas. Foi empregado para construção da versão móvel do **Phaser**.
4. **MongoDB**[4], banco de dados não-relacional de código aberto e orientado a documentos. Utilizado no armazenamento dos dados coletados pelos aplicativos.
5. **DigitalOcean**[5], provedor de serviços de infraestrutura de computação em nuvem. Aplicado para implantação do sistema.

5.2 Aplicativos de *Benchmark*

Apesar dos motores usarem linguagens de desenvolvimento diferentes, tentou-se ao máximo deixar ambos os códigos parecidos, ou seja, utilizando a mesma lógica de programação, para que não haja interferência na *performance*, consequentemente na análise dos dados. Por outro lado, em outros momentos foi necessário implementar recursos do **Phaser** não existentes no **LÖVE**. Por exemplo, no motor **LÖVE** não existe *tilemaps*, sendo assim houve a necessidade de implementá-lo. Nesta situação e em outras semelhantes, optou-se por utilizar tutoriais oficiais do motor, para garantir que as soluções construídas estejam de acordo com a documentação do motor em questão. A seguir,

[1] https://nodejs.org/en/
[2] https://nwjs.io/
[3] https://cordova.apache.org/
[4] https://www.mongodb.com/
[5] https://www.digitalocean.com/

nos Códigos 5.1 e 5.2 são mostrados exemplos de funções de ambos motores em suas respectivas linguagens, **Phaser** (JavaScript) e **LÖVE** (*Lua*).

```
1 launchSprite() {
2     this.sendDataToServer(this.test, this.game.time.fps, 1, this.sprites.
         countLiving(), this.getPlatformType(), this.getOs(), this.timer.ms,
         navigator.hardwareConcurrency, 1)
3
4     for (let i = 0; i < this.spriteSpawn; i++) {
5             this.sprites.add(new SpriteTest1(this.game, this.game.rnd.
                 integerInRange(0,1000), this.game.rnd.integerInRange(0,1000),
                  'player'))
6     }
7 }
```

Código 5.1. Função em JavaScript

```
1 function test01:launchSprite()
2   sendDataToServer(self.testName, tostring(love.timer.getFPS()), 2, self.size
       , getPlatformType(love.system.getOS()), love.system.getOS(), math.floor
       (self.count), love.system.getProcessorCount(), 1)
3
4   for i = 1, spriteSpawn do
5     local newSprite = SpriteBounce(love.math.random(0, 1000), love.math.
         random(0, 1000), love.math.random(1, 10), love.math.random(1, 10), 1,
          'player', 49, 72)
6     self.sprites[table.getn(self.sprites)+1] = newSprite
7     self.size = self.size + 1
8   end
9 end
```

Código 5.2. Função em Lua

Os códigos 5.1 e 5.2 representam a função *launchSprite*, que é responsável por enviar dados do *benchmark* para o servidor e criar as *sprites* que serão lançadas na tela. Para enviar estes dados, é chamado a função *sendDataToServer*, que recebe por parâmetro, respectivamente: <*título do teste*, *FPS*, *id* para identificar se o teste pertence ao **Phaser** ou ao **LÖVE**, *quantidade de sprites*, *plataforma do dispositivo*, *sistema operacional*, *tempo* em milissegundos, *número de núcleos* e *id* para identificar qual o número do teste>. Logo abaixo desta função, foi feito um laço de repetição para que seja desenhado mais de uma *sprite* na tela por vez, com o objetivo de diminuir o tempo dos *benchmarks*. Dentro deste laço é onde ocorre a criação das *sprites*, que para criá-las, no caso do *Phaser* é passado, respectivamente: <*contexto do jogo*, *posições X e Y* e o *id* da imagem utilizada na *sprite*>. Já no **LÖVE**, é passado respectivamente: <*posições X e Y*, *velocidades X e Y*, *escala*, *id da imagem*, *largura e altura da sprite*>.

De forma geral, para a construção dos aplicativos nas plataformas alvo, o **LÖVE** se mostrou mais prático , disponibilizando ferramentas e formas de construir seus projetos em diferentes plataformas, como Desktop e Móvel, enquanto para o **Phaser** foi necessário o auxílio de ferramentas externas.

Finalizado a construção *client-side*, ou seja, os aplicativos de *benchmark*, deu-se início ao *software server-side*, que possui a função de recebimento e armazenamento dos dados coletados pelos aplicativos. O Código 5.3 representa um exemplo de rota criada pelo servidor.

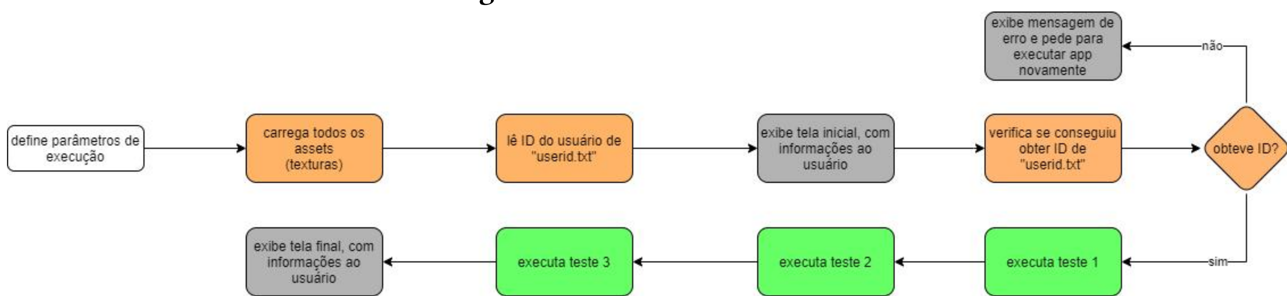
```
1 router.post('/sendResults', (req, res, next) => {
2   const { testName, fps, typeID, qtdSprites, platform, os, time,
       numLogicalCore, numTest, userId } = req.body;
3    try {
4     ResultsModel.create({ testName, fps, typeID, qtdSprites, platform, os,
         time, numLogicalCore, numTest, userId });
5     res.status(200).json({ 'status': 'ok' });
6    } catch (error) {
7      res.status(500).json({ 'stats': 'internal server error' });
8    }
9 });
10 }
```

Código 5.3. Exemplo de Rota

A rota exemplificada acima é responsável pelo envio de dados coletados pelos aplicativos, cujos campos esperados são respectivamente: *nome do teste*, *FPS*, *id para identificar o motor*, *quantidade de sprites*, *plataforma do dispositivo*, *sistema operacional*, *tempo em milissegundos*, *número de núcleos lógicos*, *número do teste* e *id do usuário*. Na linha 4 estes atributos são utilizados para criar uma tabela no banco e armazenar os dados.

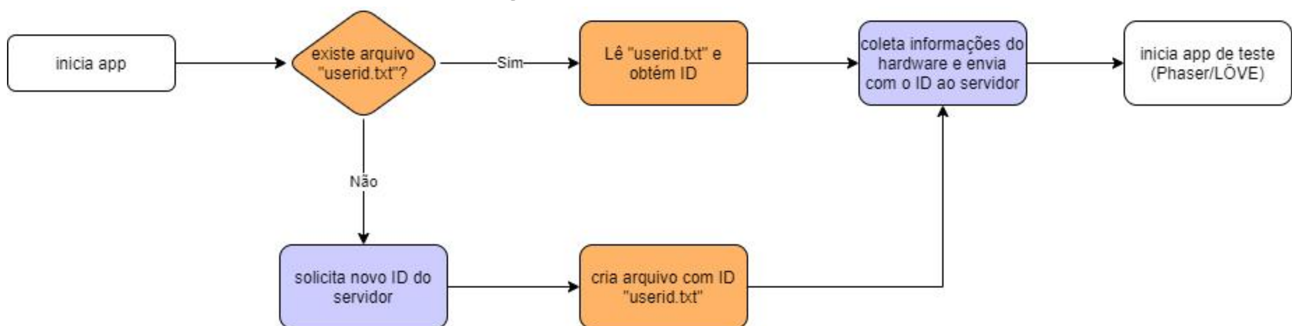
Os aplicativos de *benchmark* de ambas versões, desktop e móvel, possuem a mesma estrutura. São 3 aplicativos: o primeiro cujo denominação foi dada por *starter* é responsável pela coleta de dados relacionados as informações de *hardware* e também responsável por executar os outros dois aplicativos; o segundo aplicativo são os testes construídos no motor **Phaser** e o terceiro aplicativo contém os testes do motor **LÖVE**. Na Figura 5.1 é mostrado fluxogramas do funcionamento dos aplicativos.

