BehaviorStates: Uma Arquitetura de Comportamento Next-Gen para Godot

> **Visão:** Prover um Framework de Comportamento nível AAA, orientado a dados, que rivalize com os padrões da indústria (como o GAS da Unreal), permitindo que Designers e Programadores construam sistemas reativos complexos sem acoplamento de código.

Os Pilares da Arquitetura

O sistema é construído sobre quatro pilares customizados, orquestrando uma separação de responsabilidades que garante escalabilidade.

Pilar	Componente	Descrição
O Cérebro	`Behavior` (Node)	O orquestrador de intenção. Faz a ponte entre o Input Bruto e o Contexto Semântico.
A Engine	`Machine` (Node)	O Executor e Interpretador. Processa decisões O(1) e executa funções especializadas (`apply_jump` , `apply_damage`).
O DNA	Resources	Comportamento é Dado. Mutável, trocável e extensível sem recompilação.
A Bancada	Editor Panel	Uma IDE totalmente integrada dentro da Godot. Visual, intuitiva e livre de código.

🚀 O Roadmap para o Nativo (Vision)

1. **Fase 1 (GDScript Plugin):** Prototipagem rápida e adoção pela comunidade. Foco na DX e estabilidade da API.
2. **Fase 2 (Rust GDExtension):** Core (Machine e Algoritmos) reescritos em Rust para performance bare-metal.

3. **Fase 3 (Godot Native):** Propor como módulo oficial C++, preenchendo a lacuna de uma State Machine visual nativa.

1. A Bancada (Integrated Workspace)

O Painel `BehaviorStates` transforma a Godot em uma **IDE especializada** para criação de comportamentos. É adicionado como um **Bottom Panel** via `EditorPlugin.add_control_to_bottom_panel()`.

1.1. Aba: Library (Biblioteca de Assets)

A **Library** é o gerenciador central de todos os Resources `.tres` do projeto.

Feature	Descrição
Tree View	Hierarquia organizada em categorias: **Systems** (Config, CharacterSheet, Inventory), **Composes** (com States filhos), e **Unlinked** (States órfãos agrupados por pasta).
Drag & Drop	Arraste qualquer asset diretamente para o Inspector ou para slots de outras abas.
Filtro de Busca	Campo de texto para filtrar assets por nome instantaneamente.
Menu de Contexto	Clique Direito abre o Resource no **Editor** (Blueprint). Clique Esquerdo abre no Inspector padrão da Godot.
Botão "+ Novo"	Redireciona para a aba **Factory** para criar novos Resources.

1.2. Aba: Editor (Blueprint Visual)

O **Editor** é o inspetor especializado para visualização e edição de Resources em forma de grafo.

Feature	Descrição
GraphEdit	Os Resources são exibidos como `GraphNode` s interconectados.
Campos Dinâmicos	Cada tipo de Resource (`State` , `Item` , `Skill` , etc.) renderiza campos de edição inline baseados em suas propriedades exportadas.
Blocos Lógicos	States possuem blocos visuais para configurar **Filters** (requisitos de entrada), **Actions** (o que fazer) e **Triggers** (reações a eventos).
Conexões	Para `SkillTree` , conectar nós visualmente define a propriedade `prerequisites` automaticamente. Para `Compose` , conectar States os adiciona às listas.
Save/Discard	Rodapé com indicador de "dirty state" e botões para salvar ou descartar alterações.

Blocos do Editor (Block System)

Os Resources complexos são compostos por **Blocos** modulares:

Bloco	Aplicável a	Descrição
`FilterBlock`	State	Define os **Requisitos de Entrada** (ex: `Motion: RUN` , `Physics: GROUND`).
`ActionBlock`	State	Define **O Que Fazer** (velocidade, dano, animação, spawn de projétil).
`TriggerBlock`	State	Define **Reações** a eventos (ex: `on_hit: flinch` , `on_duration_end: idle`).
`RequirementBlock`	Skill	Define os **Pré-requisitos** para aprender a Skill (Level, Atributos, outras Skills).
`UnlockBlock`	Skill	Define o que a Skill **Desbloqueia** (States, Items, Buffs).
`ModifierBlock`	Item	Define **Modificadores de Stats** quando o item está equipado.
`PropertyBlock`	Item	Define propriedades gerais (Stackable, Durability, Consumable).

