RESIGeT On-Line
Revista Eletrônica de Sistemas de Informação e Gestão Tecnológica
Vol. 15 Nro. 1 2025
Uni-FACEF Centro Universitário Municipal de Franca

# ESTUDO DE CASO DE UM RPG DE TURNO NA ENGINE GODOT

Yune Uzuki
Graduanda em Engenharia de Software – Uni-FACEF
yune.uzuki@gmail.com

Márcio Maestrelo Funes
Mestre em Ciência da Computação – Uni-FACEF
marciofunes@gmail.com

**Resumo**

Este trabalho analisou a viabilidade da *engine* Godot para o desenvolvimento de um RPG de turno. A partir da criação de um protótipo funcional e da avaliação de suas principais características, concluiu-se que a Godot oferece recursos técnicos adequados e flexibilidade criativa superior em relação a *engines* especializadas, como o *RPG Maker*. O modelo de código-fonte aberto e as questões de licenciamento foram considerados diferenciais relevantes, apesar dos desafios relacionados à publicação em plataformas fechadas. A *engine* mostrou-se viável para a maioria dos projetos, desde que os requisitos específicos de cada equipe sejam cuidadosamente avaliados. Recomenda-se, para trabalhos futuros, expandir a análise para outros gêneros de jogos e desenvolver aplicações práticas de maior escopo.

**Palavras-chave**: *Desenvolvimento de Jogos. Godot Engine. RPG de Turno*.

**Abstract**

*This study evaluated the feasibility of the Godot engine for turn-based RPG development. Through the creation of a functional prototype and an analysis of its core features, it was concluded that Godot provides suitable technical resources and greater creative flexibility compared to specialized engines such as RPG Maker. Its open-source model and licensing aspects were identified as significant advantages, despite challenges related to publishing on closed platforms. The engine proved to be viable for most projects, provided that the specific requirements of each development team are carefully considered. Future research is encouraged to explore other game genres and to develop broader practical applications.*

**Keywords**: *Godot Engine. Turn Based RPG, Game Development.*

## 1 Introdução

Um dos principais desafios enfrentados por equipes e empresas de desenvolvimento de jogos ocorre logo na etapa inicial: a escolha da *engine* responsável por viabilizar o projeto. A *engine* é uma ferramenta crucial no desenvolvimento, se trata da ferramenta principal em que o jogo será criado, onde todos os elementos que o compõem, desde arte, música, jogabilidade, entre outros, se encontram para formar o artefato final. Uma escolha inadequada de *engine* pode comprometer significativamente o desenvolvimento, resultando em perdas de tempo, recursos financeiros e esforço técnico.

Atualmente, a indústria de jogos oferece uma ampla variedade de *engines* disponíveis para os desenvolvedores, estes podem escolher uma *engine* comercial geral, podem desenvolver sua própria, escolher uma *engine* específica para algum gênero de jogo, entre outras possibilidades. Embora a diversidade de *engines* cubra diferentes requisitos técnicos, ela também torna o processo de escolha mais complexo e desafiador.

Segundo dados do site *SteamDB* (Technologies, 2025), a *engine* mais utilizada atualmente para jogos publicados na plataforma *Steam* é a *Unity*, destacando-se por sua estrutura modular e flexível . Embora proporcione grande liberdade para a criação de mecânicas únicas devido à sua natureza modular, trata-se de uma ferramenta comercial, que impõe dificuldades relacionadas ao licenciamento e apresenta custos elevados para pequenas equipes.

No caso dos RPGs de turno, a engine *RPG Maker* surge como uma opção especializada, mas apresenta limitações significativas que devem ser consideradas no processo de escolha. Focada no desenvolvimento de RPGs de turno, a *RPG Maker* oferece funcionalidades pré-configuradas que atendem aos principais elementos do gênero. No entanto, este mesmo foco traz limitações a *engine*, fazendo com que certas funcionalidades que possam fugir um pouco do padrão do gênero sejam difíceis ou até mesmo impossíveis de se implementar na *RPG Maker*. Além disso, a *RPG Maker* é limitada ao desenvolvimento em 2D, sendo necessário recorrer a plugins com capacidades restritas para qualquer funcionalidade em 3D.

Considerando as limitações observadas nas *engines Unity* e *RPG Maker*, este artigo propõe a avaliação da viabilidade da engine *Godot* como engine para desenvolvimento de RPGs de turno. A *Godot* é uma *engine* de código aberto que também adota uma arquitetura modular, semelhante à da *Unity*. No entanto, a *Godot engine* é relativamente recente, e ainda carecemos de estudos sobre sua viabilidade para o desenvolvimento de jogos de RPG de turno. Este trabalho busca preencher essa lacuna, propondo a *Godot* como uma solução potencial para os desafios encontrados com outras *engines*, ao mesmo tempo em que testa sua viabilidade no cenário proposto.

## 2 Referencial Teórico

Para que seja claro o artigo, e para que seu objetivo de analisar a viabilidade de desenvolvimento de um jogo RPG utilizando da *engine Godot*, é necessário um breve entendimento de alguns assuntos diretamente relacionados. Para isso, um breve resumo da história das *engines*, quando, quais e o motivo de terem sido criadas, sua distribuição, alguns exemplos e tipos. Além disso, para que seja possível analisar a viabilidade da *Godot* especificamente para o desenvolvimento

de um jogo do gênero RPG, é necessário também uma definição de um jogo de tal gênero, seus aspectos principais e comuns entre vários destes.

## 2.1 Um pouco da história dos jogos e o efeito de *hardware* e *software* sobre o desenvolvimento

Inicialmente concebidos como breves experimentos para testar os limites e capacidades dos computadores, os jogos digitais podem ser considerados como tendo sua origem no algoritmo criado por Alan Turing, que permitia a um computador jogar xadrez contra um humano. Embora esse algoritmo não constitua um jogo propriamente dito, ele representa ideias iniciais que influenciaram o desenvolvimento futuro dos jogos. Segundo Donovan (2010), com o passar do tempo, os jogos deixaram de ser apenas demonstrações técnicas e passaram a refletir a criatividade e a expressão artística de seus desenvolvedores e colaboradores. A história dos jogos é, portanto, também a história de pessoas e de arte. Para uma compreensão completa dessa trajetória, é necessário considerar outros elementos fundamentais, como o *hardware* e as ferramentas utilizadas no desenvolvimento. Essas ferramentas incluem as chamadas *engines*, bem como recursos voltados à arte, música, programação, entre outros aspectos técnicos e criativos.

O *hardware* sempre atuou como um fator limitante, não como a fonte primária de inovação ou a causa direta da criação de determinados jogos, mas como o meio que possibilitou a sua existência. O console Nintendo 64, por exemplo, um dos primeiros a oferecer uma capacidade satisfatória de renderização 3D em tempo real, não foi o catalisador para a criação do jogo Mario 64, mas sim a plataforma que permitiu a concretização das ideias concebidas para ele. À medida que o *hardware* evolui e incorpora novas funcionalidades e capacidades, os desenvolvedores de jogos adquirem maior liberdade para executar suas concepções criativas (Donovan, 2010).

O *software* relacionado ao desenvolvimento de jogos, ao contrário do *hardware*, não impõe um fator limitante. De forma geral, sua ausência não cria limitações intrínsecas; pelo contrário, a existência de *softwares* de apoio, não exclusivos ao desenvolvimento de jogos, proporciona suporte e melhora a eficiência do processo. Se todas as etapas, como a criação de arte, música, modelos 3D e a física do jogo, precisassem ser realizadas exclusivamente por meio de linhas de código, o desenvolvimento de jogos seria inacessível para a grande maioria das pessoas.

## 2.2 O que são as *engines* e alguns exemplos

A definição de game *engines* pode variar significativamente, e diferentes artigos oferecem suas próprias interpretações, dado que essas ferramentas diferem amplamente em suas funcionalidades. Para os propósitos deste projeto, a definição adotada — por se adequar melhor às suas necessidades — é a de que uma *engine* é um sistema que auxilia no desenvolvimento de jogos, fornecendo, no mínimo, um sistema de visualização, um sistema de áudio ou acústica, e um sistema de controle. Essa definição é baseada nos sistemas principais da maioria dos jogos, conforme estabelecido por Sobota e Pietriková (2023), essa abordagem não apenas engloba praticamente todas as *engines* modernas, sem impor limitações a serem apenas 2D ou 3D, mas também abrange tanto as *engines* de propósito geral quanto aquelas especializadas em gêneros específicos de jogos.

