🕹️ Guia Definitivo: Criando e Entendendo um Jogador no Godot Engine 🎮

Neste guia, vamos detalhar **como cada parte do código funciona**, explicando tipos de retorno, lógica de execução e armazenamento de dados. Nosso objetivo é que mesmo iniciantes possam entender o funcionamento completo do script.

📂 Estrutura Básica do Script

```
extends CharacterBody2D
class_name player
```

1. `extends CharacterBody2D`:
   - Este script herda a funcionalidade da classe `CharacterBody2D`.
   - **Por que usar?** Esta classe tem ferramentas úteis para criar um jogador, como `move_and_slide()`.
2. `class_name player`:
   - Nomeia a classe como `player`, tornando-a acessível em outros scripts ou cenas.
   - Por exemplo, você pode instanciar este jogador em outra cena usando `player.new()`.

📊 Declarando Variáveis

As variáveis armazenam **dados** e **estados** do jogador. Veja como cada uma é usada:

```
var _arma_atual: String = "_machado"
@export var _speed_pixel: float = 120
@export var _temporizador_de_acoes: Timer
@export var _area_de_ataque: Area2D
@onready var _animador_do_personagem: AnimationPlayer = $Animation
@export var _texto_armar_atual: Label
var direcao: Vector2 = Vector2.ZERO
var _sufixo_da_animacao: String = "_baixo"
var _pode_atacar: bool = true
```

Detalhamento das Variáveis

Variável	Tipo	Função
`_arma_atual`	`String`	Armazena o tipo de arma atualmente equipada (`"_machado"` , `"_espada"` , etc.).
`@export var _speed_pixel`	`float`	Define a velocidade de movimento do jogador (pode ser ajustada no editor do Godot).
`@export var _temporizador_de_acoes`	`Timer`	Controla o intervalo entre ações do jogador (como ataques).
`@export var _area_de_ataque`	`Area2D`	Define a área onde o ataque atinge (posição é alterada dinamicamente).
`@onready var _animador_do_personagem`	`AnimationPlayer`	Controla as animações do jogador.
`@export var _texto_armar_atual`	`Label`	Atualiza o nome da arma equipada na interface do usuário.
`direcao`	`Vector2`	Armazena a direção do movimento (eixo X e Y).
`_sufixo_da_animacao`	`String`	Indica a direção atual do jogador, usada para escolher a animação.
`_pode_atacar`	`bool`	Define se o jogador pode atacar (evita sobreposição de comandos).

🔁 Funções do Ciclo de Vida

Godot Engine utiliza ciclos de vida para gerenciar scripts e nós. Este código usa duas funções principais para controle:

1. `_ready()`

```
func _ready() -> void:
	pass
```

- **Função:** Executada automaticamente quando o nó é carregado na cena.
- **Tipo de Retorno:** `void` (não retorna valores).

- **Uso no Script:** Atualmente está vazia, mas pode ser usada para inicializar variáveis ou configurar estados.

---

2. `_process()`

```
func _process(_delta: float) -> void:
        _animar()
        _atacar()
        _sufixo_da_animacao = _sufixo_do_personagem()
        _definir_arma_atual()
        _movimento_personagem()
```

1. **Parâmetro `_delta`:**
   - Representa o tempo entre quadros (em segundos). Isso ajuda a sincronizar a execução do código independentemente do desempenho.
2. **Execução:**
   - A cada quadro do jogo:
     - Chama `_animar()` para atualizar animações.
     - Chama `_atacar()` para verificar comandos de ataque.
     - Atualiza `_sufixo_da_animacao` com a direção atual.
     - Chama `_definir_arma_atual()` para verificar a arma equipada.
     - Chama `_movimento_personagem()` para mover o jogador.
3. **Retorno:** `void` (não retorna valores).
4. **Ligação:** Essa função é automaticamente chamada pela engine.

🏃 Movimentação do Jogador

3. `_movimento_personagem()`

```
func _movimento_personagem() -> void:
        direcao = Input.get_vector("move_esquerda", "move_direita", "move_cima",
"move_baixo")
        velocity = direcao * _speed_pixel
        move_and_slide()
```

1. **Parâmetro:** Nenhum.

2. **Retorno:** `void` (não retorna valores).
3. **Função:**
    - `Input.get_vector()`:
        - Detecta quais teclas de movimento estão sendo pressionadas.
        - Retorna um `Vector2` (exemplo: `(1, 0)` para direita).
    - `velocity`:
        - Multiplica a direção pela velocidade para calcular a rapidez do jogador.
    - `move_and_slide()`:
        - Move o jogador e gerencia colisões automaticamente.

Execução no Ciclo: É chamada em `_process()`, o que garante que o jogador se mova em tempo real.

🎞 Gerenciando Animações

4. `_animar()`

```
func _animar() -> void:
	if _pode_atacar == false:
		return
	if velocity:
		_animador_do_personagem.play("move" + _sufixo_da_animacao)
		return
	_animador_do_personagem.play("parado" + _sufixo_da_animacao)
```

1. **Parâmetro:** Nenhum.
2. **Retorno:** `void`.
3. **Função:**
    - Se o jogador está atacando (`_pode_atacar == false`), nenhuma animação é executada.
    - Caso contrário, verifica a `velocity`:
        - Se houver movimento, a animação correspondente é reproduzida (`move_esquerda`, `move_direita`, etc.).
        - Se não houver movimento, a animação "parada" é reproduzida.

Execução no Ciclo: É chamada em `_process()`.

🗡️ Sistema de Ataques

5. `_atacar()`

```
func _atacar() -> void:
        var acao: String = _definir_arma_atual()
        if Input.is_action_pressed(acao) and _pode_atacar:
                _animador_do_personagem.play("atacar" + _arma_atual +
_sufixo_da_animacao)
                _temporizador_de_acoes.start(0.4)
                set_process(false)
                _pode_atacar = false
```

1. **Parâmetro:** Nenhum.
2. **Retorno:** `void`.
3. **Função:**
    - `_definir_arma_atual()`:
        - Retorna o nome da ação correspondente à arma equipada.
    - **Verificação:**
        - Se o comando de ataque (`Input.is_action_pressed(acao)`) é pressionado e `_pode_atacar == true`, o ataque é executado.
    - **Temporizador:**
        - Inicia `_temporizador_de_acoes` para criar um intervalo entre ataques.
    - **Interrupção do Ciclo:**
        - Desativa o `_process()` temporariamente (`set_process(false)`).

Execução no Ciclo: Chamada em `_process()`.

---

6. `_definir_arma_atual()`

```
func _definir_arma_atual() -> String:
        var tipo_acao: String
        tipo_acao = "ataque_normal"
        # Comandos de troca de arma omitidos
        _texto_armar_atual.text = _arma_atual
        return tipo_acao
```

1. **Parâmetro:** Nenhum.

2. **Retorno:** `String` (nome da ação associada à arma).
3. **Função:**
    - Determina qual arma está equipada e ajusta a variável `_arma_atual`.
    - Atualiza o texto exibido na interface (`_texto_armar_atual.text`).

Uso: É chamado por `_atacar()` e `_process()`.

📄 Link para o Código Completo

Baixe o código completo no formato PDF clicando no link abaixo:

➡️ **Código Completo do Player**

🔔 Conclusão

Agora você tem um entendimento completo do código, desde como ele é **armazenado**, **executado** e **chamado**, até as interações entre as funções. Este projeto é uma base sólida para desenvolver sistemas mais complexos, como combos de ataque, inventários e interação com o ambiente.