1.3. Aba: Factory (Wizard de Criação)

A **Factory** é o assistente para criar novos Resources com presets inteligentes.

Feature	Descrição
Seleção de Tipo	Escolha entre `State`, `Item`, `Skill`, `Compose`, `SkillTree`, `CharacterSheet`, `Inventory`.
Presets	Para `State`: presets como "Idle", "Walk", "Attack Light", "Dash". Para `Item`: "Consumable", "Weapon", "Equipment".
Nomeação e Path	Define o nome do arquivo e o diretório de destino. Sugestões automáticas baseadas no tipo.
Criação Automática	O Resource é criado com propriedades pré-configuradas baseadas no preset selecionado.

1.4. Aba: Grimório (Documentação Integrada)

O **Grimório** é um visualizador de Markdown integrado ao painel.

Feature	Descrição
RichTextLabel	Renderiza arquivos `.md` do projeto com formatação (headers, código, listas).
Navegação	Lista de arquivos de documentação no projeto (ex: `README.md`, `EMENTA.md`).
Sem Alt-Tab	Consulte a documentação técnica sem sair da engine.

2. O DNA (Hiper-Resources)

Scripts que estendem `Resource`. São micro-serviços de comportamento autocontidos.

2.1. Recursos Estáticos (Blueprints)

Estes são os "moldes" de gameplay. Não são modificados em runtime.

`State` (state.gd)

A **unidade atômica de comportamento**. Define visual, animação e lógica.

Grupo	Propriedade	Descrição
Identity	`id` , `name` , `debug_color`	Identificação e cor de debug para o editor.
Visual	`texture` , `h_frames` , `v_frames` , `animation_row`	SpriteSheet e configuração de animação.
Pivot	`pivot_offset`	Offset do ponto de origem para ataques e efeitos.
Combat	`hitbox_scene` , `damage_multiplier` , `knockback`	Área de dano e multiplicador (aplicado sobre stats do CharacterSheet).
Movement	`speed_multiplier` , `friction` , `jump_force` , `lock_movement` , `ignore_gravity`	Parâmetros de física.
Timing	`duration` , `cooldown`	Duração do estado e tempo de recarga.
Requirements	`entry_requirements: Dictionary`	Filtros de contexto (Motion, Physics, Weapon).
Hooks	`_on_enter()` , `_on_exit()` , `_on_update(delta)`	Funções de ciclo de vida (virtual, para sobrescrita).

`Compose` (compose.gd)

O **Aglomerador de States**. Define o "Moveset" atual.

Propriedade	Descrição
`move_states: Array[State]`	Lista de States de movimento (Idle, Walk, Run, Dash).
`attack_states: Array[State]`	Lista de States de ataque (Slash, Thrust, Combo).
`interactive_states: Array[State]`	Lista de States de interação (Pickup, Talk).

Propriedade	Descrição
`move_rules: Dictionary`	HashMap gerado automaticamente (`@tool`) para lookup O(1).
`attack_rules: Dictionary`	HashMap gerado automaticamente para ataques.
`parent_compose: Compose`	Herança de outro Compose (para extensão de movesets).

`Item` (item.gd) / `Weapon` (extends Item)

Itens do jogo, desde consumíveis até armas.

Propriedade	Descrição
`id` , `name` , `icon`	Identificação visual.
`stackable` , `max_stack` , `quantity`	Lógica de empilhamento.
`consumable` , `durability` , `max_durability`	Uso e desgaste.
`category`	Tipo do item (Weapon, Armor, Consumable, Material).
`compose: Compose`	**Moveset próprio** do item. Se `null` , usa fallback.
`effects: Array[Effects]`	Efeitos aplicados ao usar/equipar.
`craft_recipe: Dictionary`	Receita de crafting `{ Item: quantidade }` .

`Skill` (skill.gd)

Habilidades que desbloqueiam mecânicas.