Figura 5.1. Fluxo do Benchmark



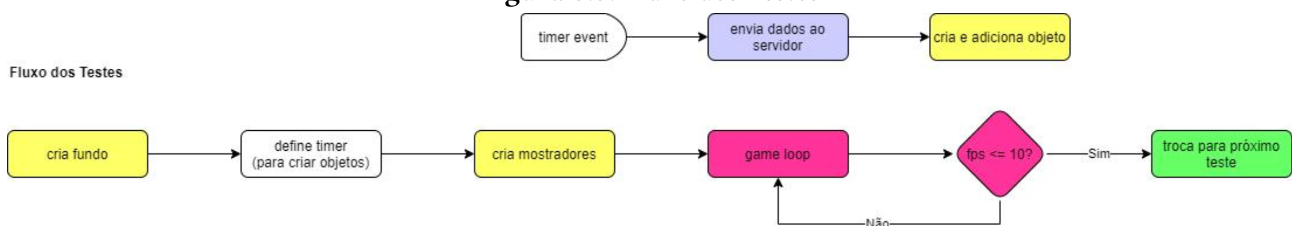
A Figura 5.1 representa o funcionamento do *benchmark*. Primeiro é definido alguns parâmetros de execução, como por exemplo altura e largura da tela. Depois é feito o carregamento de todos os *assets* (texturas) utilizadas, que no caso são os objetos e a imagem de fundo. Logo em seguida lê-se o arquivo chamado "userid.txt" e obtém um ID, exibindo a tela inicial com algumas informações. Caso o ID seja obtido com sucesso, é executado o primeiro teste e assim sucessivamente, caso contrário, é exibido uma tela com uma mensagem de erro pedindo para que o usuário execute novamente o aplicativo.

Figura 5.2. Fluxo do Starter



A Figura 5.2 simboliza o primeiro aplicativo, chamado de *Starter*. Ele é responsável pelo ID do usuário e criação do arquivo "userid.txt" e também pela coleta e envio de informações de *hardware*. Após isso, ele inicia os outros aplicativos, **Phaser** e **LÖVE**.

Figura 5.3. Fluxo dos Testes



A Figura 5.3 corresponde ao funcionamento dos testes. Assim que iniciados, é criado o fundo, definido um *timer* para criação dos objetos e criado os mostradores (nome do teste, quantidade de *sprites* na tela, tempo e FPS). Os objetos são criados a cada 100 milissegundos e os dados coletados também são enviados para o servidor dentro deste período de tempo. Quando o teste atual atinge a taxa de 10 FPS, o próximo teste é iniciado.

5.2.1 Dificuldades Encontradas e Soluções Utilizadas

Durante a construção dos aplicativos, surgiram dificuldades técnicas que não haviam sido previstas, por se tratarem de questões específicas das plataformas alvo. Esses problemas técnicos tomaram um tempo considerável para serem solucionados. Algumas dessas dificuldades e suas respectivas soluções encontradas são apresentados abaixo:

Obter as informações de *hardware* dos dispositivos

Não foi possível obter todas as informações de *hardware* dos dispositivos diretamente do código JavaScript em **Phaser** ou por Lua no **LÖVE**.

A solução foi criar um aplicativo de inicialização que obteria as informações *hardware* e que também iniciaria os aplicativos de *benchmark* do **Phaser** e do **LÖVE**. Para a versão Móvel, utilizou-se o *kit* de desenvolvimento nativo do `Android` (SDK). Já para a versão Desktop, utilizou-se o `Node.js`.

Executar os *benchmarks* dos diferentes motores em sequência

A princípio, a intenção era construir um *starter* que executaria em sequência os aplicativos de *benchmark* do **Phaser** e do **LÖVE**. Dessa forma, haveria garantia que o usuário executaria os dois aplicativos de *benchmark* e exigiria o mínimo possível de interação. Em um primeiro momento, descobriu-se que era possível fazer um aplicativo iniciar outro. Assim, o aplicativo *starter* poderia executar os aplicativos **Phaser** e **LÖVE**. Entretanto, não encontrou-se uma forma de encerrar estes aplicativo de *benchmark* após a conclusão dos testes, mesmo tentando soluções que exigiram a interação do usuário para retornar ao *starter* e dar sequência ao próximo aplicativo de *benchmark*.

Como não era possível encerrar cada aplicativo de *benchmark* após a conclusão dos testes, optou-se fazer dois *starters*, cada um ficando responsável de iniciar um *benchmark* diferente. Dessa forma, o usuário deve executar ambos os *benchmarks*, como por exemplo, iniciar o *starter-phaser* e assim que terminar os testes, fechar o aplicativo e iniciar o *starter-love*.

Relacionar as informações de *hardware* e as informações do *benchmark* ao mesmo usuário

O aplicativo *starter* específico de cada *benchmark* ficou responsável por obter as informações de *hardware* que seriam enviadas ao servidor. O problema era como garantir que as informações de *hardware* enviadas ao servidor pertenciam ao usuário que estava executando os *benchmarks*.

A solução encontrada foi utilizar um arquivo de texto para guardar o ID. Assim que coletar as informações de *hardware*, o *starter* busca o último ID que foi guardado no banco de dados, soma mais um, salva em um arquivo de texto chamado "userid.txt" este novo ID e envia para o banco de dados novamente. Os *starters* inicia os aplicativos de *benchmark*, **Phaser** e **LÖVE**, eles leem este arquivo que contém o ID e envia as informações do *benchmark* juntamente com este ID, dessa forma relacionando as informações de *hardware* coletadas ao usuário que está executando os *benchmarks*.

Salvar id do usuário em um arquivo de texto na versão móvel do LÖVE

Existem lugares na versão móvel que não é possível acessar. Foi possível criar o arquivo através do aplicativo *starter*, porém o aplicativo do **LÖVE** não conseguia acessar a mesma pasta, pois existiam problemas de permissões que não puderam ser resolvidos devido as particularidades das ferramentas que foram utilizadas para gerar o aplicativo. Além disso, existiam restrições que a própria tecnologia do motor do jogo impunha sobre o acesso a arquivos.

A solução encontrada foi ao invés de salvar em um arquivo de texto, salvar o ID no *clipboard*[6] do dispositivo. Assim que o *starter* coleta as informações de *hardware*, busca o útimo ID no banco de dados, soma mais um e salva-o no *clipboard* do dispositivo. Desse modo, quando inicia-se os aplicativos de *benchmark*, resgata-se este ID do *clipboard* e envia-o junto às informações do *benchmark*.

Nivelar os recursos necessários entre LÖVE e Phaser

Diferente do **LÖVE**, o motor **Phaser** possui muitos recursos já implementados e disponibilizados. Devido a este motivo, teve-se de implementar tais recursos no motor **LÖVE** manualmente, como por exemplo, *tilemap*, animação das *sprites* e os estados do jogo (telas).

A solução encontrada foi procurar tutoriais oficiais do motor que sugerissem uma solução padrão a ser adotada no **LÖVE** para construção dos recursos desejados, uma vez que são recursos comuns e de amplo emprego em jogos. Além disso, foi empregada a biblioteca Hump (RICHTER, 2015), sugeria na documentação do **LÖVE** para simular classes e estados de jogos.

5.3 Interpretadores das Linguagens

As linguagens que os motores **Phaser** e **LÖVE** utilizam para desenvolver jogos são, respectivamente, `JavaScript` e `Lua`. A principal diferença dessas linguagens são seus interpretadores. Lua possui seu próprio interpretador, já o `JavaScript` possui vários interpretadores, como por exemplo `V8`, `JavaScriptCore`, `SpiderMonkey`, entre outras, o que pode acabar interferindo na *performance*.

[6] Em português, área de transferência, que é um recurso utilizado pelo sistema operacional para armazenar pequenas quantidades de dados e transferir entre documentos ou aplicativos.

Tabela 5.1. Tecnologias JavaScript e Seus Interpretadores

	JavaScript *Engine*
Node.js	V8
Cordova	V8
NW.js	JavaScriptCore

Fontes: (NODEJS, 2009; ELHADY, 2020; JENSEN, 2015)

Na seção 5.1 foi apresentado as tecnologias auxiliares utilizadas para o desenvolvimento dos aplicativos. A Tabela 5.1 mostra algumas dessas tecnologias e seus respectivos interpretadores. Tanto o `Node.js` quanto o `Cordova` utilizam o `V8` como interpretador, que é o motor `JavaScript` que alimenta o `Google Chrome`[7]. Já o `NW.js` utiliza o interpretador `JavaScriptCore`, que é um *framework* utilizado no projeto `WebKit`[8] e também utilizado por outras aplicações, como o `Safari`[9].

Como parâmetro de comparação, encontrou-se dois *benchmarks* de performance que realizam uma comparação entre as linguagens JavaScript e Lua, visando apenas a velocidade de execução(GOUY, 2018)(SIRAIT, 2017). Em ambos *benchmarks*, foi utilizado o Node.js como tecnologia JavaScript, ou seja, o interpretador V8. A linguagem JavaScript se mostrou muito mais rápido. Por exemplo no *benchmark* de Sirait (2017), Lua demorou 6.87 segundos para executar, já o JavaScript demorou 2.49 segundos. No *benchmark* de Gouy (2018) foram realizados vários testes com códigos diferentes, porém em todos, JavaScript se mostrou mais veloz, em alguns chegando a ser 20 vezes mais rápido quando comparado ao Lua. Em contrapartida, os resultados obtidos a partir dos aplicativos construídos neste trabalho mostram o oposto: a superioridade do Lua.

5.4 Discussão e Análise

Houve um total de 30 dispositivos que executaram os aplicativos de *benchmark*, dos quais 13 foram Desktop e 17 foram Móveis (Android). Considerou-se três perspectivas para comparar os motores:

1. Ser superior ao outro a maior parte do tempo, ou seja, qual obteve *performance* superior por mais tempo;
2. Considerando a quantidade de *sprites* necessária para interromper os testes, qual alcançou o maior número de objetos na tela;
3. Até quantos *sprites* é possível manter uma *performance* ideal.

