Universidade de São Paulo
Instituto de Matemática,
Estatística e Ciência da Computação
Bacharelado em Ciência da Computação

Desenvolvimento de um Jogo Auto Battler com Elementos de Roguelike

Gustavo Ueti Fukunaga

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Ricardo Nakamura

São Paulo
2025

Resumo

Gustavo Ueti Fukunaga. **Desenvolvimento de um Jogo Auto Battler com Elementos de Roguelike**. Monografia (Bacharelado). Instituto de Matemática, Estatística e Ciência da Computação, Universidade de São Paulo, São Paulo, 2025.

O objetivo deste trabalho foi desenvolver um protótipo de um jogo digital 2D que combinasse os elementos estratégicos encontrados em jogos do gênero auto battler com sistemas e progressão inspirados em jogos roguelike. Para alcançar esse objetivo, foram analisados os principais jogos de cada gênero, catalogando as suas principais mecânicas, a partir disso foi montado o design do projeto, descrito no game design document. O desenvolvimento então implementou as mecânicas mais fundamentais para que o projeto alcançasse seu objetivo.

Palavras-chave: Desenvolvimento de jogos digitais. Auto battler. Roguelike. Game Design.

Abstract

Gustavo Ueti Fukunaga. **Development of an Auto Battler Game with Roguelike Elements**. Capstone Project Report (Bachelor). Institute of Mathematics, Statistics, and Computer Science, University of São Paulo, São Paulo, 2025.

The objective of this work was to develop a prototype of a 2D digital game that combines the strategic elements found in games of the auto battler genre with progression systems inspired by roguelike games. To achieve this objective, the main games of each genre were analyzed, and their core mechanics were identified. Based on this analysis, the project design was created and documented in the game design document. The development process then implemented the most fundamental mechanics required for the project to achieve its objective.

Keywords: Digital game development. Auto battler. Roguelike. Game Design.

Lista de figuras

Sumário

Capítulo 1

Introdução

A indústria de jogos digitais representa um dos maiores mercados do setor de entretenimento e mídia, previsto para superar a receita das indústrias do cinema e da música combinadas (PwC, 2025). Nesse cenário, um dos gêneros de jogos digitais popularizado mais recentemente é o *auto battler*.

O gênero *auto battler* foi popularizado em 2019 pelo jogo *Dota Auto Chess*, é caracterizado por batalhas entre jogadores por meio de personagens posicionados em um tabuleiro, os quais os jogadores não têm controle direto sobre eles durante a batalha. Atualmente o gênero é uma aposta de grandes estúdios, que lançaram suas próprias versões seguindo a popularização do gênero, entre eles o *Teamfight Tactics* (desenvolvido pela *Riot Games*), *Dota Underlords* (desenvolvido pela *Valve*) e *Hearthstone's Battlegrounds* (desenvolvido pela *Blizzard Entertainment*) (Wikipedia, 2025a).

Por outro lado, diferente dos jogos citados com grandes investimentos e times de desenvolvimento, alguns títulos recentes como *Slay The Spire*, *Vampire Survivors* e *Balatro*, representam jogos indies produzidos por equipes menores e com menos recursos, mas que obtiveram grande reconhecimento da crítica e vendas. Uma das semelhanças entre eles é o uso de elementos de jogos do gênero roguelike para adicionar aleatoriedade entre cada partida, tornando cada experiência única e aumentando a rejogabilidade do jogo (Wikipedia, 2025e).

Objetivos

Diante desse cenário, o objetivo deste projeto é desenvolver um protótipo de um jogo 2D que une características do gênero auto battler com sistemas de progressão inspirados em jogos do gênero roguelike. O projeto busca integrar os diferentes sistemas encontrados nos dois gêneros de forma modular, com foco no projeto e implementação de uma arquitetura de software que permita a extensibilidade, reutilização de componentes e manutenção do código. Dessa forma, o foco do trabalho é abordar conceitos como design de jogos, arquitetura de sistemas, organização de estados e aplicação de boas práticas de engenharia de software no contexto do desenvolvimento de jogos digitais.

Assim, este trabalho está estruturado da seguinte forma: inicialmente são apresentados os conceitos relacionados a game design, mecânicas de jogo e à engine Godot. Em seguida, são apresentados os principais jogos dos gêneros auto battler e roguelike, destacando as principais características que definem cada um. Depois, é descrito o design escolhido para o protótipo do jogo e o processo de desenvolvimento, detalhando os principais sistemas implementados. Por fim as conclusões do trabalho.

Metodologia

O trabalho se iniciou com o estudo dos principais jogos de cada gênero, levantando quais eram as características fundamentais de cada um. Usando essas informações, foi criado o *Game Design Document* (GDD), o documento que define todas as informações presentes no design do jogo. O GDD serviu como referência para a escolha de quais sistemas e mecânicas seriam implementados no protótipo do jogo, com o objetivo de criar uma experiência mais próxima possível da versão completa, com as funcionalidades mais essenciais.

O protótipo foi desenvolvido utilizando a Godot Engine, com a linguagem GDScript. A arquitetura do sistema foi estruturada de forma modular, separando responsabilidades entre os diferentes componentes do jogo, para isso foram utilizados recursos da Godot como *signals*, *groups* e *resources*, facilitando a comunicação entre os diferentes módulos do sistema e deixando o código mais limpo. Essa organização facilitou a manutenção do código, a adição de novas funcionalidades e a expansão de sistemas presentes no jogo.

A validação do protótipo foi feita por meio de testes manuais durante o desenvolvimento, verificando o funcionamento dos sistemas e a integração entre os componentes do jogo. Esse processo possibilitou ajustes contínuos durante o desenvolvimento e garantiu que os objetivos definidos para o protótipo fossem alcançados.

Parte da arquitetura inicial dos sistemas básicos do protótipo foi desenvolvida a partir de materiais de apoio e tutoriais disponíveis na internet, utilizados para o aprendizado das principais funcionalidades e boas práticas da Godot Engine. A partir dessa base, mais funcionalidades foram adicionadas e adaptações foram necessárias para integrar as funcionalidades específicas do protótipo proposto.

Todas as artes utilizadas no jogo vieram de um pacote disponível gratuitamente na plataforma *itch.io* (https://sethbb.itch.io/32rogues).

Capítulo 2

Conceitos e Tecnologia

2.1 Game design

O *game design* refere-se ao processo de concepção e planejamento de um jogo digital, englobando a definição de suas regras, objetivos, sistemas e interações que estruturam a experiência do jogador. Trata-se de uma atividade interdisciplinar que envolve aspectos técnicos, criativos e analíticos, tendo como principal objetivo a criação de uma experiência coerente, desafiadora e significativa para o jogador. O *game design* pode ser compreendido como a arte de decidir o que um jogo deve ser, considerando simultaneamente a experiência do jogador, as mecânicas envolvidas e as emoções que o jogo busca provocar (Schell, 2023).

O *game design* também envolve a definição do *game loop*, que representa o ciclo principal de ações repetidas pelo jogador durante a partida. A clareza e consistência desse ciclo são fundamentais para garantir que o jogo tenha uma progressão e mantenha o engajamento do jogador ao longo do tempo. Dessa forma, o *game design* não se limita apenas à criação de ideias, mas estabelece uma estrutura lógica que guia a implementação técnica do jogo.

