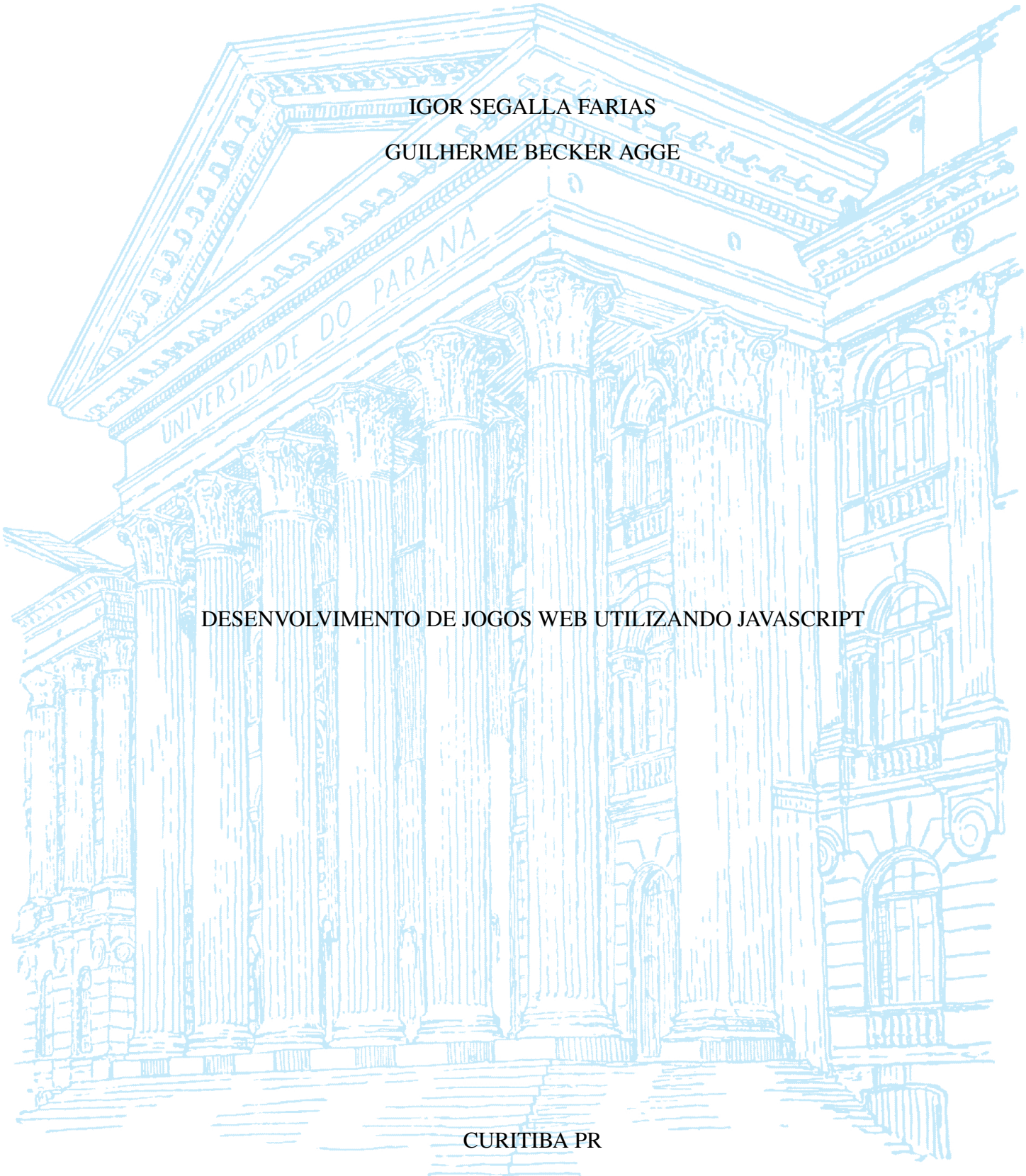
UNIVERSIDADE FEDERAL DO PARANÁ

IGOR SEGALLA FARIAS

GUILHERME BECKER AGGE



DESENVOLVIMENTO DE JOGOS WEB UTILIZANDO JAVASCRIPT

CURITIBA PR

2022

IGOR SEGALLA FARIAS
GUILHERME BECKER AGGE

DESENVOLVIMENTO DE JOGOS WEB UTILIZANDO JAVASCRIPT

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Bruno Müller Junior.

CURITIBA PR

2022

LISTA DE FIGURAS

3.1 Tela inicial do jogo Mazogs. 13
3.2 Baú a ser encontrado e trazido de volta pelo jogador. 14
3.3 Caminho mostrado pelas entidades indicando a direção correta para a saída do labirinto e rastros deixado pelo deslocamento do jogador. 15
3.4 Jogador pronto para enfrentar uma Mazogs, empunhando sua espada. 16
3.5 Tela final, com o jogador chegando a saída do labirinto carregando o baú de tesouro. 16

4.1 À esquerda, protótipo de jogo implementado durante o tutorial da engine Phaser 3, mostrando o personagem pulando. À direita, versão modificada para o nosso jogo. 20
4.2 Exemplo de texto sendo renderizado. 22
4.3 Exemplo de imagens sendo renderizadas. 22
4.4 Exemplo do *sprite sheet* do personagem. 23
4.5 Arquivo utilizado para representar o mapa do jogo. 26
4.6 Labirinto gerado a partir do arquivo texto. 26
4.7 Fluxo entre cliente e servidor para a sincronização de entidades. 30
4.8 Fluxograma para detectar colisão no servidor. 31

LISTA DE TABELAS

4.1 Tabela mostrando uma comparação entre as três *game engines* finais. 19
4.2 Tabela mostrando as funções padrão da Phaser para a *scene*. 20
4.3 Mensagens definidas para a comunicação entre servidor e cliente.. 28

SUMÁRIO

1 INTRODUÇÃO 8
1.1 MOTIVAÇÃO 8
1.2 PROPOSTA 8
1.3 DESAFIO 9
1.4 CONTRIBUIÇÃO 9
1.5 ORGANIZAÇÃO DO TEXTO 9
2 REVISÃO BIBLIOGRÁFICA 10
2.1 DEFINIÇÃO DE JOGOS 10
2.2 JOGOS NA SOCIEDADE 10
2.3 EVOLUÇÃO DOS JOGOS DIGITAIS 11
2.4 JOGOS PARA PLATAFORMA *WEB* 11
3 MAZOGS 13
4 DESENVOLVIMENTO 17
4.1 ESCOLHA DA GAME ENGINE 17
4.2 FUNCIONAMENTO DA ENGINE 19
4.3 RENDERIZAÇÃO 21
4.4 JOGABILIDADE 23
4.4.1 MOVIMENTAÇÃO DO PERSONAGEM 23
4.4.2 COLISÃO ENTRE ENTIDADES 24
4.4.3 GERAÇÃO DO LABIRINTO 25
4.4.4 CRIAÇÃO DE ENTIDADES 27
4.5 MODO MULTIJOGADOR 27
4.5.1 TROCA DE MENSAGENS 27
4.5.2 MAPA DO LABIRINTO 28
4.5.3 SINCRONIZAÇÃO DE ENTIDADES 28
4.5.4 COLISÃO ENTRE ENTIDADES 30
5 CONCLUSÃO 32
5.1 TRABALHOS FUTUROS 32
REFERÊNCIAS 33

RESUMO

Este trabalho se dedica ao estudo de motores de jogo para a linguagem de programação JavaScript, visando a plataforma *web*. Os principais motores de jogo para JavaScript de código aberto foram comparados, e um deles foi escolhido para que se fosse implementado um jogo simples em duas dimensões nele. Por fim, utilizou-se da biblioteca Socket.IO para realizar a implementação de um modo multijogador *online*, testando a junção desta biblioteca de comunicação com o motor de jogo escolhido, o Phaser3, com grande sucesso.

Palavras-chave: jogos. web. javascript.

ABSTRACT

This paper is dedicated to studying JavaScript game engines, focusing on the web platform. The main open-source JavaScript engines were compared, and one of them was chosen to create a simple bidimentional game with it. Finally, the Socket.IO library was used to implement an online multiplayer mode for this game, testing the integration between this widely used communication library, and the game engine chosen, Phaser3, with great sucess.

Keywords: games. web. javascript.

AGRADECIMENTOS

Este trabalho é dedicado a nossas famílias, que nos apoiaram no decorrer não só da produção deste trabalho, mas de todo o curso de Ciência da Computação, e no resto de nossas vidas. Gostaríamos também de agradecer à Ana Paula Zaniolo, pelas revisões do nosso texto, nos ajudando a corrigir erros ortográficos e a deixar nosso texto um pouco mais fácil de entender. Por fim, gostaríamos de agradecer ao professor Bruno Müller, por sugerir o jogo Mazogs, que acabou sendo muito produtivo para este estudo, e por nos auxiliar e guiar durante toda a produção deste trabalho. Sem ele este trabalho não seria possível. Agradecemos também aos professores Andrey Pimentel e Armando Delgado, por aceitarem participar da banca e avaliarem este trabalho.

1 INTRODUÇÃO

Motores de jogo, ou *Game Engines*, são bibliotecas que oferecem ferramentas e um conjunto de recursos para o o desenvolvimento de um jogo. Estes motores oferecem recursos como tratamento de *Input* e *Output* (I/O), renderização 3D e 2D, som e física (LEWIS e JACOBSON, 2002). Com estes recursos, o desenvolvedor consegue focar na construção da jogabilidade do jogo em si, obtendo um desenvolvimento mais simples e rápido.

As *Game Engines* não tem seu uso limitado somente à área de jogos recreativos. Pode-se encontrar jogos como o *EducaTrans*, um jogo de educação de trânsito (ASSIS et al., 2006); jogos de simulação como o *Flight Simulator*, onde uma situação real é simulada através de modelos matemáticos (ASSIS et al., 2006) e também em pré-visualização de cenas de cinema e televisão com o motor *Unreal Engine* (ESTEVES, 2021). Logo, seu uso pode ser encontrado em áreas educacionais, simulações, aplicações cinematográficas e jogos recreativos, que é o mais comum.

Este trabalho tem como objetivo comparar diferentes *Game Engines* para interface *web*, levantando os pontos fortes e fracos e assim chegar à escolha de uma *game engine* mais apropriada para o desenvolvimento deste trabalho.

1.1 MOTIVAÇÃO

Atualmente existem muitas bibliotecas para desenvolvimento de jogos, o que torna difícil escolher a mais apropriada, em especial para o nosso projeto. Como todas essas bibliotecas têm recursos muito parecidos, o que acaba diferenciando uma *game engine* de outra são questões mais técnicas, ou até mesmo a documentação, quando existente.

