UNIVERSIDADE FEDERAL DE SANTA MARIA
COLÉGIO POLITÉCNICO DA UFSM
CURSO DE SISTEMAS PARA INTERNET

João Senna de Andrade da Rosa

DESENVOLVIMENTO DE UM JOGO MULTIJOGADOR COM TECNOLOGIAS WEB

Santa Maria, RS
2021

João Senna de Andrade da Rosa

DESENVOLVIMENTO DE UM JOGO MULTIJOGADOR COM TECNOLOGIAS WEB

Trabalho de Conclusão de Curso apresentado ao Curso de sistemas para internet da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Tecnólogo em Sistemas para Internet**

Orientador: Prof. Dr. Rafael Gressler Milbradt

Santa Maria, RS

2021

João Senna de Andrade da Rosa

DESENVOLVIMENTO DE UM JOGO MULTIJOGADOR COM TECNOLOGIAS WEB

Trabalho de Conclusão de Curso apresentado ao Curso de sistemas para internet da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Tecnólogo em Sistemas para Internet**

Aprovado em 04 de fevereiro de 2021:

Rafael Gressler Milbradt, Dr.
(Presidente/Orientador)

Fernando Emilio Puntel, Msc. (UFSM)

Thales Nicolai Tavares, Tecg. (UFSM)

Santa Maria, RS

2021

AGRADECIMENTOS

A realização deste trabalho foi possível graças ao auxílio das pessoas ao meu redor, agradeço:

- *a minha família, em especial aos meus pais Erasmo e Rosangêla por todo suporte, e também ao meu irmão Pedro.*
- *ao meu orientador Rafael Milbradt que me proporcionou a melhor experiência possível na produção de um Trabalho de Conclusão de Curso.*
- *a Carolina Leal que me ajudou nas inúmeras revisões do presente trabalho.*
- *ao Denilson Ebling pela consultoria com os códigos que compõem o jogo.*
- *ao Nathan da Rosa pela ajuda com as artes que foram utilizadas no jogo.*
- *e a minha gata Renê que ouviu todos os ensaios das apresentações.*

Software engineering, of course, presents itself as another worthy cause, but that is eyewash: if you carefully read its literature and analyse what its devotees actually do, you will discover that software engineering has accepted as its charter "How to program if you cannot".

(EDSGER DIJKSTRA)

RESUMO

DESENVOLVIMENTO DE UM JOGO MULTIJOGADOR COM TECNOLOGIAS WEB

AUTOR: JOÃO SENNA DE ANDRADE DA ROSA
ORIENTADOR: RAFAEL GRESSLER MILBRADT

No passado, toda vez que se pensava em desenvolver um novo jogo, era preciso criar todas as ferramentas e tecnologias que seriam necessárias para sua concepção. Com o passar dos anos, e a chegada dos jogos com gráficos 3D, muitas tecnologias e propostas diferentes foram surgindo para dar suporte à criação dos mesmos, as *game engines*, que foram formas menos focadas em um sistema, e permitiam mais flexibilidade no desenvolvimento. Na atualidade, as tecnologias web têm sido respostas para muitas demandas no desenvolvimento de diferentes tipos de software, podendo, inclusive, fornecer soluções possíveis e viáveis para a criação de jogos que rodem e sejam escritos para navegadores. Desta forma, no presente trabalho serão implementados um jogo *multiplayer*, utilizando o *framework* Phaser, fazendo as vezes de uma *engine*, e um servidor implementado em Java, com o intuito de demonstrar a efetividade desse tipo de solução.

Palavras-chave: Framework. jogos. phaser. engine.

ABSTRACT

DEVELOPING A MULTIPLAYER GAME WITH WEB TECHNOLOGIES

AUTHOR: JOÃO SENNA DE ANDRADE DA ROSA
ADVISOR: RAFAEL GRESSLER MILBRADT

In the past, every time one thought about developing a new game, it was necessary to create all the tools and technologies that would be necessary for its design. Over the years, and the arrival of games with 3D graphics, many different technologies and proposals emerged to support the creation of it, as game engines, which were less focused on a system, and allowed more flexibility in development. Nowadays, web technologies have responses to many demands in the development of different types of software, and can even provide possible and viable solutions for the creation of games that are written for browsers. Thus, in the present work a multiplayer game will be implemented, using the framework Phaser, acting as an engine, and a server implemented in Java, in order to demonstrate the effectiveness of this type of solution.

Keywords: Framework. games. phaser. engine.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
DOM	Document Object Model
ES	EcmaScript
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
ID	Identificador
JS	JavaScript
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
PHP	PHP: Hypertext Preprocessor
TS	TypeScript
URL	Uniform Resource Locator

SUMÁRIO

1 INTRODUÇÃO

Segundo (WILLIAMS, 2017) o desenvolvimento de jogos eletrônicos já foi uma tarefa árdua. Os jogos eram desenvolvidos especificamente para um *hardware*, cada funcionalidade dependia completamente de qual aparelho iria executar o jogo, as capacidades eram exploradas até o limite da máquina, cada porção de recurso era muito valiosa. Eventualmente, trabalhar com recursos muito limitados, pode gerar efeitos inesperadamente positivos, e não só com soluções criativas, mas também com simples efeitos colaterais que se tornam parte fundamental do resultado final.

Ainda segundo o autor, Space Invaders, que foi desenvolvido por Tomihiro Nishkado e lançado no ano de 1978, é um jogo originalmente de *arcade*, que consiste em o jogador atirar da parte inferior da tela em naves que descem da parte superior. O objetivo é impedir que elas cheguem até embaixo, entretanto, a máquina que o rodava não tinha desempenho suficiente para processar tantos elementos diferentes na tela, o que gerava um resultado curioso. No início da partida, com a tela cheia de elementos, o jogo rodava devagar, fazendo com que fosse fácil atingir as naves inimigas, contudo, quando as naves diminuíam com o decorrer do jogo, os elementos na tela diminuíam, e como consequência, a velocidade aumentava, pois a máquina não estava mais sobrecarregada. Esse elemento de gradual acréscimo da dificuldade, foi completamente acidental, mas que acabou se tornando uma marca do jogo.

Existem também os casos em que as limitações forçam o desenvolvimento de soluções novas de tecnologia, que expandem os limites que estão impostos. Segundo (LEBEL, 2013), o jogo *Star Fox*, de 1993, da Nintendo, possuía dentro de seu cartucho um processador extra de 16-bit, funcionando como um acelerador gráfico. Ele era capaz de produzir polígonos, ou seja, possibilitava o Super Nintendo realmente produzir gráficos em 3D.

Hoje, esse cenário se mostra muito distinto. Os recursos de *hardware* existem em abundância para a maior parte das tarefas que são executas no dia a dia, possibilitando uma abordagem completamente diferente. Se no passado era necessário desenvolver todo um ecossistema de *hardware* e *software* para criar um jogo, no presente podemos utilizar tecnologias e recursos que já existem e têm outras finalidades. Para se desenvolver um jogo, as ferramentas atuais nos entregam uma gama tão ampla de possibilidades, que basta algumas poucas adaptações para que elas servirem perfeitamente ao proposito da criação de um jogo.

Anteriormente ao HyperText Markup Language 5 (HTML5), caso se quisesse executar mídia no navegador, era necessário se instalar algum *plugin* ou uma aplicação que pudesse lidar com esse tipo de conteúdo. Com jogos não poderia ser diferente, e a abordagem mais famosa e bem sucedida foi a utilização da tecnologia Adobe Flash para desenvolvimento de jogos 2D. Uma grande quantidade de jogos para navegador foi construída usando essa tecnologia. Esses jogos foram criados utilizando o Adobe Flash Professional, um *software* que permite a criação tanto de jogos como de vídeos. O Flash, mesmo sendo uma das tecnologias mais difundidas para criação de jogos de navegador, tinha fraquezas muito aparentes, como a necessidade de *plugins* para funcionar e o alto consumo de recursos, atrelado a um baixo desempenho, motivo pelo qual a Apple não dava suporte para a tecnologia. Jogos de Facebook utilizaram muito dessa tecnologia, com destaque para o *FarmVille*, um simulador de fazenda muito popular lançado em 2009.

Outra forma para se desenvolver jogos de navegador era o Java Applet, que são pequenas aplicações escritas em Java, ou em outra linguagem que possa ser compilada para Java bytecode. O Java Applet era carregado de uma página web e então era executado dentro da Java virtual machine (JVM), em um processo separado do navegador. À época, os Applets tinham um desempenho muito superior ao Java Script, e possibilitavam tecnologias como aceleração por *hardware* de gráficos 3D, porém da mesma forma que o Flash, essa tecnologia dependia da instalação de *plugins* e junto com o Flash, acabou por se tornar ultrapassada após a chegada do HTML 5. Como jogos feitos em Apple's são precursores ao Flash, não há exemplos marcantes como o FarmVille, porém, o mais comum era ser utilizado em jogos de *puzzle*.

1.1 OBJETIVOS

O objetivo central deste trabalho é desenvolver um jogo de videogame online multijogador para navegadores.

1.1.1 Objetivos específicos

Para alcançar o resultado proposto, os seguintes objetivos foram delimitados para nortear o trabalho.

- Identificar as limitações das tecnologias de desenvolvimento *web*.
- Delimitar os recursos necessários para o desenvolvimento.
- Explorar o uso do *web browser* para desenvolvimentos de jogos *multiplayer*.
- Construir o jogo utilizando as tecnologias propostas.

1.2 CONTEÚDO DESTE TRABALHO

O Trabalho de Conclusão de Curso está dividido em 03 (três) capítulos pertinentes aos objetivos do trabalho.

- Capítulo 1: introdução e contextualização sobre o desenvolvimento de jogos.
- Capítulo 2: descrição das tecnologias utilizadas no desenvolvimento do jogo, comparando com outras soluções existentes.
- Capítulo 3: Implementação do jogo com as tecnologias expostas no capítulo 2.
- Capítulo 4: refere-se à conclusão do Trabalho de Conclusão de Curso.

2 BASES TECNOLÓGICAS

2.1 INTRODUÇÃO DO CAPÍTULO

A chave para fazer uma grande variedade de produtos de *software* em um curto espaço de tempo é fazer com que uma parte do esforço de desenvolvimento sirva para muitos outros produtos (DESMOND FRANCIS; WILLS, 1999). A construção de um jogo online, partindo do zero, carrega uma quantidade grande de complexidade e de esforço, porém, utilizando tecnologias já consolidadas, apenas lhes dando uma nova abordagem, faz com que o processo se torne muito mais simples. Mesmo com tecnologias que não têm o propósito de criação de jogos, podemos encontrar suporte para essa demanda.

O centro desta aplicação, o jogo propriamente dito, utiliza um *framework* que muito se assemelha a outros que são utilizados para construção de sites e aplicações *web* no geral. As ferramentas genéricas nos trazem a liberdade de utilizá-las de formas que nem foram previstas pelos criadores, e que podem se encaixar perfeitamente em necessidades completamente específicas.

Neste capítulo serão apresentadas as tecnologias utilizadas para desenvolver o jogo. O centro do software é o *framework* Phaser, desenvolvido para facilitar o desenvolvimento de jogos para navegador *web*, que é feito em JavaScript (JS), isso possibilita a utilização do TypeScript, trazendo o poder da linguagem para a tecnologia. Para que o jogo seja empacotado, e transformado em um código minificado, foi escolhido o Webpack, a tecnologia da suporte para aplicações JS modernas, possibilitando diferentes *pipelines* durante o empacotamento.