Normalmente, a especificação do design de um jogo é documentada em um *game design document* (GDD), que funciona como um documento de referência ao longo de todo o processo de desenvolvimento. O GDD centraliza todas as informações relacionadas às mecânicas, sistemas, regras, narrativa, interface e fluxo de progressão do jogo, sendo um guia tanto para as decisões de design quanto para a implementação técnica.

2.2 Mecânicas de jogo

As mecânicas de jogo correspondem ao conjunto de regras, ações e sistemas que definem como o jogador interage com o jogo e como o estado do jogo evolui ao longo da partida. Elas determinam quais ações são possíveis, como essas ações afetam o ambiente e quais são as consequências das decisões tomadas pelo jogador. Dessa forma, as mecânicas são a representação das ideias definidas durante o processo de game design.

As mecânicas constituem a “estrutura fundamental do jogo”, sendo responsáveis por moldar diretamente a experiência do jogador por meio das ações permitidas e das regras

que governam suas consequências (Schell, 2023).

No contexto deste trabalho, as mecânicas de jogo estão diretamente relacionadas aos sistemas implementados no protótipo, descritos no capítulo de Desenvolvimento. Cada sistema implementa regras específicas que orientam o comportamento do jogo e estruturam a experiência do jogador.

2.3 Game Engine - Godot

A Godot Engine é uma *game engine* voltada para o desenvolvimento de jogos digitais 2D e 3D, que oferece um conjunto integrado de ferramentas para criação, edição e execução de projetos interativos. Um de seus principais diferenciais é o fato de ser uma plataforma gratuita e *open source*, licenciada sob a *MIT License*, o que permite seu uso, modificação e distribuição sem custos ou restrições comerciais (Godot Engine, 2025). Essa característica torna a Godot especialmente atrativa tanto para desenvolvedores independentes quanto para fins acadêmicos, pois elimina barreiras de acesso e possibilita o estudo e a adaptação de seu código-fonte. Além disso, a engine oferece suporte multiplataforma, um editor visual integrado e uma linguagem própria, o GDScript, facilitando o desenvolvimento rápido de protótipos e a implementação de arquiteturas modulares. A Godot foi a *engine* escolhida para o desenvolvimento deste projeto.

A GDScript é uma linguagem de programação de alto nível, orientada a objetos e de natureza imperativa, desenvolvida especificamente para a Godot Engine. Ela utiliza uma sintaxe baseada em indentação, semelhante à de linguagens como Python, o que contribui para maior legibilidade e facilidade de escrita do código. Além disso, a GDScript adota tipagem gradual, permitindo ao desenvolvedor optar entre tipagem dinâmica ou estática conforme a necessidade. Seu principal objetivo é ser otimizada e profundamente integrada à Godot Engine, oferecendo flexibilidade para a criação de conteúdo e facilitando a integração entre código, cenas e sistemas do jogo (Godot Engine Documentation, 2025). A GDScript foi a linguagem escolhida para o desenvolvimento dos scripts deste projeto.

O projeto foi estruturado usando os conceitos e recursos que a Godot Engine oferece, foram usados principalmente *nodes* e *scenes* para o desenvolvimento, com o auxílio de outras funcionalidades da *engine*. Um *node* representa a menor unidade parte de um sistema, que define um comportamento específico, como uma área para detecção de colisões entre objetos. Um *node* pode ter um script associado a ele para configurar comportamentos personalizados via código.

Uma *scene* corresponde a uma composição de *nodes* que em conjunto representam um sistema ou entidade mais complexa dentro do projeto, uma *scene* também pode conter outras *scenes*. Essa estrutura hierárquica permite a criação de componentes que podem ser reutilizados em diversas partes do projeto, facilitando a organização e manutenções futuras. Desta forma, a arquitetura geral do jogo buscou separar responsabilidades entre diferentes componentes, tentando reduzir o acoplamento entre os módulos e repetição de código.

Além disso, outras funcionalidades auxiliares da Godot foram utilizadas, como *signals*, *groups*, *export variables* e *resources*.

Signals são um mecanismo de comunicação orientado a eventos fornecido pela Godot

Engine. Um signal é a emissão de um evento por um objeto, que pode ser utilizado por outros objetos interessados sem que exista uma dependência entre eles, ou seja, o emissor não precisa saber quem são os receptores do evento. Para o projeto, signals foram utilizados por exemplo para a comunicação entre diferentes *scenes*.

Groups são um recurso da Godot Engine que permite agrupar *nodes* em conjuntos independentemente de sua posição na hierarquia de cenas. No projeto, *groups* foram usados para organizar entidades com comportamentos semelhantes, como unidades ativas em uma batalha. Isso permite a realização de operações coletivas, como aplicação de efeitos para unidades em batalha, sem a necessidade de manter referências explícitas para cada objeto individual.

Export variables são utilizadas para expor parâmetros do código de um *node* diretamente no editor da Godot Engine. Esse recurso facilita a configuração de valores sem a necessidade de modificações diretas do código e também é usado para a comunicação entre nodes de uma mesma scene, criando uma referência direta a um *node* externo que pode ser usado dentro do código.

Resources são estruturas de dados que armazenam informações de forma independente da lógica de execução do jogo. Um *resource* representa um objeto de dados reutilizável, que pode ser compartilhado entre diferentes sistemas sem duplicação de informações. Essa separação entre dados e comportamento do jogo permite ajustes e extensões no conteúdo sem a necessidade de alterações no código-fonte, além de favorecer a reutilização e a consistência das informações utilizadas pelos diferentes sistemas. A configuração de quais dados são representados nas *resources* é definida via *script*.

Capítulo 3

Estudo de jogos relevantes

Esta seção apresenta uma análise dos gêneros *auto battler* e *roguelike*, com o objetivo de identificar suas principais características, mecânicas recorrentes e abordagens de design. O estudo desses gêneros serviu como base conceitual para o desenvolvimento do protótipo, auxiliando na definição dos sistemas, da progressão e das escolhas da arquitetura do projeto.

3.1 Jogos do gênero roguelike

O gênero *roguelike* é caracterizado por experiências de jogo estruturadas em partidas independentes, com forte presença de aleatoriedade, progressão incremental e foco na rejogabilidade. De modo geral, cada partida apresenta variações significativas em relação às anteriores, seja por mudanças nos recursos disponíveis, nos desafios enfrentados ou nas combinações de habilidades oferecidas ao jogador (Wikipedia, 2025e).

Entre as principais características do gênero estão a ausência de progressão linear tradicional, a tomada constante de decisões estratégicas e a presença de sistemas que modificam ou expandem as regras do jogo ao longo da partida. Jogos como *Slay the Spire*, *Balatro* e *Vampire Survivors* exemplificam essas características ao utilizar sistemas de modificadores, cartas, itens ou habilidades que alteram o funcionamento básico da mecânica central do jogo.

Em *Slay the Spire*, por exemplo, a progressão ocorre por meio da construção de um baralho ao longo da partida, onde cada escolha de carta, relíquia ou rota influencia diretamente o desempenho futuro do jogador (Wikipedia, 2025f). Já em *Balatro*, a mecânica base de formação de mãos de pôquer é constantemente modificada por efeitos especiais, criando variações significativas nas estratégias possíveis a cada execução do jogo (Wikipedia, 2025b). Em *Vampire Survivors*, a combinação aleatória de armas e aprimoramentos define o estilo de jogo de cada partida (Wikipedia, 2025h).

