Universidade de Brasília – UnB
Faculdade de Ciências e Tecnologias em Engenharia - FCTE
Engenharia de Software

Desenvolvimento de Jogos Digitais na Prática: Um Caso Aplicado ao estilo Tron

Autor: Victor Yukio Cavalcanti Miki
Orientador: Prof. Dr. Ricardo Matos Chaim

Brasília, DF
2025

Victor Yukio Cavalcanti Miki

Desenvolvimento de Jogos Digitais na Prática: Um Caso Aplicado ao estilo Tron

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade de Ciências e Tecnologias em Engenharia - FCTE

Orientador: Prof. Dr. Ricardo Matos Chaim

Brasília, DF

2025

Victor Yukio Cavalcanti Miki
Desenvolvimento de Jogos Digitais na Prática: Um Caso Aplicado ao estilo Tron/ Victor Yukio Cavalcanti Miki. – Brasília, DF, 2025.
70 p. : il. (algumas color.) ; 30 cm.
Orientador: Prof. Dr. Ricardo Matos Chaim
Trabalho de Conclusão de Curso (TCC) – Universidade de Brasília – UnB Faculdade de Ciências e Tecnologias em Engenharia - FCTE , 2025.
1. Gamificação. 2. Game development. 3. Estudo de caso. 4. Engenharia de software. I. Prof. Dr. Ricardo Matos Chaim. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Desenvolvimento de Jogos Digitais na Prática: Um Caso Aplicado ao estilo Tron

CDU 005.1:794

Victor Yukio Cavalcanti Miki

Desenvolvimento de Jogos Digitais na Prática: Um Caso Aplicado ao estilo Tron

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 28 de julho de 2025:

Prof. Dr. Ricardo Matos Chaim
Orientador

Profa. Dra. Milene Serrano
Convidado 1

Profa. Dra. Carla Denise Castanho
Convidado 2

Brasília, DF
2025

Agradecimentos

A jornada acadêmica é repleta de desafios, e chegar até aqui não teria sido possível sem o apoio e a presença de pessoas especiais. À minha família, por estar sempre por perto, oferecendo suporte incondicional nos momentos de dificuldade e sendo a base que me manteve firme ao longo desse percurso. Seu apoio foi essencial para que eu pudesse seguir em frente. Aos meus amigos, que compartilharam comigo os momentos difíceis e as conquistas ao longo da minha vida acadêmica. A companhia e a cumplicidade de vocês tornaram essa caminhada mais leve e significativa. Ao meu orientador, que aceitou me guiar neste projeto e abraçou este trabalho com entusiasmo, mesmo sendo um tema novo e pouco explorado. Sua orientação, paciência e dedicação foram fundamentais para a construção deste TCC. A todos que, de alguma forma, contribuíram para que este momento se tornasse realidade, meu mais sincero agradecimento. Esta conquista não é apenas minha, mas de todos que estiveram ao meu lado, incentivando-me a continuar. Que este seja apenas o começo de muitas realizações!

"Games são a arte de criar experiências interativas que tocam o jogador."

— Hideo Kojima

Resumo

O desenvolvimento de jogos eletrônicos é uma área multidisciplinar que exige a integração de diversas práticas e conhecimentos, como design, música, psicologia, programação e gestão de projetos. Este trabalho tem como objetivo o desenvolvimento de um jogo digital estilo Tron. O processo de desenvolvimento foi estruturado utilizando metodologias como Scrum, para organizar as fases do projeto e gerenciar o tempo e as entregas, MDA (*Mechanics-Dynamics-Aesthetics*) para guiar o design, alinhando mecânicas, dinâmicas e estética com a experiência desejada, além de Prototipagem Rápida e Design Iterativo para testar e ajustar o jogo em desenvolvimento. O trabalho aplica conhecimentos de engenharia de software, incluindo Metodologias Ágeis e estratégias de modelagem, com o intuito de oferecer uma abordagem prática e organizada para o desenvolvimento de jogos digitais.

Palavras-chave: Jogo digital. Game Design. Scrum. MDA. Tron.

Abstract

This work focuses on the development of a 2D digital game in the Tron genre, aiming to deliver a playable prototype. The development process is organized using methodologies such as Scrum for project management, ensuring proper scheduling and delivery tracking. The Mechanics-Dynamics-Aesthetics (MDA) framework will guide the game design, ensuring a cohesive experience by aligning gameplay mechanics, player interactions, and aesthetics. Additionally, Rapid Prototyping and Iterative Design will be employed to test and refine the game prototype throughout development. This project applies software engineering principles, including agile methodologies and modeling strategies, providing a structured approach to digital game development while offering practical insights into the design and implementation processes.

Key-words: Digital game, Development, Scrum, MDA, Engines, Tron.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas
autoload	Script carregado globalmente em tempo de execução pela Godot.
CamelCase	Padrão de nomenclatura em que cada palavra inicia com letra maiúscula.
CanvasLayer	Nó da Godot usado para interface sobreposta à cena principal.
containers	Nós do tipo Control utilizados para organizar automaticamente a interface gráfica com base em regras de alinhamento, preenchimento e hierarquia.
Dr.	Doutor
FCTE	Faculdade de Ciência e Tecnologia em Engenharia
GameManager	Script responsável por gerenciar transições de cenas e resolver interações durante o jogo.
GameState	Script global (autoload) responsável por armazenar e gerenciar dados persistentes do jogo, como pontuações.
HBoxContainer	Tipo de container da Godot que organiza elementos filhos horizontalmente.
HUD	Sigla para Heads-Up Display; elementos gráficos da interface que exibem informações durante o jogo, como pontuação, tempo e status.
keybinds	Associações entre ações do jogo e teclas específicas do teclado, configuráveis pelo jogador.
lobby	Tela ou ambiente onde os jogadores realizam a seleção de personagens e configurações antes do início da partida.
low-code	Abordagem de desenvolvimento que permite criar funcionalidades com pouco ou nenhum código, geralmente por meio de interfaces visuais.
MarginContainer	Container que aplica margens internas aos seus elementos filhos, útil para criar espaçamento interno uniforme.
MDA	Mechanics, Dynamics, Aesthetics (Mecânicas, Dinâmicas e Estética).

Node2D Tipo de nó base para objetos 2D na Godot Engine.

p. Página

`physics_process` Função de script da Godot chamada em intervalos fixos, utilizada para lógica relacionada à física e movimentação contínua.

S.d Sem data

`snake_case` Padrão de nomenclatura com palavras minúsculas separadas por sublinhado.

TCC Trabalho de Conclusão de Curso.

trail Rastro visual deixado pelos jogadores durante a movimentação, geralmente utilizado como elemento de colisão e estratégia no gameplay.

VBoxContainer Tipo de container da Godot que organiza elementos filhos verticalmente.

Sumário

1 Introdução

Para compreender os objetivos e as implicações deste projeto, é fundamental entender o que caracteriza um jogo. Um jogo pode ser definido como uma atividade estruturada, guiada por regras e objetivos, que envolve desafios e, muitas vezes, interação social. Segundo Johan Huizinga (2000), em sua obra *Homo Ludens*, o jogo é uma manifestação cultural que transcende o mero entretenimento, sendo parte essencial do desenvolvimento humano (HUIZINGA, 2000). Jogos têm a capacidade de estimular criatividade, habilidades de resolução de problemas e o pensamento estratégico, promovendo uma experiência lúdica que envolve engajamento emocional e intelectual.

Ao incorporar tecnologia computacional em um jogo, surge o conceito de jogo eletrônico, uma mídia interativa que combina elementos de narrativa, gráficos, áudio e jogabilidade. Os jogos eletrônicos possuem diversas categorias, e uma delas é representada pelos jogos *Tron*. Esses jogos têm origem no filme "Tron: Uma Odisseia Eletrônica", da Disney, lançado em 1982, que foi pioneiro em misturar computação gráfica avançada com *live-action* (LAWRENCE, 2002). A franquia é ambientada em um universo digital chamado "Grade"(ou *Grid*), onde os personagens interagem com programas e sistemas como se fossem pessoas. Um dos principais elementos desse estilo são os "*Light Cycles*", corridas em que os jogadores deixam rastros luminosos que funcionam como barreiras mortais. O objetivo é "trancar"os oponentes em um espaço limitado.

Considerando os conceitos e os aspectos citados, junto com os conhecimentos adquiridos no curso de Engenharia de Software, a proposta deste trabalho é desenvolver o software de um jogo digital utilizando elementos dos *Light Cycles* do estilo de jogo *Tron*. Além disso, busca-se apresentar a metodologia utilizada no desenvolvimento do jogo digital, com a finalidade de servir como objeto de estudo e aprendizagem.

Objetivo Geral

Desenvolver um jogo digital inspirado nos *Light Cycles* do universo *Tron*, A experiência servirá como um estudo de caso para compreender o ciclo de desenvolvimento, aplicando Metodologias da engenharia de software e fornecendo um guia para futuros desenvolvedores.

Objetivos Específicos

- Pesquisar e analisar as características dos jogos *Tron* e suas mecânicas principais, com foco nos *Light Cycles*.
- Implementar o jogo digital empregando a Engine Godot, explorando recursos de narrativa, gráficos, áudio e jogabilidade.
- Adotar uma metodologia de desenvolvimento baseada em Scrum e MDA (Mecânica, Dinâmica e Estética), documentando cada etapa do processo.
- Integrar os conceitos de Engenharia de Software, como levantamento de requisitos, modelagem e testes, ao ciclo de desenvolvimento do jogo.
- Realizar testes simples de caixa-preta e exploração com foco na identificação e correção de erros, garantindo o funcionamento básico e a jogabilidade do sistema.
- Apresentar o jogo e a metodologia aplicada como uma ferramenta de estudo para auxiliar iniciantes no desenvolvimento de jogos digitais.

Estrutura do Trabalho

Este trabalho está organizado em oito capítulos, cada um dividido em seções para melhor estruturação e clareza. A divisão seguiu as diretrizes fornecidas pelos professores da FCTE – Faculdade de Ciências e Tecnologias em Engenharia da Universidade de Brasília, conforme o Guia para a Elaboração de Trabalhos de Conclusão de Curso em Engenharia de Software, elaborado pelo Prof. Dr. George Marsicano Corrêa, além do template LaTeX fornecido pelo Prof. Dr. Edson Júnior e das orientações do Prof. Dr. Ricardo Matos Chaim, meu orientador.

O primeiro capítulo apresenta a introdução, oferecendo uma visão geral do trabalho, contextualizando a ideia principal do projeto e detalhando seus objetivos gerais e específicos. Além disso, descreve a estrutura do documento, facilitando a compreensão do leitor.

O segundo capítulo aborda a fundamentação teórica, reunindo os principais conceitos e áreas de conhecimento essenciais tanto para o desenvolvimento de software quanto para a criação de jogos. Esses fundamentos foram aplicados ao longo do projeto para garantir um melhor embasamento e compreensão do tema. Além disso, o capítulo apresenta as ferramentas utilizadas no desenvolvimento do jogo, destacando seus papéis no processo.

O terceiro capítulo apresenta as ferramentas utilizadas no trabalho e seus respectivos usos.

O quarto capítulo apresenta o processo de execução da pesquisa, detalhando as fases e atividades realizadas. São descritas as etapas desde a revisão da literatura até a aplicação e avaliação das técnicas utilizadas, destacando os dados coletados, as metodologias empregadas e a análise dos resultados.

O quinto capítulo apresenta as especificações do jogo, detalhando o processo criativo e as decisões tomadas durante o desenvolvimento. Além disso, conceitos e ideias são explorados com base em pesquisas, análises de materiais e etapas do processo criativo.

O sexto capítulo apresenta os resultados obtidos ao longo do desenvolvimento do jogo. Nele, são discutidas as principais lições aprendidas durante o processo, abordando desafios enfrentados, soluções adotadas e insights adquiridos. Além disso, o capítulo inclui demonstrações do jogo, destacando suas principais funcionalidades e o impacto das decisões tomadas ao longo do projeto.

O sétimo capítulo apresenta a conclusão do projeto, reunindo uma reflexão sobre todo o processo de desenvolvimento. Nele, é exposta minha opinião geral, uma autoavaliação do trabalho realizado e considerações sobre os desafios enfrentados e as soluções adotadas. Além disso, discuto as contribuições do projeto, possíveis melhorias e sugestões para trabalhos futuros.

Por fim, os últimos capítulos apresentam os apêndices e o referencial bibliográfico utilizado na elaboração do trabalho.

2 Metodologia Científica

Este capítulo apresenta a metodologia utilizada para o desenvolvimento deste projeto, fundamentando os processos adotados na criação do jogo eletrônico inspirado em *Tron: Light Cycles*. A proposta do trabalho é orientar o desenvolvimento com base em abordagens bem estabelecidas, como o Scrum e o MDA (Mecânica, Dinâmica e Estética), garantindo que cada etapa seja sistematizada e documentada de forma clara.

Além disso, são detalhadas as estratégias de modelagem aplicadas ao projeto, bem como as ferramentas utilizadas, como Godot, GDScript e Aseprite, que permitiram o desenvolvimento e a implementação do jogo. Essas escolhas têm como objetivo não apenas a entrega de um produto funcional, mas também a criação de uma base didática para futuros desenvolvedores.

2.1 Scrum

O Scrum é uma metodologia ágil que foi escolhida para gerenciar o desenvolvimento do projeto devido à sua capacidade de promover colaboração, flexibilidade e iterações rápidas. Essa abordagem divide o projeto em ciclos curtos, chamados *sprints*, que têm duração definida e objetivos claros. Ao final de cada sprint, é possível revisar os resultados alcançados e ajustar o planejamento para os próximos passos (SCHWABER; SUTHERLAND, 2013).

No contexto deste projeto, o Scrum foi adaptado para permitir o desenvolvimento iterativo do jogo. Cada sprint envolveu as seguintes etapas:

- **Planejamento:** Definição das tarefas prioritárias, como implementação de mecânicas, design de níveis e testes.
- **Execução:** Desenvolvimento das funcionalidades planejadas, utilizando as ferramentas escolhidas.
- **Revisão:** Testes e validação das funcionalidades implementadas, com foco na experiência do usuário.

Mesmo em um projeto individual, o uso do Scrum promove organização das etapas e entrega contínua de resultados tangíveis. Para montar o cronograma, foi considerado o período de 2 meses e meio para elaboração do trabalho, com sprints de duas semanas.

Nos Quadros 1 e 2 temos a identificação dos requisitos funcionais e não funcionais, respectivamente.

Tabela 1 – Requisitos funcionais do jogo

ID	Requisito	Descrição	Prioridade
RF01	Criar e gerenciar salas de jogo	Permitir que até 6 jogadores ingressem e joguem juntos em uma mesma sala.	Alta
RF02	Controlar os personagens	O jogador deve conseguir movimentar seu personagem para esquerda ou direita.	Alta
RF03	Gerar e exibir o rastro	Cada jogador deixa um rastro letal no mapa, exceto nos pontos de brechas aleatórias.	Alta
RF04	Aplicar regras de pontuação	O jogo deve calcular a pontuação conforme a equação definida e exibir o placar.	Média
RF05	Implementar poderes especiais	Os poderes devem surgir aleatoriamente no mapa e ser ativados pelo jogador que os coletar.	Alta
RF06	Detectar colisões	O jogo deve encerrar a rodada quando um jogador colidir com um rastro ou com a parede.	Alta
RF07	Reposicionar os jogadores	Após cada rodada, os jogadores devem ser reposicionados nas posições iniciais.	Média
RF08	Exibir feedback visual e sonoro	O jogo deve apresentar efeitos visuais e sonoros ao ativar poderes, colidir ou vencer a partida.	Média

Fonte: Elaboração própria.