2.2 PHASER:

Phaser (STORM, 2020) é um *framework* gratuito e *open source* de desenvolvimento de jogos para HTML5, que oferece renderização WebGL e Canvas tanto para navegadores de *desktop* quando para navegadores *mobile*. Os jogos podem ser compilados para IOS, Android, e aplicativos nativos usando ferramentas de terceiros. O *framework* permite tanto a utilização de JavaScript como TypeScript para o desenvolvimento. A figura 1 apresenta o logo da tecnologia, que traz ao fundo o mini jogo que serve de tutorial para começar a conhecer a ferramenta.

Figura 1 – Logo do Phaser 3

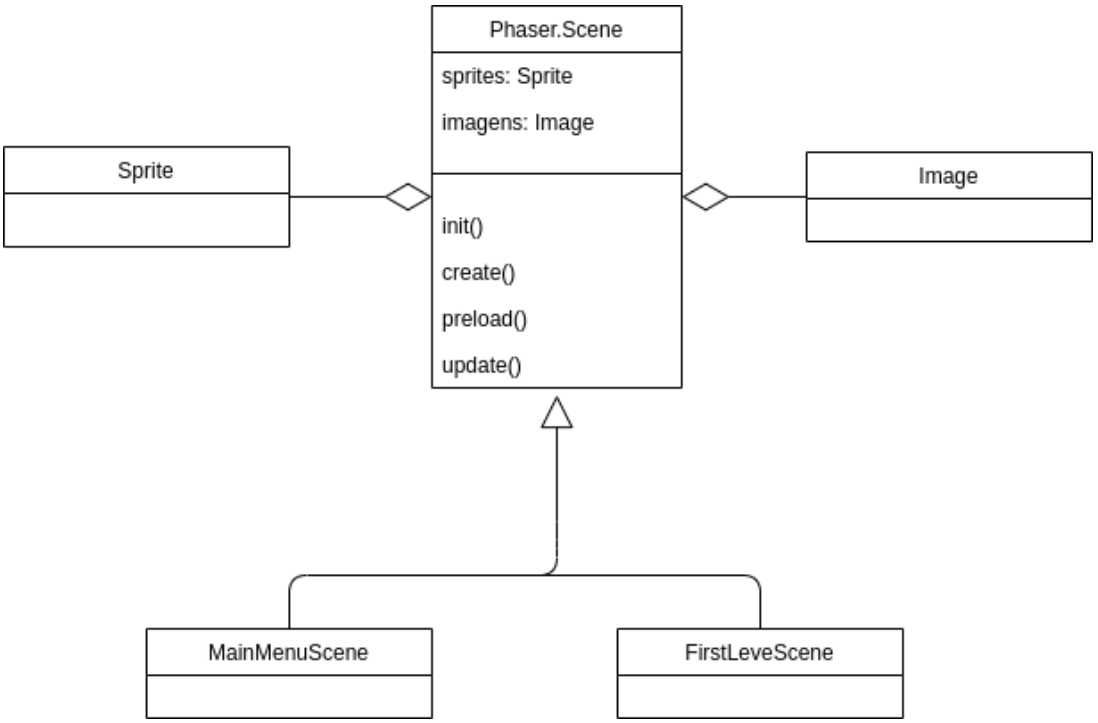
Fonte: https://phaser.io/content/tutorials/making-your-first-phaser-3-game/tutorial_header.png

Segundo (FAAS, 2017) o framework tem alguns recursos fundamentais para a construção de qualquer jogo, por menor ou mais simples que seja. A Application Programming Interface (API) que ele provém para invocar seus comportamentos, em muito se assemelha à Unity, uma *game engine* muito popular, utilizada no desenvolvimento de jogos como Magic Arena. O Phaser, é voltado principalmente para os jogos 2D, tendo em suas estruturas e ferramentas formas muitos simples para trabalhar nesse contexto, que serão vistas a seguir.

2.2.1 Scene:

Tudo que acontece no jogo está dentro de uma cena, chamada de “scene” no Phaser 3, representada por uma classe, que pode ser herdada para se fazer um uso mais específico da mesma, figura 2.

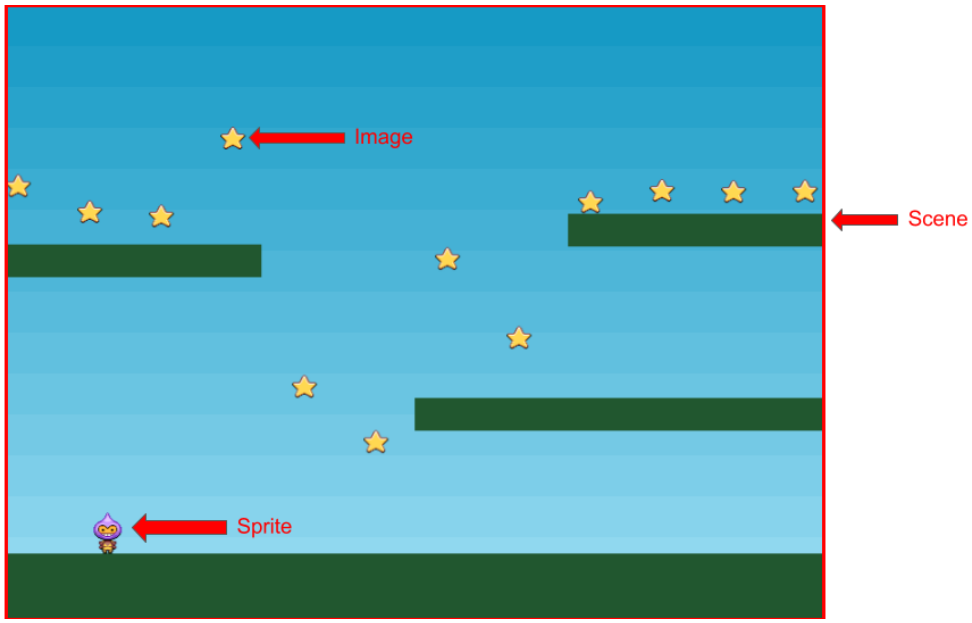
Figura 2 – Diagrama de classe Phaser



Fonte: Acervo Pessoal

Após ser renderizado, a estrutura demonstrada no diagrama de classes presente na figura 2 resulta em uma *Scene*, aos moldes do que é demonstrado na figura 3. Onde estão presentes: os objetos do tipo imagem, representados pelas estrelas e um objeto do tipo *sprite*, representado pelo personagem.

Figura 3 – Cena Phaser



Modificado de: https://phaser.io/content/tutorials/making-your-first-phaser-3-game/part8.png

A *Scene* implementa quatro métodos, init(), preload(), create(), update(), que são responsáveis pela criação e manutenção da cena atual do jogo. Esses métodos permitem delimitar em que etapa da vida do objeto, um determinado comportamento será executado, assim permitindo que se tenha um controle mais fino da sequência de eventos que culmina na execução do jogo.

- init(): Esse método é invocado quando a cena é iniciada, antes mesmo do método preload() e create(). Ela permite fazer a transferência de dados entre uma cena e outra, uma vez que quando se invoca o método de navegação entre cenas, qualquer dado que é passado como parâmetro é passado como parâmetro do método init() da cena destino.

- preload(): Método usado para carregar os *assets*, que são, no desenvolvimento *web*, todo conteúdo complementar dos *websites*, como imagens, sons, cripts, dentre outros. Esse método é chamado após o método init() e antes do método create(), apenas se a *Scene* possuir o *plugin* de carregamento do Phaser.

- create(): Utilizado para criar os objetos do jogo, esse método é chamado pelo gerenciador de cena após o preload(), para que só seja executada após o *load* dos *assets*. Com esse método, é possível tanto instanciar objetos que só existem de forma interna ao escopo da *Scene*, como também os que existem no “mundo” do jogo, e podem ser acessados por todos os outros objetos existentes fora do escopo da cena.

- update(): Baseado na taxa de atualização, esse método permite executar comportamentos periodicamente.

A utilização desses quatro métodos basicamente constrói e manipula a cena, assim podendo oferecer um grande controle e consequentemente uma enorme eficiência da implementação do jogo. Não é necessário que uma cena implemente os quatro métodos, pois nem toda cena faz uso de todos, podendo uma cena ser estática (sem método update) durante toda sua vida, ou não ter nenhum comportamento específico, não demandando a escrita do método init().

2.2.2 LoaderPlugin:

A *Scene* possui um atributo responsável pelo carregamento dos *assets*, o LoaderPlugin. Por padrão, o LoaderPlugin está salvo dentro do atributo “load”, nele estão contidos os métodos que fazem o carregamento correto de cada tipo direfente de *asset*, e os mesmos devem ser invocados dentro do método “preload()”.

Figura 4 – Utilização dos métodos do LoaderPlugin

```
this.load.audio();

this.load.image();

this.load.spritsheet();
```

Fonte: Acervo próprio

2.2.3 Physics:

Pertencente à uma Scene, o *Plugin* responsável pelo gerenciamento da simulação de física do jogo traz importantes ferramentas de manipulação de corpos físicos. Esses corpos são agrupados e acessíveis dentro do parâmetro physics. A classe physics traz ferramentas que se responsabilizam pelas colisões, animações, e outros comportamentos, como estado da cena (pause/resume) e gravidade, caso exista.

Todos os objetos em cena serão administrados por essa classe, tanto estáticos como os objetos dinâmicos. Todo objeto criado pela classe ou adicionado a ela irá receber propriedade de corpo dinâmico, caso não seja definido no momento que o mesmo for criado/adicionado, sendo colocado na estrutura *Group*. Caso o objeto seja um corpo estático, deve ser adicionado ao *StaticGroup*, possibilitando que atribuições genéricas de corpos dinâmicos não afetem corpos estáticos e vice-versa.

2.2.3.1 Group

Um grupo é uma forma de criar, manipular, ou reciclar objetos no jogo, um objeto pode tanto pertencer a um, vários ou nenhum grupo. Grupos, em si, não são exibidos e não podem ser posicionados, rotacionados, dimensionados ou ocultados. Todo objeto criado ou adicionado a esse grupo, recebe automaticamente propriedade de corpos físicos no jogo, caso ele ainda não possua.

Figura 5 – Criando um Group no atributo physics

```
this.physics.add.Group();
```

Fonte: Acervo próprio

2.2.3.2 StaticGroup

StaticGroup é um objeto que guarda objetos com propriedades estáticas, da mesma forma que sua contraparte dinâmica (Group), caso o objeto não tenha um corpo físico, ele receberá um automaticamente. Também não será exibido ou manipulado de forma visual.

Figura 6 – Criando um StaticGroup no atributo physics

```
this.physics.add.staticGroup();
```

Fonte: Acervo próprio

2.2.4 Sprite:

Um *sprite* (DAM et al., 1990) é uma pequena região retangular na memória que é misturada com o resto do *frame-buffer*, um *buffer* a nível de vídeo responsável pelo que será exibido na tela. Os registradores guardam a posição na memória dessa região, e conforme o valor do registrador vai sendo alterado, é carregado para o *frame-buffer* a próxima parte do *sprite*, criando o movimento. Desde que o registrador contenha a localização do *sprite*, é possível também verificar a colisão entre *sprites*, comparando a coordenada que eles serão exibidos na tela.

No Phaser, os *sprites* são implementados em uma classe de mesmo nome “Sprite”, e podem ser usados tanto para exibição de animações como imagens estáticas. Eles podem possuir *input events* que são comportamentos disparados por comandos apertados pelo jogador, e também possuir um corpo dentro da physics da cena. Sprites podem ser interpolados, coloridos, “rolados” e animados (pois possuem componente de animação), porém, tal flexibilidade de uso tem um custo maior no processamento, sendo necessária uma avaliação da necessidade de se usar um *sprite* ao invés de uma imagem, o que será tratado a seguir.