Para interface *web*, JavaScript é a linguagem mais comum de ser escolhida para o desenvolvimento. JavaScript é uma linguagem de programação interpretada estruturada, de *scripting* em alto nível com tipagem dinâmica fraca e multi paradigma[1]. Sua aplicação pode ser encontrada tanto para o *backend*, como o servidor de um jogo multijogador, como também no *front-end*, que é o jogo propriamente dito, onde são desenhados os gráficos e processada a lógica do jogo.

A grande vantagem de se desenvolver um jogo em JavaScript é a possibilidade do jogo ou aplicação funcionar em navegadores *web*, ou dispositivos móveis. Além disso, JavaScript é uma linguagem relativamente simples para se desenvolver e permite testes quase em tempo real, por rodar no próprio navegador *web*.

1.2 PROPOSTA

Apresentar diferentes *Game Engines* da linguagem JavaScript e implementar uma releitura do jogo *Mazogs*, original dos anos 80, adicionando recursos e usufruindo de tecnologias atuais.

Foi decidido também adicionar o modo multijogador ao jogo final, a fim de avaliar a facilidade da integração de outras bibliotecas à *Game Engine* escolhida.

[1]Flanagan, David; Ferguson, Paula (2002). JavaScript: The Definitive Guide 4th ed. [S.l.]: O'Reilly Associates. ISBN 0-596-00048-0

1.3 DESAFIO

Implementar o modo multijogador ao jogo escolhido para o desenvolvimento do trabalho. Como as *Game Engines* não oferecem uma maneira nativa de integrar o jogo com a rede, e também não oferecem protocolos de comunicação, a implementação deve ser feita por conta do desenvolvedor, utilizando outros meios através do JavaScript.

Ao integrar o jogo com a rede e protocolos de comunicação, deseja-se que seja possível a participação de diferentes usuários em uma única sessão, permitindo o modo cooperativo, visando deixar o jogo mais dinâmico e divertido.

1.4 CONTRIBUIÇÃO

Espera-se que este trabalho contribua com a sociedade ao aprofundar o entendimento daqueles que desejam desenvolver jogos com gráficos e uma física simples em JavaScript.

Também, que mostre uma alternativa simples para o desenvolvimento deste tipo de jogo, que comporta sistemas multiusuários em tempo real (multijogadores, no caso de jogos eletrônicos).

Além disso, este texto pode ser útil às pessoas que queiram saber quanto esforço é necessário para fazer um jogo em JavaScript. Aliadas a estudos sobre outras linguagens de programação e ferramentas, as informações deste trabalho podem ajudar a decidir se JavaScript é uma alternativa viável, ou até mesmo, se é a melhor alternativa.

1.5 ORGANIZAÇÃO DO TEXTO

O restante deste texto está dividido da seguinte forma: no capítulo 2, Revisão Bibliográfica, é realizada uma introdução à área de jogos e como ela esteve presente em nossa sociedade, até atualmente, com a chegada dos jogos digitais. No capítulo 3 é detalhado o funcionamento do jogo Mazogs, o qual foi escolhido como referência para o desenvolvimento do jogo deste trabalho, e, por fim, no capítulo 4 será abordado todo o desenvolvimento do trabalho, tanto a parte de pesquisa, quanto a parte de desenvolvimento técnico.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo procura contextualizar o tema do presente texto, jogos. Será apresentada a definição de jogos, qual seu papel na sociedade, como foi sua evolução até os dias atuais, e, por fim, como eles se relacionam com a interface *web*, o foco do trabalho.

2.1 DEFINIÇÃO DE JOGOS

Uma definição simples para jogo, é que ele seja interativo, tenha objetivos e uma competição (CRAWFORD, 2003). Apesar de ser uma definição simples e muitos jogos se enquadrarem nessas características, não é suficiente para generalizar a toda categoria de jogo. Outras características de um jogo é que ele possua regras fixas, resultado variável, consequências negociáveis e ligação do resultado com o esforço do jogador (JUUL, 2009). Uma definição mais detalhada sobre jogos é a seguinte:

> "Jogos são atividades sociais e culturais voluntárias, significativas, fortemente absorventes, não-produtivas, que se utilizam de um mundo abstrato, com efeitos negociados no mundo real, e cujo desenvolvimento e resultado final é incerto, onde um ou mais jogadores, ou equipes de jogadores, modificam interativamente e de forma quantificável o estado de um sistema artificial, possivelmente em busca de objetivos conflitantes, por meio de decisões e ações, algumas com a capacidade de atrapalhar o adversário, sendo todo o processo regulado, orientado e limitado, por regras aceitas, e obtendo, com isso, uma recompensa psicológica, normalmente na forma de diversão, entretenimento, ou sensação de vitória sobre um adversário ou desafio."(XEXÉO et al., 2017).

2.2 JOGOS NA SOCIEDADE

Jogos estiveram presentes em todas as civilizações e tiveram em todas um papel muito importante. Também foram espalhados pelas diferentes culturas no mundo todo, e seguem até hoje se transformando, se adaptando e evoluindo.

SERRA (1999) menciona que o jogo é um fenômeno universal, presente em todas as épocas e civilizações. A permanência do lúdico em todo o percurso histórico e civilizacional, no mundo das crianças, dos jovens e dos adultos, é um bom indicador da sua importância.

Os jogos são muito utilizados no ambiente educativo, onde tem uma participação fundamental no desenvolvimento de uma criança. Os jogos promovem interações sociais, desenvolvem a imaginação, ensinam a seguir regras, e por fim, podem ser útil para ensinar algo. Um exemplo disso é o jogo de tabuleiro GO, que o imperador chinês Yao (2337 – 2258 a.C.) criou para ensinar seu filho disciplina, concentração e equilíbrio (BANASCHAK, 1999), utilizando-se do para o treinamento de estratégia. Baseado nas afirmações acima, pode-se concluir que os jogos sempre tiveram um papel diverso, seja para o divertimento, entretenimento, educação ou autoconhecimento.

O maior exemplo que temos de jogos tradicionais no ambiente educativo, é a disciplina de Educação Física presente nas escolas. Atualmente, Educação Física é disciplina obrigatória na grade curricular da educação básica, segundo o Artigo 26 da lei nº 9.394/96, de 20 de Dezembro de 1996 (Brasil, 1996). Amarelinha, "bets", pique bandeira e bola de gude são exemplos de jogos

tradicionais encontrados nas práticas sociais das crianças (FREIRE e GUERRINI, 2017) e são encontrados também na disciplina de Educação Física.

2.3 EVOLUÇÃO DOS JOGOS DIGITAIS

Com a popularização dos computadores e o barateamento dos circuitos eletrônicos em geral, os jogos tiveram grande evolução. Máquinas específicas acabaram sendo criadas para jogar esses jogos, os consoles (ou "video games", como conhecidos popularmente aqui no Brasil), novas mecânicas foram sendo desenvolvidas e novas aplicações foram surgindo.

O desenvolvimento de jogos eletrônicos se deu a partir da década de 1950, na Terceira Revolução Industrial, fase marcada pelo aprimoramento e novos avanços no campo tecnológico. Inicialmente, os jogos eletrônicos surgiram como experiência dos desenvolvedores de software, e acabaram mais tarde ganhando espaço e projeção na área de entretenimento (SANTOS, 2015). Até então, os jogos não utilizavam um *hardware* específico. O primeiro jogo eletrônico criado na história foi feito pelo físico *William Higinbotham* em 1958, era jogado através de um osciloscópio e se chamava *Tennis for Two* (AMORIN, 2006). A mecânica deste primeiro jogo ainda era muito simples e funcionava em um computador analógico.

Na década de 1970 os "computadores pessoais"se tornaram realidade e os primeiros jogos eletrônicos começaram a ser desenvolvidos. Estudantes e pesquisadores começaram a desenvolver os primeiros jogos para os computadores pessoais, tirando proveito de todos os recursos que só os computadores até então ofereciam (LEITE, 2003), como a oferta de um maior poder computacional e acessórios que foram surgindo com o tempo, como *mouse*, teclado e caixas de som.

Os jogos eletrônicos vem conquistando um espaço cada vez mais forte na sociedade. Em uma pesquisa realizada pela PricewaterhouseCoopers (PWC) em 2021[1], a indústria de jogos cresceu 10% no ano de 2020. Este crescimento em 2020 esteve diretamente relacionado à pandemia, com uma rápida mudança para serviços de conteúdo digital e também com as pessoas passando mais tempo em casa conectadas (PWC, 2021). Para os anos seguintes, previsões apontam um crescimento de 4,4% ao ano até 2025. No cenário Brasileiro, segundo Grupo de Estudos e Desenvolvimento da Indústria de Games (FLEURY et al., 2014), os jogos eletrônicos são os que apresentaram uma maior taxa de crescimento no mercado se comparado a outros meios de entretenimento.

2.4 JOGOS PARA PLATAFORMA *WEB*

Apesar de os primeiros jogos terem sido desenvolvidos para computadores com outras finalidades, foram as máquinas dedicadas, como os consoles e *arcades* (conhecidos no Brasil como fliperamas) que popularizaram de fato os jogos eletrônicos com o público.

Com a chegada dos *smartphones* e o desenvolvimento de aplicativos para estes dispositivos, a popularização dos jogos se tornou ainda maior. Hoje em dia, qualquer pessoa com um celular simples pode ter acesso a milhares de estilos de jogos diferentes, evitando a antiga necessidade de ter um vídeo-game específico ou então um computador potente (SANTOS, 2015) para poder jogar jogos eletrônicos.

[1]TAKAHASHI, Dean. PwC: Games grew 10% in 2020 and will grow 4.4% per year through 2025. VentureBeat, 2021. Disponível em: https://venturebeat.com/2021/07/11/pwc-games-grew-10-in-2020-and-will-grow-4-4-per-year-through-2025/. Acesso em: 2 de maio de 2022.

Atualmente, com o *smartphone* sendo o principal dispositivo eletrônico da maioria da população, os jogos passaram também a competir bastante nesta plataforma, exigindo assim novas tecnologias, novas ferramentas e novos estilos de jogos.

Nesse sentido, o desenvolvimento para interface *web* se torna algo muito interessante, uma vez que os dispositivos móveis são bem adaptados para navegadores de internet e o desenvolvimento *web* há muitos anos já prioriza dispositivos móveis como um dos principais ambientes em que as aplicações serão utilizadas.

3 MAZOGS

Para este trabalho, o jogo Mazogs foi escolhido para ser implementado e assim demonstrar os conceitos estudados. O Mazogs foi um jogo publicado pela primeira fez em 1982 para o computador pessoal Timex Sinclair 1000, e depois para o ZX81, também da Sinclair.