Estas perspectivas foram utilizados para comparar os motores com base no `Teste 1` (*Sprites* em Movimento e Sem Alpha) e o `Teste 2` (*Sprites* em Movimento e Com Alpha), que são os testes de *Sprites* em Movimento sem Alpha e *Sprites* em Movimento com Alpha, respectivamente. O `Teste 3` (*Tilemaps* Com Várias Camadas) foi analisado separadamente devido sua particularidade, que será mostrado ao decorrer do texto. Foi separado em alguns grupos para melhor análise e discussão

[7] https://www.google.com/intl/pt-BR/chrome/
[8] https://webkit.org/
[9] https://www.apple.com/br/safari/

dos resultados obtidos. Os resultados gráficos de todos os dispositivos estão disponibilizados no Apêndice A.

5.4.1 Dispositivos Desktop

Dentre os dispositivos Desktop, houve uma variação de processadores, sendo a maioria de fabricação Intel. Dos 13, foram 7 dispositivos `Intel` e 6 AMD, variando também suas gerações.

Teste 1: Sprites em Movimento sem Alpha

O `Teste 1` consiste em *sprites* se movimentando na tela e sem a utilização do alpha. Por dispositivos Desktop conseguirem manter uma *performance* melhor comparado a dispositivos Móveis, as *sprites* foram lançadas de 300 em 300 na tela.

A Tabela 5.2 exibe os resultados de todos os dispositivos Desktop para o Teste 1, considerando qual motor obteve *peformance* superior por mais tempo. Foi possível verificar que em todos os dispositivos o motor **LÖVE** foi superior. Em alguns casos o motor **LÖVE** ficou com o FPS maior em mais de 80% do tempo, como é o caso do dispositivo com processador AMD A12-9720P, onde o **LÖVE** foi superior 88.89% do tempo e houve um empate nos outros 11.11%. Um fato interessante é a diferença entre os sistemas operacionais de um mesmo dispositivo. O **Phaser** no i5-7300HQ Linux, não foi superior em nenhum momento, já neste mesmo dispositivo no Windows, conseguiu ser superior 40.23% do tempo.

Tabela 5.2. Teste 1 com Porcentagem do Tempo em que o Motor obteve *Performance* Superior

Processador	***Phaser***	**LÖVE**	**Empate**
i7-7500U	16,67%	**80,95%**	2,38%
Ryzen 3 2200G	21,25%	**73,75%**	5,00%
FX 6300	27,87%	**70,49%**	1,64%
A12-9720P	0,00%	**88,89%**	11,11%
i5-6600	0,00%	**73,86%**	26,14%
Ryzen 5 3600	0,00%	**71,85%**	28,15%
Ryzen 5 3500X	0,00%	**72,59%**	27,41%
i7-8700	32,14%	**64,29%**	3,57%
FX 8350	33,90%	**62,71%**	3,39%
i5-7300HQ(Linux)	0,00%	**72,97%**	27,03%
i5-7300HQ	40,23%	**51,72%**	3,66%
i7-4790	25,24%	**72,82%**	1,94%
i5-9400F	0,00%	**73,33%**	26,67%
TOTAL	0%	**100%**	0%

Fonte: Autoria própria

A Tabela 5.3 mostra os resultados dos dispositivos Desktop utilizando a perspectiva (2), ou seja, a quantidade de *sprites* na tela até atingir 10 FPS. Houve uma grande diferença entre os motores, chegando a ser quase de 100 mil *sprites*, como é o caso do dispositivo i7-8700, onde o Teste 1 no **Phaser** terminou em 33.300 *sprites* e no **LÖVE** terminou em 121.200 *sprites*.

Tabela 5.3. Teste 1 com Quantidade de *Sprites* na Tela

Processador	***Phaser***	LÖVE
i7-7500U	12600	**86100**
Ryzen 3 2200G	24000	**74100**
FX 6300	18300	**56700**
A12-9720P	10500	**36600**
i5-6600	26100	**96000**
Ryzen 5 3600	40200	**87300**
Ryzen 5 3500X	40200	**100200**
i7-8700	33300	**121200**
FX 8350	17400	**45900**
i5-7300HQ(Linux)	21900	**84600**
i5-7300HQ	25800	**82800**
i7-4790	30600	**94500**
i5-9400F	35700	**123000**
TOTAL	0%	**100%**

Fonte: Autoria própria

A Tabela 5.4 apresenta os resultados dos dispositivos Desktop para o Teste 1 com quantidade de *sprites* mantendo*Performance* ideal em 60 FPS. Utilizando como parâmetro de *performance* ideal 60 FPS, notou-se que para dispositivos Desktop, o **Phaser** consegue ser superior. Isso mostra que caso o desenvolvedor esteja construindo um jogo cujo características seja necessário ter reflexo, como jogos de ação, tiro, plataforma, o **Phaser** se torna uma melhor opção, pois consegue manter 60 de FPS por mais tempo. Ao todo, **Phaser** foi melhor em 53,84% dos dispositivos nesta perspectiva.

Tabela 5.4. Teste 1 com Quantidade de Sprites Mantendo*Performance* Ideal em 60 FPS

Processador	***Phaser***	LÖVE
i7-7500U	**2400**	600
Ryzen 3 2200G	**6300**	1500
FX 6300	**5400**	600
A12-9720P	2100	**5700**
i5-6600	6900	**16500**
Ryzen 5 3600	11100	**15600**
Ryzen 5 3500X	10800	**15900**
i7-8700	**8100**	900
FX 8350	**4200**	600
i5-7300HQ(Linux)	6000	**16200**
i5-7300HQ	**7800**	1500
i7-4790	**7500**	600
i5-9400F	9900	**20100**
TOTAL	**53,84%**	46,16%

Fonte: Autoria própria

Teste 2: Sprites em Movimento com Alpha

O Teste 2 consiste na mesma premissa do Teste 1, com a diferença da utilização do alpha, que é a transparência do objeto. Por se tratar de Desktop, também foi lançado de 300 em 300 *sprites* na tela.

A Tabela 5.5 exibe os resultados de todos os dispositivos Desktop para o Teste 2. Considerando qual motor obteve *peformance* superior por mais tempo, foi possível averiguar que, da mesma forma do Teste 1, o motor **LÖVE** foi superior em todos dispositivos. Nota-se que a diferença entre o número de casos que o **Phaser** vence em relação ao número de casos que o **LÖVE** vence, é menor que no Teste 1.

Tabela 5.5. Teste 2 com Porcentagem do Tempo em que o Motor obteve *Performance* Superior

Processador	***Phaser***	**LÖVE**	**Empate**
i7-7500U	19,57%	**78,26%**	2,17%
Ryzen 3 2200G	26,03%	**68,49%**	5,48%
FX 6300	31,15%	**67,21%**	1,64%
A12-9720P	0,00%	**92,31%**	7,69%
i5-6600	0,00%	**78,31%**	21,69%
Ryzen 5 3600	0,00%	**71,09%**	28,91%
Ryzen 5 3500X	0,00%	**70,97%**	29,03%
i7-8700	0,85%	**80,51%**	18,64%
FX 8350	5,66%	**84,91%**	9,43%
i5-7300HQ(Linux)	0,00%	**73,53%**	26,47%
i5-7300HQ	39,02%	**57,32%**	3,66%
i7-4790	27,38%	**70,24%**	2,38%
i5-9400F	2,59%	**73,28%**	24,14%
TOTAL	0%	**100%**	0%

Fonte: Autoria própria

A Tabela 5.6 mostra os resultados dos dispositivos Desktop com a quantidade de *sprites* na tela. Quando comparado ao Teste 1, é possível perceber que o recurso alpha possui um peso em relação a *performance*, pois a quantidade de *sprites* até chegar aos 10 de FPS de ambos motores foram menores.

Tabela 5.6. Teste 2 com Quantidade de Sprites na Tela

Processador	***Phaser***	LÖVE
i7-7500U	13500	**74700**
Ryzen 3 2200G	21600	**64200**
FX 6300	18000	**45600**
A12-9720P	7500	**28500**
i5-6600	24600	**86400**
Ryzen 5 3600	38100	**82800**
Ryzen 5 3500X	36900	**74400**
i7-8700	35100	**107100**
FX 8350	15600	**42900**
i5-7300HQ(Linux)	20100	**69900**
i5-7300HQ	24300	**65400**
i7-4790	24900	**80700**
i5-9400F	34500	**98700**
TOTAL	0%	**100%**

Fonte: Autoria própria

A Tabela 5.7 apresenta os resultados dos dispositivos Desktop para o Teste 2 com quantidade de *sprites* mantendo*Performance* ideal em 60 FPS. Diferente do Teste 1 nesta mesma perspectiva, para este teste o **LÖVE** foi superior, mostrando que com o recurso do alpha, ele é capaz de manter por mais tempo os 60 FPS. Nesta perspectiva, **LÖVE** se manteve superior em 61,53% dos dispositivos.