Esses exemplos evidenciam que o núcleo do gênero roguelike está menos relacionado a uma temática específica e mais à forma como o jogo promove variação, adaptação e aprendizado contínuo do jogador. Esses conceitos influenciaram diretamente o projeto desenvolvido, especialmente na definição dos sistemas de progressão, itens e modificadores

de atributos.

Figura 3.1: *Tela de* Slay the Spire.

Figura 3.2: *Tela de* Balatro.

3.2 Jogos do gênero auto battler

O gênero *auto battler* é definido por batalhas automáticas nas quais o jogador não participa diretamente durante o combate. O papel do jogador concentra-se na fase de preparação, que envolve a escolha, posicionamento e aprimoramento das unidades que irão participar da batalha (Wikipedia, 2025a).

Jogos como *Dota Auto Chess* (Wikipedia, 2025c), *Teamfight Tactics* (Wikipedia, 2025g) e *Hearthstone Battlegrounds* (Wikipedia, 2025d) consolidaram o gênero ao introduzir sistemas baseados em tabuleiros, lojas de unidades, sinergias entre personagens e progressão por rodadas. Um elemento central desses jogos é a construção de equipes a partir de

unidades com atributos, classes ou tipos distintos, incentivando combinações estratégicas que oferecem bônus específicos.

Outro aspecto relevante do gênero é o uso de progressão econômica e gerenciamento de recursos. O jogador deve decidir como utilizar sua moeda ao longo das rodadas, equilibrando a aquisição de novas unidades, melhorias, atualizações da loja e avanço de nível. Essa tomada de decisão estratégica ocorre sob restrições de tempo e recursos, aumentando a complexidade do jogo.

Além disso, o posicionamento das unidades no tabuleiro exerce influência direta no resultado das batalhas, afetando fatores como alcance de ataque, alvos prioritários e movimentação automática. Dessa forma, mesmo sem controle direto durante o combate, o jogador mantém um papel ativo e estratégico no desempenho de sua equipe.

Figura 3.3: *Tela de* Teamfight Tactics.

Figura 3.4: *Tela de* Hearthstone Battlegrounds.

Capítulo 4

Design do Jogo

A análise dos gêneros roguelike e auto battler possibilitou a identificação das suas principais características que serviram de base para o desenvolvimento do protótipo apresentado neste trabalho. Do gênero auto battler, foram adotados conceitos como batalhas automáticas, posicionamento estratégico em tabuleiro, gerenciamento de equipes e progressão por fases com economia e recursos. Do gênero roguelike, foram incorporados sistemas de progressão baseados em partidas, uso de modificadores que alteram atributos e mecânicas, e incentivo à rejogabilidade por meio de escolhas variáveis a cada execução do jogo.

A combinação desses elementos orientou o design do protótipo, buscando integrar características centrais de ambos os gêneros em uma estrutura modular, coerente e adequada ao escopo do trabalho.

4.1 Game Design Document (GDD)

O game design document deste projeto foi elaborado com o objetivo de definir de forma estruturada a proposta do jogo, seus sistemas centrais, regras e fluxo de progressão. O documento serviu como base conceitual para o desenvolvimento do protótipo, orientando as decisões de design e delimitando o escopo inicial do projeto, em lugar de outros documentos de especificação de requisitos.

A proposta do jogo combina características dos gêneros auto battler e roguelike, inspirando-se em títulos como *Teamfight Tactics* e *Vampire survivors*. O jogador assume o papel de defensor de uma vila, enfrentando sucessivas invasões de inimigos ao longo de uma única execução do jogo (*run*), na qual as decisões tomadas influenciam diretamente o desempenho nas fases subsequentes.

A progressão do jogo foi concebida de forma linear e segmentada. O jogo é dividido em quatro estações do ano, cada uma composta por seis batalhas consecutivas, totalizando vinte e quatro confrontos. A dificuldade das batalhas aumenta gradativamente, acompanhando a evolução do time do jogador.

A sexta batalha de cada estação foi projetada como um confronto especial contra um inimigo do tipo chefe (*boss*), caracterizado por atributos superiores e habilidades

exclusivas. A vitória em todas as batalhas representa a condição principal de conclusão da execução do jogo.

Seguindo a lógica de jogos do gênero roguelike, a derrota em qualquer batalha resulta no encerramento da execução atual, fazendo com que o jogador perca todo o progresso obtido. Os desempenhos das runs do jogador podem ser armazenados por meio de um sistema de leaderboard, incentivando a rejogabilidade.

O jogador possui acesso a um conjunto inicial de dez personagens. Em cada batalha, no máximo seis personagens podem ser posicionados no tabuleiro. Todos os personagens iniciam o jogo com os mesmos atributos base, evitando diferenças iniciais e reforçando a progressão baseada em escolhas feitas ao longo da partida.

Os atributos definidos no GDD incluem vida, dano de ataque, velocidade de ataque, dano de habilidade e alcance de ataque. Esses atributos determinam o comportamento dos personagens durante a batalha e podem ser modificados por meio de melhorias adquiridas ao longo do jogo.

As batalhas ocorrem em um tabuleiro organizado em cinco linhas e oito colunas. Durante a fase de preparação, o jogador pode posicionar seus personagens livremente nas quatro colunas à esquerda do tabuleiro, enquanto as quatro colunas à direita são reservadas aos inimigos, cujo número, tipo e posicionamento variam de acordo com a fase.

Não há limite de tempo para a fase de preparação, permitindo que o jogador planeje suas ações com cuidado. A batalha é iniciada manualmente pelo jogador por meio de um botão.

Durante a fase de combate, os personagens e inimigos agem de forma automática. O comportamento básico das unidades segue um ciclo de movimentação em direção a um adversário, ataque de acordo com a velocidade de ataque e troca de alvo quando o inimigo é derrotado. Melhorias aplicadas aos personagens podem alterar ou expandir esse fluxo de comportamento.

Um personagem é considerado derrotado ao perder toda a sua vida e permanece fora de combate até o final da *run*, sem possibilidade de recuperação.

Ao vencer uma batalha, o jogador é recompensado com pontos de experiência. A quantidade de pontos obtidos é calculada com base em fatores como a vitória na batalha, a diversidade de personagens utilizados e o número de aliados derrotados durante o confronto.

Os pontos de experiência podem ser gastos na loja, que ocorre após batalhas vencidas. A loja apresenta cinco opções de melhorias aleatórias, sendo possível ao jogador gastar pontos adicionais para atualizar as opções disponíveis.

O GDD define diferentes tipos de melhorias, incluindo modificadores individuais simples, melhorias de classe, modificadores globais, melhorias avançadas e bônus de suporte. As melhorias podem alterar atributos, adicionar habilidades especiais ou fornecer efeitos passivos que influenciam o andamento da partida.

Além das melhorias permanentes, o jogo também prevê a existência de itens consumíveis, que podem ser utilizados durante a batalha para gerar efeitos temporários, como cura, bônus de atributos ou ações diretas contra os inimigos.

As melhorias não podem ser revertidas após aplicadas, e não há um limite máximo para a quantidade de modificadores ativos em um personagem ou no time como um todo, incentivando combinações variadas e estratégias emergentes.

4.2 Adaptações para o Protótipo

Durante o desenvolvimento do protótipo, diversas adaptações em relação ao GDD original foram necessárias, principalmente em função das limitações de tempo, escopo e complexidade do projeto. O foco do protótipo esteve na validação da arquitetura do sistema e na integração das mecânicas centrais, em vez da implementação completa de todos os sistemas planejados inicialmente.