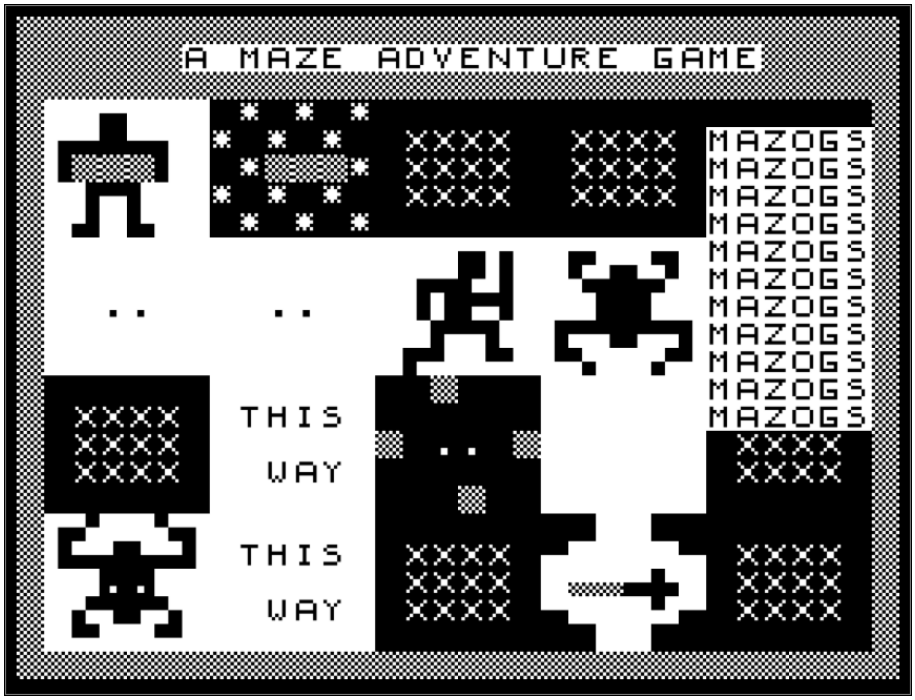


Figura 3.1: Tela inicial do jogo Mazogs.

Neste jogo, o jogador se aventura por um labirinto para coletar um baú de tesouro e levá-lo até a saída do labirinto. Pelo fato da tela ser pequena, a visão do jogador é bem limitada, tornando o jogo mais desafiador e divertido.

Os controles do jogo são bem simples: o jogador pode deslocar o personagem vertical e horizontalmente usando duas teclas para cada eixo, fazendo o personagem andar pelo labirinto. Fora as ações de andar, o jogador tem apenas outros dois comandos: um deles para visualizar um mini-mapa do labirinto, e outro comando para abandonar o jogo em andamento.

O jogo conta com um indicador de passos, que mostra a quantidade de deslocamentos que o jogador realizou, e outro indicador de energia. A cada passo dado pelo jogador, ele consome uma unidade de energia. Caso a quantidade de energia chegue a zero, o jogo é encerrado e o jogador perde a partida.

Ao iniciar a partida, o jogo dá uma dica ao jogador de quantos passos serão necessários para chegar à saída do labirinto. Assim, o jogador consegue ter uma noção da distância para a saída e de como ele deve controlar sua quantidade de energia.

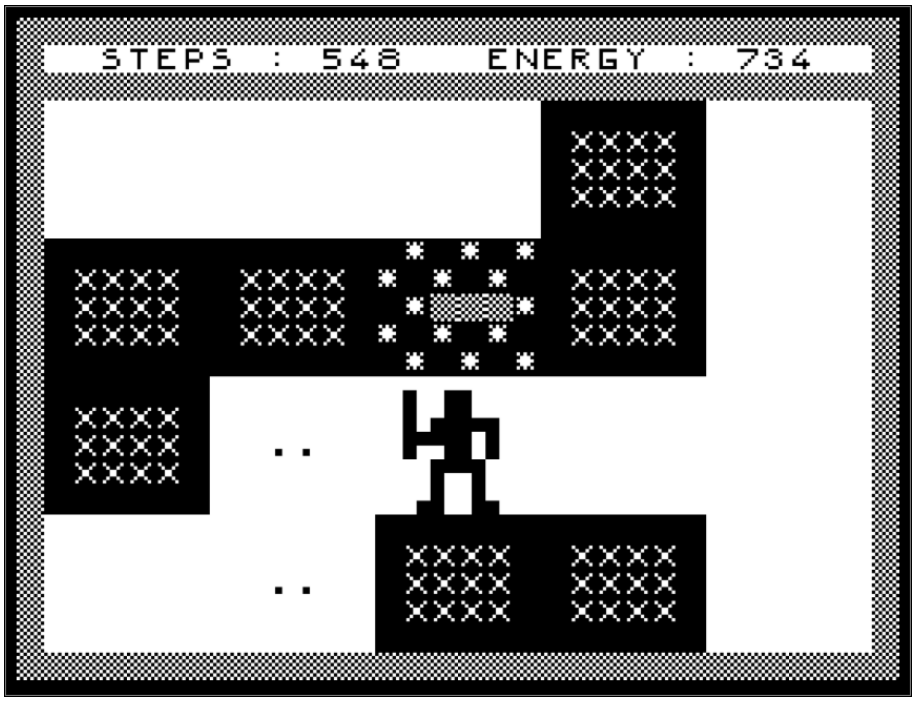


Figura 3.2: Baú a ser encontrado e trazido de volta pelo jogador.

Para auxiliar o jogador a encontrar o baú de tesouro, existem algumas entidades no labirinto que quando acionadas, indicam a direção que o jogador deve seguir para chegar à saída. Estas indicações, são simples textos com a frase "*this way*"que aparecem no mapa, possibilitando saber qual a direção correta.

Também, quando o jogador caminha pelo labirinto ele deixa rastros da sua trajetória, como se fossem pegadas. Estes rastros podem ser utilizados pelo jogador para saber por quais corredores ele já passou e evitar passar pelo mesmo caminho repetidamente, considerando que aquele caminho não o levará à saída do labirinto.

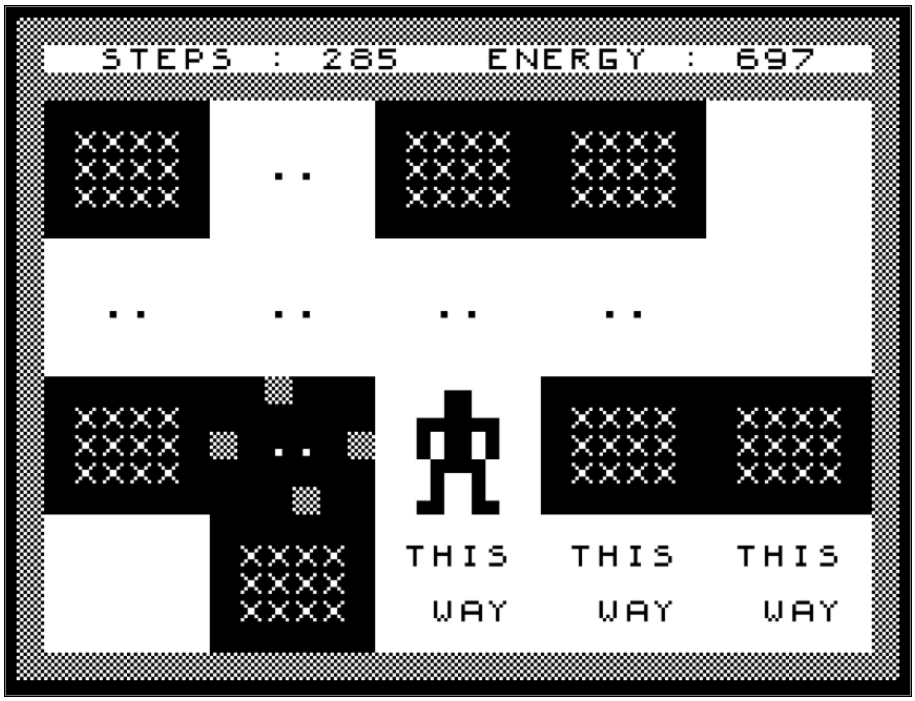


Figura 3.3: Caminho mostrado pelas entidades indicando a direção correta para a saída do labirinto e rastros deixado pelo deslocamento do jogador.

Além disso, existem monstros espalhados pelo mapa, as Mazogs, que, ao colidirem com o jogador, tem metade de chance de causar um fim de jogo. Isso pode ser evitado caso o jogador encontre uma espada no labirinto e equipe-a, fazendo com que o jogador ganhe a batalha contra o próximo monstro encontrado. Caso o jogador tenha uma espada equipada e mate a Mazogs, a espada é descartada logo em seguida, e o jogador fica vulnerável novamente. Quando o jogador vence a batalha contra as Mazogs, ele ganha uma quantidade de energia, fazendo com que aumente a quantidade de passos possíveis para chegar a saída do labirinto. As Mazogs adicionam um elemento de sorte (ou azar), que deixa o jogo mais dinâmico e emocionante.

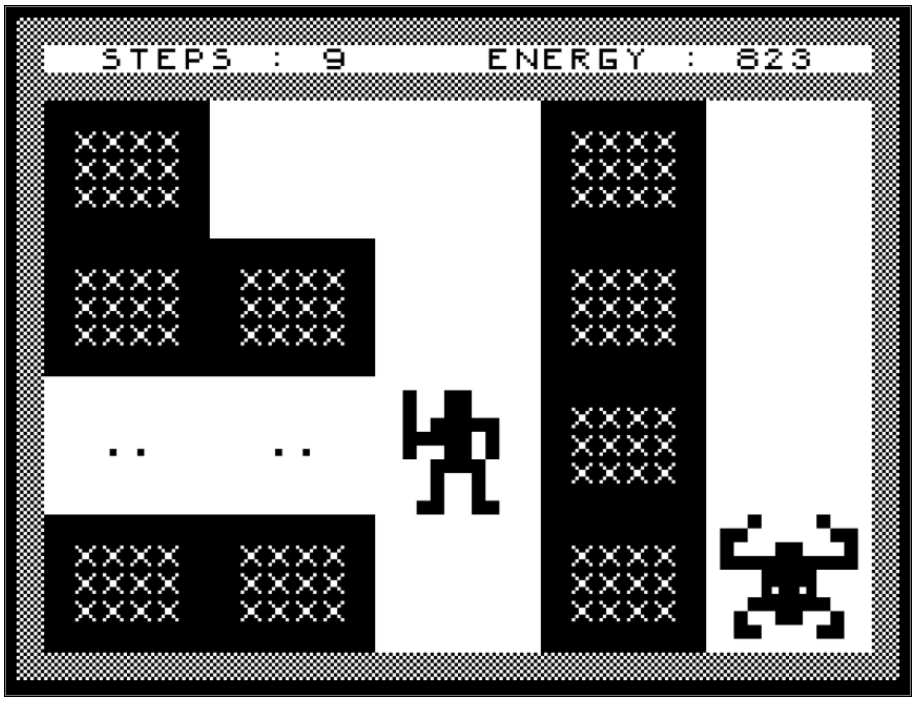


Figura 3.4: Jogador pronto para enfrentar uma Mazogs, empunhando sua espada.

