UNIVERSIDADE DE TAUBATÉ
Lucas Eustáquio Trajano Vieira
Rodolfo Candido de Sousa Santos

REESTRUTURAÇÃO DO GAME DESIGN DE UM TOWER DEFENSE CLÁSSICO APLICANDO GAMEPLAY LOOP ATIVO

Taubaté
2022

Lucas Eustáquio Trajano Vieira
Rodolfo Candido de Sousa Santos

REESTRUTURAÇÃO DO GAME DESIGN DE UM TOWER DEFENSE CLÁSSICO APLICANDO GAMEPLAY LOOP ATIVO

Trabalho de Conclusão de Curso apresentado para obtenção do Certificado de Especialização pelo Curso Engenharia de Computação do Departamento de Informática da Universidade de Taubaté,

Área de Concentração: Software Básico

Orientador: Prof. Dawilmar Guimarães Araújo

Taubaté - SP
2022

Grupo Especial de Tratamento da Informação - GETI
Sistema Integrado de Bibliotecas – SIBi
Universidade de Taubaté - Unitau

V658r Vieira, Lucas Eustáquio Trajano

Reestruturação do Game Design de um Tower Defense Clássico aplicando Gameplay Loop Ativo / Lucas Eustáquio Trajano Vieira , Rodolfo Candido de Sousa Santos. -- 2022.

42 f. : il.

Monografia (graduação) – Universidade de Taubaté, Departamento de Informática, 2022.

Orientação: Prof. Me. Dawilmar Guimarães Araujo, Departamento de Informática.

1. Tempo real. 2. Gameplay Loop. 3. Game Design. 4. Onipresença. 5. Tower Defense. I. Santos, Rodolfo Candido de Sousa. II. Universidade de Taubaté. Departamento de Informática. Graduação em Engenharia de Computação. III. Titulo.

CDD – 651.3822

Ficha catalográfica elaborada **por Shirlei Righeti – CRB-8/6995**

Lucas Eustáquio Trajano Vieira
Rodolfo Candido de Sousa Santos

REESTRUTURAÇÃO DO GAME DESIGN DE UM TOWER DEFENSE CLÁSSICO APLICANDO GAMEPLAY LOOP ATIVO

TCC apresentado para obtenção do Certificado de Especialização pelo Curso Engenharia de Computação do Departamento de Informática da Universidade de Taubaté,

Área de Concentração: Software Básico

Data:_______________

Resultado:_________

BANCA EXAMINADORA

Prof. Dr._______________________________

____

Assinatura_____________________________

Prof. Dr._______________________________

____

Assinatura_____________________________

Prof. Dr._______________________________

____

Assinatura_____________________________

Prof. Dr._______________________________

____

Assinatura_____________________________

Prof. Dr._______________________________

____

Assinatura_____________________________

“Acima de tudo, videogames são feitos apenas para um propósito. Diversão. Diversão para todo o mundo.”

Satoru Iwata, presidente da Nintendo (2002 - 2015)
Game Developers Conference 2006

RESUMO

Este trabalho de graduação é sobre o desenvolvimento de um jogo eletrônico de estratégia em tempo real *tower defense* (em português, “defesa de torres”) com modificações no *game design* que buscam aumentar o nível de interações jogador-jogo durante o *gameplay*. Um *tower defense* é um jogo onde o jogador assume o papel de um protetor onipresente que tem como principal objetivo impedir que entidades invasoras completem um pré-determinado percurso em um cenário. Para impedi-las o jogador dispõe apenas de ferramentas indiretas, e automáticas, com a capacidade de neutralizar essas entidades invasoras. Quando todos os invasores no cenário forem neutralizados, uma nova onda de invasores é iniciada e o ciclo se repete. Este gênero é uma sub-classificação dentro do gênero de jogos de estratégia e pode ser desenvolvido utilizando-se um de dois modelos distintos de *game design*. O primeiro, chamado de “estratégia baseada em turnos”, onde os eventos que compõem o *gameplay loop* acontecem de forma **sequencial**, e o segundo conceito denominado de “estratégia em tempo real”, onde esses mesmos eventos acontecem em **simultâneo**. Como o objetivo do jogo desenvolvido é aumentar o nível de interação jogador-jogo, o modelo que melhor se encaixa na proposta é o de estratégia em tempo real. Outra modificação no *game design* é de remover a onipresença do jogador, permitindo ele apenas controlar uma entidade dentro do jogo. Essa segunda modificação cria inúmeras limitações para o jogador, porém todas elas aumentam a necessidade do mesmo de interagir com o jogo.

Palavras-chave: Estratégia em Tempo Real, G*ameplay Loop*, *Game Design*, Onipresença, *Tower Defense*

ABSTRACT

This graduation work is about the development of a real-time strategy electronic game tower defense with game design modifications that seek to increase the level of player-game interactions during gameplay. A tower defense is a game where the player assumes the role of an omnipresent protector whose main objective is to prevent invading entities from completing a predetermined path through a scenario. To stop them, the player has only indirect, and automatic, tools with the ability to neutralize these invading entities. When all the invaders in the scenario have been neutralized, a new wave of invaders is initiated and the cycle repeats. This genre is a sub-classification within the strategy games genre and can be developed using one of two distinct game design models. The first, called "turn-based strategy", where the events that make up the gameplay loop happen **sequentially**, and the second concept called "real-time strategy", where these same events happen **simultaneously**. As the objective of the developed game is to increase the level of player-game interaction, the model that best fits the proposal is real-time strategy. Another modification in game design is to remove the omnipresence of the player, allowing him to control only one entity within the game. This second modification creates numerous limitations for the player, but all of them increase the need for him to interact with the game.

Keywords: Real-Time-Strategy, Gameplay Loop, Game Design, Omnipresence, Tower Defense

SUMÁRIO

1 INTRODUÇÃO **8**
1.1 OBJETIVO 10
1.2 ESTRUTURA E METODOLOGIA DO TRABALHO 10
2 PRINCIPAIS FERRAMENTAS **12**
2.1 MOTOR DE JOGO 12
2.1.1 GODOT 13
2.1.2 PORQUE A GODOT 17
2.2 FERRAMENTAS GRÁFICAS 18
3 GAME DESIGN **20**
3.1 GAMEPLAY LOOP 22
3.2 FATOR REPLAY 23
3.3 BALANCEAMENTO 25
4 DESENVOLVIMENTO DAS ENTIDADES **26**
4.1 JOGADOR 26
4.1.1 ESTADOS: BÁSICO E MODO CONSTRUÇÃO DE TORRETA 27
4.1.2 FUNÇÃO DE MOVIMENTO 29
4.2 TORRETA 32
4.3 INIMIGOS 34
4.3.1 CARACTERÍSTICAS BASE 34
4.3.2 FUNÇÕES PRINCIPAIS 36
5 CONCLUSÃO **38**
5.1 IMPLEMENTAÇÕES FUTURAS 38
REFERÊNCIAS **39**

LISTA DE SIGLAS

TBS : Turn Based Strategy (Estratégia Baseada em Turnos)

RTS : Real Time Strategy (Estratégia em Tempo Real)

IDE : Integrated development environment (Ambiente de Desenvolvimento Integrado)

BSD : Berkeley Software Distribution (Distribuição de Software Berkeley)

GUI : Graphical User Interface (Interface de Usuário Gráfica)

VR : Virtual Reality (Realidade Virtual)

1 INTRODUÇÃO

A definição para jogos do gênero de estratégia, à primeira vista, pode ser considerada abrangente e ambígua, pois muitos jogos, de diferentes estilos e gêneros, também dispõem de elementos que estimulam o pensamento estratégico por parte do jogador, tais como a capacidade de leitura de cenários e a capacidade de elaboração de soluções lógicas para determinados problemas. Para um jogo ser considerado um jogo de estratégia não basta apenas que ele tenha elementos de estratégia, mas sim que tais elementos tenham muito mais relevância que os demais. Em suma, um jogo de estratégia é um jogo onde a condição de vitória depende majoritariamente da capacidade do jogador de planejar e executar uma ou mais estratégias, tendo essa habilidade cognitiva muito mais relevância do que quaisquer outras habilidades do jogador. (ADAMS, 2014)

Jogos do gênero de estratégia se dividem em dois principais subgêneros: *TBS (turn-based strategy)* ou estratégia baseado em turnos, onde as ações do jogador e das entidades antagônicas do jogo são intercaladas em turnos, e *RTS (real-time strategy)* ou estratégia em tempo real, onde as ações do jogador e das entidades antagônicas acontecem em simultâneo. (ADAMS, 2014)(MOSS, 2017)

Dentro do gênero de jogos de estratégia, a classificação que melhor define o jogo desenvolvido neste trabalho de graduação é a de um *tower defense*[1]. Um *tower defense* é um jogo de estratégia onde o jogador assume o papel de um protetor onipresente que tem como principal objetivo impedir que entidades invasoras completem um determinado percurso, para impedi-las o jogador dispõe apenas de ferramentas indiretas, e automáticas, com a capacidade de neutralizar os invasores, quando todos os invasores no cenário forem neutralizados, uma nova onda de invasores é apresentada e o ciclo se repete. Posto isso, se um determinado número de invasores conseguirem percorrer todo o percurso, assim atingindo seu objetivo, o jogador perde o jogo. (AVERY, 2014)

Jogos de estratégia *tower defense* podem ser classificados tanto como jogos de estratégia em tempo real quanto como jogos de estratégia baseada em turnos, a definição desse gênero não implica nas limitações de *game design*[2] imposta por ambas classificações, porém como ambas classificações se anulam por

---

[1] Tradução direta: defesa de torres. Um gênero de jogos eletrônicos de estratégia.

[2] Tradução direta: projeto de jogo. São todos os conceitos do projeto que sustentam o produto final no caso de jogos eletrônicos.

conta da sua natureza, o desenvolvedor deve optar por qual estilo ele pretende aplicar a seu jogo *tower defense*. Em jogos mais recentes desse gênero é mais comum o estilo de estratégia por turnos, onde o jogador escolhe o momento que as entidades invasoras começam a percorrer o cenário, assim tendo total controle sobre quando se inicia uma nova rodada. No estilo estratégia em tempo real, os invasores invadem o cenário a todo momento, o jogador não tem controle e deve se adaptar "em tempo real" aos novos desafios apresentados, novas rodadas são apresentadas de forma automática. (MOSS, 2017)

O jogo desenvolvido neste trabalho, denominado "*Robots are trying to destroy my capsule. I hate them*", ilustrado na Figura 1, é um jogo de estratégia em tempo real *tower defense* onde o jogador não é um protetor onipresente, mas sim uma das entidades participantes no cenário. Usando um jogo de damas como exemplo ilustrativo dessa ideia da eliminação da onipresença do jogador, é como se o jogador não fosse o ser por trás do tabuleiro, mas sim uma das peças, porém uma peça com a capacidade extra de comandar e mover as demais peças aliadas. No jogo, o jogador assume o papel de um engenheiro mecânico que no momento enfrenta uma invasão de robôs no seu laboratório. O objetivo desses robôs é destruir uma cápsula de vidro localizada no centro do laboratório. Para impedir os robôs, o engenheiro precisa construir torres sentinelas que disparam projéteis nos invasores, assim impedindo o seu avanço. O jogo só acaba quando a cápsula for destruída, ou seja, o objetivo do jogador é suportar as hordas de robôs pelo máximo de tempo possível, assim obtendo uma pontuação final e não uma vitória definitiva. Podemos identificar todas as entidades de um *tower defense* no jogo proposto, o engenheiro assume o papel da entidade protetora, os robôs o papel dos invasores, o laboratório o cenário a ser percorrido pelos invasores e a cápsula a entidade a ser protegida, ou o objetivo final das entidades invasoras. (AVERY, 2014)

Figura 1: Captura de tela de um momento do jogo.[3]

1.1 OBJETIVO

Em síntese, este trabalho de graduação busca fazer a elaboração e o desenvolvimento de um jogo eletrônico de estratégia em tempo real *tower defense*, tendo como principal objetivo um produto que tenha bons padrões de qualidade definidos (KRAMER, 2000), isso sempre mantendo em perspectiva os limites impostos pela proposta inicial de aumentar o dinamismo do jogo. Com um *game design* mais voltado a característica a jogos do gênero de estratégia em tempo real e com a alteração do papel do jogador de entidade onipresente para entidade presente.