Para se carregar as imagens que serão transformadas em um *sprite*, é utilizado um *sprite sheet*, um arquivo de imagem, que contém todos os *frames* que compõem a animação. Todos os *frames* devem ter o mesmo tamanho em *pixels*, para que seja possível sua execução de forma correta. O arquivo é carregado através do método “spritesheet()”, ele será invocado no atributo “load” que cada *scene* possui.

Figura 7 – Carregamento de um spritesheet

```
this.load.spritesheet('nome_da_animação',
                      'caminho_do_arquivo',
                      { frameWidth: 37, frameHeight: 45 });
```

Fonte: Acervo próprio

Todo *asset* deve ser carregado dentro do método “preload()”, presente dentro da *scene*. O formato da imagem que deve ser usada como *spritesheet* é no modelo da figura 8.

Figura 8 – Sprite sheet do mascote do Phaser

Fonte: https://phaser.io/content/tutorials/making-your-first-phaser-3-game/dude.png

2.2.4.1 Play():

Esse método permite executar uma animação em um objeto do jogo que tenha o componente de animação, como por exemplo a objetos da classe Sprite. As animações ficam armazenadas no gerenciador global de animações, sendo referenciadas com uma *string* única como chave. Para se criar uma animação, basta utilizar o método create() do gerenciador de animações, e então ela existirá globalmente, não estando vinculada a uma cena específica. O método “play()” tem uma importância fundamental para a utilização do *framework*, pois com ele se pode acionar qualquer animação já criada, em qualquer entidade “animável”, figura 9.

Figura 9 – Executando a animação de um sprite

```
sprite.anims.play('nome_da_animação', true);
```

Fonte: Acervo próprio

O atributo “anims” é o componente de animação necessário para executar as animações.

2.2.5 Image:

Imagens são objetos mais “leves” para o processamento do jogo do que *sprites*, utilizados para exibir imagens estáticas no jogo, como logos, *backgrounds*, cenários

ou qualquer outro elemento que não será animado. Da mesma forma que os *sprites*, imagens podem possuir *input events*, ser interpoladas, coloridas e “roladas”, porém, o que as difere é o fato de que não podem ser animadas pois não possuem componente de animação. Por demandar menor processamento que os *sprites*, havendo a possibilidade, é sempre uma melhor escolha utilizar imagem.

2.2.6 Audio:

O controle padrão de som é extremamente simples: basta criar um objeto de som e adicionar ao atributo “sound” que a “Scene” tem por padrão, utilizando a chave que foi atribuída a ele no momento do *load*. Assim que o objeto é instanciado, ele já pode ter o seu som executado.

Figura 10 – Carregamento e execução de um som

```
let som = this.sound.add('chave_do_som');

som.play();
```

Fonte: Acervo próprio

O *loader* aceita uma ampla gama de formatos de sons, como mp3, wav, ogg, permitindo que se escolha entre formas mais eficientes ou com maior qualidade.

2.2.7 KeyboardPlugin:

Um *plugin* de *inputs*, responsável por “ouvir” os eventos do teclado nativos do Document Object Model (DOM) e então fazer o processamento deles, não sendo necessário instanciar essa classe pois os sistema de *inputs* irá criá-lo automaticamente. O objeto criado ficará dentro do atributo “input” da cena, se houver múltiplas cenas em paralelo tentando acessar a mesma entrada do teclado é necessário desabilitar na cena onde não será usado. O *plugin* não lida com nenhum tipo de extensão que altere os *inputs*, podendo haver anomalias caso não sejam desabilitadas. É possível criar um *set* de teclas específico, porém também é possível ter um conjunto básico de teclas criadas através do método createCursorKeys() que criará e retornara um objeto contendo as teclas cima, baixo, esquerda, direita, espaço e shift.

Figura 11 – Criando um objeto de inputs

```
const comandos = this.input.keyboard.createCursorKeys();
```

Fonte: Acervo próprio

2.2.8 GameConfig:

Esse objeto é responsável por declarar as configurações globais do jogo. Na inicialização do jogo, o objeto de configuração precisa ser enviado como parâmetro,

nele conterá informações como resolução da tela, zoom, tamanho relativo da tela, tipo de física, dentre outras configurações específicas do jogo.

Figura 12 – Objeto de configuração global do jogo

```
const config = {
    width: 1366,
    height: 768,
    scene: Main,
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 300 },
            debug: false
        }
    }
}

const game = new Phaser.Game(config);
```

Fonte: Acervo próprio

Quando um novo jogo é instanciado, o objeto “config” é enviado como parâmetro, o objeto de configuração deve ser no modelo indicado pela documentação do framework.

2.2.9 Execução padrão:

O fluxo padrão de um jogo no phaser, composto de uma ou mais cenas, começa no método init(), onde as informações de fora da cena são recebidas por parâmetro e as variáveis são iniciadas. Em seguida chama-se o método preload(), que carrega os *assets*, caso a cena precise de algum que ainda não esteja carregado. Após o carregamento, o método create() se responsabiliza por criar todos os objetos e estruturas necessárias para a execução da cena, por fim, caso a cena necessite, o método update() recarrega a cena para atualizar as possíveis mudanças que ocorreram. Todo processo se repete a cada nova cena que seja carregada.

Figura 13 – Execução do preload() em uma cena simples

```
export class Cena extends Phaser.Scene {
    constructor(){
    }

    preload(){

        this.load.image('sky', 'assets/sky.png');
        this.load.image('ground', 'assets/platform.png');
        this.load.image('star', 'assets/star.png');
        this.load.spritesheet('dude',
            'assets/dude.png',
            { frameWidth: 32, frameHeight: 48 });
    }
```

Fonte: Acervo próprio

Após todos os *assets* serem carregados, o método create() é invocado para que sejam criados todos os objetos que serão utilizados nessa cena, como mostrado na figura 14.

Figura 14 – Execução do create de uma cena simples

```
create(){

    this.add.image(400, 300, 'sky');
    this.add.image(400, 300, 'star');
    const = platforms = this.physics.add.staticGroup();

    platforms.create(400, 568, 'ground')
        .setScale(2)
        .refreshBody();

    platforms.create(600, 400, 'ground');
    platforms.create(50, 250, 'ground');
    platforms.create(750, 220, 'ground')

    const = player = this.physics.add.sprite(100, 450, 'dude');

    player.setBounce(0.2);

    this.physics.add.collider(player, platforms);

    const = stars = this.physics.add.group({
        key: 'star',
        repeat: 11,
        setXY: { x: 12, y: 0, stepX: 70 }
    });

    stars.children.iterate(function (child) {

        child.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));

    });

    this.physics.add.collider(stars, platforms);


}
```

Fonte: Acervo próprio

O resultado é a cena da figura 3, algumas ferramentas utilizadas serão explicadas posteriormente, durante a implementação.

2.3 WEBPACK:

Webpack (WEBSOCKET.ORG, 2020) é um empacotador de módulos estático para aplicações JavaScript modernas e roda em Node.js. Quando ele processa a apli-

cação, ele internamente constrói um grafo de dependências que mapeia cada módulo que o projeto precisa e gera um ou mais pacotes.

Em programação modular, programadores dividem os programas em partes discretas de funcionalidades chamadas módulos. O webpack suporta módulos escritos em uma variedade de linguagens e pré-processadores via *loaders*, que descrevem ao empacotador como processar módulos não escritos em JS.

2.3.1 Conceitos Básicos:

2.3.1.1 Entry:

Um *entry point* indica por qual módulo o webpack deve começar a construir seu grafo de dependências interno. O empacotador resolverá quais outros módulos e bibliotecas que esse *entry point* tem como dependência. Por padrão, o valor é ./src/index.js, mas um diferente pode ser especificado (podendo ser mais de um), basta adicionar a propriedade “entry” no arquivo de configuração.

2.3.1.2 Output:

A propriedade *output* comunica ao webpack onde ele deve gerar os pacotes e como nomear esses arquivos. A saída padrão é ./dist/main.js para o arquivo principal, e ./dist para qualquer outro arquivo gerado, podendo ser especificada outra saída no arquivo de configuração.

2.3.1.3 Loaders:

O webpack só consegue entender arquivos JavaScript e JSON, porém, utilizado Loaders, é possível que o empacotador processe outros tipos de arquivos e converta-os em módulos válidos, que podem ser utilizados pela aplicação que está sendo empacotada.

2.3.1.4 Plugins:

Os plugins, por definição, expandem as capacidades de uma aplicação. No webpack eles possibilitam uma ampla gama de tarefas durante o empacotamento, gerenciamentos dos *assets* e injeção de variáveis de ambiente.

2.3.1.5 Mode:

Declarando a variável mode, em “development”, “production” ou “none”, é possível habilitar as opções de otimização do webpack correspondente a cada ambiente. O valor padrão é “roduction”.

2.3.1.6 Compatibilidade com navegadores:

Todos os navegadores que são compatíveis com ECMAScript 5 (ES5) são suportados pelo webpack. Caso haja a necessidade de dar suporte para algum navegador mais antigo, é necessário carregar uma biblioteca de “polyfill” que traduza os comandos que o empacotador utiliza.

2.3.2 Configuração:

Caso se queira garantir uma estabilidade das configurações, para que possíveis mudanças nas configurações padrões do empacotador não afetem o projeto, pode-se criar o arquivo onde se declara todos os aspectos importantes, como mostrado na figura 15.

Figura 15 – webpack.config.js

```
module.exports = {
    entry: {
        main: './src/index.js'
    },

    mode: 'production',

    output: {
        path: path.resolve(__dirname, 'dist'),
        filename: 'main.js'


    }
}
```

Fonte: Acervo próprio

Outra vantagem de se ter um arquivo de configurações é que ele pode ser alterado a qualquer tempo para se adicionar mais funcionalidades ao webpack.

O principal motivo de se utilizar um empacotador é o fato a aplicação ser composta de muitos componentes diferentes e separados, podendo ser necessário aplicar processamentos específicos e otimizações para partes distintas deles, ou até mesmo criar rotinas na hora de se gerar o projeto final. Graças ao empacotador, se garante que existe uma coesão entre todas as partes da aplicação.

2.4 TYPESCRIPT:

TypeScript (MICROSOFT, 2020) é um superconjunto tipado de JS que transpila para JavaScript simples. Ele adiciona tipos opcionais, permitindo aos programadores utilizarem ferramentas e práticas para otimizar o seu trabalho. Como sua sintaxe é um

superconjunto de ECMAScript 2015, todo programa JS é também um programa TypeScript (TS). Possui todos os recursos do ES6, incluindo classes e módulos, também fornece a capacidade de converter esses recursos em código compatível com ES 3 ou 5.

A linguagem fornece aos desenvolvedores um sistema opcional de anotações de tipo, possibilitando que sejam expressos limites aos recursos de objetos JS. Porém, para minimizar o uso extensivo delas, o TS utiliza inferência de tipo, baseado no dado que é colocado em um variável ou retornado em algum método, o TS utiliza equivalência estrutural de tipos ao invés de equivalência por nome, como outras linguagens, conferindo a ele uma flexibilidade na utilização desses recursos.

2.4.1 Tipos básicos:

Esse superconjunto de JS oferece os mesmos tipos básicos que o próprio JavaScript, com a adição de enums, esses tipos de dados tornam o desenvolvimento mais sólido e claro, e são eles:

- Boolean
- Number
- String
- Array
- Tuple
- Enum
- Unknown
- Any
- Void
- Never
- Null e Undefined
- Object

Em JS, existe uma diferença entre *undefined* e *null*, *undefined* é apenas o tipo de uma variável que não teve um valor atribuído a ela, enquanto o *null* é considerado um objeto e pode ser atribuído a uma variável, fazendo que ela não seja mais do tipo *undefined*.

O tipo *never* representa um valor que nunca ocorrerá, em um retorno de uma função ou método que sempre retorna uma exceção. Esse tipo pode ser inferido, então em uma função que recebe um erro e joga uma exceção, o retorno sempre será do tipo *never*.