Tabela 2 – Requisitos não funcionais

ID	Requisito	Descrição	Prioridade
RNF01	Engine de desenvolvimento	O jogo deve ser desenvolvido na Godot Engine devido à sua leveza e suporte a 2D.	Alta
RNF02	Suporte multiplataforma	O jogo deve ser compatível com Windows e Linux.	Média
RNF03	Desempenho otimizado	O jogo deve rodar a pelo menos 60 FPS em hardware modesto.	Média
RNF04	Interface responsiva	O jogo deve apresentar uma interface clara e intuitiva para facilitar o entendimento das mecânicas.	Alta
RNF05	Código modular e expansível	O código deve ser estruturado de forma a permitir futuras adições e melhorias sem grandes refatorações.	Alta
RNF06	Conectividade com controle Xbox	O usuário pode conectar um controle analógico e o game deve poder reconhecer o dispositivo.	Baixa
RNF07	Tratamento de bugs	Os bugs do jogo não devem atrapalhar a jogabilidade.	Alta

Fonte: Elaboração própria.

O Quadro 3 possui o cronograma inicial do projeto dividido em seis *sprints* com duração de 2 semanas cada.

Tabela 3 – Cronograma de desenvolvimento

Sprint	**Duração**	**Objetivo**	**Requisitos envolvidos**
Sprint 1 - Planejamento e Configuração	2 semanas	Configurar ambiente na Godot, estruturar projeto, definir assets básicos e movimentação inicial.	RF01, RF02, RNF01, RNF02
Sprint 2 - Mecânicas Básicas	2 semanas	Implementar colisões, rastros de luz e regras fundamentais da partida.	RF03, RF05
Sprint 3 - Implementação dos Poderes	2 semanas	Criar e integrar poderes especiais, garantindo variedade e equilíbrio.	RF06
Sprint 4 - Interface e Experiência do Usuário	2 semanas	Criar menus, HUD, sistema de pontuação e feedback visual para melhor UX.	RNF04, RF04
Sprint 5 - Testes e Ajustes de Jogabilidade	2 semanas	Testar fluxo do jogo localmente, ajustar balanceamento e melhorar usabilidade.	RF08, RNF03, RNF05, RF07
Sprint 6 - Refinamento e correção de bugs	1 semana	Revisão final, correção de bugs e otimizações.	RNF07, RNF06

Fonte: Elaboração própria.

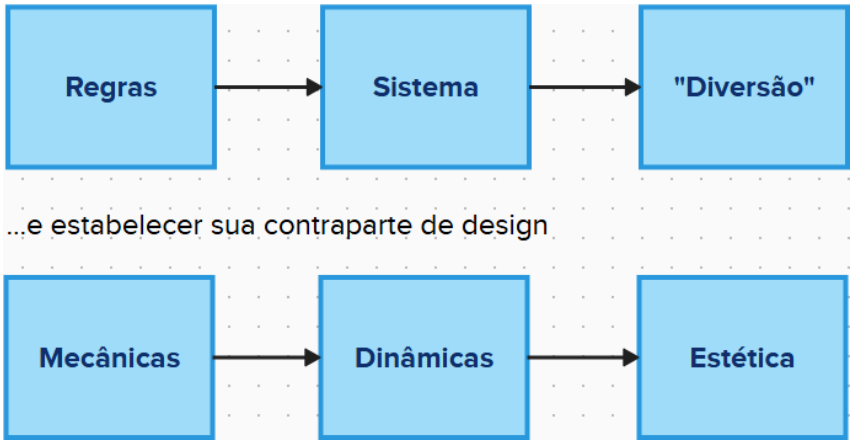
A metodologia Scrum foi escolhida por já fazer parte do repertório de práticas com as quais havia familiaridade. Ter experiência prévia facilita a aplicação e torna mais simples adaptar a técnica ao contexto de um projeto individual. Como não se trata de um trabalho em equipe, nem todos os ritos tradicionais foram seguidos. Optou-se por aplicar apenas o que realmente faria diferença na organização e ritmo de trabalho. No desenvolvimento espera-se observar como essas técnicas foram utilizadas.

2.2 MDA (Mecânica, Dinâmica e Estética)

O modelo MDA foi adotado como base para estruturar os elementos do jogo. Este modelo é amplamente utilizado no desenvolvimento de jogos digitais por oferecer uma perspectiva clara sobre como cada componente de um jogo contribui para a experiência do jogador. Assim como o autor Johan Huizinga, o estudo feito por Hunicke em *"MDA: A Formal Approach to Game Design and Game Research"* formaliza o consumo dos jogos em três componentes distintos e estabelece uma relação de contraparte para o desenvolvimento de jogos (HUNICKE; LEBLANC; ZUBEK, 2004).

A Figura 1 ilustra essa divisão, mostrando como a mecânica (regras e sistemas) influencia a dinâmica (comportamento emergente durante o jogo) e, por fim, a estética (emoções e experiências divertidas geradas no jogador).

Figura 1 – Componentes de um jogo modelo MDA



Adaptado de: HUNICKE, Robin; LEBLANC, Marc; ZUBEK, Robert. MDA: A Formal Approach to Game Design and Game Research. 2004. p. 2.

Para entender melhor o que é cada camada, fazemos a seguinte descrição:

- **Mecânica**: Refere-se às regras, interações e sistemas que governam o funcionamento do jogo. No contexto do *Light Cycles*, inclui aspectos como o controle do veículo, a criação de barreiras luminosas e a detecção de colisões.
- **Dinâmica**: Diz respeito às interações emergentes resultantes da aplicação das mecânicas. Exemplos incluem o comportamento dos jogadores ao tentar "trancar" os oponentes ou escapar de barreiras.
- **Estética**: Envolve a experiência emocional e sensorial do jogador. Neste projeto, busca-se criar uma experiência visual simples, uma vez que o tempo é curto e o foco está no conteúdo funcional e didático.

O uso do MDA possibilita uma abordagem holística no desenvolvimento do jogo, alinhando os aspectos técnicos à experiência pretendida para os jogadores (CARROLL, 2000).

Com base nessas definições, foram elaboradas as respectivas Tabelas 4 e 5 de Mecânica, Dinâmica e Estética do jogo proposto.

Tabela 4 – MDA: Mecânicas do jogo

Mecânicas	**Descrição**	**Observação**
Movimentação	O jogador pode girar para a esquerda ou para a direita, sendo a curva maior conforme a velocidade.	A mecânica exige precisão e estratégia para evitar colisões.
Rastros	Cada jogador deixa um rastro letal ao se movimentar, criando barreiras no cenário.	Brechas aleatórias surgem ocasionalmente nos rastros, permitindo a passagem.
Poderes Especiais	Jogadores podem coletar poderes aleatórios que afetam a jogabilidade.	Alguns poderes afetam apenas o jogador, enquanto outros impactam todos.
Colisão e eliminação	O jogador que colidir com uma parede ou rastro é eliminado da rodada.	A rodada continua até restar apenas um jogador vivo.
Sistema de pontos	A cada rodada, jogadores recebem pontos conforme sua colocação.	O jogo termina quando um jogador atinge a pontuação necessária.

Fonte: Elaboração própria.

Tabela 5 – MDA: Estética e Dinâmicas

Estética	**Dinâmicas**
Competição	Os jogadores disputam a sobrevivência, tentando eliminar os oponentes.
Desafio	A necessidade de reflexos rápidos e pensamento estratégico cria uma experiência intensa.
Estratégia	O uso inteligente dos rastros e poderes pode definir a vitória.
Satisfação	Jogadas bem executadas e vitórias são gratificantes para o jogador.
Caos e imprevisibilidade	Os poderes aleatórios e as brechas nos rastros tornam cada rodada única.
Surpresa	Momentos inesperados ocorrem constantemente, mantendo o jogo dinâmico.
Pressão crescente	Conforme a partida avança, o espaço no mapa fica mais restrito, aumentando a tensão.
Imersão	O ritmo acelerado e a disputa constante mantêm os jogadores envolvidos.

Fonte: Elaboração própria.

2.2.1 Estratégias de Modelagem

Para garantir a consistência e qualidade do projeto, foram empregadas estratégias de modelagem baseadas em princípios da Engenharia de Software. As etapas adotadas no processo de desenvolvimento do jogo visam estruturar o fluxo de trabalho de maneira eficiente e *iterativa*, permitindo não apenas a construção do jogo, mas também a criação de um conteúdo funcional e didático para futuras implementações e para o aprendizado de novos desenvolvedores (SOMMERVILLE, 2011).

> "[...] *Os modelos são usados durante o processo de engenharia de requisitos para ajudar a extrair os requisitos do sistema; durante o processo de projeto, são usados para descrever o sistema para os engenheiros que o implementam; e, após isso, são usados para documentar a estrutura e a operação do sistema.*"
>
> (SOMMERVILLE, 2011), p. 82.

As seguintes estratégias foram usadas:

- **Levantamento de Requisitos:** A primeira etapa foi a identificação das funcionalidades essenciais do jogo. Isso incluiu a definição dos controles responsivos, garantindo que os jogadores tivessem uma experiência fluida e intuitiva, a criação de uma interface gráfica simples e acessível, e o balanceamento das mecânicas do jogo. O levantamento de requisitos foi fundamental para alinhar as expectativas do produto com as necessidades dos jogadores, além de facilitar as decisões durante o processo de desenvolvimento.

- **Modelagem de Dados:** Após o levantamento de requisitos, foi realizada a modelagem de dados, que envolveu a estruturação de informações sobre os cenários do jogo, personagens, eventos e interações. O objetivo foi organizar os dados de forma que a implementação das funcionalidades fosse eficiente, sem redundâncias e com alto desempenho. A modelagem de dados também serviu como base para a criação de scripts e animações dentro da Engine utilizada, o que contribuiu para a integração das mecânicas de forma coesa.

- **Prototipagem:** A prototipagem foi uma etapa relevante para validar as mecânicas do jogo antes de sua implementação final. Modelos iniciais das interações e funcionalidades foram criados, permitindo testar as ideias em um estágio inicial e fazer ajustes rápidos com base no feedback obtido. A prototipagem não só acelerou o processo de desenvolvimento, mas também proporcionou uma visão prática de como as mecânicas poderiam ser experienciadas pelos jogadores, ajudando na identificação de melhorias e possíveis problemas de usabilidade.

A adoção dessas estratégias de modelagem visa proporcionar um processo de desenvolvimento mais organizado e focado, com potencial para resultar em um produto funcional e didático, que possa contribuir para a aprendizagem de desenvolvedores iniciantes.

2.2.1.1 Modelo de Caso de Uso

A modelagem de casos de uso, conforme definido por Sommerville (2011), é uma técnica essencial na engenharia de software para elicitar e especificar os requisitos funcionais de um sistema. Um caso de uso representa uma interação entre um ator (um usuário ou outro sistema) e o sistema em si, descrevendo uma sequência de ações que o ator realiza para atingir um objetivo específico.

No contexto do desenvolvimento de jogos, a modelagem de casos de uso oferece uma maneira clara e concisa de descrever as interações dos jogadores com o jogo, auxiliando no projeto e implementação das funcionalidades. A abordagem escolhida para o jogo da cobrinha 2D buscou representar as ações dos jogadores e as mecânicas do jogo de forma abrangente, utilizando um diagrama de casos de uso que incorpora elementos como atores, casos de uso, relacionamentos (*include* e *extend*) e anotações.

Atores Os atores, Jogador 1 e Jogador 2, representam os jogadores humanos que interagem com o jogo. Eles são os iniciadores das ações e se beneficiam das funcionalidades oferecidas pelo jogo.

Casos de Uso Os casos de uso descrevem as ações que os jogadores podem realizar ou que ocorrem como parte do jogo. A listas e a Figura 2 abaixo ilustram as ações dos jogadores e do sistema dentro do jogo.

- **Jogar**: Abrange toda a experiência do jogador no jogo, desde o início até o fim.
- **Mover Animal**: Representa a ação do jogador de controlar a direção da cobrinha (esquerda ou direita).
- **Gerar rastro**: depois que o jogador se deslocar de sua posição um rastro é gerado no lugar como um novo obstáculo a ser evitado.
- **Coletar Poder**: Descreve a mecânica do jogo em que a cobrinha coleta automaticamente um poder ao passar por cima do item.
- **Colidir**: Representa o evento em que a cobrinha colide com a parede ou com o próprio corpo.
- **Morte**: Descreve o resultado da colisão, em que a cobrinha perde uma vida ou o jogo termina.

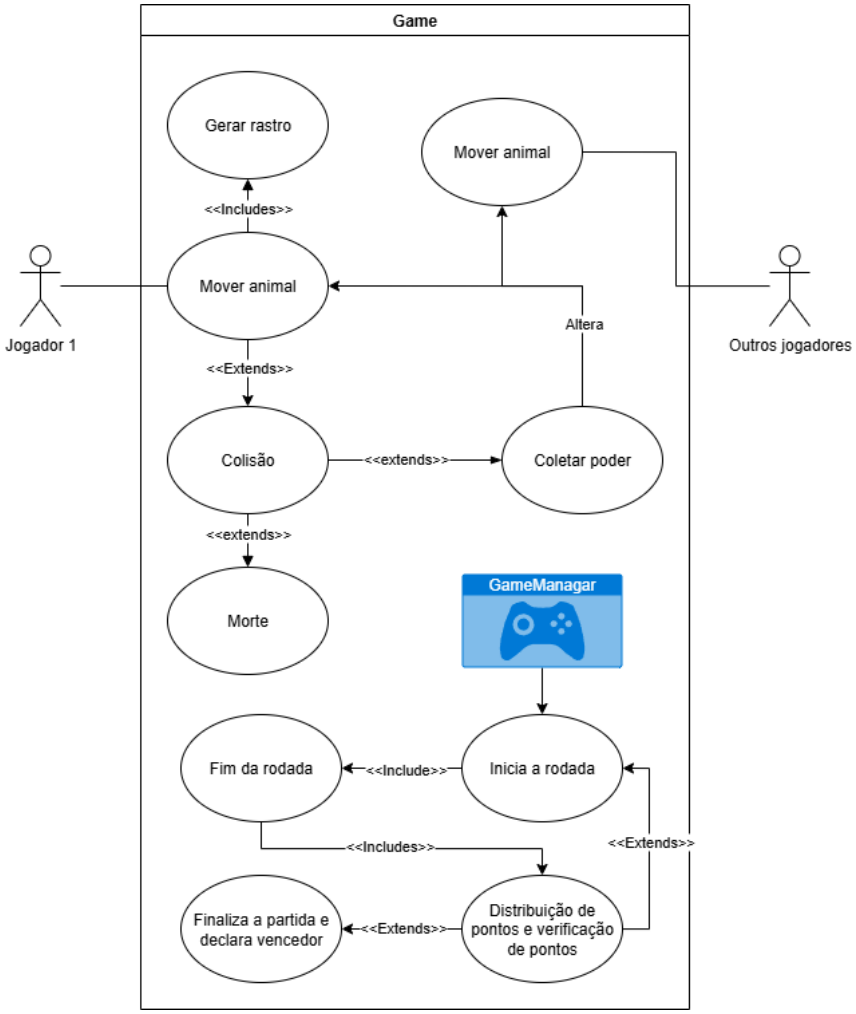
- **Inicía a Rodada**: Descreve o início da rodada, no qual os jogadores
- **Fim da rodada**: Representa o momento em que uma rodada termina, seja por um jogador perder ou por outros critérios.
- **Distribuição de pontos e verificação de pontos**: Ação de receber pontos ao final de uma rodada, de acordo com o desempenho e verifica se houve um ganhador.
- **Fim do jogo**: Ocorre quando um jogador atinge o limite de pontos ou quando todas as rodadas são concluídas.

Relacionamentos

Include: O relacionamento *include* («include») é utilizado para indicar que um caso de uso é parte integrante de outro. Por exemplo, "Jogo" inclui "Rodada", "Rodada" inclui "Fim da Rodada" e "Fim da Rodada" inclui "Ganha Pontos".

Extend: O relacionamento *extend* («extend») é utilizado para representar variações ou extensões de um caso de uso. Por exemplo, "Mover" pode ser estendido por "Colidir" e "Mudar Direção" pode ser estendido por "Coletar Poder".