Existe, porém, uma limitação, que faz com que o jogador só possa carregar um item de cada vez: o baú de tesouro ou uma espada. Desta forma, caso o jogador esteja carregando o baú, ele deve jogar com mais cuidado, já que só poderá garantir uma vitória contra a próxima Mazogs se trocar o baú por uma espada, temporariamente.

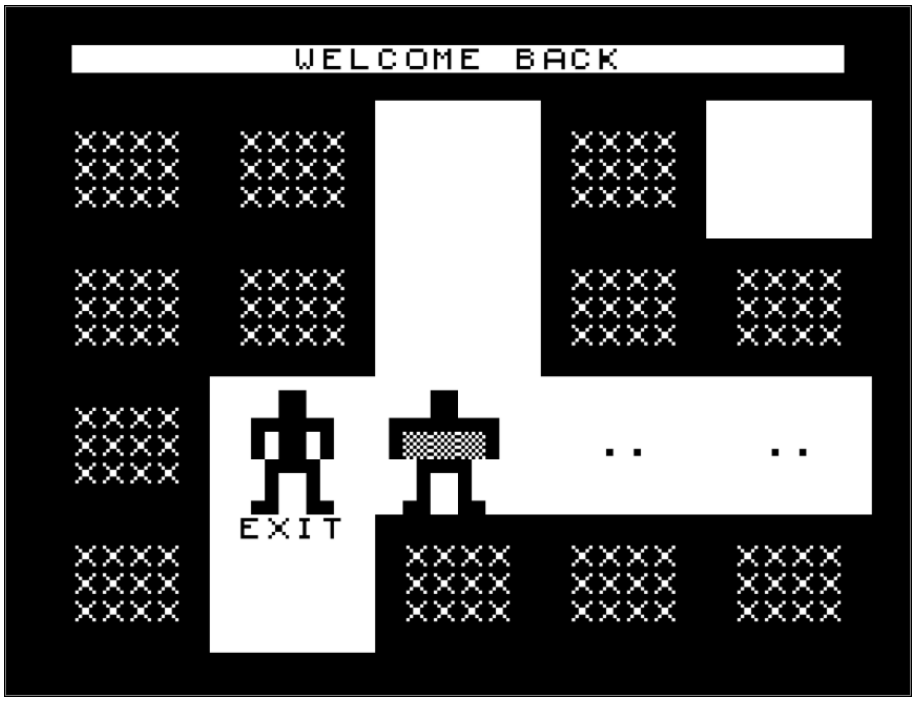


Figura 3.5: Tela final, com o jogador chegando a saída do labirinto carregando o baú de tesouro.

4 DESENVOLVIMENTO

Este capítulo será usado para detalhar os desafios envolvidos durante o desenvolvimento da releitura do jogo Mazogs. Como mencionado anteriormente, o objetivo deste trabalho é avaliar possíveis *Game Engines* para a linguagem de programação JavaScript, escolher a *Game Engine* mais apropriada para o contexto, e assim, realizar a implementação do jogo Mazogs para plataforma *web*.

4.1 ESCOLHA DA GAME ENGINE

Para implementar o jogo Mazogs em JavaScript, foram comparadas diversas *Game Engines* para encontrar a mais apropriada para o desenvolvimento do jogo. Realizando uma pesquisa rápida pela *internet*, é possível encontrar uma enorme quantidade de *Game Engines* para JavaScript, sendo que a maioria delas apresentam diversos níveis de funcionalidades e facilitadores para o desenvolvimento do jogo (como documentação, por exemplo).

Inicialmente foi realizada uma pesquisa na ferramenta de busca Google para o conhecimento de todas as *game engines* possíveis. Como o Google prioriza resultados mais populares ou até mesmo propagandas, muitas *game engines*, principalmente as menos conhecidas, acabaram não sendo exibidas e o resultado acaba muito concentrado em *game engines* mais populares e já consolidadas na comunidade.

De modo a diversificar a pesquisa e ter uma visão mais técnica das possibilidades, foi realizada uma segunda pesquisa diretamente no site GitHub. O GitHub é um site para o versionamento de códigos-fonte, também utilizado para o compartilhamento de projetos com código aberto, ou seja, projetos onde qualquer pessoa pode contribuir para o desenvolvimento. Como o site é focado para desenvolvedores, a pesquisa acaba sendo menos genérica que a realizada pelo buscador Google. Para isso, foi utilizado o recurso de coleções que o GitHub oferece, sendo uma lista de repositórios de um tema e assunto específico. No caso, a coleção utilizada foi a "JavaScript Game Engines"[1]. Desta coleção, foram consideradas as *game engines* mais relevantes dentro do site e que haviam atualizações recentes. Ao final, foram selecionadas as seguintes *game engines*: PixiJs, Phaser3, MelonJS, Kiwi.js, Crafty, Matter-js, Stage.js, Cocos2d, PlayCanvas, Lychee.js, Babylon.js, Panda Engine, QICI Engine, WhitestormJS, Goo Engine, Planck.js, Isogenic Game Engine, GDevelop, Three.js, Impact, Taro, Kaboom e Kontra.js.

Conduzindo uma análise inicial de cada uma dessas *Game Engines* selecionadas, foi possível notar que algumas eram limitadas à parte de renderização de um jogo ou então simulações físicas. Como o objetivo deste trabalho era desenvolver um jogo completo, seriam necessários recursos que essas *Game Engines* não continham, como tratamento de I/O (*Input* e *Output*), reprodução de efeitos sonoros e colisão entre objetos. Logo, por estas *Game Engines* não oferecerem o que seria necessário para a implementação do Mazogs, elas não foram consideradas para uma comparação posterior.

Além disso, algumas dessas *Game Engines* exigiam licenças pagas e outras focavam em jogos 3D ou jogos mais complexos. Como o objetivo é estudar *Game Engines* para criação de jogos simples em duas dimensões (2D) e que sejam de código-aberto, a seleção foi limitada a estes critérios.

No fim, apenas sete *Game Engines* foram consideradas. Sendo elas:

[1]JavaScript Game Engines. GitHub, 2022. Disponível em: https://github.com/collections/javascript-game-engines. Acesso em: 4 de maio de 2022.

- Phaser3: uma das *Game Engines* mais populares para o desenvolvimento de jogos em JavaScript, possui grande comunidade ativa e diversos tutoriais;
- MelonJS: descrita como uma *game engine* moderna e rápida, possui muitas ferramentas auxiliares para o desenvolvimento do jogo;
- Kiwi.JS: se autodeclarando a "engine mais fácil de usar para jogos em HTML5", possui foco em desempenho;
- Crafty: uma *game engine* simples, feita para ajudar a criar jogos em JavaScript de forma estruturada;
- Stage.JS: provê um modelo de árvore para organização dos objetos do jogo, foca mais em renderização;
- PlayCanvas: game engine de HTML5 e WebGL bem completa. Possui um editor gráfico a parte, de código-aberto;
- Impact: *game engine* mais antiga (última versão é de 2014), mas mesmo assim parece ser uma opção interessante para desenvolvimento de jogos.

Desta lista final, vale destacar a Phaser3, a ImpactJS e a MelonJS, as três engines que foram escolhidas para serem comparadas na prática, no final do processo de decisão. Estas engines foram escolhidas por terem documentação robusta, diversos exemplos prontos comparáveis ao jogo objetivo do trabalho, e pela aparente facilidade de desenvolvimento nessas engines.

Para testar as engines finais, foi realizada uma implementação de um jogo simples, que deveria ser o mesmo nas 3 engines. Esse jogo basicamente só continha a movimentação do personagem do jogador nas 4 direções (cima, baixo, direita e esquerda) e as paredes, que deveriam "parar"(limitar) o movimento do personagem. Com isso, ao menos a mecânica mais básica do movimento no labirinto do jogo foi testada, mas mesmo esse teste simples revelou bastante sobre as 3 engines.

Com a MelonJS, é notável que há uma dependência muito forte de uma ferramenta externa para edição de *level*[2] do jogo. Nesta ferramenta é possível também editar as imagens, adicionar física, adicionar entidades e configurar o mapa do jogo. Como esta ferramenta externa acaba sendo um requisito, exige mais etapas no desenvolvimento de um jogo, independente da complexidade. Além disso, não existe uma abundância de tutoriais oficiais disponibilizados pela MelonJS, o que acaba dificultando o aprendizado.

Em contraponto, a Phaser3 oferece os mesmos recursos da *game engine* MelonJS, também focando em jogos de duas dimensões (2D), porém com algumas vantagens. Por ser uma biblioteca bem popular, a comunidade é bem ativa e há uma grande quantidade de material espalhado pela internet, facilitando o aprendizado. Além disso, um dos pontos fortes dessa *game engine* é a simplicidade. Com poucas linhas de código já é possível realizar a integração da *game engine* a um servidor *web* e começar a renderizar textos e imagens.

Por fim, a ImpactJS se mostrou ser uma engine bem capaz, tendo diversos recursos úteis, incluindo um editor de níveis bem desenvolvido, o "Weltmeister". Apesar disso, a engine se mostrou bastante dependente deste editor, e a criação de um nível do jogo, mesmo simples, acabou se tornando bem mais complicada que nas demais engines, ao não se utilizar deste editor (já que os labirintos do Mazogs são gerados aleatoriamente). Além disso, os diferentes modos de

[2] A tradução literal de *level* seria nível, mas uma definição mais apropriada para o contexto do presente trabalho, seria todo o ambiente disponível para o jogador interagir, onde é construído o espaço em que o usuário irá jogar.

lidar com colisões da ImpactJS se mostraram confusos ao lidar com as paredes geradas no nosso jogo básico, gerando problemas como arrastar as paredes, o que não era a intenção.

A tabela abaixo mostra uma comparação resumida das três *game engines*:

Tabela 4.1: Tabela mostrando uma comparação entre as três *game engines* finais.

	Prós	Contras
MelonJS	Editor de níveis com várias funções	Dependência no editor de níveis, documentação
ImpactJS	Editor de níveis completo, facilidade em iniciar jogos	Dependência no editor de níveis, modos de física confusos
Phaser3	Simplicidade, recursos, documentação/comunidade	Física padrão pode ser um pouco restrita

Como o Mazogs é um jogo simples, e não exige lógicas complexas ou muito poder gráfico, a *game engine* escolhida para a implementação do jogo foi a Phaser3. Os critérios decisivos para esta escolha foram a quantidade de material disponível para o aprendizado e a facilidade de implementação, onde toda lógica complexa ocorre internamente na biblioteca, e o desenvolvimento do jogo acaba sendo mais simples.