2.4.2 Interfaces:

Um dos princípios básicos do TS é a checagem de tipo utilizando a estrutura de um dado, e esse papel de criar uma estrutura esperada é da interface. Através da interface pode ser decidido que estrutura é esperada para um tipo de dado, como mostra a figura 16.

Figura 16 – Exemplo de interface em TypeScript

```
interface Exemplo {
    nome: string;
    idade: number;
}
```

Fonte: Acervo próprio

Existem situações em que nem todas as propriedades de uma interface são obrigatórias, e nessas situações basta que seja adicionado o operador “?” após o nome da propriedade como mostra a figura 17.

Figura 17 – Exemplo de interface com propriedade opcional em TypeScript

```
interface Exemplo {
    nome: string;
    idade: number;
    email?: string;
}
```

Fonte: Acervo próprio

2.4.3 Funções:

Da mesma forma que em JS, funções podem ser nomeadas ou anônimas. Essa possibilidade nos permite escolher qual abordagem é mais apropriada para cada situação. E devido a proposta da linguagem, as funções também podem receber tipos de retorno, como mostra a figura 18.

Figura 18 – Exemplo de funções anônimas e nomeadas em TypeScript

```
//função nomeada com retorno number
function add(x:number, y:number): number {
    return x + y;
}
//função anônima atribuída à variável add com retorno number
const add = (x: number, y: number): number  => {
    return x + y;
}
```

Fonte: Acervo próprio

2.4.4 Classes:

Na especificação ECMAScript 6, foi implementada a sintaxe para a utilização de classes. Porém, com o TS, é possível se aproveitar dessa possibilidade sem a preocupação com quais navegadores suportam essa sintaxe, pois, o compilador irá traduzir o código para uma versão que não possuía essa *feature*, garantindo que exista uma estabilidade no funcionamento da aplicação mesmo em diferentes cenários de uso. As classes tem uma sintaxe muito semelhante a outras linguagens orientadas a objetos com a particularidade de que todos os atributos da classe são públicos por padrão. No entanto isso pode ser declarado como mostra a figura 19.

Figura 19 – Exemplo de classes em TypeScript

```
class Exemplo {
    nome: string;
    idade: number;
    private email: string;

    constructor(){}
}
```

Fonte: Acervo próprio

2.4.5 Genéricos:

Com o tipo “any” é possível que nosso componente aceite qualquer tipo de dado, e retorne também qualquer tipo, sem que haja qualquer impedimento, porém, existem situações em que mesmo não estando definido que tipo de dado será recebido ou retornado, essa informação é relevante. Para resolver isso existe a variável de tipo, “T”, que nos permite capturar o tipo do dado em questão e utilizarmos ele para a finalidade que desejamos. A utilização dessa variável é muito semelhante ao Java, como mostra a figura 20.

Figura 20 – Exemplo de tipos genéricos em TypeScript

```
const exemplo<T> = (x: T): T => {
    return x;
}
```

Fonte: Acervo próprio

2.5 OUTRAS TECNOLOGIAS

2.5.1 Unity

Unity é uma *game engine* que tem a proposta de tornar simples e rápido o desenvolvimento de jogos. Voltada à criação de jogos 3D, segundo Goldstone (GOLDSTONE, 2009) a Unity provém um processo de produção simples, separado em pequenos passos que permitem a construção de qualquer jogo. A *engine* estabelece o conceito de *Game Object* (GO), permitindo dividir o jogo em vários elementos simples e de fácil administração. Ela também oferece um editor que torna todo processo fácil, com poucos *clicks* é possível criar objetos com propriedades, tudo viabilizado pela forma com que o motor se estrutura. É possível utilizar a API em C, porém ela é mais utilizada na criação de comportamentos específicos em pequenos *scripts*.

2.5.2 Three.js

Lentamente os navegadores estão ganhando funcionalidades mais poderosas que podem ser acessadas através do JavaScript. Com a utilização da *tag* canvas do HTML5 é possível adicionar componentes interativos, uma importante adição a essa funcionalidade é o suporte ao WebGL, que provém uma interface de baixo nível para a utilização da unidade de processamento gráfico do dispositivo que está rodando a aplicação, possibilitando uma melhor performance na renderização de gráficos 2D e 3D. Como programar o WebGL diretamente com JS é um processo extremamente complexo e propenso a erros, algumas bibliotecas criam ambientes mais confortáveis para essa tarefa, uma delas é a Three.js. Segundo Dirksen (DIRKSEN, 2013) esta biblioteca possibilita criar objetos geométricos 3D simples e complexos, animar e mover objetos em uma cena 3D, aplicar texturas aos objetos, carregar objetos 3D de *softwares* de modelagem e criar gráficos 2D baseados em *sprites* de forma muito mais simples e direta, abstraindo todas as complexidades que estão envolvidas nessas tarefas.

2.5.3 PHP

Segundo Rutledge (RUTLEDGE, 2004) o PHP: Hypertext Preprocessor (PHP) traz como grande vantagem para o desenvolvimento de jogos o fato de ser uma linguagem orientada a objetos, fazendo com que seja flexível e compatível com práticas comuns no desenvolvimento desse tipo de *software*. A biblioteca GD é a ferramenta que possibilita ao PHP formas de manipular gráficos, construída em C, ela traz um po-

der que o PHP não teria de forma nativa. Obviamente todo processamento é feito na parte do servidor e o elemento HTML é desenhado na tela que precisa ser atualizada, fazendo com que essa seja uma solução muito limitada e uma escolha difícil de ser a mais adequada.

2.5.4 Java

Java é uma linguagem de programação muito consolidada e poderosa, apresentando ferramentas para os mais diversos fins, e não seria diferente para o desenvolvimento de jogos. Muito além da capacidade de fazer servidores com um ótimo desempenho, para jogos online, graças a bibliotecas como LibGDX, é possível manipular elementos visuais de forma muito simples. A biblioteca LibGDX, segundo Stemkoski (2015), possibilita, além de compilar o jogo para multiplataformas, uma série de vantagens em tarefas como:

- Renderizar gráficos 2D, animações, fontes baseadas em *bitmap* e efeitos de partículas
- Fazer *stream* de músicas e reproduzir efeitos sonoros
- Processar *input* de teclado, mouse, *touch screens*, acelerômetro ou controles de videogames.
- Organizar interface de usuário
- Integrar com *plugins* de terceiros
- Renderizar gráficos 3D, carregar modelos de *softwares* de modelagem e reproduzir efeitos de luz

2.5.5 Godot

Segundo Bradfield (BRADFIELD, 2018), Godot é um motor gráfico moderno e completo, que provém as funcionalidades das *engines* mais difundidas, como renderização de gráficos 2D e 3D, manipulação de física, suporte para múltiplas plataformas e um ambiente de desenvolvimento familiar para os desenvolvedores. Um motor de código aberto e gratuito, o que o torna um prato cheio para criadores de jogos *indie*. O fato de a *engine* ser de código aberto possibilita que, em caso de alguma funcionalidade não estar suprindo a necessidade do projeto, os próprios desenvolvedores do jogo podem alterar a estrutura da mesma, e também, torna a depuração do projeto muito mais fácil, uma vez que o código está todo exposto, diferente de outras soluções do mercado.

A escolha do Phaser como a base para o trabalho, se fundamenta no fato de que ele, além de utilizar o JS, entrega todas as funcionalidades básicas necessárias para criação de um jogo, além de que, pelo fato de ser focada em jogos 2D, abstrai escolhas que precisariam ser feitas em outras tecnologias.

As outras tecnologias mencionadas, com exceção do PHP que cria muitas barreiras, apresentam pontos fortes e vantagens muito interessantes, com destaque para o Godot, que mesmo sendo uma tecnologia muito nova, criada em 2014, traz grandes possibilidades, fazendo com que, em um cenário diferente, ele fosse a melhor escolha.

Three.js, por mais que tenha suporte para 2D, e também utilize JS, claramente foi feita para se desenvolver jogos 3D, tornando o desenvolvimento do jogo proposto mais dispendioso, pelo fato de que, enquanto o Phaser é especializado na produção de gráficos com *sprite*, o Three.js precisaria de algumas adaptações, que facilmente poderiam gerar anomalias pela inexperiência do desenvolvedor com a ferramenta.

2.5.6 Adobe Flash

Como já foi brevemente apontado anteriormente, o Adobe Flash foi uma ferramenta muito popular para o desenvolvimento de jogos para navegadores. Segundo (CURRAN; GEORGE, 2012), jogos feitos com essa tecnolgia, como o já mencionado *FarmVille* são responsáveis por uma expressiva fatia dos usuários de jogos de navegador. Com 80 (oitenta) milhões de usuários, apenas o *FarmVille* tem uma base de jogadores maior que alguns consoles. A principal desvantagem do Flash é o fato de precisar de *plugins* para funcionar, enquanto o HTML5 já da suporte às mesmas capacidades sem necessitar instalar nada. Ainda segundo o autor, o fato de a internet estar migrando para um contexto de *"plug-in free web"* e o HTML5 proporcionar uma forma de atingir um número maior de plataformas, o Flash gradativamente tem perdido espaço no mercado.

2.5.7 Conclusão do capítulo

Com a utilização dessas tecnologias é possível construir um jogo completo, da mesma forma que seria desenvolvida uma aplicação *web* tradicional. As formas e metodologias utilizadas para criança de *webpages* modernas, se tornaram tão consolidadas e amplamente utilizadas, que podem ser estendidos para outros tipos de aplicações.

Por mais distantes que possam parecer as abordagens para se criar um site e para se criar um jogo, com mínimas modificações na metodologia *web*, pode-se aplicar todos os paradigmas tradicionais do desenvolvimento de jogos.

O desafio maior se concentra na parte artística, pois, será a parte mais evidente da aplicação, e vai depender quase que exclusivamente de encontrar arte de terceiros disponível. Também é a parte que pode decidir as configurações internas do jogo, uma vez que não sendo de produção própria, será necessário adaptar à situação.

3 DESENVOLVIMENTO

3.1 INTRODUÇÃO DO CAPÍTULO

O jogo desenvolvido nesse trabalho é inspirado no jogo *Bomberman*. A escolha do projeto ser inspirado nele é pela simplicidade. Sendo baseado em partidas disputadas entre 2 (dois) jogadores em um campo de batalha restrito.

Segundo (CRUZ LOPES, 2016), *Bomberman*, também conhecido como *Dynablaster*, é um *video game* de estratégia que acontece dentro de um labirinto. Originalmente desenvolvido pela Hudson Soft, o jogo tem mais de 80 (oitenta) versões diferentes. Algumas dessas versões são jogos multijogador em que os jogadores são adversários, e o vencedor é o último restante. Independente do modo de jogo, a lógica básica é sempre a mesma. Cada cenário é um retângulo cercado por blocos indestrutíveis, o caminho pode ser bloqueado por diferentes tipos de blocos, formando uma espécie de labirinto. O jogador pode destruir alguns desses blocos colocando bombas perto deles. Quando a bomba é detonada, uma explosão é criada em uma linha vertical e uma horizontal com o centro sendo o bloco onde estava a bomba. O contato da explosão irá destruir qualquer elemento destrutível no cenário.

Dessa forma, a criação de um jogo inspirado no *Bomberman* se apresenta como uma escolha viável. Tanto as regras como a dinâmica do jogo são simples, sendo necessário apenas reproduzir esses comportamentos descritos e animar os *sprites*. Acrescentando o elemento *on-line*, é possível expandir as possibilidades de interação entre jogadores. Colocando esses elementos dentro de uma página *web*, o contexto em que o jogo será executado facilitará muito essa conexão entre jogadores.