Figura 2 – Diagrama casos de uso



Fonte: Elaboração própria.

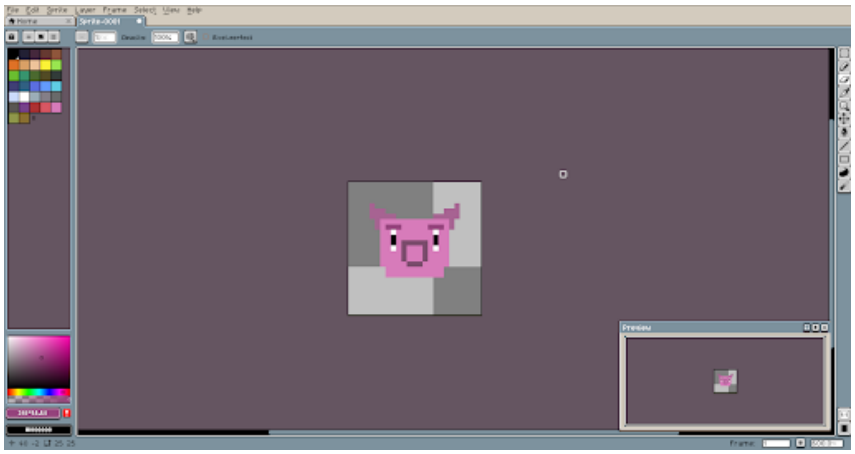
3 Ferramentas utilizadas

O desenvolvimento do jogo contou com o apoio de ferramentas específicas, selecionadas pela sua capacidade de atender às demandas do projeto. As principais ferramentas utilizadas foram:

3.1 Aseprite

O Aseprite foi utilizado para criar os elementos gráficos do jogo, como sprites de personagens, veículos e efeitos visuais. Essa ferramenta é amplamente reconhecida no desenvolvimento de jogos 2D devido à sua interface intuitiva e recursos especializados em *pixel art* (ASEPRITE, 2024). A Figura 3 ilustra um exemplo prático da tentativa de criar um design para cada animal.

Figura 3 – software aseprite



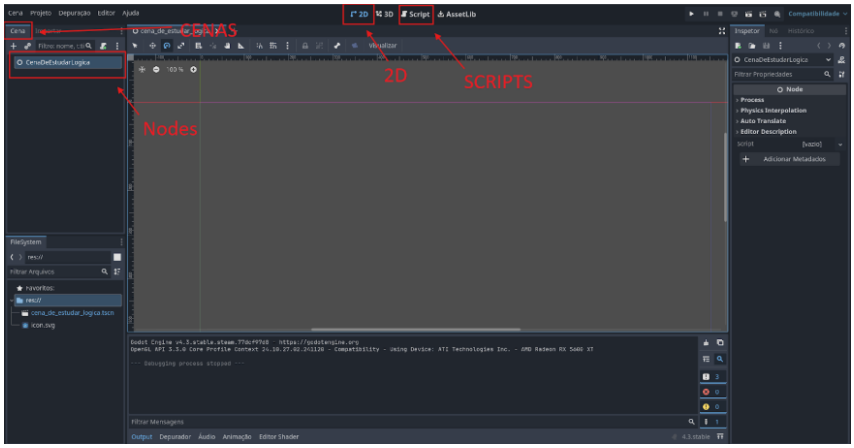
Fonte: Elaboração própria.

3.2 Godot e GDScript

A Engine Godot foi escolhida como plataforma principal para o desenvolvimento do jogo, devido à sua versatilidade e comunidade ativa. O uso do GDScript, uma linguagem integrada à Engine, permitiu a implementação de mecânicas e sistemas com agilidade. A Godot oferece recursos nativos para jogos 2D, como sistemas de colisão e animações, que foram essenciais para o projeto (ENGINE, 2024a). A Figura 4 apresenta a interface godot com algumas marcações dos principais recursos dentro do jogo (Nodes, Cenas, Scripts e a previsualização 2D).

Além disso, a Godot fornece suporte para múltiplas plataformas, abrangendo os principais sistemas operacionais e navegadores web.

Figura 4 – software Godot



Fonte: Elaboração própria.

3.3 Trello

O Trello foi utilizado para gerenciar as tarefas do projeto, permitindo organização visual e controle do progresso em cada sprint. Essa ferramenta ajudou a manter o fluxo de trabalho alinhado com os princípios do Scrum.

Ao utilizar essas ferramentas em conjunto, foi possível criar um ambiente de desenvolvimento eficiente, onde cada etapa do processo contribuiu para o alcance dos objetivos do projeto.

3.4 GitHub

O GitHub foi utilizado para gerenciar as versões do projeto e fazer backups na nuvem. Ele permitiu acompanhar o progresso do trabalho, facilitando o controle das mudanças feitas no código e garantindo que o projeto estivesse sempre seguro e acessível. O repositório com o código-fonte está disponível no *Anexo A*.

4 Referencial Teórico

4.1 História e Evolução dos Jogos Digitais

Neste capítulo, exploraremos a história dos jogos eletrônicos e sua evolução ao longo do tempo, com um foco específico no Brasil, estabelecendo uma conexão direta com o tema deste documento.

4.1.1 Primeiros Jogos e o Surgimento da Indústria

O desenvolvimento dos primeiros jogos eletrônicos ocorreu em um cenário muito diferente do atual, quando os próprios criadores precisavam construir seus sistemas do zero, sem *engines* ou ferramentas especializadas. Jogos como *Tennis for Two* (1958) e *Spacewar!* (1962) foram experimentos acadêmicos desenvolvidos em computadores de grande porte, sem qualquer preocupação comercial. A indústria começou a se consolidar na década de 1970, com o lançamento de *Pong* (1972) pela Atari, popularizando o conceito de jogos eletrônicos como uma forma viável de entretenimento (KENT, 2001).

No Brasil, o contato com os jogos eletrônicos começou nos anos 1980, ainda com forte influência externa. Naquela época, a maior parte dos consoles e jogos chegava importada ou por meio de clones nacionais, como o Telejogo da Philco/Ford e os modelos compatíveis com o Atari 2600. A ausência de uma indústria formal robusta e as barreiras de importação levaram à popularização de cartuchos pirateados e adaptações locais. Nas décadas seguintes, computadores como o MSX e os primeiros PCs também ajudaram a criar uma base de jogadores e curiosos por programação e desenvolvimento de games, ainda que de forma amadora e restrita. (DINIZ RODRIGO GAVIOLI, 2024)

4.1.2 Jogos nos Dias Atuais e Tendências

Atualmente, a indústria dos jogos digitais está mais acessível, principalmente devido ao surgimento de *engines* como Unity, Unreal Engine e Godot, que permitem que estudantes e desenvolvedores independentes criem jogos sem precisar programar tudo do zero. Esse avanço é particularmente relevante no Brasil, onde o desenvolvimento de jogos ainda enfrenta desafios, como a falta de investimentos e o alto custo de hardware. Além disso, o acesso a hardware e computadores para o desenvolvimento de jogos está bem mais barato, o que incentiva mais pessoas a explorarem e descobrirem cada vez mais sobre o jogos eletrônicos.

Hoje, o Brasil é o maior mercado de games da América Latina e ocupa a 10ª posição

mundial em receita, movimentando cerca de US$ 2,6 bilhões em 2023, com previsão de alcançar US$ 3,5 bilhões até 2025. O número de estúdios saltou de 150, em 2014, para mais de 1.000 em 2024, empregando mais de 13 mil profissionais. Esse crescimento é impulsionado por fatores como avanços tecnológicos, a cultura gamer consolidada e o Marco Legal dos Jogos Eletrônicos (Lei 14.852/2024), que estabelece regras claras para fabricação e comercialização. Com uma comunidade estimada em mais de 100 milhões de jogadores, o país não apenas consome, mas também exporta criatividade, atraindo parcerias internacionais e se firmando como um polo promissor no cenário global (Avell, 2025).

Um dos fenômenos mais marcantes da atualidade é o crescimento dos jogos *indie*, desenvolvidos por pequenos estúdios ou até mesmo por indivíduos. Diferente das grandes produções [1], que exigem orçamentos milionários, os jogos independentes apostam em criatividade, mecânicas inovadoras e narrativas únicas para conquistar o público. Plataformas como a Steam facilitaram a distribuição desses jogos, permitindo que produções de baixo orçamento alcançassem grandes sucessos.

Com essa democratização das ferramentas e do acesso ao conhecimento, o desenvolvimento de jogos digitais se tornou uma oportunidade viável para iniciantes aprenderem na prática. Este trabalho explora esse processo por meio da criação de um jogo estilo Tron, analisando os desafios e aprendizados envolvidos.

4.2 Desenvolvimento de Jogos

Neste capítulo, abordaremos a parte técnica do trabalho, com foco no desenvolvimento de jogos utilizando práticas de engenharia de software. O objetivo é demonstrar como essas práticas podem contribuir para a criação de jogos eletrônicos, considerando que, assim como qualquer outro produto digital, um jogo também é essencialmente composto por software. Também discutiremos, de forma breve, o funcionamento das *engines* de jogos, ressaltando que, embora o foco do trabalho não seja um estudo aprofundado sobre *engines*, elas desempenham um papel fundamental no processo de desenvolvimento.

4.2.1 Engenharia de Software Aplicada a Jogos

No desenvolvimento de jogos, a Engenharia de Software desempenha um papel fundamental para garantir que o processo seja eficiente, escalável e focado na entrega de um produto de alta qualidade.

> "Engenharia de software é uma disciplina de engenharia cujo foco está em todos os aspectos da produção de software, desde os estágios iniciais da especificação do

[1] o termo triple A ou AAA se refere a títulos de jogos com altos orçamentos

> sistema até sua manutenção, quando o sistema já está sendo usado." (SOMMERVILLE, 2011)

A aplicação de práticas de Engenharia de Software no contexto de desenvolvimento de jogos não se limita à codificação, mas envolve também o gerenciamento de projetos, design e testes.

Conforme Nikhil Malankar discute em seu vídeo, o ciclo de vida de um jogo segue uma estrutura semelhante ao de um software, com etapas de testes e elaboração de requisitos, desde a escolha da plataforma até a implementação final (MALANKAR, 2023).

No projeto em questão, as práticas de engenharia de software foram úteis para organizar, planejar e orientar as etapas de desenvolvimento do jogo. Nas etapas iniciais, mesmo sem iniciar o desenvolvimento, já pude perceber como a aplicação de metodologias como Scrum e MDA pode trazer clareza e eficiência para o processo. Ao seguir essas práticas, o projeto será conduzido de maneira estruturada, com entregas claras e bem definidas em cada ciclo. Além disso, a abordagem de modelagem e os testes iterativos contribuirão para refinar o jogo conforme ele evolui, minimizando erros e ajustando o produto para oferecer a melhor experiência possível.

4.2.2 Escolha da Engine para o Jogo

Para desenvolver um jogo, é essencial definir as necessidades técnicas. As *engines*, ou motores de jogos, desempenham um papel central nesse processo. Uma *engine* é um software que integra um conjunto de ferramentas e recursos projetados para simplificar e otimizar o desenvolvimento de jogos, abrangendo elementos como gráficos, física, som e muito mais. Além de acelerar o processo de produção, o uso de uma *engine* garante maior eficiência e qualidade no resultado final (STUDIOS, 2014).

Embora seja possível criar uma *engine* própria, essa abordagem geralmente é recomendada apenas em casos específicos, como atender a requisitos altamente personalizados ou aprofundar o entendimento técnico do desenvolvimento de jogos. No entanto, construir uma *engine* do zero é uma tarefa complexa e demorada, exigindo meses de trabalho para implementar funcionalidades básicas, como renderização gráfica e gerenciamento de recursos, antes mesmo de iniciar o desenvolvimento do jogo em si. Por outro lado, *engines* amplamente utilizadas, como Unity, Unreal Engine e Godot, já oferecem essas funcionalidades de maneira robusta e otimizada. Além disso, elas contam com suporte técnico, documentação abrangente e comunidades ativas, permitindo que o desenvolvedor concentre seus esforços na criação e no design do jogo, sem a necessidade de programar ferramentas fundamentais a partir do zero (ULLMANN et al., 2025).

Para garantir a escolha mais adequada da *engine* para este projeto, foi realizado um estudo comparativo entre duas opções amplamente reconhecidas: Unity e Godot. Cada

uma foi avaliada com base em critérios específicos, como:

- Facilidade de aprendizagem e qualidade da documentação;
- Suporte a múltiplas plataformas;
- Licenciamento e custos;
- Linguagem de programação utilizada;
- Recursos e ferramentas disponíveis;
- Adequação ao tipo de jogo a ser desenvolvido.

Este estudo forneceu uma base sólida para a escolha da *engine* mais alinhada às necessidades e objetivos do projeto.

4.2.2.1 Comparação entre Game Engines (Godot vs Unity)

O desenvolvimento de jogos exige a escolha de uma *engine* que atenda às necessidades do projeto. Neste comparativo, analisamos Unity 6 e Godot 4 como principais alternativas, considerando fatores como facilidade de uso, desempenho em jogos 2D, custos e requisitos técnicos. Como o objetivo deste trabalho é apresentar um guia e desenvolver um jogo de baixo custo, a escolha deve priorizar acessibilidade e eficiência (ENGINE, 2024b) (TECHNOLOGIES, 2024).

As Tabelas 6 e 7 mostram comparativos técnicos das engines relacionados a suas funcionalidades e desempenho.

Tabela 6 – Análise comparativa geral entre engines

Critério	**Unity 6**	**Godot 4**
Facilidade de Uso	Interface robusta com muitas funcionalidades, mas complexa para iniciantes. Utiliza C# como principal linguagem de programação.	Interface leve e simplificada, curva de aprendizado menos íngreme. Oferece GDScript, semelhante ao Python, facilitando o desenvolvimento.
Desempenho em Jogos 2D	Suporte a 2D, mas originalmente projetado para 3D. Recursos são adaptados do ambiente 3D, podendo resultar em menos eficiência.	Motor 2D do Godot, apesar de funcionar tanto para 3D como 2D, o motor 2D é considerado um aspecto forte da engine.
Custo e Licenciamento	O uso é gratuito, mas após o lançamento, o modelo de licenciamento pode incluir taxas baseadas no número de downloads.	Open-source e totalmente gratuito, sem taxas ou restrições comerciais.

Fonte: Autoria própria.

Tabela 7 – Comparativo técnico baseado em requisitos

Requisitos Técnicos	**Unity 6**	**Godot**
CPU	X64 com suporte a SSE2 ou ARM64. Exemplo: Intel Core 2 Duo E8200, AMD Athlon XE BE-2300.	X86_32 com SSE2, X86_64 ou ARMv8. Exemplo: Intel Core 2 Duo E8200, Raspberry Pi 4.
GPU	DX10, DX11, DX12 ou Vulkan-capaz. Exemplo: Intel HD Graphics 5500, AMD Radeon R5.	Vulkan 1.0 ou OpenGL 3.3. Exemplo: Intel HD Graphics 2500, AMD Radeon R5.
RAM	Mínimo de 8GB, recomendado 16GB ou mais para projetos complexos.	Nativo: 4GB; Web editor: 8GB.
Armazenamento	Ocupa mais espaço em disco, especialmente com projetos grandes.	200MB para execução; exportação requer 1.3GB.
Sistema Operacional	Windows 10 21H1+, macOS 11+ (Big Sur), Ubuntu 22.04+	Windows 7+, macOS 10.13+, Linux pós-2016, Web Editor compatível com navegadores modernos.

Fonte: Elaboração própria.

4.2.2.2 Análise e critérios para escolha da engine

Após a análise comparativa entre as duas Engines, Godot foi escolhida como a Engine mais adequada para o desenvolvimento deste jogo. A decisão foi fundamentada em vários critérios técnicos e de projeto, conforme detalhado abaixo.