*Engines* de propósito geral são projetadas para permitir a criação de qualquer gênero de jogo. Elas são mais complexas e oferecem uma vasta gama de

ferramentas, o que, embora possa diminuir a eficiência em certos casos por terem menos sistemas pré-prontos, compensa com maior liberdade e criatividade para o desenvolvedor. Em contrapartida, as *engines* focadas visam otimizar o processo de desenvolvimento para gêneros específicos. Exemplos notáveis incluem o *RPG Maker*, ideal para a criação de RPGs, e a *engine Ren'Py*, focada em *Visual Novels*. Embora esses sejam apenas alguns casos, a variedade de *engines* focadas é imensa, abrangendo desde as que se concentram em gêneros inteiros até as mais raras, criadas especificamente para um único jogo. O mercado de *softwares* para desenvolvimento de jogos, mais conhecidos como *game engines*, é diversificado, incluindo opções comerciais como *Unity* e *Unreal*, outras de código aberto e gratuitas como *Godot*, e as chamadas *engines* proprietárias. Estas últimas são desenvolvidas por empresas para uso interno, a exemplo da *RE Engine* da *Capcom*, que atualmente é utilizada em todos os seus jogos mais recentes.

Para os fins deste projeto, as principais *engines* a serem abordadas serão a *Godot* e a *Unity*. Devido à sua relevância, uma análise mais detalhada de cada uma será apresentada nos subtópicos seguintes. A seleção dessas ferramentas foi baseada em dois fatores principais: a *Godot*, por ser uma *engine* de propósito geral e de código aberto, compete diretamente com a *Unity*, a qual, por sua vez, foi escolhida por sua popularidade e semelhança com a *Godot*. Como a *Godot* constitui a principal *engine* de estudo neste trabalho, ela receberá maior foco, com informações mais detalhadas sobre suas funcionalidades e diferenciais. Grande parte do conteúdo relacionado ao *game design* apresentado a seguir é oriundo de comunidades e sites oficiais, que servirão como as principais fontes de informação.

### 2.2.1 Godot

A *Godot* é uma *engine* de propósito geral, gratuita e de código aberto, com suporte a múltiplas plataformas para o desenvolvimento de jogos 2D e 3D. Na data de elaboração deste artigo, sua versão estável é a 4.4, com versões beta da 4.5 já disponíveis para teste. A ferramenta suporta nativamente as linguagens *GDScript*, C# e C++, e ainda permite que os usuários criem ou utilizem extensões para outras linguagens de programação (Godot, 202–).

Lançada em 2014, a *Godot* foi inicialmente desenvolvida por Juan Linietsky e Ariel Manzur (First, 2014). Atualmente, sua manutenção é realizada pela comunidade e pela *Godot Foundation*, que aceita doações para financiar o desenvolvimento e a manutenção, além de oferecer suporte oficial e moderar os canais da comunidade.

### 2.2.2 Unity

Publicada em Junho de 2005, também uma *engine* de propósito geral, a *Unity* foi criada, inicialmente, com o objetivo de democratizar o desenvolvimento de jogos, eventualmente se tornando a *game engine* mais popular tanto entre grandes empresas quanto desenvolvedores pequenos. Se trata de uma *engine* primariamente 3D, mas que permite o desenvolvimento de jogos 2D, uma *engine* prioritária de código fechado, atualmente mantida pela empresa *Unity Technologies* (Unity, 202–).

## 2.3 Sobre a importância das *engines* no desenvolvimento

Como já mencionado no primeiro tópico do referencial, ao longo do tempo, os jogos se tornaram mais complexos, o que se manifestou em equipes de desenvolvimento maiores, na implementação de mecânicas e funcionalidades únicas, e em sistemas de renderização gráfica mais sofisticados. Essas evoluções, entre outras, contribuíram para que o processo de desenvolvimento de jogos se tornasse mais complexo e, consequentemente, menos acessível.

Para resolver esses desafios, surgiram as *engines*, que atuam como um nível de abstração fundamental para a criação de jogos, especialmente nos tempos atuais (Anderson, Engel, Comninos, McLoughlin, 2008). As *engines* estão para os jogos assim como as linguagens de programação estão para as máquinas. Quando uma *engine* lida com a renderização de ambientes, o desenvolvedor não precisa compreender em detalhes como esse processo funciona, nem tampouco implementar o sistema por conta própria. Esse princípio se aplica a qualquer funcionalidade oferecida pela *engine*, permitindo que o foco seja na criação e no design do jogo.

Outro fator importante das *engines* é a sua capacidade de unificar as diversas frentes do desenvolvimento de jogos e os profissionais a elas atrelados. De acordo com Toftedahl e Engström (2019), que analisaram a equipe de desenvolvimento de *Assassin's Creed: Odyssey*, foi constatado que o jogo envolveu um total de 692 funções diretamente relacionadas ao processo. Uma *engine* deve ser capaz de gerenciar e unificar todas essas funções, oferecendo suporte e funcionalidades que permitam que todos os membros da equipe contribuam, independentemente de sua experiência ou função.

Por fim, as *engines* padronizam e permitem a reutilização de implementações prontas de funcionalidades de jogos (Anderson, Engel, Comninos, McLoughlin, 2008). Muitas delas já incluem sistemas de física prontos, que o desenvolvedor pode personalizar ajustando valores numéricos. Além disso, sistemas de áudio e gráficos exigem apenas a importação dos *assets* e uma breve programação para que a *engine* saiba quando reproduzir os sons ou renderizar os gráficos. Essa abordagem se estende a uma vasta gama de outras funcionalidades que as *engines* podem incluir para facilitar o processo de desenvolvimento.

## 2.4 Jogos RPG de turno, fatores comuns e franquias relevantes

Os RPGs de turno têm suas raízes nos jogos de RPG de mesa, com o exemplo mais notório sendo *D&D* (*Dungeons & Dragons*). Nesses jogos, cada participante cria um personagem, enquanto um "mestre" desenvolve um mundo e descreve eventos e ações. O objetivo principal é proporcionar uma narrativa imersiva, permitindo que os jogadores usem a imaginação para interpretar os acontecimentos e se sintam na pele de seus personagens. As ações são executadas em turnos, baseando-se nas habilidades do personagem e, frequentemente, em rolagens de dados — mais especificamente o D20, um dado de 20 faces. As ações podem variar de combate e negociação a diálogo e investigação, sendo limitadas apenas pela criatividade dos jogadores (Wyatt, Schwalb, Cordell, 2014).

No contexto do combate em turno, como o nome sugere, as ações de jogadores e inimigos são executadas de forma alternada. Esse modelo, por geralmente conceder tempo ilimitado para que o jogador tome suas decisões, promove uma camada de complexidade estratégica. O desafio principal reside na preparação para a batalha e no uso inteligente das habilidades dos personagens contra as dos oponentes. Ações comuns a esse gênero incluem ataques básicos, uso

de habilidades especiais, magias, itens de apoio e ações de cura (Amereh, 2024). Além da saúde do personagem, outro recurso comum a ser gerenciado é o *MP* (*Mana Points*), geralmente utilizado para executar ações mais poderosas ou com maior impacto. Por fim, é crucial considerar as estatísticas que definem cada personagem, como força, magia, defesa, agilidade e destreza. Esses atributos influenciam diretamente o combate, determinando a eficácia das ações de cada personagem e expondo suas fraquezas.

Alguns exemplos de franquias desse gênero são *Final Fantasy*, sem dúvida a mais conhecida, com *Final Fantasy VII* sendo um dos RPGs mais vendidos da história. Outros títulos incluem *Dragon Quest*, da mesma desenvolvedora (*Square Enix*), e *The Legend of Heroes*, uma franquia menos popular, mas com grande relevância entre os fãs do gênero. Exemplos mais recentes, criados por equipes pequenas ou até por uma única pessoa, são *Chained Echoes*, *Sea of Stars* e *Deltarune*, além de muitos outros que poderiam ser citados.