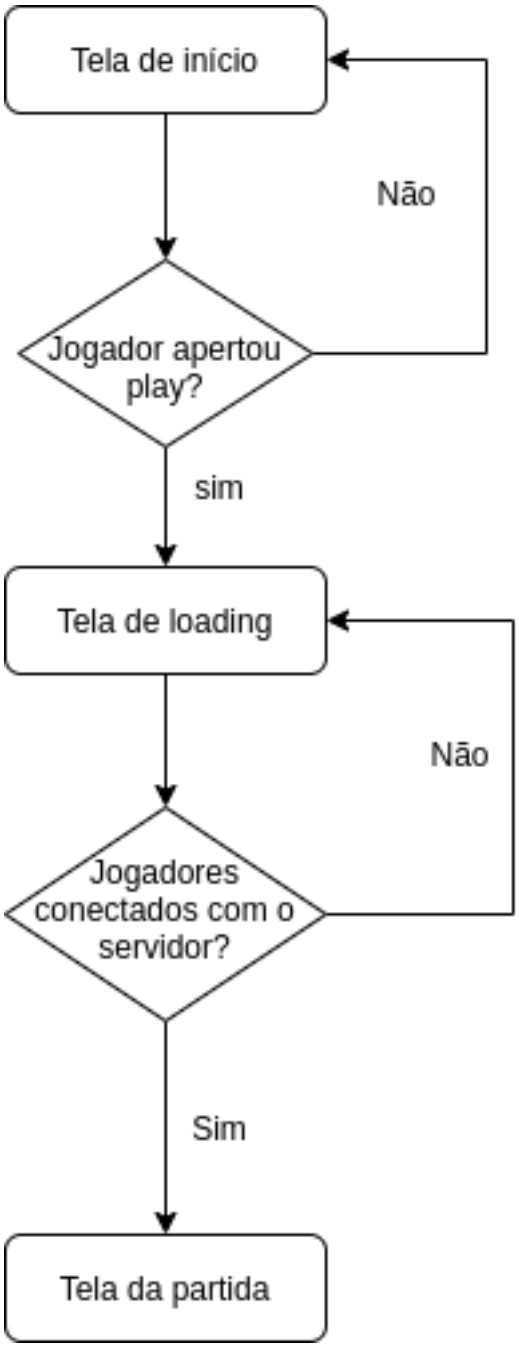
Neste capítulo são apresentados os detalhes da implementação do *software*. A parte principal da aplicação é o *client*, que no caso será o jogo em si. É nele que estarão contidos o código fonte e os *assets*, que consistem em arquivos de imagem e áudio. Também existe o lado do servidor, mas devido à forma que é feita a comunicação entre os *clients*, os detalhes de sua implementação são bem simples. Também será tratada toda a parte de configuração necessária para a *build* da aplicação e o serviço de hospedagem da página que consiste o jogo. Serão utilizadas apenas tecnologias voltadas para desenvolvimento de aplicações web, sem precisar nenhuma instalação de recursos extras, nem o jogo em si será instalado. Tudo terá o mesmo comportamento de um *web site* com a finalidade de explorar soluções que usam recursos nativos da *web*.

3.2 O JOGO

Todo jogo é composto por 3 (três) *scenes*: tela de início, contendo o título do jogo e o botão de *play*; tela de *load*, que aparece para o jogador enquanto aguarda o carregamento e configurações; tela da partida em si, onde os jogadores se enfrentam em um campo de batalha. Essa sequencia é demonstrada na figura 21 e se repete sempre que um novo jogo vai começar.

O jogo não é composto só pelo *client*, ele também tem um lado do servidor que é responsável por fazer a comunicação entre os jogadores, e também atribuir um identificador (ID) único para cada jogador bem como decidir qual o *spritesheet* e a posição de cada jogador no campo de batalha.

Figura 21 – Diagrama do fluxo das cenas



Fonte: Acervo próprio

3.2.1 Tela inicial

A tela inicial do jogo é apresentada sempre que o jogador acessa a Uniform Resource Locator (URL) do jogo como mostra a figura 22. A tela parece simples, mas ela tem um papel fundamental para o funcionamento do jogo. Ela é a cena principal do jogo, além de carregar a imagem do título, e criar o botão que inicia o jogo, ela faz uma importante comunicação com o servidor.

Quando o jogador entra nessa tela, é enviada uma requisição para o servidor do jogo, que retorna um ID para esse jogador, esse ID é gerado toda vez que essa tela inicial é carregada. Todo esse processo de conexão será explicado no tópico que trata sobre as comunicações entre *client*/servidor.

Figura 22 – Tela inicial do jogo

Fonte: Acervo próprio

O botão *play* guarda o método que foi criado para fazer o carregamento da próxima cena o método searchGame(). No momento em que ele é pressionado, o método do Phaser que carrega uma nova *scene* é chamado, sendo passado como parâmetro do método o nome da nova cena, a classe dessa cena, um booleano declarando a propriedade “autoStart” e um objeto de dados, que no caso é o ID do jogador. Logo após desse método do Phaser ser invocado, o método searchGame() destrói os elementos criados na tela inicial que não serão mais usados como mostra a figura 23.

Figura 23 – Método serachGame

```
searchGame() {
    this.scene.add('waitScreen',
                   WaitScreen,
                   true,
                   {playerId: this.playerId});

    this.playButton.destroy();
    this.title.destroy();
}
```

Fonte: Acervo próprio

3.2.2 Loading Screen

Logo após o botão *play* ser apertado, é carregada a tela de *loading*. Nessa tela não existe imagem carregada, apenas um texto em movimento, que é uma funcionalidade padrão do Phaser como mostra a figura 24.

Figura 24 – Criação de texto no Phaser

```
let loading = this.add.text(600, 450, 'Loading');
```

Fonte: Acervo próprio

O resultado é muito simples, apenas um texto indicando que o jogo está aguardando para começar, como na figura 25.

Figura 25 – Tela de loading

Fonte: Acervo próprio

Apesar de ser uma tela simples, sua função é essencial. Durante a vida dessa *scene* é feita toda comunicação através de *websocket*, o *client* envia seu ID para o servidor, se inscreve no *socket* e aguarda receber as coordenadas do jogador local, o ID do seu *spritesheet*, e as informações do jogador remoto. Todas as informações recebidas são colocadas dentro de um objeto chamado “data” contendo as configurações do jogador local, e uma lista de informações de jogadores remotos (caso o jogo venha a ter mais jogadores). Após o objeto “data” ter os valores recebidos pelo servidor atribuídos, novamente o método do Phaser responsável por carregar uma nova cena é chamado, e é passado por parâmetro o objeto “data” como demonstrado na figura 26.

Figura 26 – Envio e recebimento das informações pelo websocket

```
this.wsConnection.send('/waiting', {}, this.playerId);

this.wsConnection.subscribe('/waiting', notification => {
    let response = JSON.parse(notification.body);
    let data = {
        playerRemoto: {
            id: '',
            x: 0,
            y: 0,
            skin: ''
        }
        playerLocal: {
            id: this.playerId,
            x: 0,
            y: 0,
            skin: ''
        },
    }

    response.forEach(player => {
        if (player.playerId != this.playerId) {
            data.playerRemoto.id = playerId;
            data.playerRemoto.x = player.x;
            data.playerRemoto.y = player.y;
            data.playerRemoto.skin = player.skin;

        } else {
            data.playerLocal.x = player.x;
            data.playerLocal.y = player.y;
            data.playerLocal.skin = player.skin;
        }
    }
    this.scene.add('blastMan', BlastMan, true, data);
});
```

Fonte: Acervo próprio

Os detalhes da comunicação serão explicados no tópico específico sobre conexão com o servidor. Com o envio das informações como parâmetro do método de carregar a nova cena, podemos finalmente iniciar a partida.

3.2.3 A partida

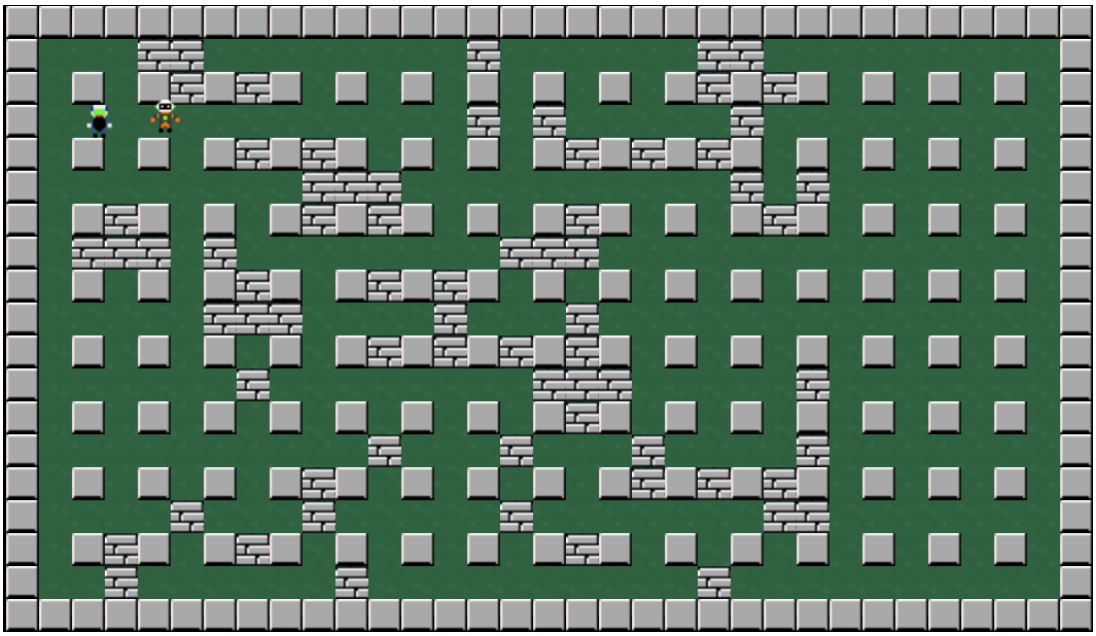
A partida orbita três elementos principais: o campo de batalha, os jogadores e a dinamite, a interação entre esses elementos dita como o jogo vai se comportar. No arquivo de configuração do Phaser, ele permite que se atribua um valor de intensidade

e a direção que a gravidade atua, porém, para esse jogo esse atributo foi omitido intencionalmente. Dessa forma as entidades da cena ficam fixas na posição que suas coordenadas foram configuradas. Com isso, se cria a ilusão de que na verdade a gravidade está agindo no eixo Z, e que estamos vendo a partida de uma perspectiva elevada. Também é possível se decidir sobre a resistência que um corpo tem a alguma força aplicada sobre ele, mas isso será tratado a seguir.

3.2.3.1 Campo de batalha

Toda partida ocorre dentro de um campo de batalha que é um tabuleiro formado por blocos quadrados de de 32x32 *pixels*, o campo tem as proporções de 33 blocos de largura por 19 de altura formando um retângulo como mostra a figura 27. O fundo dele é uma imagem que tem as mesmas dimensões, uma textura de *pixel-art* que é alternada baseada no tipo de campo que é selecionado para a partida.

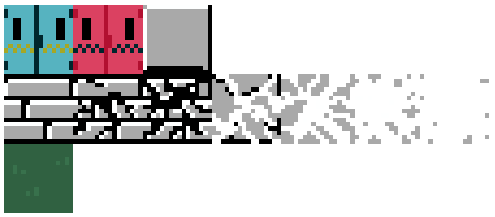
Figura 27 – Campo de batalha



Fonte: Acervo próprio

Os blocos que montam o cenário também são decididos no momento do início da partida, baseados no tipo de campo da partida. As texturas deles ficam em um arquivo de imagem junto com os *sprites* que são responsáveis por animar os blocos destrutíveis do cenário como mostra a figura 28. Os blocos estáticos têm sua posição fixa, porém, os blocos que são *sprites* têm sua posição decidida por um objeto de configuração da cena.