Entre as principais adaptações realizadas, destaca-se a simplificação do sistema de progressão. O número de fases e a divisão em estações foram reduzidos para 6, permitindo concentrar esforços na implementação do fluxo básico de preparação, batalha e recompensa. Da mesma forma, o sistema de inimigos foi simplificado, com menor variedade de tipos e ausência de batalhas específicas contra chefes.

O sistema de melhorias também passou por ajustes, sendo representado no protótipo principalmente por itens que modificam atributos das unidades e de classes. Mecânicas mais complexas, como bônus de suporte e modificadores globais não foram implementadas nesta etapa.

Apesar dessas simplificações, os conceitos fundamentais definidos no GDD foram preservados. O protótipo mantém o foco na tomada de decisões estratégicas durante a fase de preparação, na resolução automática das batalhas e na progressão baseada em escolhas feitas ao longo de uma execução única, características centrais dos gêneros auto battler e roguelike.

Essas adaptações permitiram a construção de um protótipo funcional e coerente com a proposta inicial do projeto, servindo como uma prova de conceito da viabilidade do design e da arquitetura planejados no GDD, além de fornecer uma base para possíveis expansões futuras.

Capítulo 5

Desenvolvimento

5.1 Organização do projeto e arquitetura geral

Antes da definição dos sistemas específicos do jogo, foi estabelecida a cena principal do projeto, responsável por centralizar a execução e a comunicação entre os principais componentes. O node raiz dessa cena é a `Arena`, que atua como o ponto de coordenação do jogo. Esse node é responsável por instanciar e organizar os principais nodes da cena, além de realizar a conexão entre os signals emitidos pelos diferentes componentes e as funções responsáveis por tratar esses eventos.

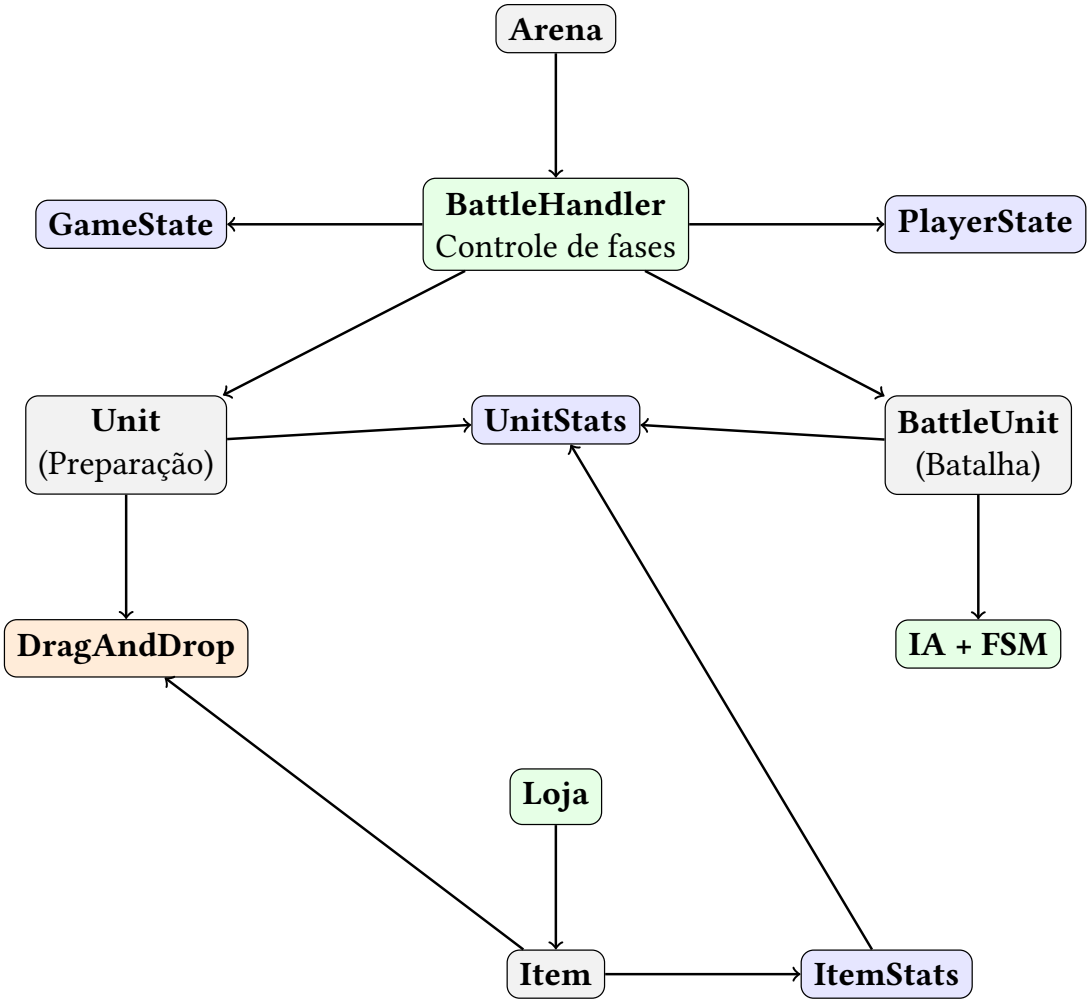

Figura 5.1: *Arquitetura geral simplificada do protótipo.*

5.2 Sistema de posicionamento de personagens

Os personagens são definidos pela scene `Unit`, que possui como nodes uma área de detecção de colisões, utilizada para detectar interações do jogador com o personagem, e o node `DragAndDrop`, que implementa funções para movimentar um target quando ele é segurado usando o mouse. A comunicação entre o node raiz da `Unit` e o `DragAndDrop` é feita por meio de signals e export variables.

Além disso, existe o node `OutlineHighlighter`, responsável por destacar a unidade com uma borda quando o cursor do mouse fica em cima da `Unit` (hover), sendo esse comportamento detectado utilizando signals.

A criação dos nodes `DragAndDrop` e `OutlineHighlighter` possibilitou o reuso deles em outros componentes do jogo. Também facilita a adição das funções de movimentação e destaque em novos componentes do jogo.

Dentro da scene principal `Arena`, os personagens podem ser movidos para a `GameArea`, que é onde acontecem as batalhas, ou para o `Bench`, onde ficam os personagens que não

participam das batalhas. As duas áreas possuem estruturas similares e são definidas por um mesmo tipo de grupo de nodes: (1) o node `UnitGrid`, que define um tabuleiro de Units utilizando um dicionário e funções auxiliares; (2) o node `PlayArea`, que converte a posição global do mouse para uma posição do tabuleiro definido por `UnitGrid`; e (3) o node `TileHighlighter`, que destaca a posição do tabuleiro para onde o usuário irá movimentar um personagem selecionado.

Como os scripts estão associados ao tipo do node e não a uma instância, o mesmo script funciona tanto para os nodes da `GameArea` quanto para os do `Bench` do mesmo tipo. O node responsável por ligar todos esses componentes e verificar se as Units estão sendo movimentadas para posições válidas é o `UnitMover`. Além disso, ele também é responsável por resetar a `Unit` para a posição inicial caso ela seja deixada em uma posição inválida.