4.2 FUNCIONAMENTO DA ENGINE

Como foi visto, a engine Phaser3 foi escolhida para o desenvolvimento pela sua facilidade e funcionalidades. Por ser uma biblioteca JavaScript, para ser utilizada, a engine é incluída no código principal do programa, que roda numa página *web* comum (num arquivo HTML). Esse código JavaScript é todo carregado no navegador do jogador quando ele carrega a página do jogo, incluindo todo o código da Phaser3 em si.

Inicialmente, o teste das engines que fizemos baseou-se na implementação do tutorial *Getting Started* da Phaser3 [3]. Esse é um tutorial básico que ensina os conceitos importantes da Phaser3, e ajuda na configuração inicial para rodar o jogo, que exige que a página HTML que contem o código seja "servida"(por questões de segurança, os navegadores não permitem carregar arquivos locais numa página diretamente, sendo necessário assim um servidor).

Ao terminar este tutorial, o jogo consistia de um protótipo estilo "plataforma". Esse estilo de jogo possui visão lateral dos níveis, movimentação em grande parte lateral e gravidade (para baixo, geralmente). Apesar de jogos de "plataforma"serem populares, o Mazogs original não era assim, e foi decidido ser fiel ao jogo. Para isso, foi necessário modificar a movimentação do jogador para ser nas 4 direções: cima, baixo, direita, e esquerda (removendo o pulo), e a visão do jogo para ser "por cima"(*top-down*), o que envolveu uma mudança dos gráficos, mais que qualquer coisa.

[3]Getting Started with Phaser 3. Phaser, 2022. Disponível em: https://phaser.io/tutorials/getting-started-phaser3. Acesso em: 4 de maio de 2022.

Tabela 4.2: Tabela mostrando as funções padrão da Phaser para a *scene*.

Função da scene	*Descrição/Funcionamento*
init	Roda quaisquer funções necessárias na inicialização da *scene*. Essa função muitas vezes é utilizada para reiniciar o valor de variáveis gerais e deixar a *scene* em um estado inicial apropriado.
preload	Função destinada a fazer o carregamento de todos os recursos necessários para nível atual. Isso envolve principalmente gráficos, mas também outros recursos que venham a ser utilizados diretamente pela Phaser, no nível atual.
create	Cria os objetos e a lógica da *scene*. É utilizada para criar as animações utilizadas, iniciar a posição e variáveis da câmera e outros objetos do jogo, e pode ser usada até mesmo para lidar com a entrada do jogador.
update	Roda a cada "passo"do *loop* jogo, após o processamento da física interna do Phaser3, então pode ser utilizada para lógicas de atualização de informações de objetos, após eles se moverem.

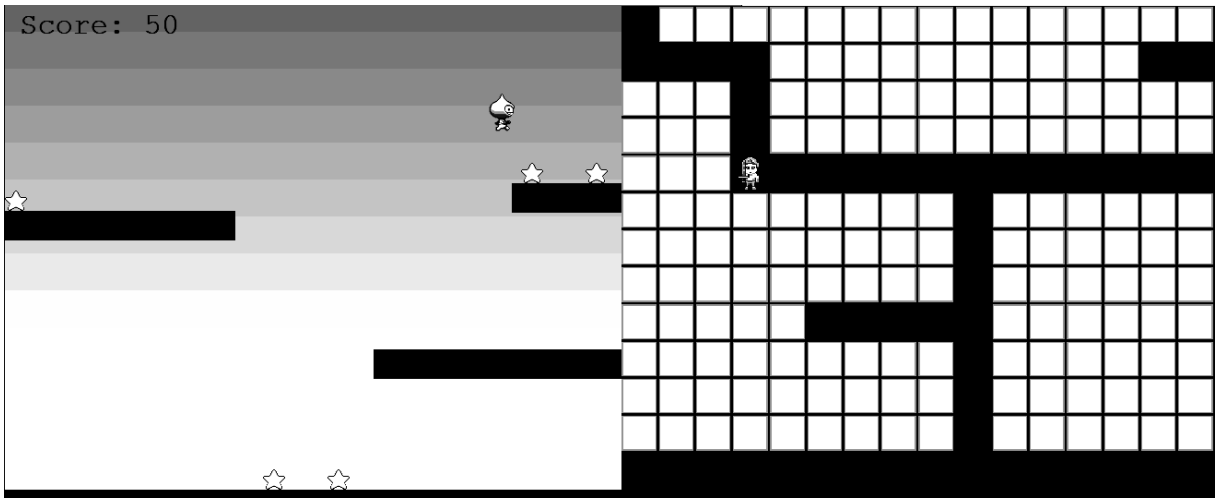


Figura 4.1: À esquerda, protótipo de jogo implementado durante o tutorial da engine Phaser 3, mostrando o personagem pulando. À direita, versão modificada para o nosso jogo.

A *engine* Phaser 3 possibilita realizar configurações rápidas gerais para o jogo, como as dimensões da janela em que o jogo será renderizado, o tipo de física que será utilizado e qual o tipo de renderização dos gráficos. No caso deste trabalho, foi utilizado a física do estilo *arcade*, o padrão, renderização *pixel-art* e o tamanho da janela escolhido foi pequeno, para ficar semelhante ao jogo Mazogs original. Todas essas configurações foram escolhidas por se tratar de um jogo de duas dimensões (2D) e relativamente simples, o que acabou facilitando no desenvolvimento posterior do jogo.

Após ser criada a instância do jogo, é necessário criar uma *scene*[4]. Esta *scene* é a classe principal do jogo, onde será feita a leitura de texturas, inicialização de entidades, e também a implementação de toda a lógica de jogabilidade.

No caso, a *scene* é uma instância criada da classe *Phaser3.Scene*, disponibilizada pela biblioteca, que possui as seguintes funções cruciais a qualquer aplicação utilizando Phaser3: *init*, *preload*, *create* e *update*. São nestas funções em que a lógica de cada parte do jogo é criada.

[4]No contexto de jogos, *scene* é a cena do estado atual em que o jogador se encontra. Um jogo pode ser composto por diversas cenas, cada uma representando um ambiente diferente. Um exemplo prático seria um jogo onde temos uma cena com um mapa de fogo, e após avançar para o próximo nível, há uma cena diferente com um mapa de gelo.

4.3 RENDERIZAÇÃO

Um dos recursos mais importantes oferecidos pela *Game Engine* é a renderização. No caso da Phaser3, ela oferece recursos de renderização gráfica que acabam tornando todo o processo muito mais simples, deixando a lógica complexa para a biblioteca em si, e não para quem está desenvolvendo o jogo.

A renderização é responsável por desenhar na página HTML tudo aquilo que será mostrado na tela do usuário, como textos, o labirinto, o personagem, os monstros e qualquer outra entidade (espada, baú de tesouro, etc).

Para as imagens, é possível utilizar imagens estáticas ou *sprite sheets*, que consistem em um único arquivo composto por diversas imagens. Estes são utilizados para imagens animadas quadro a quadro, ou então entidades que podem ter vários estados diferentes como, por exemplo, o jogador empunhando a espada, ou então o baú de tesouro, que pode estar no mapa, ou ser pego pelo jogador. Quanto às imagens estáticas, elas são renderizadas do jeito que foram carregadas e são usadas, no jogo, nas paredes do labirinto.

Como introduzido na seção anterior, as imagens são carregadas pela *scene* criada, especificamente na função *preload*. Para carregar uma imagem, é necessário especificar o diretório em que ela está localizada e também atribuir à esta imagem um identificador único. Este identificador será utilizado posteriormente em funções de renderização.

Para a renderização dessas imagens, deve-se instanciar um *sprite*[5] na *scene*, referenciando o nome dado à imagem carregada durante a inicialização da *scene*. Uma vez que o *sprite* for criado, ele permanecerá na *scene* até que em algum momento seja executado algo para remove-lo (como, por exemplo, uma colisão com o jogador).

Para a renderização de textos, o processo acaba sendo mais simples. Como não é necessário carregar nada, o texto só precisa ser adicionado na *scene* com as configurações desejadas. Nas configurações é possível definir a posição do texto, o tamanho da fonte, a cor e o texto em si.

```
this.add.text(
   20,
   106,
   "Aperte ENTER para continuar...",
   {
      fontSize: 12,
      color: "#fff",
      fontStyle: "bold"
   }
);
```

[5]Na Phaser3, *sprite* é um objeto gráfico bidimensional usado pelo motor gráfico para ser renderizado. Este objeto gráfico é vinculado a uma imagem e possuí parâmetros como posição e dimensão.

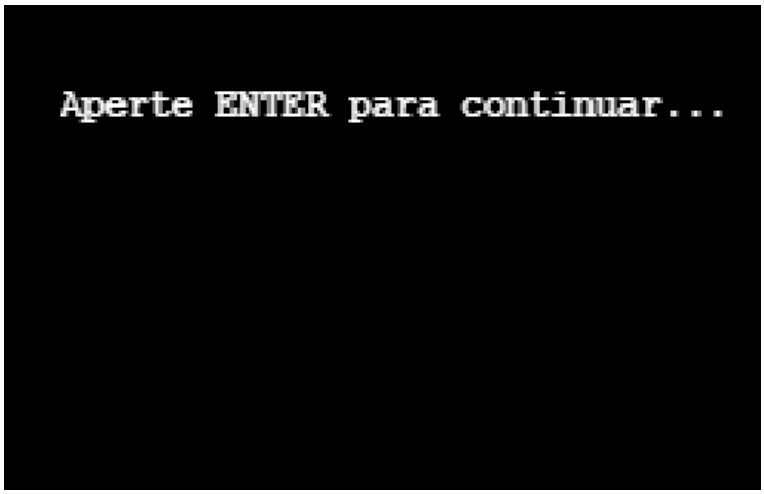


Figura 4.2: Exemplo de texto sendo renderizado.

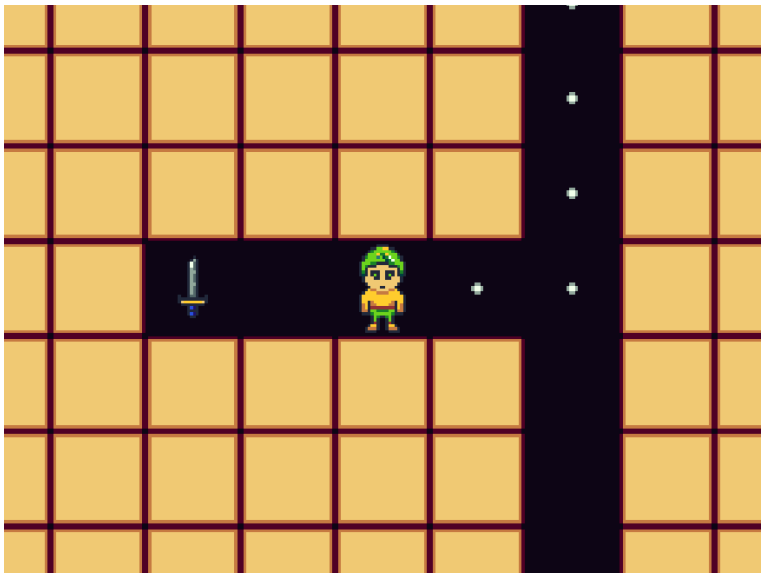


Figura 4.3: Exemplo de imagens sendo renderizadas.