Tabela 5.7. Teste 2 com Quantidade de Sprites Mantendo*Performance* Ideal em 60 FPS

Processador	***Phaser***	LÖVE
i7-7500U	**2400**	-
Ryzen 3 2200G	**6000**	1500
FX 6300	**4800**	600
A12-9720P	1500	**4800**
i5-6600	6300	**14400**
Ryzen 5 3600	10800	**14100**
Ryzen 5 3500X	10500	**15900**
i7-8700	9000	**17100**
FX 8350	4200	**6600**
i5-7300HQ(Linux)	6000	**16200**
i5-7300HQ	**7200**	600
i7-4790	**6600**	600
i5-9400F	9600	**17700**
TOTAL	38,47%	**61,53%**

Fonte: Autoria própria

5.4.2 Dispositivos Móveis

Dentre os dispositivos Móveis, assim como nos Desktop, houve uma variação de marcas e modelos. Dos 17, foram 7 dispositivos Samsung, 4 Motorola, 3 Xiaomi, 2 Asus e um Huawei.

Teste 1: Sprites em Movimento sem Alpha

Por se tratarem de dispositivos Móveis, a quantidade de *sprites* lançadas na tela foi bem inferior aos dispositivos Desktop. Foram lançados de 60 em 60 *sprites*.

Tabela 5.8. Teste 1 com Porcentagem do Tempo em que o Motor obteve *Performance* Superior

Modelo	***Phaser***	**LÖVE**	**Empate**
Galaxy Note 10 Lite	**58,28%**	14,72%	27,61%
Redmi 5 Plus	0,00%	**97,87%**	2,13%
Zenfone 3	1,82%	**94,55%**	3,64%
Redmi 6A	0,00%	**95,56%**	4,44%
Moto G7 Play	1,45%	**84,78%**	13,77%
Moto G8 Power Lite	6,06%	**93,94%**	0,00%
Moto G4	0,00%	**95,65%**	4,35%
Redmi 6	0,00%	**96,08%**	3,92%
Galaxy S10	**41,42%**	38,08%	20,50%
Moto G6	5,45%	**87,27%**	7,27%
Huawei P30 Lite	19,64%	**64,29%**	16,07%
Zenfone 5	7,23%	**82,53%**	10,24%
Galaxy A01 Core	0,00%	**93,75%**	6,25%
Galaxy Note 10 Plus	13,25%	6,02%	**80,72%**
Galaxy S10 Plus	4,04%	**81,67%**	14,29%
Galaxy S9 Plus	12,24%	2,04%	**85,71%**
Galaxy S10e	18,26%	**61,64%**	20,09%
TOTAL	11,77%	**76,46%**	11,77%

Fonte: Autoria própria

A Tabela 5.8 apresenta os modelos de todos os dispositivos Móveis e a porcentagem do tempo em que cada motor se manteve superior considerando qual motor obteve *peformance* superior por mais tempo. Nos dispositivos Móveis o motor **LÖVE** foi superior novamente, porém não em todos, como nos Desktop. Em dois exemplos o motor **Phaser** conseguiu ser superior: no Galaxy Note 10 Lite, onde foi superior 58,28% do tempo e no Galaxy S10, onde foi melhor 41,42% do tempo. Além disso, houve mais dois dispositivos em que os motores foram pareados: no Galaxy Note 10 Plus, onde 80,72% do tempo o FPS dos motores ficaram iguais e no Galaxy S9 Plus, que o empate se manteve em 85,71% do tempo. Porém, no geral, nesta perspectiva o **LÖVE** manteve-se melhor em 76,46% dos dispositivos.

Tabela 5.9. Teste 1 com Quantidade de Sprites na Tela

Modelo	***Phaser***	LÖVE
Galaxy Note 10 Lite	9840	**15360**
Redmi 5 Plus	2880	**9060**
Zenfone 3	3240	**7200**
Redmi 6A	2640	**9240**
Moto G7 Play	8220	**18180**
Moto G8 Power Lite	1920	**7080**
Moto G4	2700	**9000**
Redmi 6	3000	**8940**
Galaxy S10	**21720**	14280
Moto G6	3240	**9600**
Huawei P30 Lite	6660	**18120**
Zenfone 5	9900	**18060**
Galaxy A01 Core	1860	**4080**
Galaxy Note 10 Plus	4920	**45600**
Galaxy S10 Plus	22200	**36180**
Galaxy S9 Plus	2880	**19560**
Galaxy S10e	**19200**	13080
TOTAL	11,77%	**88,23%**

Fonte: Autoria própria

A Tabela 5.9 apresenta os resultados dos dispositivos Móveis com a quantidade de *sprites* na tela. Foi possível notar que apesar do **Phaser** ter sido superior no dispositivo Galaxy Note 10 Lite de acordo com a perspectiva (1), ele não conseguiu ser superior nessa. Ele manteve seu FPS superior na maior parte do tempo porém terminou antes do **LÖVE**. Outro fato interesante é que no Galaxy S10e, apesar do **Phaser** ter sido pior na perspectiva (1), nessa ele mostrou melhor, atingindo 10 FPS com 19.200 *sprites*, enquanto no **LÖVE** chegou apenas a 13.080 *sprites.* Ao todo, **LÖVE** manteve sua superioridade em 88,23% dos dispositivos.

Tabela 5.10. Teste 1 com Quantidade de Sprites Mantendo*Performance* Ideal em 60 FPS

Modelo	***Phaser***	**LÖVE**
Galaxy Note 10 Lite	**2520**	2280
Redmi 5 Plus	60	**1320**
Zenfone 3	60	**420**
Redmi 6A	60	**1320**
Moto G7 Play	1260	**2880**
Moto G8 Power Lite	60	**1200**
Moto G4	60	**1320**
Redmi 6	60	**1320**
Galaxy S10	5100	**6240**
Moto G6	360	**1440**
Huawei P30 Lite	**1680**	1500
Zenfone 5	1620	**2700**
Galaxy A01 Core	60	**600**
Galaxy Note 10 Plus	4860	**7140**
Galaxy S10 Plus	4740	**9540**
Galaxy S9 Plus	2820	**3660**
Galaxy S10e	4680	**5580**
TOTAL	11,77%	**88,23%**

Fonte: Autoria própria

A Tabela 5.10 apresenta os resultados dos dispositivos Móveis para o Teste 1 com quantidade de *sprites* mantendo*Performance* ideal em 60 FPS. Diferente dos dispositivos Desktop, para manter 60 de FPS nos Móveis o motor **LÖVE** foi amplamente superior, mostrando que para jogos de ação para dispositivos Móveis, como por exemplo luta ou de tiro, seja mais viável o desenvolvedor utilizar o **LÖVE**.

Teste 2: Sprites em Movimento com Alpha

Assim como no Teste 1, foi lançado na tela de 60 em 60 *sprites*, com o diferencial do alpha como recurso.

Tabela 5.11. Teste 2 com Porcentagem do Tempo em que o Motor obteve *Performance* Superior

Modelo	*Phaser*	LÖVE	Empate
Galaxy Note 10 Lite	16,76%	**58,10%**	25,14%
Redmi 5 Plus	2,13%	**91,49%**	6,38%
Zenfone 3	3,33%	**90,00%**	6,67%
Redmi 6A	2,13%	**93,62%**	4,26%
Moto G7 Play	2,56%	**83,76%**	13,68%
Moto G8 Power Lite	0,00%	**97,67%**	2,33%
Moto G4	0,00%	**93,88%**	6,12%
Redmi 6	0,00%	**95,56%**	4,44%
Galaxy S10	1,36%	**92,86%**	5,78%
Moto G6	6,25%	**85,42%**	8,33%
Huawei P30 Lite	4,13%	**80,17%**	15,70%
Zenfone 5	17,54%	**59,65%**	22,81%
Galaxy A01 Core	0,00%	**94,29%**	5,71%
Galaxy Note 10 Plus	2,31%	**77,38%**	20,31%
Galaxy S10 Plus	36,20%	**53,39%**	10,41%
Galaxy S9 Plus	15,53%	29,81%	**54,66%**
Galaxy S10e	13,00%	**82,50%**	4,50%
TOTAL	0%	**94,11%**	5,89%

Fonte: Autoria própria

A Tabela 5.11 mostra os modelos de todos os dispositivos Móveis e a porcentagem de tempo em que cada motor foi superior no Teste 2. Considerando qual motor obteve *peformance* superior por mais tempo, neste teste, diferente do Teste 1, o motor **Phaser** não foi superior em nenhum dispositivo, o **LÖVE** mostrou-se amplamente superior. Com exceção do Galaxy S9 Plus, onde 54.66% do tempo ambos motores ficaram iguais.