5.3 Sistema de personagens e atributos

O sistema de personagens foi projetado de forma a separar claramente as responsabilidades associadas às diferentes fases do jogo. Cada personagem presente no tabuleiro é representado por duas scenes distintas, utilizadas de acordo com a fase do jogo: preparação ou batalha.

Durante a fase de preparação, o personagem é representado pela scene `Unit`, responsável pela interação direta com o jogador. Conforme descrito anteriormente, essa scene concentra a lógica relacionada ao posicionamento do personagem no tabuleiro, permitindo sua movimentação de acordo com os comandos do usuário. Já durante a fase de batalha, o personagem passa a ser representado pela scene `BattleUnit`, que encapsula os sistemas responsáveis pelo comportamento autônomo da unidade na batalha.

A scene `BattleUnit` contém os componentes responsáveis pela execução das ações de combate, incluindo a detecção e aplicação de dano, realização de ataques, identificação de adversários próximos, movimentação automática e lógica de inteligência artificial. Dessa forma, durante a batalha, o personagem atua de maneira independente, sem interferência direta do jogador.

Ambas as scenes consultam e modificam uma resource do tipo `UnitStats`, que centraliza os dados associados ao personagem. Cada personagem do jogo é definido por uma instância desse tipo de resource, na qual estão armazenadas informações relevantes utilizadas por diferentes sistemas do projeto, como a identificação do time ao qual o personagem pertence, a arte utilizada para sua representação visual, seu nome, vida máxima, vida atual, dano e velocidade de ataque.

Sempre que atributos relevantes da `UnitStats` são modificados, signals são emitidos para notificar outros sistemas do jogo sobre a alteração de estado. Um exemplo desse mecanismo é a atualização da vida atual do personagem, que dispara um signal, usado, por exemplo, para atualizar visualmente a barra de vida associada à unidade. Esse modelo orientado a eventos contribui para manter a comunicação entre os sistemas desacoplada e organizada.

O uso de resources para a definição dos atributos dos personagens facilita significativamente a criação de novas unidades, uma vez que todas as informações essenciais ficam

centralizadas em uma única estrutura de dados reutilizável. Além disso, a decisão de utilizar duas scenes distintas para representar o mesmo personagem em fases diferentes do jogo permitiu isolar a lógica de batalha da lógica de posicionamento, evitando sobrecarga de responsabilidades em um único componente.

5.4 Sistema de batalha

O sistema de batalha é o sistema mais complexo do protótipo, pois integra múltiplos subsistemas responsáveis pelo controle autônomo das unidades, tomada de decisões e resolução dos confrontos.

Para que uma batalha possa ser iniciada, é necessário que ao menos um personagem esteja posicionado no tabuleiro da área de batalha. O início do confronto ocorre por meio de um botão específico da interface, implementado como um node independente. Esse node recebe signals sempre que unidades são adicionadas ou removidas da área de batalha, controlando sua habilitação. Ao ser acionado pelo jogador, o botão altera a fase do jogo e inicia o processo de transição para a batalha.

Nesse momento, o node `BattleHandler` muda o jogo para a fase de batalha e realiza todas as alterações necessárias. Esse componente é responsável por preparar o tabuleiro de combate, substituindo todas as instâncias da scene `Unit` presentes no `UnitGrid` da área de batalha por novas instâncias da scene `BattleUnit`, utilizando a mesma instância de `UnitStats` usada pela `Unit`.

O `BattleHandler` também gerencia o encerramento da batalha, removendo os nodes instanciados durante o combate, restaurando a visibilidade das `Units` da fase de preparação e emitindo signals que indicam o time vencedor. O uso de groups facilita esse processo, permitindo operações coletivas sobre os múltiplos nodes com características semelhantes.

O comportamento das `BattleUnits` durante o combate é controlado por meio de uma máquina de estados finita. Cada estado é implementado como uma subclasse da classe base `State`, que define métodos comuns como `enter()` e `exit()`, executados respectivamente na entrada e saída de cada estado. Para o protótipo, foram implementados os estados de perseguição (`ChaseState`), bloqueio (`StuckState`), ataque automático (`AutoAttackState`) e execução de habilidades (`CastState`).

O `ChaseState` é o estado inicial das `BattleUnits`. Nele, o adversário mais próximo é definido como alvo e a unidade tenta se mover em sua direção. A movimentação é calculada pelo node `UnitNavigation`, que utiliza os nodes `UnitGrid` e `PlayArea` da área de batalha em conjunto com o node `AStarGrid2D`, responsável pela implementação do algoritmo A*. O grid utilizado pelo algoritmo é derivado do `UnitGrid` de batalha, no qual posições vazias são consideradas transitáveis, enquanto posições ocupadas por outras `BattleUnits` são tratadas como obstáculos. Movimentos diagonais são desabilitados e a heurística adotada é a euclidiana. Caso um caminho válido seja encontrado, o primeiro passo do caminho é retornado e o `UnitGrid` é atualizado; caso contrário, uma posição inválida é sinalizada.

Se uma `BattleUnit` não encontrar uma posição válida para se mover, ela transita para o `StuckState`. Após um intervalo de 0,5 segundos nesse estado, a unidade retorna

ao `ChaseState`, assumindo que a configuração do grid pode ter sido alterada devido ao movimento ou eliminação de outras unidades.

Quando um adversário entra no alcance de ataque, a `BattleUnit` passa para o `AutoAttackState`. Nesse estado, ataques são realizados em intervalos definidos pelo atributo de velocidade de ataque presente na `UnitStats`. A cada ataque, uma scene do ataque é instanciada, composta por nodes responsáveis pela detecção de colisão, representação visual e animação. Existem implementações distintas para ataques corpo a corpo e ataques à distância, sendo a escolha determinada pelo alcance de ataque da unidade. Quando o ataque colide com uma `BattleUnit` inimiga, o dano é aplicado e a vida do alvo é reduzida. Caso o alvo seja derrotado ou saia do alcance de ataque, a unidade retorna ao `ChaseState`.

Algumas `BattleUnits` podem possuir habilidades especiais associadas a itens adquiridos durante o jogo. Essas habilidades são ativadas após a execução de um número específico de ataques consecutivos, conforme definido no item. Durante a execução da habilidade, a `BattleUnit` entra no `CastState` e, ao final da ação, retorna ao `ChaseState`.

O `CastState` é responsável pela execução de habilidades especiais associadas às unidades durante a batalha. Ao entrar nesse estado, a `BattleUnit` instancia uma scene do tipo `UnitAbility`, que representa a habilidade a ser executada. Cada habilidade especial é implementada como uma variação dessa scene, contendo um script com um método padrão de execução (`cast`), responsável por aplicar os efeitos e emitir um signal ao término da ação. Esse modelo é análogo ao uso de polimorfismo na programação orientada a objetos, no qual uma classe base define uma interface comum e suas implementações concretas definem comportamentos específicos. Dessa forma, novas habilidades podem ser adicionadas sem alterações no sistema de batalha, mantendo a arquitetura modular e extensível.

As transições entre estados são acionadas por meio de signals, emitidos sempre que as condições de troca de estado são atendidas. O node `UnitAI` atua como um orquestrador da máquina de estados, conectando esses signals aos métodos responsáveis pelas transições, garantindo a consistência do comportamento das `BattleUnits` durante a batalha.