Figura 28 – Componentes do cenário



Modificado de: https://opengameart.org/

Esse arquivo de texturas contém *frames* que nunca serão usados no jogo devido ao fato de que o mesmo foi encontrado na internet, e não feito especificamente para este jogo. Apenas os *frames* dos dois blocos cinza que serão utilizados.

Tanto o fundo do cenário como o arquivo contendo as texturas dos blocos são carregados dentro do método “preload” da cena. Para o campo de batalha são carregados os arquivos de imagens para dentro de dois tipos de objetos, o fundo do cenário é carregado para um objeto do tipo *image* enquanto as texturas dos blocos são carregadas para um *spritesheet*. Ambos os métodos estão disponíveis dentro do atributo *load* disponível na cena como mostra a figura 29.

Figura 29 – Carregamento das texturas

```
preload() {
    this.load.image('fundoCampo', 'texturas/background.png');

    this.load.spritesheet('texturaBlocos',
                          'texturas/blocos.png',
                          {
                              frameHeight: 32,
                              frameWidth: 32
                          });
}
```

Fonte: Acervo próprio

No *spritesheet* é necessário passar um objeto de configuração que especifica o tamanho de cada *frame*, tornando possível exibir só as partes desejadas do *spritesheet* utilizando um *offset*. Ambos os métodos recebem uma *string* que será a chave referenciada toda vez que se desejar acessar essas texturas.

Os blocos são guardados dentro de uma estrutura disponibilizada pelo Phaser chamada de *group*. Os *groups* servem para agrupar objetos que tenham as mesmas propriedades físicas, no caso dos blocos, é utilizada a contraparte estática, o *staticGroup*. O método staticGroup() está disponível dentro da propriedade “physics.add” e é invocado dentro do método da *scene* init como demonstra a figura 30.

Figura 30 – Criação do staticGroup

```
init() {
    this.staticBlocks = this.physics.add.staticGroup();
}
```

Fonte: Acervo próprio

Essa estrutura é armazenada dentro de uma variável chamada “staticBlocks” para que possa ser mais fácil de referenciá-las. Cada bloco é criado através do método “create” disponível na classe Group, essa estrutura não diferencia imagens de *sprites*, então é possível armazenar tanto os blocos destrutíveis (com animações) como os blocos que são fixos. Para adicionar um novo elemento basta passar uma latitude, uma longitude, a chave que guarda a textura e o *frame* específico que será a textura do objeto como mostra a figura 31.

Figura 31 – Adicionando um bloco ao group

```
create() {
    this.staticBlocks.create(0, 0, 'texturaBlocos', 2);
}
```

Fonte: Acervo próprio

Estão sendo enviadas as coordenadas (0,0), a chave que referencia as texturas e o *frame* do bloco, como esse *frame* é um offset, para o bloco fixo é enviada a posição 2 (dois).

O “staticGroup” atribui a todos os objetos armazenados nele a propriedade de objetos estáticos, isto é, esses objetos não podem se mover na tela e não podem ser transpassados. A possibilidade de se movimentar é algo intrínseco aos *sprites* no Phaser, caso esses blocos não estivessem dentro do “staticGroup” seria necessário configurar esse comportamento individualmente como exemplificado na figura 32.

Figura 32 – Atribuindo a propriedade de imobilidade ao sprite

```
create() {
    const bloco = this.physics.add.sprite(0, 0, 'bloco');
    bloco.setImmovable(true);
}
```

Fonte: Acervo próprio

A melhor forma encontrada para se trabalhar com as artes do jogo foi criando elas ou adaptando de sites que disponibilizam de forma gratuita. Como o jogo é composto de poucos elementos, poucas artes montam o jogo todo. O título e os personagens foram artes originais, porém, tudo que envolve o cenário foi encontrado em sites que disponibilizam *pixel-arts*.

Ainda dentro do método *create* será efetivamente criada a imagem como um objeto do jogo e também a animação que irá ser reproduzida quando um bloco for destruído. O método de criar imagem tem um funcionamento muito simples: basta enviar como parâmetro as coordenadas e a chave que guarda a própria imagem. Já a animação é um método um pouco mais complexo: ele recebe por parâmetro um objeto de configuração contendo uma chave que será a forma que a animação será referenciada, os *frames* da animação, a taxa de *frames* da animação (*frame rate*) e quantas vezes a animação será repetida quando for invocada, ambos os métodos demonstrados na figura 33.

Figura 33 – Criação de imagem e animação

```
create() {
    this.add.image(0, 0, 'fundoCampo');

    this.anims.create({
        key: 'destroy',
        frames: this.anims.generateFramesNumber('texturaBlocos',
                                                {start: 7, end: 13}),
        frameRate: 20,
        repeat: 0
    });
}
```

Fonte: Acervo próprio

No caso do *spritesheet* “texturaBlocos”, o *offset* para que se tenha os *frames* da animação é 7 e vai até o 13.

3.2.3.2 Personagem

A entidade mais complexa do jogo é o personagem, representado pela classe “Player”. Apesar de ser uma classe com as peculiaridades necessárias para o funcionamento do jogo, ela precisa se comportar como um *sprite*, então ela herdará a classe do Phaser.

A classe tem alguns métodos necessários para o funcionamento da partida. O método “move” é responsável pela movimentação do personagem na tela. Esse método recebe por parâmetro um objeto do tipo “CursorKeys” do Phaser, esse objeto é responsável por ler as entradas do teclado, baseado na tecla que está sendo pressionada no teclado, o método move executa o movimento correspondente. Caso o jogador receba um *input* de movimento para a direita no teclado, a nível de código, dois eventos acontecem: o método do *sprite* “setVelocityX” recebe um número inteiro que representa a velocidade que ele deve se deslocar nesse eixo (número negativo caso o movimento seja para a esquerda) e a animação de movimento é invocada, como demonstrado na figura 34.

Figura 34 – Movimentação de personagem

```
move(cursors) {
    if (cursors.right.isDown) {
        this.setVelocityX(160);
        this.anims.play('movimento_direita', true);

        wsConnection.send('/blasterman_server/move', {},
            JSON.stringify({
                playerId: this.playerId, direction: 'right'
            })
        );
    }
}
```

Fonte: Acervo próprio

Junto com o movimento executado de forma local, é enviado ao servidor um JavaScript Object Notation (JSON) contendo o ID do jogador que executou o movimento e a direção. Essas informações serão utilizadas para que seja possível uma partida com jogadores conectados.

Não existe na cena só o personagem controlado pelo jogador, também existe o personagem que é controlado pelo seu adversário de forma remota. Para isso, existe uma versão do método “move” que é responsável por fazer a movimentação ordenada pelo servidor. Ao invés de ser enviado por parâmetro o objeto que recebe os comandos do teclado, é enviado apenas a direção que o servidor manda o personagem seguir, como mostrado na figura 35.

Figura 35 – Movimentação de personagem vinda do servidor

```
move(direction: string) {
    if (direction === 'right') {
        this.setVelocityX(160);
        this.anims.play('movimento_direita', true);
    }
}
```

Fonte: Acervo próprio

A diferença entre os dois métodos é sutil, mas é fundamental, o método responsável por executar a movimentação vinda do servidor não envia essa movimentação de volta para ele. A forma como esse método será invocado será explicado no tópico sobre a conexão.

O outro método que a classe “Player” possui é o método “kill”, ele é responsável por rodar a animação da morte do personagem e pela destruição do *sprite* ao final dela. São três os eventos envolvidos na morte do personagem: a velocidade dos eixos X e Y é modificada para 0 (zero), a animação a morte do personagem é executada e ao final dela o *sprite* é destruído, como mostrado na imagem 36.

Figura 36 – Método que destrói o personagem

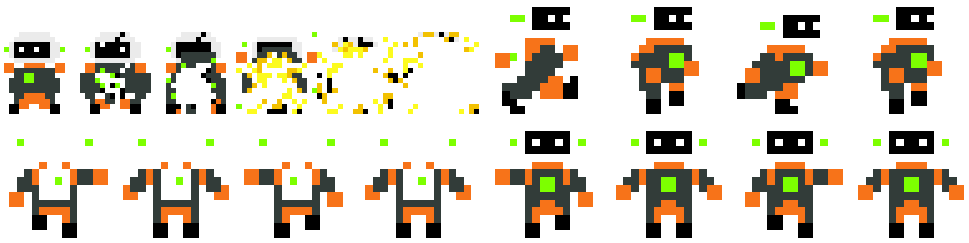
```
kill() {
    this.setVelocity(0, 0);
    this.anims.play('morte_personagem', true);
    this.once('animationcomplete', () => {
        this.destroy();
    }
}
```

Fonte: Acervo próprio

Os *sprites*, no Phaser, têm acesso ao método “once” esse método permite que uma função seja executada ao final de algum evento. No caso da destruição do personagem, o evento esperado é a animação terminar (animationcomplete).

Os *sprtesheets* do personagens estão separados em 4 (quatro) imagens, como mostra a figura 37.

Figura 37 – Spritesheet Player



Fonte: Acervo próprio

No início da vida da *scene*, dentro do método “create” serão criados os dois personagens. Esses personagens serão instanciados e sua referência armazenada dentro de duas variáveis distintas dentro da cena, como é visto na figura 38.

Figura 38 – Criação dos personagens

```
create() {
    this.playerlocal = new Player(data.playerLocal.x,
                                  data.playerLoca.y,
                                  data.playerLocal.id);

    this.playerRemoto = new Player(data.playerRemoto.x,
                                   data.playerRemoto.y,
                                   data.playerRemoto.id);
}
```

Fonte: Acervo próprio

Os dados sobre os jogadores que foram recebidos na cena anterior estão armazenados na variável “data”, e são utilizados para criação dos dois jogadores. Essa separação é necessária para que cada comando seja executado da forma correta.

3.2.3.3 A dinamite

A dinamite é a entidade que tem o poder de destruir outras entidades no jogo, todo jogador tem disponível um número ilimitado de dinamites para colocar no campo de batalha, e a partir do momento em que ele coloca uma, não existe diferença entre as dinamites colocadas por ele ou pelo adversário. Toda vez que uma é colocada no campo ela é automaticamente centralizada no espaço que foi colocada.

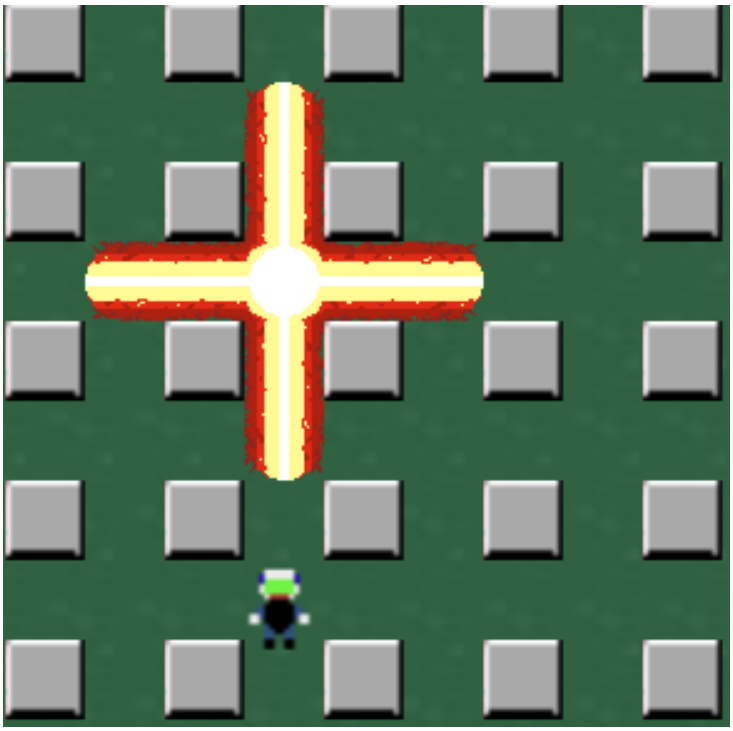
Como qualquer jogador pode ser afetado pela explosão causada por uma dinamite, existe um tempo para que seja possível se afastar, esse tempo é o tempo da animação do pavio queimando como mostra a figura 39.