Propriedade	Descrição
`id` , `name` , `description` , `icon`	Identificação.
`skill_type`	`PASSIVE` (sempre ativo) ou `ACTIVE` (requer uso).
`req_level` , `req_attributes` , `prerequisites`	Requisitos para aprender.
`unlocks_states: Array[State]`	States desbloqueados ao aprender.

Propriedade	Descrição
`unlocks_items: Array[Item]`	Items (crafts) desbloqueados.
`modifiers: Array[Effects]`	Buffs passivos aplicados.
`effects_on_use: Array[Effects]`	Efeitos ao usar (para skills ativas).

`SkillTree` (skilltree.gd)

Grafo de progressão de Skills.

Propriedade	Descrição
`id` , `name`	Identificação da árvore.
`skills: Array[Skill]`	Lista de todas as Skills na árvore.
`get_available_skills()`	Retorna Skills cujos requisitos foram atendidos.
`get_unlocked_skills()`	Retorna Skills já aprendidas.

`Effects` (effects.gd)

Modificadores aplicáveis a entidades.

Propriedade	Descrição
`effect_type`	`INSTANT` , `TEMPORARY` , `PERMANENT` .
`duration`	Duração em segundos (para `TEMPORARY`).
`stat_modifiers: Dictionary`	Modificações de stats `{ "max_health": +50, "speed": 1.2 }` .
`status_effect`	Status aplicado (Poison, Burn, Stun).

`BehaviorStatesConfig` (config.gd)

Configuração global do plugin.

Propriedade	Descrição
`game_type`	`PLATFORMER` , `TOP_DOWN` , `ISOMETRIC` .

Propriedade	Descrição
`physics_mode`	`2D` ou `3D`.
`default_compose`	Compose padrão quando sem item equipado.
`input_buffer_time`, `coyote_time`	Parâmetros de game feel.

2.2. Recursos Vivos (In-Game Editable)

Estes Resources são modificados em runtime e salvos no SaveGame.

`Inventory` (inventory.gd)

Container de itens com lógica de stacking e instanciamento.

Propriedade	Descrição
`items: Array[Item]`	Lista de itens no inventário.
`capacity`	Número máximo de slots.
`starting_items`	Itens iniciais ao criar um novo jogo.
`add_item(item)`	Adiciona item (com `duplicate(true)` para instanciar).
`remove_item(item)`	Remove item do inventário.

> **Conceito Chave:** O Inventory nunca edita o Resource original do Item. Ele armazena **instâncias** com dados delta (durabilidade atual, quantidade).

`CharacterSheet` (character_sheet.gd)

A **Ficha do Personagem**. Central da verdade.

Propriedade	Descrição
`max_health`, `max_stamina`	Stats vitais.
`max_speed`, `jump_force`	Stats de movimento.
`attributes: Dictionary`	Atributos RPG `{ "strength": 10, "agility": 15 }`.

Propriedade	Descrição
`statistics: Dictionary`	Estatísticas de gameplay `{ "kills": 0, "play_time": 0 }`.
`skill_tree: SkillTree`	Referência à árvore de skills do personagem.
`equipment: Dictionary`	Slots de equipamento `{ "weapon": Item, "armor": Item }`.

3. Os Nodes (Componentes de Runtime)

Scripts que estendem `Node`. São adicionados à cena do personagem.

3.1. `Behavior` (behavior.gd)

O **Orquestrador de Intenção**. Gerencia "O que o Player QUER fazer".

Responsabilidade	Descrição
Input Handling	Processa inputs de alto nível e os traduz para Contexto.
Validação	Antes de mudar contexto (ex: `Jump` no ar), verifica se há State ou Skill que permita.
Dono dos Dados	Possui referências para `CharacterSheet`, `SkillTree` e `Backpack`.
Orquestração	Coordena o fluxo entre `Machine` e `Backpack`.

Signals:

- `context_changed(category, value)`: Emitido quando o contexto muda.

3.2. `Machine` (machine.gd)

A **Engine de Estados**. Gerencia "Como fazer".