A utilização de uma máquina de estados finita para o controle do comportamento das `BattleUnits` trouxe vantagens significativas em termos de organização e clareza da lógica de combate. Ao separar as diferentes ações da unidade em estados bem definidos, foi possível reduzir a complexidade do código, facilitar a manutenção e tornar explícitas as transições entre comportamentos distintos. De forma complementar, o uso do algoritmo A* para a navegação das unidades no tabuleiro permitiu a busca eficiente de caminhos em um ambiente dinâmico, no qual obstáculos mudam constantemente devido à movimentação e eliminação das unidades. A adoção da heurística euclidiana direciona a busca de maneira eficiente, garantindo um comportamento de movimentação coerente e previsível durante as batalhas automáticas.

A batalha é encerrada quando todas as unidades de um dos times são derrotadas. Nesse momento, o `BattleHandler` finaliza o combate e retorna o jogo para a fase de preparação, dando continuidade ao fluxo principal do jogo.

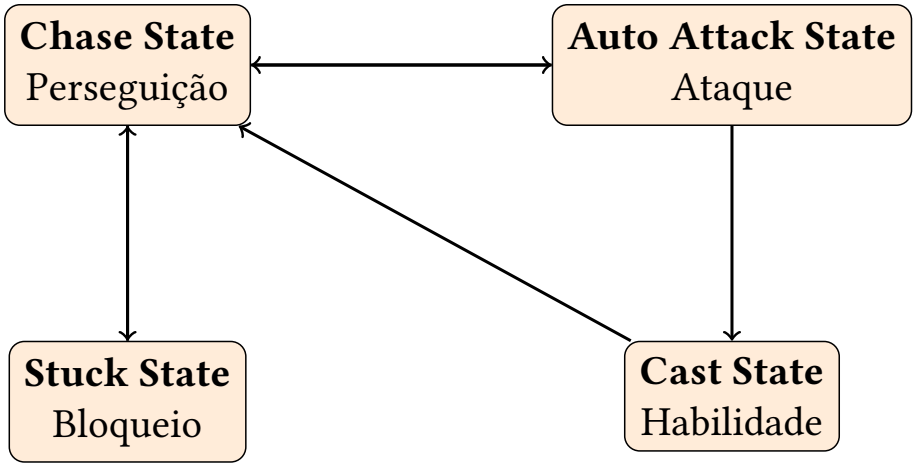

Figura 5.2: *Máquina de estados das BattleUnits durante a fase de batalha.*

5.5 Sistema de progressão e controle das fases do jogo

O sistema de progressão do jogo está diretamente relacionado ao controle das fases e à evolução do estado do jogador ao longo da partida. Esse sistema é composto principalmente pelas resources `GameState` e `PlayerState`, em conjunto com o node `BattleHandler`, que atua como coordenador do fluxo principal do jogo.

A resource `GameState` é responsável por definir os estados globais do jogo. No protótipo desenvolvido, foram implementados dois estados principais: `PREPARATION` e `BATTLE`, que representam, respectivamente, as fases de preparação e de combate. Além disso, o `GameState` armazena informações relacionadas à progressão dos níveis do jogo, como a definição dos inimigos e seus posicionamentos para cada nível, bem como a quantidade de pontos concedida ao jogador após a conclusão bem-sucedida de uma batalha.

A resource `PlayerState` armazena os dados relacionados ao progresso do jogador durante a partida. Nela são definidos os valores iniciais de pontos e nível do jogador, assim como seus valores atuais ao longo da execução do jogo. Essa resource também centraliza informações relacionadas à progressão econômica, como as probabilidades de itens aparecerem na loja de acordo com o nível do jogador. Além disso, o `PlayerState` fornece métodos auxiliares utilizados por outros nodes para consulta e modificação desses dados.

Essas resources são utilizadas pelo node `BattleHandler`, que desempenha um papel central no controle das fases do jogo. Esse node é responsável por alternar entre os estados de preparação e batalha, preparar o tabuleiro de combate e instanciar os inimigos de acordo com as configurações definidas no `GameState` e no `PlayerState`. Ao término de cada batalha, o `BattleHandler` também atualiza os atributos do jogador, como quantidade de pontos e nível, de acordo com o resultado do confronto, além de emitir signals para notificar outros sistemas sobre a transição de fase.

A centralização do controle de progressão e fases no `BattleHandler`, aliada ao uso de resources para o armazenamento de estado, trouxe vantagens significativas do ponto de

vista arquitetural. Ao separar claramente os dados do jogo (representados por `GameState` e `PlayerState`) da lógica de controle (implementada no `BattleHandler`), o sistema torna-se mais modular, coeso e de fácil manutenção. O uso de resources permite que o estado do jogo seja compartilhado entre diferentes sistemas sem duplicação de informações, além de possibilitar ajustes e extensões, como a inclusão de novos níveis, regras de progressão ou parâmetros de balanceamento sem a necessidade de alterações diretas na lógica central do jogo. Essa modelagem também reduz o acoplamento entre os componentes, favorecendo a reutilização de código e a evolução incremental do protótipo.

5.6 Sistema de itens e loja

Figura 5.3: *Interface da loja.*

O sistema de itens e loja é responsável por permitir a progressão estratégica do jogador por meio da aquisição e aplicação de modificadores às unidades. Esse sistema é implementado principalmente a partir da resource `ItemStats` e da scene `Item`, em conjunto com um conjunto de scenes responsáveis pela interface da loja.

A resource `ItemStats` centraliza todos os dados associados a um item do jogo. Nela estão definidos os atributos que podem ser modificados nas unidades, bem como informa-

ções específicas de cada item, como nome, custo, raridade e tipo. Cada item disponível no protótipo é representado por uma instância dessa resource, permitindo a reutilização e fácil expansão do conjunto de itens sem alterações diretas na lógica do jogo.

A scene `Item` representa visualmente os itens dentro do jogo e é composta por nodes responsáveis pela interação com o jogador. Essa scene contém um node de detecção de colisões, um node `DragAndDrop`, responsável pela movimentação do item por meio do mouse, e um node `OutlineHighlighter`, utilizado para destacar o item quando o cursor está sobre ele. Esses componentes seguem uma estrutura semelhante à utilizada na scene `Unit`, favorecendo a reutilização de lógica e consistência na interação.

A aplicação dos efeitos de um item em um personagem ocorre quando o jogador movimenta o item até uma unidade. Esse fluxo é tratado pela própria scene `Item` com o auxílio de groups. Quando o cursor do mouse está sobre uma unidade, ela é adicionada temporariamente ao group `hovered_unit`. Ao soltar um item, seus modificadores são aplicados à primeira unidade presente nesse group, caso exista. Após a aplicação dos efeitos, a instância do item é removida do jogo.

Os efeitos dos itens são aplicados diretamente sobre a resource `UnitStats`, modificando os atributos da unidade associada. Essa abordagem permite que os itens sejam utilizados tanto durante a fase de preparação, afetando `Units`, quanto durante a batalha, impactando `BattleUnits`, mantendo consistência no sistema de atributos.

A aquisição de itens ocorre por meio do sistema de loja, no qual o jogador utiliza os pontos acumulados para realizar compras. O sistema de loja é composto pelas scenes `Shop`, `ItemCard`, `GoldDisplay` e `RerollButton`, além da resource `ItemPool`.

A resource `ItemPool` define o conjunto de itens que podem aparecer na loja, sendo composta principalmente por uma coleção de `ItemStats` configurável por meio da interface da Godot Engine. A scene `Shop` atua como o componente principal da loja, agregando múltiplas instâncias de `ItemCard`, além das scenes `GoldDisplay` e `RerollButton`. A `Shop` também contém a lógica responsável por decidir quais itens serão exibidos, utilizando as probabilidades de aparição definidas no `PlayerState` de acordo com o nível do jogador.