Tabela 5.12. Teste 2 com Quantidade de Sprites na Tela

Modelo	***Phaser***	**LÖVE**
Galaxy Note 10 Lite	10680	**15060**
Redmi 5 Plus	2760	**8160**
Zenfone 3	1740	**6660**
Redmi 6A	2760	**8760**
Moto G7 Play	6960	**14880**
Moto G8 Power Lite	2520	**7080**
Moto G4	2880	**8040**
Redmi 6	2640	**7680**
Galaxy S10	17580	**30900**
Moto G6	2820	**8040**
Huawei P30 Lite	7200	**14400**
Zenfone 5	3360	**14820**
Galaxy A01 Core	2040	**3840**
Galaxy Note 10 Plus	23280	**28320**
Galaxy S10 Plus	**16440**	13200
Galaxy S9 Plus	9600	**14340**
Galaxy S10e	11940	**28320**
TOTAL	5,89%	**94,11%**

Fonte: Autoria própria

A Tabela 5.12 apresenta os resultados dos dispositivos Móveis com a quantidade de *sprites* na tela. Com esta perspectiva, foi possível observar que assim como nos dispositivos Desktop, o recurso alpha influenciou na *performance*, pois a quantidade de *sprites* para atingir os 10 FPS neste teste foi menor, ou seja, terminou mais rápido. O **LÖVE** foi superior novamente, com exeção do Galaxy S10 Plus, totalizando 94,11% dos dispositivos.

Tabela 5.13. Teste 2 com Quantidade de Sprites Mantendo*Performance* Ideal em 60 FPS

Modelo	***Phaser***	**LÖVE**
Galaxy Note 10 Lite	780	**2040**
Redmi 5 Plus	240	**1320**
Zenfone 3	60	**480**
Redmi 6A	60	**1140**
Moto G7 Play	1140	**2340**
Moto G8 Power Lite	-	**780**
Moto G4	60	**1140**
Redmi 6	60	**960**
Galaxy S10	1140	**1980**
Moto G6	420	**1260**
Huawei P30 Lite	**1380**	1320
Zenfone 5	1440	**2040**
Galaxy A01 Core	60	**480**
Galaxy Note 10 Plus	5280	**6660**
Galaxy S10 Plus	**4200**	1680
Galaxy S9 Plus	2520	**2580**
Galaxy S10e	780	**1320**
TOTAL	11,77%	**88,23%**

Fonte: Autoria própria

A Tabela 5.13 mostra os resultados dos dispositivos Móveis para o Teste 2 com quantidade de *sprites* mantendo*Performance* ideal em 60 FPS. Assim como no Teste 1, **LÖVE** foi melhor em 88.23% dos dispositivos, com a única diferença de a quantidade de *sprites* ser um pouco menor que no Teste 1, devido a utilização do alpha.

Dispositivos não monotônicos

Alguns dispositivos apresentaram relações não-monotônicas entre o número de *sprites* e os FPS, ou seja, nem sempre quanto mais *sprites*, menor o FPS ao longo do tempo. Estes estão em negritos nas tabelas apresentadas acima.

Tabela 5.14. Dispositivos não monotônicos

Modelo	***SoC-System-on-a-Chip***
Galaxy Note 10 Plus	Snapdragon 855
Galaxy S10 Plus	Exynos 9820
Galaxy S9 Plus	Snapdragon 845
Galaxy S10e	Exynos 9820

Fonte: Autoria própria

A Tabela 5.14 mostra os modelos dos dispositivos que tiveram comportamento não monotônicos e seus respectivos SoC's. Essas anomalias ocorreram somente em dispositivos da marca Samsung.

5.4.3 Teste 3: Tilemap Com Várias Camadas

O `Teste 3` possui a mesma premissa dos outros testes, porém ao invés de simples *sprites*, são *tilemaps*, cuja quantidade de camadas vai aumentando com o decorrer do tempo. *Tilemap* é o conjunto de pequenas imagens que compõe o cenário, em português chamado de ladrilho. Neste teste, ocorreu um equívoco no acréscimo de camadas por tempo, ficando nas seguintes proporções:

- **Desktop**: no **Phaser** foi lançado de 2 em 2 e no **LÖVE** de 500 em 500;
- **Móvel**: no **Phaser** foi lançado de 2 em 2 e no **LÖVE** de 5 em 5.

Devido a esta grande diferença no número de camadas no lançamento, não foi possível comparar graficamente os motores na versão Desktop para este teste, apenas para a versão Móvel. O motor **Phaser** já possuí o recurso de *tilemaps* e sua implementação é bastante custoso, pois provê maiores flexibilidades ao desenvolvedor, ao passo que a implementação realizada no **LÖVE**, apenas permite a construção de *tilemaps* simples.

Tabela 5.15. Teste 3 com Quantidade de *Sprites* na Tela

Dispositivo	***Phaser***	LÖVE
FX 6300(Desktop)	135	**526000**
i5-6600(Desktop)	95	**398500**
Redmi 5 Plus(Móvel)	15	**155**
Galaxy Note 10 Plus(Móvel)	45	**455**
i7-4790(Desktop)	325	**440000**

Fonte: Autoria própria

A Tabela 5.15 apresenta alguns exemplos isolados do `Teste 3`. Mesmo tendo a diferença entre os motores no acréscimo de camadas de *tilemap* com o passar do tempo, ainda sim o **LÖVE** foi superior, conseguindo carregar um maior número de *tilemaps*.

5.4.4 Considerações Finais

O **Phaser** é um motor que mostrou ter uma maior facilidade e praticidade para trabalhar por já ter vários recursos implementados, de modo que o **LÖVE** seja apenas uma biblioteca no qual deve-se implementar os recursos necessários. Esse pode ter sido um fator decisivo para a *performance* dos testes, pois quanto maior o número de recursos implementados, mais custoso se torna. Outro fator que pode ter ocasionado essa superioridade do motor **LÖVE** são os interpretadores das linguagens, pois Lua tem o seu próprio e JavaScript possui vários, fazendo com que o interpretador específico utilizado possa vir a interferir.

6 CONCLUSÕES

Para se construir um jogo, é necessário a união de pessoas de diversas áreas e essas pessoas seguem algumas etapas por padrão, que usualmente são: `pré-produção, produção e pós-produção` (BITTENCOURT; OSÓRIO, 2006). Devido a complexidade envolvida nesta construção, surgiram ferramentas para auxiliar e agilizar estes processos, conhecidos como motores de jogos.

Os motores permitem reduzir drasticamente o esforço de trabalho necessário para a construção de jogos. Devido a este fator, a escolha de um motor para a produção de jogos é de extrema importância, pois pode acontecer de ser necessário remover recursos do produto por falta de tempo ou complexidade.

Para a condução deste trabalho, a seguinte Questão de Pesquisa foi elaborada:

- **QP**: É possível construir aplicativos de *benchmark* que realizam testes similares de estresse sobre diferentes recursos de motores de jogos 2D de forma que auxilie na escolha do motor em um projeto?

Com esta Questão de Pesquisa em mente, foi-se construído os aplicativos de *benchmark*, utilizando os motores **Phaser** e **LÖVE**. Neles, realizou-se três testes de estresse, que apesar de usarem linguagem de programação para desenvolver diferentes, foi possível alcançar uma similaridade nos códigos.

O motor **Phaser** se mostrou muito mais fácil e prático de se trabalhar, pois diferente do **LÖVE**, ele é um *framework* com muitas funcionalidades já implementadas, como por exemplo gerenciamento de objetos, isso torna o trabalho consideravelmente mais fácil e mais produtivo. No entanto, mesmo ele possuindo essas características positivas, sua *performance* mostrou-se inferior quando comparado ao **LÖVE**.

Um exemplo de funcionalidade são os *tilemaps*, recurso utilizado no `Teste 3`. Como no **LÖVE** não possui tal recurso, teve de ser implementado manualmente. Os resultados mostraram que neste teste em específico, apesar da grande diferença do acréscimo de camadas, o **LÖVE** se mostrou mais eficiente. Dependendo das necessidades do jogo a ser implementado em **Phaser**, o desenvolvedor pode optar por construir o seu próprio gestor de *Tilemaps*.

De modo geral, o motor **LÖVE** mostrou-se uma grande superioridade em ambas versões, Desktop e Móvel. Apesar de sua *performance* ter sido inferior, o **Phaser** possui uma grande vantagem que é a facilidade e praticidade de uso. Outra vantagem interessante do **Phaser** é de que para dispositivos Desktop em testes mais simples, ele conseguiu manter uma *performance* ideal por mais tempo, tendo como alvo 60 FPS. Isso ocorreu no `Teste 1`, que é o teste mais simples entre os implementados.

Como trabalho futuro, é possível utilizar mais motores para construir aplicativos de *benchmark*, bem como implementar mais testes além dos três implementados neste trabalho. Também é possível ter mais perspectivas de comparação entre os motores, desssa forma, enriquecendo a análise.

A construção dos aplicativos em si é uma contribuição bastante significativa do trabalho, pois não foram encontrados esforços similares. A existência destes aplicativos pode ajudar os desenvolvedores a decidir qual motor é mais adequado para as características do seu projeto.

REFERÊNCIAS

BITTENCOURT, JR; OSÓRIO, FS. ***Motores para Criação de Jogos Digitais: Gráficos, Áudio, Interação, Rede Inteligência Artificial e Física***. São Leopoldo: Universidade do Vale do Rio dos Sinos, 2006.

BLOW, Jonathan. **Game development: Harder than you think**. *Queue*, ACM, v. 1, n. 10, p. 28, 2004.

CASTRO, Vinicius Oppido de et al. **Indie games: a atuação dos independentes no design de videogames**. Universidade Presbiteriana Mackenzie, 2015.