Facilidade de uso e curva de aprendizagem : A simplicidade da interface do Godot e a utilização do GDScript, que é uma linguagem de programação semelhante ao Python, tornam o desenvolvimento mais acessível, especialmente para quem está começando ou tem um foco maior na parte lógica do jogo. Isso permite uma curva de aprendizado mais suave, o que é um ponto crucial dado o prazo do projeto e a necessidade de uma implementação eficiente.

Desempenho em Jogos 2D : Embora o Unity ofereça suporte robusto para jogos 2D, a Godot foi projetada desde o início com um motor 2D altamente otimizado. Isso garante que o desempenho da Engine em jogos bidimensionais seja superior, além de permitir um maior controle sobre o comportamento do jogo, o que é essencial para um projeto que visa ser leve e de baixo custo, como o proposto.

Custo e Licenciamento : Godot é open-source e totalmente gratuita, sem custos adicionais ou limitações comerciais, o que representa uma vantagem significativa para o projeto. Não há taxas de licenciamento, e o código-fonte da Engine pode ser modificado conforme as necessidades específicas do desenvolvimento. Esse fator elimina preocupações com custos futuros e garante flexibilidade total, além de facilitar a utilização sem complicações de licenciamento.

Requisitos Técnicos : A Godot exige menos recursos de hardware, o que torna o desenvolvimento mais ágil, especialmente em termos de tempo e capacidade de testes. Com um espaço de armazenamento inicial de apenas 200MB e suporte para uma ampla gama de sistemas operacionais, como Windows, macOS e Linux, a Engine se adequa bem aos requisitos de recursos do projeto e possibilita um desenvolvimento mais fluido, sem depender de máquinas muito potentes.

Adequação ao tipo de jogo : Como o projeto é voltado para a criação de um jogo 2D com mecânicas simples de movimento e interação, a Godot oferece ferramentas e funcionalidades que atendem perfeitamente às necessidades do jogo. O motor 2D da Godot é mais direto e flexível para o tipo de mecânica que estamos desenvolvendo, sem a necessidade de adaptações que seriam necessárias em outras Engines.

Por todas essas razões, a Godot se mostrou a escolha mais adequada para este projeto, considerando tanto o orçamento, a complexidade técnica e a necessidade de uma Engine eficiente para o desenvolvimento de jogos 2D de baixo custo.

4.2.2.3 Conhecendo a Engine Godot

Como já foi mencionado anteriormente, desenvolver sua própria Engine de jogos é uma tarefa desafiadora e envolvente. Porém, para entender como a Engine Godot funciona, é interessante compreender quais as funcionalidades mínimas de uma Engine de jogos.

Para desenvolver uma Engine de jogo, é necessário implementar alguns sistemas essenciais. Esses sistemas são:

- **Inicialização do Sistema**: Basicamente, é abrir uma janela, obter o contexto gráfico (OpenGL/DirectX/Vulkan) e inicializar o áudio.
- **Controle de Tempo ou Game Loop**: Todo jogo precisa ter um loop para controlar a taxa de atualização e renderização do jogo.
- **Entrada de Dados**: Implementar a captura de entradas (botões pressionados).
- **Renderização**: Utilizar computação gráfica para renderizar as texturas na tela.
- **Utilitários Matemáticos**: Bibliotecas de matemática (vetores e matrizes) e funções úteis para o desenvolvimento.
- **Gestão de Objetos e Cenas**: Sistema para gerenciar objetos e cenas à medida que seu jogo se torna mais complexo.
- **Áudio**: Suporte para tocar músicas e efeitos sonoros.
- **Carregamento de Arquivos**: Utilizar um gerenciador de arquivos para evitar o carregamento redundante e permitir a adição de recursos como mods.

Tudo isso é apenas o básico, e cada sistema pode variar muito em nível de complexidade Glaiel (2021).

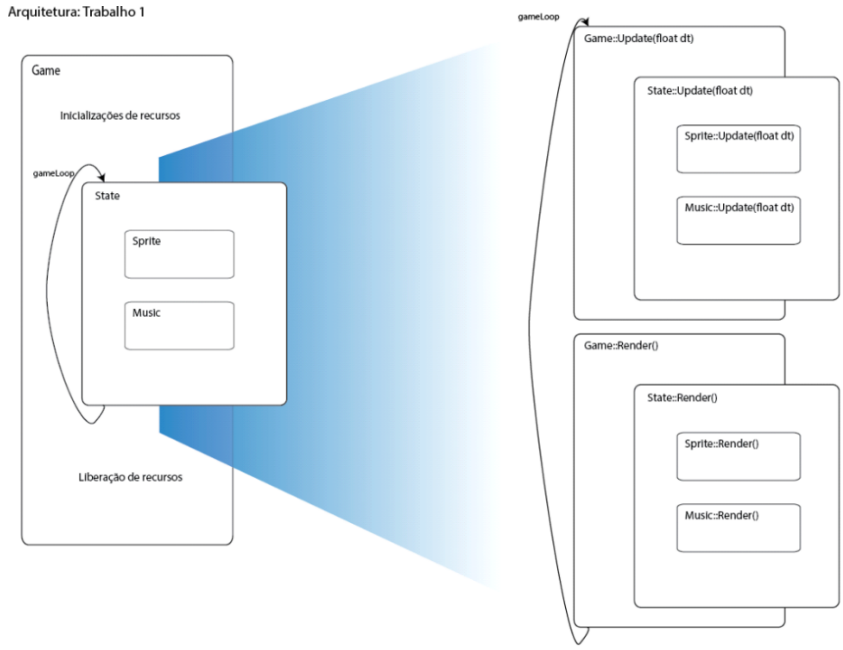
A figura 5 abaixo representa um exemplo e arquitetura de engine no qual conta com *GameState* que controla a iniciação do sistema e liberação de recursos, *GameLoop* que é responsável pela atualização de quadro e o estado dos objetos intanciado na cena. Imagem busca facilitar a visualização dos recursos de uma engine e fluxo de seu comportamento.

Agora, vamos entender como o Godot traduz tudo isso para dentro de sua Engine.

4.2.2.3.1 Nodes

No Godot, *nodes* (ou nós) são os elementos fundamentais que compõem qualquer cena. Existem dezenas de tipos de *nodes*, cada um com uma função específica, como representar objetos gráficos, controlar física, lidar com entradas de usuário ou até organizar o layout de outros nós. Eles são como os sistemas mencionados anteriormente.

Figura 5 – Game Loop, Sprite e Music



Fonte: Introdução de Desenvolvimento de Jogo - Departamento de Ciência da Computação UnB.

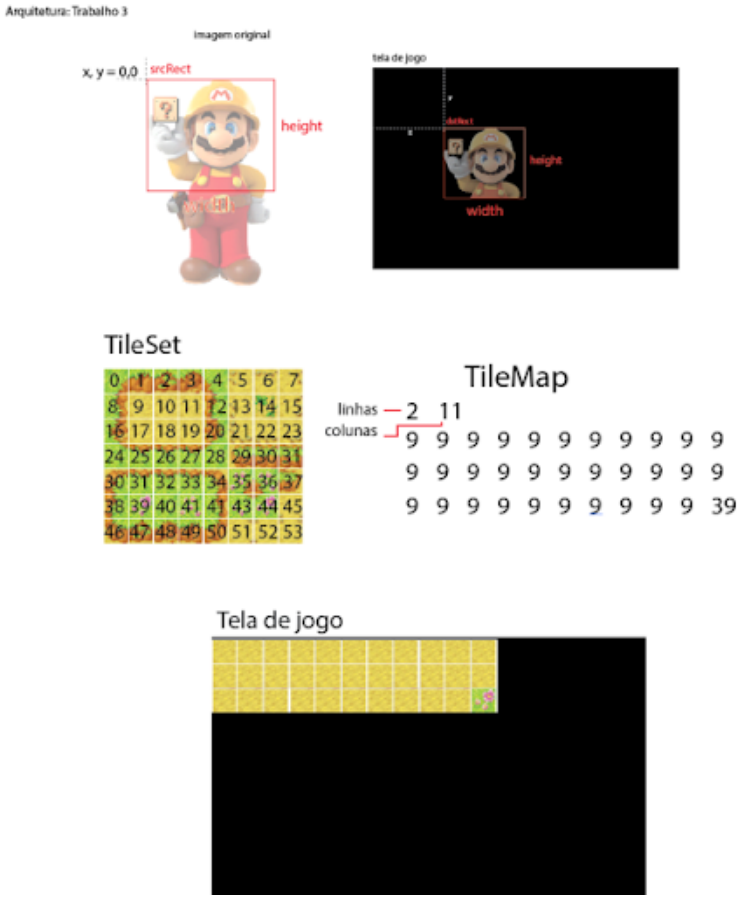
"Nodes are the fundamental building blocks of your game. They are like the ingredients in a recipe. There are dozens of kinds that can display an image, play a sound, represent a camera, and much more." (GODOT ENGINE 4.3 documentation in English, s.d., p. 1)

4.2.2.3.2 Scenes

As cenas são as telas que contêm os *Nodes*. Para entender melhor as cenas podemos exemplificar de duas maneiras: através de menus/telas ou *chunks*. No caso do jogo que estamos desenvolvendo, cada cena é representada por uma tela. Já em jogos com grandes mundos, onde a câmera segue o jogador, as cenas podem ser *chunks* e apenas os *chunks* necessários são carregados para gerar a cena.

Quem controla os recursos para gerar as cenas são os scripts do programador (de forma manual) ou, de forma dinâmica, usando *TileSet*, *TileMap* e recursos de *Resource Management*.

Figura 6 – Tile Set, Tile Map e Resource Management



Fonte: Trabalho 3 - Introdução de Desenvolvimento de Jogo - Departamento de Ciência da Computação UnB.

5 Especificação do jogo

Neste capítulo, apresentamos os conceitos gerais do jogo produzido, desde o objetivo, narrativa, regras, mecânicas e funcionalidades. A especificação do jogo é um documento essencial para definir de maneira clara e detalhada todos os aspectos que compõem a experiência do jogador, servindo como guia tanto para a equipe de desenvolvimento quanto para possíveis ajustes e melhorias no decorrer do projeto. A partir desta especificação, é possível alinhar as expectativas e assegurar que o produto final atenda aos requisitos propostos, oferecendo uma base sólida para a implementação e testes do jogo.

5.1 Narrativa

O jogo tem várias inspirações, sendo a principal delas a série "Tron – Uma Odisseia Eletrônica". Ele utiliza a mecânica dos *light-cycles*, que são rastros deixados pelos jogadores no filme. *Light-cycle* é um sub-jogo da série no qual o objetivo é forçar os jogadores a colidirem com a parede ou com os rastros de luz deixados pelos demais, muito similar ao clássico jogo *Snake*, também conhecido como jogo da cobrinha, mas com elementos de *battle royale*.

Outra inspiração relevante é o jogo de navegador "Curve Fever", cuja proposta é semelhante à adotada neste projeto. A ideia é criar uma versão renovada do jogo para guiar o aprendizado.

Figura 7 – Jogo Achtung Die Kurve

Fonte: https://www.youtube.com/watch?v=FHyALtYMfPY

A narrativa do jogo é relativamente simples. Até 6 jogadores podem ingressar numa sala virtual, onde cada um controla um dos 6 animais disponíveis. Cada animal é

representado por uma cor específica: porco (rosa), dragão (vermelho), serpente (verde), baleia (azul), lobo (branco) e águia (amarelo).

Os jogadores competem em várias rodadas e acumulam pontos de acordo com sua colocação. O número de pontos necessários para o fim da partida é definido pela equação:

$$\textbf{Pontuação para vitória} = \textbf{número de jogadores} \times 5$$

O número mínimo de jogadores para iniciar uma partida é 2. A pontuação obtida em cada rodada segue a regra:

$$\textbf{Pontos da rodada} = \textbf{número de jogadores} - \textbf{colocação do jogador}$$

O primeiro jogador a atingir a pontuação definida é coroado como *Rei dos Animais*.

5.2 Objetivo do Jogo

O objetivo do jogo é sobreviver o maior tempo possível dentro de um cenário limitado. Por se tratar de um jogo competitivo *PvP*, o último jogador vivo será o vitorioso na rodada.

Para alcançar esse objetivo, os jogadores podem adotar estratégias variadas, utilizar poderes especiais que tornam a partida mais caótica, além de exigir habilidade motora para controlar seu personagem com precisão.

5.3 Estrutura da Fase

A estrutura da fase é simples: uma caixa com tamanho adaptável ao número de jogadores, fundo cinza-escuro e bordas cinza-claro. Ao lado da arena, há um placar exibindo a colocação dos jogadores.

Ao final de cada rodada, a área de jogo é limpa e os jogadores são reposicionados em suas posições iniciais.

5.4 Poderes e Habilidades

O jogo oferece uma habilidade padrão e diversos poderes especiais que surgem aleatoriamente durante a partida.

A habilidade padrão consiste em deixar um rastro letal por onde o personagem se move, capaz de eliminar qualquer jogador (inclusive o próprio). O rastro pode ocasionalmente conter brechas que permitem a passagem.

Os poderes especiais aparecem aleatoriamente no mapa. Ao serem coletados, são ativados imediatamente e têm duração de 7 segundos. Eles podem ajudar ou prejudicar o jogador que os coletou ou afetar os demais.

- +Velocidade
- -Velocidade
- +Velocidade para os outros
- -Velocidade para os outros
- Inverter controles para os outros
- Andar em 90º
- Tornar bordas atravessáveis
- Limpar mapa
- Voar
- +Tamanho
- -Tamanho

Poderes verdes: aplicam-se somente ao jogador que os coletou.
Poderes vermelhos: afetam todos os outros jogadores.
Poderes azuis: afetam todos os jogadores.

Para uma descrição visual dos poderes, consulte o **Apêndice 2**.

5.5 Mecânicas e Jogabilidade

A mecânica do jogo: o jogador pode mover-se para a esquerda ou para a direita. No entanto, a movimentação dos personagens é limitada a curvas com raio pré-determinado. A velocidade do jogador influencia diretamente o tamanho da curva: quanto maior a velocidade, maior será o raio da curva.

5.6 Gamificação e Análise Crítica

A escolha desse tipo de jogo envolveu diversos fatores, como a facilidade de criação de *assets* (arte do jogo) e desenvolvimento curto devido ao prazo de tempo. Reforçando a ideia deste projeto ser utilizá-lo como estudo de caso, visando gerar resultados relevantes para a pesquisa.

Apesar da mecânica simplificada, o jogo conta com diversas funcionalidades, como poderes especiais e regras que o tornam dinâmico e interessante.

Para a avaliação do engajamento, utilizou-se o *framework* Octalysis, desenvolvido por Yu-kai Chou, originalmente voltado para gamificação de atividades e comportamentos. Embora seu uso mais comum esteja relacionado a sistemas gamificados em contextos educacionais e corporativos, o Octalysis também é aplicável ao *game design* digital, pois fornece uma compreensão aprofundada das motivações que mantêm os jogadores envolvidos. Ele já foi utilizado em pesquisas de experiência do usuário (UX) em jogos digitais para identificar pontos de engajamento e orientar melhorias no design, como no caso documentado de *Candy Crush* (Game Developer Staff, 2021).

O framework é composto por oito *cores* (impulsionadores motivacionais). Cada um representa um aspecto-chave da motivação humana que pode ser explorado em um jogo. Nesta análise, selecionamos os principais *cores* presentes no protótipo e os avaliamos com base na percepção de três testadores, incluindo o autor, em uma escala de 1 a 5, onde 1 indica presença pouco explorada e 5 indica presença fortemente explorada (CHOU, 2019).

O número reduzido de participantes se deveu a limitações de tempo e disponibilidade durante a etapa de testes. Ainda que a amostra pequena limite a generalização dos resultados, ela foi suficiente para apontar ajustes iniciais nas mecânicas e confirmar elementos que contribuíram para o engajamento. O formulário de avaliação completo e os resultados estão disponíveis no Anexo C.