A scene `ItemCard` representa visualmente uma opção de compra dentro da loja. Implementada como um botão, ela exibe dinamicamente informações do item associado, como nome, arte, tipo, custo e raridade, com base nos dados presentes na `ItemStats` correspondente.

A scene `GoldDisplay` exibe a quantidade atual de gold do jogador, obtida a partir da resource `PlayerState` por meio de variáveis exportadas. Já a scene `RerollButton` permite a atualização das opções disponíveis na loja, oferecendo novas possibilidades de compra ao jogador.

Ao realizar a compra de um item, a scene `Shop` emite o signal `item_bought`, que é tratado pelo node `ItemSpawner` presente na cena principal `Arena`. Esse node é responsável por instanciar uma nova scene `Item` na primeira posição livre da área `ItemsBench`.

A scene `ItemsBench` funciona de forma semelhante ao Bench de unidades, sendo composta por um node do tipo `UnitGrid` e um node `TileHighlighter`. Essa área define as posições disponíveis para armazenar os itens adquiridos, permitindo que eles sejam

organizados e movimentados pelo jogador. As regras e métodos de movimentação dos itens dentro dessa área são implementados no node `ItemMover`.

A modelagem do sistema de itens e loja dessa forma trouxe benefícios importantes para a organização e extensibilidade do protótipo. A separação entre dados e comportamento, obtida por meio do uso de resources para representar os atributos dos itens (`ItemStats`) e scenes para sua representação e interação visual (`Item` e componentes da loja), reduziu o acoplamento entre os sistemas e facilitou a manutenção do código. Essa abordagem permite a adição de novos itens, variações de raridade ou ajustes de balanceamento diretamente nos dados, sem a necessidade de alterações na lógica central do jogo. Além disso, o reaproveitamento de componentes já existentes, como `DragAndDrop` e `OutlineHighlighter`, promove consistência na interação com o usuário e reduz a duplicação de código. A comunicação baseada em signals entre a loja e a cena principal também contribui para uma arquitetura mais modular, tornando o sistema de progressão por itens flexível e preparado para expansões futuras.

Capítulo 6

Resultados

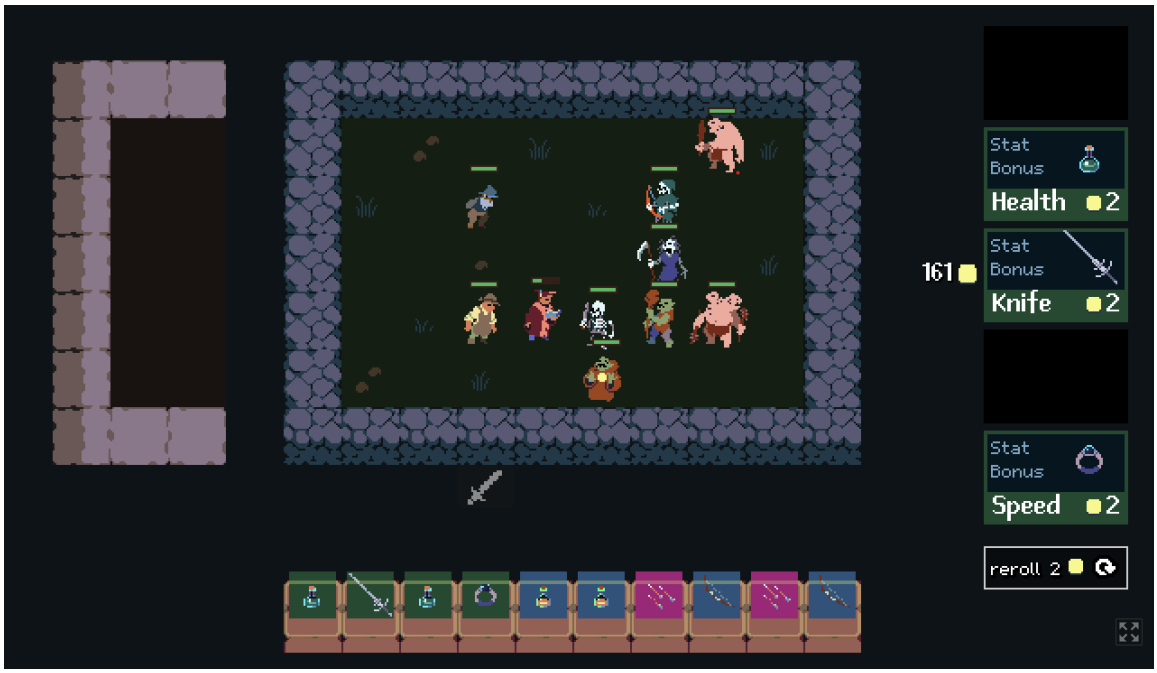

Figura 6.1: *Tela do protótipo resultante.*

Como resultado deste trabalho, um protótipo de um jogo digital 2D funcional que combina elementos do gênero *auto battler* com sistemas de progressão inspirados em jogos *roguelike* foi desenvolvido. O protótipo conseguiu implementar os elementos principais comumente encontrados nos principais jogos de cada gênero, contemplando um ciclo completo de jogo, com as fases de preparação, batalha e progressão do jogador, como descrito nos objetivos.

O sistema de posicionamento dos personagens permitiu a interação do jogador com as unidades, possibilitando o posicionamento de forma estratégica dentro das posições estabelecidas pelo jogo. Esse sistema se mostrou consistente, permitindo a movimentação

de diferentes tipos de componentes entre as diferentes áreas do jogo, impondo as regras de movimentação para cada componente.

O sistema de batalha funcionou adequadamente, permitindo que as unidades tivessem o comportamento esperado de forma autônoma e funcionando simultaneamente. A máquina de estados finita definiu o comportamento para cada estado do personagem na batalha, utilizando a navegação baseada no algoritmo A* para se movimentar em um tabuleiro com obstáculos dinâmicos no estado de perseguição, enquanto os demais estados permitiram um comportamento coerente durante a batalha. O resultado mostrou uma batalha que acontece de forma completamente automática, caracterizando o protótipo como um jogo do gênero *auto battler*.

O sistema de progressão, aliado ao sistema de itens e da loja, permitiu ao jogador evoluir seus personagens de forma única a cada partida, aplicando modificadores que influenciam diretamente seus desempenhos em batalhas, reforçando os elementos de rejogabilidade e progressão encontrados em jogos do gênero *roguelike*.

Do ponto de vista arquitetural, a organização modular do projeto demonstrou-se eficaz para o desenvolvimento de um protótipo com múltiplos sistemas interdependentes. A separação entre lógica e dados, realizada principalmente por meio do uso de *resources*, permitiu centralizar informações essenciais, como atributos de unidades, itens e estado do jogo, reduzindo a duplicação de dados e facilitando alterações e extensões futuras. Essa abordagem mostrou-se especialmente adequada em um contexto de prototipagem, no qual ajustes frequentes são necessários ao longo do desenvolvimento.