## 3 Materiais e Métodos

Este projeto adota a metodologia de estudo de caso, uma abordagem de pesquisa qualitativa que busca analisar um evento — neste caso, o desenvolvimento de um jogo RPG de turno na *engine Godot* — de forma específica, focada e aprofundada. O caso de estudo será o desenvolvimento de um protótipo, que será criado ao longo deste projeto. O objetivo principal é avaliar o processo de desenvolvimento na *Godot* e verificar se a *engine* oferece as ferramentas necessárias para a sua execução. Os principais critérios avaliados serão desempenho, via métricas do jogo, facilidade do desenvolvimento, flexibilidade da *engine* e se a mesma provém de todas as ferramentas necessárias para o desenvolvimento de um jogo do gênero. Toda a avaliação será feita apenas por parte da autora.

Inicialmente, serão identificados os desafios associados ao desenvolvimento de um RPG de turno, com a subsequente elaboração de um Documento de Design de Jogo (*GDD - Game Design Document*). Este documento detalhará o funcionamento de cada funcionalidade do jogo, incluindo os *feedbacks* visuais e interativos. Além disso, o *GDD* abordará o ambiente, os personagens, os efeitos sonoros e todos os outros recursos necessários para o projeto.

As funcionalidades e mecânicas do jogo serão analisadas sob a perspectiva da engenharia de *software*, sendo tratadas como requisitos. Para cada uma, será descrito o "porquê", "como" e "o quê". As soluções propostas deverão utilizar exclusivamente ferramentas e funcionalidades nativas da *engine Godot*, a fim de demonstrar a sua capacidade de atender às demandas específicas do gênero.

Considerando a proposta modular da *engine*, a estrutura de nós e cenas, e a natureza orientada a objetos da linguagem *GDScript*, será adotada a metodologia de composição para o desenvolvimento. Em linguagens orientadas a objetos, a composição é realizada principalmente por meio de interfaces. O desenvolvedor cria múltiplas interfaces, cada uma com funcionalidades únicas. Ao incluir uma interface em uma classe, esta adquire os elementos da interface, o que promove a reutilização de código entre diversas classes. Isso contrasta com a metodologia de herança, na qual as classes filhas adquirem as características da classe pai (Gamma, Helm, Johnson, Vlissides, 1994.). No contexto da *engine Godot*, essa metodologia é implementada ao combinar nós com funcionalidades distintas sob um nó pai. Essencialmente, o nó pai herda as funcionalidades de cada um de seus nós filhos, o que será detalhado com mais profundidade adiante.

Os problemas a serem abordados neste projeto devem ser exclusivos ou essenciais para os RPGs de turno. Sendo assim, o foco será total na funcionalidade das batalhas, que é o fator definidor do gênero. Por exemplo, funcionalidades relacionadas à exploração de masmorras, como a movimentação do personagem, não serão incluídas neste trabalho. Embora a exploração seja comum no gênero, não é um requisito fundamental para que um jogo seja classificado como RPG de turno.

Após a elaboração dos documentos, será desenvolvido um protótipo de pequeno escopo para demonstrar os resultados da análise de viabilidade e do processo de desenvolvimento. Esse protótipo será integralmente baseado no *GDD*, incluindo todas as funcionalidades nele descritas.

Todos os artefatos gerados pelo projeto estarão disponíveis em um repositório, que será usado para o versionamento do código-fonte e da documentação. Concluído o processo, as funcionalidades utilizadas e as capacidades da *engine* serão avaliadas em um caso de uso. O foco dessa avaliação será apresentar os diferenciais da *engine*, destacando como ela facilita o desenvolvimento de um RPG de turno e os desafios. Portanto, o caso de uso não abordará todas as funcionalidades utilizadas, pois muitas delas são comuns a praticamente qualquer *engine* 3D, como a renderização de modelos, texturas e luzes, ou o gerenciamento de efeitos sonoros e músicas.

Já com relação as ferramentas, para a documentação, será utilizada a ferramenta *Obsidian*[1], um editor de texto focado em anotações que permite a criação e edição de arquivos *Markdown* (MD) e diagramas. Adicionalmente, o *Draw.io*[2] será empregado para a criação dos diagramas de classe. A documentação final será formatada em arquivos MD. O desenvolvimento do projeto será realizado na *engine Godot*, utilizando a linguagem de programação *GDScript*, como já mencionado. As ferramentas específicas da *engine* serão detalhadas no tópico de desenvolvimento, pois sua explicação será mais clara por meio de exemplos práticos. O versionamento do trabalho será feito por meio do *Git* e do *GitHub*. O repositório conterá uma pasta para a documentação, uma para o projeto da *Godot* (incluindo código-fonte e todos os assets do jogo), um arquivo *README* para facilitar a navegação e um arquivo *LICENSE* detalhando a licença do projeto.

[1] A ferramenta pode ser acessada em: https://obsidian.md/

[2] A ferramenta pode ser acessada em:https://app.diagrams.net/

Para facilitar a compreensão do processo de desenvolvimento, é relevante contextualizar o funcionamento da *engine Godot*, que opera fundamentalmente com base em nós e cenas. Nós, do inglês *Nodes*, são a fundação da *engine* e existem três tipos primários: *Node*, *Node2D* e *Node3D*, dos quais todos os outros derivam. Cada tipo de nó possui funcionalidades e propriedades específicas. Por exemplo, um nó do tipo *RigidBody3D* cria um corpo com propriedades físicas que pode ser afetado por objetos externos, enquanto um nó *CollisionShape3D* define uma forma 3D que influencia interações físicas. A vasta quantidade de tipos de nós impede que todos sejam abordados neste artigo. Além disso, cada nó pode ter um *script* associado, o que permite criar funcionalidades personalizadas. Cenas são combinações de múltiplos nós que formam um único grande nó, elas podem ser salvas e reutilizadas em outros contextos, por exemplo, é possível criar uma caixa combinando os nós *RigidBody3D*, *MeshInstance3D* (nó que renderiza modelos 3D) e *CollisionShape3D*, salvá-la como uma cena e, em seguida, instanciá-la em outras cenas do projeto sem precisar replicar todos esses passos (Introduction, 2025). Tipos de nós relevantes serão mencionados ao longo do tópico de desenvolvimento, conforme forem utilizados.

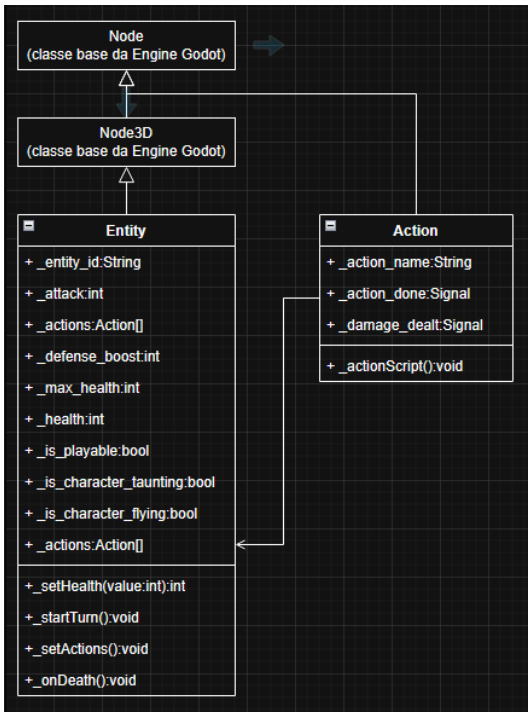
## 4 Desenvolvimento e Prototipação

Este tópico será dividido em duas partes: a documentação do projeto e o desenvolvimento do jogo em si. A primeira seção será breve, focando nos elementos considerados mais importantes para a documentação e sua formatação, além de incluir um anexo com o diagrama de classes. A segunda seção, sobre o desenvolvimento, será mais detalhada, com informações sobre cada ferramenta utilizada, uma breve explicação de sua função, seus usos e os motivos da escolha.

