SCC0204/SCC0504 – Programação Orientada a Objetos Prof. José Fernando Rodrigues Júnior – 1o./2025 Projeto de Jogo do tipo puzzle

Aluno: Felipi Adenildo Soares Sousa

Projeto Evo: Um Jogo de Evolução Procedural

1. Sobre o Projeto

Evo é um jogo 2D com visão de cima (top-down), construído na plataforma Java utilizando a biblioteca Swing para a interface gráfica. O projeto explora o tema da evolução de uma forma interativa e dinâmica. Inspirado em *E.V.O.: Search for Eden* e mecânicas de jogos como *Spore* e *Pokémon*, o jogador controla uma criatura que deve sobreviver, lutar, se alimentar e evoluir através de cinco eras geológicas distintas, cada uma representada por uma fase com biomas, desafios e ecologias únicas.

O projeto usa **geração procedural**, uma técnica utilizada para criar tanto os mundos quanto as criaturas. Isso garante que cada nova partida seja um pouco diferente e aumenta a rejogabilidade. A arquitetura do jogo usa **orientação a dados**, com a configuração de cada fase (incluindo tamanho do mapa, regras de população, e estatísticas do jogador) sendo definida em arquivos de configuração .json externos. Esta abordagem permite uma prototipagem e design de níveis rápidos e flexíveis, sem a necessidade de recompilar o código-fonte para realizar ajustes de balanceamento ou de conteúdo.

2. Referências e Inspirações

O design de "Evo" é inspirado em jogos clássicos, adaptação e sistemas de RPG.

- **E.V.O.: Search for Eden (SNES)**
  - **Aplicação:** Esta é a inspiração fundamental para o **conceito central e o ciclo de gameplay (gameplay loop)**. A própria ideia do jogo reflete a jornada de E.V.O.
  - **Detalhes:** A progressão através de diferentes eras geológicas, cada uma com um tema visual e ecológico distinto (The Primordial Pool, The Sandy Shores, etc.), é baseada na estrutura de capítulos de E.V.O. O ciclo principal de **lutar com outras criaturas -> consumir os restos para ganhar pontos -> evoluir** é a nossa adaptação da mecânica de "Evo Points". Os evolutionPoints, armazenados no StatusComponent do jogador, são o recurso chave para o avanço. Quando um limiar definido em GameConstants é atingido, o GameLogicSystem aciona a criação de um PortalComponent, permitindo a transição para a próxima era.
- **Pokémon (Geração 1)**
  - **Aplicação:** A inspiração de Pokémon foi usada para o **sistema de cálculo de status** das criaturas, trazendo uma camada de profundidade e balanceamento estratégico ao combate.

    - Detalhes: Para evitar um sistema de combate onde "maior é sempre melhor", as fórmulas de stats de Pokémon Red/Blue para criar criaturas com especializações e variações individuais. \
- O SizeComponent da nossa criatura funciona como o "Nível" (Level) na fórmula, sendo o principal multiplicador de poder. \
- O ProceduralSpriteComponent.BodyType (ex: FINNED_AQUATIC, BIPED_TERRESTRIAL) define os "Base Stats" de uma espécie, garantindo que "peixes" sejam naturalmente mais rápidos e "bípedes" mais fortes, por exemplo. \
- Uma pequena variação aleatória, gerada com base em uma ivSeed no método generateStats da EntityFactory, simula os "IVs" (Individual Values). Isso garante que duas criaturas do mesmo tipo e tamanho ainda possam ter status ligeiramente diferentes, adicionando uma camada de "genética" e variedade. \
- Os StatusComponent (health, attack, defense, speed, special) são os resultados finais dessas fórmulas, criando uma base sólida e familiar para a mecânica de combate.
- **Spore**
    - **Aplicação:** foi a principal inspiração para a **geração procedural de criaturas** e para o sistema de evolução baseado em características.
    - Detalhes: A base de Spore é a ideia de que a forma de uma criatura não é fixa, mas sim um resultado emergente de suas partes e "genes". \
- O SpriteGenerator é usado para simular esta ideia. Em vez de carregar imagens estáticas, ele constrói um sprite proceduralmente, pixel por pixel, com base nos parâmetros definidos no ProceduralSpriteComponent de uma entidade. Isso nos permite criar uma diversidade visual sem a necessidade de criar centenas de assets manualmente. \
- A mecânica de dieta (EcologyComponent) e a futura implementação da árvore de habilidades (TraitComponent) imita Spore, onde as escolhas do jogador (o que comer, quais habilidades adquirir) guiam ativamente o seu caminho evolutivo.