A comunicação entre os diferentes sistemas foi baseada majoritariamente no uso de *signals*, o que contribuiu para um baixo acoplamento entre os componentes. Esse modelo permitiu que sistemas como batalha, progressão, loja e interface reagissem a mudanças de estado sem depender diretamente da implementação interna uns dos outros. Como consequência, o código tornou-se mais organizado, legível e reutilizável, além de facilitar a manutenção e a identificação de responsabilidades dentro da arquitetura do projeto.

A estrutura modular adotada também facilitou a adição de funcionalidades, como novos itens, habilidades e personagens, sem a necessidade de reestruturações significativas no código já existente. Isso mostra que a arquitetura escolhida atende ao objetivo de servir como uma base sólida para expansões futuras.

Além da implementação dos sistemas centrais do jogo, o protótipo desenvolvido foi exportado para a plataforma Web e publicado no site *itch.io*, uma plataforma amplamente utilizada para a distribuição de jogos independentes. A Godot Engine facilitou esse processo ao oferecer suporte nativo à exportação para Web, permitindo a geração do projeto de forma integrada ao ambiente de desenvolvimento. Essa abordagem possibilitou a execução do jogo diretamente em navegadores compatíveis, sem a necessidade de instalação adicional, facilitando o acesso e a validação do protótipo. A publicação na plataforma *itch.io* permitiu a disponibilização do projeto para testes externos, evidenciando sua viabilidade como um produto funcional.

Em relação às limitações do protótipo, destaca-se a ausência de um balanceamento aprofundado das mecânicas de jogo, uma vez que o foco principal do trabalho esteve na implementação e integração dos sistemas centrais. A inteligência artificial das unidades,

embora funcional e suficiente para demonstrar o fluxo de batalhas automáticas, utiliza estratégias simples, sem técnicas mais avançadas de tomada de decisão. Além disso, não foram implementados sistemas como persistência de dados entre sessões e mais tipos de itens e habilidades especiais, por não serem essenciais para a validação do objetivo principal do projeto. Adicionalmente, destaca-se a ausência de testes formais com outros usuários. Devido a restrições de tempo durante o desenvolvimento do projeto, não foi possível realizar sessões de teste com jogadores externos ou coletar feedback sistemático sobre a experiência de uso, usabilidade e balanceamento das mecânicas.

Apesar dessas limitações, os resultados obtidos atendem aos objetivos propostos neste trabalho. O protótipo desenvolvido demonstra a viabilidade da combinação dos gêneros *auto battler* e *roguelike* em um único sistema, validando as decisões de design e arquitetura adotadas. O projeto cumpre seu papel como prova de conceito, mostrando que as decisões tomadas durante o desenvolvimento foram adequadas para a construção de jogos digitais com estruturas complexas e sistemas interdependentes.

Capítulo 7

Conclusão

Este trabalho teve como objetivo o desenvolvimento de um protótipo de jogo digital 2D que combinasse mecânicas do gênero *auto battler* com sistemas de progressão inspirados em jogos *roguelike*. Para isso, foram estudados conceitos e jogos relevantes de ambos os gêneros, a partir dos quais foi definido o design do projeto e implementados os sistemas fundamentais necessários para validar essa proposta.

Ao longo do desenvolvimento, foi possível implementar um ciclo completo de jogo, englobando as fases de preparação, batalha e progressão do jogador. O protótipo resultante apresenta sistemas funcionais de posicionamento de personagens, controle de batalhas automáticas, progressão por meio de itens e gerenciamento do estado do jogo. Esses sistemas foram integrados de forma coesa, permitindo ao jogador tomar decisões estratégicas que influenciam diretamente o desempenho nas batalhas subsequentes.

Do ponto de vista técnico, a utilização da Godot Engine mostrou-se adequada para o desenvolvimento do protótipo. A arquitetura modular adotada, baseada no uso de *nodes*, *scenes*, *resources* e *signals*, possibilitou a separação clara de responsabilidades entre os diferentes sistemas do jogo. Essa abordagem contribuiu para um código mais organizado, reutilizável e de fácil manutenção, além de facilitar a adição gradual de novas funcionalidades ao longo do desenvolvimento.

Embora o protótipo apresente limitações, como a ausência de balanceamento aprofundado, inteligência artificial simplificada e a não implementação de sistemas de persistência de dados, essas restrições são compatíveis com o escopo do trabalho, cujo foco esteve na validação da arquitetura e na integração dos sistemas centrais do jogo. Além disso, devido a limitações de tempo, não foi possível realizar testes formais com outros usuários, o que restringe a avaliação empírica da experiência do jogador, da usabilidade e do balanceamento das mecânicas. Ainda assim, os testes funcionais realizados durante o desenvolvimento indicam que os sistemas implementados operam de forma consistente e integrada.

Como trabalhos futuros, destacam-se a possibilidade de aprimorar a inteligência artificial das unidades, realizar o balanceamento das mecânicas de jogo, expandir o sistema de progressão com novos tipos de itens e habilidades, além da implementação de persistência de dados. A realização de testes de usabilidade e validação com jogadores externos também

se apresenta como um passo fundamental para a evolução do protótipo em direção a uma versão mais completa e refinada.

Dessa forma, conclui-se que o objetivo principal deste trabalho foi atingido, resultando em um protótipo funcional que demonstra, de maneira prática, a aplicação de conceitos de engenharia de software e desenvolvimento de jogos digitais na construção de um sistema complexo e modular.

Referências

[Godot Engine 2025] Godot Engine. *Godot Engine*. 2025. URL: https://godotengine.org/ (citado na pg. 4).

[Godot Engine Documentation 2025] Godot Engine Documentation. *GDScript basics*. 2025. URL: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html (citado na pg. 4).

[PwC 2025] PwC. *Global Entertainment & Media Outlook 2025*. 2025. URL: https://www.pwc.com/gx/en/news-room/press-releases/2025/pwc-global-entertainment-media-outlook.html (citado na pg. 1).

[Schell 2023] Jesse Schell. *The Art of Game Design: A Book of Lenses*. 3ª ed. CRC Press, 2023. ISBN: 978-0367212736 (citado nas pgs. 3, 4).

[Wikipedia 2025a] Wikipedia. *Auto battler*. 2025. URL: https://en.wikipedia.org/wiki/Auto_battler (citado nas pgs. 1, 8).

[Wikipedia 2025b] Wikipedia. *Balatro*. 2025. URL: https://pt.wikipedia.org/wiki/Balatro (citado na pg. 7).

[Wikipedia 2025c] Wikipedia. *Dota Auto Chess*. 2025. URL: https://en.wikipedia.org/wiki/Dota_Auto_Chess (citado na pg. 8).

[Wikipedia 2025d] Wikipedia. *Hearthstone – Game modes*. 2025. URL: https://en.wikipedia.org/wiki/Hearthstone#Game_modes (citado na pg. 8).

[Wikipedia 2025e] Wikipedia. *Roguelike*. 2025. URL: https://pt.wikipedia.org/wiki/Roguelike (citado nas pgs. 1, 7).

[Wikipedia 2025f] Wikipedia. *Slay the Spire*. 2025. URL: https://pt.wikipedia.org/wiki/Slay_the_Spire (citado na pg. 7).

[Wikipedia 2025g] Wikipedia. *Teamfight Tactics*. 2025. URL: https://en.wikipedia.org/wiki/Teamfight_Tactics (citado na pg. 8).

[Wikipedia 2025h] Wikipedia. *Vampire Survivors*. 2025. URL: https://en.wikipedia.org/wiki/Vampire_Survivors (citado na pg. 7).