Principais *cores* utilizados

Epic Meaning *Resumo*: Este *core* refere-se ao senso de propósito maior ou missão. *Uso*: 1 — Embora a narrativa tente envolver os animais na floresta, ela não é muito explorada e ficou desconexa do tema Tron. Os jogadores não ficaram interessados infelizmente e acabou sendo uma prospota que aos poucos foi perdendo prioridade.

Social Influence & Relatedness *Resumo*: Relacionado à interação social e senso de comunidade. *Uso*: 4 — Como o jogo é competitivo, a interação entre os jogadores é um ponto-chave. A disputa aumenta o engajamento e a imersão, estimulando o sentimento de conexão e rivalidade. Houve problemas em alocar muitas pessoas em um único teclado, mas foi uma bagunça divertida que reuniu amigos e situações engraçadas.

Unpredictability & Curiosity *Resumo*: Este *core* envolve a curiosidade e o desejo de explorar. *Uso*: 4 — A presença de poderes aleatórios e a variabilidade no comportamento dos jogadores tornam o jogo imprevisível, mantendo o jogador curioso sobre os resultados de cada partida.

Loss & Avoidance *Resumo*: Refere-se ao medo de perder e à motivação para

evitar consequências negativas. *Uso*: 3 — Embora o jogo seja competitivo, a perda não é tratada como uma consequência punitiva. A penalização ocorre principalmente na pontuação, com perdas menores nas rodadas, o que suaviza o impacto da derrota, mas ainda causa irritação nos jogadores.

6 Resultados Obtidos

Neste primeiro estágio, foram realizadas pesquisas e definições fundamentais para a estruturação do jogo. A análise de Engines, frameworks de gamificação e metodologias de design proporcionou uma base sólida para o desenvolvimento. Além disso, foram definidos os principais requisitos, mecânicas e elementos de jogabilidade. O fluxo de telas e a organização do desenvolvimento também foram planejados, garantindo um direcionamento claro do jogo.

6.1 Pesquisa e definição do tema

Neste primeiro estágio do trabalho, foi possível estabelecer uma base sólida para o desenvolvimento do jogo. Foram definidos os objetivos do projeto, a Engine de desenvolvimento, as mecânicas e regras do jogo, além de uma estrutura metodológica baseada em engenharia de software, Scrum e MDA. A documentação elaborada até o momento servirá como um guia para a implementação no TCC 2, garantindo que o desenvolvimento siga um planejamento estruturado e eficiente.

Durante o TCC 1, foram realizadas diversas etapas essenciais para a estruturação do projeto e o planejamento do desenvolvimento do jogo. A primeira fase consistiu em uma pesquisa aprofundada sobre Engines de desenvolvimento, avaliando opções como Unity, Unreal Engine e Godot. Testei algumas dessas Engines para entender suas capacidades, limitações e adequação ao escopo do projeto. Apesar de não me aprofundar inicialmente em uma, optei por escolher a Godot pela necessidade de um ambiente acessível, eficiente para jogos 2D e alinhado com os princípios de código aberto, garantindo maior flexibilidade e sem custos adicionais.

Além da escolha da Engine, explorei frameworks de gamificação para compreender como os elementos de design poderiam ser utilizados para engajar os jogadores. O framework Octalysis, criado por Yu-kai Chou, foi um dos principais referenciais para estruturar os aspectos motivacionais do jogo. A pesquisa ajudou a definir quais motivações e mecânicas seriam mais relevantes para criar uma experiência dinâmica e envolvente.

Outro conceito fundamental descoberto durante a leitura de artigos foi o framework MDA (Mecânica, Dinâmica e Estética), que se mostrou essencial para estruturar o design do jogo. Esse modelo permitiu uma abordagem mais sistemática na construção da jogabilidade, separando os elementos do jogo em três níveis distintos: as **mecânicas**, que englobam as regras e interações básicas; as **dinâmicas**, que emergem dessas regras e moldam o comportamento do jogador; e a **estética**, que representa as sensações e emoções

desejadas.

A aplicação do MDA no planejamento do jogo ajudou a antecipar como determinadas mecânicas impactam a experiência dos jogadores. Por exemplo, ao definir o sistema de movimentação com curvas e os poderes especiais, analisei como essas mecânicas afetam a dinâmica do jogo e quais emoções poderiam ser despertadas, como tensão, estratégia e senso de imprevisibilidade. Esse modelo se tornou uma ferramenta essencial para guiar as decisões de design, garantindo um alinhamento entre os objetivos do projeto e a experiência proporcionada ao jogador.

6.2 Desenvolvimento do jogo

Neste capítulo falaremos sobre as etapas de desenvolvimento do jogo, apresentando o fluxo de telas, desde o menu inicial até a tela de fim da partida. Descreveremos nosso primeiro contato com a Engine Godot, os principais desafios de criação de *Scenes* e *Nodes*, e como organizamos o trabalho. Vamos destacar os pontos críticos de implementação (movimentação, colisão, sistema de poderes e pontuação) e as soluções adotadas, ilustrando cada fase com trechos de código e capturas de tela.

6.2.1 Preparação do ambiente e planejamento do desenvolvimento

A configuração inicial do ambiente de desenvolvimento começou com a instalação da Engine Godot 4.4 por meio da plataforma Steam [1]., que facilitou o processo de download e atualização. Em seguida, foi criado um repositório no GitHub, clonado localmente para permitir o início imediato do projeto na raiz do versionamento.

As primeiras configurações incluíram o ajuste da resolução de tela, ativação de ferramentas de *debug* e o mapeamento de entradas no *Input Map* para controlar as ações do jogador. Esses ajustes foram essenciais para garantir um ambiente funcional e preparado para as etapas seguintes. O próprio Godot fornece ferramentas de *debug* como painel de saída (*Output*), breakpoints, o monitor de desempenho em tempo real e a exibição de variáveis no *Inspector*, permitindo acompanhar o uso de memória, FPS, colisões e mensagens personalizadas via comandos `print`. Esses recursos foram utilizados principalmente para rastrear o comportamento dos objetos em cena, identificar colisões inesperadas e ajustar a lógica dos poderes e do sistema de movimentação durante os testes iniciais.

A organização do trabalho foi orientada pelo uso de Metodologias ágeis, conforme descrito no Capítulo 2, com foco em sprints quinzenais e no uso contínuo do Trello. Essa abordagem permitiu visualizar metas específicas a cada ciclo, monitorar o progresso e reorganizar prioridades sempre que necessário. Ainda que alguns atrasos tenham ocorrido

[1] É uma plataforma digital de jogos para PC, desenvolvida pela Valve Corporation, que permite aos usuários comprar, baixar, jogar e gerenciar seus jogos.

devido a imprevistos técnicos ou demandas externas, a estrutura do Scrum possibilitou replanejamentos sem comprometer o andamento geral do projeto.

Em diversos momentos, retornar ao quadro do Trello foi fundamental para reorganizar ideias, como identificar o próximo passo e retomar o ritmo de desenvolvimento. Essa prática ajudou a contornar bloqueios, dúvidas sobre prioridade e fases de menor produtividade, tornando o processo mais fluido e direcionado.

Para manter a constância no desenvolvimento, optou-se por uma aplicação adaptada do Scrum. Entre os *ritos* adotados, destacam-se o planejamento inicial de cada sprint e uma revisão informal ao final. As reuniões diárias foram dispensadas, substituídas por acompanhamento contínuo via Trello, o que se mostrou eficaz dentro do contexto individual. Essa flexibilidade também permitiu aproveitar momentos de maior dedicação ou clareza nas tarefas para antecipar funcionalidades futuras, otimizando o uso do tempo disponível.

Na primeira retrospectiva da sprint, notei um atraso causado pelo tempo necessário para se familiarizar com a plataforma Godot. Apesar de conhecimento teórico, a prática exigiu ajustes constantes, principalmente na construção da interface inicial. No início surge dúvidas constantes sobre a melhor abordagem para determinado objetivo, oque levou a revisões muito frequentes do código e nas estruturas dos nós até acertar. Esse feedback individual me ajudou a tomar uma decisão de não subestimar a primeira atividade de entrega e estender seu prazo.

6.2.2 Organização de arquivos e estrutura de cenas

Para manter a organização e padronização do projeto, foi adotado um modelo de nomenclatura no qual arquivos e pastas seguem o padrão *snake_case*, enquanto os nós (*nodes*) definidos no editor utilizam o formato *CamelCase*. Cada cena foi estruturada em uma pasta própria, contendo obrigatoriamente dois arquivos com o mesmo nome-base: o arquivo de cena `.tscn` e o respectivo script `.gd`. Por exemplo:

`main_menu/main_menu.tscn` e `main_menu/main_menu.gd`

Essa estrutura facilita a manutenção do projeto, a navegação entre arquivos e a identificação de dependências diretas entre lógica e visual.

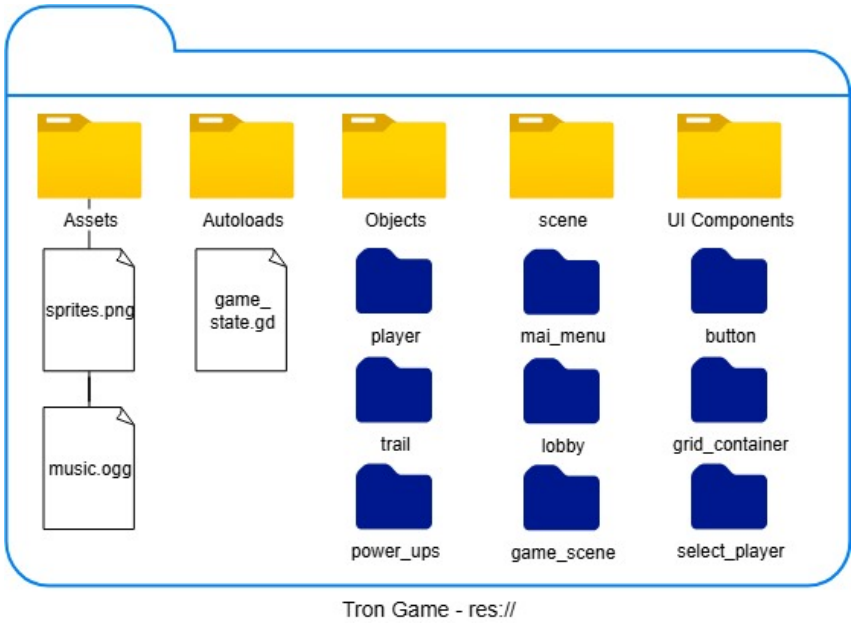
Internamente, a separação entre a lógica de jogo e a interface foi conduzida de forma clara. As cenas responsáveis pelo *gameplay* adotam `Node2D` como nó raiz, com filhos como `Sprite2D` e `CollisionShape2D`, entre outros. Já as cenas de interface, como menus e HUDs, são construídas sobre nós do tipo `Control`, organizadas em camadas distintas por meio de `CanvasLayer`, e compostas por elementos reutilizáveis, como botões, painéis e ícones.

As funcionalidades relacionadas ao jogador e aos *power-ups* foram encapsuladas em cenas independentes, cada uma com seu próprio script, seguindo um modelo baseado em componentes. Essa abordagem permite isolar responsabilidades, como movimentação, detecção de colisão e controle de tempo de efeitos, promovendo maior modularidade e reuso de código.

Por fim, a persistência de estado, como pontuação e configurações dos jogadores, bem como a música de fundo, são gerenciadas por meio do *autoload* `GameManager`. Esse script atua como um controlador global, acessível por todas as cenas, facilitando a manutenção de dados entre transições e o gerenciamento de comportamentos que devem persistir durante toda a execução do jogo.

A Figura 8 exemplifica melhor como o projeto foi organizado em pastas. As pastas azuis possuem arquivos de cena e script relativos ao nome da pasta.

Figura 8 – Diagrama de pacotes de arquivos



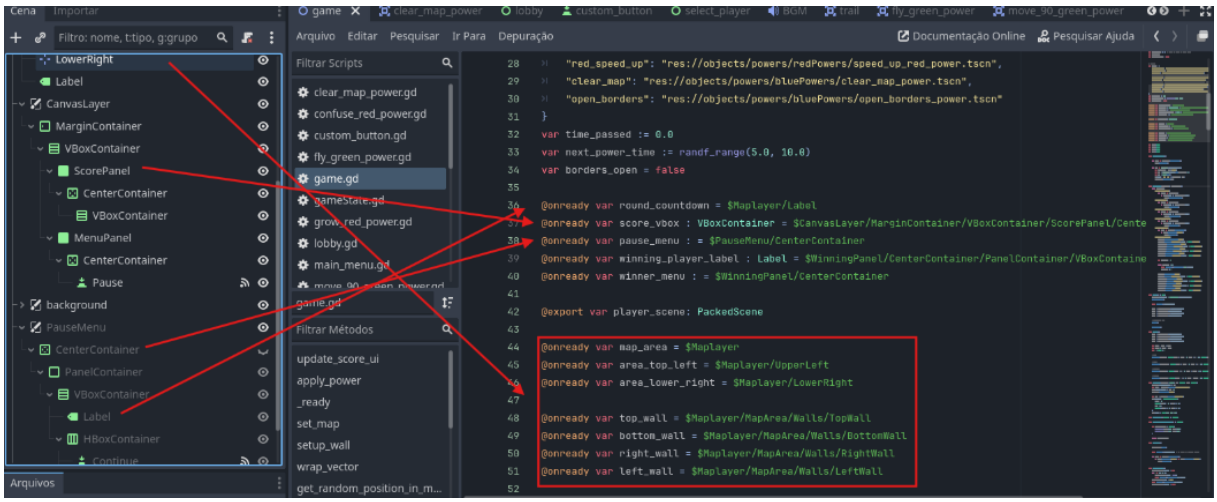
Fonte: Elaboração própria

6.2.3 Interface e usabilidade

No projeto, a interface e a usabilidade foram estruturadas em três telas principais: o menu inicial, a seleção de jogadores (lobby) e a `GameScene`. A Engine Godot oferece um sistema de construção de interfaces com baixa dependência de código (*low-code*), permitindo o ajuste de tamanhos, textos e posicionamento de `Containers` diretamente no editor 2D. A integração desses nós a scripts possibilita a conexão de sinais e a atribuição de métodos aos componentes interativos, proporcionando controle total sobre o comportamento da interface.

Na Figura 9, exemplifica como referenciar os nós (*nodes*) dentro do script. é comum usarmos o termo "@onready"seguido da variável que vai armazenar o nó para acessar e alterar suas propriedades antes e depois da cena carregar.

Figura 9 – Conexão de nós com scripts no editor Godot.



Fonte: Elaboração própria

Nas telas de menu, foram utilizados exclusivamente nós do tipo `Control`, explorando recursos como *Anchors* e *Size Flags* para garantir o alinhamento e o dimensionamento automáticos de painéis, labels e botões em diferentes resoluções. Na `GameScene`, a raiz da cena é um `Node2D`, o que facilita a renderização e o posicionamento de elementos gráficos em 2D, mantendo a interface separada em camadas por meio do uso de `CanvasLayer`, evitando interferência entre a lógica da interface e a lógica de jogo.

Com o objetivo de aprimorar a experiência do usuário, duas funcionalidades específicas foram implementadas. A primeira consiste na personalização dos comandos de movimento (*keybinds*) no lobby, com armazenamento das configurações no singleton `GameManager`, permitindo a persistência das escolhas entre as rodadas. A segunda funcionalidade é a exibição de uma seta direcional sobre o veículo de cada jogador no início de cada rodada, indicando sua orientação inicial e contribuindo para uma melhor percepção espacial logo nos primeiros movimentos. Tais melhorias foram planejadas para oferecer maior clareza visual e controle aos participantes durante a partida.