1.2 ESTRUTURA E METODOLOGIA DO TRABALHO

Em linhas gerais, o desenvolvimento desse projeto pode ser dividido em três partes: A primeira onde foram elaborados os conceitos básicos do game design do jogo, discutido com profundidade no Capítulo 3; A segunda onde foi desenvolvida as partes práticas do software, usando principalmente o motor de jogo *Godot*; A terceira parte onde foram feitas as correções de *bugs* e balanceamento do jogo.

[3] Fonte: elaborado pelo autor.

No Capítulo 1, Introdução, é trazida uma contextualização dos produtos de jogos eletrônicos, conceitos classificatórios de jogos de estratégia, noções básicas de *game design* e a proposta desse trabalho de graduação.

Em seguida, no Capítulo 2, são apresentadas as ferramentas utilizadas no desenvolvimento, dando um foco principalmente no motor de jogo *Godot*.

Já no Capítulo 3, é apresentado uma explicação detalhada sobre o *game design* formulado para o jogo desenvolvido, discorrendo sobre os objetivos, intenções e técnicas utilizadas para desenvolvimento, dando exemplos de aplicações presentes no produto.

No Capítulo 4 é explicado em detalhe os principais componentes técnicos do jogo, ou seja, o desenvolvimento prático das entidades no motor de jogo *Godot*. Entidades como: Jogador, Torreta e Inimigos.

E por fim, no Capítulo 5, é discorrido algumas ponderações referentes aos resultados do produto e ilações sobre possíveis implementações futuras.

2 PRINCIPAIS FERRAMENTAS

Para o desenvolvimento de um jogo eletrônico é necessário o conhecimento prático em distintas áreas de atuação. Jogos eletrônicos, normalmente, são desenvolvidos em grupo e por profissionais com habilidades distintas. Indubitavelmente, programação é uma das habilidades essenciais para o desenvolvimento, porém, de certo, não é a única necessária. Produção gráfica, e audiovisual como um todo, também é um dos aspectos essenciais para o desenvolvimento de qualquer jogo eletrônico. Posto essas ponderações, foram utilizadas duas principais ferramentas para o desenvolvimento do jogo em questão, uma voltada exclusivamente para parte gráfica e outra voltada a programação, estruturação e controle, no caso, o motor gráfico usado como base para o desenvolvimento.

2.1 MOTOR DE JOGO

No desenvolvimento de jogos eletrônicos é comum que os programadores, *game designers* e os outros profissionais utilizem *frameworks* para a criação de jogos, as chamadas game engines (motor de jogos). Com a complexidade cada vez maior dos jogos eletrônicos e dos *hardwares* que os executam, desenvolver tais motores de jogos é uma tarefa cada vez mais dispendiosa em recursos humanos e financeiros, de modo que existam empresas no ramo dedicadas apenas ao desenvolvimento desse tipo de *software*.

Como descrito no livro de 2016 *Technologies and Innovation: Second International Conference* (VALENCIA-GARCÍA, 2016), um motor de jogo é “uma estrutura de *software* concebida principalmente para a criação e desenvolvimento de jogos eletrônicos, de modo que inclui programas de apoio, bibliotecas relevantes e uma ou mais linguagem interpretada para ajudar a construir e unir os diferentes componentes de um projeto”. Um motor de jogo pode incluir um motor de renderização para gráficos 2D e/ou 3D, um motor de física ou detecção de colisão, som, animação, inteligência artificial, rede, gestão de recursos de *hardware*, suporte a localização, programação, etc. (ARM, 2022)

2.1.1 GODOT

A *Godot* é um motor de jogos multiplataforma de código aberto e livre, desenvolvida pela comunidade na linguagem de programação *C++*, lançada sob a

licença permissiva MIT[4] e sem custos para aplicações comerciais. Teve início de desenvolvimento em meados de 2007 pelos programadores argentinos Juan Linietsky e Ariel Manzur para uso interno no estúdio de jogos eletrônicos em que trabalhavam (GOMES, 2017), porém em janeiro de 2014 teve seu código fonte e licenciamento lançados publicamente. (LINIETSKY, 2014)

Suporta de maneira nativa as linguagens de programação *C#* (por meio de Mono[5]), *C++* e *GDScript*, esta última sua linguagem própria, de alto nível e de tipagem dinâmica, cuja sintaxe se assemelha a de *Python*. Possui um sistema chamado *GDNative*, que permite a criação de vinculação de nomes da linguagem interna (*binding*) do motor para outras linguagens de programação, como por exemplo as linguagens *Rust*, *JavaScript* e *Swift*. (Godot Engine, 2022)

Seu ambiente de desenvolvimento integrado (*IDE*) pode ser executado de forma nativa nos sistemas operacionais *Android*, *Apple macOS*, *Microsoft Windows*, e em sistemas *Linux* e outros da família BSD. Foi concebido para permitir criar jogos em 2D (duas dimensões) e 3D (três dimensões), mas também pode ser utilizado como um *toolkit* de interface gráfica de usuário (*GUI*), possibilitando a criação de softwares não relacionados a jogos eletrônicos.

É possível compilar executáveis para plataformas móveis (*Android* e *Apple iOS*), PC (todos os sistemas onde a *IDE* é executado), *Web* (*HTML5* e *WebAssembly*), plataformas de realidade virtual (*VR*), e tecnicamente os consoles de jogos (*Microsoft Xbox*, *Sony Playstation 4* e *5* e *Nintendo Switch*), embora para estes sistemas não é possível exportar oficialmente por questões de licenciamento relacionadas aos *kits* de desenvolvimento desses sistemas e a licença de código aberto do projeto. (Godot Docs, 2022)

Para a geração de gráficos a *Godot* utiliza as *APIs OpenGL ES 2.0* e *OpenGL ES 3.0*, este último com suporte ao uso de *shaders*[6] utilizados para criação de efeitos de pós-processamento e materiais. Possui dois motores gráficos independentes, um para gráfico bidimensional e o outro tridimensional, que podem ser combinados de acordo com as necessidades. O motor gráfico de duas dimensões suporta funções como *sprites* (*bitmap* bidimensional) e *sprites* animados,

---

[4] Licença originária no *Massachusetts Institute of Technology* na década de 1980 (MIT, 2022).
[5] Implementação livre e de código aberto do *framework .NET*.
[6] Programa que calcula os níveis de luz, cor e sombreamento dos *pixels* por meio de *hardware*. (WIRTZ, 2022)

luzes, sombras, shaders, conjuntos de ladrilho, rolagem de paralaxe, polígonos, animações, física e partículas, ao que o motor gráfico de três dimensões adiciona a esse suporte mapeamento de normal, especularidade, sombras dinâmicas por meio de mapas de sombra, iluminação global estática ou dinâmica e efeitos de pós-processamento. (Godot Engine, 2022)

A interface do ambiente de desenvolvimento da *Godot*, o Editor, é dividida nos componentes 2D, 3D e o *script*, onde cada seção possui elementos referentes às opções de cada área, como apresentado na Figura 2:

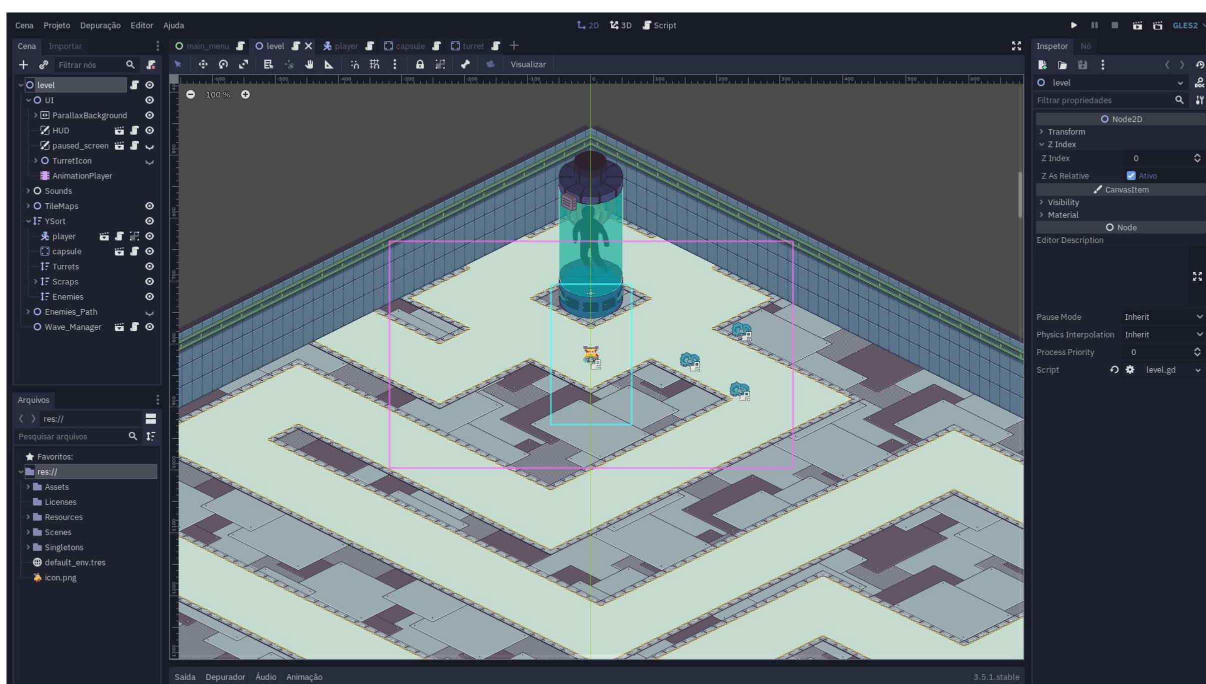

Figura 2: Interface do motor de jogo *Godot* exibindo o jogo em questão na área 2D.[7]

A arquitetura da *Godot* é dividida em quatro camadas mais o Editor que se integra a elas. O Figura 3 descreve, em forma de diagrama, a ordem das camadas e os componentes internos referentes e como se comunicam (Godot Docs, 2018).

[7] Fonte: elaborado pelo autor.