3. Arquitetura Principal: Entity-Component-System (ECS)

O projeto é construído sobre uma arquitetura **Entity-Component-System (ECS)**. Esta escolha de design, que separa os dados (Components) da lógica (Systems), foi fundamental para alcançar a flexibilidade e a escalabilidade necessárias para as funcionalidades de geração procedural e de um sistema com orientação a dados. A ECS favorece a **composição sobre a herança**, permitindo que entidades sejam definidas por aquilo que *têm*, e não por aquilo que *são*.

- **Entity:** Um identificador único, um "contêiner" leve e sem dados.
- **Component:** Classes de dados puros que representam um único aspecto de uma entidade. Por exemplo, para tornar uma entidade inflamável, em vez de criar uma subclasse, simplesmente adiciona-se um FlammableComponent a ela. Um FireSystem então operaria sobre todas as entidades com este componente, sem se importar com o tipo da entidade.
- **System:** Classes que contêm toda a lógica do jogo, operando sobre conjuntos de entidades que possuem os componentes necessários para aquela lógica.

4. Requisitos do Projeto (Análise e Implementação)

Requisito 1: Jogo tile-based com scrolling e 5 fases.

- **Status: Concluído**
- **Implementação:** O jogo opera sobre um grid lógico definido pelo GameMap. A funcionalidade de scrolling é gerenciada pela interação entre o GamePanel, que mantém as coordenadas da câmera (cameraX, cameraY), e o RenderSystem. A cada quadro, o RenderSystem utiliza esses offsets da câmera para calcular a porção exata do mundo que deve ser renderizada na tela. A classe Main contém a lógica principal para carregar sequencialmente as 5 fases, cada uma definida por um arquivo .json distinto (level-1.json a level-5.json).

Requisito 2: Diversidade de elementos, com um novo personagem por fase.

- **Status: Concluído**
- **Implementação:** A diversidade é alcançada de forma procedural. Em vez de criar uma nova classe Java para cada novo personagem, foi definimos novos arquétipos de criaturas diretamente nos arquivos .json de cada fase. O PopulationSystem lê as biomeRules, que especificam quais tipos de criaturas (SkittishNPC, AggressiveNPC, etc.) e objetos (StaticObject) podem aparecer em cada bioma, com que densidade e com que propriedades (tamanho, velocidade, dieta). Isso nos permite introduzir novos "personagens" em cada fase de forma flexível, simplesmente editando o arquivo de texto.

Requisito 3: Personagens estáticos e animados.

- **Status: Concluído**
- **Implementação:** A distinção entre personagens estáticos e animados é gerenciada pela presença ou ausência de componentes específicos.
    - **Personagens Estáticos:** São entidades criadas pela EntityFactory (tipo "StaticObject") que possuem PositionComponent e RenderableComponent (para um sprite estático), mas **não possuem um AiComponent**. A ausência deste componente garante que o AISystem os ignore, fazendo com que permaneçam imóveis.
    - **Personagens Animados:** São os NPCs que possuem um AiComponent. O AISystem processa este componente para gerar movimento. Além disso, a flag booleana isMoving no ProceduralSpriteComponent é atualizada pelos sistemas de movimento (PlayerInputSystem, AISystem), permitindo que o RenderSystem aplique animações visuais (como o efeito "sanfona") apenas quando a criatura está de fato se movendo.