Além disso, sons foram adicionados para reforçar a imersão do jogador e dar feedbacks importantes durante o jogo. A música de fundo é gerenciada por meio de um `AudioStreamPlayer` inserido em um nó global no *AutoLoad*, configurado para reprodução contínua em loop entre as cenas. Já os efeitos sonoros específicos, como o som de morte do jogador, foram implementados diretamente em suas respectivas cenas (Player.tscn). Para isso, utilizou-se o nó `AudioStreamPlayer2D`, permitindo que cada jogador reproduza sons de forma independente e com espacialização adequada, sem comprometer a

lógica principal do jogo.

6.2.4 Implementação das mecânicas do jogo

O desenvolvimento foi feito em GDScript, a linguagem nativa da Godot. Ela é baseada em Python e é fortemente integrada à arquitetura da Engine. Três métodos se destacam na construção da lógica do jogo: `ready`, chamado uma vez assim que o nó entra na árvore de cena; `physics_process(delta)`, executado a cada frame de física e utilizado para atualizar elementos em tempo real; e a anotação `@onready`, que permite inicializar variáveis com nós da cena somente após estarem carregados. Esses recursos estruturam o ciclo de vida dos objetos e organizam a execução do jogo em etapas previsíveis. Além disso, a Engine Godot adota uma estrutura orientada a nós, onde tudo é tratado como um objeto independente na hierarquia da cena. Qualquer referência a esse objeto, quando alterada, reflete automaticamente em todas as partes do código onde ele está instanciado ou referenciado, o que facilita a reutilização, modularização e manutenção do projeto.

O `ready` carrega a cena inicial. Define as dimensões do mapa e chamado a função de `start_round` que dá início ao jogo.

A função `start_round` é chamada no início de cada rodada e tem como responsabilidade configurar o estado inicial do jogo. Nela são instanciados os jogadores, com base em uma cena pré-configurada (`PackedScene`), posicionados nos marcadores definidos no mapa e conectados aos seus respectivos sinais. Além disso, o método define o tempo para início da rodada, ativando um contador regressivo e liberando o movimento dos jogadores apenas após sua conclusão. Esse controle inicial garante que todas as partidas comecem de forma sincronizada e clara para os participantes.

A Figura 10 mostra como os jogadores são instanciados na cena do jogo. Primeiro instânciamos o jogador, alinhamos todos na mesma camada, definimos sua posição incial e ajustamos os sinais de suas funções.

Durante a execução do jogo, a função `physics_process(delta)` é responsável por atualizar o tempo, movimentar os jogadores e controlar os poderes. Cada jogador move-se em linha reta com velocidade constante, ajustando sua rotação com base nas entradas recebidas. A cada ciclo de atualização, o jogo verifica se é hora de instanciar um novo rastro atrás do jogador, formando a trilha que delimita o espaço percorrido. Também é nesse método que o cronômetro de geração de poderes é verificado, e, ao atingir o tempo definido, um novo poder é instanciado em posição aleatória no mapa.

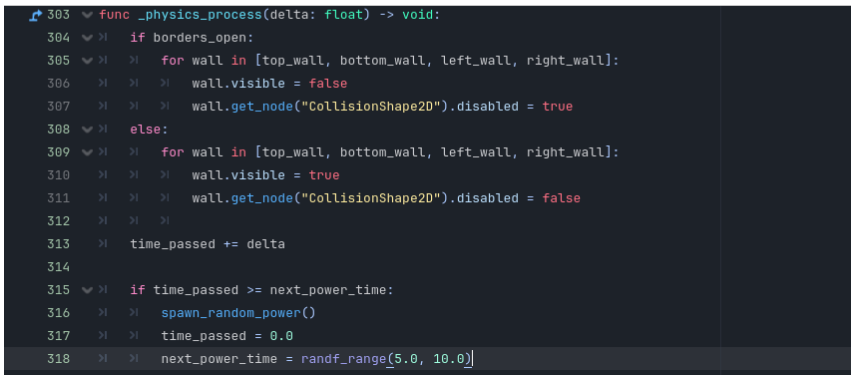
Naa Figuras 11 e 12 tem exemplos da função nativa do Godot *_physics_process* preentes no jogadores e no mapa do jogo. Nessa função é colocado tudo que for verificado ou processado pelos *game ticks*, ou seja, a cada frame do jogo como detecção da momvimentação e colisões.

Figura 10 – código: round_start

```
func start_round():

    await get_tree().create_timer(1).timeout

    is_round_ending = false
    clear_map()
    alive_players.clear()

    for data in GameState.selected_players:
        var player = player_scene.instantiate()
        player.game_scene = self
        player.z_index = 1
        player.global_position = get_random_position_in_map() # devo usar global_position?
        player.animal_data = data
        player_nodes.append(player)
        alive_players.append(player)
        player.connect("died", Callable(self, "_on_player_died"))
        map_area.call_deferred("add_child", player)

    await show_countdown()

    for player in player_nodes:
        player.can_move = true
```

Fonte: Elaboração própria

Figura 11 – gamescene: _physics_process



```
func _physics_process(delta: float) -> void:
    if borders_open:
        for wall in [top_wall, bottom_wall, left_wall, right_wall]:
            wall.visible = false
            wall.get_node("CollisionShape2D").disabled = true
    else:
        for wall in [top_wall, bottom_wall, left_wall, right_wall]:
            wall.visible = true
            wall.get_node("CollisionShape2D").disabled = false

    time_passed += delta

    if time_passed >= next_power_time:
        spawn_random_power()
        time_passed = 0.0
        next_power_time = randf_range(5.0, 10.0)
```

Fonte: Elaboração própria

A detecção de colisões foi feita por meio do nó `Area2D` com `CollisionShape2D`, presente tanto nos jogadores quanto nos rastros e paredes. Cada jogador possui um sinal `area_entered` conectado diretamente à cena `GameScene`, utilizando `Callable(self, "_on_player_died")`. Ao detectar uma colisão com uma área pertencente a outro jogador, a função de morte é acionada, removendo o jogador da cena e atualizando o ranking da rodada. A função `end_round`, chamada após restar apenas um jogador vivo, é responsável por pausar o jogo, distribuir os pontos com base na ordem de eliminação, verificar se algum jogador alcançou a pontuação de vitória e, caso não haja vencedor, iniciar uma nova rodada com `start_round`.

Nas Figuras 13 e 14 são exemplos das funções responsáveis por detectar a morte e a colisão do jogador. Para detectar a morte de um jogador, a função *_on_player_died*

Figura 12 – player: _physics_process

```gdscript
func _physics_process(delta):
    if not alive or not can_move:
        return

    if can_move and $Node2D/Polygon2D.visible:
        $Node2D/Polygon2D.visible = false

    time_since_spawn += delta

    # curvatura suave
    if movement == "curve":
        if Input.is_key_pressed(key_left):
            rotation -= turn_speed * delta
        elif Input.is_key_pressed(key_right):
            rotation += turn_speed * delta
    elif movement == "90":
        if Input.is_key_pressed(key_left) and not already_turned:
            rotation -= deg_to_rad(90)
            already_turned = true
        elif Input.is_key_pressed(key_right) and not already_turned:
            rotation += deg_to_rad(90)
            already_turned = true
        elif not Input.is_key_pressed(key_left) and not Input.is_key_pressed(key_right):
```

Fonte: Elaboração própria

identifica quem morreu usando uma função de *callback* ligado ao sinal de colisão do jogador e a colisão é identificar verificando o grupo ao qual o objeto colidido pertence, seja rastro ou parede.

Figura 13 – gamescene: _on_player_died

```gdscript
func _on_player_died(player):
    if is_round_ending:
        return

    players_rank.append(player.animal_data["name"])

    alive_players.erase(player)
    player_nodes.erase(player)

    if alive_players.size() == 1 and not is_round_ending:
        is_round_ending = true

        last_player = alive_players[0]
        players_rank.append(last_player.animal_data["name"])

        await get_tree().create_timer(3.0).timeout

        if is_instance_valid(last_player) and not last_player.is_queued_for_deletion():
            last_player.queue_free()
            alive_players.erase(last_player)
            player_nodes.erase(last_player)

        end_round()
```

Fonte: Elaboração própria

Para o áudio, foi adotada uma abordagem simples e funcional. A música de fundo é gerenciada por um nó `AudioStreamPlayer`, carregado por meio de um script no `AutoLoad`, o que permite sua reprodução contínua em loop durante todo o jogo, independentemente da cena ativa. Já os efeitos sonoros, como o som de morte, foram incorporados diretamente

Figura 14 – player: _on_area_entered

```
func _on_area_entered(area):
    if not alive:
        return

    if area.is_in_group("power"):
        var power_name = area.get_scene_file_path().get_file().get_basename()
        apply_power(power_name)
        area.queue_free()

    if not flying:
        if area.is_in_group("trail") or  area.is_in_group("wall"):
            print("Colidiu com: ", area.name)
            alive = false
            SFX_Die.play()
            emit_signal("died", self)
            await SFX_Die.finished
            queue_free()
```

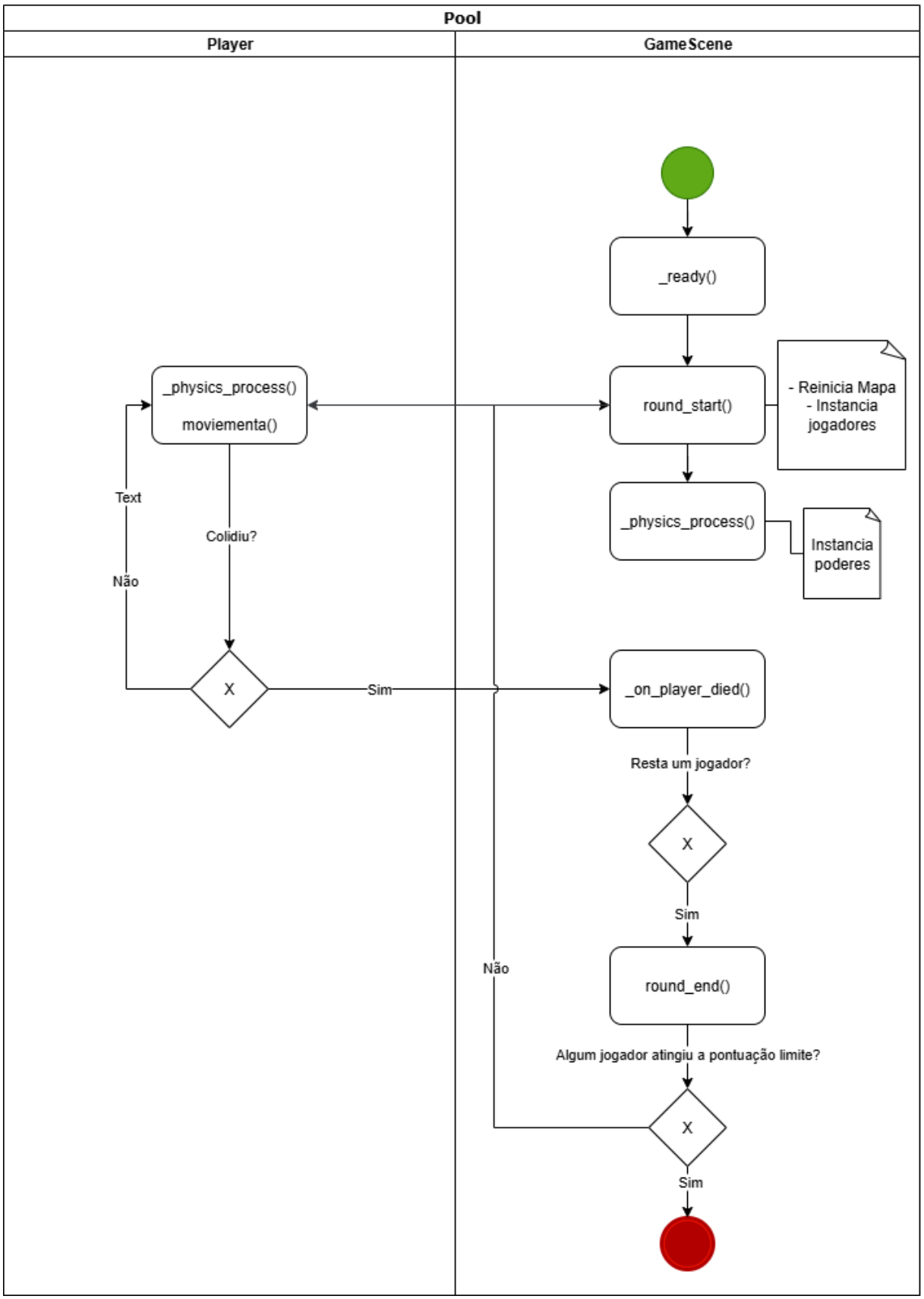
Fonte: Elaboração própria

à cena de cada jogador, utilizando o nó `AudioStreamPlayer2D`. Essa escolha permite que cada instância de jogador reproduza seus próprios sons de forma independente, garantindo uma espacialização adequada e mantendo o controle de áudio encapsulado no próprio objeto responsável pela ação.

Abaixo tem a Figura 15 do fluxo lógico simplificado que ocorre na cena de jogo.

A Tabela 8 contem o resumo das principais soluções de implementação das funcionalidades do jogo.

Figura 15 – BPMN: fluxo lógico



Fonte: Elaboração própria

Tabela 8 – Tabela de solução de implementação das funcionalidades

Funcionalidade	**Principais nós**	**Descrição resumida da implementação**
Movimentação	Player (Area2D), Polygon2D, CollisionShape2D e AudioStreamPlayer2D	Usa `Input.is_key_pressed` para ajustar a direção dentro do `physics_process` e possui velocidade variável que determina curvas mais fechadas ou abertas.
Rastros	Trail (Node2D), Polygon2D, CollisionShape2D	A cada frame um ponto é adicionado no mapa com a mesma proporção do Player. Esses pontos, em sequência, criam uma linha.
Poderes Especiais	PowerUp (Node2D), Sprite2D, CollisionShape2D	Quando o jogador colide com um PowerUp, altera atributos do jogador (ex.: velocidade, invulnerabilidade) ou das paredes do mapa dentro do `GameManager`.
Colisão e eliminação	Embutido nos rastros, jogadores e paredes, signal `body_entered` através do CollisionShape2D	Conecta o sinal de eliminação ao método que verifica se o jogador tocou parede ou rastro alheio; em caso positivo, notifica o `GameManager` e o próprio objeto remove sua instância do jogo.
Sistema de pontos	Singleton `GameManager`, Label de HUD	Mantém dicionário de pontuações; ao final de cada rodada calcula ordem de eliminação, atualiza as `labels` e declara fim da partida.
Tamanho do mapa	Dois `Marker2D` e quatro `Walls` (Area2D)	Os marcadores delimitam as bordas do mapa e as paredes, com as coordenadas dos marcadores, são instanciadas na cena.

Fonte: Elaboração própria.

6.2.5 Fluxo de telas e linha de desenvolvimento

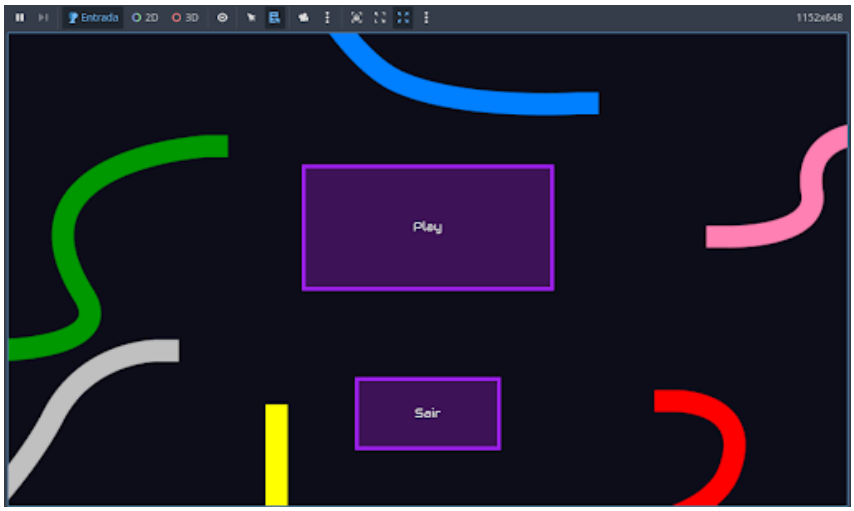
O fluxo de telas do jogo consiste em três cenas: menu principal, lobby e jogo. Abaixo é possível visualizar o resultado final e a sequência de tela do jogo.