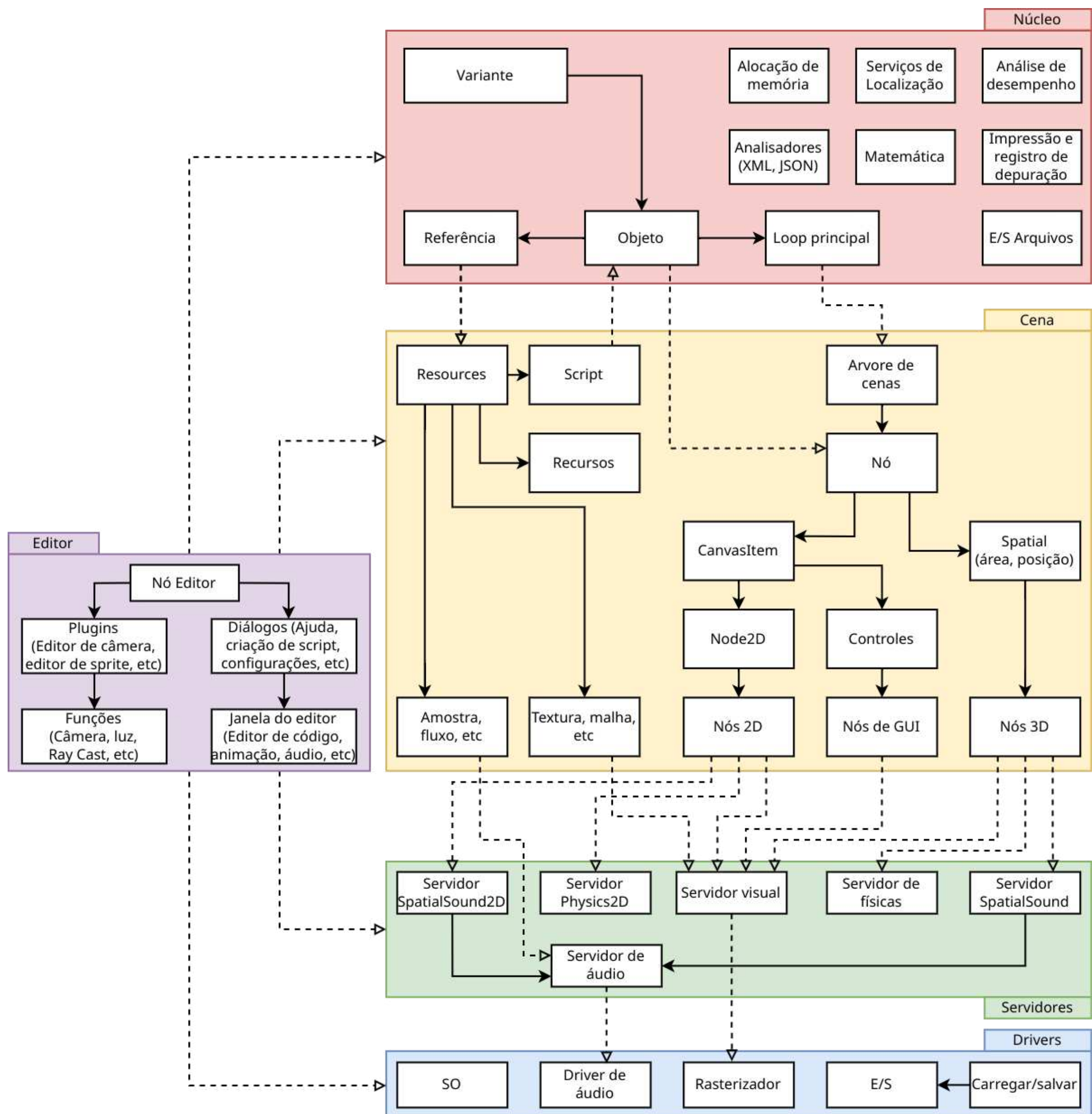

Figura 3: Diagrama da estrutura do motor de jogo *Godot*.[8]

- **Editor**: A camada Editor é uma coleção dos módulos que compõem a aplicação do Editor da *Godot*, usada para desenvolver *softwares*. O Editor não faz parte do motor de jogos em si, portanto está dependente de módulos de todas as camadas dele;
- **Núcleo**: A camada de Núcleo consiste no laço principal do programa, que o mantém em funcionamento acompanhado de módulos de classe que podem ser herdados para formar uma base consistente para cada classe, além de conter muitas funcionalidades básicas de baixo nível, tais como Matemática e

[8] Fonte: Godot Docs, 2018

Entrada/Saída de arquivos. As classes mais importantes desse módulo são Variante e Objeto, onde Variante pode conter todos os tipos de dados úteis para o motor e podem ser definidas para Objeto como propriedade. O Objeto forma uma base geral para uma classe, fornecendo elementos como *ID*, *flags* e métodos *get*. A Referência contém elementos básicos para classes que envolvem recursos, configurações, parâmetros, etc. Todos os Nós e recursos são herdados da classe Objeto;

- **Cena**: A camada de Cena consiste em tudo o que o desenvolvedor interage ao construir um jogo. Cada Cena é montada utilizando uma Árvore de Cenas (*SceneTree*), a qual compõe-se de instâncias de classes que herdam de Nó;
- **Servidores**: A camada de Servidores contém implementação detalhada para aspectos como gráficos, física, áudio, etc. Os desenvolvedores podem interagir com ela através de *scripts* e do Editor. O componente mais prevalente é o Servidor Visual, uma vez que tudo o que é visível no jogo é processado por ele;
- **Driver**: A camada de *Drivers* implementa funcionalidades fornecidas pelo layout do Servidor em um nível mais baixo, lidando com várias plataformas e sistemas externos, tornando assim os detalhes de baixo nível opacos para a camada de Servidores.

Todos os *softwares* desenvolvidos utilizando a *Godot* são construídos em torno do conceito de uma Árvores de Nós, conforme ilustra o esquema na Figura 4. Os Nós são elementos de base que funcionam como classes de objetos, podendo ser classes já existentes do motor (como Nós de exibição de imagens, de texto, de movimentação de entidades, de colisão, etc.) ou criados pelo programador, de modo a terem seu código atribuído, instanciado, reutilizado ou herdado de outros Nós. Podem possuir Nós filhos e se comunicam de maneira hierárquica dentro das Árvore de Cenas, que gerencia a hierarquia dos Nós de uma Cena, bem como as próprias Cenas.

As Cenas e os Nós podem ser instanciados dentro de outras Cenas por meio do editor gráfico da Godot ou por meio de código. A posição de determinados tipos de nós na hierarquia da classe determina a posição a qual ele é exibido no jogo.

A Figura 4 apresenta como essa estrutura funciona na prática dentro do projeto deste jogo, onde cada esfera é um Nó regido por uma Árvore de Cenas, compondo assim o nível raiz do jogo:

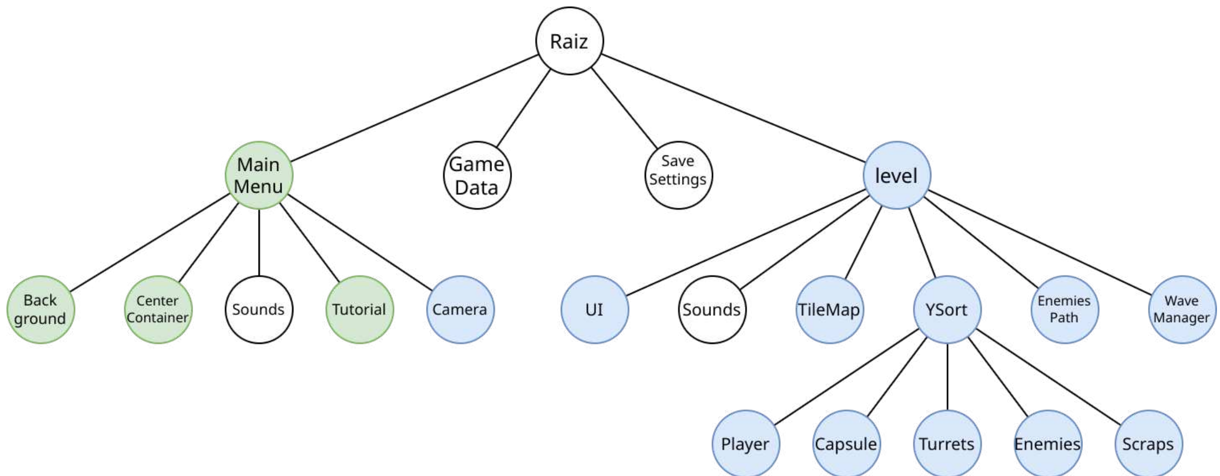

Figura 4: Estrutura do projeto em Nós.[9]

2.1.2 PORQUE A GODOT

Atualmente, os motores de jogos mais proeminentes no mercado (CB Insights, 2018) são os motores *Unreal Engine*, da empresa estadunidense *Epic Games*, cujo desenvolvimento começou em meados de 1995 e estreou em um produto comercial em 1998 (PLANTE, 2012), e o motor *Unity*, da empresa estadunidense *Unity Technologies*, apresentado em 2005 durante a conferência anual de desenvolvedores da *Apple* (Unity, 2011).

Ambos os motores de jogos apresentam qualidades e limitações cujo resultado pode levar a um determinado produto, requisitando por parte dos desenvolvedores um estudo prévio e delicado sobre qual tipo de jogo será produzido, qual a plataforma alvo e o público que se espera alcançar. O motor de jogos *Unreal Engine* conta com diversos algoritmos e técnicas atuais de computação gráfica e de cálculo de física para gerar imagens de fidelidade fotorrealista em tempo real com simulação de partículas e de corpos de maneira acurada, estando muito presente em jogos de médio ou alto orçamento, filmes, aplicações de arquitetura e *home design*, etc., e que por conta disso pode não se adequar a determinados gêneros ou tipos de jogos, uma vez que possui um elevado requisito computacional de modo a necessitar dos melhores *hardwares* em plataformas móveis e de PC

[9] Fonte: elaborado pelo autor.

(unidades centrais de processamento de múltiplos núcleos, unidade de processamento gráfico dedicada, unidades de armazenamento em estado sólido, memórias de acesso aleatório de alta velocidade) e os mais recentes consoles de jogos. O motor de jogos *Unity* permite que jogos sejam desenvolvidos utilizando a linguagem *C#* de programação e também possui algoritmos e técnicas avançadas para uma elevada qualidade gráfica e física realista, porém possui requisitos para execução inferiores aos da *Unreal Engine*, o que possibilita uma compatibilidade com uma gama maior de hardwares, gerando assim jogos acessíveis para um maior público, onde é muito popular entre os desenvolvedores de jogos independentes (chamados na indústria de *indies*) e jogos de menor escala de estúdios médios ou grandes.

Porém, embora padrões atuais da indústria do desenvolvimento de jogos, ambas possuem licenças e licenciamento (*royalties*) deveras draconianos, com restrições de usos e requerimentos de assinaturas ou mecanismos de contas online para verificação de autenticidade (Unreal Engine EULA)(Unity TOS, 2022), além de apenas permitirem o desenvolvimento nas plataformas *Mac* e *Windows*.