Requisito 4: Substituição do objeto "Fase" para trocar de nível.

- **Status: Concluído (com uma solução análoga e mais robusta)**
- **Implementação e Justificativa:** O requisito original, baseado em um protótipo de herança, sugere a criação de uma classe Fase que atuaria como um contêiner para uma lista de personagens. Na arquitetura do jogo, este conceito foi implementado de uma forma mais abstrata e alinhada com os princípios do design.
    - **O objeto World:** Atua como o contêiner de "personagens" (entidades e todos os seus componentes) para a fase atual. Ele é a representação completa do estado do nível.
    - **O método Main.loadLevel():** Orquestra a transição de fase. Ao invés de simplesmente trocar um objeto Fase por outro, o loadLevel() executa uma reinicialização completa e limpa do estado do jogo: ele **destrói a instância atual do World e cria uma instância nova**.
    - Este novo World é então populado do zero pelo PopulationSystem, que lê as regras do novo arquivo .json, garantindo que não haja vazamento de estado ou dados da fase anterior.
    -

Requisito 5: Salvar e Carregar o jogo com serialização.

- **Status: Concluído**
- **Implementação:** O sistema de save/load é gerenciado pela classe Main em conjunto com a SaveManager.
    - As ações são acionadas pelas teclas 'O' (Salvar) e 'P' (Carregar), que adicionam componentes de evento (SaveGameRequestComponent, LoadGameRequestComponent) ao jogador.
    - A classe Main detecta esses componentes, pausa o loop do jogo para evitar bugs, e mostra os diálogos necessários.
    - Foi criada uma classe GameState que encapsula os dados a serem salvos: o objeto World inteiro e o currentLevelNumber. Este objeto GameState implementa a interface Serializable.
    - Para permitir que o World seja salvo, todas as classes de Component e a classe Entity também foram marcadas como Serializable, garantindo que o estado completo de cada entidade possa ser gravado.
    - A SaveManager utiliza as classes ObjectOutputStream e ObjectInputStream do Java para escrever e ler o objeto GameState em arquivos .sav, cumprindo o requisito de uso de serialização de forma explícita.

Requisito 6: Drag-and-Drop de personagens de arquivos .zip.

- **Status: Pendente**
- **Plano de Implementação:** Esta é uma das próximas tarefas principais. A abordagem será a seguinte:
    1. **Criar um Utilitário de Geração de ZIP:** Será desenvolvido um pequeno programa Java separado (ou um script) com a única função de:
        - Criar um objeto EntityConfig com as propriedades de uma criatura customizada.
        - Serializar este objeto EntityConfig usando ObjectOutputStream.
        - Compactar o arquivo serializado resultante em um arquivo .zip com um nome descritivo.
    2. **Implementar o DropTargetListener:** A classe GamePanel será modificada para implementar a interface DropTargetListener, tornando-a um alvo válido para arrastar e soltar arquivos do sistema operacional.
    3. **Lógica de Processamento do "Drop":** Ao detectar que um arquivo foi solto sobre o painel, a lógica irá:
        - Verificar se o arquivo tem a extensão .zip.
        - Usar as classes ZipInputStream do Java para ler o conteúdo do arquivo em memória.

- Usar ObjectInputStream para desserializar o objeto EntityConfig a partir dos dados lidos.
- Usar a EntityFactory para criar uma nova entidade no World com base na configuração desserializada. A posição inicial da entidade será definida pelas coordenadas do mouse no momento do "drop".

5. Como Executar o Projeto

1. **JDK (Java Development Kit)** instalado (versão 21 ou superior).
2. Abra o projeto na IDE NetBeans (ou outra IDE com suporte a **Maven**). A IDE irá detectar o arquivo pom.xml e baixar automaticamente a dependência **Gson**.
3. Localize e execute a classe game.evo.Main.java.
4. O menu principal do jogo aparecerá, permitindo que você inicie uma nova partida ou continue um jogo salvo.