### 4.1 Elaboração do GDD e Diagramas de Classe

Para o *GDD*, o documento foi estruturado em quatro tópicos principais, cada um com seus subtópicos: Proposta e Escopo, Mecânicas, Visuais e Áudio. O tópico mais importante para o estudo foi o de Mecânicas, que incluiu os subtópicos Estatísticas, Ações, Turnos e Estados de Vitória ou Falha. Cada um desses subtópicos representa um dos "problemas" levantados sobre o desenvolvimento de um RPG de turno, abordando as mecânicas que definem o gênero. Para cada subtópico, o *GDD* descreveu o que a mecânica é, o porquê de sua inclusão e como ela será implementada utilizando as funcionalidades da *engine Godot*. A implementação das funcionalidades de estatísticas e ações exigiu a criação de uma classe, detalhada na documentação, e seu diagrama de classe foi elaborado para isso, disponível na Figura 1.

Figura 1: Diagrama de classes do projeto




Fonte: De autoria própria

### 4.2 Desenvolvimento do Jogo

O desenvolvimento do projeto foi dividido em diversas etapas, seguindo a ordem de execução. A primeira delas foi a criação dos visuais do jogo, que englobam o cenário, os personagens e as animações, seguida da implementação de toda a funcionalidade do jogo.

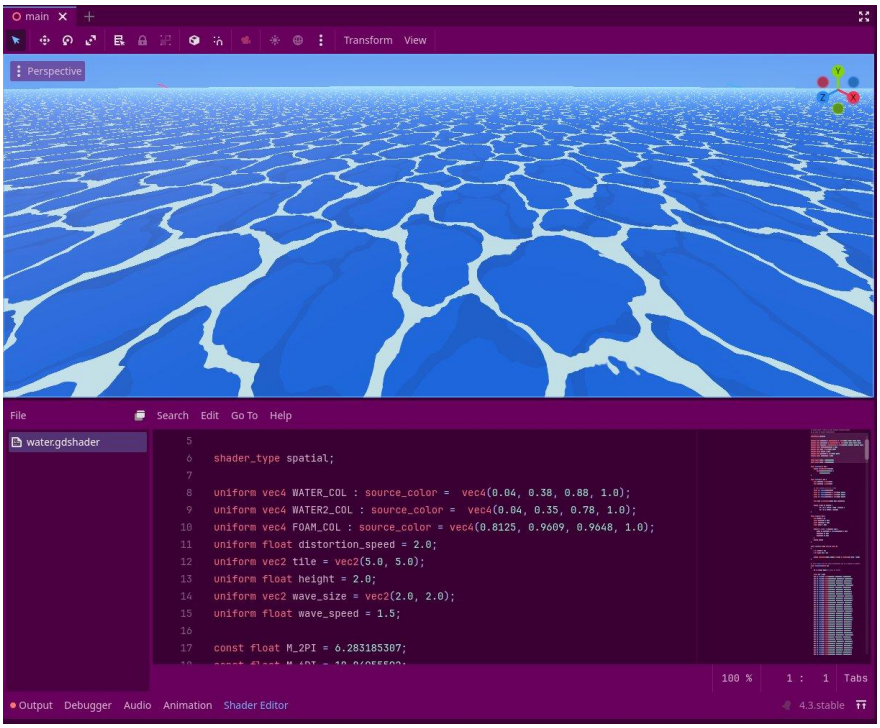
#### 4.2.1 Visuais - Ambiente e Personagens

A criação do ambiente e do cenário foi realizada com o uso de quatro funcionalidades da *engine Godot*: *WorldEnvironment*, *DirectionalLight3D*, *MeshInstance3D* e o sistema de *Shaders*. O *WorldEnvironment* é um tipo de nó usado para definir o ambiente do cenário, incluindo o céu, a neblina, as cores, as texturas e a intensidade dos efeitos. Embora exija algum conhecimento para ser utilizado de forma satisfatória, permite criar uma base visualmente atraente para o ambiente. O *DirectionalLight3D* é um nó mais simples que, como o nome indica, representa uma luz direcional que incide sobre todo o cenário, e neste projeto foi utilizado para simular a luz do sol. O *MeshInstance3D* é um nó que representa o visual de um objeto 3D, sem funcionalidades intrínsecas, mas que pode ter um *script* associado. O sistema de *Shaders* é uma ferramenta complexa que permite manipular visuais por meio de código. Foi utilizado para criar o oceano do cenário, dispensando o uso de texturas, apenas uma *MeshInstance3D* com o código dos *shaders* aplicado foi suficiente para gerar o efeito de ondas e definir as cores dos pixels da malha, o código utilizado foi obtido no site *Godotshaders*, de autoria do usuário *NekotoArts*[3]. O resultado final do

[3] O código-fonte pode ser acessado em: https://godotshaders.com/shader/wind-waker-water-no-textures-needed.

*shader* que simula o oceano, utilizado no desenvolvimento deste estudo de caso, pode ser visto na Figura 2.

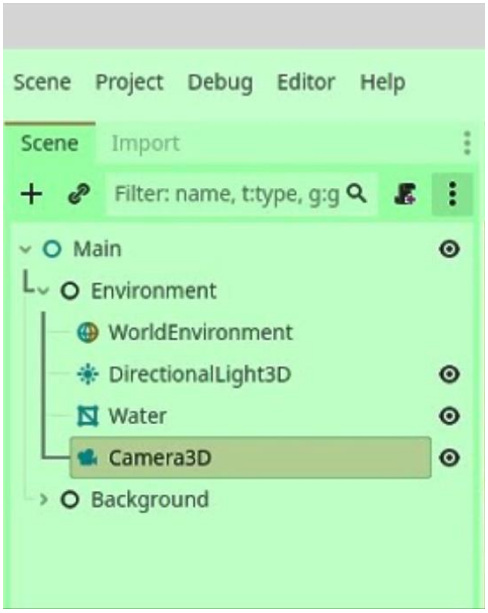
Figura 2: Captura de tela do *shader* de água em ação




Fonte: De autoria própria

A estrutura de nós do ambiente completo foi organizada de forma que os nós estivessem devidamente separados sob nós-pai, conforme ilustrado na Figura 3 onde o nó *Main* se trata do nó principal que define a cena, o nó *Environment* é um nó criado com o propósito de organizar os nós que compõe a ambientação do cenário e, por fim, o nó *Background* inclui todos os nós que compõe o fundo de tela do jogo.

Figura 3: Organização dos nós na cena principal



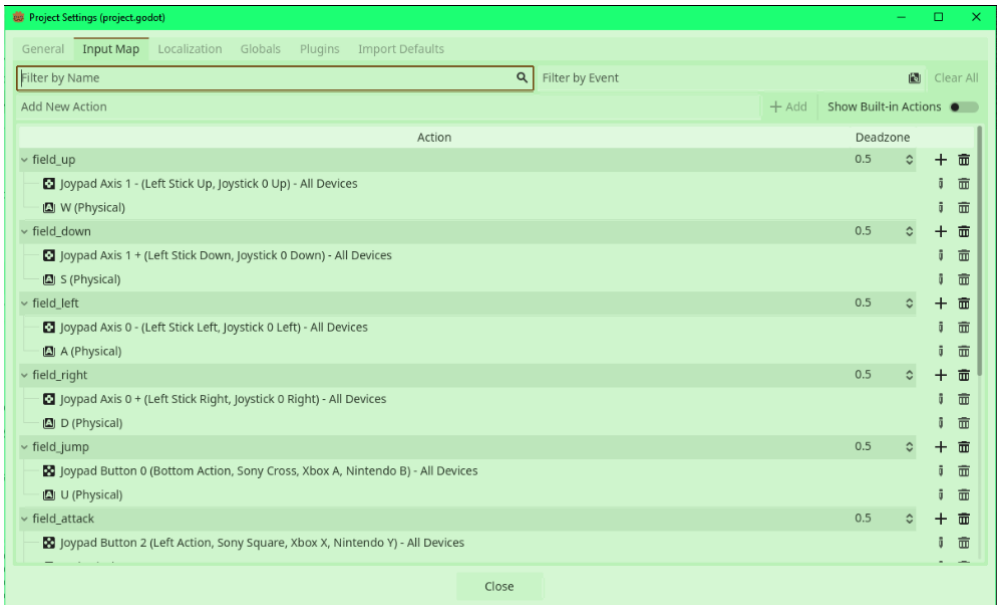
Fonte: De autoria própria.