A *Godot* nasceu publicamente como um motor de jogos livre e aberto (Linietsky, 2014), licenciada de modo que tais liberdades são garantidas legalmente (MIT, 2022), impossibilitando que problemas presentes em *software* proprietários (STALLMAN, 2013), como direito de uso, atualização obrigatória, licenciamento e *royalties*, plataforma ou combinação de *hardware* específico para execução ou publicação, vinculação com outros serviços ou produtos, etc., venha a ocorrer, permitindo assim um desenvolvimento orgânico e seguro do jogo ou *software*.

2.2 FERRAMENTAS GRÁFICAS

Decidido previamente que o jogo desenvolvido teria como conceito artístico principal algo semelhante aos arcades dos anos 80 e 90, optamos por desenvolver toda a parte gráfica do jogo ao estilo *pixel art*. Antigamente, por conta de limitações de *hardware*, o *display* gráfico dos jogos eletrônicos era limitado a poucos *pixels*, inclusive a evolução do mercado de jogos, em seus primórdios, foi marcada principalmente por essas limitações gráficas, a chamada era 8 bits com *Nintendo Entertainment System*, seguida da era 16 bits com *Super Nintendo Entertainment System* e 64 marcado principalmente pelo lançamento do *Nintendo 64*. Atualmente, centenas de jogos são lançados todos os anos com essa baixa

resolução gráfica, porém essa escolha não parte de uma limitação de hardware, mas sim puramente pelo aspecto artístico. (BARNES, 2022)

O *Aseprite* foi a principal ferramenta utilizada para o desenvolvimento da parte gráfica. Essa ferramenta pode ser classificada de forma não-técnica como uma versão aprimorada do *Microsoft Paint*, com uma variedade de ferramentas voltada principalmente para produção de animações e imagens no estilo *pixel art*.

Na Figura 5 pode-se observar a interface gráfica padrão do *Aseprite* e parte do processo de desenvolvimento gráfico de uma das entidades presentes no jogo, a cápsula que o jogador precisa proteger:

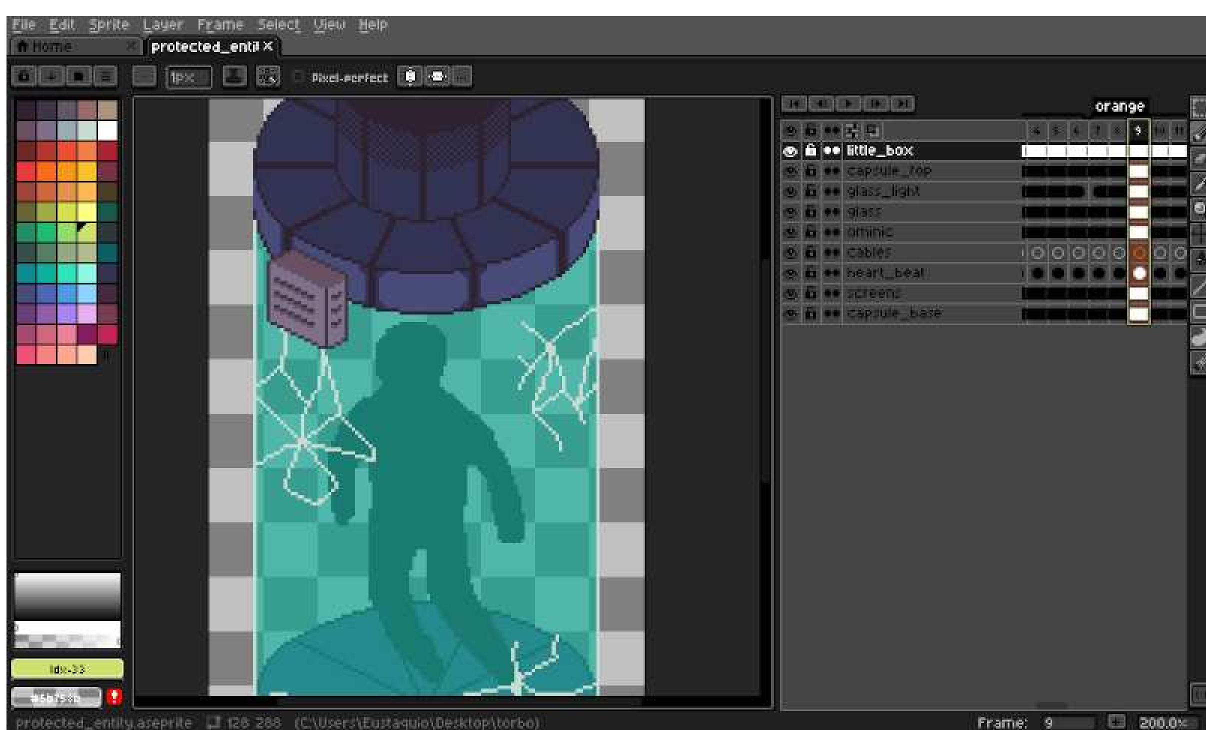

Figura 5: Interface do editor de imagem *Aseprite* com edição da entidade cápsula.[10]

[10] Fonte: elaborado pelo autor

3 GAME DESIGN

A definição do *game design* é a parte mais importante e elementar no desenvolvimento de jogos eletrônicos. No *game design*, o desenvolvedor vai elaborar toda arquitetura de relações entre as diferentes entidades que compõem o jogo, quais serão essas entidades e quais são os objetivos e propósitos delas. Além dessa visão mais específica relacionada às entidades do jogo, é nessa parte também que é definido o que o jogo se propõe a ser, quais são as intenções do desenvolvedor com o jogo e quais estímulos os jogadores devem sentir enquanto jogam. (ADAMS, 2014)

Utilizando novamente o exemplo do jogo de damas, podemos tentar traçar uma lógica e sugerir um *game design* para o mesmo. Primeiramente, de maneira concisa, um jogo de damas é um jogo de estratégia que deve ser jogado por dois jogadores com o mesmo objetivo, eliminar todas as peças do inimigo antes que o mesmo elimine as suas peças. A intenção principal é estimular um confronto cognitivo entre os dois jogadores, onde o jogador com a melhor leitura de cenário e capacidade de adaptação atinja a condição de vitória. O jogo deve estabelecer regras que sejam de fácil compreensão e, concomitantemente, justas com ambos os jogadores, evitando qualquer tipo de conflito não intencional. Após definida essa visão macro do jogo, devem ser elaboradas as diferentes entidades que compõem a estrutura funcional dele. Um exemplo de entidade que faz parte dessa estrutura do jogo de damas é o tabuleiro. Fazendo algumas ilações sobre a entidade tabuleiro, podemos assumir algumas de suas atribuições e funções no jogo, sendo uma delas relacionada a movimentação das peças no tabuleiro. O tabuleiro de damas tem a necessidade de ter uma distribuição uniforme dos espaços de movimentação das peças para ambos os jogadores no início do jogo, caso o espaço de movimentação seja diferente, e de alguma forma isso traga alguma vantagem prévia para um dos jogadores, a entidade tabuleiro vai infringir uma das regras estabelecidas pelo anteriormente pelo sugerido *game design* do jogo de damas. Esse exemplo serve para ilustrar o quão complexo pode ser a elaboração do *game design*, até mesmo em um jogo relativamente simples como damas.

Apresentadas as definições anteriores, podemos discorrer melhor acerca do *game design* do jogo desenvolvido neste trabalho de graduação. O jogo se propõe a ser um *tower defense*, ou seja, todas as regras e definições agrupadas nessa classificação de jogo fazem parte do *game design* e devem ser respeitadas

para evitar a descaracterização dele. A intenção principal do *game design* é que o jogo se mantenha um *tower defense*, porém com mecânicas que aumentem a interatividade entre o jogador e o jogo na tentativa de manter o jogador engajado ativamente durante toda a *gameplay*[11]. São duas as mecânicas de engajamento principais, aplicadas à fórmula clássica de um *tower defense*, que foram elaboradas e aplicadas no jogo.

A primeira, referente a classificação “em tempo real”, é a de unir a divisão entre os dois tempos característicos de um *tower defense* baseado em turnos: o tempo de preparo, onde o jogador elabora sua estratégia e posiciona suas defesas, e o tempo da invasão, onde o jogador, de maneira inerte, observa se a sua estratégia foi bem-sucedida ou não. No jogo em questão, sendo ele um *tower defense* em tempo real, esses dois tempos são unificados com a pretensão de aumentar o dinamismo durante o jogo, criando assim uma sensação de urgência no jogador. Ao mesmo tempo que ele precisa planejar e executar suas estratégias, as entidades inimigas vão invadir e tentar cumprir seu objetivo de destruir a entidade a ser protegida.

A segunda mecânica de engajamento aplicada, essa indubitavelmente mais relevante, é a de retirar a onipresença do jogador, essa sendo uma mecânica muito característica em todos os jogos classificados como *tower defense*, independentemente de serem do estilo “em tempo real” ou “baseado em turnos” (MOSS, 2017). No jogo, ao invés do jogador interagir e observar o cenário de maneira onipresente, ele assume o papel de uma entidade participante no jogo, no caso, essa entidade é um engenheiro de barba loira que tem a capacidade de se mover livremente pelo cenário e a habilidade de construir torretas que são capazes de impedir o avanço das entidades inimigas. A intenção principal dessa mecânica é a de aumentar a interatividade do jogador com o jogo, dando a ele o controle de uma das entidades durante o decorrer de toda *gameplay*.

Essas duas mudanças no *game design* trazem novas limitações para o jogador, mas elas também aumentam consideravelmente o desafio e o engajamento do mesmo com o jogo, pois toda interação do jogo precisa ser feita através da entidade a qual o jogador controla. O jogador não posiciona as torretas, recarrega a munição delas ou se desloca pelo cenário, ele precisa controlar o engenheiro para

---

[11] Tradução direta: jogo-jogar. É o momento que o jogador e o jogo trocam interações. O jogador dando entradas com o controle e o jogo dando respostas em tempo real.

que ele execute tais funções. Isso dentro do modelo de estratégia em tempo real, modelo esse que força o jogador a se adaptar a novos cenários constantemente.

3.1 GAMEPLAY LOOP

O *gameplay loop*[12] é o conceito mais importante a ser definido durante a elaboração do *game design*, nele é definido o ciclo de ações e eventos que o jogador fará durante o decorrer de todo o jogo. (KRAMARZEWSKI, 2018) Usando novamente o exemplo do jogo de damas, podemos identificar um ciclo padrão, o *gameplay loop*, que ocorre durante toda a partida: o jogador move uma peça no tabuleiro, em seguida, o adversário move uma peça no tabuleiro, e finalmente, o controle volta para o primeiro jogador que deve mover uma peça no tabuleiro, após esses eventos, o ciclo se reinicia até que a partida acabe.