Figura 39 – Frames dinamite

Fonte: Acervo próprio

Após a animação terminar, automaticamente o objeto da dinamite é destruído dando lugar ao objeto da explosão, que se não encontrar blocos no caminho, ocupa exatamente 5 (cinco) espaços do cenário como mostra a figura 40.

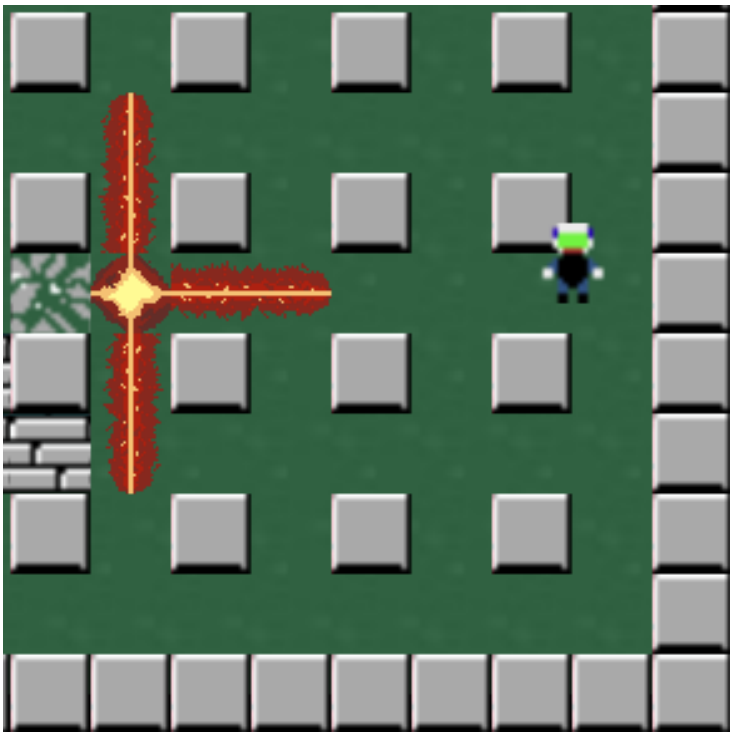
Figura 40 – Explosão



Fonte: Acervo próprio

Caso no caminho da explosão haja um bloco, a explosão é interrompida e, caso o bloco seja destrutível, esse bloco irá rodar a animação de destruição e será destruído como mostra a figura 41.

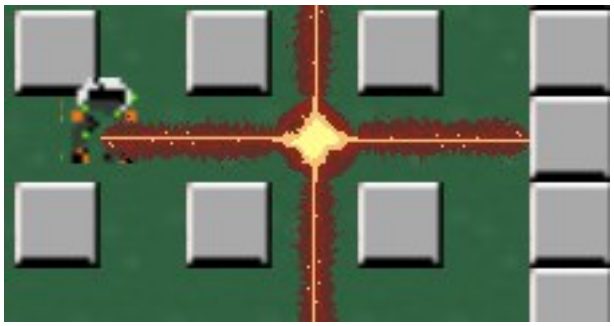
Figura 41 – Explosão destruindo bloco



Fonte: Acervo próprio

Mas se houver um jogador no caminho da explosão, ela não será interrompida, apenas irá disparar o evento de destruição desse jogador como mostra a figura 42.

Figura 42 – Destruição do jogador



Fonte: Acervo próprio

O *spritesheet* da dinamite também é carregado dentro do *load*, conforme já foi demonstrado com os blocos. Todos os *spritesheets* e imagens que serão carregados no jogo seguirão o mesmo procedimento: passando por parâmetro o endereço do arquivo e uma *string* para ser a chave que guarda esse valor. No caso da dinamite, a chave será “dinamite”.

Tanto a dinamite como a sua explosão são objetos do tipo *sprite*, mas diferente dos personagens, são as implementações simples dessa classe. O atributo “phy-

sics.add"oferece um método que cria e salva dentro da física do jogo um *sprite*. Para isso, basta que se envie por parâmetro as coordenadas, a chave da textura e o *frame* inicial da animação como demonstra a figura 9.

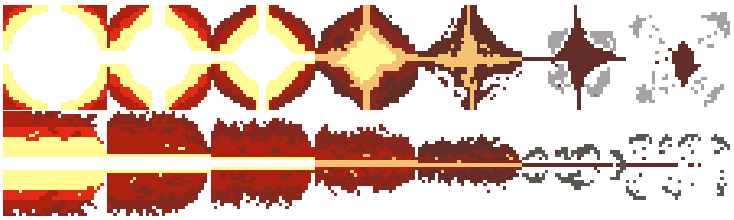
Figura 43 – Criação de um sprite

```
this.physics.add.sprite(0, 0, 'dinamite', 0);
```

Fonte: Acervo próprio

A explosão sofre um processo parecido, porém, é utilizada a mesma animação para os 4 lados como mostrado na figura 44, apenas girando o *sprite*.

Figura 44 – Sprites da explosão



Fonte: Acervo próprio

O método “setAngle” recebe como parâmetro os graus que o *sprite* deve ser rotacionado. Como a explosão tem 4 lados que são iguais porém rotacionados, basta criar 3 *sprites* que são a versão rotacionada do primeiro, como demonstra a figura 45:

Figura 45 – Adicionando um bloco ao group

```
this.physics.add.sprite(0, 0, 'explosion', 0);
this.physics.add.sprite(0, 0, 'explosion', 0).setAngle(90);
this.physics.add.sprite(0, 0, 'explosion', 0).setAngle(-90);
this.physics.add.sprite(0, 0, 'explosion', 0).setAngle(180);
```

Fonte: Acervo próprio

Outra peculiaridade da dinamite e da explosão é que seus *sprites* não são criados dentro do método “create”. Cada jogador pode colocar inúmeros explosivos durante a partida, então sua criação acontece dentro de um método criado na cena e invocado por ele. O método responsável pela criação das dinamites é o método “setBomb”, esse método usa de referência a posição do jogador e coloca uma dinamite nesse local.

3.2.4 Conexão com o servidor

Para que seja possível se comunicar com o servidor, dois *singletons* foram criados. A classe WebSocketService é um *singleton* responsável pela comunicação através do *websocket*. A conexão é feita pela biblioteca de JS “stompjs”, que retorna um

socket aberto com o servidor. Para isso é necessário enviar a URL do servidor através do método “client” como mostrado na figura 46.

Figura 46 – Singleton Websocket

```
export default class WebSocketService {
    private serverUrl = 'https://blasterman_server.com/socket';
    private static connection;

    static getInstance(){
        if (!this.connection) {
            this.connection = Stomp.client(this.serverUrl);
        }
        return this.connection
    }


}
```

Fonte: Acervo próprio

A outra classe responsável pela conexão é a ConnectionService, ela é responsável por conectar o jogador ao servidor e receber o ID que o servidor atribuiu a esse jogador. A conexão é feita utilizando a biblioteca “axios”, é ela que disponibiliza os métodos necessários para conexão Hypertext Transfer Protocol (HTTP) como mostra a figura 47.

Figura 47 – Singleton conexão HTTP

```
export class ConnectionService {
    private serverUrl = 'https://blasterman_server.com/connect';
    private conn = axios.default;
    static readonly instance = new ConnectionService();

    connectPlayer() {
        return this.conn.post(this.serverUrl)
                        .then(r => r.request.response);
    }
}
```

Fonte: Acervo próprio

3.2.5 Servidor

O Blasterman é um jogo online, significa que deve haver comunicação entre os jogadores, essa comunicação é feita através de um servidor. Ele foi escrito em Java, entretanto, como o servidor apenas atribui um ID para cada jogador na sala e

faz *broadcast* das mensagens que recebe no *websocket*, qualquer outra tecnologia com essas capacidades seria suficiente.

O servidor é composto de 4 métodos que são responsáveis por responder as requisições em 4 (quatro) *endpoints*. O método “ConnectPlayer” é responsável por receber a requisição do jogador quando entra na URL do jogo, e responder com o seu ID e posição no campo, como demonstrado na figura 48.

Figura 48 – Método ConnectPlayer

```
private static final Map<UUID, Player> playerMap = new HashMap<>();

@PostMapping("/connect")
public Object connectPlayer() {

    if(playerMap.size() < 2) {

        var p = new Player(UUID.randomUUID());

        playerMap.put(p.getPlayerId(), p);

        if(playerMap.size() == 1){
            p.setX(190);
            p.setY(48);
        }else {
            p.setX(1148);
            p.setY(554);
        }

        return p.getPlayerId().toString();
    }else {
        return "Sala cheia";
    }

}
```

Fonte: Acervo próprio

Esse método verifica se no *Map* que guarda os jogadores já tem dois jogadores. Caso não tenha, ele cria um novo objeto “Player”, um objeto responsável por guardar o ID e as coordenadas iniciais do jogador, e coloca esse objeto no *Map*. Em seguida, ele verifica se o jogador foi o primeiro a entrar no jogo, caso seja, as coordenadas do canto superior esquerdo do campo de batalha são atribuídos a ele, caso não seja o primeiro, é atribuído o canto inferior esquerdo. Caso já haja dois jogadores na partida, o servidor retorna “Sala cheia”.

O *endpoint* “/waitinig” simplesmente recebe as requisições dos jogadores que clicaram no botão “Play”. Assim que os dois jogadorem apertam, ele retorna os valores contidos no *Map* de jogadores, como mostra a figura 49.

Figura 49 – Método waiting

```
@MessageMapping("/waiting")
@SendTo("/waiting")
public Object waiting(@Payload String playerId) {

    if (playrMap.size() == 2) {
        return playerMap.values();

    } else {
        return "wait";
    }
}
```

Fonte: Acervo próprio

Os métodos de movimentação e de colocar a dinamite simplesmente fazem um *broadcast* do conteúdo da requisição. Não existe tratamento dos dados, apenas é enviada a resposta contendo o JSON que chegou pela requisição, como mostrado na figura 50.

Figura 50 – Métodos move e setBomb

```
@MessageMapping("/move") /
@SendTo("/move")
public String move(@Payload String move) {
    return move;
}

@MessageMapping("/bomb")
@SendTo("/bomb")
public String setBomb(@Payload String bomb) {
    return bomb;
}
```

Fonte: Acervo próprio

3.2.6 Cena principal

A cena principal possui todos os métodos padrões e o método responsável por criar uma dinamite, ela também possui o grupo com todos os blocos e os jogadores da partida. Porém, existe ainda uma última tarefa fundamental que ela executa que é o que torna possível a partida ser online. É a cena que é responsável por se inscrever no *websocket* para que possa prontamente executar os comandos do jogador remoto. Os comandos recebidos são 2 (dois): o comando de movimentação e o de criar uma dinamite. Essas inscrições são feitas no método “init” da cena, como demonstrando na figura 51.