Os visuais das personagens foram desenvolvidos utilizando um tipo de nó extremamente útil para prototipação provido pela engine, os nós *CSGBox3D* e *CSGCombiner3D*. *CSG* é uma categoria de nós que provém formas simples, focados em prototipação. Essas formas permitem que um desenvolvedor crie, rapidamente, um esboço de um ambiente ou personagem, para que, posteriormente, substitua por um *asset* final. Embora tenham foco em prototipação, podem ser combinados para criar um "modelo" 3D, caso queira salvar tempo e não precisar utilizar uma ferramenta de modelagem 3D. *CSGBox3D* cria um cubo, *CSGCombiner3D* serve para combinar todos os *CSGs* em um objeto só, será mais útil posteriormente para animar.

### 4.2.2 Mecânicas e jogabilidade

No contexto de desenvolvimento deste estudo de caso, foram criadas 4 mecânicas principais: estatísticas, ações, turnos e um estado final, seja vitória ou falha. Antes de dar início a implementação destas mecânicas, entretanto, algumas preparações são necessárias. Primeiro, foi necessário mapear os *inputs* do jogo, os *inputs* se tratam dos botões que são pressionados e como a *engine* interpreta os mesmos. A *engine Godot* possui a classe *Input*, uma classe criada especificamente para utilidade relacionada com *inputs*, seja identificar *inputs* pressionados, quando os mesmos foram "soltos" (o momento em que o jogador para de pressionar o botão), entre diversas outras utilidades (Input, 2025). Para que esta classe possa ser utilizada, precisa-se antes mapear os *inputs* que estarão utilizando e, para criar este mapeamento, o desenvolvedor precisa ir nas configurações do projeto e adicionar ações que são compostas de um nome e um ou mais *inputs* correspondentes como pode ser observado na Figura 4. A exemplo do mapeamento de *inputs*, caso deseje que a ação "cima" responda aos *inputs* W e seta para cima do teclado, basta criar a ação cima e adicionar ambos estes *inputs*, assim, sempre que um destes botões forem pressionados, o jogo irá identificar a ação cima.

Figura 4: Tela de configuração de *inputs* da *engine Godot*




Fonte: De própria autoria.

Para este estudo de caso, não será necessário criar novos mapeamentos de *inputs*, pois a *engine* já oferece padrões para a navegação em

menus, e todas as ações do jogo dependem exclusivamente deles. Outro passo preparatório será a criação e configuração dos nós *TurnManager* e *StateManager* como nós globais. Na *Godot*, nós globais são únicos, com um nome reservado, e estão sempre carregados e presentes em todas as cenas. Quando uma cena é reiniciada ou uma nova é carregada, esses nós permanecem ativos, o que lhes permite manter informações. Embora essa funcionalidade não fosse estritamente necessária para este projeto, já que ele possui apenas uma cena, ela será utilizada para garantir a escalabilidade do jogo no futuro.

Para a implementação de todas as mecânicas do jogo, é essencial descrever a organização da cena principal e os elementos que a compõem. A cena principal é estruturada em quatro nós principais que se comunicam entre si: *UI*, *Entities*, *TurnManager* e *StateManager*. O nó *UI* é o responsável por gerenciar toda a interface de usuário do jogo, incluindo a barra de vida de cada entidade, o menu de ações e as telas de fim de jogo. Suas funções incluem exibir, ocultar e atualizar os valores dos menus. O nó *TurnManager* é encarregado de todas as tarefas de gerenciamento e alternância de turnos. No início do jogo, ele gera um vetor que define a ordem das ações e lida com o avanço dos turnos. O nó *StateManager* gerencia o estado geral do jogo. Sempre que ocorre uma instância de dano, ele verifica se a vida total de ambas as equipes é superior a zero. Em caso de alteração, ele se comunica com o nó *UI*. Além disso, ele gerencia a seleção de alvos para ações, verificando a existência de alvos válidos no estado atual do jogo. O nó *Entities* é responsável por gerenciar os personagens de cada equipe, criando um vetor de personagens e enviando-o ao *StateManager* para que este possa verificar o estado do jogo e selecionar os alvos.

Dentro do nó *Entities*, existem nós filhos do tipo *Entity*, que representam os personagens, conforme detalhado no diagrama de classes. Essa organização modular e escalável permite que, para criar uma nova entidade, o desenvolvedor simplesmente adicione um nó do tipo *Node3D*, associe a ele o script *entity.gd* e, a partir daí, configure as estatísticas do personagem. Outra etapa necessária para a criação de um personagem é a adição do nó *Actions*, invocado pelo *script* para definir as ações às quais cada personagem tem acesso. Finalmente, dentro do nó *Actions*, basta adicionar nós do tipo *Action* para representar as ações desejadas. Os nós do tipo *Action* também estão detalhados no diagrama de classes.

A inicialização do jogo ocorre de forma sequencial, à medida que cada nó é carregado. Para isso, é utilizada a função *_ready*, disponível em qualquer nó ou em suas classes derivadas, que é executada assim que o nó termina de carregar e está presente na cena. O primeiro a agir é o nó *Entities*, que, em sua função *_ready()*, invoca a função *_setEntities* do nó *StateManager*, passando como parâmetro a lista de seus nós filhos (*self.get_children()*). A função *_setEntities* cria um vetor no nó *StateManager* a partir desse parâmetro, que contém todos os personagens do jogo. Desse modo, o *StateManager* pode validar o estado do jogo por meio das entidades e verificar quais são alvos válidos. Quando esse processo é concluído, o *StateManager* se comunica com o *TurnManager* para que este defina a ordem dos turnos. Devido à estrutura da cena principal, o nó de *UI* é o último a ser carregado, pois a *engine Godot* carrega os nós na mesma ordem em que estão posicionados no menu da cena. Sendo assim, a função *_ready* do *UI* é a que aciona a função *_nextTurn* no gerenciador de turnos, dando início à primeira rodada.

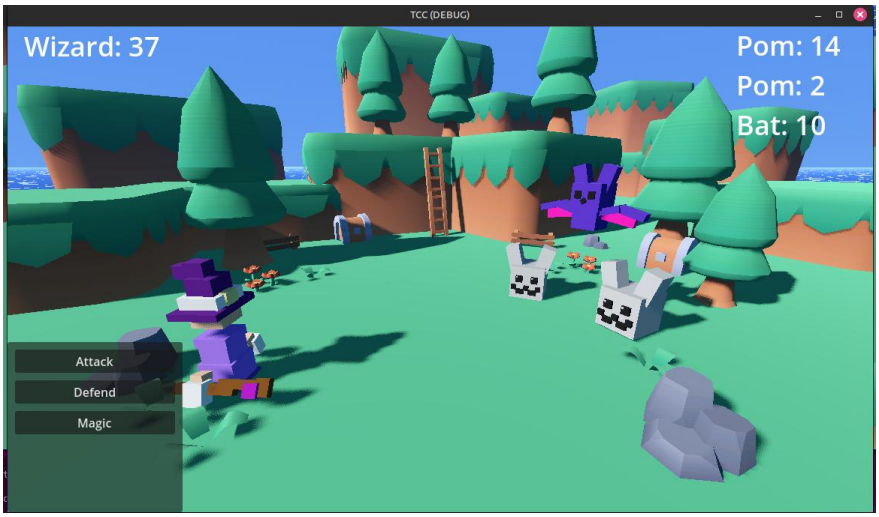
A função *_nextTurn* inicia sua execução verificando o turno atual em relação ao tamanho do vetor de entidades. Se o valor do turno for inferior ao tamanho do vetor, ele é incrementado; caso seja igual ou superior, é reiniciado para 0, reiniciando o ciclo de turnos. Após definir o valor do turno, o gerenciador de turnos identifica a entidade correspondente. A vida da entidade é verificada, e se ela for igual ou inferior a 0, seu turno é automaticamente pulado, pois a entidade é considerada abatida. No caso de uma entidade jogável, o nó *TurnManager* emite um sinal para indicar que é a vez de um jogador. É crucial notar que os nós *TurnManager* e *StateManager* são nós globais, o que significa que eles não fazem parte da hierarquia de nós da cena e, portanto, não podem se comunicar diretamente com os nós internos. Por isso, a comunicação é realizada por meio de sinais. O sinal que indica um turno de jogador é recebido pelo nó *UI*, que aciona sua função *_enableMenu*. Esta função torna o menu de ações visível e cria botões correspondentes a cada ação que o personagem pode executar. Ao selecionar uma ação, o *script* associado a ela é chamado para sua execução. Após a conclusão da ação, o ciclo é reiniciado com uma nova chamada à função *_nextTurn*. Se a ação resultar em alteração na saúde do personagem, a função *_checkHP* do *StateManager* é invocada para verificar se o jogo deve ser encerrado. Por fim, caso o turno atual seja de um personagem não jogável, a função *_nextTurn* seleciona e executa uma ação aleatória para ele.