Usando de base o *game design* clássico de um *tower defense*, definimos o seguinte gameplay loop, Figura 6, para "*Robots are trying to destroy my capsule. I hate them*":

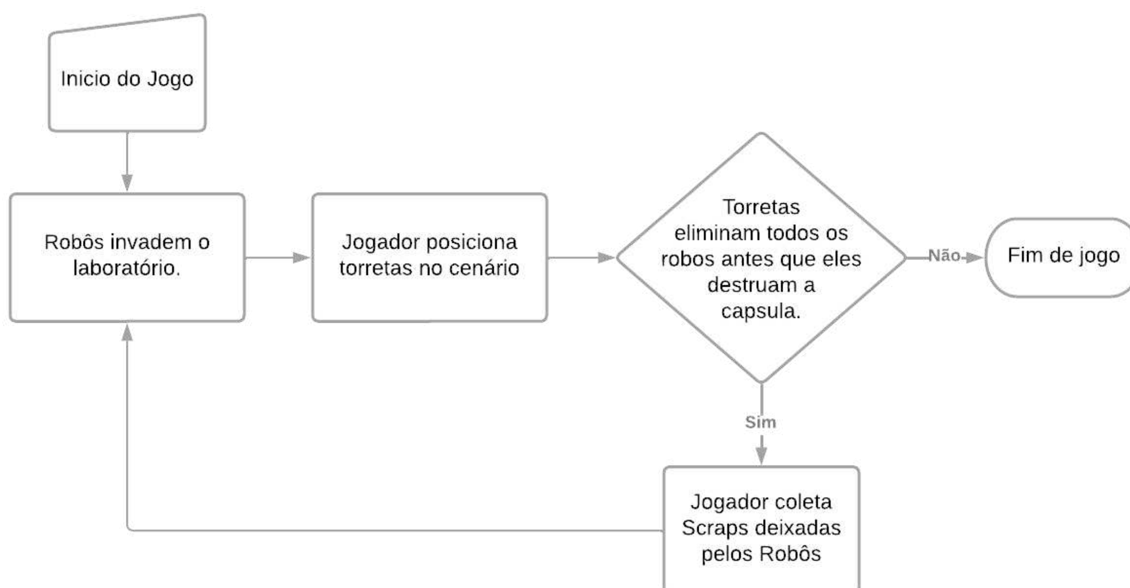

Figura 6: Fluxograma com todos os processos que compõem o *gameplay loop*.

Novamente, se tratando de um jogo de estratégia em tempo real, todos os elementos que compõem o *gameplay loop*, ilustrado na Figura 6, acontecem em

[12] Tradução direta: Ciclo de jogo-jogar. É o ciclo padrão que se repete durante a *gameplay*.

simultâneo. Ao mesmo tempo em que os robôs invadem o cenário, o jogador precisa posicionar as torretas e coletar as *scraps* deixadas pelos robôs.

3.2 FATOR REPLAY

Em suma, o fator *replay*[13] de um jogo se refere a capacidade que ele tem de instigar o jogador a jogá-lo novamente. Como regra, todas as técnicas que contribuem para o aumento do fator *replay* estão relacionadas a quantidade de variáveis presentes no *game design*. (KRAMARZEWSKI, 2018)

Independentemente do jogo, ter um fator *replay* alto é sempre preferível, porém deve-se sempre analisar o *game design* previamente estabelecido pelos desenvolvedores. Em alguns jogos, um alto fator *replay* pode ser considerado, não apenas uma qualidade positiva, mas sim uma necessidade. Essa necessidade de um alto fator *replay* se faz presente no jogo desenvolvido neste trabalho de graduação, como a intenção dele é simular um arcade antigo, onde não existia a possibilidade de manter seu progresso salvo, o fator *replay* alto tem muita extrema importância para aumentar o “tempo de vida” do jogo.

A principal ferramenta elaborada para aumentar o fator *replay* no jogo desenvolvido é o sistema de aprimoramentos das torretas, ilustrado na Figura 7, que ocorre a cada dois ciclos do *gameplay loop*. Nesse sistema, o jogador recebe, aleatoriamente e de uma lista pré-determinada, três opções de aprimoramentos que afetam todas as torretas presentes no cenário. Como essa é uma escolha aleatória, o jogador nunca tem certeza de que opções ele vai ter disponível durante a partida, criando assim uma variação na experiência do jogador em todas as partidas subsequentes. O sistema estimula o jogador a querer jogar de novo, pois ele vai imaginar que se ele tivesse escolhido diferentes opções de aprimoramento, talvez ele alcançaria uma pontuação melhor ao final do jogo.

[13] Tradução direta: reproduzir novamente.

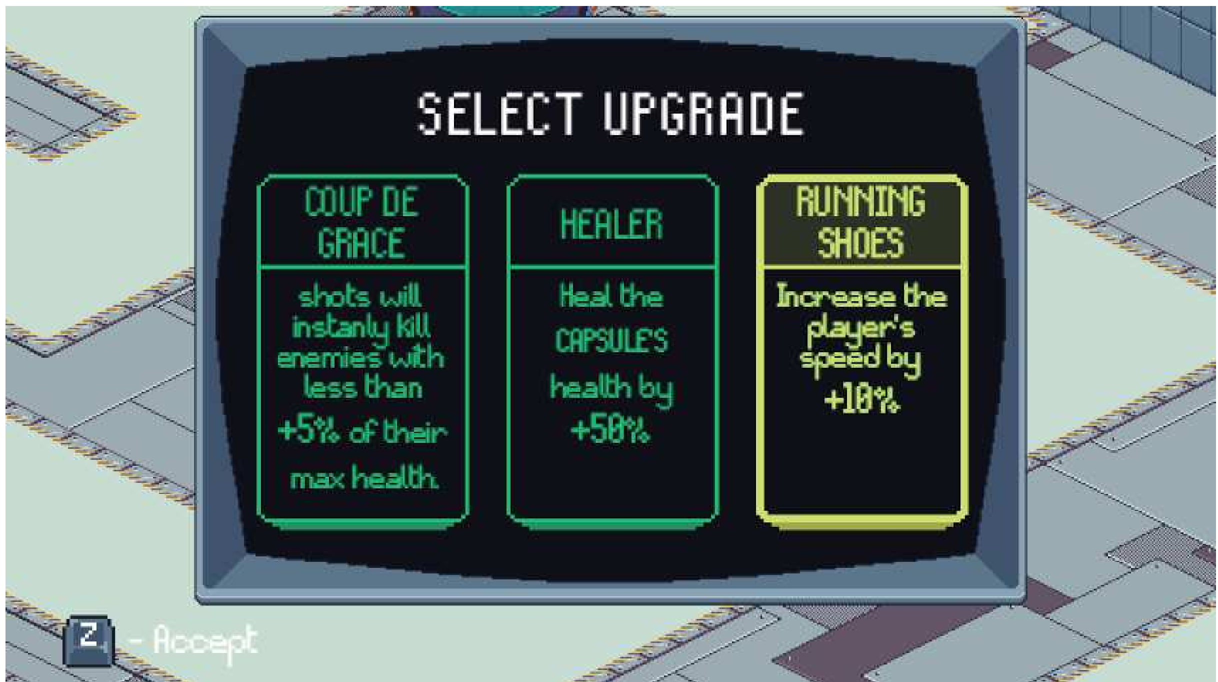

Figura 7: Interface gráfica do sistema de aprimoramento das torretas.

Não sempre o fator *replay* está relacionado a quantidade de variáveis, em muitos casos, estímulos visuais também são efetivos. Um exemplo de estímulo visual no jogo desenvolvido é a pontuação final mostrada para o jogador ao final do jogo, ilustrada na Figura 8. Quando o jogador vê, de maneira clara, dados sobre o seu desempenho na partida, isso pode estimulá-lo a tentar superar sua pontuação.

Figura 8: Interface gráfica da tela de game over.

3.3 BALANCEAMENTO

O balanceamento é um aspecto mais técnico do que teórico no game design de um jogo. Sua principal função é estabelecer uma certa “harmonia” nas relações das diferentes entidades que compõem o jogo. Essa harmonia é referente ao propósito de cada entidade e ao game design estabelecido previamente, o objetivo é fazer com que todas as entidades cumpram seu propósito da melhor maneira possível, visando sempre melhorar a experiência e a diversão do jogador com o jogo.

Um exemplo prático de balanceamento no jogo é a relação entre todos os aprimoramentos disponíveis na lista que compõem o sistema de aprimoramento, visto na Figura 7. Se algum desses aprimoramentos tivesse um impacto maior e, que de alguma forma, desse uma vantagem incontestavelmente melhor que os demais aprimoramentos, o jogador ficaria inclinado a sempre escolher esse aprimoramento, causando impacto negativo no fator replay, já que a variação entre as escolhas de aprimoramento foi drasticamente diminuída.

4 DESENVOLVIMENTO DAS ENTIDADES

Neste capítulo serão apresentadas as entidades que compõem o jogo: O Jogador, a entidade controlada pelo próprio jogador; A Torreta, entidade automática cuja função é neutralizar os invasores inimigos; Os Inimigos, entidades que invadem o cenário cujo objetivo é alcançar a cápsula para destruí-la;

4.1 JOGADOR

Como forma de interação entre o jogador e o jogo, o usuário controla a entidade "jogador" pelo cenário utilizando de controladores de jogo (*joysticks*) ou por meio do teclado, utilizando quatro botões de movimento (que ao serem pressionados em conjunto realizam a movimentação em oito direções), quatro botões de ação e dois de menu. O jogo detecta automaticamente qual o tipo de entrada o jogador está utilizando, e apresenta teclas de dica referentes ao tipo de controle.

A entidade jogadora é quem realiza as funções que levam o objetivo do jogo às condições de avanço ou derrota, e é dividida em dois estados principais, desses cada característica é baseada na situação do jogo. O primeiro estado é o chamado "básico", forma a qual o jogo se inicia e onde o jogador detém maior liberdade de movimentação e acesso às outras torretas do cenário. O segundo estado, atingido no cumprimento de condições necessárias é o chamado "modo de construção de torreta".

A Figura 9 apresenta o modelo da entidade jogador nesses dois estados em todas as posições possíveis dentro do ambiente isométrico do jogo:

Figura 9: Representação gráfica da entidade jogador nas oito direções.[14]

[14] Fonte: elaborado pelo autor.

4.1.1 ESTADOS: BÁSICO E MODO CONSTRUÇÃO DE TORRETA

No estado básico o jogador é apresentado como um personagem simples, que possui animação nas 8 direções de movimento no plano isométrico e pode se locomover livremente pelo cenário em sua velocidade normal ou de corrida, sendo este o único estado em que é possível interagir com as outras torretas presentes no cenário (para recarregar a munição ou mudá-las de lugar), e para realizar a construção de uma nova torreta, verificando se foram coletadas peças de robôs (chamadas de *scraps*) suficientes para satisfazer a condição de mudança de estado. Se todas as condições forem satisfeitas, ocorre a mudança de estado para o modo de construção de torreta.

Além de ativamente construir uma nova torreta, a outra forma de o jogador mudar para o estado "modo de construção" é na interação com alguma torreta presente no cenário para movê-la. Nesta circunstância, o objeto "jogador" se comunica diretamente com o objeto "torreta", inferindo na posição global e em outras variáveis da torreta para gerar a mudança de lugar no mapa.