Responsabilidade	Descrição
Query Engine	Consulta o `Compose` ativo pelo melhor `State` compatível com o Contexto.

Responsabilidade	Descrição
Scoring	Aplica o algoritmo de pontuação para desempatar candidatos.
Execução	Aplica física, animação, dano e efeitos conforme o `State` ativo.
Cálculo de Valores	Multiplica valores do `State` (damage_multiplier) pelos Stats do `CharacterSheet`.

Signals:

- `state_changed(old_state, new_state)` : Emitido quando o estado muda.

3.3. `Backpack` (backpack.gd)

A **Interface de Inventário (HUD)**. Gerencia visualmente o `Inventory`.

Responsabilidade	Descrição
Renderização	Exibe os slots do `Inventory` usando componentes `Slot`.
Seleção	Gerencia qual item está selecionado/equipado.
Crafting	Provê interface para receitas de craft.
Skill Tree	Exibe a árvore de skills e permite aprendizado.

Signals:

- `item_selected(item)` : Emitido quando um item é selecionado.
- `item_used(item)` : Emitido quando um item é usado (clique direito).

3.4. `Slot` (slot.gd)

Um **slot individual** do inventário.

Responsabilidade	Descrição
Renderização	Exibe ícone e quantidade do item.
Input	Detecta cliques para seleção e uso.
Drag & Drop	Suporta arrastar itens entre slots.

4. O Algoritmo (Reverse Query Hash Map)

Rejeitamos iteração O(N). O sistema usa **Indexação Reversa** para garantir seleção em tempo constante.

4.1. Index Time (Editor)

O `Compose.gd` roda como `@tool`. Ao salvar, reconstrói os índices:

```
# Compose.gd - Gerado automaticamente
@export var move_rules: Dictionary = {
    Motion.IDLE: [IdleState],
    Motion.RUN: [RunState, SprintState],
    Motion.DASH: [DashState]
}
```

4.2. Query Time (Runtime)

Quando a `Machine` precisa decidir:

1. **Chaveamento:** Constrói chave a partir do Contexto (`Motion.RUN`).
2. **Lookup O(1):** `candidates = compose.move_rules.get(Motion.RUN, [])` .
3. **Fuzzy Scoring:** Ranqueia candidatos por especificidade.
4. **Execução:** Aplica o State vencedor.

4.3. Fuzzy Scoring

Critério	Pontos
Match Exato de Atributo (ex: `Weapon: KATANA`)	+10
Match Genérico (`Weapon: ANY`)	+0
Prioridade de Chain (Combo)	+20
`priority_override` do State	+100 * valor

5. Estrutura Técnica do Plugin

5.1. Arquivos Principais

Arquivo	Descrição
`plugin.gd`	Classe principal do `EditorPlugin`. Registra nodes, resources, e o painel.
`plugin.cfg`	Metadados do plugin (nome, versão, autor).
`panel.tscn` / `panel.gd`	Cena e script do Bottom Panel. Gerencia as abas.

5.2. Estrutura de Pastas

```
addons/behavior_states/
├── assets/                  # Ícones e Temas
│   └── icons/               # SVGs para cada tipo de Resource
├── data/                    # Singleton config.tres
├── nodes/                   # Nodes de Runtime
│   ├── behavior.gd
│   ├── machine.gd
│   ├── backpack.gd / .tscn
│   └── slot.gd / .tscn
├── resources/               # Resources (o DNA)
│   ├── state.gd
│   ├── compose.gd
│   ├── item.gd
│   ├── skill.gd
│   ├── skilltree.gd
│   ├── effects.gd
│   ├── inventory.gd
│   ├── character_sheet.gd
│   ├── config.gd
│   └── blocks/              # Blocos visuais do Editor
└── scenes/                  # UI do Editor
    ├── panel.tscn           # Bottom Panel principal
    ├── components/          # Widgets reutilizáveis (AssetCard)
    └── tabs/                # Abas do painel
        ├── library.tscn     # Biblioteca de Assets
        ├── editor.tscn      # Blueprint Editor
        ├── factory.tscn     # Wizard de Criação
        └── docs_viewer.tscn # Grimório
```