### 4.2.3 Exportar jogo e performance

A *engine Godot* oferece uma funcionalidade de exportação que simplifica o trabalho do desenvolvedor. Esse recurso automatiza a compilação de binários e a criação de pacotes para diversas plataformas, eliminando a necessidade de um processo manual e, consequentemente, aumentando a eficiência e a acessibilidade do desenvolvimento.

A função de exportação, conforme ilustrado, oferece predefinições para as plataformas *Android*, *iOS*, *Linux*, *macOS*, *HTML* e *Windows*. Cada uma dessas predefinições gera um arquivo de build específico com configurações únicas, atendendo às particularidades de cada sistema. No contexto deste estudo de caso, o jogo foi exportado para a plataforma *Linux.* Segue, abaixo, uma captura de tela do jogo exportado, na Figura 5.

Figura 4: Captura de tela do jogo exportado e rodando.



Fonte: De autoria própria

Antes da exportação, foram adicionados nós para coletar métricas de performance, incluindo os quadros por segundo (*frames per second - FPS*) do jogo. A avaliação foi realizada em um notebook *Acer Aspire A515-45* com processador *Ryzen 7 5700U* e *GPU* integrada, equipado com 40 GB de memória RAM DDR4.

Os resultados de desempenho foram considerados extremamente satisfatórios. O jogo alcançou uma média de 89 *FPS* em resolução 1080p e 179 *FPS* em 720p. Essa performance é notável para um jogo 3D executado em uma máquina com *GPU* integrada, superando significativamente o mínimo de 30 *FPS* esperado para a resolução de 720p.

Em relação ao uso de memória, o jogo consumiu apenas 1124 MB. Para contextualizar, um projeto vazio da *engine* utiliza 1024 MB, o que indica que o projeto em si adicionou um consumo de apenas cerca de 100 MB. A eficiência no gerenciamento de memória pode ser otimizada ainda mais por meio da classe *ResourceLoader*, que permite ao desenvolvedor customizar o carregamento de recursos, além do uso das funções *load* e *preload*. Embora essas funcionalidades não tenham sido utilizadas neste protótipo, pois seus assets não exigiam carregamento interativo, elas são essenciais para o desenvolvimento de jogos com maior complexidade, incluindo RPGs de turno mais completos. Segue, abaixo, uma tabela descrevendo os resultados.

| Resolução | *Frames per Second* | Uso de memória |
|---|---|---|
| 720p | 179 FPS | 1124MB |
| 1080p | 89 FPS | 1124MB |

Os artefatos do desenvolvimento, incluindo a documentação e o código-fonte, o repositório do projeto no *GitHub* podem ser acessados pelo link: https://github.com/YuneUzuki/tcc.

## 5 Estudo de Caso

Com o desenvolvimento e os estudos necessários concluídos, esta seção tem como objetivo analisar a *engine* e o processo de criação do protótipo, discutindo de que forma ela facilitou ou, eventualmente, dificultou o projeto. Primeiramente, será abordado o processo de desenvolvimento em si, detalhando as ferramentas da *engine* que foram utilizadas para a criação do jogo. Em seguida, a discussão se estenderá a fatores externos que são relevantes para os desenvolvedores, como as questões de licenciamento e a natureza de código aberto da Godot. Considerando o escopo limitado do jogo desenvolvido, também serão mencionadas ferramentas que, embora não tenham sido necessárias para este projeto, seriam cruciais para a criação de um RPG de turno mais completo. Tais ferramentas se aplicariam à mecânicas comuns ao gênero, como a exploração de masmorras, sistemas de diálogo entre personagens, e sistemas de equipamento e estatísticas mais robustos, conforme discutido nos capítulos anteriores.

Esta seção do trabalho apresenta uma discussão sobre os argumentos a favor e contra o uso da *engine Godot* para o desenvolvimento de um RPG de turno.

A natureza desta análise é, em certa medida, interpretativa, uma vez que não é possível provar de forma puramente objetiva a viabilidade completa da *engine* para este gênero específico. Portanto, embora o objetivo seja apresentar a avaliação de maneira tão objetiva quanto possível, a seção incorpora considerações baseadas na experiência prática obtida no projeto, refletindo uma abordagem analítica e argumentativa.

### 5.1 Composição

Em uma discussão sobre a *engine Godot*, torna-se essencial abordar a metodologia de composição, considerando seus princípios de funcionamento, benefícios e possíveis desafios. Portanto, este será o primeiro tópico da seção. A discussão se concentra no impacto da composição no desenvolvimento do combate por turnos e na modularidade que ela confere ao processo de criação do jogo. A análise será fundamentada na experiência prática obtida no desenvolvimento do projeto e em exemplos que se relacionam com o tema.

#### 5.1.1 Benefícios no desenvolvimento do combate em turno

A *engine Godot* implementa a metodologia de composição de forma prática, através de sua estrutura de nós e cenas. Nela, o desenvolvedor compõe um objeto ao adicionar nós filhos a um nó principal, onde cada nó filho é responsável por uma funcionalidade específica. Como exemplo, em um jogo de plataforma 3D, a organização do personagem principal geralmente segue uma hierarquia de nós. O nó principal atua como um contêiner, e possui como nós filhos um nó que gerencia os aspectos visuais, o qual, por sua vez, contém o modelo 3D do personagem e um nó para lidar com as animações. Além disso, o nó principal possui outros nós filhos que são responsáveis por funcionalidades como física, estados e lógica de jogo. Essa estrutura permite a clara separação de preocupações e torna o processo de desenvolvimento mais modular.

Durante o desenvolvimento do protótipo, a metodologia de composição foi utilizada em todo o processo, por ser o princípio fundamental da *engine*. No entanto, seu impacto é mais notável na implementação do sistema de combate, em particular nas ações que cada personagem pode executar. Cada nó pode conter seu próprio *script*, e a interconexão entre *scripts* é realizada por meio de referências. Por exemplo, um nó pai pode buscar seus nós filhos e armazenar referências a eles em uma variável. A partir dessa referência, o nó pai obtém acesso às funções definidas nos *scripts* dos nós filhos. Utilizando a composição, é possível criar máquinas de estado de forma eficiente. O sistema de combate do jogo foi concebido como uma grande máquina de estado, onde cada personagem em combate representa um estado diferente. Para criar um personagem com habilidades específicas, como magia, cura ou distração, basta adicionar um novo nó da classe *Entity* e, sob ele, anexar os nós correspondentes a cada uma dessas funcionalidades. Essa abordagem, que utiliza o sistema de nós e cenas da *engine*, permite criar nós que encapsulam toda a funcionalidade de uma habilidade, tornando-os reutilizáveis e aplicáveis a qualquer entidade de combate.

A metodologia de composição provou ser extremamente benéfica para o desenvolvimento do protótipo, mas sua utilidade se estende a qualquer RPG de turno, especialmente em larga escala, por acelerar a criação de novos personagens e inimigos. Em um RPG que exige um grande número de entidades, a composição simplifica o processo de desenvolvimento, transformando a criação de novas

funcionalidades em uma tarefa de reutilização, com nós pré-codificados que podem ser simplesmente arrastados para a entidade, como foi demonstrado neste projeto. Embora o desenvolvedor ainda precise codificar a máquina de estado e a funcionalidade de cada nó individualmente, o esforço inicial é recompensado pela possibilidade de reutilizar esses nós sempre que uma funcionalidade similar for necessária. Essa abordagem pode ser aplicada a mais do que apenas personagens, um exemplo disso é a criação de habilidades complexas, como um ataque que atinge um inimigo múltiplas vezes e cura o personagem com base no dano causado. Com uma boa arquitetura de código, essa habilidade poderia ser composta por um nó pai e diversos nós filhos, cada um com uma funcionalidade específica (ataque, cura).