No caso da renderização do jogador, como ele pode ter mais de um estado, é necessário utilizar uma imagem do tipo *sprite sheet*. Para o desenvolvimento deste jogo, foi utilizado três possíveis estados, sendo eles:

1. Jogador empunhando a espada
2. Jogador carregando o baú de tesouro
3. Jogador sem nada nas mãos

Diferente das imagens estáticas, quando utilizado *sprite sheet*, é necessário também definir as animações que podem ser encontradas dentro daquele arquivo. Como para este projeto as imagens não são animadas quadro a quadro (como, por exemplo um personagem correndo), mas sim, a mesma entidade estática em diferentes estados, a definição destas animações é mais simples. No caso, só é necessário definir um identificador ao respectivo estado, de qual arquivo de imagem ele se refere, e a posição dele na imagem, que seria o quadro (*frame*) daquele estado.

```
this.anims.create({
   key: 'normal',
   frames: [ { key: 'player2', frame: 0 } ],
});
this.anims.create({
   key: 'withSword',
   frames: [ { key: 'player2', frame: 1 } ],
});
this.anims.create({
   key: 'withChest',
   frames: [ { key: 'player2', frame: 2 } ],
});
```

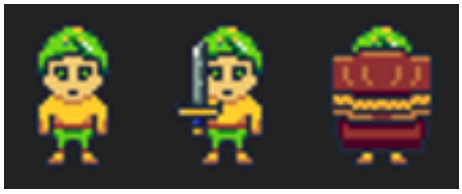


Figura 4.4: Exemplo do *sprite sheet* do personagem.

4.4 JOGABILIDADE

Tendo a parte de renderização funcional, é possível começar a trabalhar na lógica do jogo em si. É nesta etapa que será implementado a movimentação do personagem, a criação do labirinto e o controle das ações que o usuário pode realizar.

4.4.1 MOVIMENTAÇÃO DO PERSONAGEM

O estilo de jogo do Mazogs é conhecido como *top-down*. Este estilo de jogo é uma referência ao tipo de visão do jogo, como se houvesse uma câmera posicionada de cima para baixo, onde o usuário tem uma visão de cima do cenário. Este estilo de jogo é também muito comum em RPG (*Role-playing game*) de duas dimensões (2D).

Para saber quando o usuário está realizando um comando para mover o personagem, é necessário ter o controle de quando alguma tecla no teclado está sendo pressionada. Isto é feito definindo uma função *handler* ao criar a *scene* do jogo, para que sempre que uma tecla for pressionada, uma função específica seja chamada. Esta função é onde serão tratadas as teclas pressionadas e onde será checado se o personagem deve ser deslocar na horizontal ou na vertical.

No caso do Mazogs, o usuário pode se movimentar para cima, para baixo, para esquerda ou para direita, utilizando teclas do teclado, e estes são os únicos comandos que o usuário pode realizar com o personagem.

Quando o usuário pressiona alguma tecla responsável pela movimentação, o jogo detecta esse evento do teclado e assim consegue trata-lo, e realizar uma ação específica. No caso da movimentação, a ação que o jogo realizará será utilizar as funções de física da Phaser3, que vão influenciar diretamente na posição final da coordenada **x** ou **y** do jogador.

Inicialmente, a movimentação do personagem foi implementada utilizando os recursos de física que a biblioteca dispõe. Antes de qualquer coisa, foi necessário desabilitar a gravidade, que vem habilitada por padrão, para que o personagem permanecesse livre e atendesse ao estilo de jogo *top-down*. Para deslocar o personagem à alguma direção, foi utilizado o conceito de *velocity* (velocidade) na física do personagem. Quando atribuído um valor para a *velocity* do

personagem, isso indica ao jogo que ele deve se deslocar à direção indicada na unidade de velocidade especificada. Enquanto a variável *velocity* estiver diferente de zero, o personagem continua em movimento. No caso do presente jogo, isto foi realizado atribuindo uma *velocity* no eixo **x** para se deslocar na horizontal, e no eixo **y** para se deslocar na vertical. Para a direção do movimento, é indicado pelo valor sendo positivo ou negativo.

No exemplo abaixo, é possível observar o conceito de utilizar *velocity* para deslocar o personagem para baixo. Como a velocidade no eixo **x** é zero e no eixo **y** o valor é negativo, a movimentação do personagem tende a ser direcionada para baixo.

```
1 player.setVelocityX(0);
2 player.setVelocityY(-160);
```

Um dos problemas enfrentados utilizando *velocity*, é que não foi possível replicar exatamente o estilo de jogo do Mazogs. Além disso, o conceito de *velocity* tem uma semântica diferente pra *game engine* e outra para o que se precisava. Quando definido uma *velocity* no personagem, não há o controle de quantas unidades o personagem se deslocará. Ou seja, enquanto o valor estiver diferente de zero, o personagem estará em movimento e não há uma maneira de definir um deslocamento fixo. Além da movimentação ter ficado bem diferente da original do Mazogs, também acabou ocasionando problemas na jogabilidade. Como não havia controle de quantas unidades exatas o personagem se deslocou, ele acabava ficando preso em alguns pontos do mapa e não era possível acessar os corredores do labirinto. Com isto, atributos implícitos da Phaser3 precisaram ser explicitados em uma implementação própria, de modo que se tivesse o controle total da movimentação do personagem e conseguisse atender as expectativas para o Mazogs.

Como alternativa, foi decidido não utilizar a física oferecida nativamente pela *game engine* para o deslocamento do personagem, mas sim, atribuir valores fixos às coordenadas do eixo **x** e **y** do personagem. Quando o usuário pressionar a tecla da seta da direita **k** vezes, sabe-se que ao final da ação o personagem estará **k** unidades vezes à direita, não havendo mais a possibilidade dele se deslocar para uma posição inesperada. Assim, o problema de não conseguir entrar nos corredores do labirinto foi solucionado, e a movimentação ficou exatamente igual ao do jogo original Mazogs.

```
1 player.getObject().x -= 32;
2 player.getObject().body.x -= 32;
```

No exemplo acima, o código está alterando o valor da coordenada **x** do personagem, diminuindo em 32 unidades, o que faz o personagem se deslocar para a esquerda.

O único contra desta alternativa é que perde-se a física original do jogo, e a colisão entre as entidades deve ser gerenciada por quem está desenvolvendo o jogo, não havendo mais a detecção automática de colisão pela própria *game engine*. Assim, se o jogador tentar se deslocar para a esquerda mas houver um obstáculo, quem deve permitir ou não este movimento é a lógica do jogo que está sendo desenvolvido, e não mais a *game engine*, que já não é mais a responsável pelas colisões.

4.4.2 COLISÃO ENTRE ENTIDADES

Como mencionado anteriormente, ao mudar a lógica de movimentação do personagem, a colisão entre as entidades é perdida. Isto ocorre por não estar mais utilizando a física oferecida pela *game engine*, transferindo a responsabilidade de gerenciar a colisão para o desenvolvedor.

No caso do Mazogs, a colisão existente é muito simples e a única entidade com a qual o personagem pode colidir (e impedi-lo de se movimentar), são as paredes do labirinto.

Para resolver isto, foi adicionada uma lógica durante o processo de receber o evento de tecla pressionada, onde é decidida para qual direção o personagem irá se movimentar. O fluxo adicionado nesta parte do jogo foi:

1. Salvar a posição atual do personagem
2. Testar a posição desejada para verificar uma possível colisão
3. Phaser3 deve indicar se a posição desejada irá colidir ou não
4. Se a *game engine* indicar que a posição desejada irá colidir, o personagem se mantém na posição atual, caso contrário, ele movimenta o jogador para a posição desejada.

Desta forma, se o jogador colidir com alguma parede, a posição dele permanece inalterada e ele não consegue se movimentar. Para o jogo do trabalho em questão, só foi utilizado para colisão entre as paredes do labirinto e a entidade do personagem, mas poderia ser facilmente estendido a outras categorias de entidades.

Toda esta lógica extra mencionada pode ser encontrada no exemplo abaixo. Neste caso, o código irá percorrer todas as paredes do labirinto, e caso encontre alguma em que o personagem está colidindo, ele irá salvar o objeto da parede na variável **collider**. Ao final do código, se a variável não for nula, isso significa que houve colisão e o personagem deve voltar à posição original.

```
function handlePlayerMovementInput(oldPosition) {
   var collider = null;

   platforms.children.entries.every(function(platform, index){
      if (scene.physics.overlap(player.getObject(), platform)) {
         collider = platform;
         return false;
      }

      return true;
   });

   // rollback position
   if (collider) {
      player.getObject().x = oldPosition[0];...
```

4.4.3 GERAÇÃO DO LABIRINTO

No jogo Mazogs original, o labirinto do jogo é gerado aleatoriamente em tempo real, tendo sempre um desafio novo para o jogador. Apesar disso ser muito interessante, a geração de labirintos é um assunto complexo, que poderia gerar um trabalho inteiro dedicado a isto. Como o foco do presente trabalho é criar um jogo simples em JavaScript para testar a *game engine* e adicionar o modo multijogador ao final, o labirinto que será carregado é lido a partir de um arquivo auxiliar.

Este arquivo onde será feito a leitura do labirinto, é um arquivo de texto, onde caracteres específicos definem onde haverá paredes, e também, o local onde haverá qualquer outro tipo de entidade possível, como espadas ou o baú de tesouro.

- . : Nenhuma entidade, espaço vazio

- # : Parede do labirinto
- C : Baú de tesouro
- S : Espada
- P : Portal
- M : Mazogs

Como não há geração dinâmica do labirinto, o jogo pode conter diversos arquivos que representam os mapas no jogo, e ao iniciar a partida, estes arquivos podem ser sortidos e escolhidos aleatoriamente, para haver mais possibilidades de labirintos e o jogo ficar mais dinâmico. Na figura 4.5 é possível encontrar a representação de um mapa no arquivo texto, e na figura 4.6 o resultado funcionando no jogo.

```
..C.P.S
#######
```

Figura 4.5: Arquivo utilizado para representar o mapa do jogo.