COCOS2D-X. *Made With Cocos2d-x*. 2008. Acesso em: 21 abr. 2018. Disponível em: <http://www.cocos2d-x.org/games>.

COSTA, Valter José Correia. ***LiNGS: framework de desenvolvimento de jogos multijogador em rede para dispositivos móveis***. Tese (Doutorado) — Instituto Politécnico de Leiria, 2014.

CUNHA, Simone Moreira. **Engenharia de Software - Uma Abordagem À Fase de Testes**. *Trabalho de Graduação*, Universidade Federal de Minas Gerais, p. 170, 2010.

ELHADY, Hady. ***Apache Cordova Development Tools***. 2020. Acesso em: 14 out. 2020. Disponível em: <https://instabug.com/blog/apache-cordova-development-tools/>.

GOUY, Isaac. ***The Computer Language Benchmark Game***. 2018. Acesso em: 04 nov. 2020. Disponível em: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/lua.html>.

JENSEN, Paul. ***What is NW.js?*** 2015. Acesso em: 14 out. 2020. Disponível em: <https://dzone.com/articles/what-is-nwjs>.

JÓNSDÓTTIR, Rósa Dögg. ***A comparison of game engines and languages***. Tese (Doutorado), 2010.

LAMARCHE, Jeff. **An Unreal Decision**. *Disponível em: <http://martiancraft.com/blog/2014/08/an-unreal-decision/>*, 2014.

LEMES, David O; TOMASELLI, Fernando Claro; CAMAROTTI, S. **A economia digital e o mercado de jogos para dispositivos móveis**. *SIMPÓSIO BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL*, v. 11, p. 1–5, 2012.

LIPKIN, Nadav. **Examining Indie's Independence: The meaning of"Indie"Games, the politics of production, and mainstream cooptation**. *Loading...*, v. 7, n. 11, 2012.

MONOGAME. *Made With Monogame*. 2009. Acesso em: 21 abr. 2018. Disponível em: <http://www.monogame.net/showcase/>.

NODEJS. *Nodejs*. 2009. Acesso em: 14 out. 2020. Disponível em: <https://nodejs.org/en/>.

PATTRASITIDECHA, AKEKARAT. **Comparison and evaluation of 3D mobile game engines**. *Chalmers University of Technology, University of Gothenburg, Göteborg, Sweden, Master Thesis, févr*, 2014.

PEREIRA, André Luiz Kuczner; RODRIGUES, Andrei Ricardo; AMARAL, Eliane Cristina; SABINO, Eliney; MUNIZ, Mário Sérgio de Almeida; ABE, Narumi. **Frameworks Para Desenvolvimentos De Jogos: Uma Abordagem Vantajosa No Desenvolvimento De Jogos Eletrônicos**. *Revista Gestão em Foco - Edição nº9*, 2017.

PHASER. ***Teste de benchmark da Phaser***. 2013. Acesso em: 09 jun. 2018. Disponível em: <http://labs.phaser.io/index.html?dir=game%20objects/blitter/&q=>.

POLSINELLI, Pietro. ***Why is Unity so popular for videogame development?*** 2013. Acesso em: 09 jun. 2018. Disponível em: <https://designagame.eu/2013/12/unity-popular-videogame-development/>.

RAMOS, Nelson Azoubel. **Avaliação e Comparação de Desempenho de Computadores: Metodologia e Estudo de Caso**. *Trabalho de Graduação*, Universidade Federal de Pernambuco, p. 60, 2008.

RICHTER, Matthias. ***hump.gamestate***. 2015. Acesso em: 10 jun. 2020. Disponível em: <https://hump.readthedocs.io/en/latest/gamestate.html>.

SIRAIT, Daniel. ***Benchmark Language***. 2017. Acesso em: 04 nov. 2020. Disponível em: <https://github.com/dns/benchmark-language>.

UNITY. *Made With Unity*. 2005. Acesso em: 21 abr. 2018. Disponível em: <https://unity.com/madewith?_ga=2.110616453.1430968648.1526869397-1148124853.1526869397>.

APÊNDICES

APÊNDICE A. RESULTADOS DE CADA DISPOSITIVO

Neste Apêndice será exibido todos os gráficos gerados a partir da execução dos aplicativos de *benchmark*. A linha azul representa a progressão do motor **Phaser** e a linha vermelha representa do **LÖVE**. Nos nomes das figuras estão as plataformas dos dispositivos e seus respectivos processadores. Os gráficos representam de forma sequêncial os Testes 1, 2 e 3.