Figura 51 – Inscrição no *websocket*

```
wsConnection.subscribe('/blasterman_service/bomb', r => {

    let resposta = JSON.parse(r.body);
    if (this.playerRemoto.playerId == resposta.playerId)) {

        this.setBomb(this.playerRemoto);

    }
});

wsConnection.subscribe('/blasterman_service/move', r => {

    let resposta = JSON.parse(r.body);
    if (this.playerRemoto.playerId == resposta.playerId)) {

        this.playerRemoto.move(resposta.direction);

    }
});
```

Fonte: Acervo próprio

Quando qualquer jogador envia comandos ao servidor, o servidor envia esse comando a todos os jogadores conectados na partida, sem haver nenhum tipo de filtro. Isso significa que o próprio jogador que executou o comando e enviou para o servidor também vai receber esse comando de volta. A forma de evitar que exista qualquer conflito é garantir que os comandos do jogador local serão ignorados quando voltarem do servidor.

O último elemento a ser explicado na cena é o som das explosões. O som da explosão foi encontrado e um site que disponibilizava sonoplastias gratuitas. Ele é em formato mp3 e é carregado dentro do método “preload” junto com os outros *assets*. Tal som é criado dentro do método “create”, a diferença que é usado o método específico para carregar áudio como demonstrado na figura 52.

Figura 52 – Carregamento do áudio

```
        preload() {
            this.load.audio('explosion', 'sounds/explosion.mp3');
        }

        create() {
            this.explosionSond = this.sound.add('explosion');
        }
```

Fonte: Acervo próprio

3.2.7 Configurações

Para as configurações do jogo foram especificadas no objeto de configuração. O tipo de renderização foi decidia como “CANVAS” para garantir a compatibilidade com qualquer navegador que use HTML5, as dimensões da tela tela do jogo receberam os valores 1366 para largura e 768 para altura. Por último o tipo de física do escolhida foi “Arcade” como mostra a figura 53.

Figura 53 – Arquivo de configuração do Phaser

```
const config = {
    type: Phaser.CANVAS,
    width: 1366,
    height: 768,
    physics: {
        default: 'arcade',
    }
};
```

Fonte: Acervo próprio

3.2.7.1 Configurações do TypeScript

Como todo projeto foi escrito em TypeScript, é necessário que algumas configurações sejam declaradas para que o código seja transpilado da forma correta. O código gerado, após ser transpilado, é um código JS, a versão escolhida para ser o código final foi o es6, dessa forma o código gerado é menor e se torna ligeiramente mais rápido de ser carregado do servidor. Foi declarado também que ele deve incluir na transpilação todo código que estiver dentro de “src/” como mostrado na figura 54.

Figura 54 – Arquivo de configuração do TypeScript

```
{
    "compilerOptions": {
        "target": "es6",
        "moduleResolution": "node",
        "module": "CommonJS",
        "strict": false,
    },
    "include": [
        "./src/**/*"
    ]
}
```

Fonte: Acervo próprio

3.2.7.2 Configurações do webpack

Por fim, para que o projeto fosse empacotado da forma correta, era necessário que algumas configurações diferentes da configuração padrão fossem feitas. Primeiramente o "*entry point*" foi configurado para "src/game.ts", o arquivo principal do jogo, em seguida o "*mode*" foi configurado para "*production*", o "*output*" também precisou ser alterado, uma vez que nosso diretório destino precisava ser "/docs", além disso a opção de minificação do código foi ativada, para que no final se tivesse um código mais compacto. Também foram ativados alguns *plugins* do webpack para que ele conseguisse lidar com arquivos .css e .html. O arquivo de configuração ficou como mostrado na figura 55.

Figura 55 – Arquivo de configuração do webpack

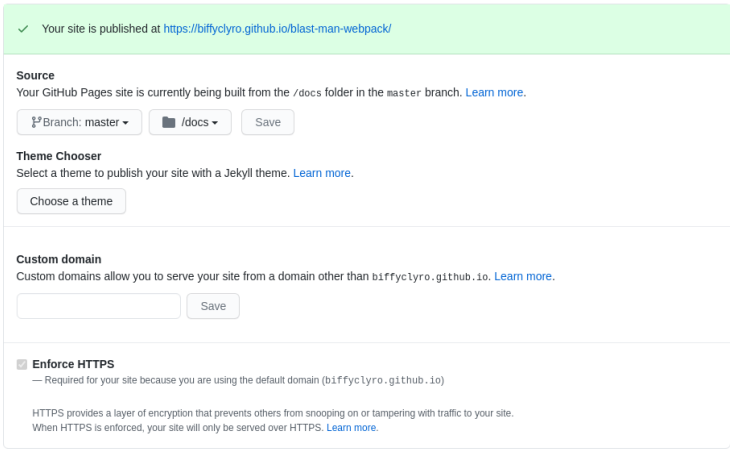
```
module.exports = {
    entry: {
        game: './src/game.ts'
    },
    mode: 'production',

    output: {
        path: path.resolve(__dirname, 'docs'),
        publicPath: './',
        filename: '[name].min.js'
    },

    optimization: {
        minimize: true,
        splitChunks: {
            cacheGroups: {
                commons: {
                    test: /[\\/]node_modules[\\/]/,
                    name: "vendors",
                    chunks: "all"
                }
            }
        }
    },
    resolve: {
        extensions: [ '.ts', '.js' ]
    },

    plugins: [
        new CopyWebpackPlugin(
        [
            {
                from: './assets',
                to: './assets',
                force: true
            },
            {
                from: './app.css',
                to: './app.css',
                force: true
            }
        ]),
        new HtmlWebpackPlugin({
            filename: 'index.html',
            template: 'index.html'
        })
    ]
};
```

Fonte: Acervo próprio

3.2.8 Github Pages

Sendo o jogo uma página *web* a sua versão final foi hospedada na ferramenta Github Pages, esse serviço é capaz de servir qualquer página estática de forma gratuita e pública. Para isso, basta que alguns pré-requisitos sejam satisfeitos: a página a ser servida deve estar dentro de um diretório chamado “docs” na raiz do projeto, o projeto precisa estar público, e precisa que seja ativada a opção no menu de configurações do projeto, como mostra a figura 56.

Figura 56 – Configuração do projeto



Fonte: Acervo próprio

É possível a utilização de um domínio próprio. No entanto como este projeto não tem fins comerciais, foi utilizado o domínio padrão. Isso tem por consequência que o domínio precisa usar Hyper Text Transfer Protocol Secure (HTTPS), uma vez que só é permitido a utilização do HTTP se for um domínio próprio. O código fonte do jogo está disponível no repositório do Github: https://github.com/Biffyclyro/blast-man-webpack.

3.2.9 Testes

Os testes foram feitos tanto no servidor local, utilizando a rede doméstica como em um servidor na *web*. Na rede local não foi detectada nenhuma anomalia, o jogo rodou de forma fluida e dentro do esperado. Na *web* foi utilizado o serviço Heroko, que disponibiliza uma hospedagem grátis para aplicações simples. Nesse cenário o jogo se comportou de forma instável, devido ao fato de a conexão com o serviço gratuito ser lenta, eventualmente se podia perceber um *lag*, porém, nada fora do esperado. Por não ser um serviço pago, é compreensível que nem sempre seja bom o desempenho da conexão.

3.2.10 Conclusão do capítulo

Após o processo de *build* o arquivo do jogo ficou com 1.3 *Megabyte*, incluso, além do código fonte, os arquivos de imagem e o arquivo de som. Nessa versão do jogo o código TypeScript foi transpilado para EcmaScript 6. Foi feito o teste de se transpilar para a versão anterior, no intuito de se ter uma maior compatibilidade com navegadores em versões antigas, porém, o tamanho final foi para 1.5 *Megabyte*. Por menor que possa parecer essa diferença, foi decidido que o código final seria na versão 6, pois, além de ter um tamanho de arquivo menor, a compatibilidade que se esperava da versão anterior não foi alcançada, uma vez que alguns módulos que o Phaser utiliza só funcionam em navegadores que dão suporte ao EcmaScript 6. O código final é uma versão minificada do código fonte, essa é a opção padrão do *webpack* quado o empacotamento é feito no modo *"production"*. Essa transformação também contribuiu para diminuição do tamanho do arquivo, fazendo com que todos os códigos ficassem em um único módulo JS.

4 CONCLUSÃO

A partir do resultado final, pode-se observar que é perfeitamente viável a construção de um jogo *multiplayer* apenas com tecnologias de desenvolvimento *web*. Seria possível a criação de jogos mais robustos e complexos da mesma forma sem encontrar maiores dificuldades.

Com efeito, algumas das tecnologias utilizadas serviram para auxiliar o desenvolvimento e a melhorar o projeto como um todo. Entretanto, apenas o uso Phaser com JS já resultaria em um jogo com as mesmas funcionalidades.

O Blasterman apresentou o comportamento esperado, possibilitando uma partida entre dois jogadores sem que fosse necessário qualquer tipo de instalação de recurso além do navegador. Ele se comportou da mesma forma que qualquer aplicação *web* convencional, e em nenhum momento sofreu por limitações tecnológicas desse contexto.

Do lado do servidor, ficou claro que, por mais que as soluções grátis de hospedagem sirvam para testes, é necessário um serviço que garanta estabilidade da conexão. Os testes locais não apresentaram problemas, mas quando o jogo rodou utilizando um servidor remoto houve problemas advindos da inconstância da comunicação.

O jogo nesses moldes serve como um teste de conceito. Caso se quisesse utilizar o mesmo para fins comerciais, seriam necessários ajustes. O servidor precisaria suportar mais de uma sala, e seria crucial que ele validasse a comunicação ao invés de fazer um *broadcast*. As partidas também simplesmente terminam com um jogador sendo destruído, e é necessário reiniciar o servidor para se jogar novamente.

O que se percebe após essa implementação, é que da mesma forma que outros tipos de sistemas que estão migrando para sistemas *web*, jogos também podem facilmente seguir esse caminho. Essa forma de criar jogos possibilita que automaticamente o jogo seja compatível com vários dispositivos diferentes, sem que seja necessário se preocupar com as peculiaridades de cada plataforma. O resultado pode ficar muito semelhante a um jogo feito para o sistema de forma nativa, fazendo com que o usuário muitas vezes nem seja capaz de perceber a diferença.

REFERÊNCIAS

BRADFIELD, C. **Godot Engine Game Development Projects**. [S.l.]: Packt Publishing Ltd, 2018.

CRUZ LOPES, M. A. da. Bomberman as an artificial intelligence platform. , [S.l.], 2016.

CURRAN, K.; GEORGE, C. The future of web and mobile game development. **International Journal of Cloud Computing and Services Science**, [S.l.], v.1, n.1, p.25, 2012.

DAM, A. V. et al. **Computer Graphics**: principles and practice. [S.l.]: Addison-Wesley, Reading, 1990.

DESMOND FRANCIS, D.; WILLS, A. C. **Objects, components, and frameworks with UML**: the catalysis approach. [S.l.]: Addison-Wesley, 1999.

DIRKSEN, J. **Learning Three. js**: the javascript 3d library for webgl. [S.l.]: Packt Publishing Ltd, 2013.

FAAS, T. **An introduction to HTML5 game development with Phaser. js**. [S.l.]: CRC Press, 2017.

GOLDSTONE, W. **Unity game development essentials**. [S.l.]: Packt Publishing Ltd, 2009.

LEBEL, S. Graphical technologies, innovation and aesthetics in the video game industry. **TECHNOLOGY EVOLUTION**, [S.l.], p.79, 2013.

MICROSOFT. **Documentation**. Acessado em setembro/2020, https://www.typescriptlang.org.

RUTLEDGE, M. **PHP game programming**. [S.l.]: Premier Press, 2004.

STORM, P. **Phaser - A fast, fun and free open source HTML5 game framework**. Acessado em junho/2020, https://photonstorm.github.io/phaser3-docs/index.html.

WEBSOCKET.ORG. **Documentation**. Acessado em setembro/2020, https://webpack.js.org/concepts/.

WILLIAMS, A. **History of digital games**: developments in art, design and interaction. [S.l.]: CRC Press, 2017.