Figura 4.6: Labirinto gerado a partir do arquivo texto.

Cada uma das entidades que podem ser representas no mapa, tem um tamanho fixo de 32 pixeis de largura por 32 pixeis de altura, contribuindo também para o sistema de detecção de colisão. Isto ocorre porque todas as entidades têm a mesma dimensão, então não existem objetos que colidem parcialmente entre si.

A criação das paredes do labirinto é realizada por uma função que recebe o conteúdo do arquivo do mapa como entrada, percorre todos caracteres existentes, e quando encontrado o caractere responsável pela parede, o objeto da parede é instanciado. No caso, essa criação é realizada em um grupo de corpos estáticos na física da Phaser3. Corpos estáticos não sofrem influência de nenhuma variável da física do jogo, como gravidade ou então velocidade, por isso acaba sendo o ideal para estas paredes. O fluxo implementado para a criação do mapa utilizando a Phaser3 foi o seguinte:

1. O jogo irá percorrer todos os caracteres existentes no arquivo que representa o mapa, deslocando 32 pixeis na coordenada **x** a cada caractere lido
2. Caso encontre um caractere simbolizado pela parede, o caractere **#**, o jogo deve adicionar uma parede na posição **x** e **y** respectiva
3. Caso encontre um caractere diferente do que simboliza uma parede, o caractere **#**, o jogo deve criar a entidade respectiva
4. Caso encontre uma quebra de linha no arquivo, o jogo deve entender que a posição na coordenada **y** deve ser deslocada em 32 pixeis e a coordenada **x** deve ser restaurada

4.4.4 CRIAÇÃO DE ENTIDADES

Todos objetos existentes no jogo, exceto as paredes do labirinto, são definidos como uma entidade. Ou seja, uma espada, um baú de tesouro, o rastro do personagem, ou até o próprio personagem, são entidades. Para isso, há uma classe em JavaScript nomeada como **Entity**, em que todos os objetos citados anteriormente criam uma instância desta classe. Desta maneira, é possível ter uma melhor organização das informações que serão úteis por todas as entidades criadas, como, por exemplo, as coordenadas **x** e **y** usadas para o sistema de colisão.

Para a instanciação dessas entidades, há uma função específica para isso, que deve receber algumas informações importantes como a coordenada **x** e **y**, um identificador único para a entidade e o tipo da entidade.

Cada uma dessas entidades pode ter um tratamento diferente na hora de sua criação. Especificamente no jogo deste trabalho, quando uma entidade do tipo **Trail** é criada, há uma lógica adicional para que essa entidade se auto destrua após um intervalo de tempo. Também, para a entidade principal do jogador, temos uma lógica adicional para gerenciar o estado do mesmo (para identificar se ele está com uma espada em mãos ou o baú de tesouro).

4.5 MODO MULTIJOGADOR

O maior desafio do presente trabalho foi implementar o modo multijogador no jogo final. Como praticamente nenhuma das *game engines* em JavaScript consideradas oferece recursos de rede (incluindo a Phaser3), esta foi uma parte do trabalho que exigiu uma maior pesquisa para o uso de outras ferramentas.

O objetivo em implementar o modo multijogador, foi validar se era possível bibliotecas diferentes trabalharem em conjunto com a *game engine* escolhida, Phaser3, e identificar qual seria o nível de dificuldade nesta integração. Também, com o modo multijogador deseja-se que o modo jogo passe a ser cooperativo, onde todos os usuários da mesma sessão podem jogar juntos com o mesmo objetivo, matar as Mazogs e entregar o baú de tesouro na saída do labirinto, para que assim, todos vençam a partida.

Após considerar algumas alternativas para a comunicação entre os jogadores (e um possível servidor), a solução escolhida acabou sendo a biblioteca Socket.IO. Esta biblioteca permite comunicação de baixa latência (o que é importante para um jogo em tempo real, como o Mazogs) entre clientes e servidores. A biblioteca é baseada em eventos, e oferece facilidades como reconexão automática.

Para implementar o modo multijogador, foi necessário criar um servidor. Este servidor, funciona junto do servidor que serve a página na qual os jogadores acessam o cliente no navegador *web* (a parte gráfica do jogo que foi explicada anteriormente). O servidor é o responsável por gerenciar as conexões, coordenar os jogadores e entidades e trocar as informações necessárias entre os usuários para a sincronização do jogo.

O servidor não só conecta um jogador ao outro, mas acaba rodando boa parte da lógica do jogo. Como agora ambos os jogadores podem interagir com outras entidades, como espadas, baús e as Mazogs no labirinto, o servidor é quem deve controlar essa interação. Como o servidor mantém todas as entidades sincronizadas, isso também é útil para evitar trapaças, já que o usuário não consegue manipular nenhuma informação falsa no servidor.

4.5.1 TROCA DE MENSAGENS

O principal recurso visando ao se implementar um servidor é a troca de mensagens entre os clientes, e com próprio servidor. Essa troca de mensagens é muito crucial para o

funcionamento do jogo, visto que no modo multijogador tudo deve estar sincronizado e para isso deve existir uma comunicação entre cliente e servidor.

O cliente consegue enviar mensagens para o servidor, e o servidor pode enviar mensagens a todos os usuários, ou enviar mensagens para um usuário específico. Essas mensagens nada mais são do que textos transmitidos de um lado para o outro. Para uma melhor legibilidade dessas mensagens, é muito comum formatar o texto em JSON (JavaScript Object Notation).

Para enviar uma mensagem através da rede, a biblioteca Socket.IO disponibiliza funções primitivas de envio, sendo uma do tipo *multicast*[6] e outra do tipo *broadcast*[7]. Esta função de envio recebe um identificador único da mensagem que está sendo enviada e o conteúdo da mensagem em si.

```
1 io.emit(NetworkMessage.ENTITIES_DATA, Array.from(entities));
```

No caso de receber e tratar uma mensagem, a biblioteca Socket.IO disponibiliza outra função específica para tal. Com esta função, é possível vincular uma função de tratamento para quando receber uma mensagem com um identificador específico.

```
1 socket.on(NetworkMessage.ENTITIES_DATA, handleEntitiesDataSync);
```

Tabela 4.3: Mensagens definidas para a comunicação entre servidor e cliente.

Mensagem	**Descrição**	**Tipo**
CHARACTER_DATA	Informações do personagem	Cliente ->Servidor
ENTITIES_DATA	Informações de todas entidades	Servidor ->Cliente
MAP_DATA	Mapa do labirinto em arquivo texto	Servidor ->Cliente
WIN_GAME	Indica uma vitória aos jogadores	Servidor ->Cliente
USE_ITEM	Colisão entre jogador e entidade	Servidor ->Cliente
DEAD	Indica que o jogador morreu	Servidor ->Cliente

4.5.2 MAPA DO LABIRINTO

Agora que vários jogadores podem participar de uma mesma sessão do jogo, o labirinto não pode mais ser criado pelo cliente. Como todos os jogadores devem ter o mesmo mapa para conseguirem jogar simultaneamente, o servidor passa a ter a responsabilidade de carregar o labirinto e enviar a todos os usuários.

A origem do mapa continua sendo de um arquivo texto, porém agora, o servidor que efetua essa leitura. Assim que um usuário é conectado ao servidor, o servidor realiza o envio deste arquivo texto do mapa para o cliente. Quando o cliente recebe a mensagem com o mapa do labirinto, o fluxo a ser seguido é o mesmo mencionado na seção 4.5, com a única diferença que agora o dado vem do servidor, e não mais direto do cliente.

4.5.3 SINCRONIZAÇÃO DE ENTIDADES

Com a implementação do modo multijogador, é desejado que todos os jogadores consigam ter em sua tela as mesmas informações espelhadas entre todos, e que seja possível a interação entre eles. Logo, o servidor deve enviar as informações de todas as entidades existentes no servidor para todos os usuários ativos.

[6]Envio da mensagem é direcionado a clientes específicos.

[7]Envio da mensagem é realizado a todos os clientes ativos.

Para isso acontecer, o servidor, assim como o cliente, deve manter uma lista de entidades existentes, com suas respectivas informações. Esta lista de entidades é utilizada para o servidor saber quem está ativo no jogo, e seu uso pode ser estendido como, por exemplo, ao sistema de colisões, que será detalhado posteriormente.

Esta lista de entidades é atualizada sempre que um jogador se conecta ou se desconecta do jogo, ou então, quando o servidor identifica que se deve criar ou remover uma entidade do jogo (como, por exemplo, um baú de tesouro).

O servidor realiza o envio desta lista de entidades a cada 16ms para todos os usuários ativos. Este intervalo de 16ms, foi escolhido a partir do número de quadros por segundo que gostaríamos que fossem sincronizados. No caso, escolhemos um intervalo de 60 FPS (quadros por segundo). Como FPS significa quadros por segundo, precisamos fazer 1000 divido por 60, que seria 1 segundo representado em milissegundos (1000), dividido por 60 que é o número de quadros que escolhemos. Com isso, chegamos no valor de 16ms. Vale destacar que este intervalo de sincronização está diretamente relacionado com a responsividade entre servidor e cliente. Utilizando um número maior de quadros por segundo, a responsividade é melhor, porém o número de mensagens trocadas entre cliente e servidor também é maior, podendo causar um sobrecarregamento na rede. Também, o intervalo escolhido não pode ser muito baixo, o que causaria uma responsividade baixa.

Desta maneira, é possível garantir que todos os usuários vão ter a mesma as mesmas entidades e com as mesmas informações sincronizadas. Como o papel do servidor na sincronização é simplesmente enviar a lista de entidades para todos os usuários, não existe nenhuma lógica adicional além de manter a lista com as informações atualizadas.

Quando o cliente recebe essa lista de entidades do servidor, ele deve ser responsável pelas seguintes tarefas:

- Identificar entidades que deixaram de existir, para não renderizar mais;
- Identificar novas entidades no cliente, para adicioná-las à renderização do jogo;
- Atualizar as informações das entidades, como a posição ou o identificador único.

Vale destacar que para entidades que não são do tipo **Jogador**, as informações atualizadas tem origem do lado do servidor, e são somente enviadas para os clientes. Porém, para entidades do tipo **Jogador**, as informações atualizadas tem origem do lado do cliente, e antes, devem ser enviadas e atualizadas no servidor. Isto ocorre pois o jogador depende de ações oriundas do cliente, como, por exemplo, detectar que uma tecla foi pressionada no navegador *web* e assim realizar a movimentação do jogador, o que é estritamente vinculado ao lado do cliente.

Por isso, para a sincronização destes jogadores há um fluxo adicional pela parte do cliente. Toda vez que algum jogador tiver seu estado modificado, como, por exemplo, uma nova posição no labirinto, o cliente deve enviar estas informações atualizadas ao servidor. Assim que o servidor recebe estas informações, ele simplesmente atualiza a lista de entidades antes mencionada, e o resto do fluxo de envio e atualização do cliente segue normal.