O fluxograma, ilustrado na Figura 10 descreve a forma como ocorrem as transições de estados de acordo com a situação em que o jogador encontra-se:

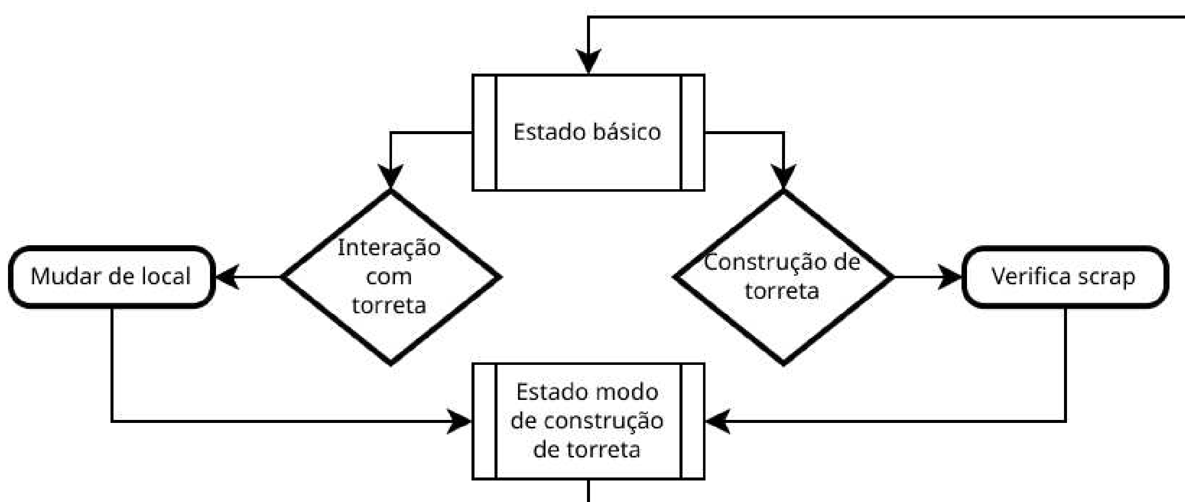

Figura 10: Fluxograma mostrando os estados do jogador.[15]

Quando o modo de construção de torreta é iniciado, o jogador locomove-se mais devagar para aumentar sua precisão de movimento e apresenta de retorno visual o personagem segurando em suas mãos uma torreta enquanto

[15] Fonte: elaborado pelo autor.

onde é exibido a sua frente (o lado que o personagem estiver faceando) a posição dela no cenário acompanhado de uma malha no chão (*grid*) como exibido na Figura 11, de modo que essa malha informa se a área onde o jogador se encontra é válida para construção, uma vez que não é possível criar torretas no caminho dos inimigos ou dentro do campo de outras torretas presentes no mapa. Caso a posição seja válida e o jogador a confirme, o modo de construção de torreta inicia uma função para instanciar um novo objeto "torreta" no cenário, retornando o objeto "jogador" ao estado básico.

Figura 11: Retorno visual do personagem no modo de construção de torreta.[16]

A função que instancia novas torretas faz o carregamento da Cena referente a nova torreta e a adiciona como filho na Cena principal onde o jogo ocorre dentro do Nó específico de torretas.

Se o estado do modo de construção for resultado da interação do jogador com alguma torreta presente no cenário, a função a ser chamada é a referente ao movimento de torretas. Esta função acessa a posição global atual da torreta para modificá-la de acordo com a posição em que o jogador vai movê-la e a impede de continuar ativa no jogo enquanto for transferida de localidade. O modo como as torretas do cenário são movidas resume-se a ocultar o objeto "torreta" (deixá-la invisível) do jogador e a desativá-la para que não continue atirando nos inimigos caso algum entre em seu campo de visão. Assim que o jogador decide colocá-la no cenário ou cancelar a ação de mover, a torreta volta a ser visível e é reativada.

O fluxograma ilustrado na Figura 12 demonstra como ocorre o tratamento interno do "modo de construção de torreta" para as duas condições possíveis, criar torreta ou mover.

---

[16] Fonte: elaborado pelo autor.

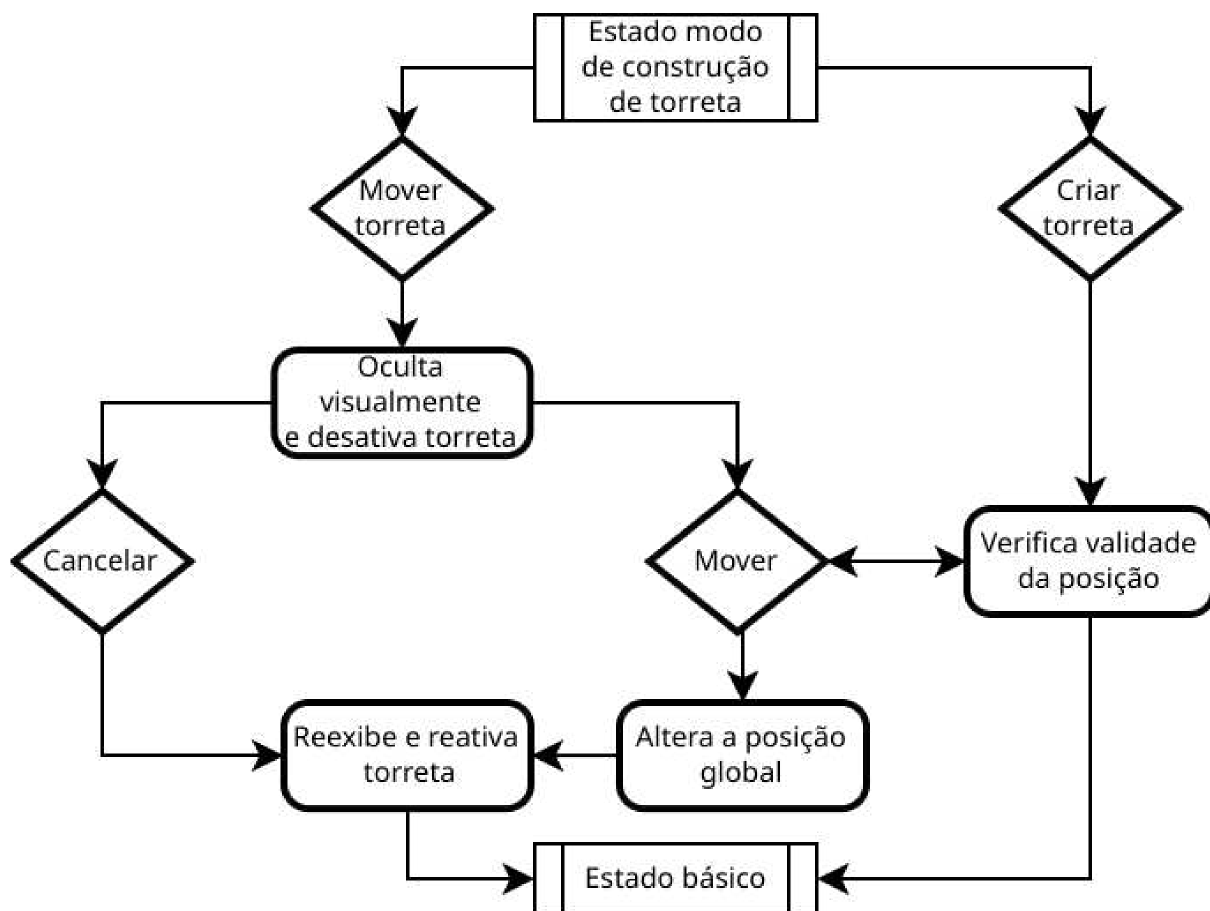

Figura 12: Tratativa do estado do modo de construção de torreta.[17]

4.1.2 FUNÇÃO DE MOVIMENTO

O objeto “jogador” é um Nó do tipo *KinematicBody2D*, um tipo de objeto que herda funções da classe *Vector2*, ambos base da *Godot*. Para a realização do movimento do personagem pelo cenário, foi implementada a função de movimento “*movement*”, programada própria para o projeto é dividida em diversas operações e chamadas de funções. Além dos vetores criados nela (como o vetor “*input_vector*”), a função realiza por meio da chamada de métodos herdados das classes *KinematicBody2D* e *Vector2*, a alteração da posição global do personagem pelo cenário, utilizando variáveis e constantes para determinar a velocidade, aceleração e a fricção do jogador, além de realizar o controle das animações do boneco.

A classe *KinematicBody2D* provê diversas funções para os chamados “corpos cinemáticos”, que são tipos especiais de objetos destinados a serem controlados pelo jogador, possuindo uma API para mover objetos enquanto executa testes de colisão. A classe *Vector2* é uma estrutura de dois elementos que podem

[17] Fonte: elaborado pelo autor

ser utilizados para representar posições no espaço 2D ou qualquer outro par de valores numéricos. Para o movimento do personagem são utilizados de *Vector2* a função “*move_toward*” e de *KinematicBody2D* a função “*move_and_slide*”. A função “*move_toward*” retorna um novo vetor, “*velocity*”, o qual se move pela quantidade fixa de delta, de modo a não ultrapassar o valor final, enquanto a função “*move_and_slide*” movimenta o objeto ao longo do vetor “*velocity*”.

O vetor “velocity” é construído pelas funções:

*velocity = velocity.move_toward(input_vector * max_speed, ACCELERATION * delta).*

A função acima é utilizada quando o objeto “jogador” está em situação de movimento, enquanto a função abaixo quando o objeto está parando:

*velocity = velocity.move_toward(Vector2.(0,0), FRICTION * delta)*

Ao final desse processo o valor “velocity” é atribuído a si próprio utilizando a função de KinematicBody2D “move_and_slide”, a qual efetivamente movimentam o personagem pelo cenário:

velocity = move_and_slide(velocity)

Toda entrada de movimento que o jogador faz com o controlador tem o valor de 1, e é atribuído ao vetor “*input_vector*”. O vetor “*input_vector*” é utilizado para armazenar valores de duas dimensões (x e y), de maneira que o eixo x seja o resultado da subtração das entradas “esquerda” e “direita” (x = direita - esquerda, -1 e 1 respectivamente) e o eixo y a subtração das entradas “cima” e “baixo” dividido por 2 (y = [baixo - cima]/2, -0.5 e 0.5 respectivamente), ao modo que no movimento diagonal (combinação do eixo x e y) o jogador se locomove em um eixo sob o ângulo de 30º, realizando o efeito isométrico, como apresentado na Figura 13.