Teste 1: Sprites em Movimento Sem Alpha

Teste 2: Sprites em Movimento Com Alpha

Teste 3: Tilemap Com Várias Camadas

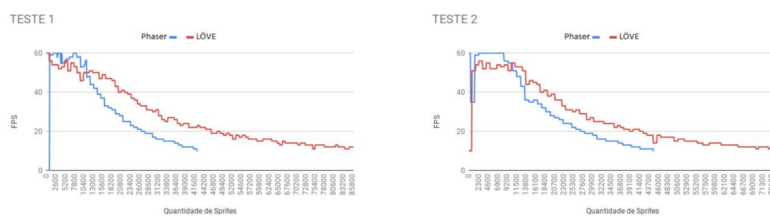


Figura A.1. Dispositivo Desktop - i7-7500U

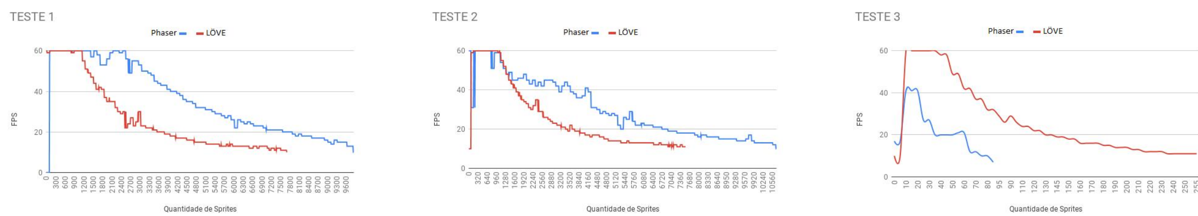


Figura A.2. Dispositivo Móvel - Galaxy Note 10 Lite

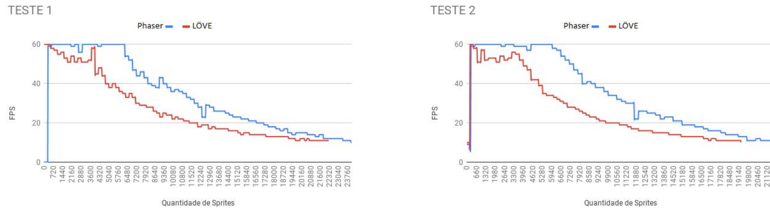


Figura A.3. Dispositivo Desktop - Ryzen 3 2200G

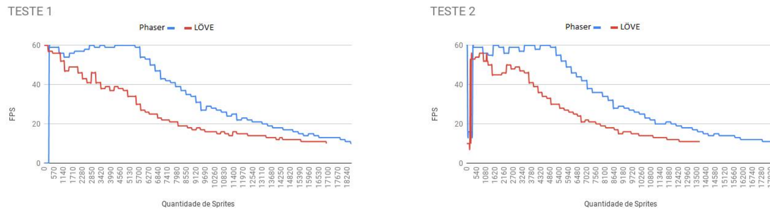


Figura A.4. Dispositivo Desktop - FX 6300

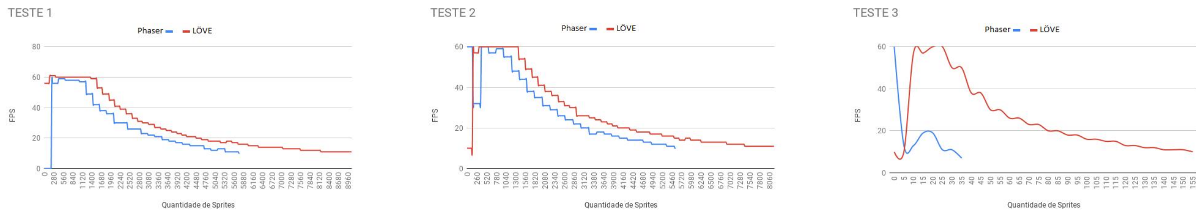


Figura A.5. Dispositivo Móvel - Redmi 5 Plus

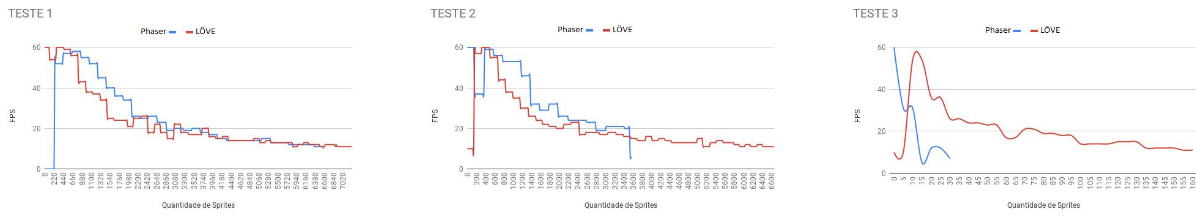


Figura A.6. Dispositivo Móvel - Zenfone 3

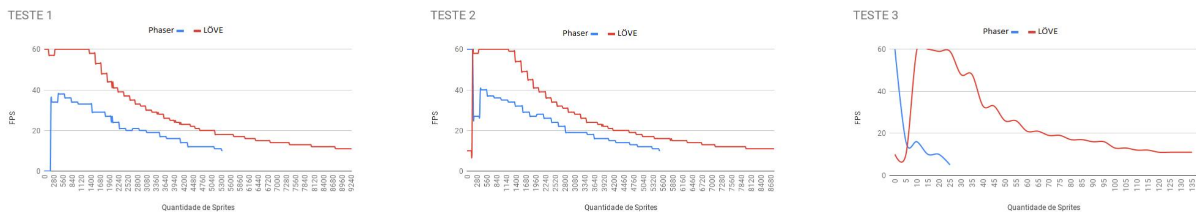


Figura A.7. Dispositivo Móvel - Redmi 6A

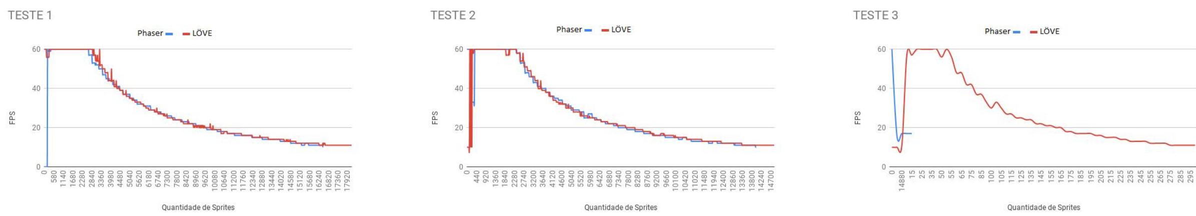


Figura A.8. Dispositivo Móvel - Moto G7 Play

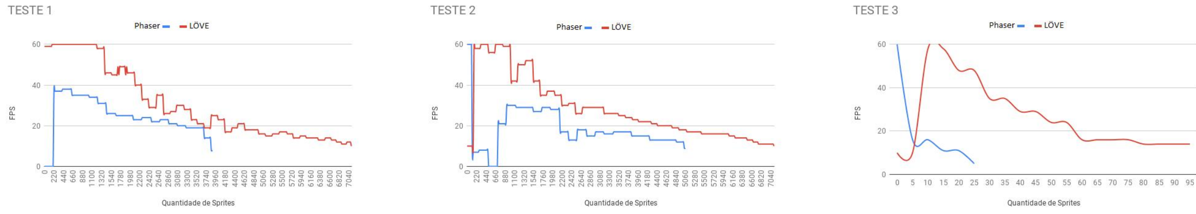


Figura A.9. Dispositivo Móvel - Moto G8 Power Lite

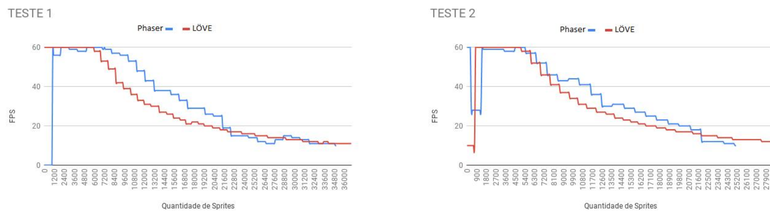


Figura A.10. Dispositivo Desktop - A12-9720P

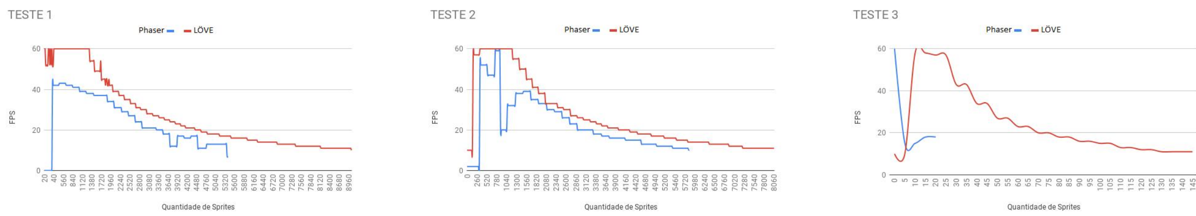


Figura A.11. Dispositivo Móvel - Moto G4

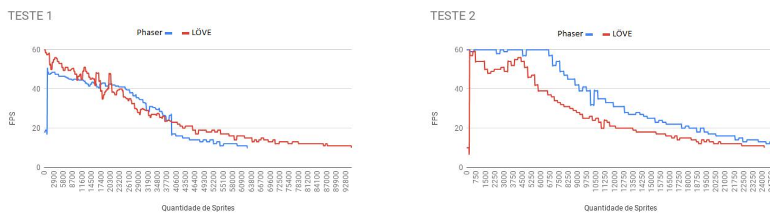


Figura A.12. Dispositivo Desktop - i7-4790

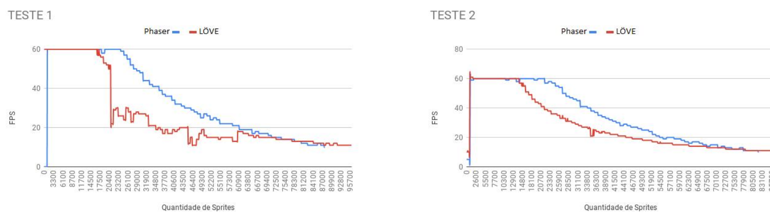


Figura A.13. Dispositivo Desktop - i5-6600

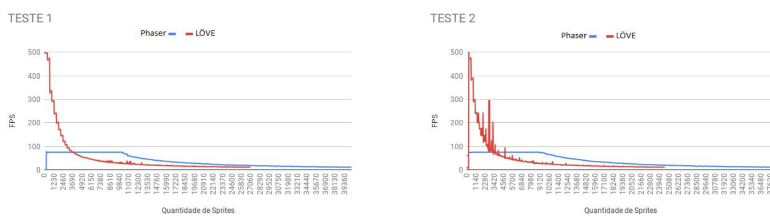


Figura A.14. Dispositivo Desktop - Ryzen 5 3600