No diagrama abaixo, é possível visualizar o fluxo que ocorre entre o cliente e o servidor para a sincronização de entidades. No caso, o fluxo mostra como um jogador é sincronizado para o servidor, e posteriormente é replicado para todos os clientes ativos.

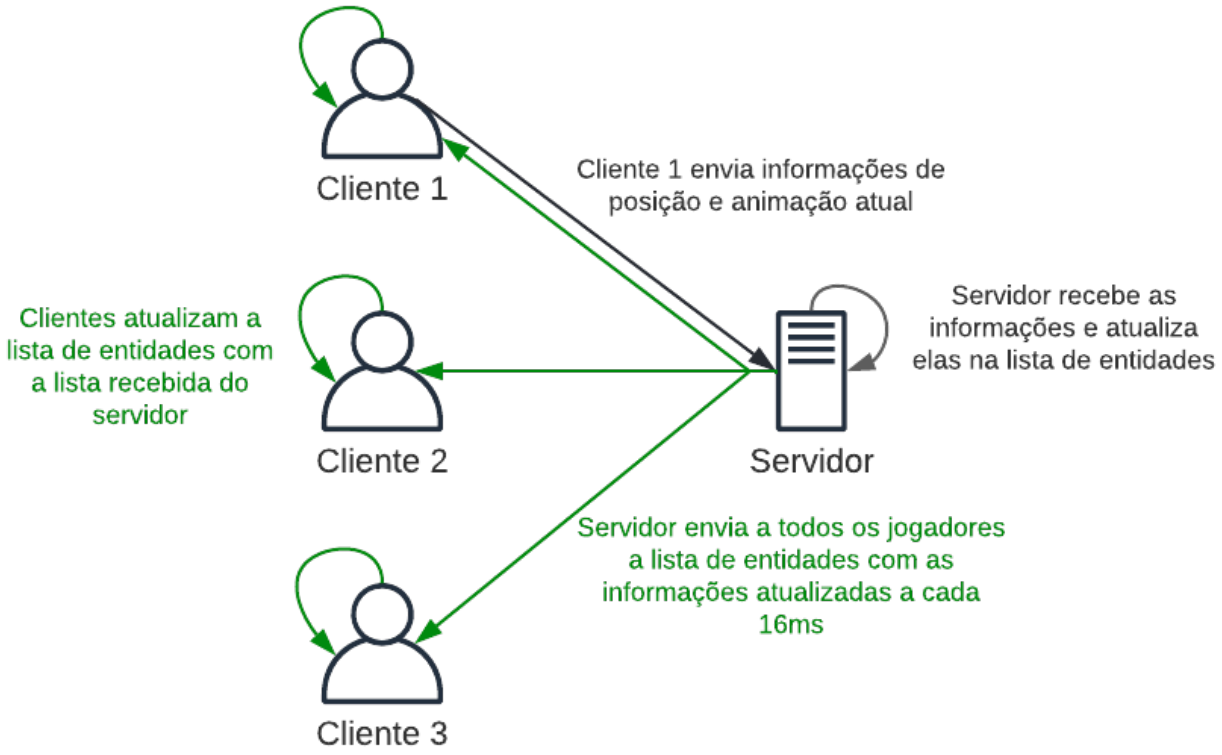


Figura 4.7: Fluxo entre cliente e servidor para a sincronização de entidades.

4.5.4 COLISÃO ENTRE ENTIDADES

Agora que múltiplos jogadores participam de uma mesma sessão, o servidor precisa saber qual jogador colidiu com determinada entidade, e assim, tomar uma ação direcionada a este jogador. Por exemplo: se algum jogador pegar uma espada, a espada não deve ficar mais disponível para os outros jogadores, e somente o jogador que pegou deverá empunhá-la. Logo, o servidor passa a ser o responsável por esta lógica de verificar as colisões.

Como o servidor mantém uma lista de entidades do jogo e suas respectivas posições, detectar a colisão entre as entidades passa a ser uma tarefa relativamente simples. Para cada jogador existente, o servidor irá testar se está colidindo com todas as outras entidades. Este teste de colisão é realizado simplesmente verificando se a posição de quem está testando é igual à posição da entidade a ser testada. Caso a colisão seja detectada, para cada categoria de entidade o servidor pode tomar uma ação específica. Por exemplo, caso o jogador colida com um baú de tesouro, a ação que o servidor realizará será: remover o tesouro da lista de entidades (para ninguém mais conseguir usar) e enviar uma mensagem ao cliente do jogador que colidiu para notificá-lo. Este fluxo descrito pode ser visualizado na figura abaixo.

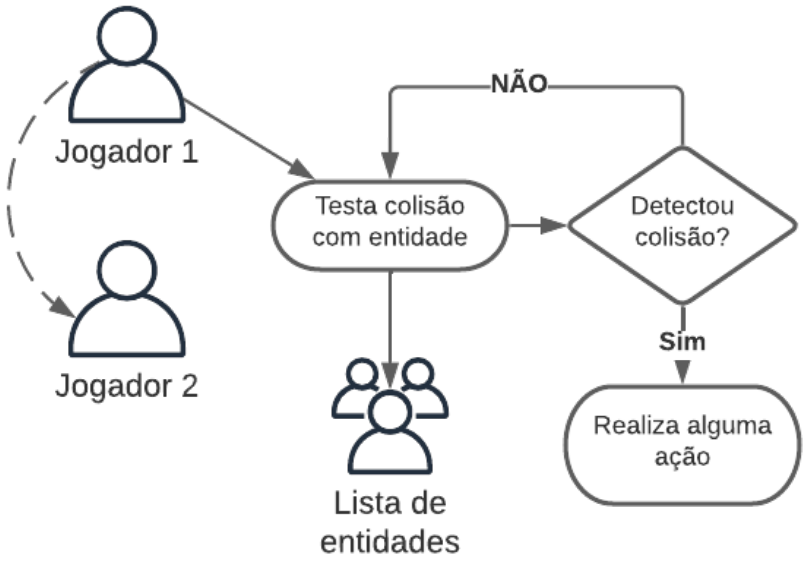


Figura 4.8: Fluxograma para detectar colisão no servidor.

A função responsável por testar as colisões é chamada a cada **16ms** (milissegundos), o mesmo tempo utilizado para o servidor realizar a sincronização das entidades. Este intervalo tempo escolhido é suficiente para manter a responsividade entre o servidor e o cliente.

5 CONCLUSÃO

Neste trabalho, foram comparadas as principais *game engines* em JavaScript com código aberto e focadas em jogos simples de duas dimensões. Das *engines* pesquisadas, foi possível observar que a Phaser3 não só possibilita grande facilidade para iniciar o desenvolvimento de um jogo, mas possui grande comunidade e documentação. Por ter se saído melhor que as *engines* Melon.Js e Impact JS nos testes deste trabalho, a *engine* Phaser3 foi escolhida para implementação da nossa versão do jogo Mazogs.

Ao modificar o exemplo de jogo implementado no tutorial "Getting Started"da engine, a Phaser3 se mostrou de fato uma ferramenta que auxilia muito o desenvolvimento de jogos 2D em JavaScript. Mesmo assim, foram necessárias modificações para evitar o uso da engine padrão de física do Phaser3 para a movimentação do personagem, o que indica que mesmo em jogos simples, pode ser necessário código extra para realizar tarefas simples.

Por fim, a implementação do modo multijogador utilizando a biblioteca Socket.IO mostrou que a mesma é uma alternativa muito eficiente para adição de um modo multiusuário a uma aplicação já existente. Não houve grandes dificuldades para implementação do modo multijogador, sendo a maior parte do trabalho reorganizar o jogo para que parte do processamento fosse feito no servidor.

Sendo assim, recomendamos a utilização das bibliotecas Phaser3, para desenvolvimento de jogos 2D em JavaScript, e da Socket.IO, para aplicações multiusuário de baixa latência, no geral.

5.1 TRABALHOS FUTUROS

Para trabalhos futuros, seria interessante pesquisar mais a fundo a parte de comunicação online. A biblioteca Socket.IO é muito interessante para projetos que precisem de comunicação de baixa latência entre clientes e servidor, o que a torna uma grande candidata para desenvolvimento de jogos ou aplicações com muitos usuários simultâneos (o que acabou não sendo o foco deste trabalho).

Sugere-se então que seja estudada a biblioteca Socket.IO mais a fundo, testando seu uso em aplicações com um número massivo de usuários, possivelmente com testes de estresse, para ver até quantos usuários podem ser suportados por uma API como a desenvolvida neste trabalho, sem perdas. Isso pode ser muito interessante para se estudar os problemas que podem surgir nessa categoria de aplicação, e quais seriam as possíveis soluções. A aplicação em si não precisa ser feita com JavaScript, já que a biblioteca é suportada por um grande número de linguagens de programação.

REFERÊNCIAS

AMORIN, A. (2006). A origem dos jogos eletrônicos.

ASSIS, G., Ficheman, I., Corrêa, A. G., Netto, M. e Lopes, R. (2006). Educatrans: um jogo educativo para o aprendizado do trânsito.

BANASCHAK, P. (1999). *Early East Asian Chess Pieces: An overview*.

CRAWFORD, C. (2003). *Chris Crawford on Game Design*. New Riders Games.

ESTEVES, D. M. P. G. (2021). A utilização do software unreal engine na pré-visualização de cenas de cinema e televisão.

FLEURY, A., Nakano, D. N. e Cordeiro, J. H. D. (2014). *Mapeamento da indústria brasileira e global de jogos digitais*. GEDIGames: NPGT.

FREIRE, G. G. e GUERRINI, D. (2017). Jogos tradicionais: As aulas de educação física e um site como espaços de preservação cultural.

JUUL, J. (2009). *A Casual Revolution: Reinventing Video Games and Their Players*. The MIT Press.

LEITE, L. (2003). Introdução à história dos jogos eletrônicos.

LEWIS, M. e JACOBSON, J. (2002). *Game Engines in Scientific Research*. Communications of the ACM.

SANTOS, D. C. d. (2015). Estudo comparativo de motores gráficos unreal e unity3d.

SERRA, M. C. (1999). *Os jogos tradicionais em Portugal. As relações entre as práticas lúdicas e as ocupações agrícolas e pastoris*. Tese de doutorado, Universidade de Trás-os-Montes e Alto Douro.

XEXÉO, G., Carmo, A., ACIOLI, A., TAUCEI, B., Dipolitto, C., Mangeli, E., KRITZ, J., COSTA, L. F., Arêas, M., Monclar, R., GARROT, R., Classe, T. e AZEVEDO, V. (2017). O que são jogos: Uma introdução ao objeto de estudo do ludes.