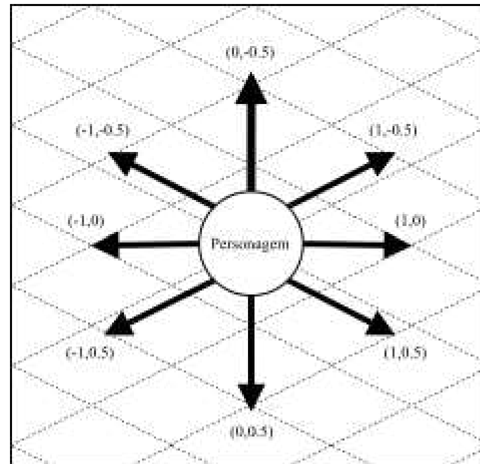

Figura 13: Movimentação realizada pelo personagem no cenário isométrico.[18]

A variável "*max_speed*" é a velocidade máxima a que o personagem pode atingir, alternando entre valores de velocidade normal, correr, segurar torreta ou segurar torreta de maneira precisa. A constante "*ACCELERATION*" define o valor de aceleração do objeto, realizando o início do movimento suave do jogador até sua velocidade final definida por "*max_speed*", e a constante "*FRICTION*" estabelece o valor de fricção que o personagem tem para parar, realizando o efeito de deslizar do jogador pelo cenário. A variável "delta" é um valor que representa a quantidade de tempo decorrido durante um quadro (*frame*), sendo velocidade unidades de distância/tempo, resulta na distância a ser percorrida durante um quadro, de modo a ser constante, ou seja, independentemente da quantidade de quadros a qual o jogo execute (exemplo 30 quadros ou 60), a distância percorrida será sempre a mesma. A função "*movement*" realiza verificações para determinar se "*input_vector*" é diferente de x=0 e y=0 (*Vector2*(0,0)), o que significa que o jogador está controlando o personagem. De acordo com essa lógica, sempre que ocorre um comando, o personagem deixa de estar na imobilidade e passa a se deslocar pelo cenário, o que é acompanhado de animação e som. Desse modo, é realizado o controle das animações, alimentando o Nó *AnimationPlayer* (responsável por criar, manter e interpolar as animações) com os valores referentes a cada tipo de animação, criando o retorno audiovisual para os jogadores acerca de como o protagonista se encontra no momento.

A função também possuí uma verificação condicional de uma variável booleana global que permite ao jogo impedir os movimentos do jogador em

[18] Fonte: elaborado pelo autor.

momentos específicos (ao posicionar uma torreta no cenário, ao ocorrer a animação da câmera, e ao fim do jogo), e uma função para a atração de *scraps* pelo jogador a partir de uma certa distância.

Para a coleta a distância das *scraps*, foi adicionado ao personagem um Nó do tipo *Area2D*, que consiste em uma área (circular, retangular, ou de outros formatos) capaz de detectar a sobreposição de nós *CollisionObject2D* (referente a colisão entre objetos e paredes), entrando ou saindo. Está *Area2D* circular do personagem, chamado de “*magnetic_field_Area*” verifica a presença de *scraps* dentro de sua área e adiciona o objeto ao vetor “*array_scrap*”.

Na função “*movement*”, é verificado se “*array_scrap*” é maior que zero para então realizar a chamada da função “*scrap_magnetic_field*”. Nesta função, um laço do tipo for do tamanho do vetor “*array_scrap*” realiza a alteração da posição global do objeto “*scrap*” pelo cenário para a posição do personagem, por meio da seguinte função:

*array_scrap[n].position += (self.position - array_scrap[n].global_position) * delta * 12*

Onde “*array_scrap[n].position*” é a posição no cenário do objetivo “*scrap*” de índex n que é atribuído e somado pelas variáveis “*self.positon*”, que é a posição do jogador no cenário subtraída pela posição global do objeto “*scrap*” de índex n multiplicado pela velocidade a qual se dá o movimento do objeto até o jogador (na velocidade de 12) e pelo valor de delta para que a movimentação ocorra de maneira fluida durante os quadros do jogo em que o objeto “*scrap*” está em cena.

4.2 TORRETA

A torreta é a entidade que o jogador constrói para realizar a defesa da cápsula contra os robôs inimigos que surgem no cenário a cada rodada. É um objeto imóvel capaz de atingir alvos em qualquer direção ao seu redor desde que esteja dentro de seu campo de visão. Pode ter a sua posição no mapa alterada pelo jogador a qualquer momento como forma de mudança de estratégia, e necessita de recarga após seu compartimento de munição chegar a zero.

A torreta possui três estados de funcionamento; o estado inativo, sendo o modo em que se mantém quando não há nenhum inimigo no campo de visão, indicado ao jogador por apresentar em sua base uma luz na cor verde; o estado de

alerta, decorrente da presença de algum inimigo em seu campo de visão, o que inicia o temporizador interno de disparo, e é visualmente representado pela cor vermelha na luz de sua base; e o estado de tiro, que ocorre após o fim do temporizador do estado de alerta, onde efetivamente infere dano ao inimigo, indicando como retorno audiovisual ao jogador o som de disparo e a animação do tiro. Após o estado de tiro a torreta alterna para o estado de alerta, e caso não haja nenhum inimigo próximo ou a munição tenha acabado, ela retorna para o estado inativo. O fluxograma da Figura 14 ilustra o processo de transição dos estados.

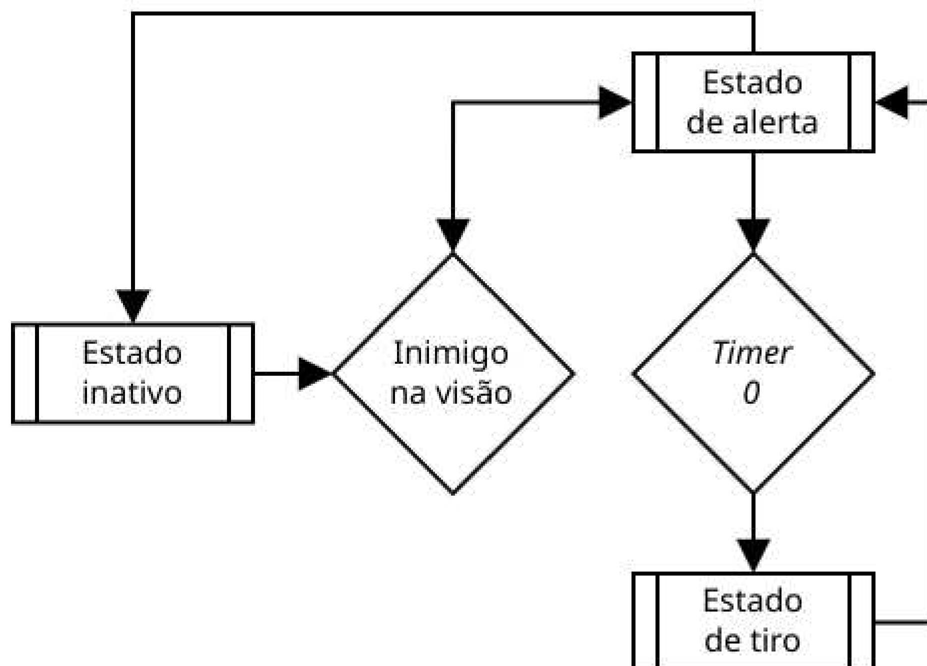

Figura 14: Estados de funcionamento da torreta.[19]

Na presença de algum inimigo no campo de visão da torreta, seu estado muda de inativo para o de alerta, iniciando seu temporizador. Ao chegar a zero, ocorre a intercalação entre o estado de alerta para o estado de tiro, onde a torreta acessa a função de dano do inimigo que está em sua mira e reproduz a animação referente ao disparo, processo que se repete entre estado de alerta e estado de tiro até que o inimigo saia do campo de visão, ou seja, efetivamente destruído. Caso o inimigo focado não esteja mais presente, o próximo da fila é selecionado, de modo que esse comportamento se suceda até não haver mais nenhum outro objeto na fila. A fila é composta pela ordem em que as entidades entram no campo de visão,

[19] Fonte: elaborado pelo autor.

resultando em situações de inimigos atravessando na frente da torreta sem qualquer tipo de dano devido a posição e o momento em que entraram no campo de visão, onde fisicamente estão próximos ao inimigo focado, porém não estão em seguida na fila.

A Figura 15 apresenta o modelo da entidade torreta no estado inativo (verde) e no estado de alerta (vermelho), em todas as posições possíveis dentro do ambiente isométrico do jogo:

Figura 15: Representação gráfica da entidade torreta nas oito direções.[20]

4.3 INIMIGOS

Os inimigos são a forma de manter o *gameplay loop* em funcionamento, de modo que seu propósito é destruir a cápsula a ser protegida, fazendo assim com que o jogador tenha de objetivo estratégias para posicionar torretas pelo cenário que irão eliminá-las. Sem inimigos, a cápsula não está em risco, o que tira o motivo do jogador em construir torretas e por consequência a necessidade de jogo.

A característica clássica dos inimigos em jogos do gênero *tower defense* é que seu objetivo é de apenas percorrem um caminho até o final, o que neste trabalho foi expandido para a necessidade de atingir a cápsula, causando-a dano e eventualmente sua destruição, porém sem interagir desta forma com o jogador ou as torretas.

4.3.1 CARACTERÍSTICAS BASE

Por decorrência do ritmo do jogo ser acelerado, inicialmente não é muito viável ao jogador dedicar tempo estudando a forma de ataque dos inimigos. Por este motivo, os inimigos são robôs com aparência baseada em bombas marítimas, o que

[20] Fonte: elaborado pelo autor.

permite ao jogador instintivamente raciocinar que a forma de dano em que os inimigos causam a cápsula é por meio da aproximação para realizar uma explosão.

Foram criados três tipos de inimigos para aumentar a diversidade e a dinâmica dos tipos de estratégia com a qual o jogador precisa ter, que se diferenciam visualmente por cores e tamanho, velocidade, quantidade de vida (levam mais ou menos danos das torretas) e por dano causado à cápsula. De acordo com a rodada em que o jogador estiver, esses diferentes tipos de inimigos começarão a surgir em quantidade variadas pelo cenário, em ordens ou sequências que os permita invalidar a estratégia do jogador para atingirem seu objetivo final.

Os inimigos internamente são chamados de *Tropper*, *Scout* e *Spartan*, de modo que o *Tropper* possui velocidade, vida e dano mediano, e é o primeiro inimigo que o jogador enfrenta. O *Scout* é o mais rápido, e por esse motivo sua vida e seu dano são menores. O *Spartan* tem a mesma velocidade que o *Tropper*, porém com muito mais vida e causa mais dano. Devido a essas características, a ordem com que eles surgem no cenário permite que em determinadas situações um proteja o outro, possibilitando que algum escape a mira da torreta. Na Figura 16 é ilustrado os respectivos modelos de inimigos:

Figura 16: Representação gráfica das entidades inimigos.[21]

Enquanto o jogador possui liberdade total para se mover pelo cenário, os inimigos locomovem-se somente pelas estradas que formam caminhos até chegar a cápsula. Como é comum do gênero de *tower defense*, esses caminhos são pré estabelecidos e são montados de modo a ter uma estrutura próxima a de um

[21] Fonte: elaborado pelo autor.

labirinto, o que permite ao jogador uma certa previsibilidade para montar sua estratégia de defesa e garantia de que suas torretas serão eficazes.

Para a realização do movimento dos inimigos pelo cenário, são utilizados os Nó base da *Godot Path2D* e *PathFollow2D*. Os caminhos são feitos com o Nó *Path2D*, que cria uma espécie de trilho por onde os inimigos realizarão o movimento. Os inimigos, ao serem instanciados no mapa, são ligados ao Nó *PathFollow2D*, cuja função é locomover o objeto conectado a ele por esse trilho, utilizando a propriedade *offset*. Essa propriedade é a distância do caminho em *pixels*, que ao ser manipulado aproxima o objeto “inimigo” ao destino a uma determinada velocidade e aceleração.