Para implementar essa metodologia na *Godot*, o desenvolvedor precisa criar *scripts* que referenciam e executam as funcionalidades dos nós filhos. A *engine* oferece funcionalidades nativas que facilitam isso, como a função *get_children*, que retorna uma lista de todos os nós filhos, e a funcionalidade *class_name*, que permite a criação de classes customizadas. Com isso, é possível, por exemplo, criar uma classe Funcionalidade com uma função executar_funcionalidade. O script da habilidade, então, usaria *get_children* para obter a lista de nós de funcionalidade filhos e, ao ser ativado, percorreria essa lista, executando a função em cada nó.

Em suma, a metodologia de composição mostra-se altamente eficaz no desenvolvimento de RPGs de turno, um gênero que, por sua natureza, envolve muitas habilidades, personagens e inimigos que compartilham mecânicas e funções. Ao promover a reutilização, a composição reduz significativamente a quantidade de código e a complexidade de diagramas de classe, além de diminuir o tempo de planejamento. Essas vantagens, somadas às funcionalidades da *engine Godot*, resultam em uma notável redução de custos e tempo de desenvolvimento. Além disso, a abordagem permite que contribuidores menos familiarizados com a programação, como *designers*, participem de forma mais direta na criação de funcionalidades, o que é um fator crucial para equipes de desenvolvimento de jogos.

### 5.1.2 A modularidade da composição

Como mencionado anteriormente, a *engine Godot*, por meio de seu sistema de nós e da metodologia de composição, proporciona um alto nível de modularidade ao processo de desenvolvimento. Essa modularidade oferece diversos benefícios para a criação de um jogo em geral, e, em particular, para um RPG de turno. Embora a discussão anterior tenha se focado nos benefícios específicos para o sistema de combate, é relevante abordar, de forma mais geral, as vantagens que a modularidade confere.

Ao separar as características de um objeto em módulos distintos — por exemplo, dividindo um personagem em módulos de visual, áudio e funcionalidade — a manutenção e as alterações se tornam mais eficientes. Considere um cenário em que as primeiras versões de um jogo priorizam a funcionalidade, utilizando modelos e animações provisórias. Em um ambiente com modularização, os artistas da equipe podem aprimorar esses elementos visuais no futuro, interagindo com interfaces mais amigáveis e dedicadas à arte, sem a necessidade de alterar o código. Esse *design* modular permite que as mudanças no aspecto visual sejam feitas de forma independente, sem afetar outros módulos, desde que o planejamento e a integração inicial tenham sido realizados corretamente.

Além de aprimorar a manutenibilidade, um sistema modular contribui significativamente para a organização da base de código. A estrutura de nós pai e filho, por si só, opera de forma similar à indentação de um código, permitindo que o desenvolvedor identifique facilmente a hierarquia e as dependências entre os nós. A *engine* também oferece a funcionalidade de "colapsar" nós, ocultando todos os seus filhos e facilitando o foco em áreas específicas do projeto. Para uma organização ainda mais aprimorada, a *Godot* permite a criação de nós vazios, que não contêm *scripts* ou recursos, mas podem ser nomeados e utilizados como contêineres para agrupar outros nós. Essa característica proporciona uma customização flexível da estrutura do projeto, sem impactar a funcionalidade do jogo, permitindo que o desenvolvedor organize os elementos de acordo com suas preferências.

### 5.2 Open Source e licenciamento

Um dos principais diferenciais da *engine Godot* em relação a seus concorrentes é o seu modelo de código-fonte aberto (*open source*). Embora existam diversas outras opções de ferramentas de desenvolvimento de jogos com essa natureza — algumas, inclusive, como bibliotecas para linguagens de programação, como é o caso do *Pygame* para *Python* (About, 20–) — a *Godot* se destaca como uma das mais populares e uma das poucas engines de propósito geral no segmento. Sua ascensão a posiciona como a principal concorrente direta da *Unity*, a atualmente mais popular engine de desenvolvimento de jogos. Este tópico se dedicará a analisar as vantagens e desvantagens do seu modelo de código aberto e o impacto que isso tem no licenciamento do *software*.

Considerando a possibilidade de uso da *Godot* para o desenvolvimento de um RPG de turno comercial, este tema assume uma grande importância para qualquer desenvolvedor, seja ele um hobbyista com intenção de vender seu jogo, uma pequena equipe de desenvolvimento ou, até mesmo, uma grande empresa.

#### 5.2.1 Licenciamento da engine

A engine *Godot* é distribuída sob a licença *MIT*, uma das licenças mais permissivas no universo do *software*. Conforme descrito na seção de licenças do *website* oficial, ela permite que o usuário utilize a *engine* para qualquer finalidade, estude e modifique seu código, e redistribua versões modificadas, inclusive para fins comerciais e sob outras licenças. Para o desenvolvedor, o licenciamento do jogo criado na *Godot* também não possui restrições de uso comercial; a escolha da licença é inteiramente sua. O único requisito é que, ao distribuir o jogo, o desenvolvedor inclua, nos créditos ou na documentação, uma notificação de *copyright* que indique o uso da engine e, no mínimo, um link para a sua página de licenciamento (License, 20–).

Em essência, a licença *MIT* redefine a relação entre o desenvolvedor e a ferramenta, tornando-a altamente adaptável às suas necessidades. A possibilidade de modificar o código da engine é um fator de grande relevância, assim como o fato de o desenvolvedor manter a total propriedade intelectual sobre sua criação, sem a necessidade de pagar *royalties* ou taxas de uso da ferramenta. Outra vantagem significativa desse modelo é a sua simplicidade: para comercializar um jogo, basta fazer o *download* e incluir a indicação de uso da *Godot*, o que cria um caminho direto e transparente para a entrada no mercado.

Este modelo de licenciamento contrasta fortemente com o de *engines* proprietárias, como a *Unity*, que frequentemente impõem maiores complicações e

preocupações. O modelo de assinatura da *Unity*, por exemplo, baseia-se na receita gerada pelo desenvolvedor ou pela empresa, e os termos de contrato e os preços podem ser alterados a qualquer momento. Consequentemente, além de um percurso menos direto para a comercialização do jogo, os desenvolvedores podem enfrentar incertezas financeiras devido aos altos custos e aos valores variáveis da *engine* a partir de determinados níveis de receita.

### 5.2.2 Modificações na engine e comunidade

Outra vantagem crucial do modelo de código-fonte aberto é a possibilidade de modificações na *engine*. Por ter o código-fonte acessível, a comunidade não só pode contribuir para a melhoria constante da plataforma, mas também criar e compartilhar *plugins* e outras modificações, adicionando funcionalidades que facilitam o desenvolvimento de terceiros. Além disso, com uma licença permissiva, o próprio desenvolvedor tem total controle sobre a *engine*, podendo adaptá-la para atender a necessidades específicas de seu projeto.

Um exemplo prático e significativo dessa vantagem é a *Godot Asset Library*, um repositório comunitário onde *plugins* e modificações são compartilhados e podem ser facilmente integrados. Essa biblioteca, apoiada pelos desenvolvedores principais da *engine*, pode ser acessada diretamente da interface da *Godot*, oferecendo ao desenvolvedor a oportunidade de encontrar soluções prontas para problemas comuns. Existem, por exemplo, *plugins* para combate por turnos que, caso atendam a todas as funcionalidades desejadas, podem eliminar a necessidade de codificação manual para essa mecânica, permitindo que o desenvolvedor se concentre na integração e na parte artística.

A capacidade de modificar a *engine* oferece a liberdade de adaptá-la completamente às necessidades do projeto. É possível modificar até mesmo sistemas centrais, como a física e a biblioteca de renderização. Esse fator é especialmente relevante para jogos que exigem *engines* customizadas por falta de suporte a funcionalidades extremamente únicas, o que, de outra forma, forçaria os desenvolvedores a construir uma engine do zero. A *Godot* pode servir como uma base sólida e personalizável para a criação de uma *engine* sob medida, oferecendo uma solução robusta e adaptável.