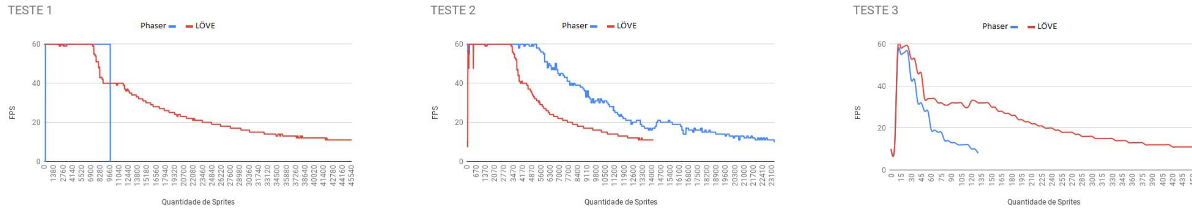


Figura A.15. Dispositivo Móvel - Galaxy Note 10 Plus

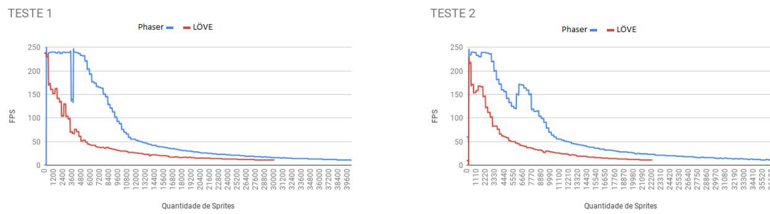


Figura A.16. Dispositivo Desktop - Ryzen 5 3500X

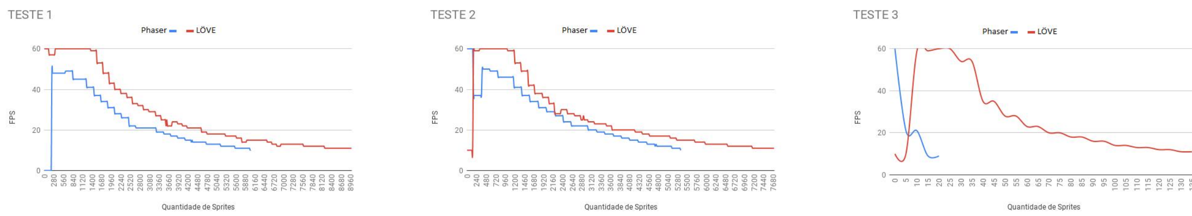


Figura A.17. Dispositivo Móvel - Redmi 6

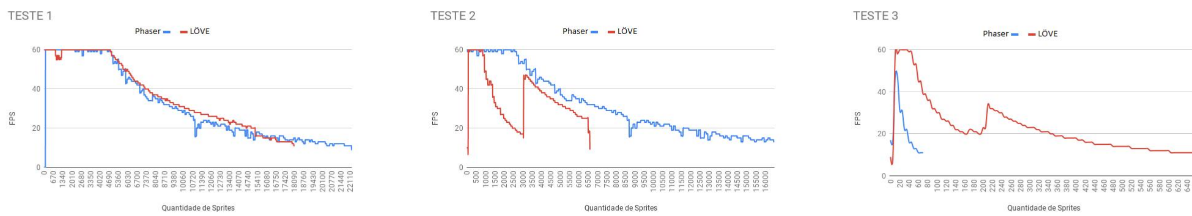


Figura A.18. Dispositivo Móvel - Galaxy S10 Plus

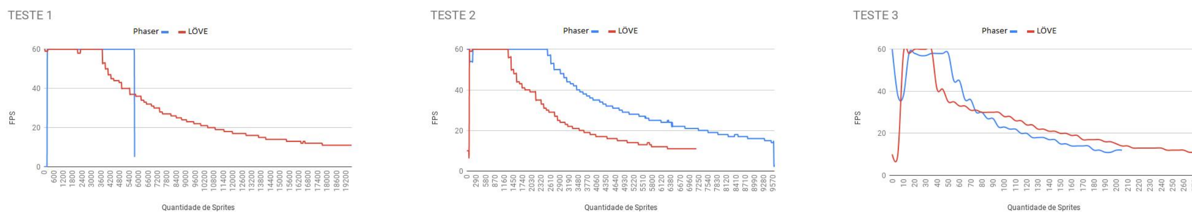


Figura A.19. Dispositivo Móvel - Galaxy S9 Plus

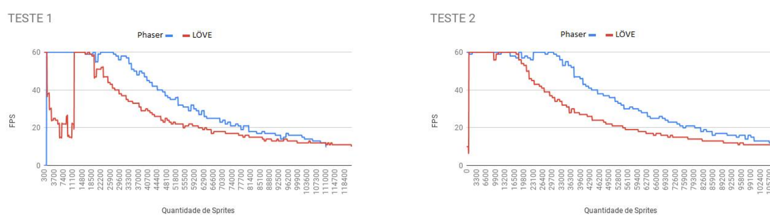


Figura A.20. Dispositivo Desktop - i7-8700

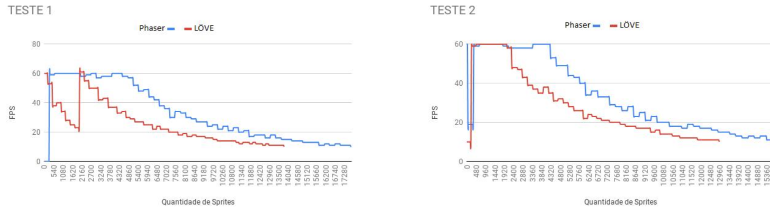


Figura A.21. Dispositivo Desktop - FX 8350

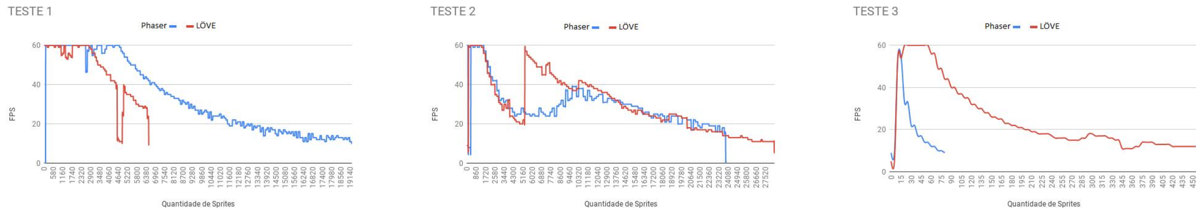


Figura A.22. Dispositivo Móvel - Galaxy S10e

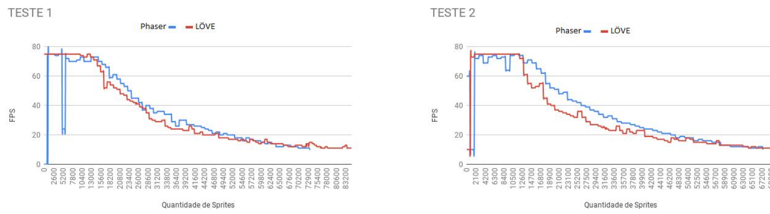


Figura A.23. Dispositivo Desktop - i5-7300HQ

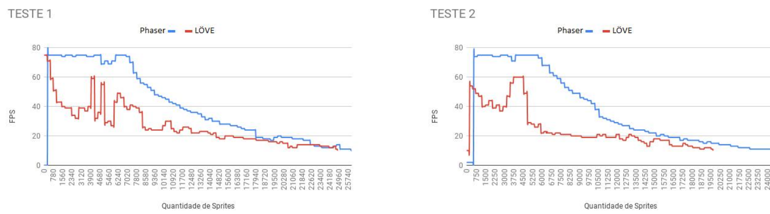


Figura A.24. Dispositivo Desktop - i5-7300HQ

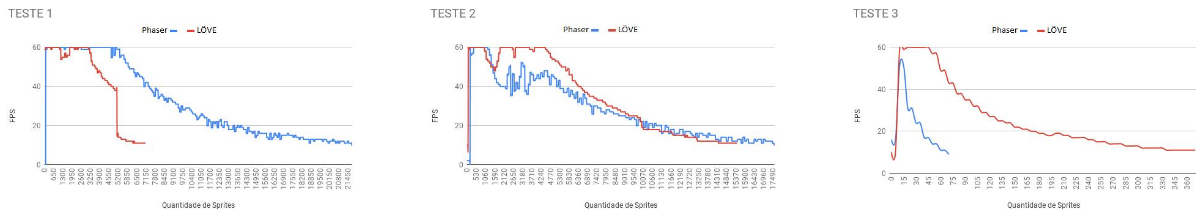


Figura A.25. Dispositivo Móvel - Galaxy S10

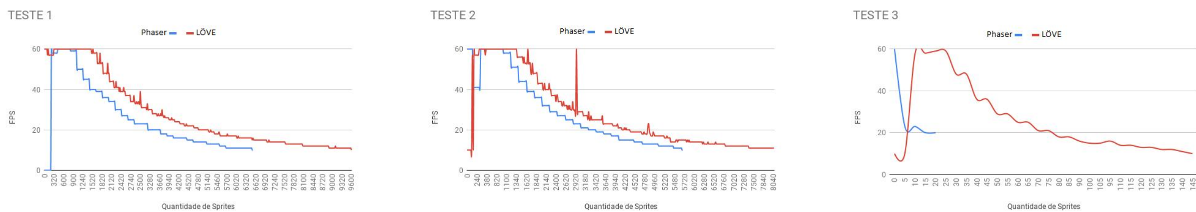


Figura A.26. Dispositivo Móvel - Moto G6

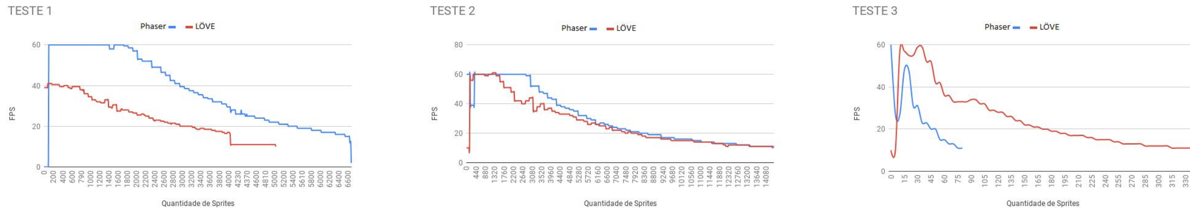


Figura A.27. Dispositivo Móvel - Huawei P30 Lite

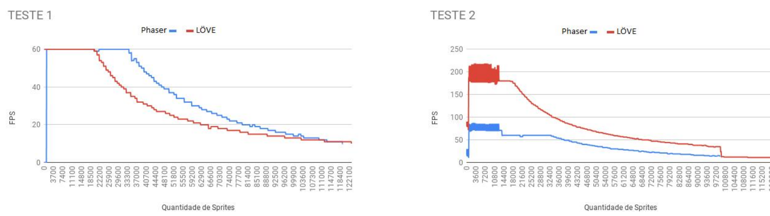


Figura A.28. Dispositivo Desktop - i5-9400F

Figura A.29. Dispositivo Móvel - Zenfone 5

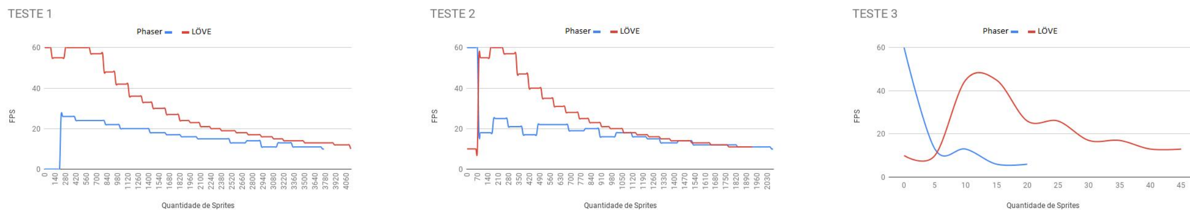


Figura A.30. Dispositivo Móvel - Galaxy A01 Core