A tela inicial possui um *background* com estética semelhante ao jogo, e o usuário pode optar por sair do jogo ou jogar através de um clique. As próximas Figuras 16, 17, 18 e 19, mostra telas do jogo desde o menu até a tela em jogo.

Na tela de Lobby tem a opção de jogar com até 6 jogadores e cada jogador pode customizar suas teclas de comando. É possível remover um jogadores caso necessário e continuar para o jogo.

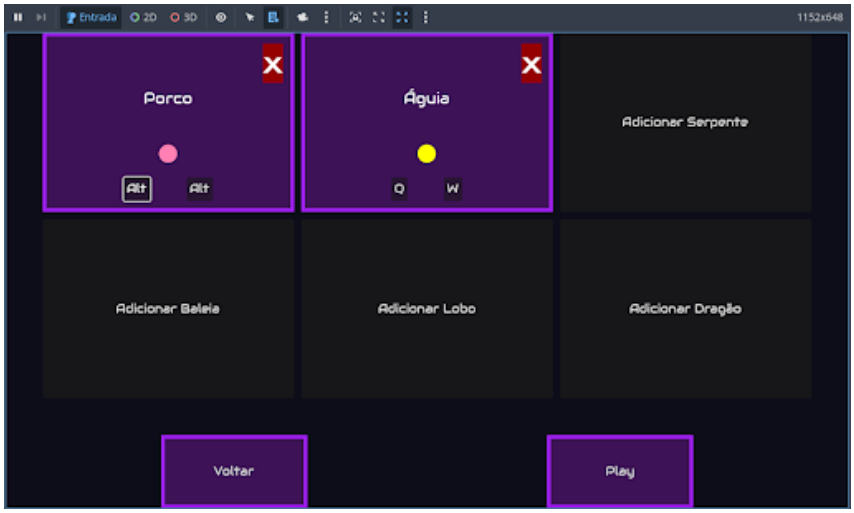
Na cena do jogo, temos um painel com a pontuação dos jogadores, um botão de pause, um contador que declara início da rodada e três jogadores dispostos no mapa com

Figura 16 – Tela menu inicial.



Fonte: Elaboração própria

Figura 17 – Tela do Lobby do jogo.



Fonte: Elaboração própria

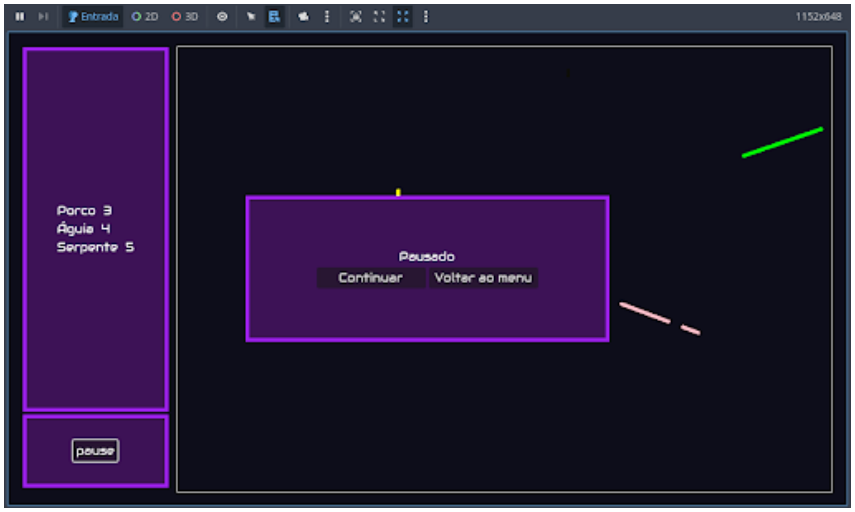
indicadores de suas atuais direções.

Figura 18 – Tela do jogo com indicadores de direção antes de começar a rodada.

Fonte: Elaboração própria

O jogo pode ser pausado a qualquer momento oferecendo as opções de voltar ou continuar a partida.

Figura 19 – Tela do jogo em pause.



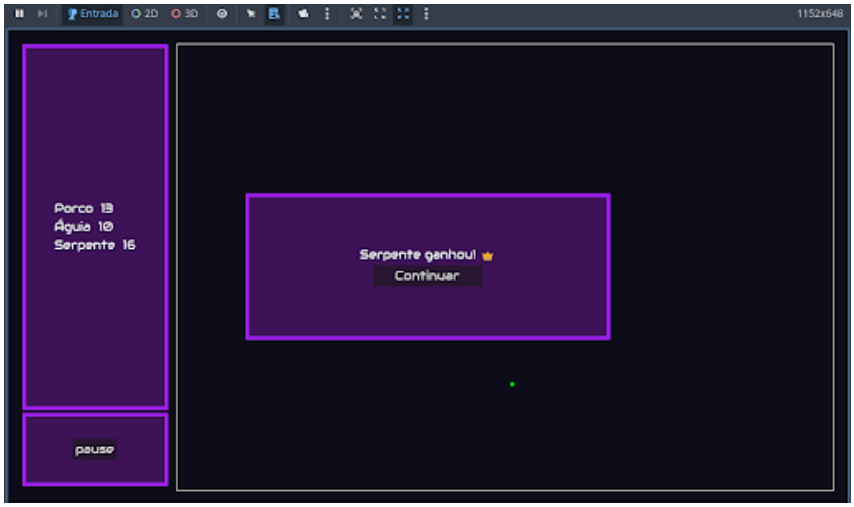
Fonte: Elaboração própria.

Quando um jogador alcança a pontuação mínima, o jogo encerra declarando o vencedor seguido de um botão de continuar.

6.2.6 Principais desafios e soluções

Durante o desenvolvimento do jogo, diversos desafios técnicos e conceituais surgiram, especialmente por se tratar da primeira experiência prática com uma Engine de

Figura 20 – Tela do jogo encerrado.



Fonte: Elaboração própria.

jogos. O primeiro obstáculo foi compreender o funcionamento da Godot Engine e explorar suas funcionalidades. Para isso, foi essencial adotar uma abordagem de aprendizagem baseada em experimentação contínua, consulta à documentação oficial, fóruns, vídeos e o uso de inteligência artificial para suporte na resolução de problemas. A própria Engine facilita esse processo ao oferecer uma documentação embutida que permite explorar métodos e propriedades diretamente nos nós utilizados.

Um segundo desafio relevante foi a construção da interface. A tentativa de criar telas responsivas exigiu o entendimento aprofundado sobre propriedades como ancoragem, margens, alinhamentos e retângulos (`Rect`). Além disso, foi necessário aprender a combinar diferentes tipos de `Container` para estruturar layouts adaptáveis, utilizando ferramentas como `HBoxContainer`, `VBoxContainer` e `MarginContainer`.

A elaboração da lógica principal do jogo representou outro desafio. Por ser baseada em uma estrutura de nós hierárquicos, o desenvolvedor precisa entender a relação entre os elementos, como a importância de posicionar corretamente objetos filhos em relação ao pai, além de saber gerenciar sua comunicação por meio de sinais e grupos. No início, quando ainda não há uma versão testável do jogo, muitos conceitos permanecem meio que abstratos, por isso é comum implementar diversas funcionalidades sem conseguir visualizar seu comportamento final. Isso exige um processo constante de revisão e adaptação, à medida que o restante da estrutura vai sendo integrada e testada.

Entre os problemas práticos mais complexos esteve a criação do rastro deixado pelos jogadores em movimento, principal elemento do jogo. A primeira abordagem testada foi o uso de um `Line2D`, que conectava os pontos por onde o jogador passava. Inicialmente era funcional. Porém, essa solução apresentou falhas ao lidar com teletransporte. Ao atravessar a borda do mapa, um traço era desenhado ligando pontos opostos da tela.

A solução adotada foi instanciar, em intervalos curtos, múltiplos rastros independentes com aparência igual à do jogador, que juntos formam uma linha visualmente contínua.

Outro problema significativo ainda relacionado ao rastro foi a colisão imediata do jogador com o próprio traço recém-gerado, o que tornava a partida impossível de continuar. Três alternativas foram testadas: aplicar um pequeno atraso antes de instanciar o rastro, adicionar uma distância mínima entre o jogador e o ponto de criação do rastro, e finalmente, a abordagem escolhida: aguardar que o jogador deixe a área de colisão do rastro antes de adicioná-lo ao grupo responsável por registrar colisões (grupo `"trail"`). Essa última estratégia demonstrou-se mais confiável e garantiu a jogabilidade esperada, mas ainda com problemas.

Após a implementação de todos os poderes, aqueles que alteravam o tamanho dos jogadores ainda apresentavam comportamentos inconsistentes, sem uma solução satisfatória no momento. Diante disso, optou-se por remover temporariamente esses poderes do jogo, até que uma abordagem mais estável e compatível com o restante da lógica fosse encontrada.

Ao longo do desenvolvimento, diversos outros problemas surgiram, de escala menor ou semelhante, mas ainda assim relevantes para o processo de aprendizagem. Alguns foram resolvidos rapidamente, enquanto outros exigiram maior abstração e análise do comportamento da Engine. Entre os exemplos recorrentes estavam situações como instanciar objetos como filhos de outros e perceber que isso altera sua `global_position`, ou tentar interagir com nós que já haviam sido removidos. Com o tempo, esses padrões de erro se tornam mais reconhecíveis, e a experiência adquirida passa a desempenhar um papel fundamental na antecipação de problemas e na escolha de soluções mais eficientes. Esses aprendizados acumulados, ainda que muitas vezes não documentados diretamente, contribuem significativamente para a formação prática do desenvolvedor.

6.2.7 Testes de Funcionalidade e Desempenho

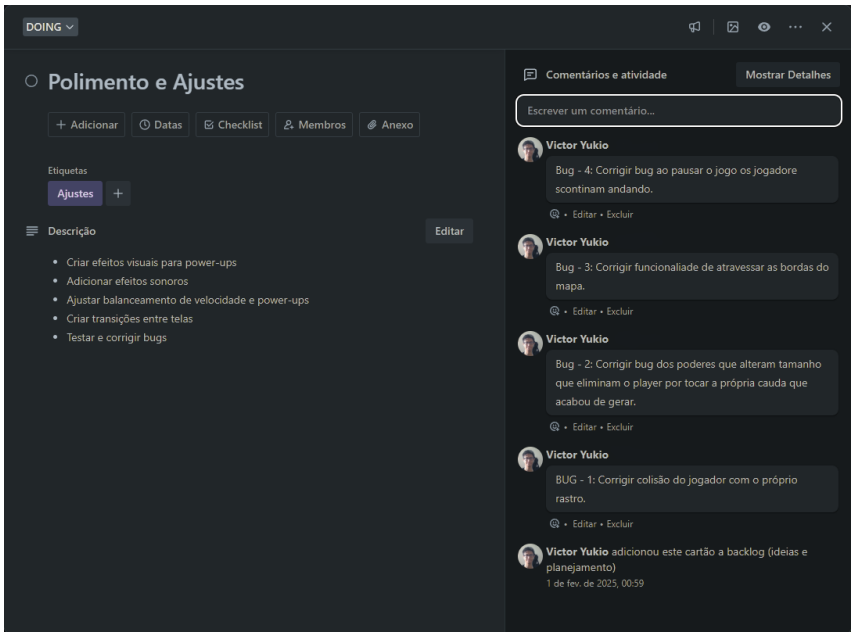
Os testes realizados no projeto seguiram uma abordagem prática e limitada, considerando os recursos disponíveis. A metodologia adotada foi a de testes manual de caixa-preta, com foco na verificação do comportamento esperado das funcionalidades implementadas, sem considerar a estrutura interna do código. Essa abordagem é comum em projetos de software com restrições de tempo ou equipe reduzida, como é o caso deste trabalho (PINHEIRO, 2024) (Check Point Software Technologies, s.d.).

Os testes foram realizados pelo próprio autor, de forma exploratória, em um ambiente local, utilizando um computador pessoal com sistema Windows 11, AMD Ryzen 5 3600 6-Core Processor, 16 GB memória RAM, placa de vídeo AMD Radeon RX 5600 XT.

Durante o processo de testes, foram identificados bugs e comportamentos inesperados, como colisões incorretas, travamentos de movimentos ou falhas em efeitos visuais. Esses problemas eram registrados na plataforma Trello, dentro da seção de tarefas “Melhorias”, para posterior análise e correção. Esse processo, ainda que informal, foi essencial para garantir a estabilidade mínima do sistema e evitar que erros simples passassem despercebidos até a entrega final.

A Figura 21 mostra o quadro de bugs encontrados, para poder haver rastreamento e identificação de bugs e comportamento inesperados do jogo.

Figura 21 – Quadro - Polimento e Ajustes



Fonte: Elaboração própria

Além da detecção de falhas, os testes também desempenharam um papel fundamental no balanceamento das mecânicas do jogo. Durante as sessões de teste, foi possível observar, por exemplo, se a velocidade base dos jogadores proporcionava uma movimentação justa, se o tempo de duração dos poderes estava adequado para não desestabilizar as partidas e se os controles respondiam de forma fluida. Pequenas observações sobre o comportamento em jogo que contribuindo para a melhoria da experiência geral e diversão dos jogadores.

Apesar das limitações, o processo de testes contribuiu para a melhoria contínua do jogo, permitindo a identificação e correção de falhas ao longo do desenvolvimento. A prática de documentar os problemas e tratá-los como parte do ciclo de produção reforçou o valor da organização e da iteração constante no desenvolvimento de software.

7 Conclusões Gerais

O trabalho teve como proposta o desenvolvimento de um jogo digital inspirado nos *Light Cycles* do universo Tron, servindo simultaneamente como exercício prático de aplicação de conceitos da Engenharia de Software e como objeto de estudo sobre o processo de criação de jogos digitais. A condução do projeto foi estruturada por meio de Metodologias Ágeis, com destaque para o uso adaptado do framework *Scrum*, que possibilitou organizar as etapas do desenvolvimento de forma iterativa e flexível, respeitando os limites temporais e o escopo previamente definido.

A utilização do modelo MDA (*Mechanics–Dynamics–Aesthetics*) na primeira etapa do trabalho contribuiu para orientar as decisões de design de forma estruturada, promovendo o alinhamento entre os componentes técnicos do jogo e a experiência de jogo pretendida. As mecânicas implementadas, como a geração de rastro, colisões e poderes temporários, foram integradas com foco em proporcionar dinâmicas de disputa rápida e interação estratégica entre jogadores. Ainda que alguns ajustes tenham sido necessários ao longo do desenvolvimento, a estrutura teórica do MDA mostrou-se útil para manter a coerência entre intenção e implementação.

Durante a fase prática, a escolha da Godot Engine atendeu bem às necessidades do projeto, especialmente por seu suporte nativo ao desenvolvimento 2D, sua estrutura baseada em nós hierárquicos e pela facilidade da linguagem `GDScript`. No entanto, nem sempre os comportamentos da Engine foram intuitivos, o que demandou muitos testes e adaptações na lógica do jogo. Ainda assim, a experiência geral com a ferramenta foi positiva, e a curva de aprendizagem, inicialmente lenta, tornou-se significativamente mais fluida a partir do meio do projeto.