4.3.2 FUNÇÕES PRINCIPAIS

Todos os três tipos de inimigos herdam um código central para realizar seu comportamento, o qual é dividido em três principais funções: movimento pelo cenário, dano recebido e destruição.

A função movimento pelo cenário verifica sempre a condição da variável booleana “*died*”, que se for falsa indica ao código que o inimigo ainda está vivo pelo mapa. A função então soma a posição do *offset* do Nó *PathFollow2D* ao valor da velocidade do inimigo multiplicado por delta, de modo a manter a velocidade constante, o que manipula a posição global da entidade “inimigo” pelo cenário. Quando o *offset* do *PathFollow2D* desse inimigo é igual a *1.0* (100% do caminho concluído), é acessado no código do objeto “capsula” sua função de dano, realizando o valor de dano que o inimigo produz na vida da cápsula a ser protegida, ao mesmo tempo, a variável booleana de controle “*finished_path_and_died”* torna-se verdadeira, o que executa a função de destruição a partir deste pretexto. Caso o inimigo seja destruído antes de atingir seu objetivo por meio de alguma torreta, a variável booleana “*died*” muda para o estado de verdadeiro, o que impede que o objeto continue a se mover pelo cenário, enquanto inicia a função de destruição que realiza toda a tratativa a partir da posição em que ele foi finalizado.

A função de dano recebido reduz a vida do inimigo baseado no valor que cada tiro que a torreta produz causa, o que verifica se o valor atual de vida desse inimigo é igual ou menor a zero, toca a animação de dano e atualiza a barra de vida. Caso o valor de vida desse inimigo for inferior ou igual a zero, a função de destruição é executada. A função de dano recebido também verifica se as melhorias que o jogador pode adquirir (as que se referem aos inimigos) estão ativas e quais

valores possuem, de modo que sempre que a função for executada, todas as melhorias ativas terão seus efeitos aplicados.

A função de destruição, ao ser iniciada, verifica de que modo o inimigo foi destruído. Se for em decorrência do cumprimento do objetivo (explodir na cápsula a ser protegida), serão executadas as funções de remoção do objeto “inimigo” do jogo e a animação referente será reproduzida. Caso a destruição tenha ocorrido por alguma torreta inferindo-lhe dano, a animação a ser reproduzida refere-se a essa situação, de modo que além das funções de remoção desse objeto do jogo, é adicionado o valor 1 ao contador global de quantos inimigos o jogador destruiu e é realizado a instanciação do objeto *scraps* no local onde este inimigo estava do mapa.

O fluxograma abaixo, ilustrado na Figura 17, mostra o funcionamento das funções principais:

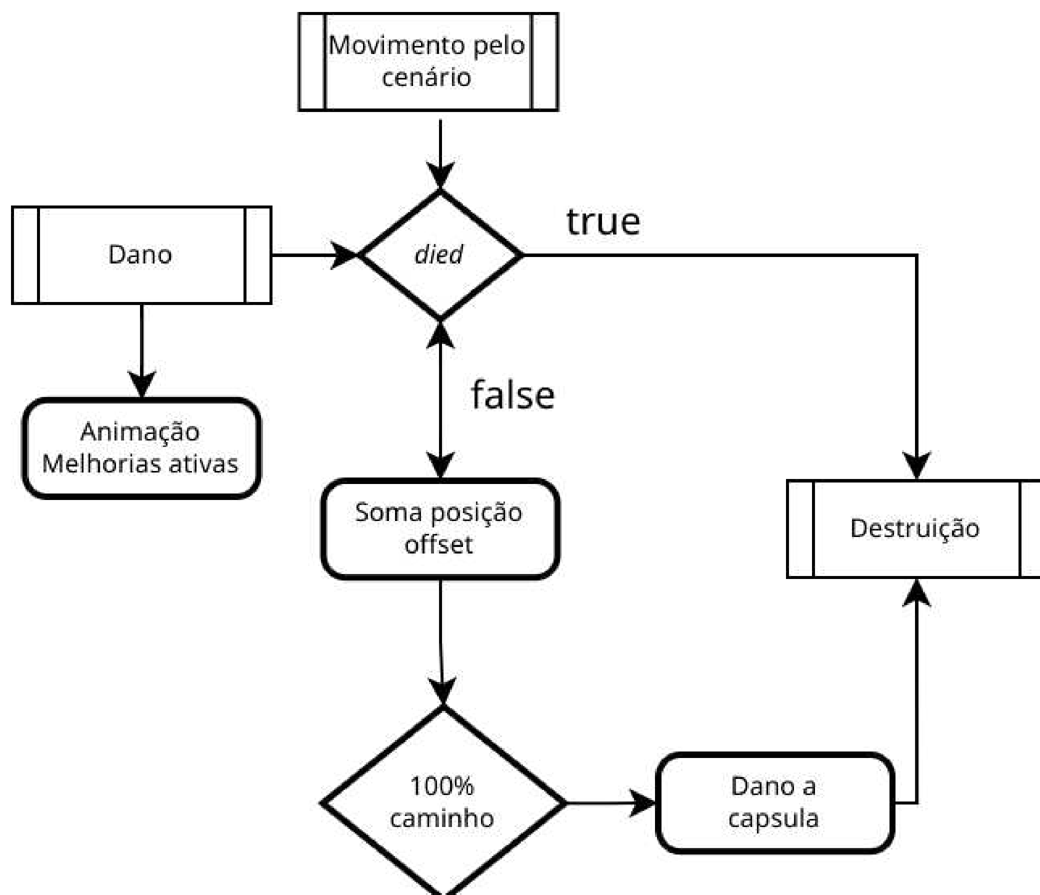

Figura 17: Interação básica entre os três estados principais.[22]

[22] Fonte: elaborado pelo autor.

5 CONCLUSÃO

A validação desse trabalho de graduação se dá no produto final. Um jogo que atende bons parâmetros de qualidade (KRAMER, 2000), que cumpri a proposta inicial de ser uma variação do gênero *tower defense* em tempo real e que limita o controle do jogador a uma das entidades participantes no jogo, mais bem descrita no **Capítulo 3**.

Uma demonstração de vídeo do jogo pode ser encontrada na seção de referências desta documentação. (EUSTÁQUIO, 2022)

5.1 IMPLEMENTAÇÕES FUTURAS

Há inúmeras melhorias que poderiam ser implementadas futuramente no jogo. Algumas melhorias simples de serem implementadas seriam, por exemplo, novos tipos de robôs, maior variedade de opções na lista de aprimoramento e mais ferramentas capazes de impedir os robôs que não as torretas. Outras mais complexas como, por exemplo, um sistema capaz de gerar um mapa aleatório a cada partida, assim aumentando drasticamente o fator *replay*, um sistema que armazena o progresso do jogador, dando a possibilidade dele parar de jogar quando quiser e retornar do ponto em que parou, um sistema de desafios que estimule o jogador a tentar cumprir determinados desafios além de impedir os robôs, entre outras.

REFERÊNCIAS

ADAMS, Ernest. **Fundamentals of Game Design**, Third Edition, USA: New Riders, 2014.

ARM. **Gaming Engines**, <https://www.arm.com/glossary/gaming-engines> Acessado em Novembro de 2022.

AVERY, Phillipa. **Computational Intelligence and Tower Defence Games**, Department of Computer Science and Engineering, University of Nevada, Reno, USA, 2011 <http://julian.togelius.com/Avery2011Computational.pdf> Acessado em Novembro de 2022.

BARNES, Sara. **How the Humble Pixel Became a Building Block To Groundbreaking Art**, 2022. <https://mymodernmet.com/what-is-pixel-art/> Acessado em Novembro de 2022.

CB Insights. **The $120B Gaming Industry Is Being Built On The Backs Of These Two Engines**, 2018 <https://www.cbinsights.com/research/game-engines-growth-expert-intelligence/> Acessado em Novembro de 2022.

EUSTÁQUIO, Lucas. **Robots are trying to destroy my capsule. I hate them - Gameplay de Demonstração**, 2022 <https://youtu.be/f9WlrJTFRtg> Acessado em Dezembro de 2022.

GNU. **What is Free Software?**, 2022 <https://www.gnu.org/philosophy/free-sw.en.html> Acessado em Novembro de 2022.

Godot Docs. **Console support in Godot**, 2022 <https://docs.godotengine.org/en/latest/tutorials/platform/consoles.html> Acessado em Novembro de 2022.

Godot Docs. **Introduction to Godot development**, 2018 <https://docs.godotengine.org/en/3.0/development/cpp/introduction_to_godot_development.html> Acessado em Novembro de 2022.

Godot Engine. **Godot Features**, 2022 <https://godotengine.org/features> Acessado em Novembro de 2022.

GOMES, Nestor. **Godot Engine, el Motor de Videojuegos Open Source más completo**, 2017. <https://www.headsem.com/godot-engine-el-motor-de-videojuegos-open-source-mas-completo/> Acessado em Novembro de 2022.

KRAMARZEWSKI, Adam. **Practical Game Design: Learn the art of game design through applicable skills and cutting-edge insights**, USA, Packt Publishing, 2018.

KRAMER, Wolfgang. **What Makes a Game Good**, 2000. <http://www.thegamesjournal.com/articles/WhatMakesaGame.shtml> Acessado em Novembro de 2022.

LINIETSKY, Juan. **First public release!**, 2014 <https://godotengine.org/article/first-public-release> Acessado em Novembro de 2022.

MIT. **The MIT License (MIT)**, 2022 <https://mit-license.org/> Acessado em Novembro de 2022.

MOSS, Richard. **Build, gather, brawl, repeat: The history of real-time strategy games**, 2017 <https://arstechnica.com/gaming/2017/09/build-gather-brawl-repeat-the-history-of-real-time-strategy-games/> Acessado em Novembro de 2022.

PLANTE, Chris. **Better with age: A history of Epic Games**, 2012 <https://www.polygon.com/2012/10/1/3438196/better-with-age-a-history-of-epic-games> Acessado em Novembro de 2022.

STALLMAN, Richard. **Free Software Is Even More Important Now**, 2013 <https://www.gnu.org/philosophy/free-software-even-more-important.html> Acessado em Novembro de 2022.

Unity TOS. **Unity Terms of Service**, 2022 <https://unity.com/legal/terms-of-service> Acessado em Novembro de 2022.

Unity. **Unity Technologies Celebrates Six Years of Continual Leadership and Innovation in Game Development**, 2011 <https://unity.com/our-company/newsroom/unity-technologies-celebrates-six-years-continual-leadership-and-innovation> Acessado em Novembro de 2022.

Unreal Engine EULA. **Unreal Engine End User License Agreement**, <https://www.unrealengine.com/en-US/eula/unreal> Acessado em Novembro de 2022.

VALENCIA-GARCÍA, Rafael, et al. **Technologies and Innovation: Second International Conference**, CITI 2016, Guayaquil, Ecuador, November 23-25, 2016, Proceedings, página 146, 2016 <https://www.google.com.au/books/edition/Technologies_and_Innovation/bZZyDQAAQBAJ> Acessado em Novembro de 2022

WIRTZ, Bryan. **Shaders in Game Design: Origin, Basic Design Types, and How to Create Your Own**, 2022 <https://www.gamedesigning.org/learn/shaders/> Acessado em Novembro de 2022.