### 5.2.3 Os desafios do código aberto

Embora o modelo de código-fonte aberto traga diversas vantagens, ele também apresenta desafios que desenvolvedores e usuários precisam enfrentar. O principal deles é a lentidão na integração de novas tecnologias e no ritmo geral de desenvolvimento. Apesar de contar com uma comunidade ativa, o processo de atualização da engine para incorporar funcionalidades de ponta, como as de inteligência artificial ou *machine learning*, é mais lento em comparação a uma engine comercial. A *Unity*, por exemplo, possui uma equipe dedicada e um poder financeiro significativamente maior, o que a coloca em uma posição mais avançada nessas áreas. Para projetos com requisitos tecnológicos específicos, como um RPG de turno com inimigos que utilizam *machine learning* para aprender com o jogador, a *Godot* pode não ser a escolha mais adequada.

A lentidão no desenvolvimento também pode resultar em funcionalidades incompletas ou instáveis, que exigem mais esforço para serem utilizadas de forma eficiente. Um exemplo disso é o nó *GridMap*, que, embora funcional, apresenta inconveniências de uso. Os comandos do teclado para rotação e

troca de funcionalidades no *GridMap* são os mesmos utilizados para mover a câmera, e a impossibilidade de travar um dos movimentos torna a interação trabalhosa. A causa desse problema é a baixa utilização da ferramenta: enquanto os *TileMaps* são extremamente populares em 2D, o desenvolvimento de cenários 3D frequentemente demanda uma abordagem mais detalhada e manual, o que faz com que os *GridMaps* sejam menos procurados, resultando em menos contribuições para a sua melhoria.

Outro desafio inerente ao modelo de código-fonte aberto e à licença *MIT* é a falta de suporte oficial para plataformas fechadas, como consoles de videogame. Para que um desenvolvedor possa publicar seu jogo nesses sistemas, ele é obrigado a trabalhar diretamente com empresas terceirizadas, como *publishers* ou estúdios especializados em *porting*. Essa necessidade representa um obstáculo para o desenvolvedor, limitando a liberdade que o modelo de código aberto e a licença *MIT* proporcionam, além de aumentar os custos de desenvolvimento. Consequentemente, diferentemente do que ocorre em plataformas abertas como o PC, a colaboração com uma *publisher* se torna uma exigência para desenvolvedores que desejam lançar seus jogos em consoles.

### 5.3 Documentação e recursos de aprendizado

Um dos pontos mais elogiados da *Unity* é a vasta quantidade de recursos de aprendizado disponíveis, como documentação completa, milhares de cursos, vídeos e artigos. A quantidade exata desses materiais é difícil de quantificar, mas sua presença é um fator crucial na avaliação de viabilidade, já que o treinamento de uma equipe gera custos. A existência de uma documentação bem elaborada é, portanto, essencial para otimizar o processo de desenvolvimento.

A *engine Godot*, neste aspecto, não fica atrás. Ela oferece uma documentação completa, com explicações claras e exemplos práticos para o desenvolvedor. Mais do que isso, essa documentação é embarcada na própria engine, permitindo acesso *offline* e eliminando a necessidade de uma conexão com a *internet*. A integração com o editor de código da *Godot* também é extremamente útil, pois o desenvolvedor pode simplesmente segurar a tecla Ctrl e clicar em qualquer classe ou função para abrir a página correspondente na documentação, diretamente no editor.

Em relação aos recursos de aprendizado, embora seja perceptível que a *Godot* ainda se encontra em desvantagem significativa em termos de volume de materiais de alta qualidade em comparação com a *Unity*, foram realizadas pesquisas exploratórias durante o processo de desenvolvimento do jogo, sempre que apoio ou novo aprendizado se demonstrou necessário, estas pesquisas demonstraram a existência de uma variedade de recursos. Foram encontrados cursos, tanto em formatos gratuitos quanto pagos, diversos tutoriais em vídeo, artigos e outros materiais de apoio.

Além desses métodos, a comunidade da *Godot* oferece canais de comunicação oficiais para suporte, como o servidor no *Discord*, que serve como um ambiente de apoio para que os usuários possam buscar assistência e solucionar dúvidas quando a documentação e os tutoriais online não são suficientes.

## 6 Conclusão

Este trabalho teve como objetivo avaliar a viabilidade da *engine Godot* para o desenvolvimento de um RPG de turno. A análise realizada, aliada à construção de um protótipo funcional, demonstrou que a *Godot* é uma ferramenta robusta e

versátil, mesmo que não ainda amplamente adotada por grandes estúdios da indústria de jogos.

Seus conceitos fundamentais — como a estrutura de nós, cenas e a metodologia de composição — mostraram-se adequados para atender às demandas específicas do gênero. Além disso, a natureza de propósito geral da *engine* proporciona uma liberdade criativa superior à de ferramentas especializadas, como o *RPG Maker*.

As questões de licenciamento e o modelo de código-fonte aberto também se destacam como diferenciais importantes. Embora apresentem desafios, como a ausência de suporte oficial para plataformas fechadas, esses aspectos contribuem para a redução de custos e maior controle sobre o projeto.

Contudo, é necessário considerar limitações específicas, especialmente no que diz respeito à publicação em consoles, que exige soluções alternativas e maior esforço técnico por parte da equipe de desenvolvimento.

Dessa forma, conclui-se que a *Godot* é uma escolha viável para a maioria dos projetos de RPG por turno, desde que as plataformas-alvo e os requisitos de segurança e sigilo sejam cuidadosamente avaliados.

Para trabalhos futuros, recomenda-se expandir a análise para outros gêneros de jogos, investigar mais profundamente as implicações do modelo *open source* em projetos comerciais e desenvolver aplicações práticas de maior escopo, que permitam uma avaliação mais abrangente da engine em contextos reais de produção.

**Referências**

ABOUT - pygame wiki [*S. l.*] 20–. Disponível em: https://www.pygame.org/wiki/about. Acesso em: 31 ago. 2025

AMEREH, Fatemeh. A Study and Implementation of Turn-Based Combat Systems in Role-Playing Games. **Aalto University School of Science**, [*s. l.*], 18 dez. 2024.

ANDERSON, E. F., ENGEL, Steffen., COMNINOS, Peter., & MCLOUGHLIN, Leigh. **The case for research in game engine architecture**, 2008.

DONOVAN, Tristan. **Replay**: The History of Videogames. [*S. l.*: *s. n.*], 2010.

FEATURES - Godot Engine. [*S. l.*], 202-?. Disponível em: https://godotengine.org/features/. Acesso em: 21 mar. 2025

FIRST Public Release! - Godot Engine. [*S. l.*], 2014. Disponível em: https://godotengine.org/article/first-public-release/. Acesso em: 21 mar. 2025.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns: Elements of Reusable Object-Oriented Software**. [*S. l.*: *s. n.*], 1994.

GODOT Docs - 4.4. [*S. l.*], 2025. Disponível em: https://docs.godotengine.org/en/stable/. Acesso em: 21 mar. 2025.

GODOT Engine - Free and open source 2D and 3D game engine. [*S. l.*], 202-?. Disponível em: https://godotengine.org/. Acesso em: 21 mar. 2025.

INPUT - Godot Engine (4.3 documentation in English). [*S. l.*], 2025. Disponível em: https://docs.godotengine.org/en/4.3/classes/class_input.html. Acesso em: 03 Sep. 2025

INTRODUCTION to Godot - Godot Engine (4.3) documentation in English. [*S. l.*], 2025. Disponível em: https://docs.godotengine.org/en/4.3/getting_started/introduction/introduction_to_godot.html. Acesso em: 03 Sep. 2025

LICENSE - Godot Engine [*S. l*], 20–. Disponível em: https://godotengine.org/license/. Acesso em: 30 ago. 2025.

SOBOTA, Branislav; PIETRIKOVÁ, Emília. The role of game engines in game development and teaching. In: **Computer Science for Game Development and Game Development for Computer Science**. IntechOpen, 2023

TOFTEDAHL, Marcus, and ENGSTROM Henrik. **A taxonomy of game engines and the tools that drive the industry.** *2019*

TECHNOLOGIES - SteamDB. [*S. l.*], 2025. Disponível em: https://steamdb.info/tech/ . Acesso em: 15 Sep. 2025

UNITY Engine. [*S. l.*], 202-?. Disponível em: https://unity.com/. Acesso em: 21 mar. 2025.

WYATT, James; SCHWALB, Robert J; CORDELL, Bruce R. **Dungeons & Dragons: Player‘s Handbook**. [*S. l.*: *s. n.*], 2014.