A prática do desenvolvimento proporcionou aprendizado técnico relevante, como o uso de instanciamento dinâmico, gerenciamento de colisões com `Area2D`, manipulação de grupos e sinais para controle de comportamento, construção de HUD com `CanvasLayer` e organização modular da lógica de jogo. Além disso, a necessidade constante de testar, depurar e refatorar consolidou competências relacionadas à modelagem, resolução de problemas e tomada de decisão em cenários práticos.

Considerando o escopo proposto, os resultados obtidos demonstram a viabilidade de aplicar fundamentos da Engenharia de Software ao desenvolvimento de jogos digitais com uma abordagem acessível e orientada à aprendizagem. O jogo desenvolvido, mesmo com limitações pontuais, funciona como um protótipo jogável e representa uma base concreta para futuras melhorias ou expansões. Dessa forma, o trabalho cumpre seu papel como objeto de estudo prático e formativo, oferecendo subsídios tanto para o desenvolvi-

mento técnico quanto para a reflexão sobre metodologias aplicadas à criação de jogos.

Por fim, é importante reconhecer que o ciclo de desenvolvimento de um jogo vai além da construção de sua versão jogável. Etapas como empacotamento e exportação multiplataforma, publicação em lojas digitais, monetização, questões de licenciamento e direitos autorais extrapolam o escopo deste trabalho, mas fazem parte do ecossistema do desenvolvimento de jogos e representam oportunidades futuras de aprendizado. No total, foram gastas 291 horas para desenvolver o jogo, marcadas por registros do tempo de uso da ferramenta de controle de sessões.

7.1 Lições aprendidas e recomendações

Durante o desenvolvimento do projeto, algumas lições importantes foram aprendidas, especialmente em relação ao planejamento, à abordagem técnica e ao gerenciamento de riscos. Um dos principais desafios enfrentados ocorreu logo no início, ao subestimar a complexidade da primeira tarefa, que também representava o primeiro contato prático com a Engine e a estrutura do jogo. Essa etapa revelou a importância de reservar mais tempo para a fase inicial de familiarização e experimentação.

Houve acertos estratégicos que contribuíram significativamente para a viabilidade do projeto. A escolha por um jogo 2D e baseado em uma mecânica já conhecida (inspirada nos *Light Cycles* de Tron) foi essencial para evitar a reinvenção de conceitos e permitiu concentrar esforços na adaptação e melhoria das funcionalidades. As soluções simplificadas, aplicadas principalmente às mecânicas e à organização do código, facilitaram o entendimento das estruturas internas da Engine e possibilitaram uma evolução técnica consistente ao longo do projeto.

Os riscos de software é um problema potencial que pode afetar negativamente o cronograma, a qualidade ou o desempenho do projeto (PRESSMAN; MAXIM, 2016) (Project Management Institute, 2017).Ficou evidente que a ausência de um gerenciamento de riscos mais estruturado dificultou a antecipação de imprevistos, como mudanças no escopo e desafios técnicos inesperados. O planejamento inicial, ainda que básico, demonstrou ser um elemento crucial para manter o foco e organizar as entregas de forma coesa.

Durante a preparação para o projeto, foi realizado um curso introdutório de Godot e GDScript por meio da plataforma Udemy, ministrado por Davi Bandeira (BANDEIRA, 2024). O curso abordava conceitos básicos de lógica de programação e construção de jogos simples na Engine. Embora a experiência tenha sido positiva e proporcionado segurança inicial, ao iniciar o desenvolvimento do jogo próprio, surgiu a percepção de que a dependência de tutoriais com soluções prontas limitava a autonomia na resolução de problemas. Acostumado a seguir uma "receita", foi desafiador lidar com decisões técnicas de forma independente e contextualizada.

Esse contraste reforçou uma lição importante: à medida que um desenvolvedor começa a construir um jogo por conta própria, sem se apoiar diretamente em tutoriais passo a passo, ele se depara com obstáculos mais reais, que exigem criatividade, leitura da documentação e adaptação ao funcionamento da Engine. Essa prática estimula o pensamento crítico e, sobretudo, gera confiança para iniciar projetos mais complexos e com mecânicas mais inovadoras.

Portanto, para quem estar começando, recomendo que futuros projetos priorizem um planejamento claro, escolham abordagens viáveis, considerem estratégias de mitigação de riscos desde as primeiras etapas e busquem um equilíbrio entre referências externas e a autonomia criativa no processo de desenvolvimento.

7.2 Melhorias futuras

Com o protótipo funcional concluído, diversas melhorias podem ser exploradas em uma continuidade do projeto. Entre as mais relevantes estão:

- Implementação de uma inteligência artificial simples para permitir partidas solo contra o computador;
- Adição de suporte a partidas remotas entre jogadores por meio de conexão ponto a ponto (*peer-to-peer*);
- Inclusão de recompensas visuais ou sonoras, como efeitos de vitória, conquistas ou feedbacks de desempenho;
- Expansão dos menus com opções de configuração de áudio, controles e parâmetros de jogo;
- Criação de mapas com tamanhos e proporções ajustáveis, permitindo personalização do campo de jogo;
- Desenvolvimento de novos tipos de poderes especiais para ampliar a variedade de estratégias;
- Aprimoramento do sistema de colisão, buscando mais precisão e confiabilidade no registro de impactos;
- Preparação do jogo para publicação, com foco em empacotamento, identidade visual e publicação em plataformas como a Steam;

Essas propostas visam transformar o protótipo em um produto mais robusto, acessível e com maior valor de rejogabilidade, mantendo a essência competitiva e estratégica da experiência original.

Referências

ASEPRITE. *Aseprite documentation*. 2024. Acesso em: 04 Feb. 2025. Disponível em: <https://www.aseprite.org/docs/>. Citado na página 26.

Avell. *Mercado de games no Brasil se destaca no cenário global*. 2025. Acesso em: 11 ago. 2025. Disponível em: <https://avell.com.br/blog/mercado-de-games>. Citado na página 29.

BANDEIRA, D. *Lógica de Programação + Projetos na Godot/GDScript 4.3+*. 2024. <https://www.udemy.com/course/aprenda-godot-e-gdscript-em-7-dias/>. Curso online, Udemy. Citado na página 58.

CARROLL, J. *Using the MDA Framework as an approach to game design*. 2000. Acesso em: 25 Jan. 2025. Disponível em: <https://medium.com/@jenny_carroll/using-the-mda-framework-as-an-approach-to-game-design-9568569cb7d>. Citado na página 20.

Check Point Software Technologies. *What is Black Box Testing?* s.d. Acessado em 14 jul. 2025. Disponível em: <https://www.checkpoint.com/pt/cyber-hub/cyber-security/what-is-penetration-testing/what-is-black-box-testing/>. Citado na página 55.

CHOU, Y.-K. *The Octalysis Framework for gamification & behavioral design*. 2019. Acesso em: 14 Nov. 2024. Disponível em: <https://yukaichou.com/gamification-examples/octalysis-complete-gamification-framework/>. Citado na página 39.

DINIZ RODRIGO GAVIOLI, F. G. A indústria de jogos eletrônicos no brasil: uma breve história e suas implicações na atualidade. *Geoingá: Revista do Programa de Pós-Graduação em Geografia*, Universidade Estadual de Maringá, 2024. Doutorando no Programa de Pós-graduação em Geografia da UEM; Mestre em Tecnologias Ambientais pela UFMS. Citado na página 28.

ENGINE, G. *Introduction to Godot*. 2024. Acesso em: 29 Jan. 2025. Disponível em: <https://docs.godotengine.org/en/stable/getting_started/introduction/introduction_to_godot.html>. Citado na página 26.

ENGINE, G. *System requirements*. 2024. Acesso em: 01 Feb. 2025. Disponível em: <https://docs.godotengine.org/en/stable/about/system_requirements.html>. Citado na página 31.

Game Developer Staff. User research: Utilizing octalysis in game user research. *Game Developer*, 2021. Acesso em 5 de agosto de 2025. Disponível em: <https://www.gamedeveloper.com/business/user-research-utilizing-octalysis-in-game-user-research>. Citado na página 39.

GLAIEL, T. *How to make your own game engine (and why)*. 2021. Acesso em: 02 Feb. 2025. Disponível em: <https://medium.com/geekculture/how-to-make-your-own-game-engine-and-why-ddf0acbc5f3>. Citado na página 34.

HUIZINGA, J. *Homo Ludens: o jogo como elemento da cultura*. 2000. Acesso em: 02 Feb. 2025. Disponível em: <http://jnsilva.ludicum.org/Huizinga_HomoLudens.pdf>. Citado na página 14.

HUNICKE, R.; LEBLANC, M.; ZUBEK, R. *MDA: A formal approach to game design and game research*. 2004. Acesso em: 26 Jan. 2025. Disponível em: <https://users.cs.northwestern.edu/~hunicke/MDA.pdf>. Citado na página 19.

KENT, S. L. *The ultimate history of video games*. Three Rivers Press, 2001. Acesso em: 21 Dec. 2024. Disponível em: <https://archive.org/details/ultimatehistoryo0000kent/page/n627/mode/2up>. Citado na página 28.

LAWRENCE, S. *Tron arcade game 2002*. 2002. Acesso em: 09 Nov. 2025. Disponível em: <https://www.csh.rit.edu/~jerry/arcade/tron/>. Citado na página 14.

MALANKAR, N. *Software engineering in gaming over the years | Game development | Evolution of gaming | @SCALER*. 2023. Acesso em: 04 Feb. 2025. Disponível em: <https://www.youtube.com/watch?v=abcde1234>. Citado na página 30.

PINHEIRO, D. *Entenda a importância dos testes para o sucesso em jogos digitais*. 2024. Acessado em 14 jul. 2025. Disponível em: <https://www.testingcompany.com.br/blog/entenda-a-importancia-dos-testes-para-o-sucesso-em-jogos-digitais>. Citado na página 55.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de Software: uma abordagem profissional*. 8. ed. [S.l.]: AMGH Editora, 2016. Citado na página 58.

Project Management Institute. *Guia PMBOK: Um guia do conhecimento em gerenciamento de projetos*. 6. ed. [S.l.]: Project Management Institute, 2017. Citado na página 58.

SCHWABER, K.; SUTHERLAND, J. *Guia do Scrum: um guia definitivo para o Scrum: as regras do jogo*. 2013. Acesso em: 04 Nov. 2024. Disponível em: <https://scrumguides.org/docs/scrumguide/v1/Scrum-Guide-Portuguese-BR.pdf>. Citado na página 17.

SOMMERVILLE, I. *Engenharia de Software*. 9. ed. São Paulo: Pearson Prentice Hall, 2011. ISBN 9788576057152. Citado 2 vezes nas páginas 22 e 29.

STUDIOS, P. *O que são as Game Engines ou motores de jogos?* 2014. Acesso em: 08 Jan. 2025. Disponível em: <https://pixstudios.com.br/blog/novidades-de-computacao-grafica-e-games/o-que-sao-engine-de-games-ou-motor-de-jogo/index.html>. Citado na página 30.

TECHNOLOGIES, U. *System requirements*. 2024. Acesso em: 01 Feb. 2025. Disponível em: <https://docs.unity3d.com/6000.0/Documentation/Manual/system-requirements.html>. Citado na página 31.

ULLMANN, G. C. et al. *Game engine comparative anatomy*. 2025. Acesso em: 05 Feb. 2025. Disponível em: <https://arxiv.org/pdf/2207.06473>. Citado na página 30.

Apêndices

APÊNDICE A – Apêndice 1 - Protótipos

Este apêndice apresenta os protótipos do jogo feitos no figma. Ele complementa as informações discutidas no trabalho, fornecendo detalhes adicionais sobre a etapa de idealização do produto final.

Figura 22 Protótipo menu inicial

Figura 22 – Menu inicial

Fonte: Elaboração própria

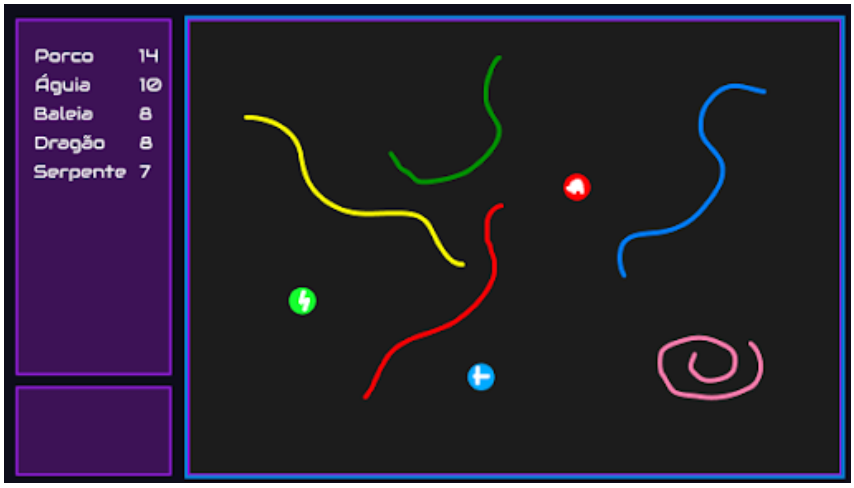
Figura 23 Protótipo Seleção dos jogadores - Sala do jogo

Figura 23 – Sala de iniciação

Fonte: Elaboração própria

Figura 24 Protótipo Cena do Jogo

Figura 24 – Tela do jogo



Fonte: Elaboração própria

Figura 25 Protótipo Fim do Jogo

Figura 25 – Fim do jogo - tela de vitória

Fonte: Elaboração própria

sectionMenu inicial

APÊNDICE B – Apêndice 2 - Imagem dos poderes com ìcones

Durante a prototipação a idealização dos poderes foi feita de forma que tente representar bem seus feitos. A Figura 26 tem o protótipo dos poderes e suas habilidades.

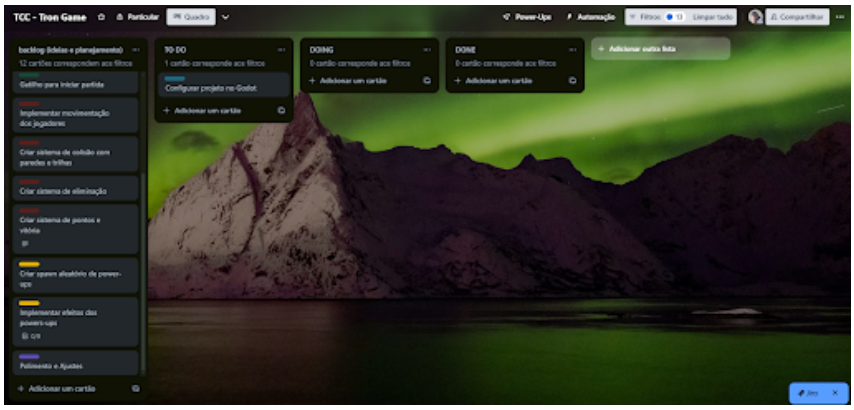
Figura 26 – Ícones dos poderes

Fonte: Elaboração própria

APÊNDICE C – Apêndice 3 - Tarefas do Trello

Figura 27, quadro do trello que foi utilizado para guiar as tarefas ao longo das semanas.

Figura 27 – Lista de tarefas



Fonte: Elaboração própria

Anexos

ANEXO A – Primeiro Anexo - link do repositório

O código-fonte do jogo desenvolvido neste trabalho está disponível no seguinte endereço:

<https://github.com/yukioz/TCC-1_Tron-Game>

ANEXO B – Segundo Anexo - link do protótipo

A visualização do protótipo está disponível no seguinte endereço:

<https://www.figma.com/design/AFDOFtHaKFXwTHDVJa9CMl/Prot%C3%B3tipo---Tro
node-id=0-1&t=wq8JGf6yV9ygoW40-1>

ANEXO C – Terceiro Anexo - Respostas Formulário experiência do jogo

O arquivo com as respostas coletadas está disponível para consulta no link:

<https://docs.google.com/spreadsheets/d/1tXdx6wGHl2u7x3QuYwRbvPLd2llJUAMm4AjZ
edit?resourcekey=&gid=1051398419#gid=1051398419>